<table>
<thead>
<tr>
<th><strong>Title</strong></th>
<th>A comparison of tabular expression-based testing strategies</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Author(s)</strong></td>
<td>Feng, X; Parnas, DL; Tse, TH; O'Callaghan, T</td>
</tr>
<tr>
<td><strong>Citation</strong></td>
<td>IEEE Transactions on Software Engineering, 2011, v. 37 n. 5, p. 616-634</td>
</tr>
<tr>
<td><strong>Issued Date</strong></td>
<td>2011</td>
</tr>
<tr>
<td><strong>URL</strong></td>
<td><a href="http://hdl.handle.net/10722/142946">http://hdl.handle.net/10722/142946</a></td>
</tr>
<tr>
<td><strong>Rights</strong></td>
<td>This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.</td>
</tr>
</tbody>
</table>
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, 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
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-e-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 which 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 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:

$$\text{DayError}(\text{year}, \text{month}, \text{day})$$

$$\equiv \text{MonthType}(\text{month}) = M_{31} \land \text{day} < 1 \lor \text{day} > 31$$

$$\lor \text{MonthType}(\text{month}) = M_{30} \land \text{day} < 1 \lor \text{day} > 30$$

$$\lor \text{MonthType}(\text{month}) = M_{28, 29}$$

$$\lor (\text{day} < 1 \lor \text{day} > 29 \land \text{YearType}(\text{year}) = \text{LeapYear})$$

$$\lor (\text{day} > 28 \land \text{YearType}(\text{year}) = \text{CommonYear})$$

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 \text{MonthType} table (see Fig. 2 in Appendix A) is a tree-structured table, and the \text{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 \text{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.

For ease of presentation, we use \(\bigwedge_{k=1}^{l} p_k\) to denote \(p_1 \land p_2 \land \cdots \land p_l\) and \(\bigvee_{k=1}^{l} p_k\) to denote \(p_1 \lor p_2 \lor \cdots \lor p_l\). In a normal table, the grids \(T[1]\) and \(T[2]\) must be proper, that is, for any input, \(T[1][i][j] = true\) if \(i \neq j\) and \(\bigwedge_{k=1}^{m} T[1][i][k] = true\), where \(m\) is the number of cells in \(T[1]\). Here, \(T[0][i][j]\) is the expression to be evaluated if \(T[1][i][j] \land T[2][j] = true\) with respect to an assignment of values to the variables. We call \(T[1][i][j] \land T[2][j]\) an evaluation condition, denoted by \(E_{i,j}\). Furthermore, \(E_{i,j} \land E_{i,j'} = false\) if \(i \neq i_2\) or \(j \neq j_2\).

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 \(l \geq 1\), 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, \(\bigwedge_{k=1}^{m} (T[1][i][k] \land T[2][j][k])\) 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 \( e_1 \text{op} e_2 \), where \( \text{op} \) 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 “\&” (which denotes “and”) and “\( \lor \)” (which denotes “or’”). A conjunction is a Boolean expression consisting of two subexpressions linked by the operator “\&”. A disjunction is a Boolean expression consisting of two subexpressions linked by the operator “\( \lor \)”.

A Disjunctive Normal Form (DNF) is a Boolean expression consisting of conjunctions of conjunctions of Boolean literals. For example, given the Boolean expression consisting of two subexpressions \( a \text{ or } (b \land c) \), the 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, \( (a \land b \land \neg c) \lor (\neg a \land b \land c) \) is normative 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 \( E_{i,j} = T[1][i] \land T[2][j] \) can be written as

\[
\bigvee_{k=1}^{w_{i,j}} \left( c_{i,j} \land \cdots \land c_{i,j} \right),
\]

where \( k \in \{1, 2, \ldots, s_{i,j} \} \) is a Boolean literal, \( w_{i,j} \) is the number of terms in \( E_{i,j} \), and \( s_{i,j} \) is the number of Boolean literals in the \( k \)th term of \( E_{i,j} \). For example, if \( T[1][2] \) is \( (x > 3 \land x < 0) \) and \( T[2][3] \) is \( (y > 10) \), then \( E_{2,3} = x > 3 \land y > 10 \lor x < 0 \land y > 10 \). In this expression, \( w_{2,3} = 2, s_{2,3} = 2, c_{2,3} = x > 3, c_{2,3} = y > 10, c_{2,3} = x < 0, \) and \( c_{2,3} = y > 10 \).

