Using the Incremental Approach to Generate Test Sets: A Case Study*

Y. T. Yu
Department of Computer Science
City University of Hong Kong
Kowloon Tong, Hong Kong
csytYu@cityu.edu.hk

M. F. Lau† and T. Y. Chen
School of Information Technology
Swinburne University of Technology
Hawthorn 3122, Australia
{elau,tychen}@it.swin.edu.au

Abstract

With the increasing complexity of software systems, the set of testing requirements can become very large. If the set of testing requirements can be naturally decomposed into smaller subsets, one may construct a test set separately to satisfy each subset of testing requirements, and then combine the test sets to form the complete test set. Such an approach is referred to as the union approach. On the other hand, the incremental approach attempts to incrementally expand a test set to satisfy the subsets of testing requirements, one at a time. This paper investigates empirically the effect of the incremental approach as compared to the union approach. Our case study indicates that the incremental approach can result in a significantly smaller test set, particularly when supplemented with the greedy heuristics.

Keywords: Empirical study, fault-based testing, partition testing, specification-based testing, test case generation

1. Introduction

When testing a program, test cases are constructed to satisfy a set of testing requirements. A typical method of constructing the test cases that satisfy the set of testing requirements is to generate a test case for each requirement. This method has been employed by most automated test case generators [15, 22]. As the cost of testing is expensive, it is desirable to have a smaller test set that yet satisfies all testing requirements.

There are many different ways to construct a smaller test set. One approach is to select a subset of test cases from the original test set, such that the same set of testing requirements is satisfied. This is known as the test suite reduction problem. Since finding the optimal solution to

---

*The work is supported by a Discovery Grant from the Australian Research Council (project ID: DP0345147), in part by a grant from City University of Hong Kong (project no. 7001233) and a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (project no. CityU 1083/00E).

†Corresponding author.
Alternatively, one may generate test cases incrementally to satisfy all def-uses pairs of a particular variable, one at a time, hoping that these test cases may be reused to satisfy some def-uses pairs of other variables as well.

In [20], Schroeder and Korel consider the generation of black box test cases for programs with multiple inputs and outputs. The problem is to generate different input combinations that can exercise all outputs. This is in fact a combinatorial problem. Noting that some outputs may depend only on some but not all of the inputs, they propose to generate test cases that exercise combinations of only the inputs that affect each particular output, considering one output at a time. The set of test cases for each output can then be combined by using the union approach or the incremental approach, hoping that some test cases for one output can be reused for another output.

This paper investigates the effect of the incremental approach on the generation of smaller test sets that satisfy the MUMCUT testing strategy [8], and compares it to the union approach. A MUMCUT test set is a test set that satisfies the MUMCUT testing strategy. Since the MUMCUT testing strategy involves three different but inter-related component strategies, the relationship among these three strategies makes the problem of generating MUMCUT test sets interesting.

The rest of this paper is organised as follows. Section 2 introduces the concepts, notation, and terminologies related to the MUMCUT strategy. Section 3 discusses how the incremental approach can be applied to generate a MUMCUT test set. Section 4 describes our experiments and reports the results. Section 5 discusses some of the previous work related to the application of the incremental approach to software testing. Section 6 summarises and concludes the paper.

2. Preliminaries

2.1. Notation and terminology

Let \( \mathbb{B} = \{0, 1\} \), where 0 and 1 denote the truth values ‘FALSE’ and ‘TRUE’, respectively. We denote the Boolean operators AND, OR, and NOT by ‘\&’, ‘\lor’, and ‘\neg’, respectively. The symbol ‘\‘ will be omitted whenever it is clear from the context. A Boolean expression \( S \) of \( m \) terms and \( n \) variables in disjunctive normal form is given by

\[
S(\vec{t}) = p_1(\vec{t}) + p_2(\vec{t}) + \cdots + p_m(\vec{t})
\]

where \( \vec{t} = (t_1, \ldots, t_n) \in \mathbb{B}^n \), and \( p_i \) is the \( i \)-th term of \( S \).

A literal is an occurrence of a variable in a Boolean expression. A Boolean expression \( S \) in disjunctive normal form is said to be irredundant if none of its terms nor any of its literals may be omitted from the expression without changing the function which \( S \) defines.

