To appear in *Communications of the ACM*

CHOC'LATE: a CHOiCe reLATion framEwork for specification-based testing *

by **Pak-Lok Poon, Sau-Fun Tang, T. H. Tse, and T. Y. Chen**

In spite of its importance in software reliability, testing is labor intensive and expensive. It has been found that software testing without a good strategy may not be more effective than testing the system with random data. Obviously, the effectiveness of testing relies heavily on how well the *test suite* — the set of test cases actually used — is generated. This is because the comprehensiveness of the test suite will affect the scope of testing and, hence, the chance of revealing software faults.

There are two main approaches to generating test suites: *specification-based* and *code-based*. The former generates a test suite from information derived from the specification, without requiring the knowledge of the internal structure of the program [9, 10, 11]. The latter approach, on the other hand, generates a test suite based on the source code of the program [4, 8]. Neither of these approaches is sufficient; they are complementary to one another [1].

In software development, the requirements have to be established before implementation, and the specification should exist prior to coding. In this respect, the specification-based approach to test suite generation is particularly useful because test cases can be generated before coding has been completed. This facilitates software development phases to be performed in parallel, thus allowing time for preparing more thorough test plans and yet shortening the length of the whole process.

Problems in Specification-Based Testing

Let us focus on the specification-based approach. Specifications — the sources for generating test suites — often exist in a spectrum of forms as depicted in Figure 1. At the left extreme is the completely *informal* specification primarily written in natural language. On the other hand, the right extreme of the spectrum corresponds to the completely *formal* specification written in a mathematical notation. A specification, in general, may be in a format lying somewhere between these two extremes.

Formal specifications, because of their mathematical basis, are more precise than informal specifications. They can be analyzed rigorously for inconsistency. Furthermore, the rigorous nature of formal specifications eases the automatic generation of test suites. Hence, there exist systematic and automated test suite generation methods for various types of formal specification such as Z [9] and Boolean [7] specifications. More recent examples of generating test suites from formal specifications include the modified condition/decision coverage (MC/DC) strategy [7, 12] and the MUMCUT strategy [7]. The MC/DC strategy can be classified as either specification-based or code-based testing, depending on whether the predicate information used to generate a test suite is derived from the specification or the source code. In particular, compliance of the MC/DC strategy has been mandated in the commercial aviation industry for the approval of airborne software [7]. Readers should note, however, that both strategies are only applicable to the detection of failures in logical decisions in Boolean specifications.

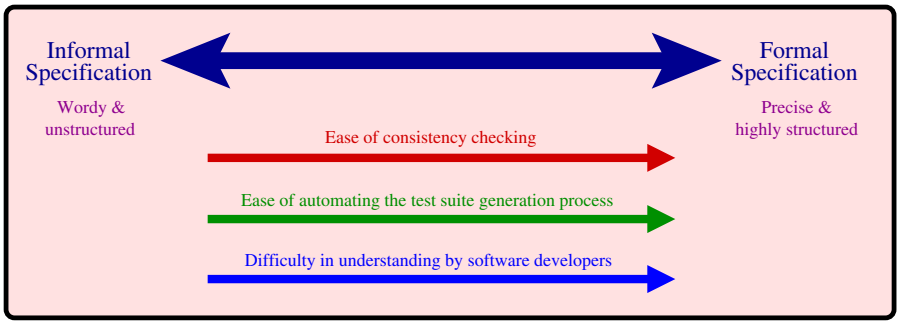

Figure 1. Spectrum of different forms of specifications.

Despite the advantages of precision and rigor, formal specifications and their associated test suite generation methods are not as popular as they should be, mainly because most software developers are not familiar with the mathematical concepts involved and find the techniques difficult to understand and use. How about informal specifications? Are there any problems for testing based on these specifications? If the answer is yes, what are the problems?

