Postprint of article in The Symposium on Engineering Test Harness (TSETH '13),
Proceedings of the 13th International Conference on Quality Software (*QSIC '13*),
IEEE Computer Society, Los Alamitos, CA (2013)

Incremental Identification of Categories and Choices for Test Case Generation

A Study of the Software Practitioners' Preferences

Pak-Lok Poon
School of Accounting and Finance
The Hong Kong Polytechnic University
Hung Hom, Kowloon,
Hong Kong
Email: afplpoon@polyu.edu.hk

Tsong Yueh Chen
Faculty of Information
and Communication Technologies
Swinburne University of Technology
Hawthorn 3122, Australia
Email: tychen@swin.edu.au

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

***Abstract*—Test case generation is a vital procedure in the engineering of test harnesses. In particular, the choice relation framework and the category-partition method play an important role, by requiring software testers to identify categories (intuitively equivalent to input parameters or environment conditions) and choices (intuitively equivalent to ranges of values) from a specification and to systematically work on the identified choices to generate test cases. Other specification-based test case generation methods (such as the classification-tree method, cause-effect graphing, and combinatorial testing) also have similar requirements, although different terminology such as classifications and classes is used in place of categories and choices. For a large and complex specification that contains many specification components, categories and choices may be identified separately from various kinds of components. We call this practice an incremental identification approach. In this paper, we discuss our study involving 16 experienced software practitioners and three commercial specifications. Our objectives are to determine, from the opinions of the practitioners, (a) the popularity of an incremental identification approach, (b) the usefulness of identifying categories and choices from various kinds of specification components, and (c) possible ways to improve the effectiveness of the identification process.**

***Keywords*—incremental identification; choice relation framework; specification-based testing; test case generation; test harness**

This work is supported in part by a departmental general research fund of The Hong Kong Polytechnic University (project no. G-UA56), a linkage grant of the Australian Research Council (project no. LP100200208), and grants of the General Research Fund of the Research Grants Council of Hong Kong (project nos. 717811 and 716612).

I. Introduction

Generally speaking, software developers spend 40–50% of predelivery development costs on testing in order to achieve reasonable quality levels [24]. Studies by IBM and others have shown that the cost of correcting a fault after coding is at least 10 times as costly as that before it, and the cost of correcting a production fault is at least 100 times [19]. Similar observations have been reported in other literature. The cost-escalation factors range from 5x to 100x, depending on the types and sizes of the software systems [1], [12]. Thus, program testing should be as thorough as possible to help software developers detect any failures so that faults will not be propagated through to the final production software, where the cost of removal is far greater. Among various quality assurance techniques, testing remains a popular and important one for improving software quality [25].

Test case generation plays a vital role in the engineering of test harnesses. In particular, *specification-based* or *black-box* test case generation is a mainstream approach, in which test cases are generated according to information derived from the specification without the need to know the implemented code. Examples of such generation methods are the CHOiCe reLATion framEwork (CHOC'LATE) [6], [8], [20], the category-partition method [3], [18], the Classification-Tree Method (CTM) [7], [11], [13], [26], cause-effect graphing [16], and combinatorial testing [10], [15], [27]. CHOC'LATE and the category-partition method require software tester to identify categories (intuitively equivalent to input parameters or environment conditions) and choices (intuitively equivalent to ranges of values) from a specification, and to systematically maneuver the identified choices to generate test cases. Other specification-based test case generation methods also have similar recommendations, although they use different terminology such as classifications and classes. For ease of presentation, we will use the terms categories and choices throughout the paper.

Since test cases are generated from combinations of compatible choices, the identification of categories and their associated choices is of vital importance. If, for example, a valid choice x is missing, then all test cases containing x will not be generated. Consequently, any failure associated with x may not be detected. In other words, the generated *test suite* (that is, the set of test cases) will not be comprehensive.

In general, there are two types of specifications, namely formal and informal. *Formal* specifications are written in a mathematical notation such as Z [29] and Boolean predicates [9], [14], whereas *informal* specifications are mainly written in natural or graphic languages such as the United Modeling Language (UML) [17]. Relatively speaking, informal specifications are more popular than formal ones in the commercial software industry.

Despite the popularity of informal specifications, the identification of categories and choices from these specifications is often done in an ad hoc manner because of the absence of a systematic technique. The quality of test cases resulting from such an ad hoc approach is in doubt.

In view of this problem, we conducted empirical studies [5], [21] using informal commercial specifications. The objective of these studies was to investigate the common mistakes made by software testers involving informal specifications under an ad hoc identification approach. We then formally defined these common mistakes under various classes of missing/problematic categories and choices. As an interim solution, we also developed a checklist [5] to serve as a simple guideline for detecting missing/problematic categories and choices. In a recent study involving large, complex, informal specifications [6], we found that experienced software practitioners ease the task by identifying categories, choices, and relations from individual specification components before refining the global relations among choices. We will call it an *incremental identification approach* in this paper, and will discuss it in more detail in Section IV-A.