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抽象 test case constraints. The following conditions that correspond to the individual evaluation expressions can be derived from the table:

<table>
<thead>
<tr>
<th>( E_{1,1} )</th>
<th>( E_{1,2} )</th>
<th>( E_{2,1} )</th>
<th>( E_{2,2} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>((y &gt; 1 \land x &lt; 1) \lor (y &gt; 1 \land x &gt; 31))</td>
<td>(y &gt; 1 \land x &gt; 1 \land x &lt; 31)</td>
<td>((y &lt; 1 \land x &lt; 1) \lor (y &lt; 1 \land x &gt; 31))</td>
<td>(y &lt; 1 \land x &gt; 1 \land x &lt; 31)</td>
</tr>
</tbody>
</table>

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

\[
\bigvee_{i=1}^{w_{i,j}} \left( c_{i,j} \land \cdots \land c_{i,j} \right),
\]

where \( O(i, j) \) denotes \( i = 1, 2, \ldots, m \land j = 1, 2, \ldots, n \land T[0][i, j] \neq \text{undefined} \) 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

\[
\{(y > 1 \land x < 1) \lor (y > 1 \land x > 31),
\]

\[
y > 1 \land x > 1 \land x < 31,
\]

\[
(y < 1 \land x < 1) \lor (y < 1 \land x > 31),
\]

\[
y < 1 \land x > 1 \land x < 31\}.
\]

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.

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.

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 $p_1 \land \cdots \land p_h \land \cdots \land p_m$, the constraint for term $p_k$ is $p_k \land \bigwedge_{h=1}^{h} \neg p_h$, 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 shows how this algorithm is applied to the DayError example.

**Lemma 1.** Consider an irreducible DNF expression $p_1 \lor \cdots \lor p_h$ ($h \geq 1$) not equivalent to false. At least one solution can be found for the constraint $p_k \land \bigwedge_{h=1}^{h} \neg p_h$, 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

$$\left( c_{i,j}^{k_1} \land \cdots \land c_{i,j}^{k_l} \land \bigwedge_{h=1 \neq k}^{h} \neg \left( c_{i,j}^{k_1} \land \cdots \land c_{i,j}^{k_l} \right) \right)_{O(i,j,k)},$$

where $O(i,j,k)$ denotes $O(i,j) \land k = 1, 2, \ldots, w_{i,j}$ 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
\[ \mathbf{y} > 1 \land x < 1 \land \neg(y > 1 \land x > 31), \\
\neg(y > 1 \land x < 1) \land y > 1 \land x > 31, \\
y > 1 \land x \geq 1 \land x \leq 31, \\
y \leq 1 \land x < 1 \land \neg(y \leq 1 \land x > 31), \\
\neg(y \leq 1 \land x < 1) \land y \leq 1 \land x > 31, \\
y \leq 1 \land x \geq 1 \land x \leq 31. \]

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 \( \bigvee_{k=1}^{h} (r_{k1} \land \cdots \land r_{kl}) \), the constraint for \( r_{ki} (k = 1, 2, \ldots, h \text{ and } l = 1, 2, \ldots, d_{k}) \) is \( \neg r_{ki} \land \bigwedge_{k=1, k \neq k}^{h} (r_{ki} \land \cdots \land r_{kl}) \).

For an expression with only one term, the constraint for \( r_{1i} \) is \( \neg r_{1i} \land \cdots \land \neg r_{1i} \land \cdots \land r_{1i} \) if \( d_{1i} > 1 \), and \( \neg r_{1i} \) otherwise.

Lemma 2. Suppose \( \bigvee_{k=1}^{h} (r_{1i} \land \cdots \land r_{d_{1i}}) \) 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 \( (r_{1i} \land \cdots \land \neg r_{1i} \land \cdots \land r_{d_{1i}}) \land \bigwedge_{k=1, k \neq k}^{h} (r_{1i} \land \cdots \land r_{d_{1i}}) \), where \( k = 1, 2, \ldots, h \text{ and } l = 1, 2, \ldots, d_{k} \).

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:

\[ \left( \bigwedge_{k=1}^{w_{ij}} (r_{1i} \land \cdots \land r_{d_{1i}}) \right) \bigwedge \left( \bigwedge_{k=1}^{w_{ij}} (r_{1i} \land \cdots \land r_{d_{1i}}) \right) \bigwedge \left( \bigwedge_{k=1}^{w_{ij}} (r_{1i} \land \cdots \land r_{d_{1i}}) \right) O_{(i,j,k)} \]

where \( \oplus \) denotes list concatenation.

