HKU CS Tech Report TR-2009-19

Postprint of article in *IEEE Transactions on Software Engineering* **37** (5): 616–634 (2011)

A Comparison of Tabular Expression-Based Testing Strategies

Xin Feng, David Lorge Parnas, *Fellow, IEEE,*
T. H. Tse, *Senior Member, IEEE,* and Tony O'Callaghan

Abstract—Tabular expressions have been proposed as a notation to document mathematically precise but readable software specifications. One of the many roles of such documentation is to guide testers. This paper 1) explores the application of four testing strategies (the partition strategy, decision table-based testing, the basic meaningful impact strategy, and fault-based testing) to tabular expression-based specifications, and 2) compares the strategies on a mathematical basis through formal and precise definitions of the subsumption relationship. We also compare these strategies through experimental studies. These results will help researchers improve current methods and will enable testers to select appropriate testing strategies for tabular expression-based specifications.

Index Terms—Tabular expression, test case constraint, subsume, unconditionally subsume, conditionally subsume.

◆

1 INTRODUCTION

In past decades, researchers, and engineers have endeavored to improve the precision, completeness, and consistency of documentation in software engineering. As mathematics is the best way to achieve precision, mathematical expressions often occur throughout the documentation. Software engineering has benefited from the use of mathematics. However, conventional mathematical expressions used in software engineering are usually complicated and hard to read and verify.

As an improvement, a tabular representation [20], [21], [22], [33], [36], [37], [40], [44] has been proposed to model such mathematical expressions in software specifications. When compared with traditional mathematical expressions, this representation improves readability and makes the documentation concise. In addition, it is easier to check the consistency and completeness of specifications in tabular expression form. This notation has proven to be useful in various examples in the industry, including the US Navy's A-7 aircraft [2], [18], the Darlington Nuclear Power Station [34], [35], a Dell keyboard test program [3], and an Ericsson telecom software system [39]. These documents are used not only by software engineers but also by software testers. The tabular structure gives testers a clear idea of how the input domain is divided, as well as the distinct boundary points of each subdomain. With these features, Liu [28] proposed the partition strategy for tabular expressions and Clermont and Parnas [11] suggested the interesting point selection strategy for test case generation; Peters and Parnas [38] developed tools to generate test oracles automatically from tabular expressions. Moreover, the tabular structure does not exclude other testing strategies. This offers flexibility in the application of testing strategies. Due to the high cost of software testing and tight delivery schedules, it is often impractical to apply all possible strategies. Furthermore, some strategies may not guarantee additional confidence in the software. Therefore, when several testing strategies are available directly or indirectly for use with a tabular expression-based specification, it will be highly beneficial for testers to have guidelines that help them select and apply the most effective strategy.

As tabular expressions can be viewed as a tabular form of conventional mathematical expressions, testing strategies based on conventional mathematical expressions can be used with tabular expressions as well. Since tabular expressions are particularly useful in describing conditional relationships between inputs and outputs, the corresponding conventional mathematical expressions usually contain several conditions with specific restrictions. More than 10 years ago, the basic meaningful impact strategy [46] was proposed for Boolean specifications. In subsequent years,

---

This research is sponsored by Science Foundation Ireland (SFI) under grants 01/P1.2/C009 and 03/CE3/1405, and the General Research Fund of the Research Grants Council of Hong Kong under project no. 717308.

- *Xin Feng worked with the Software Quality Research Laboratory, University of Limerick, Ireland and is now with the Division of Science and Technology, United International College, Zhuhai, Guangdong, China. Email: xinfeng@uic.edu.hk. Phone: (+86) 756 362 0030.*
- *David Lorge Parnas worked with the Software Quality Research Laboratory, University of Limerick, Ireland and is retired in Ottawa Canada. Email: david.parnas@ul.ie. Phone: (+1) 613 249 8038.*
- *T. H. Tse is with the Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong. Email: thtse@cs.hku.hk. Phone: (+852) 2859 2183.*
- *Tony O'Callaghan was with the Interaction Design Centre, University of Limerick, Ireland. Email: tony.ocallaghan@ul.ie. Phone: (+353) 87 746 1906.*

fault-based testing [7], [25], [26], [27], [29], [31] that generates test data from Boolean specifications was developed. Researchers in fault-based testing have established a mature hierarchy diagram of fault classes. Both the basic meaningful impact strategy and fault-based testing for Boolean specifications have been demonstrated to be effective through experimentation. Other strategies such as *MC/DC* [10] and *MUMCUT* [8] have also been suggested. Although *MC/DC* was not originally proposed for Boolean specifications, it does share similar principles with the basic meaningful impact strategy. The *MUMCUT* strategy has been evaluated in the context of fault-based testing [27] and extended by considering undetected mutation patterns collected in an experimental study [42]. A comparative study between *MC/DC* and *MUMCUT* was conducted by Yu and Yau [48]. Kaminski et al. [24] also compared a number of logic testing methods including the *MUMCUT* strategy, *MAX_A*, and *MAX_B*. *MAX_A* and *MAX_B* are extensions of the basic meaningful impact strategy.

The hierarchy diagram of fault classes in [27] illustrates the relationships among fault classes. (The diagram is reproduced in Fig. 1 in Section 3.6.) The figure shows that test cases covering the LOF and LIF classes of faults can also detect the other fault classes in the diagram. It is, therefore, worth examining fault-based testing for the LOF and LIF classes of faults.

Since the relationships between inputs and outputs in tabular expressions are very similar to the correspondences between input conditions and actions in decision table-based testing [23], it is appropriate to apply this method to tabular expressions.

As for the partition strategy [28] and the interesting point selection strategy for tabular specifications [11], we pick only the former because the latter selects special boundary points for stress testing.

Thus, as an initial exploration of test case generation from tabular expressions, we compare four testing strategies: the partition strategy, decision table-based testing, the basic meaningful impact strategy, and fault-based testing for LOF and LIF faults. The basic meaningful impact strategy and fault-based testing for Boolean specifications work on single Boolean expressions, while decision table-based testing creates a decision table from a specification. Hence, these strategies cannot be used for the tabular expressions directly. This paper provides algorithms to apply these strategies to tabular expressions and express them in terms of test case constraints.

Testing strategies can be compared using several kinds of measures, among which coverage and fault classes are popularly used.

1. **Coverage.** Coverage is a metric of completeness with respect to a test selection criterion [5]. This metric is mostly used to compare source code-based testing strategies such as *all-du-paths*, *all-uses*, *all-p-uses*, *all-c-uses*, *all-paths*, *branch*, and *statement* coverage criteria [5]. A diagram that illustrates the subsumption relationships of these strategies can be found in [5] and [45]. The *all-paths* strategy is the strongest among these strategies, while *all-du-paths* is the strongest data flow testing strategy. This metric is not only used in source code-based testing, but can also be used in some specification-based testing strategies such as equivalence class testing strategies. Consider two equivalence classes $\{x \mid x \geq 5\}$ and $\{x \mid x < 5\}$. At least two test cases are generated, one from each equivalence class. If the relations that define the classes are considered, the equivalence class $\{x \mid x \geq 5\}$ can be further separated into two equivalence classes $\{x \mid x > 5\}$ and $\{x \mid x = 5\}$. The latter has better coverage of the input domain [23].
2. **Fault classes.** Fault classes have often been used to measure fault-based testing strategies. Fault-based testing seeks to demonstrate that prescribed faults are absent in a program [29]. Hence, it is usually taken as a source code-based testing strategy. In recent years, this strategy has been extended to generate test cases from Boolean specifications [7], [25], [26], [27], [31]. Arithmetic operator faults in source code [1], [13], [19], [43] and literal insertion faults (LIF) in a specification [25], [27] are examples of fault classes. The subsumption relationship of the fault-based strategies has been verified through experimentation [12] and by the study of the fault detection conditions [25], [27], [31].

   It has been found that fault-based testing strategies based on some fault classes are more effective than those based on others. In [25], [27], [31], hierarchy diagrams show a partial ordering of fault classes that represents the subsumption relationship of the corresponding testing strategies. Test cases that reveal faults of the classes at lower levels of the diagrams can reveal faults of the classes at higher levels. Intuitively, a strategy that focuses on fault classes at lower levels should be more effective. However, the prerequisites are that faults of the classes at lower levels can exist and that a specification with such faults is not equivalent to the original specification. This is not always the case.

In addition, other measures (such as the P-measure [47], E-measure [9], and F-Measure [6]) have been proposed and are mainly used in comparing partition and random testing strategies. Some papers [4] have compared the effectiveness of testing strategies with respect to costs as well.

Since the objective of this paper is to compare the effectiveness of detecting software faults, we adopt and improve the following definition that has been commonly used to compare testing strategies:

Definition 1 (Subsumption): Criterion C_1 subsumes criterion C_2 if every test suite that satisfies C_1 also satisfies C_2.

We can see that comparisons based on coverage and fault classes follow this definition. In general, when criterion C_1 subsumes criterion C_2, C_1 is better at detecting faults. However, as pointed out in [15], this is not guaranteed. This also happens in fault-based testing when faults cannot be found for the classes at lower levels. It is possible to determine the subsumption relationship of two testing strategies that are applied to a concrete specification. Alternatively, subsumption relationships can be related to a class

TABLE 1
Function *DayError* in tabular expression (inverted)

$DayError(day,\ month,\ year) \equiv$

	$T[2]$	
	true	*false*
$MonthType(month) = M\_31$	$day < 1\ \vee\ day > 31$	$day \geq 1\ \wedge\ day \leq 31$
$MonthType(month) = M\_30$	$day < 1\ \vee\ day > 30$	$day \geq 1\ \wedge\ day < 30$
$MonthType(month) = M\_28\_29$	$day < 1\ \vee\ (day > 29\ \wedge\ YearType(year) = LeapYear)\ \vee\ (day > 28\ \wedge\ YearType(year) = CommonYear)$	$day \geq 1\ \wedge\ ((day \leq 29\ \wedge\ YearType(year) = LeapYear)\ \vee\ (day \leq 28\ \wedge\ YearType(year) = CommonYear))$
$T[1]$	$T[0]$	

of specifications or to all specifications. A testing strategy subsuming another testing strategy on a single program does not mean that this subsumption relationship can be extended to a class of specifications or to all specifications. It is possible that a subsumption relationship holds with respect to a certain condition.

If this subsumption relationship changes when these testing strategies are applied to different specifications, testers will be uncertain with respect to the choice of testing strategies. To avoid this uncertainty, we will improve the above definition by giving formal and precise definitions of the subsumption relationship. The new definitions aim to help testers obtain a clearer understanding of subsumption relationships and the necessary conditions that support them.

Several types of tables have been defined in [33] and [44]. This paper mainly discusses normal tables in two dimensions. A discussion relating to other table types and higher dimensions will be provided in the conclusion.

2 TABULAR EXPRESSIONS

Tabular expressions are a way to improve the readability of mathematical expressions. The "divide-and-conquer" structure of the table notation not only provides software engineers with clear relationships between inputs and outputs, but also helps them check the consistency and completeness of documents by inspecting the rows and columns only. It is easier to use the expression without evaluating all the subexpressions. Let us consider the following example:

$$
\begin{aligned}
& DayError(year,\ month,\ day) \\
\equiv\ & MonthType(month) = M\_31\ \wedge\ (day < 1\ \vee\ day > 31)\ \vee \\
& MonthType(month) = M\_30\ \wedge\ (day < 1\ \vee\ day > 30)\ \vee \\
& MonthType(month) = M\_28\_29\ \wedge \\
& (day < 1\ \vee\ (day > 29\ \wedge\ YearType(year) = LeapYear)\ \vee \\
& (day > 28\ \wedge\ YearType(year) = CommonYear)).
\end{aligned}
$$

The expression can be written in tabular notation as illustrated in Table 1.

When compared with the tabular notation, the previous form is typically more difficult to read and verify [33]. Two other specification examples that use tabular expressions are given in Appendix A. More examples can be found in [21], [33], and [38].

Tabular expressions are defined as an indexed set [17] of grids, and a grid is an indexed set of expressions [33], [44]. There are several table types, such as *normal*, *inverted*, and *tree-structured* [33], [44]. The specification in Table 1 uses an inverted table type; the *MonthType* table (see Fig. 2 in Appendix A) is a tree-structured table, and the *Price* table (see Fig. 3 in Appendix A) is a normal table. It has been shown that one table form can be transformed to another. In Appendix B, for instance, we have transformed the inverted table for *DayError* presented in Table 1 into both a tree-structured table and a normal table. More examples of table transformations can be found in [21], [33], [41], and [49].

Table 2 is the general format of a two-dimensional $m \times n$ normal table. There are three grids in this table: $T[0]$, $T[1]$, and $T[2]$. $T[0]$ is the main grid; $T[1]$ and $T[2]$ are the predicate grids. The expressions in grids $T[1]$ and $T[2]$ are *predicate expressions*. The expressions in grid $T[0]$ are *evaluation expressions*, which can be evaluated to give the values of the target function. Each such expression is used when the corresponding row and column predicates are both *true*. The expressions in the main grid might be *undefined*; this would occur if the conjunction of the corresponding predicates was *false* or outside of the domain of the function defined by the table.

TABLE 2
An *m* × *n* normal table

	$T[2]$				
	$T[2][1]$	$\ldots$	$T[2][j]$	$\ldots$	$T[2][n]$
$T[1][1]$	$T[0][1,1]$	$\ldots$	$T[0][1,j]$	$\ldots$	$T[0][1,n]$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$T[1][i]$	$T[0][i,1]$	$\ldots$	$T[0][i,j]$	$\ldots$	$T[0][i,n]$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$T[1][m]$	$T[0][m,1]$	$\ldots$	$T[0][m,j]$	$\ldots$	$T[0][m,n]$
$T[1]$	$T[0]$				

For ease of presentation, we use $\bigwedge_{k=1}^{l} p_k$ to denote $p_1 \wedge p_2 \wedge \cdots \wedge p_l$ and $\bigvee_{k=1}^{l} p_k$ to denote $p_1 \vee p_2 \vee \cdots \vee p_l$. In a normal table, the grids $T[1]$ and $T[2]$ must be proper, that is, for any input, $T[1][i] \wedge T[1][j] = \mathit{false}$ if $i \neq j$ and $\bigvee_{k=1}^{m} T[1][k] = \mathit{true}$, where m is the number of cells in $T[1]$. Here, $T[0][i,j]$ is the expression to be evaluated if $T[1][i] \wedge T[2][j]$ is *true* with respect to an assignment of values to the variables. We call $T[1][i] \wedge T[2][j]$ an *evaluation condition*, denoted by $E_{i,j}$. Furthermore, $E_{i_1,j_1} \wedge E_{i_2,j_2} = \mathit{false}$ if $i_1 \neq i_2$ or $j_1 \neq j_2$.

If an expression in grid $T[0]$ is identical to another expression in the same grid, then they are called *duplicated evaluation expressions*. Suppose the number of occurrences of an evaluation expression is l (≥ 1), and $T[1][i_k]$ and $T[2][j_k]$ $(k = 1, 2, \ldots, l;\ i_k = 1, 2, \ldots, m;$ and $j_k = 1, 2, \ldots, n)$

are predicates in $T[1]$ and $T[2]$ that correspond to the evaluation expressions. Then, $\bigvee_{k=1}^{l}(T[1][i_k] \wedge T[2][j_k])$ is called a *combined evaluation condition* when $l > 1$. For example, there are three *true* and three *false* occurrences in the main grid of Table 16 in Appendix B. In Section 3, some testing strategies are based on combined evaluation conditions.

3 APPLICATION OF THE TESTING STRATEGIES TO TABULAR EXPRESSION-BASED SPECIFICATIONS

This section discusses the application of the four testing strategies to tabular expression-based specifications. Every strategy produces a list of test case constraints such that no constraint is *false*. Test cases are obtained by finding values that satisfy these constraints.

3.1 Irreducible DNF

Before we define an irreducible DNF, we need to introduce a few fundamental definitions. Some of these are slightly different from the standard concepts in Boolean algebra, as we will explain below. A Boolean literal is usually defined as a Boolean variable or its negation, or the Boolean constant *true* or *false*. In this paper, we extend the definition so that a *Boolean literal* can also be a *simple predicate*, that is, it can be the result of a Boolean-valued function, or a relational expression of the form e_1 *op* e_2, where *op* is a relational operator and e_1 and e_2 are arithmetic expressions. A *Boolean expression* consists of Boolean literals linked up by the Boolean operators "$\wedge$" (which denotes "and") and "$\vee$" (which denotes "or"). A *conjunction* is a Boolean expression consisting of two subexpressions linked by the operator "$\wedge$". A *disjunction* is a Boolean expression consisting of two subexpressions linked by the operator "$\vee$". A *Disjunctive Normal Form* (*DNF*) is a Boolean expression consisting of disjunctions of conjunctions of Boolean literals. For example, given the Boolean variables a, b, and c, the expression $\neg a \ \vee \ (b \ \wedge \ c)$ is in DNF, but $\neg a \ \wedge \ (b \ \vee \ c)$ is not.

