Efficient Test Data Generation for Variables with Complex Dependencies

Armin Beer  
Siemens Austria  
armin.beer@siemens.com

Stefan Mohacsi  
Siemens Austria  
stefan.mohacsi@siemens.com

Abstract

This paper introduces a new method for generating test data that combines the benefits of equivalence partitioning, boundary value analysis and cause-effect analysis. It is suitable for problems involving complex linear dependencies between two or more variables. The method aims at covering all semantic dependencies plus all (n-dimensional) boundaries with a minimum set of test data.

To overcome the mathematical complexity of the method, a main goal of the research project was to develop a user-friendly tool that allows users to specify dependencies in a simple language and generates appropriate test data automatically. The tool has been incorporated into the IDATG (Integrating Design and Automated Test case Generation) tool-set and validated in a number of case studies.

1. Introduction

In general, testing is defined as “an activity performed for evaluating product quality, and for improving it, by identifying defects and problems” [1]. Therefore, the test object is run with a set of inputs in a defined context, and the actual behaviour and results are compared with the expected behaviour and results to determine the presence of a defect. Several systematic test design techniques and testing approaches have been developed to assist in systematically exploring the available choices and selecting a representative sample [2][3]. Criteria for selecting such a sample are typically based on requirements, design models, control flow, data flow or statistical measures.

Test design techniques can be divided into methods for finding sequences of test steps and for finding test data. A combination of both is required to create complete test cases consisting of steps and input data [4].

In the following, we will focus on test data design methods. Obviously, testing all possible input combinations is not feasible for most test objects. Even a single integer or string input parameter can have so many different values that complete test coverage is out of the question.

Thus, the purpose of test data design methods is to find a set of input values that can be tested with reasonable effort but still has high fault-detection potential.

One of the most commonly used approaches is known as the equivalence partitioning technique that is recommended by various standards, e.g. IEC61508 [4][5]. According to this method, the tester divides the range of possible values into different sets (equivalence partitions) so that each set only consists of members that cause equivalent program behaviour. Instead of testing with every possible input value, only a few members of each set are used, thus reducing the number of test cases to a reasonable figure.

Although studies by Duran and others argued that random testing would be equivalent to partition testing as regards error-finding performance [6][7][8], they were corrected by others including Hamlet and Taylor, who even claimed that equivalence partitioning would – under certain conditions – be far superior to random testing methods [9]. Their findings were later confirmed in studies on tests conducted in practical cases.

The efficiency of random testing can be improved significantly by applying statistical methods [10]. While such methods undoubtedly have their benefits, they are impractical for complex specifications that include many dependencies between the input parameters. The main reason is the enormous number of test cases that has to be generated and executed in order to achieve an adequate test coverage.

Equally, the simple equivalence partitioning method is insufficient in many cases because dependencies between the input parameters are not taken into account. The situation can be slightly improved by covering all pairwise combinations of equivalence
partitions, but important defects might still be overlooked. The problem is that the key concept of how input variables interact is missing [11][12].

The purpose of this paper is to solve this problem by introducing a new method of test data generation and to analyse its applicability, benefits and limitations.

2. Foundations

In the following, we will give a brief overview of the test data design methods on which our work has been based.

2.1. Simple equivalence partitions

One of the most commonly used approaches is the 

*equivalence partitioning* technique, i.e. dividing the input space of a variable into partitions (contiguous subspaces) so that all members of a partition cause the same program behaviour. Subsequently, only one representative of each partition is picked out as a test value.

A common improvement of the simple equivalence partitioning method is **boundary value analysis** in which the lower and upper boundaries (sometimes also a 3rd value) of a partition are used rather than a random value inside the partition. Experience has shown that the fault detection potential of boundary value analysis is greater than that of most other test data design methods [13].