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

\[ \left( \bigwedge_{k=1}^{w_{ij}} (r_{1i} \land \cdots \land r_{d_{1i}}) \right) \bigwedge \left( \bigwedge_{k=1}^{w_{ij}} (r_{1i} \land \cdots \land r_{d_{1i}}) \right) \bigwedge \left( \bigwedge_{k=1}^{w_{ij}} (r_{1i} \land \cdots \land r_{d_{1i}}) \right) O_{(i,j,k)} \]

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 “\( \lor \)” is implemented as “\( \lor \)” (ORF[1]), or “\( \lor \)” is implemented as “\( \lor \)” (ORF[1]).
- 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 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 x n normal table without duplicated evaluation expressions are

$$\begin{align*}
\neg c_{i,j} \land \bigwedge_{l=1}^{w} c_{i,l}^{k_{l}} \land \\
\bigvee_{k=1}^{w} c_{i,j}^{k_{l}} \land \neg c_{i,j} \land \\
\bigwedge_{l=1}^{w} c_{i,l}^{k_{l}} \land \neg c_{i,j} \land \\
\bigwedge_{k=1}^{w} c_{i,j}^{k_{l}} \land \neg c_{i,j} \land \\
\bigwedge_{l=1}^{w} c_{i,l}^{k_{l}} \land \neg c_{i,j} \land \\
\bigwedge_{k=1}^{w} c_{i,j}^{k_{l}} \land \neg c_{i,j} \land \\
\bigwedge_{l=1}^{w} c_{i,l}^{k_{l}} \land \neg c_{i,j} \land \\
\bigwedge_{k=1}^{w} c_{i,j}^{k_{l}} \land \neg c_{i,j}
\end{align*}$$

or LIF, where \( L_{i,j} \) is the list of all Boolean literals in \( E_{i,j} \). 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 \( x > 30 \land y < 12 \lor x < 5 \land y < 20 \). The predicate \( x > 30 \) cannot be inserted into the term \( x < 5 \land y < 20 \) 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 \( p_1 \land p_2 \lor p_3 \land p_4 \). It is possible to add \( p_1 \) to the second term because neither \( p_1 \) nor \( \neg p_1 \) occurs in the second term. If \( L_{i,j} - L_{i,j}^{L} = \emptyset \), the corresponding constraints do not exist. Hence, the list for LIF can be empty. If a term contains only one Boolean literal (that is, \( s_{i,j}^{L} = 1 \)), LIF for this literal is then taken as TOF.

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

$$\begin{align*}
\neg (y > 1) \land x < 1 \land \neg (y > 1) \land x > 31, \\
y > 1 \land \neg(x < 1) \land \neg (y > 1) \land x > 31, \\
\neg (y > 1) \land x > 1 \land \neg(x > 1), \\
\neg (y > 1) \land x > 1 \land x > 31, \\
\neg (y > 1) \land x > 1 \land \neg (x > 31), \\
\neg (y > 1) \land x > 1 \land x < 31, \\
\neg (y > 1) \land x > 1 \land \neg (x > 31), \\
\neg (y > 1) \land x > 1 \land \neg (x > 31), \\
\neg (y > 1) \land x > 1 \land \neg (x > 31)
\end{align*}$$

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. \( STCC(S, SPEC) \): The lists of test case constraints derived from strategy \( S \) over a class of specifications \( SPEC \).
12. \( stcc(S, sp) \): The list of test case constraints derived from strategy \( S \) for a specification \( sp \).
13. \( T(S, sp) \): A test suite for specification \( sp \) derived from strategy \( S \).
14. \( WT(S, sp) \): The set of all \( T(S, sp) \).

It is clear that \( SP = NDSP \cup DSP \) and \( DSP \cap NDSP = \emptyset \). 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 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.
   - 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.
   - \( S_1 \) is *equivalent* to \( S_2 \) over a specification \( sp \), denoted by \( S_1(sp) \equiv S_2(sp) \), if \( stcc(S_1, sp) \) is equivalent to \( stcc(S_2, sp) \), that is, \( stcc(S_1, sp) = stcc(S_2, sp) \).
   - \( S_1 \) is *equivalent* to \( S_2 \) over a class of specifications \( SPEC \), denoted by \( S_1(SPEC) \equiv S_2(SPEC) \), 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 \( S_1(sp) \supseteq S_2(sp) \), if for any \( T(S_1, sp) \), \( T(S_1, sp) \equiv W(T(S_2, sp)) \).

3. **Unconditional subsumption**
   - Testing strategy \( S_1 \) unconditionally subsumes testing strategy \( S_2 \) over a class of specifications \( SPEC \), denoted by \( S_1(SPEC) \gg S_2(SPEC) \), if the following conditions are satisfied:
     - \( S_1(SPEC) \equiv S_2(SPEC) \).
     - For any specification \( sp \in SPEC \), \( S_1(sp) \succ S_2(sp) \).

4. **Conditional subsumption**
   - A test strategy \( S_1 \) conditionally subsumes another testing strategy \( S_2 \) over a class of specifications \( SPEC \), denoted by \( S_1(SPEC) \gg S_2(SPEC) \), if the following conditions are satisfied:
     - \( S_1(SPEC) \equiv S_2(SPEC) \).
     - For any specification \( sp \in SPEC \), \( S_1(sp) \succ S_2(sp) \).