An *irreducible DNF* is a DNF such that the removal of any Boolean literal or conjunction will change the truth table of the expression [46]. Typically, the concept of "irreducible DNF" is based on pure Boolean expressions. As highlighted in [43], for instance, "A [pure] Boolean expression is a predicate with no relational expressions." In this paper, however, the definition of "irreducible DNF" takes into account that a Boolean literal can be a relational expression or the result of a Boolean-valued function. Thus, a DNF that is irreducible according to pure Boolean expressions may be reducible when the Boolean literals are expanded to reveal the relational expressions. For example, $(a \wedge b \wedge \neg c) \vee (\neg a \wedge b \wedge c)$ is normatively an irreducible DNF because the removal of any literal or conjunction will change its resultant truth table. However, if a is "*day* > 31" and c is "*day* < 30", then $\neg c$ and $\neg a$ are redundant.

Thereinafter, we will assume that $E_{i,j}$ is an irreducible DNF unless otherwise stated. The evaluation condition $E_{i,j} = T[1][i] \wedge T[2][j]$ can be written as

$$\bigvee_{k=1}^{w_{i,j}}(c_{i,j}^{k,1} \wedge \cdots \wedge c_{i,j}^{k,s_{i,j}^{k}}),$$

where $c_{i,j}^{k,k'}$ ($k' = 1, 2, \ldots, s_{i,j}^{k}$ is a Boolean literal, $w_{i,j}$ is the number of terms in $E_{i,j}$, and $s_{i,j}^{k}$ is the number of Boolean literals in the kth term of $E_{i,j}$. For example, if $T[1][2]$ is $(x > 3 \ \vee \ x < 0)$ and $T[2][3]$ is $(y > 10)$, then $E_{2,3} = x > 3 \ \wedge \ y > 10 \vee x < 0 \wedge y > 10$. In this expression, $w_{2,3} = 2$, $s_{2,3}^{1} = s_{2,3}^{2} = 2$, $c_{2,3}^{1,1} = x > 3$, $c_{2,3}^{1,2} = y > 10$, $c_{2,3}^{2,1} = x < 0$, and $c_{2,3}^{2,2} = y > 10$.

TABLE 3
A 2 × 2 normal table

$T[2]$

	$x < 1 \ \vee \ x > 31$	$x \geq 1 \ \wedge \ x \leq 31$
$y > 1$	x	$x + 1$
$y \leq 1$	y	$y + 1$

T[1] $T[0]$

3.2 An Illustration

In the following sections, we will discuss the application of testing strategies to tabular expressions. A list of abstract test case constraints is determined for each strategy. To help readers understand the complex formulas, an example in Table 3 is used to illustrate abstract test case constraints. The following conditions that correspond to the individual evaluation expressions can be derived from the table:

$$\begin{aligned}
E_{1,1} &= (y > 1 \ \wedge \ x < 1) \ \vee \ (y > 1 \ \wedge \ x > 31), \\
E_{1,2} &= y > 1 \ \wedge \ x \geq 1 \ \wedge \ x \leq 31, \\
E_{2,1} &= (y \leq 1 \ \wedge \ x < 1) \ \vee \ (y \leq 1 \ \wedge \ x > 31), \\
E_{2,2} &= y \leq 1 \ \wedge \ x \geq 1 \ \wedge \ x \leq 31.
\end{aligned}$$

In the above expressions, $w_{1,1} = w_{2,1} = 2$ and $w_{1,2} = w_{2,2} = 1$.

3.3 Partition Strategy for Tabular Expressions

Partition testing has been a widely used testing strategy for many years [16], [30], [32]. The partition strategy for tabular expressions was proposed by Liu [28] and his supervisor von Mohrenschildt. This strategy takes advantage of the features of tabular expressions, including the intentional division of the input domain. It is actually an equivalence class testing technique. The equivalence classes are more obvious in a tabular expression specification than in conventional mathematical expressions. The strategy requires that each cell other than those undefined in the main grid should be tried, that is, tested to see if the output is $T[0][i,j]$ with respect to an assignment that fulfills both $T[1][i]$ and $T[2][j]$. At most $m \times n$ test cases are sufficient to satisfy this requirement. The resulting list of test case constraints is

$$\Big\langle \bigvee_{k=1}^{w_{i,j}}(c_{i,j}^{k,1} \wedge \cdots \wedge c_{i,j}^{k,s_{i,j}^{k}}) \Big\rangle_{O(i,j)},$$

where $O(i,j)$ denotes $i = 1, 2, \ldots, m \ \wedge \ j = 1, 2, \ldots, n \ \wedge \ T[0][i,j] \neq$ *undefined* for ease of presentation. This notation is used throughout the rest of the paper.

The list of test case constraints derived from this formula for Table 3 is

$$\begin{aligned}
\langle \ & (y > 1 \ \wedge \ x < 1) \ \vee \ (y > 1 \ \wedge \ x > 31), \\
& y > 1 \ \wedge \ x \geq 1 \ \wedge \ x \leq 31, \\
& (y \leq 1 \ \wedge \ x < 1) \ \vee \ (y \leq 1 \ \wedge \ x > 31), \\
& y \leq 1 \ \wedge \ x \geq 1 \ \wedge \ x \leq 31 \ \rangle.
\end{aligned}$$

3.4 Decision Table-Based Testing

Decision tables have been used to describe and analyze complex logical relationships [23]. Decision table-based testing identifies test cases from a decision table, where actions and corresponding conditions that produce these actions are described. A sample decision table is shown in Table 4.

TABLE 4
Decision table

	Stubs	Entries		
		1	2	3
Conditions	c_1	T	F	F
	c_2	–	T	F
	a_1	√		
Actions	a_2		√	
	Impossible			√

As shown in Table 4, a decision table consists of four parts. The vertical line separates the stubs portion on the left from the entries portion on the right. The stubs portion lists all the conditions that are used to check the inputs and all the actions that should be done by the program. The entries portion matches the actions with the corresponding combinations of truth values of the conditions. The horizontal line then separates the conditions portion from the actions portion. Since a tabular expression also specifies the relationships between inputs and expected outputs, decision table-based testing can be used to generate test data from tabular expression-based specifications. In Table 4, there are two possible actions, a_1 and a_2, depending on the conditions c_1 and c_2 that are imposed on the inputs. Here, c_1 and c_2 are simple predicates. A "T" entry indicates *true* and an "F" entry indicates *false*. With respect to an input, if c_1 is evaluated to *true*, the action is a_1, irrespective of the value that c_2 is evaluated to; if c_1 is evaluated to *false* and c_2 is evaluated to *true*, the action is a_2. It is impossible that both c_1 and c_2 are evaluated to *false* simultaneously.

TABLE 5
Inconsistency of columns

	1	2	3	4
c_1	–	T	T	F
c_2	T	F	T	T
c_3	T	–	F	T
a_1	√			
a_2		√	√	√

TABLE 6
Redundancy of columns

	1	2	3
c_1	–	T	T
c_1	T	F	–
c_2	T	–	F
a_1	√		
a_2		√	√

The symbol "–" in these decision tables means "don't care," that is, the truth values of corresponding conditions do not affect the expected actions. For a deterministic program, inconsistencies and redundancies should be avoided. In a decision table with inconsistency, the same combination of conditions may produce different actions. In Table 5, for instance, columns 1 and 4 are inconsistent. According to column 1, (c_1 = F, c_2 = T, c_3 = T) will produce the action a_1. According to column 4, however, the same input will produce the action a_2. In a decision table with redundancy, two columns contain the same values of conditions and the same actions. In Table 6, for example, (c_1 = T, c_2 = F, c_3 = F) is implied in both columns 2 and 3. In fact, both redundancy and inconsistency are caused by an overlap of conditions in the entries portion. If there is no overlap of conditions in different columns, redundancy and inconsistency are avoided. To apply decision table-based testing in tabular expressions using either normal tables or other types of tables, we can list all the Boolean literals and actions and then construct a decision table. Alternatively, we propose the following algorithm for this application:

1. Transform a tabular expression into an equivalent conventional mathematical expression, where each evaluation expression corresponds to one evaluation condition.
2. Combine the evaluation conditions that correspond to the same evaluation expression.
3. Transform each evaluation condition or combined evaluation condition into an equivalent expression in irreducible DNF.
4. Create a constraint for every term in each expression (in irreducible DNF) that is not equivalent to *false*. If the expression in irreducible DNF is $p_1 \vee \cdots \vee p_k \vee \cdots \vee p_h$, the constraint for term p_k is $p_k \wedge \bigwedge_{k_1=1, k_1 \neq k}^{h} \neg p_{k_1}$, that is, the data that satisfy the constraint evaluate p_k to *true* and all other terms in the expression evaluate to *false*.

If there is only one term in the expression, the constraint is p_1. If no evaluation expression is duplicated, step 2 can be skipped. Appendix C illustrates how this algorithm is applied to the *DayError* example.

Lemma 1: Consider an irreducible DNF expression $p_1 \vee \cdots \vee p_k \vee \cdots \vee p_h$ ($h \geq 1$) not equivalent to *false*. At least one solution can be found for the constraint $p_k \wedge \bigwedge_{k_1=1, k_1 \neq k}^{h} \neg p_{k_1}$, where $k = 1, 2, \ldots, h$.

Lemma 1 will be used in the proof of Theorem 1. The constraints in step 4 are not *false*. Each constraint obtained from step 4 is equivalent to a combination of conditions in one column of the corresponding decision table, that is, the corresponding column in the decision table exists. Moreover, in a tabular expression, since only one evaluation condition or one combined evaluation condition is evaluated to *true* at any one time and the test cases that satisfy the constraint evaluate only one term to *true* and all other terms to *false*, there is no overlap in constraints. In other words, there is no overlap of columns in the corresponding decision table. The resulting list of test case constraints contains the constraints for every term in each evaluation condition and each combined evaluation condition. The list for an $m \times n$ normal table without duplicated evaluation expressions is

$$\left\langle c_{i,j}^{k,1} \wedge \cdots \wedge c_{i,j}^{k, s_{i,j}^{k}} \wedge \bigwedge_{k_1=1, k_1 \neq k}^{w_{i,j}} \neg (c_{i,j}^{k_1,1} \wedge \cdots \wedge c_{i,j}^{k_1, s_{i,j}^{k_1}}) \right\rangle_{O(i,j,k)},$$

where $O(i,j,k)$ denotes $O(i,j) \ \wedge \ k = 1, 2, \ldots, w_{i,j}$ for ease of presentation. This notation is used throughout the rest of paper.

The list of test case constraints derived from this formula for Table 3 is

$$\begin{aligned}\langle \ & y > 1 \ \wedge \ x < 1 \ \wedge \ \neg(y > 1 \ \wedge \ x > 31), \\ & \neg(y > 1 \ \wedge \ x < 1) \ \wedge \ y > 1 \ \wedge \ x > 31, \\ & y > 1 \ \wedge \ x \geq 1 \ \wedge \ x \leq 31, \\ & y \leq 1 \ \wedge \ x < 1 \ \wedge \ \neg(y \leq 1 \ \wedge \ x > 31), \\ & \neg(y \leq 1 \ \wedge \ x < 1) \ \wedge \ y \leq 1 \ \wedge \ x > 31, \\ & y \leq 1 \ \wedge \ x \geq 1 \ \wedge \ x \leq 31 \ \ \rangle.\end{aligned}$$

3.5 The Basic Meaningful Impact Strategy

The basic meaningful impact strategy includes a family of criteria that generate test cases from single Boolean expressions [46]. A *unique true point* for a term in a Boolean expression is a combination of truth values of Boolean variables that evaluates the term to *true* and the other terms to *false*. A *near false point* for a literal in a term is a combination of truth values of Boolean variables that evaluates the term (where the Boolean literal is negated) to *true* and evaluates the other terms to *false*.

For example, a simple strategy may generate test cases in the following steps:

1. Transform a Boolean expression to irreducible DNF.
2. For each term, create a set of unique true points.
3. For each Boolean literal, create a set of near false points.
4. Select *one point* from each set and construct a set of test case constraints.

This strategy applies the *ONE* criterion. Since it is a straightforward implementation of the basic meaningful impact strategy, it faithfully reflects all the principles of that strategy. According to the experimental study in [46], the *ONE* criterion is very effective in fault detection. Other enhanced criteria (such as *MAX-A* and *MAX-B*) select more or all points from each set. However, these criteria require significantly more test cases than the *ONE* criterion. In this paper, therefore, we will use the basic meaningful impact strategy with the *ONE* criterion. To apply this strategy in tabular expressions, the latter must first be transformed into their equivalent conventional mathematical expressions. The following steps describe how to apply the strategy in tabular expressions:

1–4. These steps are the same as those for decision table-based testing except that lists are used instead of sets.

5. Create a constraint for every Boolean literal in each evaluation condition or combined evaluation condition. For an expression of the form $\bigvee_{k_1=1}^{h}(r_{k_1}^{1} \wedge \cdots \wedge r_{k_1}^{d_{k_1}})$, the constraint for r_k^l ($k = 1, 2, \ldots, h$ and $l = 1, 2, \ldots, d_k$) is $\neg r_k^l \ \wedge \ \bigwedge_{l_1=1, l_1 \neq l}^{d_k} r_k^{l_1} \ \wedge \ \bigwedge_{k_1=1, k_1 \neq k}^{h} \neg(r_{k_1}^{1} \wedge \cdots \wedge r_{k_1}^{d_{k_1}})$.
For an expression with only one term, the constraint for r_1^l is $r_1^1 \wedge \cdots \wedge \neg r_1^l \wedge \cdots \wedge r_1^{d_1}$ if $d_1 > 1$, and $\neg r_1^1$ otherwise.

Lemma 2: Suppose $\bigvee_{k_1=1}^{h}(r_{k_1}^{1} \wedge \cdots \wedge r_{k_1}^{d_{k_1}})$ is an irreducible DNF expression that is not equivalent to *true* and not equivalent to *false*. At least one solution can be found for the constraint $(r_k^1 \wedge \cdots \wedge \neg r_k^l \wedge \cdots \wedge r_k^{d_k}) \wedge \bigwedge_{k_1=1, k_1 \neq k}^{h} \neg(r_{k_1}^{1} \wedge \cdots \wedge r_{k_1}^{d_{k_1}})$, where $k = 1, 2, \ldots, h$ and $l = 1, 2, \ldots, d_k$.

According to Lemma 2, the constraints are not equivalent to *false* in step 5. The resulting list of test case constraints is the concatenation of the two lists obtained from steps 4 and 5:

$$\begin{aligned} & \Big\langle \bigwedge_{l_1=1}^{s_{i,j}^{k}} c_{i,j}^{k,l_1} \ \wedge \ \bigwedge_{k_1=1, k_1 \neq k}^{w_{i,j}} \neg(c_{i,j}^{k_1,1} \wedge \cdots \wedge c_{i,j}^{k_1, s_{i,j}^{k_1}}) \Big\rangle_{O(i,j,k)} \\ \oplus & \Big\langle (\neg c_{i,j}^{k,l} \ \wedge \ \bigwedge_{l_1=1, l_1 \neq l}^{s_{i,j}^{k}} c_{i,j}^{k,l_1}) \ \wedge \\ & \quad \bigwedge_{k_1=1, k_1 \neq k}^{w_{i,j}} \neg(c_{i,j}^{k_1,1} \wedge \cdots \wedge c_{i,j}^{k_1, s_{i,j}^{k_1}}) \Big\rangle_{O(i,j,k) \ \wedge \ l=1,2,\ldots,s_{i,j}^{k}}, \end{aligned}$$

where $\oplus$ denotes list concatenation.