Many informal specifications, especially those written in natural language, are often wordy and unstructured, making it difficult to generate test suites directly from them. Several test suite generation methods based on informal specifications have been developed by researchers with a view to alleviating this problem. Examples are the classification-tree method (CTM) [5] and the category-partition method (CPM) [2]. They require software testers to re-express the original informal specification in a more concise and structured intermediate form, such as a classification tree in CTM, from which a test suite can be generated more easily. The idea is to re-enact the original specification in a format that lies somewhere between the two extremes of the spectrum in Figure 1, so that:

(a) On one hand, the new representation supports some degree of systematic test suite generation, even though it may not be as rigorous as formal specifications; and

(b) On the other hand, with a little training, software testers in the large can understand and accept the new format.

We support the approach of "formalizing" the original specification, but observe that the degree of formalization in both methods may not be sufficient. We have a few concerns:

(i) For CPM, the original specifications are manually converted into some intermediate representations. This process is tedious and prone to human errors, especially when the original specification is complex.

(ii) CPM does not support consistency checking of intermediate representations. If these representations happen to be constructed incorrectly but the testers are not aware of the mistakes, some testing scenarios may not be covered.

(iii) We have done some work on CTM. We observed, however, that it could not model some of the essential constraints of the input domain because of an inherent limitation of the tree structure. This resulted in the generation of illegitimate test cases. Although the problems could be solved by separate techniques, an aggregation of independent solutions might not be the most desirable in a method. This observation has inspired us to change our focus to the development of a choice relation framework as described in this article.

Overview of Choice Relation Framework

In order to solve the above problems, we have developed a **CHOiCe reLATion framEwork**, abbreviated as CHOC'LATE, to support the generation of test suites from specifications. Our framework [3] is an extension of the original CPM but has incorporated many useful features. When compared with CTM and CPM,

(a) CHOC'LATE captures relatively more formal information from the specification by means of a highly structured intermediate format (namely, a *choice relation table* to be described later), which is closer to the right of the spectrum in Figure 1. Because of this, CHOC'LATE supports the automation of test suite generation to a high degree. Although the choice relation table is relatively more formal than the intermediate representations in CTM and CPM, it

can be understood by software developers with little formal training.

(b) CHOC'LATE provides a useful mechanism for checking the consistency of the choice relation table (see concern (ii)).

Besides, CHOC'LATE also incorporates two other useful mechanisms, namely automatic deductions of choice relations (see concern (i)) and the prioritization of choices for test suite generation. We have also developed a prototype system according to the framework.

It would be worthwhile to compare CHOC'LATE with pairwise testing [6, 10], which is another common test suite generation method. Basically, pairwise testing requires that, for each pair of input parameters of a software system, every combination of valid values of these two parameters be covered by at least one test case [6, 10]. Consider, for example, a software system with four input parameters; each of them can take three different values. Using pairwise testing, the number of test cases would be reduced from $3^4 = 81$ to nine. Although pairwise testing can reduce the number of generated test cases so as to save testing resources, it mainly focuses on faults caused by interactions between any two parameters. In other words, faults caused by interactions among three or more parameters may not be effectively dealt with by pairwise testing. This problem, however, does not exist in CHOC'LATE to be described as what follows.

Basically, CHOC'LATE consists of the following steps:

1. Decompose the specification into functional units. For each functional unit, repeat steps 2 to 6 below.
2. Identify categories and their associated choices.
3. Construct a choice relation table.
4. Construct a choice priority table and define the appropriate parameters.
5. Generate a set of complete test frames (CTFs).
6. Randomly construct test cases from generated CTFs.

We shall explain these steps using the following specification:

Specification for a Course Enrollment System `enroll`:

Develop a software system `enroll` for use by the academic secretariat of a university to process course enrollments by students. In order to evaluate whether an enrollment should be approved, `enroll` accepts the following inputs regarding the students concerned. [Each of the following inputs will affect the functions of `enroll`. The details, however, are not included here because they are not directly relevant to our framework.]