### 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, \( S^D(NDSP) \gg S^D(NDSP) \).

It follows that \( S^D \) subsumes \( S^P \) over any \( sp \) in NDSP. If \( w_{ij} = 1 \) 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, \( S^P(NDSP) \gg S^D(NDSP) \).

Following this theorem, for any \( sp \) in NDSP, \( S^P(sp) \gg S^D(sp) \). 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 \( STCC(S^P, NDSP) \) 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 \( STCC(S^D, NDSP) \gg STCC(S^D, NDSP) \), it is possible that \( stcc(S^D, sp) = stcc(S^P, sp) \). In \( STCC(S^D, NDSP) \), data satisfying

\[
\neg c_{i,j}^{k_l} \land \bigcup_{i=1}^{s_{i,j}} \bigcup_{k=1}^{k_l} c_{i,j}^{k_l} \land \bigcup_{i=1}^{s_{i,j}} \neg c_{i,j}^{k_l} \land \cdots \land \neg c_{i,j}^{k_l}
\]

evaluate the expression \( T[1][i] \lor T[2][j] \) to false. According to the definition of tabular expressions, there must exist \( i' \) and \( j' \).
(i ≠ i' or j ≠ j') such that the data evaluate \( T[1][i'] \land T[2][j'] \) to true. For example, \( S^B \) and \( S^P \) are equivalent over the specification in Table 7, where \( T[1][1] \land T[2][1] = a > 3 \land b > 5 \), \( T[1][1] \land T[2][2] = a > 3 \land b \leq 5 \), \( T[1][2] \land T[2][1] = a \leq 3 \land b > 5 \), and \( T[2][1] \land T[2][2] = a \leq 3 \land b \leq 5 \).

The lists of test case constraints are \( (a > 3 \land b > 5, a > 3 \land b \leq 5, a \leq 3 \land b > 5, a \leq 3 \land b \leq 5) \) for \( S^D \) and \( (a > 3 \land b > 5, a > 3 \land b \leq 5, a > 3 \land b \leq 5) \) for \( S^B \). The lists of test case constraints are not equivalent over the first list. The strategies \( S^B \) and \( S^D \) are, therefore, equivalent over this specification.

**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, \( S^F(NDSP) \triangleright S^B(NDSP) \). 2) The basic meaningful impact strategy conditionally subsumes fault-based testing for the LOF and LIF classes of faults, that is, \( S^B(NDSP) \triangleright S^F(NDSP) \).

For any specification \( sp \in SPEC \), \( S^F \) subsumes \( S^B \) over \( sp \) only if there exists at least one LIF fault for every term in each evaluation condition; \( S^D \) subsumes \( S^B \) 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, \( S^F(NDSP) \triangleright S^D(NDSP) \).

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

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

**Table 7**

<table>
<thead>
<tr>
<th>Example Where ( S^B(sp) = S^P(sp) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T[2] )</td>
</tr>
<tr>
<td>( a &gt; 3 )</td>
</tr>
<tr>
<td>( a = 3 )</td>
</tr>
<tr>
<td>( a &lt; 3 )</td>
</tr>
<tr>
<td>( T[1] )</td>
</tr>
</tbody>
</table>

**Table 8**

<table>
<thead>
<tr>
<th>An Example Where ( S^D ) Is the Strongest</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T[2] )</td>
</tr>
<tr>
<td>( a &gt; 3 )</td>
</tr>
<tr>
<td>( a = 3 )</td>
</tr>
<tr>
<td>( a &lt; 3 )</td>
</tr>
<tr>
<td>( T[1] )</td>
</tr>
</tbody>
</table>

**Theorem 5.** Fault-based testing for the LOF and LIF classes of faults conditionally subsumes the partition strategy over NDSP, that is, \( S^B(NDSP) \triangleright S^D(NDSP) \).

For any specification \( sp \in SPEC \), \( S^F \) subsumes \( S^D \) 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.

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 due to specific reasons such as compatibility with other tables in the same system.

**Table 9**

<table>
<thead>
<tr>
<th>Another Presentation of Table 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a &gt; 3 )</td>
</tr>
<tr>
<td>( a = 3 )</td>
</tr>
<tr>
<td>( a &lt; 3 )</td>
</tr>
<tr>
<td>( T[1] )</td>
</tr>
</tbody>
</table>

**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 non-equivalent 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,

\[
\text{(month > 12} \land \neg (\text{month < 1}), \neg (\text{month > 12})
\land \neg (\text{month < 1}), \neg (\text{month > 12}) \land (\text{month < 1}),
\neg (\text{month > 12}) \land \neg (\text{month < 1})
\]

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 \((\text{month = 13}, \text{month = 1}, \text{month = 0}, \text{month = 1})\). To create the first test suite, one of the entries “month = 1” is removed because it is duplicated. The resulting test suite is \((\text{month = 13}, \text{month = 1}, \text{month = 0})\). 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 \((\text{month = 13}, \text{month = 1}, \text{month = 0}, \text{month = 2})\).