The list of test case constraints derived from this formula for Table 3 is

$$\begin{aligned} \langle \ & y > 1 \ \wedge \ x < 1 \ \wedge \ \neg(y > 1 \ \wedge \ x > 31), \\ & \neg(y > 1 \ \wedge \ x < 1) \ \wedge \ y > 1 \ \wedge \ x > 31, \\ & y > 1 \ \wedge \ x \geq 1 \ \wedge \ x \leq 31, \\ & y \leq 1 \ \wedge \ x < 1 \ \wedge \ \neg(y \leq 1 \ \wedge \ x > 31), \\ & \neg(y \leq 1 \ \wedge \ x < 1) \ \wedge \ y \leq 1 \ \wedge \ x > 31, \\ & y \leq 1 \ \wedge \ x \geq 1 \ \wedge \ x \leq 31 \ \ \rangle \\ \oplus \quad \langle \ & \neg(y > 1) \ \wedge \ x < 1 \ \wedge \ \neg(y > 1 \ \wedge \ x > 31), \\ & y > 1 \ \wedge \ \neg(x < 1) \ \wedge \ \neg(y > 1 \ \wedge \ x > 31), \\ & \neg(y > 1 \ \wedge \ x < 1) \ \wedge \ \neg(y > 1) \ \wedge \ x > 31, \\ & \neg(y > 1 \ \wedge \ x < 1) \ \wedge \ y > 1 \ \wedge \ \neg(x > 31), \\ & \neg(y > 1) \ \wedge \ x \geq 1 \ \wedge \ x \leq 31, \\ & y > 1 \ \wedge \ \neg(x \geq 1) \ \wedge \ x \leq 31, \\ & y > 1 \ \wedge \ x \geq 1 \ \wedge \ \neg(x \leq 31), \\ & \neg(y \leq 1) \ \wedge \ x < 1 \ \wedge \ \neg(y \leq 1 \ \wedge \ x > 31), \\ & y \leq 1 \ \wedge \ \neg(x < 1) \ \wedge \ \neg(y \leq 1 \ \wedge \ x > 31), \\ & \neg(y \leq 1 \ \wedge \ x < 1) \ \wedge \ \neg(y \leq 1) \ \wedge \ x > 31, \\ & \neg(y \leq 1 \ \wedge \ x < 1) \ \wedge \ y \leq 1 \ \wedge \ \neg(x > 31), \\ & \neg(y \leq 1) \ \wedge \ x \geq 1 \ \wedge \ x \leq 31, \\ & y \leq 1 \ \wedge \ \neg(x \geq 1) \ \wedge \ x \leq 31, \\ & y \leq 1 \ \wedge \ x \geq 1 \ \wedge \ \neg(x \leq 31) \ \ \rangle. \end{aligned}$$

3.6 Fault-Based Testing

Fault-based testing is typically used to demonstrate that certain faults are not present in the software. In recent years, a lot of research has been put into applying this strategy to specification-based testing. Kuhn [25] gave a hierarchy of fault classes, and then Lau and Yu [27] and Okun et al. [31] extended the diagram by adding more fault classes. However, since the research by Okun et al. is not based on Boolean expressions, we do not discuss the faults in [31] in this paper. The following are the fault classes appraised in [27]:

- Expression Negation Fault (ENF): The entire expression or a subexpression of it is implemented as its negation.
- Term Negation Fault (TNF): A term is implemented as its negation.
- Operator Reference Fault (ORF): The logical operator "$\wedge$" is implemented as "$\vee$" (ORF[.]), or "$\vee$" is implemented as "$\wedge$" (ORF[+]).

- Literal Negation Fault (LNF): A Boolean literal is implemented as its negation.
- Term Omission Fault (TOF): A term is omitted in its implementation.
- Literal Reference Fault (LRF): A Boolean literal is replaced by another Boolean literal.
- Literal Omission Fault (LOF): A Boolean literal is omitted from a term.
- Literal Insertion Fault (LIF): A Boolean literal is inserted into a term in which the literal or its negation is not present.

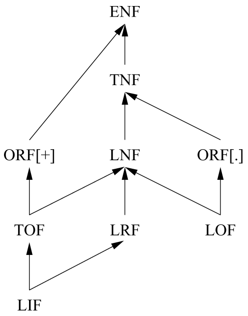


Fig. 1. Hierarchy of fault classes (from [27])

Fig. 1 shows the hierarchy diagram from Lau and Yu [27], given in terms of detection conditions, that is, the conditions for a test case to reveal the faults in a class. An arrow from fault class A to fault class B means that test cases that detect A can also detect B. LOF and LIF are at the bottom levels of the hierarchy. In other words, testing strategies based on them are more effective than those based on the other fault classes. Hence, fault-based testing in this paper takes two fault classes into account, namely LOF and LIF. The resulting lists of test case constraints for an $m{\times}n$ normal table without duplicated evaluation expressions are

$$\begin{aligned}&\Big\langle \neg c_{i,j}^{k,l} \wedge \bigwedge\nolimits_{l_1=1,l_1\neq l}^{s_{i,j}^{k}} c_{i,j}^{k,l_1} \wedge \\ &\quad \bigwedge\nolimits_{k_1=1,k_1\neq k}^{w_{i,j}} \neg(c_{i,j}^{k_1,1} \wedge \cdots \wedge c_{i,j}^{k_1,s_{i,j}^{k_1}})\Big\rangle_{O(i,j,k)\ \wedge\ s_{i,j}^{k}>1\ \wedge\ l=1,2,\ldots,s_{i,j}^{k}} \\ &\oplus \Big\langle c_{i,j}^{k,1} \wedge \bigwedge\nolimits_{k_1=1,k_1\neq k}^{w_{i,j}} \neg(c_{i,j}^{k_1,1} \wedge \cdots \wedge c_{i,j}^{k_1,s_{i,j}^{k_1}})\Big\rangle_{O(i,j,k)\ \wedge\ s_{i,j}^{k}=1}\end{aligned}$$

for LOF and

$$\begin{aligned}&\Big\langle \bigwedge\nolimits_{l_1=1}^{s_{i,j}^{k}} c_{i,j}^{k,l_1} \wedge \neg c \wedge \\ &\quad \bigwedge\nolimits_{k_1=1,k_1\neq k}^{w_{i,j}} \neg(c_{i,j}^{k_1,1} \wedge \cdots \wedge c_{i,j}^{k_1,s_{i,j}^{k_1}})\Big\rangle_{O(i,j,k)\ \wedge\ c\in(L_{i,j}-L_{i,j}^{k})}\end{aligned}$$

for LIF, where $L_{i,j}$ is the list of all Boolean literals in $E_{i,j}$ and $L_{i,j}^{k} = \langle c\rangle_{c\in L_{i,j}\ \wedge\ (c\wedge c_{i,j}^{k,d}=\mathit{false}\ \forall d=1,2,\ldots,s_{i,j}^{k})} \oplus \langle c_{i,j}^{k,d}\rangle_{d=1,2,\ldots,s_{i,j}^{k}}$. According to Boolean specification-based testing, a Boolean literal whose negation is in a term cannot be inserted into that term. Here, the list of Boolean literals that cannot be inserted is extended to include those that cannot be true simultaneously with any literal in the term. Consider the expression $x > 30 \wedge y < 12 \vee x < 5 \wedge y < 20$. The predicate $x > 30$ cannot be inserted into the term $x < 5 \wedge y < 20$ because $x > 30$ and $x < 5$ cannot be true simultaneously. If the Boolean variables p_1, p_2, p_3, and p_4 represent $x > 30$, $y < 12$, $x < 5$, and $y < 20$, respectively, the expression is $p_1 \wedge p_2 \vee p_3 \wedge p_4$. It is possible to add p_1 to the second term because neither p_1 nor $\overline{p_1}$ occurs in the second term. If $L_{i,j} - L_{i,j}^{k} = \emptyset$, the corresponding constraints do not exist. Hence, the list for LIF can be empty. If a term contains only one Boolean literal (that is, $s_{i,j}^{k} = 1$), LOF for this literal is then taken as TOF.

The list of test case constraints derived from this formula for Table 3 is

$$\begin{aligned}\langle\ &\neg(y>1)\ \wedge\ x<1\ \wedge\ \neg(y>1\ \wedge\ x>31),\\ &y>1\ \wedge\ \neg(x<1)\ \wedge\ \neg(y>1\ \wedge\ x>31),\\ &\neg(y>1\ \wedge\ x<1)\ \wedge\ \neg(y>1)\ \wedge\ x>31,\\ &\neg(y>1\ \wedge\ x<1)\ \wedge\ y>1\ \wedge\ \neg(x>31),\\ &\neg(y>1)\ \wedge\ x\geq 1\ \wedge\ x\leq 31,\\ &y>1\ \wedge\ \neg(x\geq 1)\ \wedge\ x\leq 31,\\ &y>1\ \wedge\ x\geq 1\ \wedge\ \neg(x\leq 31),\\ &\neg(y\leq 1)\ \wedge\ x<1\ \wedge\ \neg(y\leq 1\ \wedge\ x>31),\\ &y\leq 1\ \wedge\ \neg(x<1)\ \wedge\ \neg(y\leq 1\ \wedge\ x>31),\\ &\neg(y\leq 1\ \wedge\ x<1)\ \wedge\ \neg(y\leq 1)\ \wedge\ x>31,\\ &\neg(y\leq 1\ \wedge\ x<1)\ \wedge\ y\leq 1\ \wedge\ \neg(x>31),\\ &\neg(y\leq 1)\ \wedge\ x\geq 1\ \wedge\ x\leq 31,\\ &y\leq 1\ \wedge\ \neg(x\geq 1)\ \wedge\ x\leq 31,\\ &y\leq 1\ \wedge\ x\geq 1\ \wedge\ \neg(x\leq 31)\ \rangle.\end{aligned}$$

The list for LIF is empty.

4 COMPARISON OF STRATEGIES

This section compares the subsumption relationships of the strategies on a mathematical basis. The comparison is based on the assumption that only one test case is generated from each test case constraint.

4.1 Notation

The following notation is used in this paper:

1. S : A testing strategy.
2. S^P : The partition strategy for tabular expressions.
3. S^D : Decision table-based testing.
4. S^B : The basic meaningful impact strategy.
5. S^F : Fault-based testing.
6. *SP* : The class of all specifications in a two-dimensional normal table.
7. *SPEC* : Any subset of *SP*.
8. *NDSP* : The subset of *SP* containing all the specifications with no duplicated evaluation expressions.
9. *DSP* : The subset of *SP* containing all the specifications with duplicated evaluation expressions.
10. *sp* : A specification.
11. $\mathit{STCC}(S, \mathit{SPEC})$: The lists of test case constraints derived from strategy S over a class of specifications *SPEC*.
12. $\mathit{stcc}(S, \mathit{sp})$: The list of test case constraints derived from strategy S for a specification *sp*.
13. $T(S, \mathit{sp})$: A test suite for specification *sp* derived from strategy S.
14. $\mathit{WT}(S, \mathit{sp})$: The set of all $T(S, \mathit{sp})$.

Clearly, $SP = NDSP \cup DSP$ and $NDSP \cap DSP = \emptyset$. The list $stcc(S, sp)$ can be taken as an instance of $STCC(S, SPEC)$ for some $sp \in SPEC$. Since $SPEC$ is a class of specifications, the test case constraints in $STCC(S, SPEC)$ are abstract and independent of any specification, while $stcc(S, sp)$ is a list of real test case constraints. It is unknown whether a constraint in $STCC(S, SPEC)$ exists or is equivalent to *false*. If a constraint in $STCC(S, SPEC)$ is equivalent to *false* for specification sp, it is removed from $stcc(S, sp)$. Given a specification sp, there can be numerous test suites that satisfy a testing criterion.

4.2 Definitions

The following definitions are given for the purpose of the comparison:

1. *Equivalence*
   a. A constraint c_1 is *equivalent* to another constraint c_2, denoted by $c_1 = c_2$, if each solution to c_1 is a solution to c_2 and vice versa.
   b. A list of constraints C_1 is *equivalent* to another list C_2, denoted by $C_1 = C_2$, if each constraint in C_1 has an equivalent constraint in C_2 and vice versa.
   c. S_1 is *equivalent* to S_2 over a specification sp, denoted by $S_1(sp) = S_2(sp)$, if $stcc(S_1, sp)$ is equivalent to $stcc(S_2, sp)$, that is, $stcc(S_1, sp) = stcc(S_2, sp)$.
   d. S_1 is *equivalent* to S_2 over a class of specifications $SPEC$, denoted by $S_1(SPEC) = S_2(SPEC)$, if $S_1(sp) = S_2(sp)$ for all $sp \in SPEC$.
2. *Subsumption*
   Testing strategy S_1 *subsumes* testing strategy S_2 over a specification sp, denoted by $S_1(sp) \succ S_2(sp)$, if for any $T(S_1, sp)$, $T(S_1, sp) \in WT(S_2, sp)$.
3. *Unconditional subsumption*
   Testing strategy S_1 *unconditionally subsumes* testing strategy S_2 over a class of specifications $SPEC$, denoted by $S_1(SPEC) \rhd\!\rhd S_2(SPEC)$, if the following conditions are satisfied:

   ***CUS*₁.** For any specification $sp \in SPEC$, $S_1(sp) \succ S_2(sp)$.

   ***CUS*₂.** For any specification $sp \in SPEC$, if $stcc(S_1, sp) = \emptyset$, $stcc(S_2, sp) = \emptyset$.

   The unconditional subsumption relationship is transitive. If S_1 unconditionally subsumes S_2 and S_2 unconditionally subsumes S_3 over a class of specifications $SPEC$, S_1 unconditionally subsumes S_3 since for all $sp \in SPEC$, $S_1(sp) \succ S_2(sp) \succ S_3(sp)$. If $stcc(S_1, sp) = \emptyset$ and $stcc(S_2, sp) = \emptyset$, then $stcc(S_3, sp) = \emptyset$. Consider the following example. Let p_1, p_2, p_3, and p_4 be Boolean literals. Suppose $STCC(S_1, SPEC) = \langle p_1 \wedge p_2, p_1 \wedge p_3 \rangle$ and $STCC(S_2, SPEC) = \langle p_1 \wedge p_2 \rangle$. Then, $stcc(S_1, sp) \supseteq stcc(S_2, sp)$ for any $sp \in SPEC$. Both CUS_1 and CUS_2 are satisfied. Hence, $S_1(SPEC) \rhd\!\rhd S_2(SPEC)$.
4. *Conditional subsumption*
   A test strategy S_1 *conditionally subsumes* another testing strategy S_2 over a class of specifications $SPEC$, denoted by $S_1(SPEC) \rhd S_2(SPEC)$, if the following conditions are satisfied:

   ***CCS*.** For any specification $sp \in SPEC$, $S_1(sp) \succ S_2(sp)$ and $S_1(sp) \neq S_2(sp)$ provided that some sublists of $STCC(S_1, SPEC)$ exist or some sublists of $STCC(S_2, SPEC)$ do not exist with respect to sp.

   Suppose $STCC(S_1, SPEC) = \langle p_1 \wedge p_2 \wedge p_4,\ p_1 \wedge p_3 \rangle$ and $STCC(S_2, SPEC) = \langle p_1 \wedge p_2 \rangle$. Then, $S_1(SPEC) \rhd S_2(SPEC)$. For any specification $sp \in SP$, $S_1(sp) \succ S_2(sp)$ provided that $\langle p_1 \wedge p_2 \wedge p_4 \rangle$ exists with respect to sp. There are two situations where a sub-suite of $STCC(S_1, SPEC)$ does not exist for $sp \in SPEC$:
   a. Some of the predicates (such as p_4) do not exist for sp.
   b. The actual constraint of $p_1 \wedge p_2 \wedge p_4$ with respect to sp is equivalent to *false*. For instance, if p_1 is $x > 31$, p_2 is $y < 10$, and p_4 is $x < 28$, the constraint $x > 31 \ \wedge\ y < 10 \ \wedge\ x < 28$ is always *false*.

   The subsumption relationships above are defined according to the concept of abstract test case constraints. As shown in the example, some testing strategies subsume others according to certain prerequisites.
5. *Incomparability*
   a. Two testing strategies S_1 and S_2 are *incomparable* over a specification sp, denoted by $S_1(sp) \sim S_2(sp)$, if S_1 does not subsume S_2 nor vice versa.
   b. Two testing strategies S_1 and S_2 are *incomparable* over a class of specifications $SPEC$, denoted by $S_1(SPEC) \sim S_2(SPEC)$, if S_1 does not conditionally or unconditionally subsume S_2, nor vice versa.

4.3 Comparison of the Testing Strategies

The comparison in this section assumes that there are no duplicated evaluation expressions in a table. The proofs of the theorems are given in Appendix D. Section 4.4 discusses tabular specifications with duplicated evaluation expressions.

Theorem 1: Decision table-based testing unconditionally subsumes the partition strategy for tabular expressions over $NDSP$, that is, $S^D(NDSP) \rhd\!\rhd S^P(NDSP)$.

It follows that S^D subsumes S^P over any sp in $NDSP$. If $w_{i,j} = 1$ for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$, $stcc(S^D, sp) = stcc(S^P, sp)$, that is, S^D and S^P are equivalent to each other over sp.

Theorem 2: The basic meaningful impact strategy unconditionally subsumes decision table-based testing over $NDSP$, that is, $S^B(NDSP) \rhd\!\rhd S^D(NDSP)$.

Following this theorem, for any sp in $NDSP$, $S^B(sp) \succ S^D(sp)$. Since decision table-based testing unconditionally subsumes the partition strategy for tabular expressions, the basic meaningful impact strategy unconditionally subsumes the partition strategy also.