2.2. Multi-dimensional equivalence partitions

Unfortunately, in many cases the partitioning does not depend on a single input alone but on a combination of different inputs. The problem here is the possible presence of constraints defining dependencies of one input from another. For instance, a triangle can only comprise three sides such that the dependencies of one input from another. For instance, the possible presence of constraints defining combination of different inputs. The problem here is not depend on a single input alone but on a input space of a variable in to partitions (contiguous partitioning method is value.

A common improvement of the simple equivalence partitioning method is **boundary value analysis** in which the lower and upper boundaries (sometimes also a 3rd value) of a partition are used rather than a random value inside the partition. Experience has shown that the fault detection potential of boundary value analysis is greater than that of most other test data design methods [13].

D is called a **domain** of the predicate $\phi$. By expanding this concept, we can formulate a definition for n-dimensional equivalence partitions: the n-dimensional domain D of the predicate $\phi$ can be called an **n-dimensional equivalence partition** if:

- D is a closed space and
- $\phi$ is a calculation rule that is satisfied by all $(x_1,...,x_n) \in D$ but by no $(x_1,...,x_n) \notin D$.

**Example:** Consider an application that receives three integers representing the side lengths of a triangle and returns the triangle type ("equilateral", "isosceles", "scalene", or "invalid"). The set of allowed input tuples is

$$I = \{(x_1,x_2,x_3) \in \mathbb{Z} \mid (x_1 \leq \text{MAXSIZE}) \land (x_2 \leq \text{MAXSIZE}) \land (x_3 \leq \text{MAXSIZE})\}.$$ 

The equivalence partition "equilateral" is defined by

$$P = \{ (x_1,x_2,x_3) \in I \mid (x_1 = x_2) \land (x_1 = x_3) \}$$

and contains all tuples $\{(1,1,1), (2,2,2), \ldots\}$. For floating point variables, a domain is a continuous space ($D \subset \mathbb{R}$). For integer variables, the space is discrete rather than continuous ($D \subset \mathbb{Z}$). A simple way of defining a discrete domain D is to start with a continuous domain $D' \subset \mathbb{R}$ and then put $D = D' \cap \mathbb{Z}$.

As in the one-dimensional case, the fault detection potential is highest at the partition's **boundaries**. The fault is most simply due to typical programming errors like writing $(x_1 + x_2) = x_3$ instead of $(x_1 + x_2) > x_3$ or $(x_1 > 1)$ instead of $(x_1 > 0)$.

As [14] shows, a boundary in the n-dimensional continuous space can be defined as follows:

Let $D \subset \mathbb{R}$ be a closed domain. An element $E = (x_1, \ldots, x_n) \in D$ is called a boundary state of $D$ if for every $\epsilon > 0$ the ball $B_{\epsilon}(E)$ of radius $\epsilon$ and centre $E$ contains at least one point of $\mathbb{R} \backslash D$. In other words, no matter how small the radius $\epsilon$ is selected, the ball around $E$ always contains at least one point outside of $D$.

For discrete spaces, [14] introduces the term **discrete neighbourhood** of an element $E = (x_1, \ldots, x_n) \in D \subset \mathbb{Z}$:

$$V(E) = \{ (x_1 \pm 1, x_2, \ldots, x_n), (x_1, x_2 \pm 1, \ldots, x_n), \ldots, (x_1, x_2, \ldots, x_n \pm 1) \}$$

$E$ is a boundary state, if $V(E)$ contains at least one point of $\mathbb{Z} \backslash D$.

An alternate definition of a **discrete neighbourhood** also considers points that differ from $E$ in more than one coordinate:

$$W(E) = \{ (y_1, \ldots, y_n) \in \mathbb{Z} \mid \forall i = 1, 2, \ldots, n, \ | y_i - x_i \leq 1 \}$$

Obiously, $V(E) \subset W(E)$. For our purposes, we will use the more general definition $W(E)$. 

