An empirical evaluation of several test-a-few strategies for testing particular conditions [*, †]

Eric Ying Kwong Chan[1], W.K. Chan[1], Pak-Lok Poon[2, ‡] and Yuen Tak Yu[1]

[1] *Department of Computer Science, City University of Hong Kong, Kowloon Tong, Hong Kong*
[2] *School of Accounting and Finance, The Hong Kong Polytechnic University, Hung Hom, Hong Kong*

SUMMARY

Existing specification-based testing techniques often generate a comprehensive test suite to cover diverse combinations of test-relevant aspects. Such a test suite can be prohibitively expensive to execute exhaustively due to its large size. A pragmatic strategy often adopted in practice, called *test-once*, is to identify certain *particular conditions* from the specification and to test each such condition once only. This strategy is implicitly based on the *uniformity assumption* that the implementation will process a particular condition uniformly, regardless of other parameters or inputs. As the decision of adopting the test-once strategy is often based on the specification, whether the uniformity assumption actually holds in the implementation needs to be critically assessed, or else the risk of inadequate testing could be non-negligible. As viable alternatives to reduce such a risk, this paper proposes a family of test-a-few strategies for the testing of particular conditions. It further reports two rounds of experiments that evaluate the effectiveness of the test-a-few strategies as compared to test-once. Our experiments (1) provide clear evidence that the uniformity assumption often, but not always, holds, and that the assumption usually fails to hold when the implementation is faulty, (2) demonstrate that all our proposed test-a-few strategies are statistically more reliable than test-once in revealing faulty programs, (3) show that random sampling is already substantially more effective than test-once, and (4) indicate that, compared to other test-a-few strategies under study, choice coverage seems to achieve a better tradeoff between test effort and effectiveness.

KEY WORDS: particular condition; software testing; software validation; test-a-few strategy; test-once strategy; test case selection

SHORT TITLE: Evaluation of test-a-few strategies for testing particular conditions

[*] The work described in this paper was supported in part by grants (project numbers: CityU 123206, CityU 123207) from the Research Grants Council of the Hong Kong Special Administrative Region, China, and a Strategic Research Grant of City University of Hong Kong (project number: 7008039).

[†] The email addresses for the authors are chanyk@cs.cityu.edu.hk, wkchan@cs.cityu.edu.hk, afplpoon@inet.polyu.edu.hk, and csytyu@cityu.edu.hk, respectively.

[‡] Correspondence to: Pak-Lok Poon, School of Accounting and Finance, The Hong Kong Polytechnic University, Hung Hom, Hong Kong.

1. INTRODUCTION

Specification-based testing is a mainstream testing approach in which test cases are generated according to information derived from the specification without requiring the knowledge of implementation details [1, 2, 3]. This approach is popular because it can be applied even when the source code of a program is unavailable.

In specification-based testing, test cases are derived by considering the important aspects, which we call *test-relevant aspects*, that are identified from the specification [1, 4, 5, 6, 7]. Examples of test-relevant aspects are the applicant's credit status and monthly salary for a home-loan mortgage application system in the banking industry. A non-trivial application system often involves many test-relevant aspects and, hence, a large test suite may be formed to cover diverse combinations of these aspects [1, 7, 8]. For instance, it has been reported that real-world protocol software may have 448–2402 test cases per test suite [9]. The *test-all strategy*, which exhaustively executes all the test cases in a large test suite for testing, can be costly or even infeasible under limited testing resources [3, 10, 11, 12].

A popular and pragmatic means to deal with this problem is to apply the *test-once strategy* to certain *particular input conditions* (or simply called *particular conditions*) [10], that is, to identify such conditions from the specification and then test each of them once only. Particular conditions represent "special, unusual, or redundant conditions" or those features/scenarios "which will cause an exception or other error state" that are deemed to be adequately tested without the need to combine them with all other conditions [7]. Implicitly, these conditions are assumed to be somehow treated "uniformly" by the program regardless of the values of other inputs. We refer to such an assumption as the *uniformity assumption*. This paper focuses on the strategies adopted to test particular conditions and the issues related to the associated uniformity assumptions.

Consider, for instance, a program *P* which attempts to read data from an input file *F*, but *F* cannot be opened for some reasons. To keep the testing effort low, the tester may decide to select only one test case, under the belief that *P* should raise a file-open-exception, irrespective of the values of other input parameters or conditions. In so doing, the tester has in effect associated the uniformity assumption with the particular condition of file-open-exception. In other words, the tester tacitly assumes that the selected test case will either run with the exception raised (as expected), giving adequate confidence that *P* implements the file-open-exception situations correctly, or that *P* behaves incorrectly (producing a different, unexpected outcome) on execution of the selected test case, hence revealing the existence of a bug in *P*. Indeed, the uniformity assumption is analogous to the *uniformity hypothesis* [13] in partition testing [6], whereby any input within an input subdomain is considered as likely as other inputs in the same subdomain for detecting faults in the program.

Particular conditions are often determined by the tester's intuitive or professional judgement based primarily on the program specification, thus necessarily inducing a risk that the corresponding uniformity assumption may actually not hold for some implementations of the specification. This in turn implies a risk that the test-once strategy may be inadequate for the implementation under test. Whether this risk is negligible depends on the extent to which the uniformity assumption holds, which in turn needs to be critically assessed. If, however, the belief of the uniformity assumption is not so high, then ways to reduce the associated risks will be needed. These ideas form the thesis on which this research is based.

This paper serves two primary objectives. First, it assesses empirically the validity of uniformity assumptions associated with particular conditions, with respect to a large number of implementations of the same specification. Second, we propose and empirically evaluate a family of strategies, called *test-a-few strategies*, as alternatives to test-once.

Specifically, we consider three test-a-few strategies, namely, *random sampling*, *choice coverage*, and *choice-occurrence sampling*. This is followed by a description of two rounds of experiments that we conducted to compare the effectiveness of the test-once and test-a-few strategies, in terms of their ability of program code coverage and fault detection. Through these experiments, we find many interesting observations. The experiments involve 161 human participants and the programs they implemented individually, and evaluate to what extent the test-once or a test-a-few strategy can reveal failures in these programs. The empirical result shows that the test-a-few strategies are more reliable than the test-once strategy in detecting faulty programs. Moreover, in the experiments, faulty programs were found to be more likely (compared with correct programs) to deviate from the uniformity assumption, in the sense that the former exercise multiple program paths to implement the same particular condition when other inputs vary. Since the test-once strategy only executes one test case (and, hence, one program path), other program paths that implement the same particular condition may not be exercised at all. Thus, our empirical data provide a clear rationale for testers to apply test-a-few strategies instead of test-once. Comparing the choice coverage and choice-occurrence sampling strategies, the empirical result favours choice coverage as it demands fewer test cases while achieving almost the same effectiveness as choice-occurrence sampling. Furthermore, for the testing of particular conditions, the random sampling strategy is already substantially more effective than test-once. Compared with choice coverage and choice-occurrence sampling, however, random sampling is found to be somewhat less effective, particularly when the sampling rate is as low as 1%. The empirical result also shows that the uniformity assumption is more likely to hold for input conditions related to exception scenarios than those related to special conditions of processing for a business application.

The contribution of this paper, as an extended version of its preliminary one [10], is fourfold. (a) We provide empirical data to caution the tester that the risk of adopting the test-once strategy should not be lightly discarded. In particular, our experiments are the first that report to what extent program implementations (especially the faulty ones) may not conform to the uniformity assumption, on which the test-once strategy is based. (b) We propose the notion of test-a-few as a viable alternative to provide better assurance to the testing of particular conditions, and formulate three concrete and feasible strategies to exemplify the implementation of this notion. (c) We present two rounds of experiments that involve real faults committed by human participants. To our knowledge, they are the largest experiments that specifically study particular conditions, especially in terms of the number of implementations for the same specification. (d) The statistical analysis of our empirical data shows that although the test-a-few strategies require slightly more test cases, they are statistically more reliable and more effective than the test-once strategy in the testing of particular conditions.

The organization of the rest of this paper is as follows. Section 2 outlines the major steps of specification-based testing to introduce the context of this research. Section 3 discusses the test-all and test-once strategies. Section 4 proposes three test-a-few strategies as alternatives to the test-once strategy when the risk of violating the uniformity assumption is non-negligible. Section 5 describes and discusses our two rounds of experiments, with a view to empirically evaluating the effectiveness of the test-once and test-a-few strategies, in terms of their ability of program path coverage and faulty program detection. Section 6 further elaborates the thesis of this paper in light of the empirical results. Section 7 compares related studies with our work in this paper. Section 8 concludes the paper and suggests several directions for further work.

2. OVERVIEW OF SPECIFICATION-BASED TESTING

Before discussing in detail the potential problems associated with the test-once strategy, here we first outline the essential steps of many common specification-based testing methods in order to introduce our terminologies and context.

Many specification-based testing methods, such as cause-effect graphing (CEG) [14], combinatorial testing [8, 11, 15], the category-partition method (CPM) [4, 7], the choice relation framework (CHOC'LATE) [1], and the classification-tree method (CTM) [5, 6], require the tester to identify instances that influence the functions of a software system and generate a test suite by systematically varying these instances over all values of interest. Generally, these methods include the following three main steps:

(a) Identify test-relevant aspects from the specification.

(b) Identify the constraints (or relationships) among these aspects as stated or implied in the specification.

(c) Combine the test-relevant aspects to form test cases [1] in accordance with the constraints identified in (b).

Example 1 below illustrates the above three steps.

Example 1 (Specification-based testing): Consider part of the specification S_{SCS} for a smart-card system (SCS) as follows.

```
   A bank based in Hong Kong introduces a smart-card system (SCS) so that its
customers can use the smart cards issued to them as an alternative payment
method to cash when making purchases. The idea is to allow the cardholders to
withdraw the funds directly from the remaining balances on the cards when
making payments.

   To ensure sufficient funds in their smart cards, customers must deposit
money into their cards first. The deposit is made via designated fund-deposit
machines located at various branches of the bank. The deposit can be made in
local or foreign currencies. When the deposit is made in a foreign currency, it
will be automatically converted into an equivalent amount in Hong Kong dollars
(HK$) using the prevalent exchange rate at the time of deposit. Unlike fund
deposit, purchases using the smart cards must be made in HK$ because the cards
are designed for exclusive use in Hong Kong.

   There are three prestigious levels of smart cards, namely, platinum, gold,
and classic. To promote the use of platinum and gold cards, the bank has
introduced an incentive scheme. While a HK$1.00 surcharge is imposed on every
purchase with the classic card, no surcharge applies to the use of the platinum
or gold card. In addition, customers are entitled to receive a 5% and 2%
discount for any purchase with their platinum and gold cards, respectively …
```

In this paper, we are going to use CPM to illustrate how to derive a comprehensive test suite from S_{SCS}. Readers, however, should note that the discussion here is largely applicable to other specification-based testing methods, such as CEG, CHOC'LATE, and CTM.

[1] In CPM, a combination of test-relevant aspects is known as a *test frame*. In this paper, however, we will use the term "test cases" instead of "test frames".

Step (a): CPM requires the tester to identify test-relevant aspects known as *categories*. For each category [*X*], a set of associated *choices* should be identified. These choices are disjoint subsets of values for [*X*], and altogether they should cover all the possible values for [*X*]. In this paper, following the notation used in [4], categories and choices will be enclosed by [ ] and | |, respectively. Figure 1 shows some categories and choices identified from S_{SCS}. Note that the surcharge is part of a business rule. It is not an input but implied by other inputs and, hence, is not treated as a category. In this specification, if the smart card is a classic one, the surcharge must be HK$1.00 for each purchase; otherwise the surcharge is zero.

Step (b): The tester specifies the constraints among the choices of different categories in accordance with S_{SCS}. For example, the choice |Purchase| in the category [Transaction Type] cannot feasibly coexist with |Non-HK$| in [Currency] as part of any test case because, according to S_{SCS}, purchases with smart cards can only be made in HK$.

With the identified categories, choices, and constraints, a test specification is written in a test specification language (TSL) [1, 7]. Figure 1 shows a partial test specification for S_{SCS} in TSL, in which some of the choices are associated with annotations to be explained as follows.

A choice may be annotated with a property list (such as {`property` `deposit`} for |Deposit|) and/or with a selector expression (such as {`if` `deposit`} for |Non-HK$|). Together the property lists and selector expressions specify the constraints among individual choices of different categories. For example, in Figure 1, because |Deposit| is annotated with the property list {`property` `deposit`} and |Non-HK$| is annotated with the selector expression {`if` `deposit`}, |Non-HK$| will be selected to form part of a test case only if |Deposit| is also selected. Readers may refer to [1, 7] for more details. The annotation {`single`} associated with the two choices |Trans Amount < $0.00| and |Trans Amount = $0.00| will be discussed later in Section 3.