Tables 12 and 13 present the mutation scores obtained from the experiment. Grid T[2] lists the program names, numbers 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 \(S^B\) 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 \(T_1 = \{(\text{year = 2081}, \text{month = 1}, \text{day = 1}), (\text{year = 1812}, \text{month = 1}, \text{day = 1})\}\), while the test suite for the mutation score 0.646 is \(T_2 = \{(\text{year = 2081}, \text{month = 1}, \text{day = 1}), (\text{year = 1812}, \text{month = 2}, \text{day = 2})\}\). 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
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 \( h_{\text{year}} = 1812; \text{month} = 1; \text{day} = 1 \), the test case \( h_{\text{year}} = 1812; \text{month} = 2; \text{day} = 2 \) is less powerful in revealing faults in the \( n_{\text{Date}} \) 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 \( d_{\text{Error}} \) 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: \( m_{\text{Error}} \) and \( y_{\text{Error}} \). This result is not contradictory to the proof because \( S^F \) does not unconditionally subsume \( S^P \). Both the \( m_{\text{Error}} \) and \( y_{\text{Error}} \) 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 \( m_{\text{Error}} \) and \( y_{\text{Error}} \) 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 \( y_{\text{Error}} \) and \( m_{\text{Error}} \) are \( \{2081, 0, 1812\} \) and \( \{13, 0, 1\} \), respectively. The values 2,081 and 1,812 for \( y_{\text{Error}} \) correspond to the values 13 and 1, respectively. The value 0 in the test suite for \( y_{\text{Error}} \) is derived from the constraint \( \text{day} < 1812 \), while the same value in the test suite for \( m_{\text{Error}} \) is from the constraint \( \text{month} < 1 \). In other words, the BoNus algorithm gives 0 for both \( \text{year} < 1812 \) and \( \text{month} < 1 \). For a program expression such as \( \text{month} < 1 \text{|| month} > 12 \) (written in C), test case 0 is very effective in detecting common faults, while for an expression like \( \text{year} < 1812 \text{|| year} > 2080 \), test case 0 is less effective. When the test case is changed...
from 0 to 1,811 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 $m_{Error}$ and the $y_{Error}$ examples again. Using the $S_B^p$ strategy, the test suites are (13, 1) for $m_{Error}$ and (2081, 1812) for $y_{Error}$. 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 $y_{Error}$ is higher than that for $m_{Error}$. 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. Hence, there is a mutant for $m_{Error}$ where the expression $month > 12$ is changed to $month < 3$; similarly, there is a mutant for $y_{Error}$, where the expression $year > 2080$ is changed to $year < 1812$. Then, both test cases for $m_{Error}$ cannot distinguish this mutant from the original program while the test case 1812 for $y_{Error}$ can distinguish $year > 3$ from $year < 1812$.

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

   It is noted that the number of test cases for $S_F^i$ is less than the number of test cases for $S_B^j$ 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, $S_B^B$ unconditionally subsumes $S_B^p$ and $S_B^j$. These relationships are reflected in the experimental results as expected. On the other hand, $S_B^j$ and $S_F^j$ conditionally subsume each other, but $S_F^i$ 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 $S_F^i$ has higher mutation scores than $S_B^j$, testers should select $S_B^j$. Since this is not the case, let us examine the situation further. In this paper, $S_F^i$ covers two fault classes, namely, LOF and LIF. LOF is one of the fault classes that can also be detected by $S_B^j$. Hence, LOF faults should not cause $S_F^i$ 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 $S_B^j$. 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 $S_F^i$ 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 $S_B^j$ and $S_F^i$, they cover the entire hierarchy diagram with the only exception of LRF. We combine the test cases for $S_B^j$ and $S_F^i$ 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 $S_B^j$ and $S_F^i$ 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 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.

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

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

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 \( day = 31 \) and \( month = 12 \), 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 less than or equal to 60; and 16 euros, 22 euros, and 28 euros per bottle when the quantity is more than 60 bottles.

A salesperson receives a 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.

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 less than 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

\[
(m\text{-Type}(m) = M_{28,29} \
\land d \geq 28) \lor (m\text{-Type}(m) = M_{28,29} \land d < 28 \land y\text{-Type}(y) = \text{Common}) \lor (m\text{-Type}(m) = M_{28,29} \land d < 28 \land y\text{-Type}(y) = \text{Leap})
\]

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

\[
(m\text{-Type}(m) = M_{28,29} \land d \geq 1 \land d \leq 31) \lor (m\text{-Type}(m) = M_{30} \land d \geq 1 \land d \leq 30) \lor (m\text{-Type}(m) = M_{28,29} \land d \geq 1 \land d \leq 28 \land y\text{-Type}(y) = \text{Common}) \lor (m\text{-Type}(m) = M_{28,29} \land d \geq 1 \land d \leq 28 \land y\text{-Type}(y) = \text{Leap})
\]

The list of test case constraints for the decision table-based testing is
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 \( p_k \wedge \bigwedge_{h=1}^{k} \neg p_h \) \( (h \geq 1) \) not equivalent to false. At least one solution can be found for the constraint \( p_k \wedge \bigwedge_{h=1}^{k} \neg p_h \), where \( k = 1, 2, \ldots, h \).