A test case \( \vec{t} \) for a Boolean expression \( S \) is a point in the Boolean space \( \mathbb{B}^n \). A point \( \vec{t} \in \mathbb{B}^n \) is a unique true point of \( p_i \) in \( S \) if \( p_i \) evaluates to 1 on \( \vec{t} \) and \( p_j \) evaluates to 0 on \( \vec{t} \) for all \( j \neq i \). We use \( UT P_i(S) \) to denote the sets of all unique true points of \( p_i \) in \( S \), respectively. The set of all unique true points of \( S \) is given by \( UT P(S) = \bigcup_i UT P_i(S) \).

Let the \( i \)-th term of \( S \) be \( p_i = x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n} \), where \( x_j \) is the \( j \)-th literal in \( p_i \) and \( k_i \) is the number of literals in \( p_i \). We use \( p_{i,j} = x_1^{i_1} \cdots x_{j-1}^{i_{j-1}} \cdot \bar{x}_j^{i_j} \cdots x_n^{i_n} \) to denote the term obtained from \( p_i \) by negating its \( j \)-th literal \( x_j \). A point \( \vec{t} \) is said to be a near false point for the \( j \)-th literal \( x_j \) of the \( i \)-th term \( p_i \) in \( S \) if \( S \) evaluates to 0 on \( \vec{t} \) but \( p_{i,j} \) evaluates to 1 on \( \vec{t} \). The set of all near false points for \( x_j \) of \( p_i \) in \( S \) is denoted by \( N F P_{i,j}(S) \), and the set of all near false points of \( S \) is given by \( N F P(S) = \bigcup_{i,j} N F P_{i,j}(S) \).

As an example, Table 1 summarises the sets of unique true points and near false points for the specification \( S = ab + cde \), which will be referred to in later sections.

**2.2. Boolean expressions in use**

Boolean expressions or their equivalent forms are commonly used in software specifications or program predicates. For example, the SCR specification has been used to specify many industrial systems including cruise control system [2] and nuclear power plant control system [13]. SCR requirements can be formalized with logic-model semantics so that they can be analysed by model checkers such as Symbolic Model Verifier (SMV) [3]. In a SCR specification, the mode transition table is commonly used to specify the conditions and events that trigger the transitions between modes. Recently, Offutt et al. [18] have shown how to convert the mode transition table of a SCR specification into Boolean expressions from which test cases are generated.

Another well-known specification technique that involves Boolean expressions is the RSM, developed by Leveson et al. [17], for specifying safety critical systems.

<table>
<thead>
<tr>
<th>( i )</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( UT P_1(S) )</td>
<td>10000, 10001, 10010, 10110</td>
<td>00111, 01111, 11111</td>
</tr>
<tr>
<td>( N F P_{1,1}(S) )</td>
<td>00000, 00001, 00010, 00110</td>
<td>00011, 01011, 11111</td>
</tr>
<tr>
<td>( N F P_{1,2}(S) )</td>
<td>11000, 11001, 11010, 11110</td>
<td>00101, 01101, 11111</td>
</tr>
<tr>
<td>( N F P_{1,3}(S) )</td>
<td>–</td>
<td>00110, 01110, 11110</td>
</tr>
</tbody>
</table>
The state transitions of the system are specified by diagrams similar to the Statechart, whereas the conditions for the transitions are expressed in AND/OR tables, which are tabular representations of Boolean expressions. It is essential to verify that the transitions in these systems are correctly implemented as specified.

Furthermore, complex Boolean expressions frequently appear as predicates in program codes for safety critical systems [9]. Ensuring the correctness of these predicates is very important and yet can be a difficult task.

### 2.3. The MUMCUT testing strategy

In recent years, the problem of generating test cases from Boolean expressions has been under active research. Various testing strategies have been proposed, including the modified condition/decision criterion (MC/DC) [9, 14], the family of meaningful impact strategies [23], the Boolean OperatoR (BOR) strategy [21], the MUMCUT strategy [7,8,24] and the full predicate coverage criterion [18]. In particular, MC/DC has been in use in the industry for more than a decade, and is mandatory for predicate testing of flight-critical avionic programs [9]. On the other hand, the MUMCUT strategy [8], which is an integration of three component strategies (MUTP, MNFP and CUPMFP), is the only one that has been proved to guarantee the detection of seven types of fault in Boolean expressions in irredundant disjunctive normal form [8,24].