Step (c): With the test specification, a generator tool (such as those described in [7, 16]) can be used to generate a test suite, in which each test case corresponds to a unique combination of choices. Such a tool will generate a test suite TS_{SCS} of 216 test cases from the complete test specification of S_{SCS} if the annotation {`single`} associated with the two choices |Trans Amount < $0.00| and |Trans Amount = $0.00| is ignored for the moment. Note that Figure 1 shows only part of the test specification for S_{SCS}, and the annotation {`single`} will be explained in Section 3 below. ■

```
[Card Type]
   |Platinum|
   |Gold|
   |Classic|

[Transaction Type]
   |Deposit|                 {property deposit}
   |Purchase|

[Transaction Amount]
   |Trans Amount < $0.00|    {single}
   |Trans Amount = $0.00|    {single}
   |Trans Amount > $0.00|

[Currency]
   |HK$|
   |Non-HK$|                 {if deposit}
   ⋮
```

Figure 1. A partial TSL test specification for S_{SCS}.

3. THE TEST-ALL AND TEST-ONCE STRATEGIES

In Example 1, TS_{SCS} is non-redundant with respect to S_{SCS}, because every test case is formed by choices whose combinations may affect the execution behaviour of SCS in some way. Hence, TS_{SCS} is comprehensive in its consideration of test-relevant aspects. The *test-all strategy*, which executes the entire test set TS_{SCS}, is expected to be effective in detecting faults, but it is often very costly to do so as TS_{SCS} is fairly large and resource-demanding. In other words, there is a tradeoff between the scope of testing (or the effectiveness of fault detection) and the required testing resources.

In practice, when the test suite is large, many testers would try to reduce it by all means to save testing costs. One strategy, which we call the *test-once* strategy, is to identify from the specification some particular conditions, each of which is judged to be tested adequately with a single test case. Typically considered particular conditions are unusual scenarios, errors or exceptions, and special treatments that are expected or assumed to be processed uniformly by the program regardless of what other inputs or conditions are (the uniformity assumption).

Similar to other specification-based testing methods, CPM allows testers to express their intention to test a particular condition only once. For a choice $|x|$ annotated with {`error`} or {`single`}, a single test case containing only $|x|$ will be formed. The {`error`} annotation signifies an input condition that will cause an error or exception state. It is tacitly assumed that any call of the function with this particular value in the annotated choice will result in the same error. On the other hand, the {`single`} annotation describes special, redundant, or unusual conditions that are judged to need only be tested once. Notwithstanding the different rationales underlying the uniformity assumption, both {`error`} and {`single`} annotations have the same effect, that is, to generate one test case containing the annotated choice only.

Note that while the use of the annotations {`error`} and {`single`} in CPM to support the test-once strategy can greatly reduce the size of the generated test suite, it may at the same time substantially reduce the effectiveness of the test suite in fault detection, depending on how the system is implemented. This issue is illustrated in Example 2 below.

Example 2 (Test-once strategy): The uniformity assumption usually comes about when the tester has some intuition of how the program is "likely" to be implemented. For instance, the tester may judge that S_{SCS} will be implemented in a way as shown in Figure 2. If so, then there is no real need of testing every test case that contains |Trans Amount < \$0.00|. Some of these test cases are:

- Test case 1 = (|Platinum|, |Purchase|, |Trans Amount < \$0.00|, …)
- Test case 2 = (|Gold|, |Purchase|, |Trans Amount < \$0.00|, …)
- Test case 3 = (|Classic|, |Deposit|, |Trans Amount < \$0.00|, …)

Here, any one of the above three test cases has the same chance of detecting possible faults that are associated with the particular condition |Trans Amount < \$0.00|. For example, such a fault may occur, say, when statement a31 in Figure 2 is wrongly coded as "**return** msg_accept_trans;", which outputs a message that indicates acceptance instead of rejection of the transaction.

By annotating the choice |Trans Amount < \$0.00| with {`single`} in the partial test specification in Figure 1, only one test case containing this choice will be generated without prescribing all other choices. Based on a similar consideration, the tester also annotates the choice |Trans Amount = \$0.00| with {`single`}. With the addition of these two {`single`} annotations (see Figure 1), the number of test cases generated from the complete test specification will be reduced to 74.

Partial implementation for S_{SCS}

```
a01   procedure smart_card (card_type, trans_type,
                            trans_amount, … ) ;
:     :
a30   if trans_amount ≤ 0.00 then
a31     return msg_reject_trans ;  /* return msg & exit */
a32   endif;
a33   begin case
a34     case trans_type = "PURCHASE" :
a35       begin_case
a36         case card_type = "PLATINUM" :
a37           adjusted_amount = trans_amount × 0.95 ;
a38         case card_type = "GOLD" :
a39           adjusted_amount = trans_amount × 0.98 ;
a40         case card_type = "CLASSIC" :
a41           adjusted_amount = trans_amount + 1.00 ;
a42         end_case ;
:         :
a60     case trans_type = "DEPOSIT" :
a61         balance = balance + trans_amount ;
:         :
a90     end_case ;
:       :
```

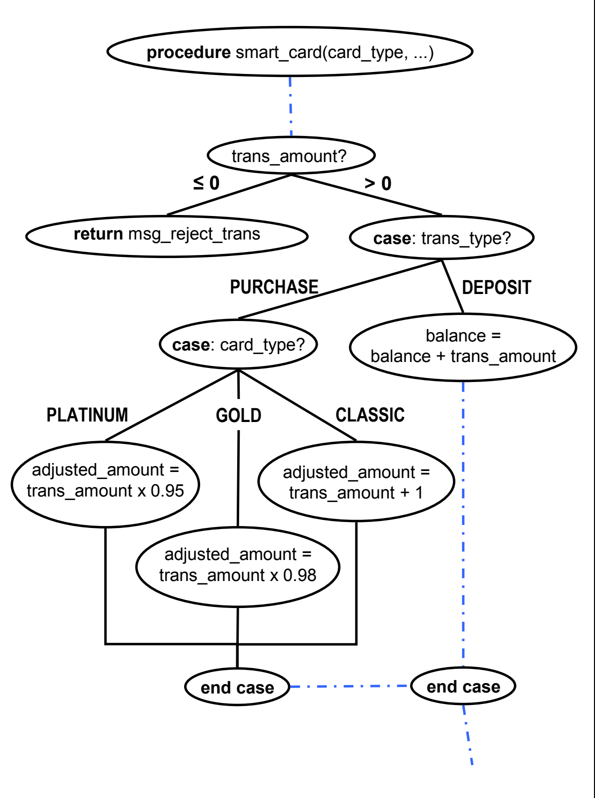

procedure smart_card(card_type, ...)
trans_amount?
≤ 0
> 0
return msg_reject_trans
case: trans_type?
PURCHASE
DEPOSIT
case: card_type?
balance = balance + trans_amount
PLATINUM
GOLD
CLASSIC
adjusted_amount = trans_amount x 0.95
adjusted_amount = trans_amount + 1
adjusted_amount = trans_amount x 0.98
end case
end case

Figure 2. A correct partial implementation of S_{SCS} and its flow graph.

Obviously, unless the tester actually has access to the code, that a particular condition will be adequately tested by only one test case is really an assumption that is sometimes not true. For instance, although the tester expects that S_{SCS} will be implemented as in Figure 2 (and so annotates the choices |Trans Amount < \$0.00| and |Trans Amount = \$0.00| with {**`single`**}), yet the function may in fact be implemented in some other ways as shown in Figures 3 and 4, which correspond to an incorrect and another correct implementations of S_{SCS}, respectively.

Consider the implementation in Figure 3 first. A careful scrutiny of the code shows a fault in statement b40: statement b41 will not be executed to output the error message when a classic card is used to purchase goods with a zero transaction amount. This is because the faulty statement b40 checks the value of the variable adjusted_amount, which would have been set to 1.00 in statement b38 even though trans_amount = 0.00.

Now, since the choices |Trans Amount < \$0.00| and |Trans Amount = \$0.00| have been annotated with {**`single`**}, only the following two of the generated test cases contain either of these two choices:

- Test case 4 = (|Trans Amount < \$0.00|)
- Test case 5 = (|Trans Amount = \$0.00|)

Refer to Figure 3 again. When test case 5 is used for testing SCS, the tester may select any one of the values "`PLATINUM`", "`GOLD`", or "`CLASSIC`" for the variable card_type, "`PURCHASE`" for the variable trans_type and 0.00 for the variable trans_amount. If "`CLASSIC`" is the value selected for the variable card_type, then statement b38 will be executed and the fault in statement b40 will be revealed. But if the tester selects "`PLATINUM`" or "`GOLD`" for the variable card_type, then the fault in statement b40 will remain hidden. In a similar way, it is not difficult to see that test case 4 may not always detect the fault either. Thus, test cases 4 and 5, generated as a result of the particular conditions |Trans Amount < \$0.00| and |Trans Amount = \$0.00|, are not effective in detecting the fault.

Partial implementation for S_{SCS}

```
b01   procedure smart_card (card_type, trans_type,
                            trans_amount, … ) ;
:     :
b30   begin case
b31     case trans_type = "PURCHASE" :
b32       begin_case
b33         case card_type = "PLATINUM" :
b34           adjusted_amount = trans_amount × 0.95 ;
b35         case card_type = "GOLD" :
b36           adjusted_amount = trans_amount × 0.98 ;
b37         case card_type = "CLASSIC" :
b38           adjusted_amount = trans_amount + 1.00 ;
b39         end_case ;
b40       if  adjusted_amount ≤ 0.00 then    /* FAULTY !! */
b41         return msg_reject_trans ;  /* return msg & exit */
b42       endif ;
:         :
b60     case trans_type = "DEPOSIT" :
b61         balance = balance + trans_amount ;
:           :
b90     end_case ;
:       :
```

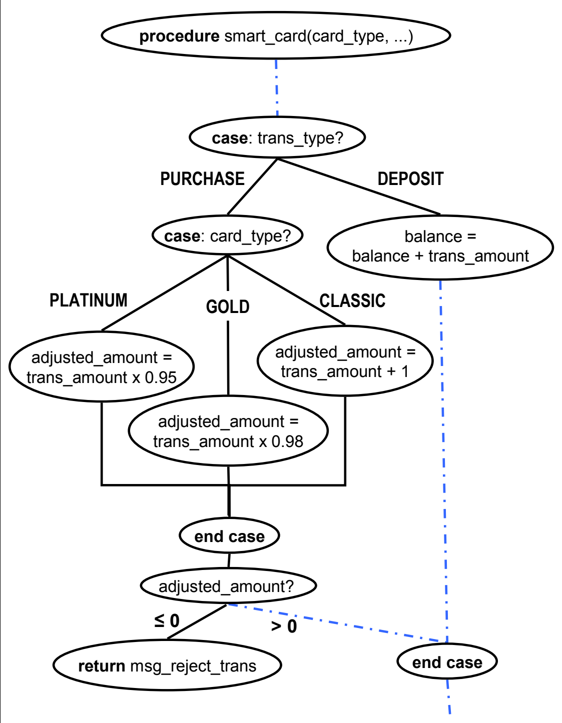

procedure smart_card(card_type, ...)
case: trans_type?
PURCHASE
DEPOSIT
case: card_type?
balance = balance + trans_amount
PLATINUM
GOLD
CLASSIC
adjusted_amount = trans_amount x 0.95
adjusted_amount = trans_amount + 1
adjusted_amount = trans_amount x 0.98
end case
adjusted_amount?
≤ 0
> 0
return msg_reject_trans
end case

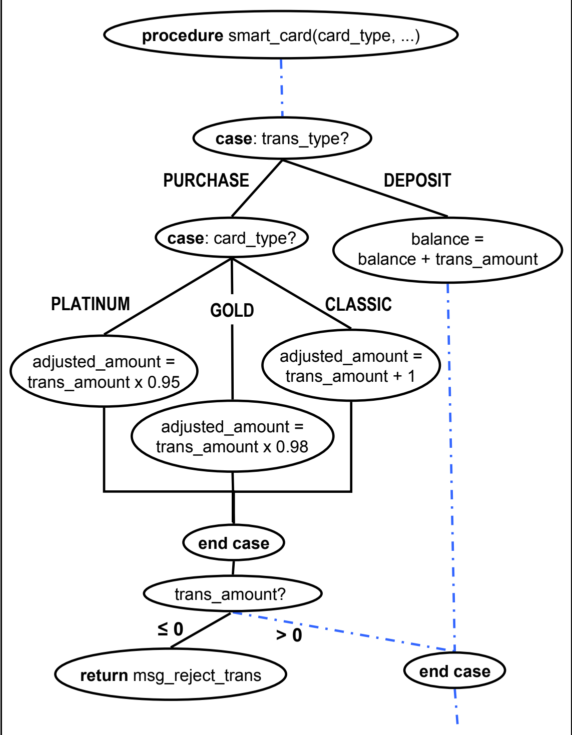
Figure 3. An incorrect partial implementation of S_{SCS} and its flow graph.