- Student ID: A 7-digit number.
- Degree Level: "Undergraduate" or "Postgraduate".
- Degree Type: "Coursework" or "Research". Note that all undergraduate degrees are by coursework, while postgraduate degrees can be by research or coursework.
- Degree: Examples are "BA", "BS", "BEng", "MBA", and "PhD".
- Number of Courses Enrolled (N): "$N = 0$", "$1 \leqslant N \leqslant 8$", or "$N > 8$".

Step 1: Decomposition of Specification. Given a large and complex system, the tester should decompose it into functional units that are smaller in size and manageable in complexity, so that each unit can be tested independently using CHOC'LATE. This will significantly ease the testing process. On the other hand, if the system is smaller and less complex, the tester can treat it as a single functional unit and, hence, no decomposition is needed. This is the case for the course enrollment system.

Step 2: Identification of Categories and Choices. The input elements of a functional unit can be divided into two types, namely *parameters* and *environment conditions*. The former are the explicit inputs to the functional unit supplied by the user or by another unit, whereas the latter are the states of the system at the time of executing the functional unit. The tester should identify all the input elements in a functional unit. Otherwise, software faults associated with a missing input element may not be detected.

The tester then identifies the *categories*, which are the properties or characteristics of a parameter or environment condition. For example, two possible categories for `enroll` are "Number of Courses Enrolled (N)" and "Status of Student's Record". The former is identified with respect to a parameter and the latter with respect to an environment condition.

For each category X, its associated *choices* should be identified. These choices are non-overlapping subsets of the values of X. Taken together, they cover every possible value of X. In `enroll`, for instance, the choices associated with the category "Number of Courses Enrolled (N)" are "$N = 0$", "$1 \leqslant N \leqslant 8$", and "$N > 8$"; while the choices associated with the category "Status of Student's Record" are "Does Not Exist", "Exists but Empty", and "Exists