**Proof.** The lemma is proven in two different cases: \( h = 1 \) and \( h > 1 \).

1. **Case 1:** \( h = 1 \). The constraint simplifies to \( p_k \wedge \bigwedge_{h=1}^{h} \neg p_h = p_1 \). Since the constraint is not equivalent to false, \( p_1 \) is not false. Therefore, there must be a solution to \( p_1 \).

2. **Case 2:** \( h > 1 \). To prove that at least one solution exists for \( p_k \wedge \bigwedge_{h=1}^{h} \neg p_h \) \( (h > 1) \), it is only required to prove that \( p_k \wedge \bigwedge_{h=1}^{h} \neg p_h \) is not equivalent to false. It therefore suffices to prove the following:
   a. \( p_k \) is not equivalent to false,
   b. \( \bigwedge_{h=1}^{h} \neg p_h \) is not equivalent to false, and
   c. if \( p_k \not\equiv \) false and \( \bigwedge_{h=1}^{h} \neg p_h \not\equiv \) false, it follows that \( p_k \Rightarrow \neg \bigwedge_{h=1}^{h} \neg p_h \) is not true.

   Case a is valid because no term equals false in an irreducible DNF expression.

   Case b is also valid. If \( \bigwedge_{h=1}^{h} \neg p_h \) \( \not\equiv \) false, then \( \bigwedge_{h=1}^{h} \neg p_h \) is not equivalent to false. It therefore suffices to prove the following:

   a. \( p_k \) is not equivalent to false,
   b. \( \bigwedge_{h=1}^{h} \neg p_h \) is not equivalent to false, and
   c. if \( p_k \not\equiv \) false and \( \bigwedge_{h=1}^{h} \neg p_h \not\equiv \) false, it follows that \( p_k \Rightarrow \neg \bigwedge_{h=1}^{h} \neg p_h \) is not true.

   Case b is also valid. If \( \bigwedge_{h=1}^{h} \neg p_h \) \( \not\equiv \) false, then \( \bigwedge_{h=1}^{h} \neg p_h \) \( \not\equiv \) false. This is impossible for an irreducible DNF expression.

   We prove case c by reductio ad absurdum. If \( p_k \Rightarrow \neg \bigwedge_{h=1}^{h} \neg p_h \), it follows that \( p_k \Rightarrow \bigwedge_{h=1}^{h} \neg p_h \). The expression \( p_k \vee \cdots \vee p_{k-1} \vee p_k \vee p_{k+1} \vee \cdots \vee p_h \) is therefore equivalent to \( p_k \vee \cdots \vee p_{k-1} \vee p_{k+1} \vee \cdots \vee p_h \), that is, the removal of \( p_k \) does not affect the result of the expression. Hence, \( p_1 \vee \cdots \vee p_{k-1} \vee p_{k+1} \vee \cdots \vee p_h \) is not an irreducible DNF expression. This contradicts the assumed premise.

### TABLE 18

**Decision Table for DayError**

<table>
<thead>
<tr>
<th>DayError(d, m, y)</th>
<th>mType(m) = M_31</th>
<th>mType(m) = M_30</th>
</tr>
</thead>
<tbody>
<tr>
<td>d &lt; 1 ( \lor ) d ( \geq ) 31</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>d ( \geq ) 31</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>d ( \geq ) 30</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>d ( \geq ) 29</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>d ( \geq ) 28</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>YType(y) = CYear</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>impossible</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>
Lemma 2. Suppose \( \bigvee_{k=1}^{h}(r_{k}^{1} \land \cdots \land r_{k}^{d_k}) \) is an irreducible DNF expression that is not equivalent to true or false. At least one solution can be found for the constraint \( (r_{k}^{1} \land \cdots \land r_{k}^{t_{kl}}) \land \bigwedge_{k=1, k \neq k}^{h}(r_{k}^{1} \land \cdots \land r_{k}^{d_k}) \), 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 \( r_{1}^{1} \land \cdots \land r_{1}^{t_{kl}} \).