Next, consider another correct implementation shown in Figure 4, which differs from Figure 3 only in the variables in statements b40 and c40, respectively. It is clear in Figure 4 (and also in Figure 3) that the uniformity assumptions associated with the particular conditions |Trans Amount < \$0.00| and |Trans Amount = \$0.00| do not hold. Consider, for example, the combinations of input values (card_type = "`PLATINUM`", trans_type = "`PURCHASE`", trans_amount = 0.00) and (card_type = "`GOLD`", trans_type = "`PURCHASE`", trans_amount = 0.00). The first combination involves the execution of statement c34, whereas the second combination involves the execution of statement c36. In other words, the implementation in Figure 4 will treat the two test cases differently. Now, suppose statement c34 in Figure 4 is wrongly coded as "adjusted_amount = trans_amount × 0.59". Annotating the choices |Trans Amount < \$0.00| and |Trans Amount = \$0.00| with {**`single`**} will result in the generation of test cases 4 and 5, which are the only ones in the test suite containing the choice |Trans Amount < \$0.00| or |Trans Amount = \$0.00|. In this case, the testers may select at their own will the values for the variables trans_type and card_type. Should the value "`PURCHASE`" for the variable trans_type and the value "`PLATINUM`" for the variable card_type be selected, the hypothetical fault in statement c34 may be revealed. On the other hand, if "`PURCHASE`" for the variable trans_type and ("`GOLD`" or "`CLASSIC`") for the variable card_type are selected, the hypothetical fault will not be uncovered. ■

In summary, with the test-once strategy, only one test case is executed for each particular condition, and there is no obvious way to determine whether or not the uniformity assumption for the particular condition holds. It is with this observation and for this purpose that we introduce in the next section several alternative strategies for testing particular conditions.

Partial implementation for S_{SCS}

```
c01    procedure smart_card (card_type, trans_type,
                             trans_amount, … ) ;
:      :
c30    begin case
c31      case trans_type = "PURCHASE" :
c32        begin_case
c33          case card_type = "PLATINUM" :
c34            adjusted_amount = trans_amount × 0.95 ;
c35          case card_type = "GOLD" :
c36            adjusted_amount = trans_amount × 0.98 ;
c37          case card_type = "CLASSIC" :
c38            adjusted_amount = trans_amount + 1.00 ;
c39          end_case ;
c40        if  trans_amount ≤ 0.00 then        /* CORRECT */
c41          return msg_reject_trans ;  /* return msg & exit */
c42        endif ;
:          :
c60      case trans_type = "DEPOSIT" :
c61          balance = balance + trans_amount ;
:            :
c90      end_case ;
  :      :
```



procedure smart_card(card_type, ...)
case: trans_type?
PURCHASE
DEPOSIT
case: card_type?
balance = balance + trans_amount
PLATINUM
GOLD
CLASSIC
adjusted_amount = trans_amount x 0.95
adjusted_amount = trans_amount + 1
adjusted_amount = trans_amount x 0.98
end case
trans_amount?
≤ 0
> 0
return msg_reject_trans
end case

Figure 4. Another correct partial implementation of S_{SCS} and its flow graph.

4. TEST-A-FEW STRATEGIES

Let N (> 1) be the total number of test cases that involve a particular condition C, and n be the number of test cases *actually executed* for testing C. Then we have $1 \leq n \leq N$. The test-once and test-all strategies correspond to the two extreme cases when $n = 1$ and $n = N$, respectively.

When no code information is available, in order to reduce the possible risk of violating the uniformity assumption concerning a particular condition C, we propose to execute the program with *a few* (but not too many) test cases involving C (that is, $1 < n < N$). We refer to such a strategy as *test-a-few strategy*. Instead of reverting to the test-all strategy, however, n should normally be small (compared to N) so that the overall testing costs should not be greatly increased. For instance, if each of the two particular conditions in Example 2 is tested with three test cases instead of just one, the total number of test cases selected for execution will only increase from 74 to 78 (that is, an increase of 5.4%).

The value of n should depend on how strongly the tester believes in the uniformity assumption for C. Obviously, if the tester's belief is firm, the test-once strategy ($n = 1$) may be used. At the other extreme, if the tester suspects that C will interact with all other conditions in a variety of ways to affect the program's behaviour, the test-all strategy ($n = N$) should be more appropriate. The test-a-few strategy ($1 < n < N$) may be seen as positioned between the two extremes, but is closer to test-once than test-all.

There is another rationale that suggests using $n > 1$. If the uniformity assumption in fact does not hold, it will have a non-zero chance (which increases with n) of being revealed by the execution of two or more test cases. In other words, the use of more than one test case is a *necessary* condition for revealing the falsity of the uniformity assumption.

Depending on the way or criterion used to select the few (n) test cases, the test-a-few strategy may take several variations. We now formulate three concrete strategies to implement the notion of test-a-few.

Strategy 1 (*Random sampling* at rate *r*). This strategy selects a *random sample* of n test cases from amongst the choice combinations involving C. The ratio $r = n/N$ is called the *sampling rate*.

The intuitive rationale of this strategy is obvious. Unless additional information is utilized, there is no preference among all of the N test cases involving C. Hence, this strategy simply selects a random sample of n test cases.

One implementation of this strategy is to fix the number n so that every particular condition will be tested with the same number of test cases. Alternatively, it seems more reasonable to preset the sampling rate r so that a particular condition C will be tested a number (n) of times in proportion to the number (N) of ways that C may be combined with other compatible choices. In other words, fixing the sampling rate $r = n/N$ satisfies the intuition that the more frequently C appears in the original test suite, the more often it should be tested.

The main benefit of this strategy lies in its simplicity and relatively low cost of implementation. Since the $n - 1$ additional test cases (per particular condition, compared with test-once) are selected at random, the cost of test case selection is minimized, and the extra cost basically consists of the overhead in executing the $n - 1$ test cases and checking their results.

Strategy 2 (*Choice coverage*). This strategy selects a sample of test cases involving C in such a way that C is *combined with every other* (*compatible*) *choice at least once*, whenever possible.

This strategy utilizes the information of other choices that combine with C in the original test suite. The rationale is that even though C, being a particular condition in nature, probably dominates the program's behaviour, other choices $|c|$ that combine with C may as well have some (albeit perhaps slight or unexpected) effect on the program's execution (which accounts for the risk of ignoring $|c|$ if C is tested just once). Since it is not known a priori which other choice $|c|$ may in fact have (unexpected) effect on the program behaviour (under the occurrence of the particular condition C), it seems safer to ensure all compatible choices are covered at least once by the tests.

Thus, choice coverage at least ensures the following. If C is actually processed differently by the implementation whenever another choice $|c|$ occurs in the test case, the program's non-uniform treatment of C due to the presence of $|c|$ will always be revealed. For example, in Figure 3, the particular condition |Trans Amount = \$0.00| is treated non-uniformly by the program, but adopting choice coverage (so that the choice |CLASSIC| is necessarily covered) will ensure that the fault in statement b40 be revealed. Clearly, random sampling with a sampling rate < 100% cannot guarantee that C will be combined with every other compatible choice at least once.

Strategy 3 (*Choice-occurrence sampling* at rate *r*). This strategy selects a random sample of test cases involving C such that it is *combined with every other* (*compatible*) *choice at least a certain number or percentage of times*, whenever possible.

Choice-occurrence sampling may be regarded as a generalisation of both random sampling and choice coverage. It requires every choice to be tested together with *C* using at least a certain number of test cases or a certain percentage of the test cases related to *C*, whenever possible. Thus, it subsumes choice coverage (actually covering other compatible choices multiple times) while incorporating the notion of sampling rate. Generally, it demands significantly more test cases than choice coverage, but only slightly more than random sampling at the same sampling rate.

Again, unless the tester trivially adopts a sampling rate of 100%, random sampling cannot guarantee that *C* will be combined with other compatible choices in a way proportional to their occurrence frequency, whereas choice-occurrence sampling can.

5. EVALUATION

In this section, we report an empirical evaluation of the effectiveness of the test-a-few strategies versus the test-once strategy. We first conducted a smaller experiment [10] to make observations on the performance of different strategies and gained experiences in dealing with human participants and other settings so that we can manage and control a later and larger experiment more efficiently. In what follows, we shall refer to the first experiment as Experiment 1, and the follow-up experiment as Experiment 2.

5.1. Experimental setup

5.1.1. Specifications. In our evaluation, we used two specifications of modules, both of which were adapted from commercial-like applications in the banking industry. We chose these two specifications because one of the authors (Poon) previously worked in this particular industry sector as a certified systems auditor. His expert domain knowledge helps to judge the realism of the two specifications and their particular conditions.

For Experiment 1, we used the module specification S_{PURCHASE}, which previously appeared in [4] for studying the problem of identifying categories and choices from specifications. S_{PURCHASE} concerns with the purchase of goods using credit cards issued by an international bank. The main functions of the system are to determine whether a purchase transaction using a credit card should be approved, and to calculate the number of bonus points that will result from an approved purchase. The number of bonus points further determines the type of benefit, such as shopping vouchers, that the customer is entitled.

Figure 5 shows part of the test specification derived from S_{PURCHASE} using CPM. Intuitively, when a transaction involves a purchase amount (PA) which is zero or negative, or which exceeds the remaining credit, the transaction is expected to be simply rejected without the need for further processing such as updating the credit balance or calculating the bonus points. Here, four particular conditions can be identified, namely, |CR-LO < 0.00|, |CR-OV < 0.00|, |PA < 0.00|, and |PA = 0.00|, where CR-LO and CR-OV denote the remaining credit (CR) after a purchase is made locally (LO) and at overseas (OV), respectively. We shall label these particular conditions as C_1, C_2, C_3, and C_4, respectively. Accordingly, the choices associated with these particular conditions are annotated with {**`single`**} as shown in Figure 5. The tester may reasonably expect the program to process these four particular conditions uniformly without regard to other inputs or conditions. Note that when a choice, such as |CR-LO < 0.00| or |CR-OV < 0.00| in Figure 5, is annotated with a selector expression together with {**`error`**} or {**`single`**}, the selector expression will be ignored at test suite generation.

```
[Purchase Location (PL)]
  |Local|         {property local}
  |Overseas|      {property overseas}

[Remaining Credit After Local Purchase (CR-LO)]
  |CR-LO < 0.00|  {if local and positive}{single}
  |CR-LO = 0.00|  {if local and positive}
  |CR-LO > 0.00|  {if local and positive}

[Remaining Credit After Overseas Purchase (CR-OV)]
  |CR-OV < 0.00|  {if overseas and positive}{single}
  |CR-OV = 0.00|  {if overseas and positive}
  |CR-OV > 0.00|  {if overseas and positive}

[Purchase Amount (PA)]
  |PA < 0.00|     {single}
  |PA = 0.00|     {single}
  |PA > 0.00|     {property positive}

  ⋮
```

Figure 5. Part of the test specification for S_{PURCHASE}.

For Experiment 2, we used the module specification S_{REWARDS} taken from [12], which is related to an application in the same banking domain as S_{PURCHASE}. S_{REWARDS} is, however, more complex in terms of the number of categories and choices identified from the specification, as well as the constraints that govern the combination of choices from different categories, resulting in a much larger potential test suite. This characteristic, together with the existence of more particular conditions, make S_{REWARDS} a suitable specification for our second experiment.

S_{REWARDS} is the specification for a system which determines the reward to be earned subsequent to a validated credit purchase transaction. To validate a credit purchase transaction, the system accepts the details of the transaction, including the card status (such as normal or expired), type (such as corporate or personal), credit limit, cumulative balance, and the class of ticket (such as first, business, or economy) when an air ticket is purchased.