Following on our previous studies [5], [6], [21], we have conducted a further comparative study with 16 experienced software practitioners. The objectives of the study are:

(a) To determine the popularity of an incremental identification approach involving informal specification components.

(b) To determine, from the opinions of experienced software practitioners, (i) the usefulness of various kinds of specification components for category and choice identification and (ii) possible ways to improve the effectiveness of the identification exercise.

II. Basic Concepts and Procedure

Before proceeding with the details of the study, let us outline the main concepts and procedure of specification-based test case generation in the context of CHOC'LATE [6], [8], [20].

A *category* is defined as a major property or characteristic of a parameter or an environment condition of the system under test (SUT) that affects its execution behavior. For ease of presentation, parameters and environment conditions are collectively known as *factors* in this paper. In addition, any factor is said to be *influencing* if it affects the execution behavior of the SUT. *Choices* are disjoint subsets of values of a category, with the assumption that all values in the same choice are similar in their effect on the system behavior.

Following the notation in [6] we will use variable names (such as Q) to denote categories, and use variable names with subscripts (such as Q_x) to denote their associated choices. When the categories are obvious from the context, we may simply write Q_x as x.

Given a category Q, all its associated choices Q_1, Q_2, …, Q_n should cover the subset of the *input domain* (the set of all possible inputs to a program) relevant to Q. Consider, for example, an undergraduate award classification system whose main function is to determine whether a student is eligible for graduation. A possible category for the system is "GPA Score (G)", and a possible choice associated with this category is "GPA Score $(G)_{\,3.5 \le G \le 4.0}$".

A set of choices is called a *test frame*, denoted by B. A test frame B is said to be *complete* if, whenever a single value is selected from each choice in B, a test case is formed.

In general, test case generation involves the following five steps:

(1) Decompose the specification into functional units that can be tested separately.

(2) Identify categories and their associated choices from each functional unit.

(3) Determine the relations among the identified choices as stated or implied in each functional unit.

(4) Using the predefined algorithms, combine compatible choices together to form complete test frames according to the choice relations determined in (3).

(5) Generate a test case from each complete test frame by selecting and combining a value from each choice in that test frame.

III. Experimental Setting

A. Specifications

Our study uses three commercial specifications that are written primarily in an informal manner.

The first specification $\mathbb{S}_{\mathsf{TRADE}}$ is related to the credit sales of goods by a wholesaler to retail customers. The main function of the system is to decide whether credit sales should be approved for individual retail customers. Such a decision considers several issues, such as the credit status and credit limit of the customer and the billing amount of the transactions.

The second specification $\mathbb{S}_{\mathsf{PURCHASE}}$ is related to the purchase of goods using credit cards issued by an international bank. Each credit card is associated with several attributes such as status (diamond, gold, or classic), type (corporate or personal), and credit limit (different card statuses will have different credit limits). The main functions of the system are to decide whether a purchase using a credit card should be approved, and to calculate the number of reward points to be granted for an approved purchase. The number of reward points further determines the type of benefit (such as free airline

tickets and shopping vouchers) that the customer is entitled to.

The third specification $\mathbb{S}_{\mathsf{MOS}}$ is related to a meal ordering system (MOS), which is being used by an international company (denoted by AIR-FOOD) providing catering service for many different airlines. The main function of MOS is to help AIR-FOOD determine the types (such as normal, child, and vegetarian) and numbers of meals to be prepared and loaded onto each flight served by AIR-FOOD.

Since MOS contains numerous modules and is fairly complex in logic, we decompose $\mathbb{S}_{\mathsf{MOS}}$ into several functional units that can be tested independently. For example, there is a functional unit $\mathbb{U}_{\mathsf{MEAL}}$ directly related to the generation of daily meal schedules and other units related to the maintenance of airline codes and city codes. We do not apply such decomposition to $\mathbb{S}_{\mathsf{TRADE}}$ and $\mathbb{S}_{\mathsf{PURCHASE}}$ because the corresponding systems are less complex and can be tested in their entirety. Thus, we treat $\mathbb{S}_{\mathsf{TRADE}}$ and $\mathbb{S}_{\mathsf{PURCHASE}}$ themselves as functional units, denoted by $\mathbb{U}_{\mathsf{TRADE}}$ and $\mathbb{U}_{\mathsf{PURCHASE}}$, respectively.

B. Subjects

We recruited 16 experienced software practitioners as subjects of our study. The minimum qualifications were an undergraduate degree related to IT and five years of relevant commercial experience. After the recruitment exercise, we found that the subjects have 8 to 20 years of commercial experience in software development and testing, with a mean of 11.9 years. Each of them has an undergraduate and/or postgraduate degree in information technology, information systems, business computing, computer science, computing studies, or computer engineering. Some of them also have other non-IT academic qualifications such as MBA degrees.