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 \( r_{1}^{1} \land \cdots \land \neg r_{1}^{1} \land \cdots \land r_{1}^{d_{1}} \). We prove by case by reductio ad absurdum. If no solution exists for this constraint, \( r_{1}^{1} \land \cdots \land \neg r_{1}^{1} \land \cdots \land r_{1}^{d_{1}} \) is equivalent to false. Since the expression is in irreducible DNF, neither \( \neg r_{1}^{1} \land \cdots \land \neg r_{1}^{d_{1}} \) nor \( r_{1}^{1} \land \cdots \land \neg r_{1}^{d_{1}} \) is equivalent to false. Hence, \( \neg r_{1}^{1} \Rightarrow \neg (r_{1}^{1} \land \cdots \land \neg r_{1}^{1} \land \cdots \land \neg r_{1}^{d_{1}}) \), that is, \( r_{1}^{1} \land \cdots \land \neg r_{1}^{1} \land \cdots \land \neg r_{1}^{d_{1}} \Rightarrow r_{1}^{1} \). Therefore, \( r_{1}^{1} \land \cdots \land \neg r_{1}^{1} \land \cdots \land \neg r_{1}^{d_{1}} \equiv r_{1}^{1} \land \cdots \land \neg r_{1}^{d_{1}} \), that is, \( r_{1}^{d_{1}} \) can be removed without changing the result of the expression. This cannot take place in an irreducible DNF expression.

2. \( h > 1 \).

a. \( d_{h} = 1 \). In this case, the constraint \( (r_{h}^{1} \land \cdots \land \neg r_{h}^{1} \land \cdots \land r_{h}^{d_{h}}) \land \bigwedge_{k=1, k \neq h}^{h}(r_{k}^{1} \land \cdots \land r_{k}^{d_{k}}) \) becomes \( r_{h}^{1} \land \bigwedge_{k=1, k \neq h}^{h}(r_{k}^{1} \land \cdots \land r_{k}^{d_{k}}) \). If \( r_{h}^{1} \land \bigwedge_{k=1, k \neq h}^{h}(r_{k}^{1} \land \cdots \land r_{k}^{d_{k}}) = false \), \( r_{h}^{1} \land \bigwedge_{k=1, k \neq h}^{h}(r_{k}^{1} \land \cdots \land r_{k}^{d_{k}}) = true \). This contradicts the premise that the expression \( \bigvee_{k=1}^{h}(r_{k}^{1} \land \cdots \land r_{k}^{d_{k}}) \) is not equivalent to true.

b. \( d_{h} > 1 \). To prove \( (r_{h}^{1} \land \cdots \land \neg r_{h}^{1} \land \cdots \land r_{h}^{d_{h}}) \land \bigwedge_{k=1, k \neq h}^{h}(r_{k}^{1} \land \cdots \land r_{k}^{d_{k}}) \) is not equivalent to false, we need only prove that

i. \( r_{1}^{1} \land \cdots \land r_{1}^{t_{kl}} \land r_{1}^{d_{1}} \land \cdots \land r_{k}^{t_{kl}} \) is not equivalent to false,

ii. \( \neg r_{k}^{1} \land \bigwedge_{k=1, k \neq k}^{h}(r_{k}^{1} \land \cdots \land r_{k}^{d_{k}}) \) is not equivalent to false, and

iii. \( r_{1}^{1} \land \cdots \land r_{1}^{t_{kl}} \land r_{1}^{d_{1}} \land \cdots \land r_{k}^{d_{k}} \) is not equivalent to false.

If \( E_{i,j} \) is not false for \( sp \in \text{NDSP} \), according to Lemma 1, each constraint in \( STCC_{i,j}(S^{D}, \text{NDSP}) \) 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

\[
\begin{align*}
&c_{i,j}^{1,1} \land \cdots \land c_{i,j}^{k,1} \land \bigwedge_{k=1, k \neq j}^{w_{i,j}}(c_{i,j}^{1} \land \cdots \land c_{i,j}^{k,1}) \\
&k = 1, 2, \ldots, w_{i,j}
\end{align*}
\]
with respect to \( sp \). Since
\[
c_{i,j}^1 \land \cdots \land c_{i,j}^{k_s} \land \bigwedge_{k_1=1,k_1 \neq k}^{w_{i,j}} \neg \left( c_{i,j}^{k_1} \land \cdots \land c_{i,j}^{k_{s+1}} \right)
\]
implies the constraint
\[
c_{i,j}^1 \land \cdots \land c_{i,j}^{k_s} \land \bigvee_{k_1=1,k_1 \neq k}^{w_{i,j}} \left( c_{i,j}^{k_1} \land \cdots \land c_{i,j}^{k_{s+1}} \right)
\]
it follows that \( t_{i,j} \) also satisfies this constraint so that \( (t_{i,j},O(i,j)) \in WT(S^B, NDSP) \).

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

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

**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, \( S^F(NDSP) \supset S^B(NDSP) \). 2) The basic meaningful impact strategy conditionally subsumes fault-based testing for the LOF and LIF classes of faults, that is, \( S^B(NDSP) \supset S^F(NDSP) \).