5.1.2. Test sets. For Experiment 1, we used a tool called EXTRACT [16] to construct a test suite TS_{PURCHASE} from the test specification in Figure 5 but with the four {**single**} annotations ignored. This test suite consists of 1734 test cases. For each particular condition C, we form a corresponding test set by extracting every test case which involves C but not other particular conditions from TS_{PURCHASE}. We denote these four test sets for the particular conditions C_1, C_2, C_3, and C_4 by TS_1, TS_2, TS_3, and TS_4, respectively.

For Experiment 2, to construct the pools of test cases such that each test case in the same pool involves the same particular condition, we followed the same procedure as for Experiment 1, except that S_{REWARDS} was used instead of S_{PURCHASE}. Moreover, in Experiment 2, we label the 10 particular conditions as C_{11} to C_{20}, and the corresponding test sets as TS_{11} to TS_{20}, respectively.

Table I summarizes the number of test cases in the 14 test sets (one for each particular condition) in the two experiments and their descriptive statistics. The abbreviations ABTLC and ABTFC corresponding to C_{17} and C_{18} stand for "Available Balance on Transaction in Local Currency" and "Available Balance on Transaction in Foreign Currency", respectively.

Table I. Test sets involved in particular conditions.

Experiment	Test set	Associated particular condition		Number of test cases	Descriptive statistics
1	TS_1	|CR-LO < 0.00|	C_1	144	Mean: 216 Standard deviation: 83.1 Sum: 864
	TS_2	|CR-OV < 0.00|	C_2	144	
	TS_3	|PA < 0.00|	C_3	288	
	TS_4	|PA = 0.00|	C_4	288	
2	TS_{11}	|File Not Exists|	C_{11}	36	Mean: 587 Standard deviation: 699.1 Sum: 5868
	TS_{12}	|File Empty|	C_{12}	36	
	TS_{13}	|Card Not Found|	C_{13}	36	
	TS_{14}	|Card Lost|	C_{14}	1584	
	TS_{15}	|Card Expired|	C_{15}	1584	
	TS_{16}	|Card Suspended|	C_{16}	1584	
	TS_{17}	|ABTLC < 0|	C_{17}	144	
	TS_{18}	|ABTFC < 0|	C_{18}	144	
	TS_{19}	|Credit Purchase < 0|	C_{19}	288	
	TS_{20}	|Credit Purchase = 0|	C_{20}	432	

In Experiment 1, to evaluate the test-once strategy, four sample test sets of size one were selected from TS_{PURCHASE} such that the selected test cases meet the four particular conditions, respectively. Moreover, to evaluate the test-a-few strategies, for each particular condition C_i, four sample test sets were generated from TS_i by using each of the following four strategies, respectively: (a) choice coverage, (b) random sampling at a rate of 5%, (c) random sampling at a rate of 10%, and (d) choice-occurrence sampling at a rate of 10%. Thus, a total of 4 + (4 × 4) = 20 sample test sets were formed.

The procedure to select sample test sets from the test pools for Experiment 2 was the same as for Experiment 1, except that 10 particular conditions were involved instead of 4. Moreover, we further evaluated different strategies at a finer granularity of sampling rates. More specifically, in addition to test-once and choice coverage, we experimented with both random sampling and choice-occurrence sampling at sampling rates of 1%, 2%, 3%, 4%, 5%, and 10%, respectively. Thus, a total of 10 + 10 + (10 × 6) + (10 × 6) = 140 sample test sets were formed.

Table II summarizes the number of test cases in the resulting 20 sample test sets in Experiment 1 and 140 sample test sets in Experiment 2. We repeated either experiment 10 times so as to average out and minimize the effect of randomness during the test sampling process.

5.1.3. Procedure and programs. We conducted our experiments in a class-setting environment. For Experiment 1, we firstly conducted a three-hour workshop, in which we lectured on the principle and procedure of CPM, and the participants had to complete one hands-on exercise. The specification used in the exercise is irrelevant to the two specifications used in our study. All participants were computer-science undergraduates at their final year of study. At the time of the experiment, on average, the students had one year of relevant work experience in the computer software field. After completing the workshop, we distributed the program specification to the participants. Each participant was allowed to raise any question and request further clarifications. We refrained ourselves from offering

Table II. Size of sample test sets.

	Strategies for testing particular conditions	Number of test cases associated with				Descriptive statistics			
		C_1	C_2	C_3	C_4	Mean	Median	Sum	SD
Experiment 1	Test-once	1	1	1	1	1	1	4	0.0
	Choice coverage	6	6	6	6	6	6	24	0.0
	Random sampling (5%)	8	8	15	15	12	12	46	4.0
	Random sampling (10%)	15	15	29	29	22	22	88	8.1
	Choice-occurrence (10%)	18	18	31	31	25	25	98	7.5

	Strategies for testing particular conditions	Number of test cases associated with										Descriptive statistics			
		C_{11}	C_{12}	C_{13}	C_{14}	C_{15}	C_{16}	C_{17}	C_{18}	C_{19}	C_{20}	Mean	Median	Sum	SD
Experiment 2	Test-once	1	1	1	1	1	1	1	1	1	1	1	1	10	0.0
	Choice coverage	6	6	6	6	6	6	6	6	6	6	6	6	60	0.0
	Random sampling (1%)	1	1	1	16	16	16	2	2	3	5	6	3	63	6.8
	Random sampling (2%)	1	1	1	32	32	32	3	3	6	9	12	5	120	14.0
	Random sampling (3%)	2	2	2	48	48	48	5	5	9	13	18	7	182	20.8
	Random sampling (4%)	2	2	2	64	64	64	6	6	12	18	24	9	240	28.0
	Random sampling (5%)	2	2	2	80	80	80	8	8	15	22	30	12	299	35.1
	Random sampling (10%)	2	2	2	159	159	159	15	15	29	44	59	22	586	70.5
	Choice-occurrence (1%)	6	6	6	18	18	18	6	6	6	7	10	6	97	5.7
	Choice-occurrence (2%)	6	6	6	36	36	36	6	6	8	12	16	7	158	14.1
	Choice-occurrence (3%)	6	6	6	49	49	49	7	7	12	18	21	10	209	19.7
	Choice-occurrence (4%)	6	6	6	66	66	66	8	8	13	19	26	11	264	27.6
	Choice-occurrence (5%)	6	6	6	84	84	84	12	12	18	24	34	15	336	35.2
	Choice-occurrence (10%)	6	6	6	162	162	162	18	18	31	48	62	25	619	70.3

any advice or hints related to the uniformity assumption, or how a program may implement a requirement in the program specification. Participants were given four weeks to develop, individually and on their own, an implementation of the program specification. At the end, we collected 161 programs written for S_{PURCHASE}. We applied all the test cases in TS_{PURCHASE} to each collected program, and eliminated those programs that failed at least 20% of test cases or executed the same paths for all 1734 test cases in TS_{PURCHASE}. This threshold was also used in previous test experiments (such as [17]) that evaluated program-based testing techniques. The main reasons are to eliminate faulty programs that obviously did not reasonably implement the required function and, hence, violate the competent-programmer hypothesis [18], which hypothesizes that the programs written by a competent programmer are close to being correct. As a result, 31 programs were eliminated. The remaining 130 programs were then instrumented and executed with all the 864 test cases in TS_1, TS_2, TS_3, and TS_4. Important information such as the execution path associated with each test case and the faulty behaviour that each test case revealed (if any) were monitored and recorded. Faults were found in 87 programs by these test sets and no fault was found in the other 43 programs. Experiment 2 was done in the same way as Experiment 1, except that it used 5868 test cases in TS_{11} to TS_{20}. Faults were found in 80 programs by these test sets, and no fault was found in the other 50 programs.

5.1.4. Test oracles. In each experiment, we used our own implementation as a test oracle. Note that all the faults in the programs were real faults committed by the human subjects. We had no ideas what the faults were and where they were located in the programs. In view of the large number of programs and our limited human resources, we did not manually inspect each program in detail. However, we are certain that each program which exhibited failures on execution of the corresponding test pools contains at least one fault.

5.1.5. Research questions and hypotheses. The objectives of our experiments were twofold. First, as explained in previous sections, the rationale for the adequacy of the test-once strategy rests mainly on the uniformity assumption regarding the particular conditions. Thus, we would like to evaluate to what extent the assumption is true in our collected programs. Second, since the ultimate goal of defect testing is to detect program faults, we would like to know the effectiveness of the test-once strategy in detecting faults, particularly when compared with our proposed alternative test-a-few strategies. We now refine our objectives into specific research questions, expressed in hypotheses for statistical tests.

The uniformity assumption stipulates that the implementation should treat the particular condition uniformly, regardless of other inputs or conditions. In the present study, we use the operational definition which considers the uniformity assumption for a given program and a particular condition C_i to be satisfied if all the test cases in TS_i execute the same program path. Indeed, executing the same program path implies that the inputs are processed by the same set of program statements in the same sequence and, hence, may serve as a good indicator of being treated uniformly by the program. Conversely, inputs that traverse different program paths are either processed by different sets of program statements, or in a different sequence and, hence, treated non-uniformly by the program. Using program paths as the criterion to assess the uniformity assumption is conservative because some different program paths (for example, different program interleaving in multi-threaded programs with deterministic outputs) may be semantically the same but different in their ordering of the executed statements. Although using other criteria (for example, executing the same set of program statements) to assess the uniformity assumption is possible, the path executed provides an objective criterion to determine whether different inputs have been processed in the same manner by the program. Therefore, we chose to use paths as the basis in the experiment.

Thus, our first objective is to find out whether a program is *likely* to follow the same path for the same particular condition. To this end, we apply the Pareto Principle (or popularly known as the 80-20 rule) [19] as a benchmark to examine the validity of the uniformity assumption. Moreover, among all the programs (which implement the same specification) under study, we are particularly concerned with the behaviour of faulty programs, because they are exactly what testing is trying to reveal.

Hence, our first research question,

Q1: *Given a specification, how likely will a program that implements the specification indeed treat the particular conditions uniformly?*

can be formulated as the following five closely related null hypotheses:

H_1: For each particular condition, at least 80% of the programs execute the same path.

H_2: The proportion of programs that execute the same path is the same for different particular conditions.

H_3: For each particular condition, at least 80% of the *faulty* programs execute the same path.

H_4: The proportion of *faulty* programs that execute the same path is the same for different particular conditions.

H_5: To implement a particular condition, programs that execute the same path are equally likely to result in failures as programs that execute different paths.

The second objective of this study concerns with the detection of program faults. Ideally, any test case selection methods should aim at retaining, as much as possible, the fault-detecting ability of the test-all strategy (which executes the entire test suite). Hence, the second criterion is to evaluate the capability of the test-once and the proposed test-a-few strategies in detecting faulty programs. Thus, we ask the following question:

Q2: *Is there any significant difference in fault detection effectiveness between the test-once and test-a-few strategies?*

which translates to the following two null hypotheses:

H_6: The test-once and test-a-few strategies are similar in fault detection effectiveness.

H_7: Random sampling and choice-occurrence sampling at the same sampling rate are similar in fault detection effectiveness.

As explained in Section 5.1.4, we had no idea what the faults in the programs are or where they are, and so collecting the more conventional measure of the number or percentage of faults detected would be infeasible. To study these hypotheses, we shall measure the fault detection effectiveness of a strategy in terms of the percentage of test sets (generated by the strategy) that can detect all faulty programs over the total number of test sets for the strategy used in the experiments. Our set of faulty programs represents a real sample of possible faulty implementations (of the same specification) with genuine (instead of artificially seeded) faults committed by the participants. From the tester's point of view, the program under test could be any one of these implementations, which are all detected by the test-all strategy. The question is to what extent (or with what probability) the test-once or test-a-few strategies can achieve the same effectiveness as the test-all strategy.

5.2. Findings and discussions

5.2.1. Uniformity assumption for all programs.

Experiment 1: To investigate to what extent the uniformity assumption holds for the subject programs under study, we tabulate the statistics of the execution paths of all the test cases in TS_1, TS_2, TS_3, and TS_4 (see Table III).

Consider the particular condition C_1. There are 130 programs under study. Table III shows that, among them, the uniformity assumption holds in 110 (84.6%) of the programs, of which all test cases in TS_1 execute the same path. The assumption fails to hold in the other 20 programs, which constitute 15.4% of the batch of 130 programs. For the other particular conditions C_2, C_3, and C_4, the uniformity assumption holds in 97 (74.6%), 88 (67.7%), and 45 (34.6%) of the programs, and fails to hold in 33 (25.4%), 42 (32.3%), and 85 (65.4%), respectively, of the programs. Averaged over all the four particular conditions, the assumption fails to hold in 34.6% (that is, about 1 in 3) of the programs.