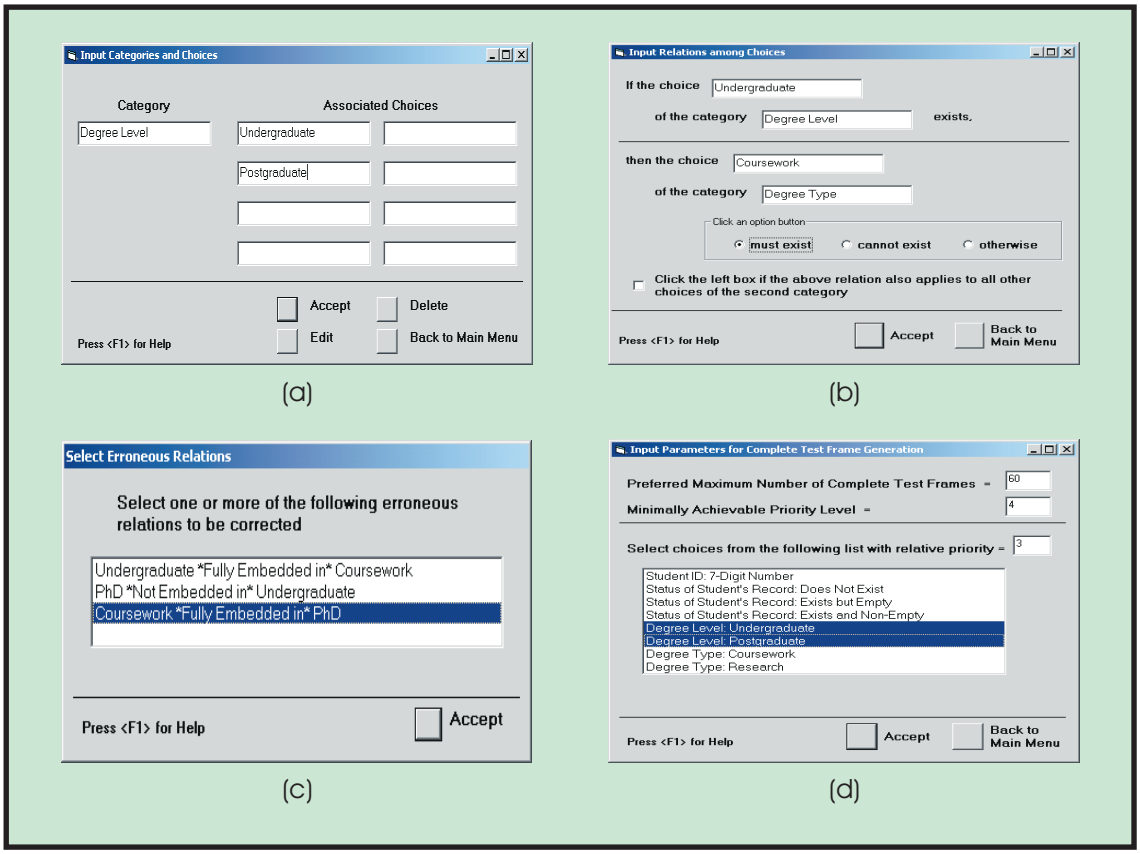

Figure 2. Sample screens of our prototype system.

and Non-Empty". See Figure 2(a) for the input screen for the categories and their associated choices provided by our prototype system for CHOC'LATE.

Step 3: Construction of Choice Relation Table. The tester then defines the constraints of the input domain. This step is very important because the correctness of the constraints affects the quality of the generated CTFs. An example of a constraint in `enroll` is that the choices "Undergraduate" and "PhD" in the categories "Degree Level" and "Degree", respectively, should not be combined to form part of any CTF. If this constraint were ignored, some of the generated CTFs might not correspond to legitimate inputs to the system.

In our framework, given a pair of choices x and y, their choice relation must be in one of three types, namely *full embedding* ($x \sqsubset y$), *partial embedding* ($x \sqsubset\!\!\!\!\cdot\; y$), and *nonembedding* ($x \not\sqsubset y$). Please refer to the callout "Types of Choice Relation" for an explanation of these relations. Basically, these choice relations aim at capturing the constraints of the input domain. They indicate how x and y should be combined to form part of a CTF. We use a choice relation table to capture these constraints. In order to improve on the effectiveness and efficiency of defining choice relations, we have identified numerous properties for these relations. These properties, then, form the basis for automatic deductions and consistency checking of choice relations, whose details will be described later. Readers may refer to [3] for a full list of these properties. We only list two of them here for illustration:

(Property 1) Given three choices x, y, and z, if $x \sqsubset y$ and $x \sqsubset\!\!\!\!\cdot\; z$, then $y \sqsubset\!\!\!\!\cdot\; z$.

(Property 2) Given three choices x, y, and z, if $x \sqsubset y$ and $z \not\sqsubset x$, then $y \sqsubset\!\!\!\!\cdot\; z$ or $y \not\sqsubset z$.

The "then" part of Property 1 consists of a definite relation and, hence, provides a basis for automatic deductions of choice relations. More specifically, if $x \sqsubset y$ and $x \sqsubset\!\!\!\!\cdot\; z$ are manually defined by the tester, $y \sqsubset\!\!\!\!\cdot\; z$ can be automatically deduced without human intervention. We have used four real-life commercial systems (or modules) to conduct a total of 20 trial runs, with a view to determining the effectiveness of the automatic deduction mechanism. The results of our studies show that, on average, 42 percent of choice relations can be deduced automatically [3].

As for Property 2, the "then" part contains two possible relations. Although this property cannot be used for