Before commencing our study, we prepared the subjects by giving them a one-hour introduction of CHOC'LATE (and also CTM, which is another specification-based test case generation method). We also issued them with the relevant supporting documents. The introduction was followed by a one-hour discussion in which some examples of CHOC'LATE and CTM were used to reinforce the subjects' understanding of these techniques. A hands-on exercise was given to the subjects to assess whether they were properly trained in applying the techniques.

C. Category and Choice Identification

As introduced in Section II, categories are the major properties or characteristics of influencing factors of the SUT. For every category Q proposed by the subjects, it may either be identified according to the definition, or incorrectly identified with something else in mind. In view of this situation, we will refer to any Q identified by the subjects as a *potential category*. Similarly, any Q_x identified by the subjects is called a *potential choice*.

Any potential category Q is said to be *relevant* if it is defined with respect to an influencing factor. Otherwise, it is said to be *irrelevant*. Only relevant categories are useful for test case generation. In the rest of the paper, relevant categories are simply referred to as "categories" unless otherwise stated.

TABLE I. NUMBERS AND PERCENTAGES OF SUBJECTS USING AN INCREMENTAL IDENTIFICATION APPROACH.

Functional Unit	Number (Percentage) of Subjects
$\mathbb{U}_{\mathsf{TRADE}}$	7 (44%)
$\mathbb{U}_{\mathsf{PURCHASE}}$	8 (50%)
$\mathbb{U}_{\mathsf{MEAL}}$	16 (100%)

After the subjects had learned CHOC'LATE and CTM, we asked each of them to identify potential categories and their associated potential choices from each of $\mathbb{U}_{\mathsf{TRADE}}$, $\mathbb{U}_{\mathsf{PURCHASE}}$, and $\mathbb{U}_{\mathsf{MEAL}}$ in an ad hoc manner. Furthermore, for each identified potential category and potential choice, the reason of its identification had to be stated.

IV. RESPONSES FROM SUBJECTS

After the identification exercises, we organized a meeting with the 16 subjects, with a view to finding out their answers to the following three research questions:

(RQ1) How many subjects use an incremental identification approach?

(RQ2) Which kinds of components in a specification are more useful for identifying categories and choices?

(RQ3) In what ways can the effectiveness of the identification process be improved?

These three issues will be discussed in Sections IV-A, IV-B, and IV-C below.

A. Popularity of an Incremental Identification Approach (RQ1)

Unlike specifications that are written in formal languages such as Z [29] and Boolean predicates [9], [14], informal specifications are often expressed in many different styles and formats, and contain a large variety of components [6]. Examples of these components are narrative descriptions, use cases, activity diagrams, swimlane diagrams, state machines, data flow diagrams (DFD), and data dictionaries. For a complex informal specification with many different components, software testers may find it difficult to identify categories and choices from the *entire* specification in one single round. Rather, testers may be inclined to decompose the identification process into several steps, each step focusing on only one specification component. We call this practice an *incremental identification approach*.

Table I shows the number and percentage of subjects who used an incremental identification approach for each functional unit. Note that, among the three functional units, $\mathbb{U}_{\mathsf{MEAL}}$ is the most complex in terms of the system logic and the number of specification components, and $\mathbb{U}_{\mathsf{TRADE}}$ is the least complex. Thus, Table I shows that the subjects are more inclined to use an incremental identification approach for a more complex functional unit.

B. Usefulness of Various Kinds of Specification Components (RQ2)

Inspired by the observation in Table I, we asked the subjects which kinds of specification components they would

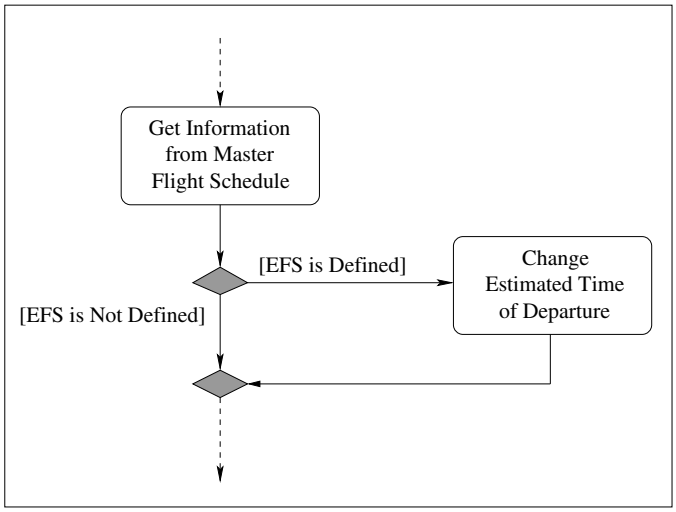

Fig. 1. Part of an activity diagram for the generation of daily meal schedules.

consider more useful for the identification of categories and choices.

1) Activity Diagrams and Swimlane Diagrams: In response to this question, 15 subjects (94%) considered activity diagrams and swimlane diagrams to be very useful for category and choice identification.