Let $\mathcal{J}$ be an invariant predicate that specifies the permissible values of the input variables $x_1, \ldots, x_n$ and $I$ the set of all tuples $(x_1, \ldots, x_n)$ that satisfy the invariant $\mathcal{J}$. If $\phi$ is a predicate representing a semantic condition, the set $D$ defined as a subset of $I$ by $\phi$ is:

$$D = \{ (x_1, \ldots, x_n) \in I \mid (x_1, \ldots, x_n) \text{satisfies } \phi \}$$
2.3. Cause-effect analysis

Another method that uses the dependencies between input variables to identify test data has been introduced by [15]. The logical relationships between the causes and their effects in a component or system are displayed in a cause-effect graph. Every cause is described as a condition that can consist of several input conditions connected with logical operators (e.g., AND, OR, NOT). The effects are treated in the same way and are also depicted in the graph.

The graph must be transformed into a decision table from which the test data can be derived. The following steps are used to transform a graph into a table:

1. Choose an effect.
2. Find combinations of causes that have this effect and combinations that do not have this effect.
3. Add one column to the decision table for each of these cause combinations and the caused states of the remaining effects.
4. Check if decision table entries occur several times and delete them if they do [16].

Cause-effect analysis is well suited for examples in which there are complex dependencies between the input variables and multiple effects that affect the end result. However, it does not take into account the fact that many defects occur at the boundaries of the multi-dimensional equivalence partitions that are implicitly defined by the conditions.

3. The CECIL method

While each of the methods described in the last section has its benefits, they also have some weaknesses: the (n-dimensional) equivalence partition method combined with boundary value analysis has a high fault-detection potential but is difficult to apply if the dependencies between the input variables become too complex. On the other hand, cause-effect analysis is able to deal with complex dependencies but disregards the higher fault probability at the boundaries.

We thus aimed at developing a method for deriving test data that combines the benefits of both approaches. The result is the CECIL method (Cause-Effect Coverage Incorporating Linear boundaries) that will be introduced in the following with the aid of a practical example.

3.1. Task 1: Problem analysis

The first task for the test designer is to analyse the tested application's specification that often exists only as plain text. It is particularly important to identify the input variables, their types and definition ranges.

For complex problems, it is impractical to specify how the end result is calculated from the input variables in a single step. It is better to introduce interim results that are later used to calculate the final result. Such interim results are called effect variables in CECIL.

Example:

We want to test an application that calculates the annual insurance premium for a vehicle (motorcycle, car or van). The basic premium depends on the engine power (in HP) and vehicle type:

<table>
<thead>
<tr>
<th>Table 1 - Basic insurance premium</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motorcycle</td>
</tr>
<tr>
<td>&lt; 25 HP</td>
</tr>
<tr>
<td>25 - 49 HP</td>
</tr>
<tr>
<td>&gt;= 50 HP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Car</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 60 HP</td>
</tr>
<tr>
<td>60 - 99 HP</td>
</tr>
<tr>
<td>&gt;= 100 HP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Van</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 60 HP</td>
</tr>
<tr>
<td>60 - 99 HP</td>
</tr>
<tr>
<td>&gt;= 100 HP</td>
</tr>
</tbody>
</table>

For groups with a higher accident risk, the premium is 20% higher. These groups are: all persons older than 65 years, men younger than 25 and women younger than 21.

Only persons aged between 21 and 65 are allowed to drive a van. To drive a car or motorcycle, a person must be at least 18.

The engine power is a positive integer with up to 4 digits, the age a positive integer with up to 3 digits.

**Analysis:** In this example, we have four input variables: the vehicle Type, the HP, the Age, and the Gender. The definition range for HP is [0...9999], that for Age [0...999], Type and Gender are enumeration types that have to be represented as numbers before they can be processed by the CECIL algorithm (0=Motorcycle, 1=Car, 2=Van; 0=Male, 1=Female).