Test Frames and Their Completeness:
This callout refers to the `enroll` system described in the main body of the article.
A *test frame* TF is a set of choices. For example,

$$\text{TF}_1 = \{\text{ Exists and Non-Empty, Postgraduate, Research, PhD, } 1 \leqslant N \leqslant 8\ \}$$

is a test frame for `enroll` that contains five choices.
A TF is said to be *complete* if, whenever a single value is selected from each choice, a standalone input (that is, a test case) will be formed. Otherwise, it is *incomplete*. Consider the following combination of values:

$$C = \{\text{ Exists and Non-Empty, Postgraduate, Research, PhD, } N = 4\ \}.$$

C is formed by selecting a single value from each choice in the test frame TF_1 above. C cannot serve as a test case, however, because we also need the student ID in order to execute `enroll` for testing. Hence, TF_1 is incomplete.
Readers are reminded not to confuse the elements "Exists and Non-Empty", "Postgraduate", "Research", and "PhD" in TF_1 and C. Take "PhD" as an example. It is a choice in TF_1. This choice happens to have only one value, also known as "PhD". The latter is listed in C. ■

automatic deduction, it nevertheless allows us to check the consistency of the relations among choices. For example, we know that $x \sqsubset y$, $z \not\sqsubset x$, and $y \sqsubset z$ cannot coexist, or else they would contradict Property 2. The results of our studies show that almost all the choice relations incorrectly defined by the tester can be automatically detected as inconsistencies by CHOC'LATE immediately after these mistakes are made [3].

In summary, by means of the automatic deduction mechanism, the number of choice relations manually defined by the tester is significantly reduced. Furthermore, for those choice relations defined manually, their correctness will be verified by the consistency checking mechanism. These mechanisms thus decrease the chance of human errors. Readers may refer to Figure 2(b) for the input screen for the relation between a pair of distinct choices. Figure 2(c) shows a screen of our prototype system, which informs the tester that an inconsistency among relations has been detected, and asks the tester to pick the erroneous relation to be corrected.

Readers may note the check box (with the caption "Click the left box if the above relation also applies to all other choices of the second category") near the bottom of the input screen in Figure 2(b). The check box provides the tester with an option of defining group constraints through one single manual definition. This will further reduce the number of manual definitions required. Because of this option, according to the results of our studies [3], on average, an extra 28 percent of choice relations need not be defined individually. Thus, when considering the automatic deduction mechanism and group constraint definitions together, the amount of human effort is significantly reduced — only about 30 percent of the total number of choice relations have to be specified manually.

Step 4: Construction of Choice Priority Table and Definition of Parameters. Many real-life situations impose resource constraints on testing and, hence, not all the CTFs generated will actually be used in the testing process. Intuitively, it would be more effective to have an idea of the kinds of fault that are most probable or most damaging, and then generate CTFs that are likely to reveal these significant faults. One approach is to define the relative priorities for the choices based on expertise in testing and experience in the application domain. In this way, the choices with higher priorities can first be used to generate *test frames* (TFs). This generation process will continue until the number of generated TFs reaches the ceiling allowed by the testing resources. The relative priorities of the choices are captured in a *choice priority table*. In this table, choices with higher relative priorities are expected to have higher chances of revealing more significant faults.

Besides constructing the choice priority table, the tester also needs to define the ceiling permitted by the testing resources. This is achieved through the parameter *preferred maximum number of test frames* ($\overline{M}$). The word "preferred" implies that $\overline{M}$ is not absolute, as the ceiling may be overwritten by the parameter to be described in the next paragraph.

In addition to $\overline{M}$, we have another parameter, which indicates the minimal priority level. Any choice having a relative priority higher than this minimum will always be selected for inclusion as part of a TF, no matter whether the number of generated TFs exceeds $\overline{M}$. In essence, the minimal priority level guarantees that the choices more likely to detect significant faults will always be used to form TFs irrespectively of the testing resources. Figure 2(d) shows an input screen that allows testers to define the relative priority for every choice, $\overline{M}$, and the

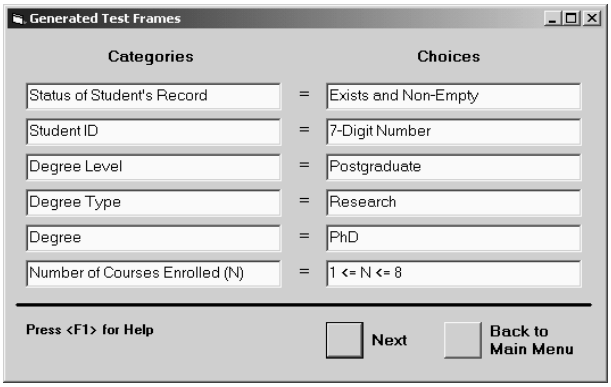

Figure 3. A test frame for `enroll`.

minimal priority level.