The MUTP strategy selects enough test points from $UTPi(S)$ so that all possible truth values of every variable not occurring in the $i$-th term are covered for every $i$. The MNFP strategy selects enough test points from $NFP_{i,j}(S)$ so that all possible truth values of every variable not occurring in the $i$-th term are covered for every $i$ and every $j$. The CUTPFP strategy requires, for every $i$ and $j$, a unique true point $\tilde{t}$ from $UTPi(S)$ and a near false point $\tilde{t}$ from $NFP_{i,j}(S)$ such that $\tilde{t}$ and $\tilde{t}$ differ only in the truth value of the $j$-th literal $x_i^j$ in the $i$-th term. A test set is said to satisfy the MUMCUT strategy (and called a MUMCUT test set) if it satisfies all the three component strategies.

### 3. The Incremental Approach for generating test sets

Since the MUMCUT strategy is an integration of three component strategies (CUPMFP, MUTP and MNFP), the problem of constructing a MUMCUT test set can be naturally decomposed into three subproblems:

1. find a test set $C$ that satisfies the CUPMFP strategy;
2. find a test set $U$ that satisfies the MUTP strategy; and
3. find a test set $N$ that satisfies the MNFP strategy.

Using the union approach, a MUMCUT test set can be obtained by the union of the sets $C$, $U$ and $N$.

Consider now the sub-problem of finding a test set $C$ to satisfy the CUPMFP strategy. Since the sets $UTPi(S)$ are mutually exclusive for all $i$, the problem of finding corresponding pairs from $UTPi(S)$ and $NFPi(S)$ can be further decomposed into the following “sub-subproblems”:

1.1 for each $i$, collect unique true points from $UTPi(S)$ and near false points from $NFP_{i,j}(S)$ (for every $j$) to $C_i$ such that they form corresponding pairs, and

1.2 obtain the test set $C$ from $C = \bigcup_i C_i$.

There are two different ways to generate the set $C_i$. One obvious way is to randomly generate a unique true point $\tilde{t}_i$ from $UTPi(S)$. If $\tilde{t}_i$ can form yet-to-be-formed corresponding pairs with some near false points in $NFP_{i,j}(S)$ for some $j$, these corresponding pairs ($\tilde{t}_i$ and the corresponding near false points) will be added to $C_i$. The process of finding corresponding pairs continues until all such pairs are found or all points in $UTPi(S)$ are exhausted. Alternatively, we can apply the greedy heuristics to select a point from $UTPi(S)$ such that it forms the largest number of yet-to-be-formed corresponding pairs with near false points in $NFP_{i,j}(S)$ for all possible $j$.

The sub-problem of finding the set $U$ to satisfy the MUTP strategy can also be decomposed into the following “sub-subproblems”:

2.1 collect points from $UTPi(S)$ to $U_i$ so that they cover all possible truth values of every missing variable for each $i$, and

2.2 obtain the test set $U$ from $U = \bigcup_i U_i$.

For the construction of $U_i$, test cases can be randomly generated from $UTPi(S)$ to cover some yet-to-be-covered truth values of some missing variables. Alternatively, we can apply the greedy heuristics to select an element from $UTPi(S)$ that can cover the most yet-to-be-covered truth values for every missing variable. The sub-problem of finding the test set $N$ that satisfies the MNFP strategy is similar to that of finding the test set $U$ and is omitted.

Up to now, we have presented two methods to generate a test set, namely the random union (or, simply R-Union) method, where test cases are randomly generated, and the greedy union (or, simply G-Union) method, where test cases are constructed using the greedy heuristics.