Basically, an activity diagram or a swimlane diagram represents the actions and decisions that occur as some function is performed. The diagrams use rounded rectangles to denote activities, arrows to represent control flows of activities, diamond icons (corresponding to *decision points*) to depict branching decisions, and solid thick bars to indicate the occurrence of parallel activities. Arrows are used to indicate *alternative threads* that emerge from every decision point depending on the *guard conditions* enclosed in square brackets. The diamond icon can also be used to show where the alternative threads merge again.

The swimlane diagram is a useful variation of the activity diagram. Both types of diagrams contain decision points and their associated guard conditions. A swimlane diagram, however, can also indicate which actor (if there are multiple actors involved in a specific function) or analysis class is responsible for the action described by an activity rectangle. Responsibilities are represented as parallel segments that divide the diagram into swimlanes, like the lanes in a swimming pool [23].

Intuitively, the decision points and the guard conditions in an activity diagram or a swimlane diagram indicate where and how a software system *behaves differently*. (Readers may recall that a category corresponds to an influencing factor of the SUT.) Thus, about 94% of the subjects consider that this characteristic makes activity diagrams and swimlane diagrams very useful for identifying categories and choices.

The following Example 1 further illustrates the usefulness of activity diagrams. We note that the usefulness of activity diagrams in the example applies also to swimlane diagrams.

Example 1 (Generation of Daily Meal Schedules). In $\mathbb{U}_{\mathsf{MEAL}}$ of MOS, there are many master flight schedules, each of which corresponds to a flight served by AIR-FOOD. The information captured in master flight schedules will be used by MOS to generate the corresponding daily meal schedules. These daily meal schedules contain important information to help AIR-FOOD determine the types and numbers of meals to be prepared and loaded onto the flights. During the generation process for daily meal schedules, some information in master flight schedules (MFS) can be overridden by the corresponding exceptional flight schedules (EFS), if the latter are defined. Basically, an EFS allows users to change the estimated time of departure of a flight on a particular date after the MFS of this flight has been defined.

Consider the partial activity diagram in Fig. 1, which shows part of the generation process of daily meal schedules. The upper diamond icon represents a decision point associated with two guard conditions, "EFS is Defined" and "EFS is Not Defined". Because these two guard conditions correspond to different flows of control and, in turn, different execution behavior (or influencing factors) of MOS, the category "Defined EFS" should be identified, with two associated choices "Defined $\mathrm{EFS}_{\mathrm{yes}}$" and "Defined $\mathrm{EFS}_{\mathrm{no}}$".

2) Sample Input Screens and Data Dictionaries: One half (50%) of the subjects also reported that sample input screens and data dictionaries are useful for the identification task, although these specification components were not considered to be as useful as activity diagrams and swimlane diagrams.

Consider sample input screens first. Eight subjects reported that, by showing the input parameters explicitly, sample input screens represent a good repository in which input parameters corresponding to influencing factors can be identified as categories. Suppose there is a parameter E on a sample input screen corresponding to an influencing factor of the SUT, and Q^E is the category corresponding to E. If the input screen also shows different values (or ranges of values) of E, then these values or ranges may correspond to the choices associated with E.

Consider, for instance, the sample input screen in Fig. 2 for the maintenance of master flight schedules (MFS) in MOS, in which the inputs are entered inside square brackets [ ]. It indicates that the input parameter "Frequency of Departure" has two possible values: "daily" and "non-daily". The screen also shows that for a non-daily flight, users need to enter its "Weekly Flight Pattern", which shows the day(s) of the flight within a week. This is because MOS will check the weekly flight pattern for a non-daily flight when generating the corresponding daily meal schedule from an MFS. Because of this reason, "Frequency of Departure" is an influencing factor and, hence, should be identified as a category with two associated choices "daily" and "non-daily".

For similar reasons, the eight subjects also considered data dictionaries to be useful for category and choice identification, but not as useful as sample input screens. This was because, unlike sample input screens, the elements in a data dictionary do not necessarily correspond to input parameters. For example, some elements in a data dictionary may correspond to

Maintenance of Master Flight Schedules

Airline **[JJ]** Flight Number **[701]** Flight Sector **[HKG/TPE/HKG]**

Aircraft Type: (1) Boeing−747 (2) Boeing−737 (3) Airbus−340 (4) Airbus−330 **[1]**

Estimated Time of Departure **[15:45]**

Effective Period **[15/Nov/2010]** to **[30/Apr/2011]**

Frequency of Departure: Daily **[ ]** Non−daily **[X]**

Weekly Flight Pattern: Mon **[X]** Tue **[X]** Wed **[ ]** Thu **[ ]** Fri **[ ]** Sat **[X]** Sun **[ ]**
(for non−daily flights only)

Save **[X]** Delete **[ ]** Exit **[ ]**

Fig. 2. Sample input screen for the function "Maintenance of Master Flight Schedules".