According to Lemma 2, the second list in $STCC(S^B, NDSP)$ is never empty with respect to any $sp \in NDSP$. It does not mean, however, that decision table-based testing is never equivalent to the basic meaningful impact strategy for a specification in $NDSP$. Although $STCC(S^B, NDSP) \supset STCC(S^D, NDSP)$, it is possible that $stcc(S^B, sp) = stcc(S^D, sp)$. In $STCC(S^B, NDSP)$, data satisfying

$\neg c_{i,j}^{k,l} \ \wedge \ \bigwedge_{l_1=1,l_1\neq l}^{s_{i,j}^k} c_{i,j}^{k,l_1} \wedge \bigwedge_{k_1=1,k_1\neq k}^{w_{i,j}} \neg(c_{i,j}^{k_1,1} \wedge \cdots \wedge c_{i,j}^{k_1,s_{i,j}^{k_1}})$ evaluate the expression $T[1][i] \wedge T[2][j]$ to *false*. According to the definition of tabular expressions, there must exist i', j' ($i \neq i'$ or $j \neq j'$) such that the data evaluate $T[1][i'] \wedge T[2][j']$ to *true*. For example, S^B and S^D are equivalent over the specification in Table 7, where $T[1][1] \wedge T[2][1] = a > 3 \wedge b > 5$, $T[1][1] \wedge T[2][2] = a > 3 \wedge b \leq 5$, $T[1][2] \wedge T[2][1] = a \leq 3 \wedge b > 5$, and $T[2][1] \ \wedge \ T[2][2] = a \leq 3 \ \wedge \ b \leq 5$.

The lists of test case constraints are $\langle \ a > 3 \ \wedge \ b > 5$, $a > 3 \ \wedge \ b \leq 5$, $a \leq 3 \ \wedge \ b > 5$, $a \leq 3 \ \wedge \ b \leq 5 \ \rangle$ for S^D and $\langle a > 3 \ \wedge \ b > 5$, $a > 3 \ \wedge \ b \leq 5$, $a \leq 3 \ \wedge \ b > 5$, $a \leq 3 \ \wedge \ b \leq 5 \ \rangle \oplus \langle \neg(a > 3) \ \wedge \ b > 5$, $a > 3 \ \wedge \ \neg(b > 5)$, $\neg(a > 3) \ \wedge \ b \leq 5$, $a > 3 \ \wedge \ \neg(b \leq 5)$, $\neg(a \leq 3) \ \wedge \ b > 5$, $a \leq 3 \ \wedge \ \neg(b > 5)$, $\neg(a \leq 3) \ \wedge \ b \leq 5$, $a \leq 3 \ \wedge \ \neg(b \leq 5) \ \rangle$ for S^B. Since $a > 3$ = $\neg(a \leq 3)$ and $b > 5 = \neg(b \leq 5)$, the second list for S^B is equivalent to the first list. The strategies S^B and S^D are, therefore, equivalent over this specification.

TABLE 7
Example where $S^B(sp)$ = $S^D(sp)$

$T[2]$

	$b > 5$	$b \leq 5$
$a > 3$	$a + b$	$a - b$
$a \leq 3$	a/b	$a \times b$

$T[1]$ $T[0]$

Theorem 3: 1) Fault-based testing for the LOF and LIF classes of faults conditionally subsumes the basic meaningful impact strategy over *NDSP*, that is, $S^F(\mathit{NDSP}) \rhd S^B(\mathit{NDSP})$. 2) The basic meaningful impact strategy conditionally subsumes fault-based testing for the LOF and LIF classes of faults, that is, $S^B(\mathit{NDSP}) \rhd S^F(\mathit{NDSP})$.

For any specification $sp \in \mathit{SPEC}$, S^F subsumes S^B over sp only if there exists at least one LIF fault for every term in each evaluation condition; S^B subsumes S^F over sp only if there is no LIF fault for all the terms in all the evaluation conditions.

If two testing strategies S_1 and S_2 are not equivalent and S_1 unconditionally subsumes S_2, it is impossible that S_2 unconditionally subsumes S_1. However, if S_1 conditionally subsumes S_2, it is possible that S_2 conditionally subsumes S_1.

Theorem 4: Fault-based testing for the LOF and LIF classes of faults conditionally subsumes decision table-based testing over *NDSP*, that is, $S^F(\mathit{NDSP}) \rhd S^D(\mathit{NDSP})$.

Nevertheless, decision table-based testing does not conditionally subsume fault-based testing. Although $\mathit{stcc}(S^F, sp) = \mathit{stcc}(S^D, sp)$ for some $sp \in \mathit{NDSP}$ when some subsets of $S^F(\mathit{NDSP})$ do not exist, *CCS* is not satisfied.

For any specification $sp \in \mathit{SPEC}$, S^F subsumes S^D over sp only if there exists at least one LIF fault for every term in each evaluation condition.

Theorem 5: Fault-based testing for the LOF and LIF classes of faults conditionally subsumes the partition strategy over *NDSP*, that is, $S^F(\mathit{NDSP}) \rhd S^P(\mathit{NDSP})$.

For any specification $sp \in \mathit{SPEC}$, S^F subsumes S^P over sp only if at least one term has a LIF fault in each evaluation condition.

4.4 Duplication of Evaluation Expressions

Theorems 2, 3, and 4 are still true despite the presence of duplicated evaluation expressions in a table. This is due to the fact that decision table-based testing, the basic meaningful impact strategy, and fault-based testing are derived from the same equivalent conventional mathematical expressions. However, comparison results with the partition strategy are no longer valid because the number of test case constraints required for the partition strategy can be larger than that for any of the other three strategies. Furthermore, the partition strategy may subsume any of the other three test strategies over some specifications. Table 8 is an example where the partition strategy subsumes the other three strategies.

TABLE 8
An example where S^P is the strongest

$T[2]$

	$b > 5$	$b = 5$	$b < 5$
$a > 3$	$a + b$	$a + b$	$a + b$
$a = 3$	$a \times b$	$a \times b$	$a \times b$
$a < 3$	$a - b$	$a - b$	$a - b$

$T[1]$ $T[0]$

The equivalent conventional mathematical expression with combined evaluation conditions is

$$f(a,b) = \begin{cases} a + b & \text{if } \ a > 3 \\ a \times b & \text{if } \ a = 3 \\ a - b & \text{if } \ a < 3. \end{cases}$$

Since the three columns in the main grid are identical, it is equivalent to the specification in Table 9. However, a software engineer may use the form in Table 8 because of specific reasons such as compatibility with other tables in the same system.

TABLE 9
Another presentation of *Table 8*

$T[1]$	$T[0]$
$a > 3$	$a + b$
$a = 3$	$a \times b$
$a < 3$	$a - b$

Table 10 shows the respective lists of test case constraints for the four strategies.

TABLE 10
Test case constraints

S^P	$\langle a > 3 \ \wedge \ b > 5, a > 3 \ \wedge \ b = 5, a > 3 \ \wedge \ b < 5, a = 3 \ \wedge \ b > 5, a = 3 \ \wedge \ b = 5, a = 3 \ \wedge \ b < 5, a < 3 \ \wedge \ b > 5, a < 3 \ \wedge \ b = 5, a < 3 \ \wedge \ b < 5\rangle$
S^D	$\langle a > 3, a = 3, a < 3\rangle$
S^B	$\langle a > 3, a = 3, a < 3, a \leq 3, a \neq 3, a \geq 3\rangle$
S^F	$\langle a > 3, a = 3, a < 3\rangle$

5 EXPERIMENTAL STUDY

As we have demonstrated in the previous section, the subsumption relationship may depend on the features of the real specifications. Therefore, we further compare these testing strategies with respect to some real programs.

We use two applications in the experiment: *NextDate* and *Sales*. The specifications are in Appendix A. The *NextDate* application contains seven tables while the *Sales* application contains four. Three table types are used in these specifications: *normal* (N), *inverted* (I), and *tree-structured* (T). The expressions in the tables are not limited to nonduplicated expressions.

In the experiment, the testing strategies are compared in terms of their mutation scores. In theory, a *mutation score* is defined as the number of killed mutants divided by the number of all nonequivalent mutants with respect to a test suite. Since the scores in the experiment are collected only for the purpose of comparison, we do not separate the nonequivalent mutants from the equivalent ones. Hence the mutation scores are actually computed as the number of killed mutants divided by the number of all mutants with respect to a test suite. This does not affect the actual comparison results since the same denominator applies to all the strategies under study.

In the experiment, we use the mutation generator developed in our group [14] to automatically generate mutants of the programs. Table 11 lists the 20 mutation operators (syntactic changes to a program) implemented in the mutant generator. These mutation operators are extracted from [1], and mainly concern syntactic changes in statements, expressions, and brackets. The *coupling effect* [12] indicates that software engineers, in their multiple iterations during the design process, constantly narrow down the difference between what their programs currently look like and what they are intended to look like. It is typically more difficult to uncover faults in programs that are near completion as opposed to programs that are in earlier stages of development. Hence, in this experiment, every mutant is obtained by applying a single mutant operator per application.

In addition to the mutant generator, we also use a constraint solver, a test driver, and a data analyzer [14] in the experiment. The constraint solver *BoNus* is third-party software. It generates test cases from arithmetic constraints. The test driver reads the test cases, runs the original program and its mutants, and then compares the results. The data analyzer calculates the mutation scores and lists all the mutants that have passed the test data (either because these mutants are equivalent to the original programs, or because the test data fail to kill the mutants).

For every specification table, two test suites are derived from each testing strategy. In both suites, one test case is generated from each test case constraint. In the first suite, duplicated test cases are removed. In the second suite, duplicated test cases are not removed; instead, if a test case includes a value used in another test case, the value will be replaced with a different one if available. For example, $\langle month > 12 \ \wedge \ \neg(month < 1), \neg(month > 12) \ \wedge \ \neg(month < 1), \neg(month > 12) \ \wedge \ (month < 1), \neg(month > 12) \ \wedge \ \neg(month < 1)\rangle$ is a list of test case constraints for the *mError* specification. The second and the fourth constraints in the list are the same. If one test case is chosen for each constraint in this list, the test suite is $\langle month = 13,\ month = 1,\ month = 0,\ month = 1\rangle$. To create the first test suite, one of the entries "$month = 1$" is removed because it is duplicated. The resulting test suite is $\langle month = 13,\ month = 1,\ month = 0\rangle$. In constructing the second test suite, one "$month = 1$" is replaced by "$month = 2$" because the latter is another test case that satisfies the same test case constraint. Thus, the second test suite is $\langle month = 13,\ month = 1,\ month = 0,\ month = 2\rangle$.

TABLE 11
Mutation operators

Name	Definition
OCOR	Cast operator replacement or type replacement
SMVB	Move a brace up or down
SSOM	Exchange the sequence of the statements in the same level
SSDL	Delete a simple statement
SCBR	Replace "break" by "continue" or replace "continue" by "break"
SCBM	Remove "continue" or "break" to the outer or inner level
SICC	Insert semicolon after "if", "while", or "for"
SSCB	Delete or add the "break" in "switch" statement
EARA	Replace an arithmetic assignment operator "$+\ =$", "$-\ =$", "$=$", "$\&\ =$", "$<<=$", "$\mid\ =$", "$*\ =$" by another legal assignment operator
EORO	Replace a binary operator by another legal operator
EVRV	Replace a variable by another variable of the same type
ERRV	Replace a reference by a variable of the same type
EVRC	Replace a variable or a constant by a positive value, a negative value, and 0. If it is a string constant, replace it by a constant string and an empty string
EURU	Replace a unary operator by another unary operator
EADV	Add or delete a variable
EADP	Add or delete a pair of parenthesis in an arithmetic expression
EADU	Add or remove a unary operator
EACE	Add a positive constant and a negative constant to the end of an expression
ELCN	Negate the whole logical expression
EEAI	Exchange the index of an array with multiple dimensions

Tables 12 and 13 present the mutation scores obtained from the experiment. Grid $T[2]$ lists the program names, number of mutants, and the table type for each program; Grid $T[1]$ contains the strategy names; $T[0]$ gives the mutation scores along with the numbers of test cases in brackets.

We have the following observations from the experimental results:

1. For each testing strategy, there is no clear relationship between the number of test cases and the subsequent test effectiveness.

   We first compare Tables 12 and 13. One could expect the mutation scores in the second test suite to be higher. However, some mutation scores in Table 13 are lower than their counterparts in Table 12. For instance, the number of test cases for *dError* derived from S^B in the second test suite is almost twice that in the first test suite, but the mutation score is lower. We first use the simpler *nDate* example to explain the situation. The mutation score for *nDate* with S^P in Table 13 is 0.646. This is lower than the score of 0.722 in Table 12. The test suite for the mutation score 0.722 is $T_1 = \langle\langle year = 2081,\ month = 1,\ day = 1\rangle,\ \langle year =$

TABLE 12
Mutation scores: minimum sets

$T[2]$

	$dError$ (316, I)	$mError$ (52, N)	$mType$ (160, T)	$nDate$ (168, N)	$tDate$ (464, I)	$yError$ (52, N)	$yType$ (111, N)	$Bonus$ (145, N)	$Comm$ (333, I)	$Level$ (269, N)	$Price$ (222, N)
S^P	0.509 (6)	0.692 (2)	0.900 (5)	0.722 (2)	0.746 (6)	0.712 (2)	0.468 (2)	0.830 (6)	0.649 (6)	0.713 (6)	0.853 (9)
S^D	0.718 (11)	0.827 (3)	0.900 (5)	0.741 (4)	0.746 (8)	0.769 (3)	0.847 (4)	0.830 (6)	0.703 (6)	0.713 (6)	0.853 (9)
S^B	0.785 (18)	0.827 (3)	0.900 (5)	0.741 (4)	0.756 (13)	0.769 (3)	0.847 (4)	0.830 (6)	0.763 (7)	0.713 (8)	0.853 (9)
S^F	0.772 (17)	0.596 (2)	0.900 (5)	0.741 (4)	0.750 (12)	0.519 (2)	0.847 (4)	0.830 (6)	0.763 (8)	0.713 (6)	0.853 (9)

$T[1]$ $T[0]$

TABLE 13
Mutation scores

$T[2]$

	$dError$ (316, I)	$mError$ (52, N)	$mType$ (160, T)	$nDate$ (168, N)	$tDate$ (464, I)	$yError$ (52, N)	$yType$ (111, N)	$Bonus$ (145, N)	$Comm$ (333, I)	$Level$ (269, N)	$Price$ (222, N)
S^P	0.563 (6)	0.692 (2)	0.900 (5)	0.646 (2)	0.746 (6)	0.712 (2)	0.468 (2)	0.803 (6)	0.796 (6)	0.737 (6)	0.858 (9)
S^D	0.706 (11)	0.827 (3)	0.900 (5)	0.729 (4)	0.746 (8)	0.769 (3)	0.847 (4)	0.803 (6)	0.796 (6)	0.737 (6)	0.858 (9)
S^B	0.753 (39)	0.846 (7)	0.906 (12)	0.747 (11)	0.765 (25)	0.788 (7)	0.847 (10)	0.844 (18)	0.826 (18)	0.749 (17)	0.862 (31)
S^F	0.753 (33)	0.615 (4)	0.906 (9)	0.741 (9)	0.761 (19)	0.519 (4)	0.847 (6)	0.844 (12)	0.826 (15)	0.749 (15)	0.862 (21)

$T[1]$ $T[0]$

1812, *month* $= 1$, *day* $= 1\rangle\rangle$, while the test suite for the mutation score 0.646 is $T_2 = \langle\langle$*year* $= 2081$, *month* $= 1$, *day* $= 1\rangle$, $\langle$*year* $= 1812$, *month* $= 2$, *day* $= 2\rangle\rangle$. The second test case in T_2 is different from the second test case in T_1. In T_2, *day* and *month* could both be assigned the value of 1 but were given the value of 2 so that the values of *day* and *month* would not be repeated. The consequence is that assigning different values may create less effective test cases. When compared with $\langle$*year* $= 1812$, *month* $= 1$, *day* $= 1\rangle$, the test case $\langle$*year* $= 1812$, *month* $= 2$, *day* $= 2\rangle$ is less powerful in revealing faults in the *nDate* program. Thus, even though there is no difference in the numbers of test cases between the two test suites, the above discussion helps explain why the second test suite produces a lower mutation score in the *dError* program.

2. S^B is the strongest among the four strategies under study.

As proven in Section 4.3, S^B unconditionally subsumes S^P and S^D, and hence it is not surprising that the mutation scores for this strategy are higher than the scores for S^P and S^D. We have shown that S^B and S^F conditionally subsume each other; nevertheless, S^B always has higher mutation scores in the experiment. In any case, it must also be noted that, although S^B is the most effective among the strategies, the number of test cases is also the highest. When selecting a test strategy, a trade-off has to be made between effectiveness and cost if the testing resource is limited.

3. S^P can be more effective than S^F in certain circumstances.

S^F has higher mutation scores for most programs, but there are two exceptions: *mError* and *yError*. This result is not contradictory to the proof because S^F does not unconditionally subsume S^P. Both the *mError* and *yError* programs have no test case constraints for LIF faults derived from S^F and the test cases generated for LOF are less powerful than the test cases generated for S^P in these two programs.