We next consider the incremental approach. Since the CUPMFP strategy requires the selection of unique true points from $UTPi(S)$ and near false points from $NFP(S)$ such that they form corresponding pairs, it is considered to be the most difficult to satisfy among the three strategies. If this strategy is satisfied first, the selection of unique true points for the MUTP strategy and the selection of near false points for the MNFP strategy will probably be easier.
By adopting the “most-constrained-first” principle, fewer test cases may be required at the later stages. With this approach, \( M \) is first initialised as an empty set and then gradually augmented to become a MUMCUT test set as follows.

1. Expand \( M \) by collecting points from \( UTP(S) \) and \( NFP(S) \) until \( M \) satisfies the CUTPNFP strategy.

2. Expand \( M \) by collecting additional points from \( UTP(S) \) until \( M \) also satisfies the MUTP strategy.

3. Further expand \( M \) by collecting additional points from \( NFP(S) \) until \( M \) also satisfies the MNFP strategy.

On completion of Stage (3), \( M \) is necessarily a MUMCUT test set.

Again, there are two ways of generating points at every stage, namely, random and greedy. Thus, we have the random incremental (or, simply R-Incremental) method, where test cases are randomly generated as needed, and the greedy incremental (or, simply G-Incremental) method, where test cases are generated using the greedy heuristics.

We are interested in the question of which of the following four methods will deliver a smaller test set: R-Union, G-Union, R-Incremental and G-Incremental.

Since the incremental approach attempts to “reuse” as many points collected at each stage as possible, and collects more points only if necessary to construct the test set for the next stage, it is reasonable to believe that the incremental approach will deliver a smaller test set than the union approach. Also, the greedy approach is usually expected to perform better than the random generation approach. Thus, one would expect the greedy incremental method to deliver a smaller test set and the random union method to deliver a larger test set. However, Example 1 below shows that this is not always true.

**Example 1** Consider the specification \( S = ab + cde \) whose sets of unique true points and near false points are shown in Table 1.

Figure 1 shows diagrammatically how the MUMCUT test set \( \{0001, 0011, 0010, 0011, 0011, 10001, 10110, 11001, 11011, 11110, 11111\} \), containing 12 points, can be generated by the R-Union method.

Figure 2 shows a MUMCUT test set generated by the G-Incremental method for \( S = ab + cde \).
Table 2: Size of exhaustive test sets

<table>
<thead>
<tr>
<th>Spec. id.</th>
<th>No. of variables</th>
<th>Size of exhaustive test set</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>128</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>512</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>4096</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>512</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>2048</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>1024</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>256</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>128</td>
</tr>
<tr>
<td>10</td>
<td>13</td>
<td>8192</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
<td>8192</td>
</tr>
<tr>
<td>12</td>
<td>14</td>
<td>16384</td>
</tr>
<tr>
<td>13</td>
<td>12</td>
<td>4096</td>
</tr>
<tr>
<td>14</td>
<td>7</td>
<td>128</td>
</tr>
<tr>
<td>15</td>
<td>9</td>
<td>512</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>4096</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>2048</td>
</tr>
<tr>
<td>18</td>
<td>10</td>
<td>1024</td>
</tr>
<tr>
<td>19</td>
<td>8</td>
<td>256</td>
</tr>
<tr>
<td>20</td>
<td>7</td>
<td>128</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>9.7</strong></td>
<td><strong>2689.6</strong></td>
</tr>
</tbody>
</table>

Example 1 shows that the greedy incremental approach is not always better than the random union approach. Yet intuitively one would expect that the incremental approach will have a higher chance to be better than the union approach, and the greedy approach to be better than the random generation approach. To validate the intuition and to quantify the effects of these approaches, we performed an empirical study as detailed in the next section.

### 4. The empirical study and results

In this study, we use the same 20 Boolean specifications which were originated from the specification of TCAS II, an aircraft collision avoidance system [17] and published in [23]. Table 2 shows the number of variables and the total number of test cases for exhaustive testing for each of the 20 Boolean specifications.

As in [23], after transforming these specifications into irredundant disjunctive normal form, we apply the methods as discussed in Section 3 to generate the corresponding test sets. For most of these 20 specifications, the number of possible MUMCUT test sets generated can be very large [8]. Whenever the number of possible MUMCUT test sets exceeds 1000, we arbitrarily generate 1000 test sets and record their sizes. Otherwise, we generate all possible MUMCUT test sets.