Another important observation is that for none of the particular conditions does the uniformity assumption hold in all of the 130 programs.

Overall, while the uniformity assumption may hold more often than not, the possibility (and hence the risk) of its failure to hold is not very low and therefore should not be lightly discarded. These results support our argument that the test-once strategy, which is based on the uniformity assumption, should be used judiciously. We further validate this result in Experiment 2.

Table III. Program execution paths with respect to different particular conditions.

	Particular condition and test set			Number (%) of programs executing	
				Same path (A)	Different paths (B)
Experiment 1	C_1	TS_1	|CR-LO < 0.00|	110 (84.6%)	20 (15.4%)
	C_2	TS_2	|CR-OV < 0.00|	97 (74.6%)	33 (25.4%)
	C_3	TS_3	|PA < 0.00|	88 (67.7%)	42 (32.3%)
	C_4	TS_4	|PA = 0.00|	45 (34.6%)	85 (65.4%)
			Mean	**85.0 (65.4%)**	**45.0 (34.6%)**
			Standard deviation	**28.2**	**28.2**
Experiment 2	C_{11}	TS_{11}	|File Not Exists|	126 (96.9%)	4 (3.1%)
	C_{12}	TS_{12}	|File Empty|	126 (96.9%)	4 (3.1%)
	C_{13}	TS_{13}	|Card Not Found|	126 (96.9%)	4 (3.1%)
	C_{14}	TS_{14}	|Card Lost|	121 (93.1%)	9 (6.9%)
	C_{15}	TS_{15}	|Card Expired|	119 (91.5%)	11 (8.5%)
	C_{16}	TS_{16}	|Card Suspended|	119 (91.5%)	11 (8.5%)
	C_{17}	TS_{17}	|ABTLC < 0|	111 (85.4%)	19 (14.6%)
	C_{18}	TS_{18}	|ABTFC < 0|	98 (75.4%)	32 (24.6%)
	C_{19}	TS_{19}	|Credit Purchase < 0|	88 (67.7%)	42 (32.3%)
	C_{20}	TS_{20}	|Credit Purchase = 0|	43 (31.1%)	87 (66.9%)
			Mean	**107.7 (82.8%)**	**22.3 (17.2%)**
			Standard deviation	**26.1**	**26.1**

Experiment 2: The result of this experiment is also shown in Table III. Compared to Experiment 1, we find that the chances of executing the same path are higher, and the result of Experiment 2 essentially echoes that of Experiment 1, that is, the uniformity assumption holds more often than not. This experiment shows that, overall, in terms of the mean value, the uniformity assumption holds in more than 80% of cases. On closer look, we find that the uniformity assumption almost always holds for exception handling conditions such as C_{11}, C_{12}, and C_{13}. Industrial-strength programs have many codes that comprehensively handle various exceptions. Our result indicates that the chance of uniform treatment by code to these parts of requirements is higher than that involving special functionality part of the requirements. However, for each particular condition, there is still a non-negligible proportion of programs executing different paths. The result indicates that methods to enhance the strategies for testing particular conditions are needed. We also observe that there are quite large differences among the percentages in column (A) of Table III, ranging from 31.1% to 96.9%. It may indicate that the mean value can be skewed by some values in the columns. To further analyze the data, we conduct hypothesis testing to review the result.

Hypothesis H_1: To validate H_1, we performed the Mann-Whitney test to compare whether a program has at least 80% of chance (in terms of median) that it implements a particular condition via the same path. Specifically, if a percentage value in column (A) of Table III is at least 80%, we mark the corresponding row as 1, otherwise 0. We compare this sequence of 0s and 1s with a sequence of all 1s, and test whether statistically the two sequences can be considered the same. Note that column (A) of Table III shows that 8 out of 14 rows contain values of at least 80%. Statistical calculation shows that, for size = 14 for either sequence, $z = -1.91$, $U = 140$, and p-value = 0.0281. It successfully rejects H_1 at a 5% significance level.

Hypothesis H_2: To validate H_2, we compute the ratio of the values in column (A) to the sum of values in columns (A) and (B) from Table III, and compare the set of all such ratios against the set of their mean values. Statistical calculation shows that, for size = 14 for either set of samples, z = 0.62, U = 84, and p-value = 0.2676. We cannot reject H_2 based on the data at a 5% significance level.

5.2.2. Uniformity assumption for faulty programs.

Experiment 1: The upper part of Table IV summarizes the execution path statistics of the programs that were revealed to be faulty by the test sets TS_1, TS_2, TS_3, and TS_4, which correspond to the particular conditions C_1, C_2, C_3, and C_4, respectively. Consider the row containing C_1 in Table IV. The test set TS_1 reveals only one faulty program. For this faulty program, since all test cases in TS_1 execute the same path, the uniformity assumption for C_1 does hold. In contrast, for the particular conditions C_2, C_3, and C_4, the uniformity assumption rarely holds in the faulty programs. The results suggest that, for programs that are faulty, it is more likely for the uniformity assumption to be violated than satisfied. This is in sharp contrast to the corresponding result in Section 5.2.1, where the uniformity assumption is seen to hold more often than not when all programs (including both correct and faulty ones) are considered. Clearly, faulty programs exhibit very different characteristics with respect to the uniformity assumption. It is likely that when conceiving the uniformity assumption, the tester is biased towards the specification, that is, biased towards the characteristics of *correct* programs instead of *incorrect* programs which the testing attempts to reveal.

Table IV. Faulty programs' execution paths with respect to different particular conditions.

	Particular condition and test set		Number of *faulty* programs revealed	Number (%) of *faulty* programs executing	
				Same path (C)	Different paths (D)
Experiment 1	C_1	TS_1	1	1 (100.0%)	0 (0.0%)
	C_2	TS_2	14	0 (0.0%)	14 (100.0%)
	C_3	TS_3	7	1 (14.3%)	6 (85.7%)
	C_4	TS_4	68	2 (2.9%)	66 (97.1%)
			Mean	**29.3%**	**70.7%**
			Standard deviation	**47.5%**	**47.5%**
Experiment 2	C_{11}	TS_{11}	7	4 (57.1%)	3 (42.9%)
	C_{12}	TS_{12}	7	4 (57.1%)	3 (42.9%)
	C_{13}	TS_{13}	7	4 (57.1%)	3 (42.9%)
	C_{14}	TS_{14}	4	0 (0.0%)	4 (100.0%)
	C_{15}	TS_{15}	6	0 (0.0%)	6 (100.0%)
	C_{16}	TS_{16}	6	0 (0.0%)	6 (100.0%)
	C_{17}	TS_{17}	1	1 (100.0%)	0 (0.0%)
	C_{18}	TS_{18}	14	0 (0.0%)	14 (100.0%)
	C_{19}	TS_{19}	7	1 (14.3%)	6 (85.7%)
	C_{20}	TS_{20}	70	2 (2.9%)	68 (97.1%)
			Mean	**28.9%**	**71.1%**
			Standard deviation	**36%**	**36%**

Experiment 2: The lower part of Table IV shows the result of Experiment 2. We observe that the data exhibit a very similar pattern as that of Experiment 1. Moreover, we observe that, in both experiments, the average percentages of faulty programs executing the same path and different paths are 29% and 71%, respectively. We conclude that, for programs that are faulty, it is indeed more likely for the uniformity assumption to be violated than satisfied. Thus, relying on the test-once strategy can be very risky when the program is faulty.

Hypothesis *H_3*: To validate H_3, we repeat the same procedure that validates H_1 except that we use the data in Table IV instead of those in Table III, column (C) instead of column (A), and column (D) instead of column (B). Statistical calculation shows that for size = 14 for either set of samples, we have z = –4.16, U = 189, and p-value < 0.0001. It successfully rejects H_3 at a 5% significance level.

Hypothesis *H_4*: By the same token, to validate H_4, we repeat the same procedure that validates H_2 except that we use the data in Table IV instead of those in Table III, column (C) instead of column (A), and column (D) instead of column (B). Statistical calculation shows that for size = 14 for either set of samples, we have z = –3.84, U = 182, and p-value < 0.0001. It successfully rejects H_4 at a 5% significance level.

Hypothesis *H_5*: To validate H_5, for each corresponding row in Tables III and IV, we compute the ratios of the values in columns (C) and (A), and those in columns (D) and (B). We then compare these two sequences of ratios by the Mann-Whitney test. Statistical calculation shows that for size = 14 for each set of samples, we have z = –3.42, U = 173, and p-value = 0.0003. It successfully rejects H_5 at a 5% significance level.

Table V summarizes the Mann-Whitney test results for the hypotheses H_1 to H_5.

5.2.3. Effectiveness in revealing faulty programs. We evaluate the fault-detecting ability of the test sets selected by the different strategies for testing particular conditions. Table VI shows, for each testing strategy, the number of test sets that reveal all the faulty programs (that is, detect the presence of at least one fault in each program).

Experiment 1: We have several observations here. First, the effectiveness of the test-once strategy varies widely. At best, it can be as effective as all the test-a-few strategies under study in that it always reveals all faulty programs (see, for example, the column for C_1 in Table VI). On the other hand, the test-once strategy may also never reveal all faulty programs (see, for example, the column for C_3 in Table VI). In contrast, the effectiveness of all the test-a-few strategies is relatively stable. Second, choice-occurrence sampling using a sampling rate of 10% can be as effective as test-all for the testing of *every particular condition*. Third,

Table V. Mann-Whitney test result summary for *H_1* to *H_5*.

Type of comparison	Hypothesis	Size of population	z	U	p-value	Rejected at a 5% significance level
Median	H_1	14	–1.91	140	0.0281	Yes
	H_2		0.62	84	0.2676	No
	H_3		–4.16	189	< 0.0001	Yes
	H_4		–3.84	182	< 0.0001	Yes
	H_5		–3.42	173	0.0003	Yes

Table VI. Number of test sets that reveal all faulty programs by different strategies.

	Strategies for testing particular conditions	Number of sample test sets that each reveals all the faulty programs (out of 10 test sets)				Descriptive statistics			
		C_1	C_2	C_3	C_4	Mean	Median	Sum	SD
Experiment 1	Test-once strategy	10	1	0	4	3.8	3	15	4.5
	Choice coverage	10	6	10	10	9.0	10	36	2.0
	Random sampling (5%)	10	4	10	10	8.5	10	34	3.0
	Random sampling (10%)	10	6	10	10	9.0	10	36	2.0
	Choice-occurrence (10%)	10	10	10	10	10.0	10	40	0.0

	Strategies for testing particular conditions	Number of sample test sets that each reveals all the faulty programs (out of 10 test sets)										Descriptive statistics			
		C_{11}	C_{12}	C_{13}	C_{14}	C_{15}	C_{16}	C_{17}	C_{18}	C_{19}	C_{20}	Mean	Median	Sum	SD
Experiment 2	Test-once strategy	4	5	3	0	0	1	10	10	0	1	3.4	2	34	3.9
	Choice coverage	10	10	10	10	10	9	10	10	10	10	9.9	10	99	0.3
	Random sampling (1%)	4	5	3	10	10	9	10	10	7	7	7.5	8	75	2.7
	Random sampling (2%)	4	5	3	10	10	10	10	10	10	10	8.2	10	82	2.9
	Random sampling (3%)	8	8	5	10	10	10	10	10	10	10	9.1	10	91	1.7
	Random sampling (4%)	8	8	5	10	10	10	10	10	10	10	9.1	10	91	1.7
	Random sampling (5%)	8	8	5	10	10	10	10	10	10	10	9.1	10	91	1.7
	Random sampling (10%)	9	9	10	10	10	10	10	10	10	10	9.8	10	98	0.4
	Choice-occurrence (1%)	10	10	10	10	10	10	10	10	10	10	10.0	10	100	0.0
	Choice-occurrence (2%)	10	10	10	10	10	10	10	10	10	10	10.0	10	100	0.0
	Choice-occurrence (3%)	10	10	10	10	10	10	10	10	10	10	10.0	10	100	0.0
	Choice-occurrence (4%)	10	10	10	10	10	10	10	10	10	10	10.0	10	100	0.0
	Choice-occurrence (5%)	10	10	10	10	10	9	10	10	10	10	9.9	10	99	0.3
	Choice-occurrence (10%)	10	10	10	10	10	10	10	10	10	10	10.0	10	100	0.0

random sampling using a sampling rate of 10% can be as effective as choice coverage, and both are much better than test-once. These results clearly suggest that, without the knowledge of the program code, it is generally safer to use a test-a-few strategy than test-once to reduce the risk of inadequate testing. Also, in many instances, a test-a-few strategy can be (almost) as effective as the test-all strategy. Since the sampling rate can substantially affect the effectiveness of the random sampling and choice-occurrence sampling strategies, in the follow-up experiment, we study whether the two strategies will still be effective with lower sampling rates.