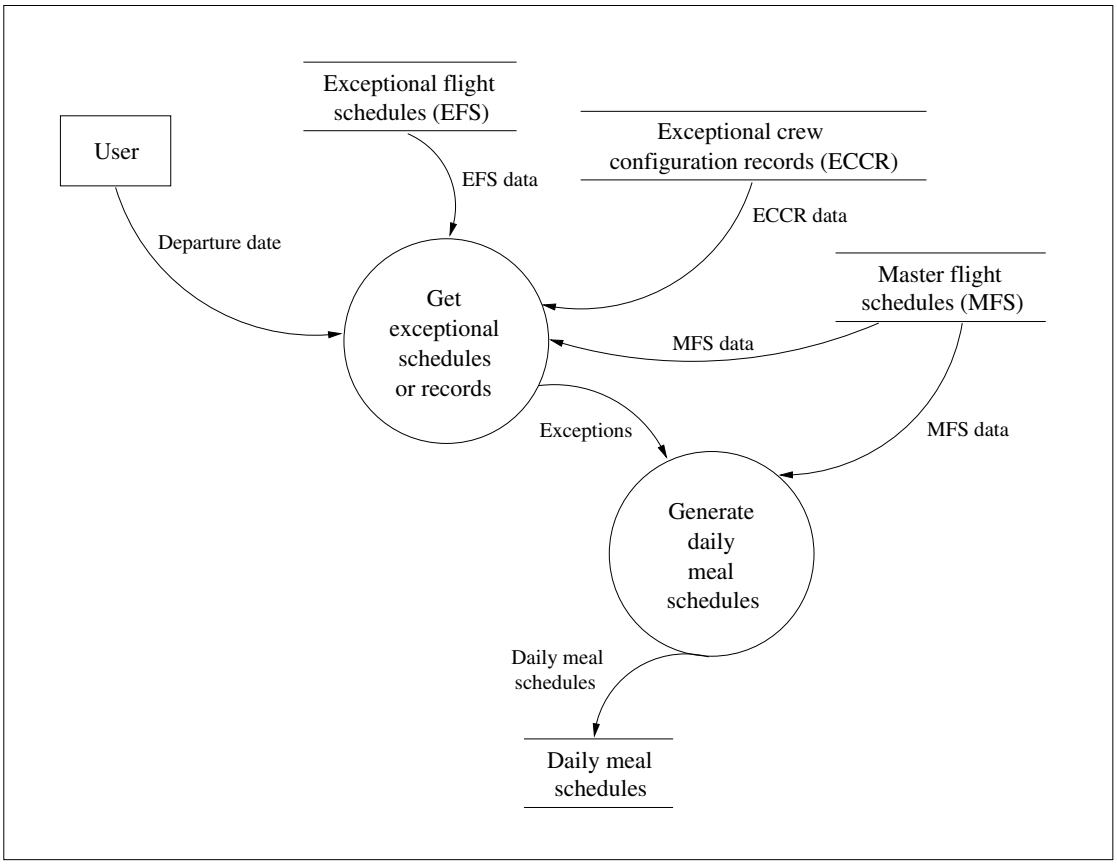

Fig. 3. A level-1 data flow diagram for the generation of daily meal schedules.

intermediate attributes or system outputs. Obviously, categories should not be identified from such elements. Thus, the subjects need to spend extra effort to identify a proper subset of the elements in a data dictionary that correspond only to input parameters.

The eight subjects also gave the two reasons why they considered sample input screens and data dictionaries less useful than activity diagrams and swimlane diagrams. First, the usefulness of activity diagrams and swimlane diagrams is largely due to the decision points and the guard conditions, which do not exist in sample input screens and data dictionaries. Second, as illustrated in Example 1, the guard conditions largely ease the identification of choices for each category (which corresponds to a decision point in an activity diagram or swimlane diagram). On the other hand, in the sample input screens and data dictionaries, even though software testers can determine which input parameters should be used for category identification, relatively little information is provided for identifying the choices for each category.

3) Data Flow Diagrams: Finally, six subjects (38%) pointed out that lower-level data flow diagrams are useful for identifying categories and choices. In short, the DFD takes an input-process-output view of a system [23]. That is, data flow into the software and are transformed by processing elements

into useful information that flows out of the software. The flows of data are indicated by labeled arrows, transformations are represented by bubbles, and data stores are denoted by double lines. Data flow diagrams are arranged in levels. The highest level data flow model (also called a *level-0* DFD or *context diagram*) represents the system as a whole. Lower-level data flow diagrams refine higher-level data flow diagrams, providing more details. Some software practitioners argue that although data flow diagrams are not a formal part of UML, they can be used to complement UML diagrams and provide additional insight into system requirements and flows [23].