In the analysis, we ask the following questions:

Q.1 How often is the incremental approach better than the union approach? Does the incremental approach make a significant improvement over the union approach, and if so, by how much on average?

Q.2 How often is the greedy approach better than the random generation approach? Does the greedy approach make a significant improvement over the random generation approach, and if so, by how much on average?

Q.3 If it is as expected that the incremental approach is better than the union approach and the greedy approach is better than the random generation approach, which of the two effects is more prominent?

Figure 3 shows the mean sizes of the MUMCUT test sets generated from the 20 Boolean specifications by the four methods under study. For specifications 2, 8, 9 and 20, only one MUMCUT test set can be formed, and hence the size is the same for all four methods. Table 3 shows the average sizes of test sets for all 20 specifications under study, and the performance improvements of each method over another method.

Our study provides the following empirical answers.

A.1 From Figure 3, we have the following observations:

(a) The mean sizes of the MUMCUT test sets generated by the greedy incremental method are always smaller than or equal to those generated by the greedy union method.

(b) The mean sizes of the MUMCUT test sets generated by the random incremental method are always smaller than or equal to those generated by the random union method.

These observations confirm that the incremental approach is usually better than the union approach, whether combined with greedy or random generation.

Table 3 shows that (1) the improvement of the greedy incremental method over the greedy union method is about 52% on average; and (2) the improvement of the random incremental method over the random...
union method is about 34% on average. Thus, the incremental approach results in a significant reduction in the size of test sets over the union approach.

A.2 From Figure 3, we also notice the following:

(a) The mean sizes of the MUMCUT test sets generated by the greedy incremental method are always smaller than or equal to those generated by the random incremental method.

(b) The mean sizes of the MUMCUT test sets generated by the greedy union method are smaller than or equal to those generated by the random union method, except for specifications 16 and 18. For each of specifications 16 and 18, the difference between the mean sizes of the MUMCUT test sets is insignificant. Thus, the greedy union method is better than the random union method most of the time.

These observations confirm that the greedy heuristics is usually better than random generation, whether used with the incremental or union approach.

Table 3 shows that (1) the improvement of the greedy incremental method over the random incremental method is about 25% on average; and (2) the improvement of the greedy union method over the random union method is about 10% on average. Thus, the greedy approach results in a good reduction in size of test sets over the random generation approach.

A.3 Based on the above, the effect of test set size reduction that can be achieved by applying the incremental approach over the union approach (34% or 52%) is far more significant than applying the greedy approach over the random generation approach (10% or 25%).

5. Related Work

The incremental approach has been considered in other contexts of software testing. In this section, we discuss some of the related work.

5.1. Incremental Regression Testing

The problem of incremental regression testing is formalized as: Given a program $P$, a test set $T$ that satisfies a particular testing criterion for $P$, and a program $P'$ which is newly modified from $P$, the aim is to generate a test
set \( T' \) for the program \( P' \) such that \( T' \) satisfies the same criterion by reusing as many test cases in \( T \) as possible.

Most incremental regression testing techniques [4, 11] discuss how to generate test cases from the program based on a particular test data adequacy criterion such as branch coverage or all-uses data-flow coverage. For example, Bates and Horwitz [4] propose the use of program dependence graphs as a basis for incremental regression testing when using test data adequacy criteria. Their idea is to identify those test cases in \( T \) that can be reused in testing \( P' \) via the program dependence graphs of \( P \) and \( P' \). However, new test cases have to be generated to satisfy those yet-to-be-satisfied components (e.g., all-uses pair) after all test cases in \( T \) have been considered.

On the other hand, Agrawal et al. [1] recognise that regression testing actually consists of two different but related problems. The first problem is the correctness of the modified program \( P' \) for those functionalities that need to be modified or implemented. The second problem is to ensure that \( P' \) behaves correctly for those functionalities that \( P \) behaves correctly. Hence, it is also important to generate test cases from the existing test suite \( T \) that can reveal the differences between \( P \) and \( P' \) on such functionalities. They then propose to use program slicing techniques to identify test cases from the existing test suite \( T \) that may cause the outputs of \( P \) and \( P' \) to be different.