Experiment 2: We observe encouraging results from this experiment. First, it confirms the result of Experiment 1 that test-once is quite unreliable in terms of its ability to detect faults. Its effectiveness ranges from always revealing all faulty programs (for C_{17} and C_{18}) to never (for C_{14}, C_{15}, and C_{19}). Second, choice-occurrence sampling can be effective even at a relatively low sampling rate (say, 1%). Indeed, choice-occurrence sampling can always reveal all faulty programs, with the only exception at 5% sampling rate for C_{16}. The same is true for choice coverage except for C_{16}. In comparison, only 1 out of the 10 test sets generated by the test-once strategy can reveal all faulty programs for C_{16} and C_{20}. Third, random sampling is inferior to choice-occurrence sampling, as the former has a lower or equal chance to reveal all faulty programs than choice-occurrence sampling at the same sampling rates.

Hypotheses H_6 and H_7: We also conducted a multiple comparison on the mean number of sample test sets using Matlab with Tukey's honestly significance difference procedure at a 5% significance level. The result is depicted in Figure 6.

Based on Figure 6, we find that the bars for choice coverage, choice-occurrence sampling, and random sampling (at all sampling rates under study) do not overlap with the bars for test-once. The result indicates that each of the test-a-few strategies is better (in terms of mean fault detection effectiveness) than the test-once strategy at a 5% significance level. Therefore, we can reject H_6 at a 5% significance level. In contrast, we do find that the bars for choice coverage, choice-occurrence sampling, and random sampling overlap in all the graphs in Figure 6. Therefore, we cannot reject H_7 based on their mean comparisons.

On the other hand, based on Figure 6, we observe that random sampling and choice-occurrence sampling at the same rate do have some observable difference when the rate is at 1%, 2%, 3%, 4%, and 5%, respectively. We therefore further analyse these two strategies by using the Mann-Whitney test separately to compare their median values. We find that at sampling rate of 1%, there is significant difference in terms of median fault detection effectiveness at a 5% significance level. The result summary of the two hypotheses H_6 and H_7 is shown in Table VII. In this table, we use the testing result of the analysis of variance (ANOVA) F test to identify the p-values for H_6.

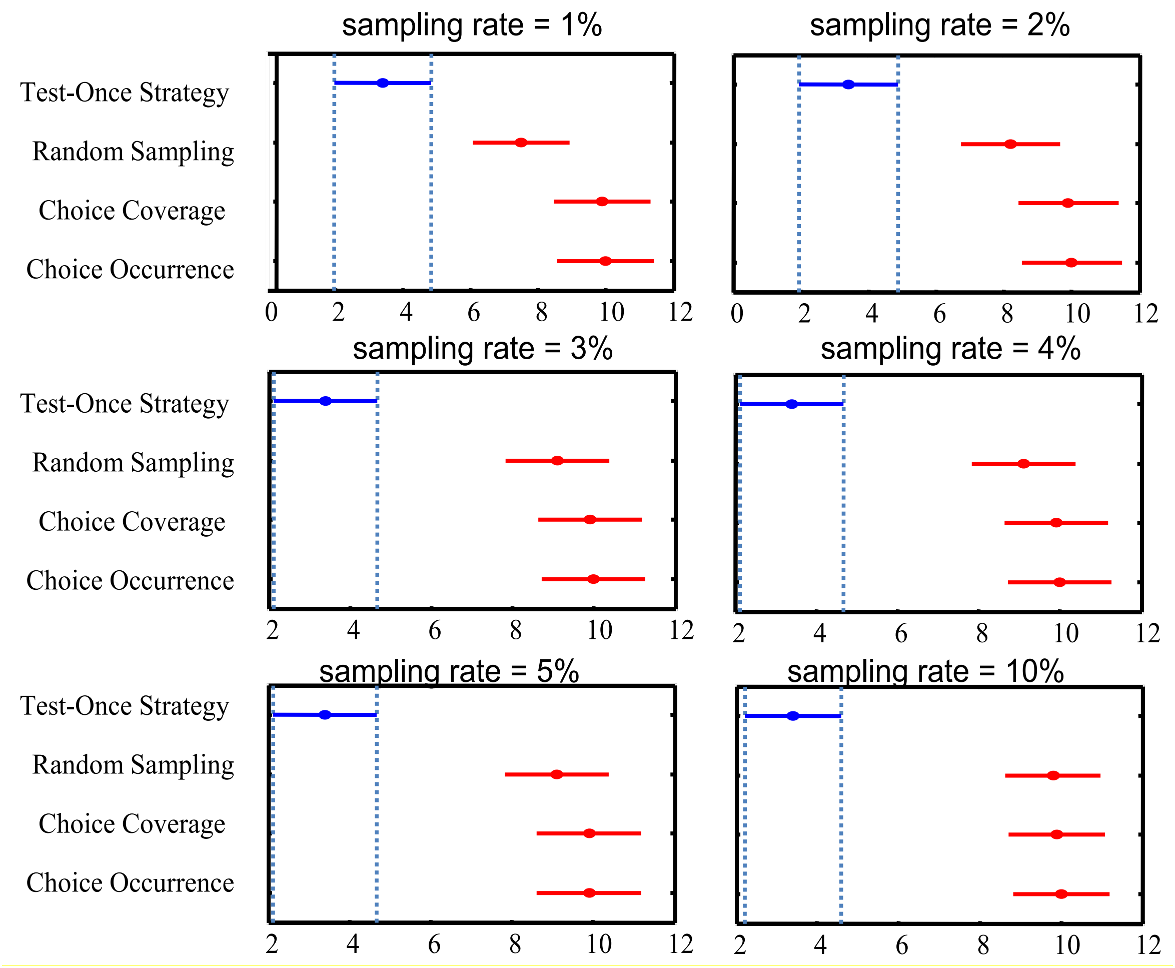


Figure 6. Multiple mean comparisons.

Table VII. Summary of results for H_6 and H_7.

Applied test	Type of comparison	Hypo-thesis	Size of population	Sampling rate	z	U	p-value	Rejected at a 5% significance level
Tukey's honestly signficance difference	Mean	H_6	10	1%	–		5.2002×10^{-7}	Yes
				2%			8.6661×10^{-7}	Yes
				3%			2.6617×10^{-8}	Yes
				4%			2.6617×10^{-8}	Yes
				5%			3.2744×10^{-8}	Yes
				10%			2.1024×10^{-9}	Yes
Mann-Whiney U test	Median	H_7	10	1%	−2.23	80	0.0129	Yes
				2–5%	−1.10	65	0.1357	No
				10%	−0.72	60	0.2358	No

Summary: We summarize our findings of the fault-detecting ability among the proposed test-a-few strategies with reference to Table VI and Figure 6. Let us focus on Experiment 2. In general, both the choice coverage and choice-occurrence sampling strategies (with all sampling rates under study) almost always reveal all faulty programs (see Table VI). Furthermore, both choice coverage and choice-occurrence sampling with a sampling rate of 5% have a mean fault detection rate of 99% (9.9 out of 10 test sets) and a standard deviation of 0.3. Choice-occurrence sampling with other sampling rates always reveals all the faulty programs (10 out of 10 test sets). On the other hand, only random sampling with a sampling rate of 10% can achieve the nearest result (with standard deviation 0.4). The mean fault detection rates of random sampling drop as the sampling rates decrease. Furthermore, when the sampling rate decreases to 1%, random sampling can only reveal all faulty programs 75% of the time on average. Thus, our result shows that the mean fault detection effectiveness of choice coverage and choice-occurrence sampling are better than random sampling when applying the test-a-few strategy to test the particular conditions. If we consider both Experiments 1 and 2, we observe that the mean fault detection rates of the test-once strategy are 3.8 and 3.4 out of 10 test sets, respectively, which are much lower than those of choice coverage, random sampling, and choice-occurrence sampling. Hence, the observation suggests that the test-a-few strategies are all better than the test-once strategy with respect to the faulty program detection capability.

We now consider the fault detection effectiveness together with the test set size of different testing strategies (see Table II). On average, choice coverage requires fewer test cases (six for each particular condition) when compared with other test-a-few strategies. Although random sampling at a sampling rate of 1% requires the same mean test set size, its fault detection effectiveness is the worst when compared with other test-a-few strategies (see the results of Experiment 2 in Table VI). Choice-occurrence sampling at sampling rates of 1% to 10% achieves better fault detection effectiveness, but their larger test set sizes (from 97 to 619) result in higher testing costs. The test-once strategy obviously uses the fewest test cases (one for each particular condition). Nevertheless, choice coverage only requires less than 3% (24 out of 864) and about 1% (60 out of 5868) of all possible test cases for testing all particular conditions in Experiments 1 and 2, respectively. In short, there is always a tradeoff between testing cost and test quality. The optimal tradeoff depends on many factors, but obviously if testing cost is not the sole concern, test-once is not necessarily optimal.

5.2.4. Strengths and limitations of different strategies. We summarize the strengths and limitations of different strategies based on our experimental results as follows.

Test-once strategy: This strategy requires the smallest number of test cases and, hence, minimum testing cost and effort. As shown in Tables I and II, it only requires 4 out of 864 possible test cases and 10 out of 5868 possible test cases to test the particular conditions in Experiments 1 and 2, respectively. This strategy, however, cannot effectively reveal the faulty programs corresponding to the particular conditions, especially when the uniformity assumption does not hold. For example, as shown in Table VI, in all test runs for the particular conditions C_3, C_{14}, C_{15} and C_{19}, this strategy fails to reveal all the faulty programs.

Random sampling: This strategy is efficient with respect to the testing cost among all the test-a-few strategies under study. It simply picks at random a certain number of test cases involving the particular conditions. It is substantially more effective than test-once. Despite this advantage, the strategy may be less effective in detecting faults compared with other test-a-few strategies. For instance, as shown in Table VI, not all the faulty programs associated with the particular conditions C_2, C_{11}, and C_{12} were revealed by random sampling.

Choice coverage: This strategy usually uses the smallest test set among the test-a-few strategies under study. As shown in Tables I and II, it requires 24 out of 864 possible test cases and 60 out of 5868 possible test cases to test the particular conditions in Experiments 1 and 2, respectively. In the experiments, this strategy is capable of revealing all the faulty programs in almost all of the test runs. Table VI shows that the strategy revealed all the faulty programs associated with 12 out of 14 particular conditions in all test runs. Compared to the test-once strategy, although choice coverage requires five more test cases for each particular condition in our experiments, the selected test set still constitutes only a very small portion of the entire test suite. Nevertheless, choice coverage is not always as reliable as choice-occurrence sampling (which requires even more test cases), as shown in the test results for C_2.

Choice-occurrence sampling: This is the most effective test-a-few strategy among all strategies under study in terms of the ability to detect faulty programs. It reveals all the faulty programs in all test runs except one test run for C_{16} in Experiment 2 with a sampling rate of 5%. The strategy, however, involves more test cases (and, hence, more testing resources) when compared to the test-once strategy and other test-a-few strategies. Hence, it can be relatively inefficient. For example, choice-occurrence sampling at the rate of 1% involved 97 test cases for testing the 10 particular conditions C_{11} to C_{20}, which is substantially more than those required for the test-once strategy (10), choice coverage (60), and random sampling at the same rate (63).

5.3 Threats to validity

Threats to internal validity. To increase the confidence and accuracy of data collection, we implemented a tool to assist in the test case generation and execution of the experiments. A threat to internal validity is the correctness of the tool. We have invited a non-author researcher who has over 10 years of software development experience to independently validate the results generated by the tool. We did not observe anomaly from this validation exercise. To collect the path profiles of individual test cases, we used TCAT C/C++ (see http://www.soft.com/TestWorks/Products/ Coverage/tcat.html) to instrument and record the execution paths of the subject programs. Owing to our limited resources, we have only sampled some paths profiled by the tools for manual verification.

Another threat is about the realism of the subject programs. Each participant was given four weeks to implement the programs in the C programming language individually. Their levels of programming skills naturally vary. However, they have learnt C programming and problem solving prior to the experiments. We have ensured that the programs can be developed based on such learnt skills rather than requiring some tricks and particular know-how of the application domain. In addition, students can receive course lecturer's and tutors' support when they encountered unclear parts of the specification or difficulty in some general programming techniques. The lecturer and tutors attempted to answer all of their questions without telling them the solutions. We have also tried to implement such a program ourselves, and concluded that it could not be completed within a short period. Another concern is about plagiarism. We have manually compared the program codes and found no plagiarized cases.