Step 5: Generation of Complete Test Frames. CHOC'LATE adopts an incremental approach to generating TFs based on the choice relation table, the choice priority table, $\overline{M}$, and the minimal priority level. Most parts of the generation process are automatically performed by CHOC'LATE without human intervention. This is achieved by means of automatic deduction of the relation between a single choice x and a TF, which is similar to the automatic deduction mechanism for a pair of choices as described in step 3. Readers may refer to [3] for details. Figure 3 shows a TF generated by our prototype system for `enroll`.

Step 6: Generation of Test Cases from Complete Test Frames. For every CTF, the tester then randomly selects a single element from each choice. The set of elements thus selected in every CTF will constitute a test case. Consider, for example, the TF shown in Figure 3. We can randomly select a test case from it:

> Exists and Non-Empty, 0411875, Postgraduate, Research, PhD, $N = 4$

When generating the above test case, "0411875" and "$N = 4$" are randomly selected from the choices "Student ID" and "Number of Courses Enrolled (N)", respectively.

Instead of randomly picking an element from each choice, another approach is to use the concept of boundary value analysis (BVA). This concept is based on the observation that test cases exploring boundary conditions have a higher payoff in revealing failures than test cases that do not. Here, *boundary conditions* are those situations directly on, above, and beneath the edges of partitions (which are similar to choices in CHOC'LATE). Consider, for instance, the choice "$1 \leq N \leq 8$" in the category "Number of Courses Enrolled (N)". According to BVA, boundary values such as 1 and 8 are better candidates for test case generation than other elements such as 4 and 5.

As mentioned in step 3 above, we have successfully applied CHOC'LATE to four real-life commercial systems (or modules) [3]. They include the inventory registration module and the purchase-order generation module of an inventory management system used by a group of public hospitals, an online telephone inquiry system used in a large telecom company, and the meal scheduling module of a meal ordering system used by an international airline catering company. We also note that CTM has been successfully used in several industrial applications, such as modules in an airfield lighting system and an adaptive cruise control system. Since CHOC'LATE is similar to CTM with respect to the identification of categories, choices, and constraints (but with many improvements such as automatic deductions and consistency checks of choice relations), it is not difficult to see that CHOC'LATE can also be applied to these application domains. In short, CHOC'LATE can be applied to software systems in different application domains, provided that these systems can be decomposed into functional units which can be tested independently (step 1 in our framework), and that categories, choices, and choice relations can be identified (steps 2 and 3 in our framework).

Summary and Conclusion

Specification-based testing remains a popular approach of software testing, for which numerous test suite generation methods have been proposed by researchers. These methods, however, suffer from similar problems that hinder their effective application. We have developed a choice relation framework CHOC'LATE with a view to solving these problems. CHOC'LATE outperforms other methods in several aspects. First, the concept of choice relations allows testers to systematically re-enact an unstructured informal specification in a more formal representation — a choice relation table — from which a test suite can be effectively generated and the generation process can be automated. This degree of formalization does not exist in other generation methods. Secondly, unlike formal specifications, the choice relation table is easy to understand by software developers with little formal training. Thirdly, CHOC'LATE incorporates mechanisms for the consistency checking and automatic deductions of choice relations, as well as the prioritization of choices for test suite generation. These useful features have contributed to the uniqueness of the framework. Because of these merits, we believe that CHOC'LATE will have a significant contribution to software quality assurance in the industry. ■

Types of Choice Relation:
Given a pair of choices x and y, their choice relation must be of one of the following three types:

- x is ***fully embedded*** in y (denoted by $x \sqsubset y$) if, any CTF that contains x will also contain y. In `enroll`, for example, PhD programs are research-based. Hence, we have "PhD" $\sqsubset$ "Research", indicating that every CTF containing the choice "PhD" must also contain the choice "Research".
- x is ***partially embedded*** in y (denoted by $x \Subset y$) if (i) there exists some CTF that contains x and y, and (ii) there exists some other CTF$'$ that contains x but not y. In `enroll`, for instance, there is no logical relationship between undergraduates and the number of courses enrolled. Hence, we have "Undergraduate" $\Subset$ "$N = 0$", indicating that any CTF containing "Undergraduate" may or may not contain "$N = 0$".
- x is ***not embedded*** in y (denoted by $x \not\sqsubset y$) if any CTF cannot contain both x and y. In `enroll`, for example, all PhD programs are at the postgraduate level. Hence, we have "Undergraduate" $\not\sqsubset$ "PhD", indicating that it is impossible for a CTF to contain both the choices "Undergraduate" and "PhD". ■

References

[1] S. Beydeda, V. Gruhn, and M. Stachorski. A graphical class representation for integrated black- and white-box testing. In *Proceedings of the IEEE International Conference on Software Maintenance* (*ICSM 2001*), pages 706–715. IEEE Computer Society Press, Los Alamitos, CA, 2001.

[2] R. V. Binder. Category-partition test design pattern. Excerpt from *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison Wesley, Reading, MA, 2000. Available at http://www.rbsc.com/docs/TestPatternCategoryPartition.pdf.

[3] 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.

[4] N. Gupta, A. P. Mathur, and M. L. Soffa. Generating test data for branch coverage. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering* (*ASE 2000*). IEEE Computer Society Press, Los Alamitos, CA, 2000.

[5] R. M. Hierons, M. Harman, and H. Singh. Automatically generating information from a Z specification 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, Germany, 2003.

[6] N. Kobayashi, T. Tsuchiya, and T. Kikuno. A new method for constructing pair-wise covering designs for software testing. *Information Processing Letters*, 81 (2): 85–91, 2002.

[7] M. F. Lau and Y. T. Yu. An extended fault class hierarchy for specification-based testing. *ACM Transactions on Software Engineering and Methodology*, 14 (3): 247–276, 2005.

[8] J. J. Li, D. Weiss, and H. Yee. Code-coverage guided prioritized test generation. *Information and Software Technology*, 48 (12): 1187–1198, 2006.

[9] P. Stocks and D. A. Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22 (11): 777–793, 1996.

[10] K.-C. Tai and Y. Lei. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering*, 28 (1): 109–111, 2002.

[11] T. Tsuchiya and T. Kikuno. On fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering and Methodology*, 11 (1): 58–62, 2002.

[12] S. A. Vilkomir and J. P. Bowen. From MC/DC to RC/DC: formalization and analysis of control-flow testing criteria. *Formal Aspects of Computing*, 18 (1): 42–62, 2006.

Pak-Lok Poon (afplpoon@inet.polyu.edu.hk) is an associate professor at the School of Accounting and Finance in The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong, responsible for the teaching of accounting information systems, systems analysis and design, and computer security, audit, and control.

Sau-Fun Tang (satang@ict.swin.edu.au) is a PhD candidate at the Faculty of Information and Communication Technologies in Swinburne University of Technology,

Melbourne, Australia. She was an instructor at the Department of Finance and Decision Sciences of Hong Kong Baptist University and a lecturer at the School of Accounting and Finance in The Hong Kong Polytechnic University.

T. H. Tse (thtse@hku.hk) is a professor at the Department of Computer Science of The University of Hong Kong, Pokfulam, Hong Kong.

T. Y. Chen (tchen@ict.swin.edu.au) is a chair professor of software engineering in the Faculty of Information and Communication Technologies, Swinburne University of Technology, Melbourne, Australia.

This research is supported in part by a grant of the Research Grants Council of Hong Kong (project no. PolyU 5177/04E)