5.2. Incremental Test Data Generation

Gupta and Soffa [10] propose a technique that groups testing requirements based on the data-flow analysis of the source code, arrange those requirements in order, and generates a test case that satisfies a group of requirements, one group at a time, until all requirements are satisfied. As a result, the resources spent on test case generation would be reduced. The technique is based on the data-flow analysis of the program source. Hence, whenever the program needs to be changed, testers need to generate a new test set for the modified program. This latter problem becomes the concern of the incremental regression testing.

Schroeder and Korel [20] consider the problem of generating test cases using black box methods (such as equivalence partitioning and boundary value analysis) for programs with multiple inputs and outputs. Their focus is on the generation of combinations of the inputs that can affect the output of the program under test. For a program with \( m \) inputs, if each input needs to be verified for at least 2 different values or has at least 2 equivalence classes to be considered, the total number of possible situations needed to be tested is at least \( 2^m \), causing a combinatorial explosion problem. They observe that in some situations some outputs may not depend on certain inputs. In other words, no matter how the values of certain inputs are changed, some outputs will remain unaffected. Hence they propose to identify the input-output relationship of the program and then generate test cases in two steps:

1. For each output of the program, generate a test set by considering only inputs that may affect the output, and
2. Merge the resulting test sets to form the final test set.

They illustrate by a case study that a simple union of the test sets obtained in Step (1) may not result in a minimal test set. However, they have not specifically proposed any alternatives other than the brute force method.

Our work in this paper differs from previous work in the four aspects. First, test cases are generated Boolean expressions to satisfy the MUMCUT strategy. Secondly, the MUMCUT strategy is a fault-based testing strategy, as it guarantees to detect seven types of fault that may appear in Boolean expressions in irredundant disjunctive normal form [7]. In other words, whichever approach used for generating a smaller test set will never result in any loss of its ability to detect these types of faults. Thirdly, the MUMCUT strategy is a combination of three different but inter-related strategies, namely, the CUTPNFP, the MUTP and the MNFP strategies [7]. Unlike other techniques in [10, 20], extra effort to identify or group different requirements are not necessary. Finally, we study different methods to incrementally generate test cases and compare their effectiveness using published specifications taken from TCAS II, a safety critical system used in the industry.

6. Summary and conclusion

This paper investigates the effect of the incremental approach as compared to the union approach on generating smaller test sets to satisfy the MUMCUT strategy [8]. The empirical results show that the incremental approach is usually better than the union approach in delivering smaller test sets. Moreover, the greedy approach is usually better than the random generation approach. On average, the improvement of the incremental approach over the union approach is more prominent than the improvement of the greedy approach over the random generation approach.

The MUMCUT strategy comprises three component strategies: CUTPNFP, MUTP and MNFP, among which the CUTPNFP strategy is the most difficult to satisfy. In this paper, when using the incremental approach, we have adopted the “most-constrained-first” principle, whereby the CUTPNFP strategy is satisfied first, followed by the MUTP and MNFP strategies. Obviously, other permutation orders are possible. A more comprehensive comparison among different orders is needed to determine which order is the best. In a separate experiment reported elsewhere [24], the MUMCUT strategy is compared with other existing strategies that detect the same types of fault, and the relationship between the size of a MUMCUT test
set and that of the Boolean specification is explored. In that paper, the greedy but not the random generation approach has been used and combined with different orders, showing that the order which we are using here is indeed one of the best among the various possible permutations. Interestingly, there is another order which has a similar and comparable performance.

Further work on using the incremental approach to generate test sets is needed. For instance, as discussed in Section 1, there are different ways to apply the incremental approach for the all def-uses pairs [19]. Empirical studies are needed to find out ways to generate smaller test sets to satisfy the all def-uses pairs criterion. Another instance is to study the effect of the incremental approach on the size of test sets generated for testing programs with multiple inputs and outputs using the methods of Schroeder and Korel [20]. In particular, we would like to find out the improvements, if any, of the incremental approach over the simple union method under these circumstances.

Acknowledgement

We are grateful to Emily Wong who has helped in conducting the empirical study and analysis in this paper.

References