Previously, we have developed an experimental prototype tool, called EXTRACT, to support specification-based testing [16]. Although EXTRACT was designed to enhance test case generation using CTM, it can be adapted to handle test specifications written in TSL. Additionally, it generalizes the functionalities of a generator tool for CPM. In particular, it allows the tester to annotate a choice with a tag {`OCCUR` *n*} to specify the automatic extraction of at least *n* test cases that involve the annotated choice. In our experiments, we used this tool for constructing the test sets and implementing the random sampling strategy.

In the experiments, we had used the specifications in the published work [4, 12]. To ensure that the sets of particular conditions in both experiments are valid and all possible particular conditions have been identified, the first author (Chan) of the paper firstly identified the sets, and then two other authors (Poon and Yu) and a non-author researcher (who has commercial and research experiences in specification-based testing) were invited to review the validity of these sets independently. Indeed, our results also confirm that many programs did execute each particular condition with the same path. That is, some input conditions we identified do possess the characteristics of particular conditions, in the sense that the uniformity assumption is usually (though not always) satisfied.

Threats to external validity. The specifications we used in the experiments were taken from our prior experience and knowledge about banking systems. Obviously, there are other types of application systems in practice. However, banking systems such as purchase-transaction processing with credit cards have been popularly included in textbooks as illustrations for teaching students about software engineering principles. We believe that they do capture some generic aspects that the software engineering community treasures. Moreover, particular conditions such as special scenarios or exception situations can be found in diverse kinds of systems. Thus, the use of our specifications and the corresponding particular conditions should not be atypical.

Threats to construct validity. Threats to construct validity occur when the metrics used do not precisely represent the intentions to be justified. In our study, we test the uniformity assumption of a subject program by measuring whether the test cases with respect to the particular conditions execute the same program path. Indeed, executing the same program path implies that the inputs are processed by the same set of program statements in the same sequence and, hence, can reasonably be regarded as being treated uniformly by the program. In principle, it is possible that using other metrics will produce different results, and this may be addressed in future experiments. Besides, we have measured fault detection effectiveness by considering whether a test suite can detect all faulty programs. This is a conservative measure which enables us to check whether the test suites are reliable in their detection of faults in all the implementations. Other measures, such as the percentage of faults detected, can be used in further work, but may demand more effort.

6. DISCUSSION

We now further elaborate the rationale of our work, and position it in the broader context of published work. This paper focuses specifically on the testing of particular input conditions, and is not concerned with "normal scenarios" or "main functionalities" (handled by code that implements the so-called *function rules* in [4]) of the software under test. Although testing the "normal scenarios" is certainly no less important, particular input conditions are special in that, for many of them, there is often justifiable professional judgement or intuitive ground for a tester to believe in the uniformity assumption. When such a belief holds, the tester may choose to apply the test-once strategy, hoping that there will be no loss of test effectiveness, thereby saving a great deal of testing costs. This explains why the test-once strategy is so appealing to some testers. Indeed, our experiments also confirm that, with regard to the particular input conditions identified from the specifications, the uniformity assumption generally holds more often than not, when all the subject programs are considered (see Table III).

Appealing as it might be, the test-once strategy is not without caveat, and the associated risk can be higher than what one perceives, as demonstrated by Example 2 in Section 3 and our empirical findings in Section 5.2. For example, our finding presented in Section 5.2.2 shows that the program needs not be implemented in the same manner as conceived by the tester. Specifically, Table IV shows that the uniformity assumption actually does not hold in the majority of the faulty programs in our collection. On closer thought, a plausible explanation is as follows. The tester usually has in mind the specification information which provides the valid ground for considering the use of the test-once strategy. Yet it is not always as certain or obvious whether the uniformity assumption actually holds *with respect to the specific implementation under test*. For one reason, since faults in an incorrect program are usually unpredictable prior to testing, their effect on the program's potential "uniform treatment" of particular input conditions can also be uncertain. For another reason, our example illustrated in Figure 4, as well as our empirical finding in Section 5.2.1, also remind testers that such "unexpected" implementations are not necessarily confined to incorrect programs only. Therefore, any tester should seriously take into account the potential risk due to inadequacy of testing when using the test-once strategy. This brings back to the unavoidable tradeoff between the amount of resources expended and the test effectiveness achieved.

Our recommendation is that the test-once strategy should be used judiciously, and due consideration should be given to assessing the strength of belief in the uniformity assumption, choosing (a few) more test cases for a particular condition when the confidence is not strong. An analytical study of such a relationship is beyond the scope of this paper, but can be an interesting direction for further research (see our suggestion of further work in Section 8). From our experiments, test-a-few using random sampling is already found to be substantially more effective than test-once, as seen in Figure 6. Furthermore, it seems that choice coverage fares quite well when compared with both random sampling and the choice-occurrence sampling strategies, in the sense that the former demands (literally) just a few more test cases for each particular condition (see Table II) but can approach the effectiveness of the test-all strategy in most cases (see Figure 6). Whether this observation can be generalized into other settings remains to be verified by more experiments.

One potential criticism to the test-a-few strategy is that it ("unnecessarily") increases the ("already high") testing cost, defeating the very purpose of test-once. Our view is that whether test-a-few is "necessary" or even "affordable" is typically subject to tradeoff analysis, which depends on the specific application or context on hand. This work shows that, in our study setting, the test-a-few strategies are all potentially much more preferable to the test-once strategy, substantiated on both empirical and theoretical grounds. Although the generality of our conclusion is limited by the scope of this work, a clear message of practical implication is that the potential risk of the test-once strategy should not be lightly discarded by a tester, but rather, should at least be carefully evaluated by the tester with reference to the specific implementation under test.

7. RELATED WORK

This study is concerned with the selection of test cases involving particular input conditions from a test suite generated from a specification. Specification-based test case generation has long been an active area of software engineering research. As explained in Sections 1 and 2, a variety of systematic specification-based testing methods (such as CEG [14], combinatorial testing [8, 11, 15], CPM [4, 7], and CTM [5, 6]) have been developed and extensively studied. Although these methods are generally effective in generating a comprehensive test suite based on the test-relevant aspects of the specification, the resulting test suite tends to be very large and, if executed in full, could demand an unduly heavy amount of testing resources. In the literature, many different strategies and heuristics have been proposed to select a subset of the test suite for execution. We now review some of the related studies in this section.

Earlier, we have presented a preliminary empirical study [10] on the effectiveness of specification-based testing of particular conditions, and its results are further reported and statistically analysed as Experiment 1 in this paper. This paper extends our work in [10] by performing a follow-up experiment using an enhanced specification which involves a more complex set of test-relevant aspects as well as a much larger test suite. This paper has also introduced a more rigorous framework that permits further in-depth and refined analysis of the research questions in the form of statistical hypotheses testing based on the combined empirical findings, as well as a more elaborated discussion of the rationale and practical implication of this work in the preceding section.

Due to their importance, input validation and exception conditions, which are among the most common particular conditions, have been the subject of research by many authors [20, 21, 22, 23]. Liu and Tan [20], for instance, argue that input validation plays a key role in controlling user inputs and maintaining the robustness of software systems, especially for data-intensive applications. To improve the accuracy and adequacy of input validation testing, they proposed and empirically evaluated a code-based method to extract valid and invalid input conditions from the program code as a means to supplement the usual specification-based testing methods. On the other hand, Tracey et al. [21] contend that, due to the expected rare occurrence of exceptions, the exception handling code of a system is, in general, least documented and tested, potentially causing great hazard in safety-critical systems. In such cases, the test-once strategy would obviously be very risky. To facilitate the testing of exception handling code, Tracey et al. [21] have developed a technique for automatically generating test data for testing exception conditions. Where their technique is applicable and successful, the test-a-few strategies (notably random sampling) can potentially be implemented with little extra cost over the test-once strategy, since test data generation can be automated. Tracey et al. [21] also outline a process that combines proof with automated testing which will, hopefully, substantially reduce the costs of ensuring that the exception handling code is safe. Recently, Jiang et al. [22] have proposed a fault-injection approach for automated testing of exception handling mechanisms in code. Ngo et al. [23] have proposed a new approach to test one specific type of conditions, namely, the user-interactive undo (UI-undo) features in database applications.

The work of Liu and Tan [20], as well as that of Tracey et al. [21], attempt to automatically generate test data from code for testing certain types of particular conditions. Even though there are many advancements in recent years, automatic test data generation, however, is still notoriously difficult and may not always succeed even when source code is available. A different direction of research would be to perform automatic static analysis of the code. One potentially applicable technique is conditioned program slicing [13, 24, 25, 26], which seeks to identify statements and predicates that contribute to the computation of a selected set of variables when some chosen condition is satisfied. Another related technique constructs the path conditions of programs for safety analysis [27]. So far, the conditions in these studies are all restricted to elements of the program code, whereas the particular conditions referred to in this paper are identified from the specification. If these two kinds of conditions bear simple known relations, then these static code analysis techniques may be applied to validate the uniformity assumption or its falsity with respect to the particular conditions. This will provide an objective means of determining statically whether test-once is adequate or not.

In the literature, many heuristics have been developed to reduce the size of a test suite in the absence of code information. For example, as reviewed in Section 3, CPM [7] proposes the test-once strategy, and provides a notation in TSL to express the intention of testing certain particular input conditions only once without cautioning the associated risk. Grochtmann and Grimm [6] go even further to state the *minimality criterion* as mandatory, which requires each parameter value of every test-relevant aspect be tested at least once, whenever possible. In a sense, adopting the minimality criterion is equivalent to applying the test-once strategy to *all* test-relevant aspects, not just those involving particular conditions. Hence, the criterion is weak and suffers from even more limitations than the test-once strategy since, for "non-particular conditions", the intuitive or theoretical basis for the uniformity assumption is in doubt or perhaps even lacking. Nevertheless, the minimality criterion was meant to be a bottom line of testing requirements, not as a strategy by itself per se.

A test suite which includes all feasible combinations of test-relevant aspects can be huge due to the combinatorial explosion problem [1, 3, 12, 15]. To address this problem, some studies (such as [8]) have advocated the use of experimental design techniques [28] in software testing. Experimental design techniques were originally developed on the basis of well-known mathematical results for the purpose of planning statistical experiments, which attempt to minimize the number of experiments needed to extract the required statistical information. In the context of software testing, a statistical experiment corresponds to a test case, while a factor that potentially affects the outcome of a statistical experiment corresponds to a test-relevant aspect. One of such early attempts was made by Mandl [29], who applied an orthogonal array (OA) technique to the testing of Ada compilers. Essentially, in the language of CPM, OA design creates a test suite in such a way that every pair of choices from different categories occurs at least once. Moreover, because the OA technique originates from its use in statistical experimentation, it further has a balance requirement that every pair of factor values (corresponding to choices in CPM) must occur the same number of times.

There are several difficulties in applying OA-based techniques to software testing [8]. One major difficulty is that there exists no OA corresponding to certain number of combinations of categories and choices, such as when there are 6 categories each with 7 choices. Moreover, the balance requirement of an OA is not only unnecessary in the context of software testing, but actually sometimes makes it impossible to construct such a test suite when certain combination of choices are infeasible due to constraints inherent to the software application [8].

To relax the strict requirements of OA, many software engineering researchers study the criterion of *pairwise coverage* or, more generally, *n-way coverage* for a fixed small value of $n > 1$ [11, 15, 30, 31]. Essentially, n-way coverage testing requires that every *feasible n*-tuple of distinct parameter values (analogous to choices in CPM) of test-relevant aspects be covered by the test suite at least once, while pairwise coverage refers to the instance of n-way coverage when $n = 2$. Subsets of the test suite satisfying these coverage requirements can be generated by using heuristics such as the in-parameter-order (IPO) algorithms [30, 31].

There are some interesting relationships among (1) the minimality criterion, (2) pairwise coverage, (3) test-once, and (4) choice coverage, one of our proposed test-a-few strategies. First, if we consider applying pairwise coverage only to the subset of test cases in which a particular condition C is involved, then C will form pairs by combining with every other choice at least once and, hence, choice coverage involving C will be guaranteed. Conversely, choice coverage effectively requires the application of pairwise coverage to all test cases that involve particular conditions. Hence, pairwise coverage subsumes choice coverage. Finally, just as pairwise coverage (which requires *every pair of parameter values* to be covered at least once) subsumes the minimality criterion (which only requires *every parameter value* to appear at least once), analogously, we have choice coverage subsumes test-once.