The usefulness of lower-level data flow diagrams for identifying categories and choices is mainly contributed by their data stores. Fig. 3, for instance, shows a level-1 DFD for generating daily meal schedules in $\mathbb{U}_{\mathsf{MEAL}}$ of MOS. There are four data stores in this DFD. Let us consider the data store "Master Flight Schedules (MFS)". Intuitively, an MFS indicates an environment condition of $\mathbb{U}_{\mathsf{MEAL}}$ with three possible statuses (namely, an undefined MFS, a defined but empty MFS, and a defined and nonempty MFS), each of which will result in a different execution behavior of the system. Thus, the data store "Master Flight Schedules (MFS)" corresponds to an influencing factor and, hence, the category "Status of MFS" should be identified with three associated choices "Status of $\text{MFS}_{\text{not defined}}$", "Status of $\text{MFS}_{\text{defined but empty}}$", and "Status of $\text{MFS}_{\text{defined and nonempty}}$". The first two choices are identified to test how the system behaves under abnormal situations. Categories can also be identified for influencing environment conditions corresponding to the data stores "Exceptional Flight Schedules (EFS)" and "Exceptional Crew Configuration Records (ECCR)" in a similar manner. We should point out that, for any data store D that serves as an input to a process in a DFD and for the corresponding category Q^D, the three standard choices "$Q^D_{\text{not defined}}$" "$Q^D_{\text{defined but empty}}$", and "$Q^D_{\text{defined and nonempty}}$" are always applicable.

Readers should note that, although data flow diagrams (by virtue of the data stores) are useful for identifying influencing *environment conditions* from which their corresponding categories and choices can be defined, they provide little information for identifying influencing *parameters* (and their corresponding categories and choices). Although some data flows in a DFD may potentially correspond to parameters, we do not know whether they are "influencing" by inspecting the DFD alone. In addition, instead of representing individual data items, such data flows may correspond to groups of data items, some of which may correspond to influencing parameters while the others may not.

C. Improvement to the Ad Hoc Identification Process (RQ3)

In the discussion meeting with the 16 subjects, we found that 11 (69%) of them also constructed choice relation tables (for CHOC'LATE) or classification trees (for CTM), even though they were not asked to do so. Basically, the construction of such tables or trees represents the next step in CHOC'LATE and CTM after the identification of categories and choices. The main purpose of the tables and trees is to capture the relations among choices (in CHOC'LATE) (see step (3) of CHOC'LATE in Section II) or categories (in CTM), through which complete test frames can be subsequently generated (see step (4) of CHOC'LATE in Section II).

These 11 subjects argued that the identification of categories and choices is closely related to the identification of their relations and, hence, these two steps should not be totally separate. More specifically, the subjects need to consider the relations when identifying categories and choices. Otherwise, the identification task may not be well executed. This point is illustrated in the following example:

Example 2 (Exceptional Schedules and Records). In the functional unit $\mathbb{U}_{\mathsf{MEAL}}$ of MOS, the information captured in master flight schedules (MFS) will be used to generate the corresponding daily meal schedules. During the generation process, some information in master flight schedules can be overridden by exceptional flight schedules (EFS) and exceptional crew configuration records (ECCR). Some details of exceptional flight schedules have been provided in Example 1 and will not be repeated here.

Let us focus on exceptional crew configuration records. They allow users to change the number of crewmembers in a flight on a particular date after the MFS of this flight has been defined. Such records are necessary since AIRFOOD needs to prepare meals for the crews as well as for passengers.

In view of the possible definition of exceptional flight schedules and exceptional crew configuration records, any MFS will fall into one of the following situations:

(a) it is associated with an EFS but not an ECCR;
(b) it is associated with an ECCR but not an EFS;
(c) it is associated with an EFS and an ECCR; and
(d) it is not associated with any EFS or ECCR.

There are two approaches to identifying categories and choices:

- **Approach 1:** Intuitively, in order to cater for all the above situations, two categories should be identified: "Defined EFS" and "Defined ECCR". The first category should have two associated choices: "Defined EFS_{yes}" and "Defined EFS_{no}". Similarly, the second category should have two associated choices: "Defined ECCR_{yes}" and "Defined ECCR_{no}". With these categories and choices, each of the above four situations can be tested *separately* by selecting one choice in "Defined EFS" and one in "Defined ECCR". For example, the selection of both "Defined EFS_{no}" and "Defined ECCR_{yes}" will cover situation (b).
- **Approach 2:** Some software testers, however, may argue that we do not need two categories and four choices. The rationale is that such an approach will increase the number of complete test frames (and in turn the number of test cases) generated and, hence, will require more testing effort. Instead, these testers propose that only one category "Exceptional Schedules/Records Defined", with "Exceptional Schedules/Records $\text{Defined}_{\text{yes}}$" and "Exceptional Schedules/Records $\text{Defined}_{\text{no}}$" as its two associated choices, should be identified. Note that the choice "Exceptional Schedules/Records $\text{Defined}_{\text{yes}}$"

caters for situations (a), (b), and (c) collectively; whereas "Exceptional Schedules/Records Defined$_{\text{no}}$" caters for situation (d) only. On one hand, the new proposal will result in fewer complete test frames and test cases, and some test effort can be saved. On the other hand, the resulting test suite will be less "refined", in the sense that test cases cannot be generated to cater for situations (a), (b), and (c) on an individual basis.