4. The mutation scores depend on constraint solvers.

Our intuitive understanding was that the mutation scores for the *mError* and *yError* programs should be the same since they have similar specifications and implementations. The results are surprising in that they have different mutation scores. Further study reveals that the constraint solving algorithm causes the different scores. *BoNus* [14] is the constraint solver used in the toolset developed in our group. The test suites derived from S^D for *yError* and *mError* are $\langle 2081, 0, 1812\rangle$ and $\langle 13, 0, 1\rangle$, respectively. The values 2081 and 1812 for *yError* correspond to the values 13 and 1, respectively. The value 0 in the test suite for *yError* is derived from the constraint "*year* < 1812", while the same value in the test suite for *mError* is from the constraint "*month* < 1". In other words, the *BoNus* algorithm gives 0 for both "*year* < 1812" and "*month* < 1". For a program expression such as "*month* $< 1 \mid\mid$ *month* > 12" (written in C), the test case 0 is very effective in detecting common faults, while for an expression like "*year* $< 1812 \mid\mid$ *year* > 2080", the test case 0 is less effective. When the test case is changed from 0 to 1811 for "*year* < 1812", the mutation score increases.

5. The mutation scores depend on the mutants.

Mutation scores always depend on the mutants for a single program. However, when two programs are compared, the generated mutants can also affect the comparison results. Consider the *mError* and the *yError* examples again. Using the S^P strategy, the test suites are $\langle 13, 1\rangle$ for *mError* and $\langle 2081, 1812\rangle$ for *yError*. Intuitively, there should not be any difference between the mutation scores using these two test suites since they involve similar programs and similar test cases. However, the mutation score for *yError* is higher than that for *mError*. This is caused by the generation of the mutants. The *EVRC* mutation operator requires that a constant in the source code be changed to a positive constant, a negative constant, and 0. The mutation generator uses the number 3 as the positive constant to replace a constant in the source code[1]. Hence, there is a mutant for *mError*, where the expression $month > 12 \mid\mid month < 1$ is changed to $month > 3 \mid\mid month < 1$; similarly, there is a mutant for *yError*, where the expression $year > 2080 \mid\mid year < 1812$ is changed to $year > 3 \mid\mid year < 1812$. Then, both test cases for *mError* cannot distinguish this mutant from the original program while the test case 1812 for *yError* can distinguish $year > 3 \mid\mid year < 1812$ from $year > 2080 \mid\mid month < 1812$.

6. Many terms in the expressions have no LIF faults.

   It is noted that the number of test cases for S^F is less than the number of test cases for S^B in some programs. For some specifications, no LIF faults exist for any term in an expression. For some of the terms having LIF faults, no test cases can distinguish the expression with LIF faults from the original one because these two expressions are equivalent.

With regard to the above observations, test effectiveness depends on many factors: testing strategies, specifications, faults, constraint solvers, and so on. For the same testing strategy, if we apply it to a different specification, or to the same specification with a different implementation, or if we use a different method to generate test cases from the test case constraints, we may obtain different results. For instance, S^B unconditionally subsumes S^P and S^D. These relationships are reflected in the experimental results as expected. On the other hand, S^B and S^F conditionally subsume each other, but S^F did not show a higher mutation score in any program throughout the experiment. Although this result does not contradict the proofs, further discussion is required.

If S^F has higher mutation scores than S^B, testers should select S^F. Since this is not the case, let us examine the situation further. In this paper, S^F covers two fault classes, namely LOF and LIF. LOF is one of the fault classes that can also be detected by S^B. Hence, LOF faults should not cause S^F to be less effective. Suppose we conduct a test for detecting LIF faults only. Let us concentrate on two major factors — specifications and faults — and ignore the less important factor of constraint solvers. Two possibilities should be taken into account in terms of these two factors: 1) the possibility for LIF faults to exist in a specification with available test cases, and 2) the possibility for a faulty program to exist to reflect the faulty specification with LIF faults. The experimental results show that both possibilities are low in terms of fault-based testing for LIF, and hence it is clearly better to select S^B. The same analysis can be done for the LRF class of faults, which is also in the fault class hierarchy diagram. According to the definitions of LIF and LRF in [27], if a Boolean literal cannot be inserted into a term (LIF), it cannot be used to replace any literal in that term (LRF). It is possible, however, that both LIF and LRF faults exist but there are no test cases available for LIF faults. This situation exists in some programs used in the experiment. The test cases for LRF either do not exist or are duplicated with other test cases in the same test suite. As a result, the scores for S^F in the experiment cannot be improved by considering LRF faults.

An open area of discussion in this comparison is the choice between *MUMCUT* [8] and the basic meaningful impact strategy. The *MUMCUT* strategy can cover all fault types in the hierarchy diagram of fault classes, and yet requires significantly more test cases than the basic meaningful impact strategy [24]. The detection of the LIF and LRF fault classes is where the *MUMCUT* strategy has a clear advantage over the basic meaningful impact strategy [8]. If we use both S^B and S^F, they cover the entire hierarchy diagram with the only exception of LRF. We combine the test cases for S^B and S^F to test the programs in the experiment, but find the mutation scores to be the same as those for the basic meaningful impact strategy. Even though we do not include the *MUMCUT* strategy in the comparison, the effectiveness of this strategy can be approximated by the effectiveness of S^B and S^F and the previous analysis of LIF and LRF faults. This holds true until it is shown that *MUMCUT* detects other fault types that cannot be ignored. The consideration of LIF and LRF faults does not improve the test effectiveness in the experiment. In any case, it is an open research question to uncover how the number of infeasible LIF and LRF faults or the consideration of LIF and LRF faults can affect mutation scores. It is also unclear whether the *MUMCUT* strategy can detect other important fault types not included in the hierarchy diagram of fault classes to justify the cost of generating significantly more test cases. These are issues that need further research and empirical study.

6 CONCLUSION

Four testing strategies have been compared on a mathematical basis through a precisely defined subsumption relationship. For a two-dimensional normal table without duplicated evaluation expressions, decision table-based testing unconditionally subsumes the partition strategy. The basic meaningful impact strategy unconditionally subsumes decision table-based testing and conditionally subsumes fault-based testing. On the other hand, fault-based testing conditionally subsumes all the other three strategies. For two-dimensional normal tables, duplicated evaluation expressions have no effect on the subsumption relationship among decision table-based testing, the basic meaningful

---

[1] If the constant happens to be 3, the generator uses the number 17 to replace this constant.

impact strategy, and fault-based testing. However, the subsumption relationship with respect to the partition strategy is affected. The partition strategy subsumes any of the other three testing strategies for some specifications.

We have also compared these strategies using real programs where the table types are not limited to *normal*, and the expressions can either be duplicated or nonduplicated. The experiment shows that the basic meaningful impact strategy is the strongest while the partition strategy is the weakest in most cases. Although fault-based testing conditionally subsumes the partition strategy, it can be weaker than partition testing in certain circumstances. The experimental study also shows that the constraint solving algorithm can affect the effectiveness of a testing strategy. The theoretical proofs and the experimental study together provide testers with useful information on how to choose testing strategies and generate test data from the test case constraints. A summary of the comparison is shown in Tables 14 and 15. Incidentally, the summary is presented in the format of normal tables.

TABLE 14
Subsumption relationships (NDSP)

	S^P	S^D	S^B	S^F
S^P	$=$	$\lhd\lhd$	$\lhd\lhd$	$\lhd$
S^D	$\rhd\rhd$	$=$	$\lhd\lhd$	$\lhd$
S^B	$\rhd\rhd$	$\rhd\rhd$	$=$	$\rhd$
S^F	$\rhd$	$\rhd$	$\rhd$	$=$

TABLE 15
Subsumption relationships (DSP)

	S^P	S^D	S^B	S^F
S^P	$=$	$\sim$	$\sim$	$\sim$
S^D	$\sim$	$=$	$\lhd\lhd$	$\lhd$
S^B	$\sim$	$\rhd\rhd$	$=$	$\rhd$
S^F	$\sim$	$\rhd$	$\rhd$	$=$

The symbols in cell (S_1, S_2) indicate the subsumption relationship between S_1 and S_2. For example, the "$\rhd$" symbol in (S_1, S_2) means $S_1 \rhd S_2$. We have also introduced two more symbols: "$\lhd\lhd$" and "$\lhd$". The "$\lhd\lhd$" symbol in (S_1, S_2) means $S_2 \rhd\rhd S_1$ while the "$\lhd$" symbol in (S_1, S_2) means $S_2 \rhd S_1$. Hence, if the symbol in cell (S_1, S_2) is "$\rhd\rhd$", the symbol in cell (S_2, S_1) must be "$\lhd\lhd$".

In this paper, only two-dimensional normal tables are discussed. The comparison results for one-dimensional and higher dimensional normal tables are exactly the same as those for two-dimensional normal tables. For other table types, we note that, under the concept of combined evaluation conditions, the equivalent conventional mathematical expressions do not depend on table types. Therefore, the subsumption relationships among decision table-based testing, the basic meaningful impact strategy, and fault-based testing are not influenced, but the results related to the partition strategy for tabular expressions are affected.

APPENDIX A
TABULAR SPECIFICATION EXAMPLES

A.1 Example 1: *NextDate*

NextDate (Fig. 2) is an example of a specification in tabular expressions. The program computes the next date according to the input current date. It performs the following functions.

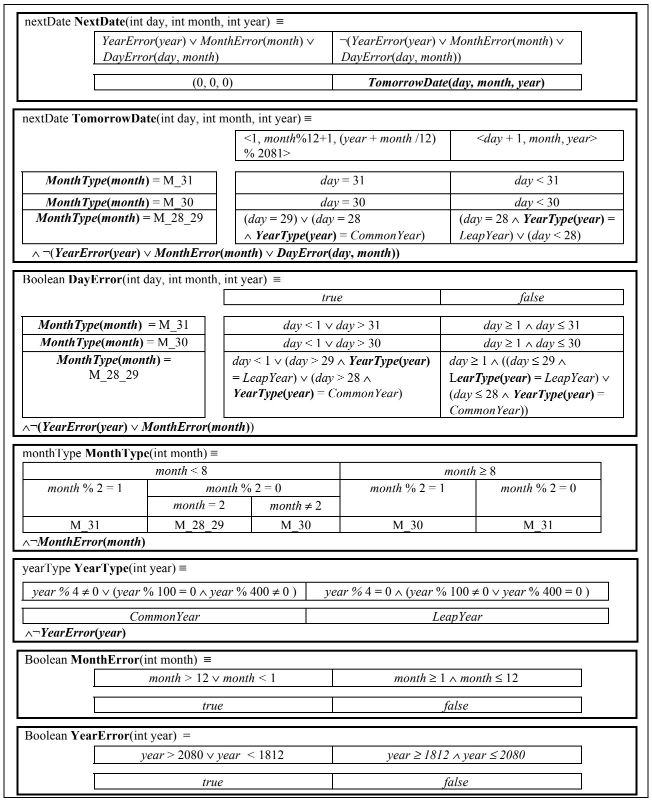
1. Check the validity of the input date. The input $\langle year, month, day \rangle$ is not valid when any of the following is satisfied:
   a. *year* is outside the range of 1812 to 2080;
   b. *month* is outside the range of 1 to 12;
   c. *day* is outside the range of 1 to 31 when *month* is 1, 3, 5, 7, 8, 10, or 12;
   d. *day* is outside the range of 1 to 30 when *month* is 4, 6, 9, or 11;
   e. *day* is outside the range of 1 to 28 when *month* is 2 and *year* is not a leap year;
   f. *day* is outside the range of 1 to 29 when *month* is 2 and *year* is a leap year.
2. Calculate the next date. If the current date is not valid, set $day = 0$, $month = 0$, and $year = 0$; otherwise, the next date is calculated according to the following rules:
   a. If *day* is not the last date of *month*, add 1 to *day*.
   b. If *day* is the last date of *month*, but *month* is not 12, set $day = 1$ and add 1 to *month*.
   c. If *day* is 31 and *month* is 12, set $day = 1$ and $month = 1$, and add 1 to *year*.

In Fig. 2, *DayError* and *TomorrowDate* are in inverted tables, *MonthType* is in a tree-structured table where the last row contains evaluation expressions, and all the others are normal tables. The normal tables in this example are all in one-dimension, that is, there are only two grids: $T[1]$ and $T[0]$. A function occurring in a cell can be a table itself. For instance, the *MonthType* function in $T[1]$ of the *NextDate* table is defined by a table also.

A.2 Example 2: *Sales*

This program calculates the promotion levels for a salesperson according to the number of health food products the salesperson has sold. There are three kinds of products: Vitamin A, Vitamin C, and Vitamin E. The respective prices for Vitamins A, C, and E are 20 euros, 26 euros, and 32 euros per bottle when the quantity is not more than 30 bottles; 18 euros, 24 euros, and 30 euros per bottle when the quantity is above 30 bottles but not more than 60; and 16 euros, 22 euros, and 28 euros per bottle when the quantity is beyond 60 bottles.

A salesperson receives commission for the sold products. If the salesperson is not in Europe, the commission is 10, 15, or 20 percent of the sales amount when the amount is not more than 3,000 euros, above 3,000 euros but not more than 4,800 euros, or beyond 4,800 euros, respectively; if the salesperson is in Europe, the commission is 10, 15, or 20 percent of the sales amount when the amount is not more than 2,800 euros, above 2,800 euros but not more than 4,500 euros, or beyond 4,500 euros, respectively.



nextDate **NextDate**(int day, int month, int year) ≡

YearError(*year*) ∨ *MonthError*(*month*) ∨ *DayError*(*day*, *month*)	¬(*YearError*(*year*) ∨ *MonthError*(*month*) ∨ *DayError*(*day*, *month*))
(0, 0, 0)	***TomorrowDate*(*day*, *month*, *year*)**

nextDate **TomorrowDate**(int day, int month, int year) ≡

	<1, *month*%12+1, (*year* + *month* /12) % 2081>	<*day* + 1, *month*, *year*>
***MonthType*(*month*)** = M_31	*day* = 31	*day* < 31
***MonthType*(*month*)** = M_30	*day* = 30	*day* < 30
***MonthType*(*month*)** = M_28_29	(*day* = 29) ∨ (*day* = 28 ∧ ***YearType*(*year*)** = *CommonYear*)	(*day* = 28 ∧ ***YearType*(*year*)** = *LeapYear*) ∨ (*day* < 28)

∧ ¬(***YearError*(*year*)** ∨ ***MonthError*(*month*)** ∨ ***DayError*(*day*, *month*)**)

Boolean **DayError**(int day, int month, int year) ≡

	true	*false*
***MonthType*(*month*)** = M_31	*day* < 1 ∨ *day* > 31	*day* ≥ 1 ∧ *day* ≤ 31
***MonthType*(*month*)** = M_30	*day* < 1 ∨ *day* > 30	*day* ≥ 1 ∧ *day* ≤ 30
***MonthType*(*month*)** = M_28_29	*day* < 1 ∨ (*day* > 29 ∧ ***YearType*(*year*)** = *LeapYear*) ∨ (*day* > 28 ∧ ***YearType*(*year*)** = *CommonYear*)	*day* ≥ 1 ∧ ((*day* ≤ 29 ∧ L***earType*(*year*)** = *LeapYear*) ∨ (*day* ≤ 28 ∧ ***YearType*(*year*)** = *CommonYear*))

∧¬(***YearError*(*year*)** ∨ ***MonthError*(*month*)**)

monthType **MonthType**(int month) ≡

month < 8			*month* ≥ 8	
month % 2 = 1	*month* % 2 = 0		*month* % 2 = 1	*month* % 2 = 0
	month = 2	*month* ≠ 2		
M_31	M_28_29	M_30	M_30	M_31

∧¬***MonthError*(*month*)**

yearType **YearType**(int year) ≡

year % 4 ≠ 0 ∨ (*year* % 100 = 0 ∧ *year* % 400 ≠ 0)	*year* % 4 = 0 ∧ (*year* % 100 ≠ 0 ∨ *year* % 400 = 0)
CommonYear	*LeapYear*

∧¬***YearError*(*year*)**

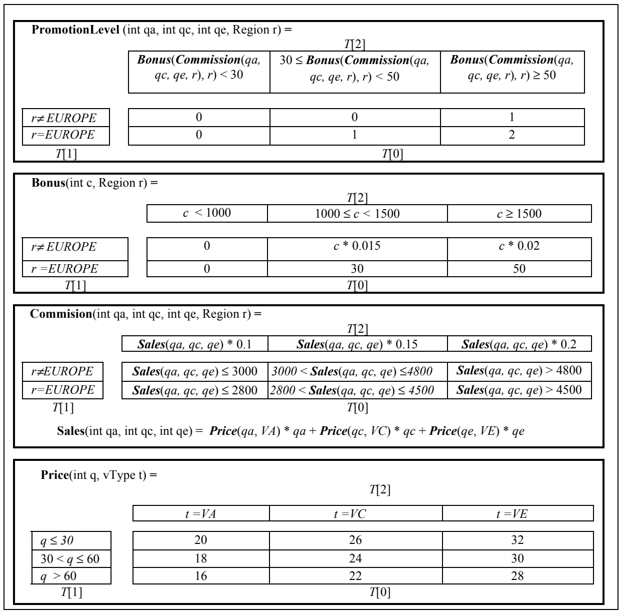
Boolean **MonthError**(int month) ≡

month > 12 ∨ *month* < 1	*month* ≥ 1 ∧ *month* ≤ 12
true	*false*

Boolean **YearError**(int year) =

year > 2080 ∨ *year* < 1812	*year ≥ 1812 ∧ year ≤ 2080*
true	*false*

Fig. 2. Specification of *NextDate* in tabular expressions



PromotionLevel (int qa, int qc, int qe, Region r) =

T[2]