**Proof.**

\[
STCC(S^B, NDSP) = \left( c_{i,j}^{k_1} \land \cdots \land c_{i,j}^{k_s} \land \bigwedge_{k_1=1,k_1 \neq k}^{w_{i,j}} \neg \left( c_{i,j}^{k_1} \land \cdots \land c_{i,j}^{k_{s+1}} \right) \right)_{O(i,j,k)} \oplus \left( c_{i,j}^{k_1} \land \cdots \land c_{i,j}^{k_s} \land \bigwedge_{k_1=1,k_1 \neq k}^{w_{i,j}} \neg \left( c_{i,j}^{k_1} \land \cdots \land c_{i,j}^{k_{s+1}} \right) \right)_{O(i,j,k)}
\]

In fault-based testing, the list of test case constraints for LOF is
\[
\left( \neg c_{i,j}^{k_1} \land \bigwedge_{k_1=1,k_1 \neq k}^{w_{i,j}} c_{i,j}^{k_1} \land \bigwedge_{k_1=1,k_1 \neq k}^{w_{i,j}} \neg \left( c_{i,j}^{k_1} \land \cdots \land c_{i,j}^{k_{s+1}} \right) \right)_{O(i,j,k)}
\]

The list of test case constraints for LIF is
\[
\left( c_{i,j}^{k_1} \land \cdots \land \neg c \land \bigwedge_{k_1=1,k_1 \neq k}^{w_{i,j}} \neg \left( c_{i,j}^{k_1} \land \cdots \land c_{i,j}^{k_{s+1}} \right) \right)_{O(i,j,k)}
\]

Let \( STCC_1(S^B, NDSP), STCC_2(S^B, NDSP), \ldots, STCC_r(S^B, NDSP) \) denote the sublists in \( STCC(S^B, NDSP) \) such that

\[
STCC(S^B, NDSP) = STCC_1(S^B, NDSP) \oplus STCC_2(S^B, NDSP) \oplus \cdots \oplus STCC_r(S^B, NDSP).
\]

We can write \( STCC_1(S^F, NDSP) \) in the following format:
\[
\left( \bigwedge_{k_1=1} c_{i,j}^{k_1} \land \neg c \land \bigwedge_{k_1=1,k_1 \neq k}^{w_{i,j}} \neg \left( c_{i,j}^{k_1} \land \cdots \land c_{i,j}^{k_{s+1}} \right) \right)_{O(i,j,k)}
\]

where \( c_{i,j}^{k_1}, c_{i,j}^{k_2}, \ldots, c_{i,j}^{k_{s+1}} \) are the elements in \( L_{i,j} - L_{i,j}^k \).

The sublist \( (c_{i,j}^{k_1} \land \cdots \land c_{i,j}^{k_{s+1}}) \) is a test case that satisfies the constraint \( \bigwedge_{k_1=1,k_1 \neq k}^{w_{i,j}} \neg \left( c_{i,j}^{k_1} \land \cdots \land c_{i,j}^{k_{s+1}} \right) \) with respect to \( sp \). The sublist \( (c_{i,j}^{k_1} \land \cdots \land c_{i,j}^{k_{s+1}}) \) is a test case that satisfies \( (c_{i,j}^{k_1} \land \cdots \land c_{i,j}^{k_{s+1}}) \) with respect to \( sp \).

**Theorem 4.** Fault-based testing for the LOF and LIF classes of faults conditionally subsumes decision table-based testing over NDSP, that is, \( S^F(NDSP) \supset S^B(NDSP) \).

**Proof.** The proof is similar to that of Theorem 3. \( \square \)

**Theorem 5.** Fault-based testing for the LOF and LIF classes of faults conditionally subsumes the partition strategy over NDSP, that is, \( S^F(NDSP) \supset S^B(NDSP) \).

**Proof.** Clearly, if there exists \( k (1 \leq k \leq w_{i,j}) \) for each \( E_{i,j} \) such that
\[
\bigwedge_{k_1=1}^{k_{s+1}} c_{i,j}^{k_1} \land \neg c \land \bigwedge_{k_1=1,k_1 \neq k}^{w_{i,j}} \neg \left( c_{i,j}^{k_1} \land \cdots \land c_{i,j}^{k_{s+1}} \right)
\]
exists or \( s_{i,j}^k = 1 \) with respect to \( sp \), then \( S^F(NDSP) \supset S^B(NDSP) \); otherwise, \( S^F(NDSP) \) does not subsume \( S^D(NDSP) \).

**Acknowledgments**

This research is sponsored by Science Foundation Ireland (SFI) under grants 01/P12/C009 and 03/CE3/1405, and the General Research Fund of the Research Grants Council of Hong Kong under project no. 717308.

**References**


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 QSC 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. 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.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.