In order to judge which identification approach should be adopted, the subjects needed to consider a question: Which approach will generate a test suite with a better coverage and, hence, a higher chance of revealing failures? The question could only be answered if the subjects knew:

- What are the other categories and their associated choices to be identified?
- What are the relations among the identified categories/choices? These relations determine how choices are combined to form part of any complete test frame.
- Suppose B_1^c, B_2^c, and B_3^c are any complete test frames containing the following three pairs of choices:

  ("Defined EFS$_{\text{yes}}$" and "Defined ECCR$_{\text{no}}$"),
  ("Defined EFS$_{\text{no}}$" and "Defined ECCR$_{\text{yes}}$"), and
  ("Defined EFS$_{\text{yes}}$" and "Defined ECCR$_{\text{yes}}$"),

  respectively. Suppose, further, that B_1^c, B_2^c, and B_3^c only differ in the above choice pairs; all the other choices contained in these three complete test frames are identical (that is, $B_1^c \setminus \{\text{Defined EFS}_{\text{yes}}, \text{Defined ECCR}_{\text{no}}\} = B_2^c \setminus \{\text{Defined EFS}_{\text{no}}, \text{Defined ECCR}_{\text{yes}}\} = B_3^c \setminus \{\text{Defined EFS}_{\text{yes}}, \text{Defined ECCR}_{\text{yes}}\}$). In this case, are B_1^c, B_2^c, and B_3^c associated with different execution behavior of MOS? If yes, then approach 1 should be used to generate a test suite with a better coverage. Otherwise, approach 2 should be used to generate a smaller test suite (that is, with fewer test cases) without jeopardizing its effectiveness of revealing failures.

In summary, the subjects suggested that identifying categories and choices and identifying relations among categories/choices should not be viewed as two totally distinct processes.

V. Threats to Validity

Our current empirical study had two limitations owing to various settings. First, our study involved only 16 experienced subjects. The study would certainly be better if more experienced subjects participated. It was not easy, however, to find a large group of experienced software testers willing to participate in the study (with or without remuneration). Second, only three specifications were used in the study. Nevertheless, we believe that even with 16 experienced subjects and three specifications, our findings still provide an inspiring insight into the answers to the three research questions RQ1, RQ2, and RQ3 stated at the beginning of Section IV.

One may argue that our study largely involves observations of human performance and, hence, the contribution of the findings to the software community is in question. In this regard, Tichy [28] argues that "*observation* and experimentation can lead to new, useful, and unexpected insights and open whole new areas of investigation." In addition, Briand [2] argues that the training and skills of testers have a strong impact on the cost-effectiveness of testing techniques because these techniques are often not entirely automated and require at the very least human inputs. This is particularly the case for CHOC'LATE and CTM because they are partly based on human intuition and understanding of the system behavior (for example, the ad hoc identification of categories and choices). Thus, we argue that observations of human performance do play an important role in software engineering research. Our argument is supported by numerous publications, primarily involving human subjects, in leading software engineering journals, including the work by Carver et al. [4] and Porter and Johnson [22].

VI. Summary and Conclusion

Our study has confirmed that an incremental identification of categories and choices for the engineering of test harnesses is very popular when the specification is large and complex (RQ1). In terms of the usefulness of various kinds of specification components in identifying categories and choices (RQ2), the experienced tester subjects favored activity diagrams, swimlane diagrams, sample input screens, and data dictionaries in various extents. Furthermore, the subjects suggested that, in order to improve the effectiveness of an ad hoc approach, the identification of categories, choices, and the relations among categories/choices should not be considered separately (RQ3).

We end this paper with two final reminders. First, our study results are not restricted to CHOC'LATE [6], [8], [20] and the category-partition method [3], [18] only. As pointed out in Section I, the identification of categories and choices (or their equivalents) is also needed in other specification-based test case generation methods such as CTM [7], [11], [13], [26], cause-effect graphing [16], and combinatorial testing [10], [15], [27]. Second, in line with the thoughts of the software community [2], [4], [22], [28], the observation of human behavior is an essential element in software engineering, including the development of effective test harnesses. In this regard, our results should play a part in the contributions to software engineering research.

Acknowledgment

We are grateful to the 16 anonymous software practitioners for their invaluable time and effort in participating in the study.

References

[1] B.W. Boehm and V.R. Basili, "Software defect reduction top 10 list," *IEEE Computer*, vol. 34, no. 1, pp. 135–137, 2001.

[2] L.C. Briand, "A critical analysis of empirical research in software testing," *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement* (*ESEM 07*), IEEE Computer Society, 2007, pp. 1–8.