	***Bonus*(*Commission*(*qa*, *qc*, *qe*, *r*), *r*) < 30**	30 ≤ ***Bonus*(*Commission*(*qa*, *qc*, *qe*, *r*), *r*) < 50**	***Bonus*(*Commission*(*qa*, *qc*, *qe*, *r*), *r*) ≥ 50**
r≠ EUROPE	0	0	1
r=EUROPE	0	1	2

T[1] *T*[0]

Bonus(int c, Region r) =

T[2]

	c < 1000	1000 ≤ *c* < 1500	*c* ≥ 1500
r≠ EUROPE	0	*c* * 0.015	*c* * 0.02
r =EUROPE	0	30	50

T[1] *T*[0]

Commision(int qa, int qc, int qe, Region r) =

T[2]

	Sales(*qa, qc, qe*) * 0.1	***Sales***(*qa, qc, qe*) * 0.15	***Sales***(*qa, qc, qe*) * 0.2
r≠EUROPE	***Sales***(*qa, qc, qe*) ≤ 3000	*3000 <* ***Sales***(*qa, qc, qe*) *≤4800*	***Sales***(*qa, qc, qe*) > 4800
r=EUROPE	***Sales***(*qa, qc, qe*) ≤ 2800	*2800 <* ***Sales***(*qa, qc, qe*) *≤ 4500*	***Sales***(*qa, qc, qe*) > 4500

T[1] *T*[0]

Sales(int qa, int qc, int qe) = ***Price***(*qa*, *VA*) * *qa* + ***Price***(*qc*, *VC*) * *qc* + ***Price***(*qe*, *VE*) * *qe*

Price(int q, vType t) =

T[2]

	t =VA	*t =VC*	*t =VE*
q ≤ 30	20	26	32
30 < *q* ≤ 60	18	24	30
q > 60	16	22	28

T[1] *T*[0]

Fig. 3. Specification of *Sales* in tabular expressions

The salesperson's bonus is then calculated to decide his/her promotion level. There is no bonus if the commission is below 1,000 euros. If the commission is not less than 1,000 euros but below 1,500 euros, the number of bonus points will be 1.5 percent of the commission (for instance, 1,000 euros in commission translates to 15 points) for a salesperson in Europe and 30 points for a salesperson outside Europe. If the commission is not less than 1,500 euros, the number of bonus points will be 2 percent of the commission for a salesperson in Europe and 50 points for a salesperson outside Europe. If the bonus reaches 50 points, a salesperson can be promoted by two levels in Europe and one level outside Europe. If the bonus reaches 30 points but is below 50, a salesperson can be promoted by one level in Europe.

In Fig. 3, the specification consists of four tables: *Price*, *Bonus*, and *Level*, and *Commission*. *Commission* is an inverted table while the others are normal tables.

APPENDIX B
TABLE TRANSFORMATION EXAMPLES

Tables 16 and 17 show, respectively, a normal table and a tree-structured table transformed from the inverted table in Table 1. To save space, d, m, y, C and L are used to represent *day*, *month*, *year*, *Common*, and *Leap*, respectively.

APPENDIX C
APPLICATION OF DECISION TABLE-BASED TESTING TO THE *DayError* EXAMPLE

For the *DayError* example, the DNF of the combined evaluation condition that corresponds to *true* is

$$(\mathit{mType}(m) = M\_31 \wedge d < 1) \vee (\mathit{mType}(m) = M\_31 \wedge d > 31) \vee (\mathit{mType}(m) = M\_30 \wedge d < 1) \vee (\mathit{mType}(m) = M\_30 \wedge d > 30) \vee (\mathit{mType}(m) = M\_28\_29 \wedge d < 1) \vee (\mathit{mType}(m) = M\_28\_29 \wedge d > 29 \wedge \mathit{yType}(y) = \mathit{LYear}) \vee (\mathit{mType}(m) = M\_28\_29 \wedge d > 28 \wedge \mathit{yType}(y) = \mathit{CYear})$$

The DNF form of the combined evaluation condition that corresponds to *false* is

$$(\mathit{mType}(m) = M\_31 \wedge d \geq 1 \wedge d \leq 31) \vee (\mathit{mType}(m) = M\_30 \wedge d \geq 1 \wedge d \leq 30) \vee (\mathit{mType}(m) = M\_28\_29 \wedge d \geq 1 \wedge d \leq 29 \wedge \mathit{yType}(y) = \mathit{LYear}) \vee (\mathit{mType}(m) = M\_28\_29 \wedge d \geq 1 \wedge d \leq 28 \wedge \mathit{yType}(y) = \mathit{CYear})$$

TABLE 16
DayError in normal table

$\mathit{DayError}(d, m, y) \equiv$

$(d < 1 \vee d > 31) \wedge \mathit{mType}(m) = M\_31$	$d \geq 1 \wedge d \leq 31 \wedge \mathit{mType}(m) = M\_31$	$(d < 1 \vee d > 30) \wedge \mathit{mType}(m) = M\_30$	$d \geq 1 \wedge d \leq 30 \wedge \mathit{mType}(m) = M\_30$	$(d < 1 \vee (d > 29 \wedge \mathit{yType}(y) = \mathit{LYear}) \vee (d > 28 \wedge \mathit{yType}(y) = \mathit{CYear})) \wedge \mathit{mType}(m) = M\_28\_29$	$d \geq 1 \wedge ((d \leq 29 \wedge \mathit{yType}(y) = \mathit{LYear}) \vee (d \leq 28 \wedge \mathit{yType}(y) = \mathit{CYear})) \wedge \mathit{mType}(m) = M\_28\_29$
true	*false*	*true*	*false*	*true*	*false*

TABLE 17
DayError in tree-structured table

$\mathit{DayError}(d, m, y) \equiv$

$\mathit{mType}(m) = M\_31$		$\mathit{mType}(m) = M\_30$		$\mathit{mType}(m) = M\_28\_29$	
$d < 1 \vee d > 31$	$d \geq 1 \wedge d \leq 31$	$d < 1 \vee d > 30$	$d \geq 1 \wedge d \leq 30$	$d < 1 \vee (d > 29 \wedge \mathit{yType}(y) = \mathit{LeapYear}) \vee (d > 28 \wedge \mathit{yType}(y) = \mathit{CYear})$	$d \geq 1 \wedge ((d \leq 29 \wedge \mathit{yType}(y) = \mathit{LeapYear}) \vee (d \leq 28 \wedge \mathit{yType}(y) = \mathit{CYear})$
true	*false*	*true*	*false*	*true*	*false*

The list of test case constraints for the decision table-based testing is

⟨ // Derived from the combined evaluation condition corresponding to *true*
$(\mathit{mType}(m) = M\_31 \wedge d < 1) \wedge \neg(\mathit{mType}(m) = M\_31 \wedge d > 31) \wedge \neg(\mathit{mType}(m) = M\_30 \wedge d < 1) \wedge \neg(\mathit{mType}(m) = M\_30 \wedge d > 30) \wedge \neg(\mathit{mType}(m) = M\_28\_29 \wedge d < 1) \wedge \neg(\mathit{mType}(m) = M\_28\_29 \wedge d > 29 \wedge \mathit{yType}(y) = \mathit{LYear}) \wedge \neg(\mathit{mType}(m) = M\_28\_29 \wedge d > 28 \wedge \mathit{yType}(y) = \mathit{CYear})$,
$\neg(\mathit{mType}(m) = M\_31 \wedge d < 1) \wedge (\mathit{mType}(m) = M\_31 \wedge d > 31) \wedge \neg(\mathit{mType}(m) = M\_30 \wedge d < 1) \wedge \neg(\mathit{mType}(m) = M\_30 \wedge d > 30) \wedge \neg(\mathit{mType}(m) = M\_28\_29 \wedge d < 1) \wedge \neg(\mathit{mType}(m) = M\_28\_29 \wedge d > 29 \wedge \mathit{yType}(y) = \mathit{LYear}) \wedge \neg(\mathit{mType}(m) = M\_28\_29 \wedge d > 28 \wedge \mathit{yType}(y) = \mathit{CYear})$,
$\neg(\mathit{mType}(m) = M\_31 \wedge d < 1) \wedge \neg(\mathit{mType}(m) = M\_31 \wedge d > 31) \wedge (\mathit{mType}(m) = M\_30 \wedge d < 1) \wedge \neg(\mathit{mType}(m) = M\_30 \wedge d > 30) \wedge \neg(\mathit{mType}(m) = M\_28\_29 \wedge d < 1) \wedge \neg(\mathit{mType}(m) = M\_28\_29 \wedge d > 29 \wedge \mathit{yType}(y) = \mathit{LYear}) \wedge \neg(\mathit{mType}(m) = M\_28\_29 \wedge d > 28 \wedge \mathit{yType}(y) = \mathit{CYear})$,
$\neg(\mathit{mType}(m) = M\_31 \wedge d < 1) \wedge \neg(\mathit{mType}(m) = M\_31 \wedge d > 31) \wedge \neg(\mathit{mType}(m) = M\_30 \wedge d < 1) \wedge (\mathit{mType}(m) = M\_30 \wedge d > 30) \wedge \neg(\mathit{mType}(m) = M\_28\_29 \wedge d < 1) \wedge \neg(\mathit{mType}(m) = M\_28\_29 \wedge d > 29 \wedge \mathit{yType}(y) = \mathit{LYear}) \wedge \neg(\mathit{mType}(m) = M\_28\_29 \wedge d > 28 \wedge \mathit{yType}(y) = \mathit{CYear})$,
$\neg(\mathit{mType}(m) = M\_31 \wedge d < 1) \wedge \neg(\mathit{mType}(m) = M\_31 \wedge d > 31) \wedge \neg(\mathit{mType}(m) = M\_30 \wedge d < 1) \wedge \neg(\mathit{mType}(m) = M\_30 \wedge d > 30) \wedge (\mathit{mType}(m) = M\_28\_29 \wedge d < 1) \wedge \neg(\mathit{mType}(m) = M\_28\_29 \wedge d > 29 \wedge \mathit{yType}(y) = \mathit{LYear}) \wedge \neg(\mathit{mType}(m) = M\_28\_29 \wedge d > 28 \wedge \mathit{yType}(y) = \mathit{CYear})$,
$\neg(\mathit{mType}(m) = M\_31 \wedge d < 1) \wedge \neg(\mathit{mType}(m) = M\_31 \wedge d > 31) \wedge \neg(\mathit{mType}(m) = M\_30 \wedge d < 1) \wedge \neg(\mathit{mType}(m) = M\_30 \wedge d > 30) \wedge \neg(\mathit{mType}(m) = M\_28\_29 \wedge d < 1) \wedge (\mathit{mType}(m) = M\_28\_29 \wedge d > 29 \wedge \mathit{yType}(y) = \mathit{LYear}) \wedge \neg(\mathit{mType}(m) = M\_28\_29 \wedge d > 28 \wedge \mathit{yType}(y) = \mathit{CYear})$,
$\neg(\mathit{mType}(m) = M\_31 \wedge d < 1) \wedge \neg(\mathit{mType}(m) = M\_31 \wedge d > 31) \wedge \neg(\mathit{mType}(m) = M\_30 \wedge d < 1) \wedge \neg(\mathit{mType}(m) = M\_30 \wedge d > 30) \wedge \neg(\mathit{mType}(m) = M\_28\_29 \wedge d < 1) \wedge \neg(\mathit{mType}(m) = M\_28\_29 \wedge d > 29 \wedge \mathit{yType}(y) = \mathit{LYear}) \wedge (\mathit{mType}(m) = M\_28\_29 \wedge d > 28 \wedge \mathit{yType}(y) = \mathit{CYear})$,
//Derived from the combined evaluation condition corresponding to *false*
$(\mathit{mType}(m) = M\_31 \wedge d \geq 1 \wedge d \leq 31) \wedge \neg(\mathit{mType}(m) = M\_30 \wedge d \geq 1 \wedge d \leq 30) \wedge \neg(\mathit{mType}(m) = M\_28\_29 \wedge d \geq 1 \wedge d \leq 29 \wedge \mathit{yType}(y) = \mathit{LYear}) \wedge \neg(\mathit{mType}(m) = M\_28\_29 \wedge d \geq 1 \wedge d \leq 28 \wedge \mathit{yType}(y) = \mathit{CYear})$,
$\neg(\mathit{mType}(m) = M\_31 \wedge d \geq 1 \wedge d \leq 31) \wedge (\mathit{mType}(m) = M\_30 \wedge d \geq 1 \wedge d \leq 30) \wedge \neg(\mathit{mType}(m) = M\_28\_29 \wedge d \geq 1 \wedge d \leq 29 \wedge \mathit{yType}(y) = \mathit{LYear}) \wedge \neg(\mathit{mType}(m) = M\_28\_29 \wedge d \geq 1 \wedge d \leq 28 \wedge \mathit{yType}(y) = \mathit{CYear})$,
$\neg(\mathit{mType}(m) = M\_31 \wedge d \geq 1 \wedge d \leq 31) \wedge \neg(\mathit{mType}(m) = M\_30 \wedge d \geq 1 \wedge d \leq 30) \wedge (\mathit{mType}(m) = M\_28\_29 \wedge d \geq 1 \wedge d \leq 29 \wedge \mathit{yType}(y) = \mathit{LYear}) \wedge \neg(\mathit{mType}(m) = M\_28\_29 \wedge d \geq 1 \wedge d \leq 28 \wedge \mathit{yType}(y) = \mathit{CYear})$,
$\neg(\mathit{mType}(m) = M\_31 \wedge d \geq 1 \wedge d \leq 31) \wedge \neg(\mathit{mType}(m) = M\_30 \wedge d \geq 1 \wedge d \leq 30) \wedge \neg(\mathit{mType}(m) = M\_28\_29 \wedge d \geq 1 \wedge d \leq 29 \wedge \mathit{yType}(y) = \mathit{LYear}) \wedge (\mathit{mType}(m) = M\_28\_29 \wedge d \geq 1 \wedge d \leq 28 \wedge \mathit{yType}(y) = \mathit{CYear})$ ⟩

The corresponding decision table is shown in Table 18.

APPENDIX D
PROOFS

The proofs for all the lemmas and theorems are given in this appendix.

Lemma 1. Consider an irreducible DNF expression $p_1 \vee \cdots \vee p_k \vee \cdots \vee p_h$ $(h \geq 1)$ not equivalent to *false*. At least one solution can be found for the constraint $p_k \wedge \bigwedge_{k_1=1, k_1 \neq k}^{h} \neg p_{k_1}$, where $k = 1, 2, \ldots, h$.

Proof: The lemma is proven in two different cases: $h = 1$ and $h > 1$.

1. ***h* = 1.** The constraint simplifies to p_1 as follows: $p_k \wedge \bigwedge_{k_1=1, k_1 \neq k}^{h} \neg p_{k_1} = p_1$. Since the constraint is not equivalent to *false*, p_1 is not *false*. Therefore, there must be a solution to p_1.
2. ***h* > 1.** To prove that at least one solution exists for $p_k \wedge \bigwedge_{k_1=1, k_1 \neq k}^{h} \neg p_{k_1}$, it is only required to prove that $p_k \wedge \bigwedge_{k_1=1, k_1 \neq k}^{h} \neg p_{k_1}$ is not equivalent to *false*. It therefore suffices to prove the following:
   a. p_k is not equivalent to *false*,
   b. $\bigwedge_{k_1=1, k_1 \neq k}^{h} \neg p_{k_1}$ is not equivalent to *false*, and
   c. if $p_k \neq \mathit{false}$ and $\bigwedge_{k_1=1, k_1 \neq k}^{h} \neg p_{k_1} \neq \mathit{false}$, it follows that $p_k \Rightarrow \neg \bigwedge_{k_1=1, k_1 \neq k}^{h} \neg p_{k_1}$ is not *true*.