Both OA-based techniques and the pairwise coverage criterion are theoretically appealing in the following sense: the resulting test suite is guaranteed to detect all systematic failures that arise due to the faulty interaction of any two choices. As a consequence, the choice coverage strategy proposed in this paper inherits a similar property, that is, the strategy will always detect all faults that arise due to the interaction of a particular condition with another choice.

On the other hand, our work can be distinguished from OA-based techniques and the pairwise coverage criterion as follows. First, our proposal is to use test-a-few strategies to reduce the risk due to potential violation of the uniformity assumption (on which the test-once strategy is founded). Choice coverage is but one of the test-a-few strategies proposed, and our experiment actually also shows the benefits of random sampling (which requires less testing resources) and choice-occurrence sampling (see Sections 5.2.3 and 5.2.4 for detailed comparison). For instance, when testing resources are very constrained and a large number of choices (factors) are involved, it may even be not affordable to run all the test cases required by choice coverage, OA, or pairwise coverage. In such situations, however, the test-a-few strategy of random sampling can still be applied by using a low sampling rate. In our experiments, using this strategy is still substantially better than using the test-once. Second, since our work in this paper is specifically focused on the testing of particular conditions, the testing of other categories or choices is beyond the present scope. Indeed, when particular conditions are not involved, the behaviour of the specific implementation under test is often more complex and may be affected by a larger number of test-relevant aspects. In such situations, a (specification-based) test suite constructed to satisfy the n-way coverage or OA-based criteria only may not be effective enough to detect the complex faults that are unrelated to particular conditions.

Finally, there is a variety of test suite reduction techniques proposed in the literature to address the excessive costs required by a large test suite. These techniques typically demand additional information that is not required in our present study. Examples include input-output analysis which makes use of the input-output relations (identified from the specification) for selection of combinations of input parameter values [3], and heuristics-based techniques (such as those based on variations of the greedy heuristics) which make use of the relationship between test cases and test objectives [2, 32, 33, 34]. These studies differ from our work in that they are not specifically concerned with particular conditions. Interested readers may consult the cited articles in detail.

8. CONCLUSION AND FURTHER WORK

Many testing methods consider various test-relevant aspects from the specification to generate a comprehensive test suite which practically cannot be fully tested due to resource constraints. To keep the number of test cases to an affordable size yet hopefully retaining the test adequacy of the test suite as much as possible, a popular pragmatic strategy is to identify some particular conditions from the specification, and then select test cases in such a way that each particular condition is tested only once. This strategy, which we call test-once, implicitly assumes that one test case is adequate to detect the presence of faults related to the particular condition in question. It largely depends on the validity of the uniformity assumption, which in turn often relies on the tester's intuition or judgment based primarily on the specification and, hence, is not guaranteed to hold for the specific implementation under test.

To reduce the risk of inadequate testing incurred due to possible violation to the uniformity assumption, this paper has introduced several test-a-few strategies and reported an empirical evaluation on the extent to which the uniformity assumption may hold, as well as the effectiveness of the proposed strategies in detecting faults.

Among the findings of our study, two are of vital importance. First, the programs used in the study do confirm our claim that the uniformity assumption *often*, *but not always*, holds. Indeed, the uniformity assumption is found to be violated in a significant proportion of implementations that cannot be simply neglected. Second, the fault-detecting ability of the test-once strategy varies widely. We have provided empirical evidence that the test-a-few strategies (each requiring only a small number of additional test cases) can make a difference not only in assessing the validity of the uniformity assumption, but also in improving the adequacy of the testing.

Further investigation of this line of research can proceed in several directions. First, the uniformity assumption may be *probabilistically* validated by means of dynamic analysis through program path profiling using code instrumentation tools. The idea is as follows. The program can be executed with a sample of n (> 1) test cases involving the concerned particular condition C, with the execution paths monitored. If all n test cases traverse the same program path, then the tester's confidence on the validity of the uniformity assumption will increase, more so for larger values of n. The value of n can be determined in such a way that the tester's confidence will be statistically significant. Conversely, if some of the n test cases execute different program paths, this serves to indicate that the uniformity assumption is likely to be in doubt (contrary to the tester's belief when judging only from the specification), and follow-up investigation should be in place, such as manual code analysis. Obviously, this idea of validating the uniformity assumption through program profiling does not make sense if $n = 1$, that is, if the test-once strategy is adopted. In other words, it must be implemented with a test-a-few strategy, be it random sampling, choice coverage, or choice-occurrence sampling. In so doing, as the testing proceeds, dynamic analysis captures the execution path information of the sample test cases as an additional basis for assessing the validity of the uniformity assumption [10].

Both the test-a-few strategy and the probabilistic validation of the uniformity assumption through dynamic analysis are feasible with the availability of instrumentation tools, usually incurring only a small amount of additional overhead to the overall testing costs. Besides, other more effective strategies can be devised. Further experiments should be performed to gain more experiences of how effective these proposed alternative strategies are.

Finally, automatic static analysis (such as conditioned slicing) of the source code, where applicable, is a potentially powerful technique to provide information on the execution behavior of the program under the particular conditions. This is also clearly a promising research direction to pursue.

ACKNOWLEDGEMENTS

The authors are grateful to F.F. Chan for his help in the empirical analysis and validation reported in this paper. A preliminary version of this paper has been presented at the 28th International Computer Software and Applications Conference [10].

REFERENCES

1. Chen TY, Poon P-L, Tse TH. A choice relation framework for supporting category-partition test case generation. *IEEE Transactions on Software Engineering* 2003; **29** (7): 577–593. DOI: 10.1109/TSE.2003.1214323.
2. Fraser G, Gargantini A, Wotawa F. On the order of test goals in specification-based testing. *Journal of Logic and Algebraic Programming* 2009; **78** (6): 472–490. DOI: 10.1016/j.jlap.2009.01.004.
3. Schroeder PJ, Korel B. Black-box test reduction using input-output analysis. *Proceedings of the International Conference on Software Testing and Analysis* (*ISSTA*) 2000; 173–177. DOI: 10.1145/347324.349042.
4. Chen TY, Poon P-L, Tang S-F, Tse TH. On the identification of categories and choices for specification-based test case generation. *Information and Software Technology* 2004; **46** (13): 887−898. DOI:10.1016/j.infsof.2004.03.005.
5. Chen TY, Poon P-L, Tse TH. An integrated classification-tree methodology for test case generation. *International Journal of Software Engineering and Knowledge Engineering* 2000; **10** (6): 647−679. DOI: 10.1142/S0218194000000353.
6. Grochtmann M, Grimm K. Classification trees for partition testing. *Software Testing, Verification and Reliability* 1993; **3** (2): 63−82. DOI: 10.1002/stvr.4370030203.
7. Ostrand TJ, Balcer MJ. The category-partition method for specifying and generating functional tests. *Communications of the ACM* 1988; **31** (6): 676–686. DOI: 10.1145/62959.62964.
8. Cohen DM, Dalal SR, Fredman ML, Patton GC. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering* 1997; **23** (7): 437–444. DOI: 10.1109/32.605761.
9. Jiang B, Tse TH, Grieskamp W, Kicillof N, Cao Y, Li X. Regression testing process improvement for specification evolution of real-world protocol software. *Proceedings of the 10th International Conference on Quality Software* (*QSIC*) 2010; 62–71. DOI: 10.1109/QSIC.2010.55.
10. Chan EYK, Poon P-L, Yu YT. On the testing of particular input conditions. *Proceedings of the 28th International Computer Software and Applications Conference* (*COMPSAC*) 2004; 318–325. DOI: 10.1109/CMPSAC.2004.1342850.
11. Kuhn DR, Kacker R, Lei Y. Automated combinatorial test methods − Beyond pairwise testing. *CrossTalk – The Journal of Defense Software Engineering* 2008; **21** (6): 22–26.
12. Yu YT, Tang S-F, Poon P-L, Chen TY. A study on a path-based strategy for selecting black-box generated test cases. *International Journal of Software Engineering and Knowledge Engineering* 2001; **11** (2): 113−138. DOI: 10.1142/S0218194001000475.
13. Hierons R, Harman M, Fox C, Ouarbya L, Daoudi M. Conditioned slicing supports partition testing. *Software Testing, Verification and Reliability* 2002; **12** (1): 23–28. DOI: 10.1002/stvr.232.
14. Myers GJ. *The Art of Software Testing*, 2nd edition, Wiley, ISBN 0471469122, 2004.
15. Lei Y, Carver RH, Kacker R, Kung D. A combinatorial testing strategy for concurrent programs. *Software Testing, Verification, and Reliability* 2007; **17** (2): 207–225. DOI: 10.1002/stvr.369.
16. Yu YT, Ng SP, Chan EYK. Generating, selecting and prioritizing test cases from specifications with tool support. *Proceedings of the 3rd International Conference on Quality Software* (*QSIC*) 2003; 83–90. DOI: 10.1109/QSIC.2003.1319089.

17. Elbaum S, Malishevsky AG, Rothermel G. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering* 2002; **28** (2): 159–182. DOI: 10.1109/32.988497.
18. DeMillo RA, Lipton RJ, Sayward FG. Hints on test data selection: Help for the practicing programmer. *IEEE Computer* 1978; **11** (4): 34–41. DOI: 10.1109/C-M.1978.218136.
19. Pareto V, Page AN. *Translation of Manuale di economia politica (*"*Manual of political economy*"*)*, A.M. Kelley, ISBN 9780678008812, 1972.
20. Liu H, Tan HBK. Covering code behaviour on input validation in functional testing. *Information and Software Technology* 2009; **51** (2): 546–553. DOI: 10.1016/j.infsof.2008.07.001.
21. Tracey N, Clark J, Mander K, McDermid J. Automated test-data generation for exception conditions. *Software — Practice and Experience* 2000; **30** (1): 61–79. DOI: 10.1002/(SICI)1097-024X(200001)30:1.
22. Jiang S, Zhang Y, Yan D, Jiang Y. An approach to automatic testing exception handling. *ACM SIGPLAN Notices* 2005; **40** (8): 34–39. DOI: 10.1145/1089851.1089858.
23. Ngo MN, Tan HBK. Automated verification and testing of user-interactive undo features in database applications. *Software Testing, Verification, and Reliability* (to appear). DOI: 10.1002/stvr.439.
24. Canfora G, Cimitile A, Lucia AD. Conditioned program slicing. *Information and Software Technology* 1998; **40** (11/12): 595–607. DOI: 10.1016/S0950-5849(98)00086-X.
25. Ward M, Zedan H. Slicing as a program transformation. *ACM Transactions on Programming Languages and Systems* 2007; **29** (2): Article 7. DOI: 10.1145/1216374.1216375.
26. Ward M, Zedan H, Ladkau M, Natelberg S. Conditioned semantic slicing for abstraction; industrial experiment. *Software — Practice and Experience* 2008; **38** (12): 1273–1304. DOI: 10.1002/spe.869.
27. Snelting G, Robschink T, Krinke J. Efficient path conditions in dependence graphs for software safety analysis. *ACM Transactions on Software Engineering and Methodology* 2006; **15** (4): 410–457. DOI: 10.1145/1178625.1178628.
28. Cochran WG, Cox GM. *Experimental Designs*. 2nd edition, Wiley, ISBN 0471162035, 1957.
29. Mandl R. Orthogonal Latin squares: An application of experimental design to compiler testing. *Communications of the ACM* 1985; **28** (10): 1054–1058. DOI: 10.1145/4372.4375.
30. Lei Y, Kacker R, Kuhn D, Okun V, Lawrence J. IPOG/IPOD: Efficient test generation for multi-way software testing. *Software Testing, Verification, and Reliability* 2008; **18** (3): 125–148. DOI: 10.1002/stvr.381.
31. Tai K-C, Lei Y. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering* 2002; **28** (1): 109–111. DOI: 10.1109/32.979992.
32. Cartaxo EG, Machado PDL, Neto FGO. On the use of a similarity function for test case selection in the context of model-based testing. *Software Testing, Verification and Reliability* 2011. DOI: 10.1002/stvr.413.
33. Fraser G, Gargantini A. An evaluation of specification based test generation techniques using model checkers. *Testing: Academic and Industrial Conference – Practice and Research Techniques (TAIC-PART)*, 2009; 72–81. DOI: 10.1109/TAICPART.2009.20.
34. Lin J-W, Huang C-Y. Analysis of test suite reduction with enhanced tie-breaking techniques. *Information and Software Technology* 2009; **51** (4): 679–690. DOI: 10.1016/j.infsof.2008.11.004.