[3] L.C. Briand, Y. Labiche, Z. Bawar, and N.T. Spido, "Using machine learning to refine category-partition test specifications and test suites," *Information and Software Technology*, vol. 51, no. 11, pp. 1551–1564, 2009.

[4] J.C. Carver, N. Nagappan, and A. Page, "The impact of educational background on the effectiveness of requirements inspections: an empirical study," *IEEE Transactions on Software Engineering*, vol. 34, no. 6, pp. 800–812, 2008.

[5] 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*, vol. 46, no. 13, pp. 887–898, 2004.

[6] T.Y. Chen, P.-L. Poon, S.-F. Tang, and T.H. Tse, "DESSERT: a divide-and-conquer methodology for identifying categories, choices, and choice relations for test case generation," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 794–809, 2012.

[7] T.Y. Chen, P.-L. Poon, and T.H. Tse, "An integrated classification-tree methodology for test case generation," *International Journal of Software Engineering and Knowledge Engineering*, vol. 10, no. 6, pp. 647–679, 2000.

[8] 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*, vol. 29, no. 7, pp. 577–593, 2003.

[9] Z. Chen, T.Y. Chen, and B. Xu, "A revisit of fault class hierarchies in general Boolean specifications," *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 3, article no. 13, 2011.

[10] D.M. Cohen, S.R. Dalal, M.L. Fredman, and G.C. Patton, "The AETG system: an approach to testing based on combinatorial design," *IEEE Transactions on Software Engineering*, vol. 23, no. 7, pp. 437–444, 1997.

[11] M. Grochtmann and K. Grimm, "Classification trees for partition testing," *Software Testing, Verification and Reliability*, vol. 3, no. 2, pp. 63–82, 1993.

[12] M. Grottke and K.S. Trivedi, "Fighting bugs: remove, retry, replicate, and rejuvenate," *IEEE Computer*, vol. 40, no. 2, pp. 107–109, 2007.

[13] R.M. Hierons, M. Harman, and H. Singh, "Automatically generating information from a Z specification to support the classification tree method," *Proceedings of the 3rd International Conference of B and Z Users*, Lecture Notes in Computer Science, vol. 2651, Springer, 2003, pp. 388–407.

[14] M.F. Lau and Y.T. Yu, "An extended fault class hierarchy for specification-based testing," *ACM Transactions on Software Engineering and Methodology*, vol. 14, no. 3, pp. 247–276, 2005.

[15] Y. Lei, R.H. Carver, R. Kacker, and D. Kung, "A combinatorial testing strategy for concurrent programs," *Software Testing, Verification and Reliability*, vol. 17, no. 4, pp. 207–225, 2007.

[16] G.J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, John Wiley, 2011.

[17] *OMG Unified Modeling Language* (*OMG UML): Superstructure*, Version 2.4.1, Object Management Group, 2011, http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/.

[18] T.J. Ostrand and M.J. Balcer, "The category-partition method for specifying and generating functional tests," *Communications of the ACM*, vol. 31, no. 6, pp. 676–686, 1988.

[19] W.E. Perry, *Effective Methods for Software Testing*, John Wiley, 2006.

[20] P.-L. Poon, S.-F. Tang, T.H. Tse, and T.Y. Chen, "CHOC'LATE: a framework for specification-based testing," *Communications of the ACM*, vol. 53, no. 4, pp. 113–118, 2010.

[21] P.-L. Poon, T.H. Tse, S.-F. Tang, and F.-C. Kuo, "Contributions of tester experience and a checklist guideline to the identification of categories and choices for software testing," *Software Quality Journal*, vol. 19, no. 1, pp. 141–163, 2011.

[22] A.A. Porter and P.M. Johnson, "Assessing software review meetings: results of a comparative analysis of two experimental studies," *IEEE Transactions on Software Engineering*, vol. 23, no. 3, pp. 129–145, 1997.

[23] R.S. Pressman, *Software Engineering: a Practitioner's Approach*, McGraw-Hill, 2010.

[24] J.W. Sanders and E. Curran, *Software Quality: a Framework for Success in Software Development and Support*, Addison-Wesley, 1994.

[25] T. Shepard, M. Lamb, and D. Kelly, "More testing should be taught," *Communications of the ACM*, vol. 44, no. 6, pp. 103–108, 2001.

[26] H. Singh, M. Conrad, and S. Sadeghipour, "Test case design based on Z and the classification-tree method," *Proceedings of the 1st IEEE International Conference on Formal Engineering Methods* (*ICFEM 97*), IEEE Computer Society, 1997, pp. 81–90.

[27] K.-C. Tai and Y. Lei, "A test generation strategy for pairwise testing," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 109–111, 2002.

[28] W.F. Tichy, "Should computer scientists experiment more?" *IEEE Computer*, vol. 31, no. 5, pp. 32–40, 1998.

[29] J.B. Wordsworth, *Software Development with Z: a Practical Approach to Formal Methods in Software Engineering*, Addison-Wesley, 1992.