   Case a is valid because no term equals *false* in an irreducible DNF expression.
   Case b is also valid. If $\bigwedge_{k_1=1, k_1 \neq k}^{h} \neg p_{k_1} = \mathit{false}$, then $\bigvee_{k_1=1, k_1 \neq k}^{h} p_{k_1} = \mathit{true}$. This is impossible for an irreducible DNF expression.
   We prove case c by reductio ad absurdum. If $p_k \Rightarrow \neg \bigwedge_{k_1=1, k_1 \neq k}^{h} \neg p_{k_1}$, it follows that $p_k \Rightarrow \bigvee_{k_1=1, k_1 \neq k}^{h} p_{k_1}$. The expression $p_1 \vee \cdots \vee p_{k-1} \vee p_k \vee p_{k+1} \vee \cdots \vee p_h$ is therefore equivalent to $p_1 \vee \cdots \vee p_{k-1} \vee p_{k+1} \vee \cdots \vee p_h$, that is, the removal of p_k does not affect the result of the expression. Hence, $p_1 \vee \cdots \vee p_k \vee \cdots \vee p_h$ is not an irreducible DNF expression. This contradicts the assumed premise.

□

Lemma 2. Suppose $\bigvee_{k_1=1}^{h} (r_{k_1}^1 \wedge \cdots \wedge r_{k_1}^{d_{k_1}})$ is an irreducible DNF expression that is not equivalent to *true* or *false*. At least one solution can be found for the constraint $(r_k^1 \wedge \cdots \wedge \neg r_k^l \wedge \cdots \wedge r_k^{d_k}) \wedge \bigwedge_{k_1=1, k_1 \neq k}^{h} \neg (r_{k_1}^1 \wedge \cdots \wedge r_{k_1}^{d_{k_1}})$, where $k = 1, 2, \ldots, h$ and $l = 1, 2, \ldots, d_k$.

Proof: We prove the lemma in two different cases: $h = 1$ and $h > 1$.

1. ***h* = 1.** The expression contains only one term $r_1^1 \wedge \cdots \wedge r_1^l \wedge \cdots \wedge r_1^{d_1}$.
   a. ***d*₁ = 1.** The expression is r_1^1 and the constraint is $\neg r_1^1$. Since r_1^1 is not equivalent to *true*, a solution for $\neg r_1^1$ exists.
   b. ***d*₁ > 1.** The constraint is $r_1^1 \wedge \cdots \wedge \neg r_1^l \wedge \cdots \wedge r_1^{d_1}$. We prove the case by reductio ad absurdum. If no solution exists for this constraint, $r_1^1 \wedge \cdots \wedge \neg r_1^l \wedge \cdots \wedge r_1^{d_1}$ is equivalent to *false*. Since the expression is in irreducible DNF, neither $\neg r_1^l$ nor $r_1^1 \wedge \cdots \wedge r_1^{l-1} \wedge r_1^{l+1} \wedge \cdots \wedge r_1^{d_1}$ is *false*. Hence, $\neg r_1^l \Rightarrow \neg(r_1^1 \wedge \cdots \wedge r_1^{l-1} \wedge r_1^{l+1} \wedge \cdots \wedge r_1^{d_1})$, that is, $r_1^1 \wedge \cdots \wedge r_1^{l-1} \wedge r_1^{l+1} \wedge$

TABLE 18
Decision Table for *DayError*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$mType(m) = M\_31$	T	–	–	–	–	–	–	–	T	T	F	F	F	F	F	T	F	F	F
$mType(m) = M\_30$	T	–	–	–	–	–	–	–	F	F	T	T	F	F	F	F	T	F	F
$d < 1$	–	T	T	T	T	–	–	–	T	F	T	F	T	F	F	F	F	F	F
$d > 31$	–	T	–	–	–	T	–	–	F	T	F	–	F	–	–	F	F	F	F
$d > 30$	–	–	T	–	–	F	T	–	F	T	F	T	F	–	–	–	F	F	F
$d > 29$	–	–	–	T	–	–	F	T	F	T	F	T	F	–	T	–	–	F	F
$d > 28$	–	–	–	–	T	–	–	F	F	T	F	T	F	T	T	–	–	–	F
$YType = CYear$	–	–	–	–	–	–	–	–	–	–	–	–	–	T	F	–	–	F	T
true									√	√	√	√	√	√	√				
false																√	√	√	√
impossible	√	√	√	√	√	√	√	√											

$\cdots \wedge r_1^{d_1} \Rightarrow r_1^l$. Therefore, $r_1^1 \wedge \cdots \wedge r_1^{l-1} \wedge r_1^l \wedge r_1^{l+1} \wedge \cdots \wedge r_1^{d_1} \equiv r_1^1 \wedge \cdots \wedge r_1^{l-1} \wedge r_1^{l+1} \wedge \cdots \wedge r_1^{d_1}$, that is, r_1^l can be removed without changing the result of the expression. This cannot take place in an irreducible DNF expression.

2. $h > 1$.

a. $d_k = 1$. In this case, the constraint $(r_k^1 \wedge \ldots \wedge \neg r_k^l \wedge \ldots \wedge r_k^{d_k}) \wedge \bigwedge_{k_1=1,k_1\neq k}^{h} \neg(r_{k_1}^1 \wedge \ldots \wedge r_{k_1}^{d_{k_1}})$ becomes $\neg r_k^1 \wedge \bigwedge_{k_1=1,k_1\neq k}^{h} \neg(r_{k_1}^1 \wedge \ldots \wedge r_{k_1}^{d_{k_1}})$. If $\neg r_k^1 \wedge \bigwedge_{k_1=1,k_1\neq k}^{h} \neg(r_{k_1}^1 \wedge \ldots \wedge r_{k_1}^{d_{k_1}}) = false$, $r_k^1 \vee \bigvee_{k_1=1,k_1\neq k}^{h} (r_{k_1}^1 \wedge \ldots \wedge r_{k_1}^{d_{k_1}}) = true$. This contradicts the premise that the expression $\bigvee_{k_1=1}^{h} (r_{k_1}^1 \wedge \ldots \wedge r_{k_1}^{d_{k_1}})$ is not equivalent to *true*.

b. $d_k > 1$. To prove $(r_k^1 \wedge \cdots \wedge \neg r_k^l \wedge \cdots \wedge r_k^{d_k}) \wedge \bigwedge_{k_1=1,k_1\neq k}^{h} \neg(r_{k_1}^1 \wedge \cdots \wedge r_{k_1}^{d_{k_1}})$ is not equivalent to *false*, we need only prove that

i. $r_k^1 \wedge \cdots \wedge r_k^{l-1} \wedge r_k^{l+1} \wedge \cdots \wedge r_k^{d_k}$ is not equivalent to *false*,

ii. $\neg r_k^l \wedge \bigwedge_{k_1=1,k_1\neq k}^{h} \neg(r_{k_1}^1 \wedge \cdots \wedge r_{k_1}^{d_{k_1}})$ is not equivalent to *false*, and

iii. if $r_k^1 \wedge \cdots \wedge r_k^{l-1} \wedge r_k^{l+1} \wedge \cdots \wedge r_k^{d_k} \neq false$ and $\neg r_k^l \bigwedge_{k_1=1,k_1\neq k}^{h} \neg(r_{k_1}^1 \wedge \cdots \wedge r_{k_1}^{d_{k_1}}) \neq false$, then $r_k^1 \wedge \cdots \wedge r_k^{l-1} \wedge r_k^{l+1} \wedge \cdots \wedge r_k^{d_k} \Rightarrow \neg(\neg r_k^l \wedge \bigwedge_{k_1=1,k_1\neq k}^{h} \neg(r_{k_1}^1 \wedge \ldots \wedge r_{k_1}^{d_{k_1}}))$ is not *true*.

Case i is valid because the expression is in irreducible DNF.

We prove case ii by reductio ad absurdum. Since $\neg r_k^l \wedge \bigwedge_{k_1=1,k_1\neq k}^{h} \neg(r_{k_1}^1 \wedge \cdots \wedge r_{k_1}^{d_{k_1}}) \equiv false$, we have $r_k^l \vee \bigvee_{k_1=1,k_1\neq k}^{h} (r_{k_1}^1 \wedge \cdots \wedge r_{k_1}^{d_{k_1}}) \equiv true$. Hence, $r_k^1 \wedge \cdots \wedge r_k^{l-1} \wedge r_k^l \wedge r_k^{l+1} \wedge \cdots \wedge r_k^{d_k} \vee \bigvee_{k_1=1,k_1\neq k}^{h} (r_{k_1}^1 \wedge \cdots \wedge r_{k_1}^{d_{k_1}}) = r_k^1 \wedge \cdots \wedge r_k^{l-1} \wedge r_k^{l+1} \wedge \cdots \wedge r_k^{d_k} \vee \bigvee_{k_1=1,k_1\neq k}^{h} (r_{k_1}^1 \wedge \cdots \wedge r_{k_1}^{d_{k_1}})$, that is, r_k^l can be removed.

We also prove case iii by reductio ad absurdum. $r_k^1 \wedge \ldots \wedge r_k^{l-1} \wedge r_k^{l+1} \wedge \ldots \wedge r_k^{d_k} \Rightarrow \neg(\neg r_k^l \wedge \bigwedge_{k_1=1,k_1\neq k}^{h} \neg(r_{k_1}^1 \wedge \ldots \wedge r_{k_1}^{d_{k_1}})) \equiv r_k^1 \wedge \ldots \wedge r_k^{l-1} \wedge r_k^{l+1} \wedge \ldots \wedge r_k^{d_k} \Rightarrow r_k^l \vee \bigvee_{k_1=1,k_1\neq k}^{h} (r_{k_1}^1 \wedge \ldots \wedge r_{k_1}^{d_{k_1}})$. Therefore, $r_k^1 \wedge \ldots \wedge r_k^{l-1} \wedge r_k^l \wedge r_k^{l+1} \wedge \ldots \wedge r_k^{d_k} \vee \bigvee_{k_1=1,k_1\neq k}^{h} (r_{k_1}^1 \wedge \ldots \wedge r_{k_1}^{d_{k_1}}) \equiv r_k^1 \wedge \ldots \wedge r_k^{l-1} \wedge r_k^{l+1} \wedge \ldots \wedge r_k^{d_k} \vee \bigvee_{k_1=1,k_1\neq k}^{h} (r_{k_1}^1 \wedge \ldots \wedge r_{k_1}^{d_{k_1}})$, that is, r_k^l can be removed without changing the result of the expression. This contradicts the premise that the expression is irreducible.

□

Theorem 1. Decision table-based testing unconditionally subsumes the partition strategy for tabular expressions over *NDSP*, that is, $S^D(NDSP) \rhd\!\rhd S^P(NDSP)$.

Proof: To prove the theorem, it is only necessary to prove that the two strategies satisfy CUS_1 and CUS_2.

***CUS*₁.** The list of test case constraints for decision table-based testing can be rewritten in the following form for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$.

$$STCC_{i,j}(S^D, NDSP) = \Big\langle c_{i,j}^{1,1} \wedge \cdots \wedge c_{i,j}^{1,s_{i,j}^1} \wedge \bigwedge_{k_1=1,k_1\neq 1}^{w_{i,j}} \neg(c_{i,j}^{k_1,1} \wedge \cdots \wedge c_{i,j}^{k_1,s_{i,j}^{k_1}}), \ldots, c_{i,j}^{k,1} \wedge \cdots \wedge c_{i,j}^{k,s_{i,j}^k} \wedge \bigwedge_{k_1=1,k_1\neq k}^{w_{i,j}} \neg(c_{i,j}^{k_1,1} \wedge \cdots \wedge c_{i,j}^{k_1,s_{i,j}^{k_1}}), \ldots, c_{i,j}^{w_{i,j},1} \wedge \cdots \wedge c_{i,j}^{w_{i,j},s_{i,j}^{w_{i,j}}} \wedge \bigwedge_{k_1=1,k_1\neq w_{i,j}}^{w_{i,j}} \neg(c_{i,j}^{k_1,1} \wedge \cdots \wedge c_{i,j}^{k_1,s_{i,j}^{k_1}})\Big\rangle.$$

If $E_{i,j}$ is not *false* for $sp \in NDSP$, according to Lemma 1, each constraint in $STCC_{i,j}(S^D, NDSP)$ is not *false* with respect to *sp*. Therefore, $stcc_{i,j}(S^D, sp)$ contains $w_{i,j}$ constraints. Suppose $t_{i,j}$ is a test case that satisfies $c_{i,j}^{k,1} \wedge \cdots \wedge c_{i,j}^{k,s_{i,j}^k} \wedge \bigwedge_{k_1=1,k_1\neq k}^{w_{i,j}} \neg(c_{i,j}^{k_1,1} \wedge \cdots \wedge c_{i,j}^{k_1,s_{i,j}^{k_1}})$ $(k = 1, 2, \ldots, w_{i,j})$ with respect to *sp*. Since $c_{i,j}^{k,1} \wedge \ldots \wedge c_{i,j}^{k,s_{i,j}^k} \wedge \bigwedge_{k_1=1,k_1\neq k}^{w_{i,j}} \neg(c_{i,j}^{k_1,1} \wedge \ldots \wedge c_{i,j}^{k_1,s_{i,j}^{k_1}})$ implies the constraint $c_{i,j}^{k,1} \wedge \cdots \wedge c_{i,j}^{k,s_{i,j}^k} \vee \bigvee_{k_1=1,k_1\neq k}^{w_{i,j}} (c_{i,j}^{k_1,1} \wedge \cdots \wedge c_{i,j}^{k_1,s_{i,j}^{k_1}})$, it follows that $t_{i,j}$ satisfies this constraint also, so that $\langle t_{i,j}\rangle_{O(i,j)} \in WT(S^P, NDSP)$.

***CUS*₂.** If no test case satisfies decision table-based testing, according to Lemma 1, $E_{i,j}$ is *false* for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$. As a result, there is no test case for the partition strategy. □

Theorem 2. The basic meaningful impact strategy unconditionally subsumes decision table-based testing over *NDSP*, that is, $S^B(NDSP) \rhd\!\rhd S^D(NDSP)$.

Proof: Since $STCC(S^B, NDSP) \supset STCC(S^D, NDSP)$ and the first list in $STCC(S^B, NDSP)$ is exactly $STCC(S^D, NDSP)$, both CUS_1 and CUS_2 are satisfied. □

Theorem 3. a) Fault-based testing for the LOF and LIF classes of faults conditionally subsumes the basic meaningful impact strategy over *NDSP*, that is, $S^F(\mathit{NDSP}) \rhd S^B(\mathit{NDSP})$. b) The basic meaningful impact strategy conditionally subsumes fault-based testing for the LOF and LIF classes of faults, that is, $S^B(\mathit{NDSP}) \rhd S^F(\mathit{NDSP})$.

Proof: $\mathit{STCC}(S^B, \mathit{NDSP})$

$$= \left\langle c_{i,j}^{k,1} \wedge \cdots \wedge c_{i,j}^{k,s_{i,j}^k} \wedge \bigwedge_{k_1=1,k_1\neq k}^{w_{i,j}} \neg(c_{i,j}^{k_1,1} \wedge \cdots \wedge c_{i,j}^{k_1,s_{i,j}^{k_1}}) \right\rangle_{O(i,j,k)}$$

$$\oplus \left\langle \neg c_{i,j}^{k,l} \wedge \bigwedge_{l_1=1,l_1\neq l}^{s_{i,j}^k} c_{i,j}^{k,l_1} \wedge \right.$$

$$\left. \bigwedge_{k_1=1,k_1\neq k}^{w_{i,j}} \neg(c_{i,j}^{k_1,1} \wedge \cdots \wedge c_{i,j}^{k_1,s_{i,j}^{k_1}}) \right\rangle_{O(i,j,k)\ \wedge\ l=1,2,\ldots,s_{i,j}^k}$$

In fault-based testing, the list of test case constraints for LOF is $\left\langle \neg c_{i,j}^{k,l} \wedge \bigwedge_{l_1=1,l_1\neq l}^{s_{i,j}^k} c_{i,j}^{k,l_1} \wedge \bigwedge_{k_1=1,k_1\neq k}^{w_{i,j}} \neg(c_{i,j}^{k_1,1} \wedge \cdots \wedge c_{i,j}^{k_1,s_{i,j}^{k_1}}) \right\rangle_{O(i,j,k)\ \wedge\ s_{i,j}^k>1\ \wedge\ l=1,2,\ldots,s_{i,j}^k}$ $\oplus \left\langle c_{i,j}^{k,1} \wedge \bigwedge_{k_1=1,k_1\neq k}^{w_{i,j}} \neg(c_{i,j}^{k_1,1} \wedge \cdots \wedge c_{i,j}^{k_1,s_{i,j}^{k_1}}) \right\rangle_{O(i,j,k)\ \wedge\ s_{i,j}^k=1}$. The list of test case constraints for LIF is $\left\langle c_{i,j}^{k,1} \wedge \cdots \wedge c_{i,j}^{k,s_{i,j}^k} \wedge \neg c \wedge \bigwedge_{k_1=1,k_1\neq k}^{w_{i,j}} \neg(c_{i,j}^{k_1,1} \wedge \cdots \wedge c_{i,j}^{k_1,s_{i,j}^{k_1}}) \right\rangle_{O(i,j,k)\ \wedge\ c\in(L_{i,j}-L_{i,j}^k)}$.

Let $\mathit{STCC}_1(S^B, \mathit{NDSP})$, $\mathit{STCC}_2(S^B, \mathit{NDSP})$, …, $\mathit{STCC}_r(S^B, \mathit{NDSP})$ denote the sublists in $\mathit{STCC}(S^B, \mathit{NDSP})$ such that $\mathit{STCC}(S^B, \mathit{NDSP}) = \mathit{STCC}_1(S^B, \mathit{NDSP}) \oplus \mathit{STCC}_2(S^B, \mathit{NDSP}) \oplus \ldots \oplus \mathit{STCC}_r(S^B, \mathit{NDSP})$. We can write $\mathit{STCC}_1(S^F, \mathit{NDSP})$ in the following format:

$$\left\langle \bigwedge_{l_1=1}^{s_{i,j}^k} c_{i,j}^{k,l_1} \wedge \neg c_{i,j}'^{k,1} \wedge \bigwedge_{k_1=1,k_1\neq k}^{w_{i,j}} \neg(c_{i,j}^{k_1,1} \wedge \cdots \wedge c_{i,j}^{k_1,s_{i,j}^{k_1}}), \ldots, \bigwedge_{l_1=1}^{s_{i,j}^k} c_{i,j}^{k,l_1} \wedge \neg c_{i,j}'^{k,u} \wedge \bigwedge_{k_1=1,k_1\neq k}^{w_{i,j}} \neg(c_{i,j}^{k_1,1} \wedge \cdots \wedge c_{i,j}^{k_1,s_{i,j}^{k_1}}) \right\rangle_{O(i,j,k)},$$

where $c_{i,j}'^{k,1}$, $c_{i,j}'^{k,2}$, …, $c_{i,j}'^{k,u}$ are the elements in $L_{i,j} - L_{i,j}^k$. $\bigwedge_{l_1=1}^{s_{i,j}^k} c_{i,j}^{k,l_1} \wedge \neg c_{i,j}'^{k,1} \wedge \bigwedge_{k_1=1,k_1\neq k}^{w_{i,j}} \neg(c_{i,j}^{k_1,1} \wedge \cdots \wedge c_{i,j}^{k_1,s_{i,j}^{k_1}})$ implies $\bigwedge_{l_1=1}^{s_{i,j}^k} c_{i,j}^{k,l_1} \wedge \bigwedge_{k_1=1,k_1\neq k}^{w_{i,j}} \neg(c_{i,j}^{k_1,1} \wedge \cdots \wedge c_{i,j}^{k_1,s_{i,j}^{k_1}})$. Let $t_{i,j}^k$ denote a test case that satisfies the constraint $\bigwedge_{l_1=1}^{s_{i,j}^k} c_{i,j}^{k,l_1} \wedge \neg c_{i,j}'^{k,1} \wedge \bigwedge_{k_1=1,k_1\neq k}^{w_{i,j}} \neg(c_{i,j}^{k_1,1} \wedge \cdots \wedge c_{i,j}^{k_1,s_{i,j}^{k_1}})$ with respect to *sp*. The sublist $\langle t_{i,j}^k \rangle_{O(i,j,k)}$ satisfies $\mathit{stcc}_1(S^B, \mathit{sp})$. If $s_{i,j}^k > 1$ for each term in $E_{i,j}$ ($i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$), $\mathit{STCC}_2(S^F, \mathit{NDSP}) = \mathit{STCC}_2(S^B, \mathit{NDSP})$. Hence, if $s_{i,j}^k > 1$ and $L_{i,j} - L_{i,j}^k \neq \emptyset$, S^F subsumes S^B; otherwise, S^F may not subsume S^B. That is, S^F subsumes S^B only when some sublists of $\mathit{STCC}(S^F, \mathit{NDSP})$ exist. Thus, fault-based testing conditionally subsumes the basic meaningful impact strategy.

It is obvious that, if $\mathit{STCC}_1(S^F, \mathit{SPEC}) = \emptyset$ over a specification *sp*, then S^B subsumes S^F. Thus, the basic meaningful impact strategy conditionally subsumes fault-based testing. □

Theorem 4. Fault-based testing for the LOF and LIF classes of faults conditionally subsumes decision table-based testing over *NDSP*, that is, $S^F(\mathit{NDSP}) \rhd S^D(\mathit{NDSP})$.

Proof: The proof is similar to that of Theorem 3. □

Theorem 5. Fault-based testing for the LOF and LIF classes of faults conditionally subsumes the partition strategy over *NDSP*, that is, $S^F(\mathit{NDSP}) \rhd S^P(\mathit{NDSP})$.

Proof: Clearly, if there exists k $(1 \leq k \leq w_{i,j})$ for each $E_{i,j}$ such that $\bigwedge_{l_1=1}^{s_{i,j}^k} c_{i,j}^{k,l_1} \wedge \neg c \wedge \bigwedge_{k_1=1,k_1\neq k}^{w_{i,j}} \neg(c_{i,j}^{k_1,1} \wedge \cdots \wedge c_{i,j}^{k_1,s_{i,j}^{k_1}})$ exists or $s_{i,j}^k = 1$ with respect to $\mathit{sp} \in \mathit{NDSP}$, S^F subsumes S^P; otherwise, S^F does not subsume S^P. □

REFERENCES

[1] H. Agrawal, R.A. DeMillo, B. Hathaway, W.M. Hsu, W. Hsu, E.W. Krawser, R.J. Martin, A.P. Mathur, and E. Spafford, "Design of mutant operators for the C programming language," Technical Report SERC-TR-41-P, Software Engineering Research Center, Department of Computer Sciences, Purdue University, W. Lafayette, IN, 1989.

[2] T.A. Alspaugh, S.R. Faulk, K.H. Britton, R.A. Parker, and D.L. Parnas, "Software Requirements for the A-7E Aircraft," Technical Report NRL/FR/5530-92-9194, Naval Research Lab, Washington, DC, 1992.

[3] R.L. Baber, D.L. Parnas, S.A. Vilkomir, P. Harrison, and T. O'Connor, "Disciplined methods of software specification: a case study," *Proceedings of the International Conference on Information Technology: Coding and Computing* (*ITCC 2005*), vol. 2, pp. 428–437. Los Alamitos, CA: IEEE Computer Society Press, 2005.

[4] V.R. Basili and R.W. Selby, "Comparing the effectiveness of software testing strategies," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 12, pp. 1278–1296, 1987.

[5] B. Beizer, *Software Testing Techniques*. New York, NY: Van Nostrand Reinhold, 1990.

[6] T.Y. Chen, F.-C. Kuo, and R.G. Merkel, "On the statistical properties of testing effectiveness measures," *Journal of Systems and Software*, vol. 79, pp. 591–601, 2006.

[7] T.Y. Chen and M.F. Lau, "Test case selection strategies based on Boolean specifications," *Software Testing, Verification and Reliability*, vol. 11, no. 3, pp. 165–180, 2001.

[8] T.Y. Chen, M.F. Lau, and Y.T. Yu, "MUMCUT: a fault-based strategy for testing Boolean specifications," *Proceedings of the 6th Asia-Pacific Software Engineering Conference* (*APSEC '99*), pp. 606–613. Los Alamitos, CA: IEEE Computer Society Press, 1999.

[9] T.Y. Chen and Y.T. Yu, "On the expected number of failures detected by subdomain testing and random testing," *IEEE Transactions on Software Engineering*, vol. 22, no. 2, pp. 109–119, 1996.

[10] J.J. Chilenski and S.P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, vol. 9, no. 5, pp. 193–200, 1994.

[11] M. Clermont and D.L. Parnas, "Using information about functions in selecting test cases," *in Proceedings of the 1st International Workshop on Advances in Model-Based Testing (A-MOST 2005) (in conjunction with Proceedings of the 27th International Conference on Software Engineering* (*ICSE 2005*)*), ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–7, 2005.

[12] R.A. DeMillo and A.J. Offutt, "Experimental results from an automatic test case generator," *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 2, pp. 109–127, 1993.

[13] X. Feng, *MIST: Towards a MInimum Set of Test Cases*, PhD Thesis. The University of Hong Kong, Pokfulam, Hong Kong, 2002.

[14] X. Feng, S. Marr, and T. O'Callaghan, "ESTP: an experimental software testing platform," *Proceedings of the Testing: Academic and Industrial Conference: Practice And Research Techniques* (*TAIC PART 2008*), pp. 59–63. Los Alamitos, CA: IEEE Computer Society Press, 2008.

[15] P.G. Frankl and E.J. Weyuker, "A formal analysis of the fault-detecting ability of testing methods," *IEEE Transactions on Software Engineering*, vol. 19, no. 3, pp. 202–213, 1993.

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

[17] P.R. Halmos, *Naive Set Theory*. Princeton, NJ: Van Nostrand, 1960. Reprinted by New York, NY: Springer, 1974.

[18] K.L. Heninger, "Specifying software requirements for complex systems: new techniques and their application," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 1, pp. 2–13, 1980.

[19] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, pp. 371–379, 1982.

[20] R. Janicki and R. Khedri, "On a formal semantics of tabular expressions," *Science of Computer Programming*, vol. 39, no. 2-3, pp. 189–213, 2001.

[21] R. Janicki, D. L. Parnas, and J. I. Zucker, "Tabular representations in relational documents," *Relational Methods in Computer Science*, C. Brink, W. Kahl, and G. Schmidt, eds., pp. 184–196. New York, NY: Springer, 1997.

[22] R. Janicki and A. Wassyng, "Tabular expressions and their relational semantics," *Fundamenta Informaticae*, vol. 67, no. 4, pp. 343–370, 2005.

[23] P. C. Jorgensen, *Software Testing: a Craftsman's Approach*. Boca Raton, FL: Auerbach Publications, 2008.

[24] G. Kaminski, G. Williams, and P. Ammann, "Reconciling perspectives of software logic testing," *Software Testing, Verification and Reliability*, vol. 18, no. 3, pp. 149–188, 2008.

[25] D. R. Kuhn, "Fault classes and error detection capability of specification-based testing," *ACM Transactions on Software Engineering and Methodology*, vol. 8, no. 4, pp. 411–424, 1999.

[26] M. F. Lau and Y. T. Yu, "On the relationships of faults for Boolean specification based testing," *Proceedings of the 2001 Australian Software Engineering Conference* (*ASWEC 2001*), pp. 21–28. Los Alamitos, CA: IEEE Computer Society Press, 2001.

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

[28] S. Liu, *Generating Test Cases from Software Documentation*, MEng Thesis. Department of Electrical and Computer Engineering, McMaster University, Hamilton, Ontario, Canada, 2001.

[29] L. J. Morell, "A theory of fault-based testing," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 844–857, 1990.

[30] G. J. Myers, *The Art of Software Testing*. New York, NY: Wiley, 1979.

[31] V. Okun, P. E. Black, and Y. Yesha, "Comparison of fault classes in specification-based testing," *Information and Software Technology*, vol. 46, no. 8, pp. 525–533, 2004.

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

[33] D. L. Parnas, "Tabular representation of relations," Technical Report 260, McMaster University, Hamilton, Canada, 1992.

[34] D. L. Parnas, "Inspection of safety-critical software using program-function tables," *Proceedings of the IFIP Congress*, vol. 3, pp. 270–277, 1994.

[35] D. L. Parnas, G. J. K. Asmis, and J. Madey, "Assessment of safety-critical software in nuclear power plants," *Nuclear Safety*, vol. 32, no. 2, pp. 189–198, 1991.

[36] D. L. Parnas and J. Madey, "Functional documents for computer systems," *Science of Computer Programming*, vol. 25, no. 1, pp. 41–61, 1995.

[37] D. L. Parnas, J. Madey, and M. Iglewski, "Precise documentation of well-structured programs," *IEEE Transactions on Software Engineering*, vol. 20, no. 12, pp. 948–976, 1994.

[38] D. K. Peters and D. L. Parnas, "Using test oracles generated from program documentation," *IEEE Transactions on Software Engineering*, vol. 24, no. 3, pp. 161–173, 1998.

[39] C. Quinn, S. Vilkomir, D. L. Parnas, and S. Kostic, "Specification of software component requirements using the trace function method," *Proceedings of the International Conference on Software Engineering Advances* (*ICSEA 2006*). Los Alamitos, CA: IEEE Computer Society Press, 2006.

[40] E. Sekerinski, "Exploring tabular verification and refinement," *Formal Aspects of Computing*, vol. 15, no. 2-3, pp. 215–236, 2003.

[41] H. Shen, "Implementation of Table Inversion Algorithms," MEng Thesis, Department of Electrical and Computer Engineering, McMaster University, Hamilton, Ontario, Canada, 1995.

[42] C.-A. Sun, Y. Dong, R. Lai, K. Y. Sim, and T. Y. Chen, "Analyzing and extending MUMCUT for fault-based testing of general Boolean expressions," *Proceedings of the 6th IEEE International Conference on Computer and Information Technology* (*CIT 2006*), pp. 184–189. Los Alamitos, CA: IEEE Computer Society Press, 2006.

[43] K.-C. Tai, "Theory of fault-based predicate testing for computer programs," *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 552–562, 1996.

[44] A. J. van Schouwen, D. L. Parnas, and J. Madey, "Documentation of requirements for computer systems," *Proceedings of the 1st IEEE International Symposium on Requirements Engineering*, pp. 198–207. Los Alamitos, CA: IEEE Computer Society Press, 1993.

[45] E. J. Weyuker, "More experience with data flow testing," *IEEE Transactions on Software Engineering*, vol. 19, no. 9, pp. 912–919, 1993.

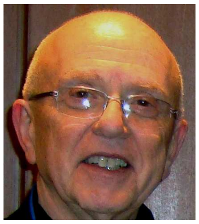
[46] E. J. Weyuker, T. Goradia, and A. Singh, "Automatically generating test data from a Boolean specification," *IEEE Transactions on Software Engineering*, vol. 20, no. 5, pp. 353–363, 1994.

[47] E. J. Weyuker and B. Jeng, "Analyzing partition testing strategies," *IEEE Transactions on Software Engineering*, vol. 17, no. 7, pp. 703–711, 1991.

[48] Y. T. Yu and M. F. Lau, "A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions," *Journal of Systems and Software*, vol. 79, no. 5, pp. 577–590, 2006.

[49] J. I. Zucker, "Transformations of normal and inverted function tables," *Formal Aspects of Computing*, vol. 8, no. 6, pp. 679–705, 1996.

Xin Feng received the BEng degree from Donghua University, China, the MEng degree in software engineering from Nanjing University, China, and the PhD degree in software testing from The University of Hong Kong. She is an assistant professor in the Division of Science and Technology at United International College, Zhuhai, China. She worked in industry for years. She also worked as a senior researcher at the University of Limerick, Ireland. Her current research interests include mutation testing, test data generation, test automation, and software quality assurance. She is active in applying the research in software testing to industry.



David Lorge Parnas received the BSc, MSc, and PhD degrees in electrical engineering from Carnegie Mellon University, and honorary doctorates from ETH Zurich, Switzerland, the Catholic University of Louvain, Belgium, and the University of Italian Switzerland, Lugano. He has been studying industrial software development and publishing widely cited papers since 1969. He is a professor emeritus of McMaster University in Hamilton, Canada, and of the University of Limerick, Ireland. He is an honorary professor at Ji Lin University in China. He is licensed as a professional engineer in Ontario, Canada. Many of his papers have been found to have lasting value. For example, a paper written 25 years ago, based on a study of avionics software, was recently awarded a SIGSOFT IMPACT award. In all, he has won more than 20 awards for his contributions. In 2007, he was proud to share the IEEE Computer Society's *onetime* 60th anniversary award with computer pioneer Professor Maurice Wilkes of Cambridge University. He is a member of the Royal Irish Academy. He is the author of more than 265 papers and reports. Many of his papers have been repeatedly republished and are considered classics. A collection of his papers can be found in: *Software Fundamentals: Collected Papers by David L. Parnas*, D.M. Hoffman and D.M. Weiss, editors (Addison-Wesley, 2001). He is a fellow of the Royal Society of Canada (RSC), the ACM, the Canadian Academy of Engineering (CAE), the Gesellschaft für Informatik (GI) in Germany, and the IEEE.

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

Tony O'Callaghan received the BEng degree in computer engineering from the University of Limerick (UL) and is awaiting confirmation of the MSc degree by research in computer science conducted at the Interaction Design Centre at UL. He previously worked as a software engineer at the Software Quality Research Laboratory at UL with Professor David L. Parnas and Dr. Xin Feng. He is a practitioner in a wide gamut of computer-related research. His research interests include interaction design, human computer interaction, software engineering, and software testing. He also enjoys teaching, reading, and traveling.