Thus, the set of possible input tuples is

$I = \{(\text{Type}, \text{HP}, \text{Age}, \text{Gender}) \in \mathbb{Z}^4 \mid (\text{Type} \in \{0..2\}) \land (\text{HP} \in \{0..9999\}) \land (\text{Age} \in \{0..999\}) \land (\text{Gender} \in \{0..1\})\}.$

We note that there are three distinct types of effects that all affect the end result (= the insurance premium):

- The Baseprice which depends on the Type and the HP
- The Extracharge which depends on the Age and the Gender
- Invalid combinations that depend on the Age and the Type.
Baseprice and Extracharge are interim results required for calculating the final end result and are thus effect variables. Invalid is an invariant that is used for invalid combinations of input variables.

3.2. Task 2: Define causes and effects

Next we have to express the dependencies between the variables as cause/effect pairs. The possibility of assigning causes to different effect variables is one of the advantages of the CECIL method because it allows a complex problem to be divided into smaller parts.

Each cause/effect pair consists of:
- a unique ID
- a cause (a predicate \( \varphi \) defined by a Boolean expression)
- an effect defined by the name of an effect variable \( e \) and the value to which it is set, or the invariant Invalid if the cause defines an invalid combination of input variables

Note that a cause can affect more than one effect variable and can therefore appear in more than one cause/effect pair.

Let \( C_{1...k} \) be the sets of all causes that affect the effect variables \( e_{1...k} \) and \( C_{\text{Invalid}} \) be the set of all causes that are assigned to the invariant Incorrect. The following points have to be taken into account:
- The behaviour of the tested application must be deterministic. This means that each possible input tuple \( (x_1,...x_n) \in I \) must satisfy exactly one cause from each set \( C_{1...k} \) or exactly one cause from \( C_{\text{Invalid}} \).
- It is not necessary to include the definition ranges of the input variables in the causes. For instance, \( Age > 65 \) automatically assumes \( Age \leq 999 \).
- Neither is it necessary to explicitly exclude invalid cases. For instance, \( Type = 0 \ AND \ HP < 25 \) automatically assumes that \( Age < 18 \) is not true at the same moment.
- All causes must be linear in nature. This means that it must be possible to transform them into the form \( c_1 x_1 + c_2 x_2 + \ldots + c_n x_n + c_{n+1} \leq 0 \) where \( c_1...c_{n+1} \) are constant numbers and \( x_1...x_n \) variables. In particular, it is not allowed to multiply two variables (\( 5 \ast Age < HP \) is allowed, \( HP \ast Age < 5 \) is not).

Example:

Table 2 shows the textual specification formalized as cause/effect pairs:

<table>
<thead>
<tr>
<th>ID</th>
<th>Cause</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALL_BIKE</td>
<td>Type = 0 &amp; HP &lt; 25</td>
<td>Baseprice = 50 €</td>
</tr>
<tr>
<td>MEDIUM_BIKE</td>
<td>Type = 0 &amp; HP &gt;= 25 &amp; HP &lt; 50</td>
<td>Baseprice = 75 €</td>
</tr>
<tr>
<td>BIG_BIKE</td>
<td>Type = 0 &amp; HP &gt;= 50</td>
<td>Baseprice = 100 €</td>
</tr>
<tr>
<td>SMALL_CAR</td>
<td>Type = 1 &amp; HP &lt; 60</td>
<td>Baseprice = 100 €</td>
</tr>
<tr>
<td>MEDIUM_CAR</td>
<td>Type = 1 &amp; HP &gt;= 60 &amp; HP &lt; 100</td>
<td>Baseprice = 200 €</td>
</tr>
<tr>
<td>BIG_CAR</td>
<td>Type = 1 &amp; HP &gt;= 100</td>
<td>Baseprice = 300 €</td>
</tr>
<tr>
<td>SMALL_VAN</td>
<td>Type = 2 &amp; HP &lt; 60</td>
<td>Baseprice = 200 €</td>
</tr>
<tr>
<td>MEDIUM_VAN</td>
<td>Type = 2 &amp; HP &gt;= 60 &amp; HP &lt; 100</td>
<td>Baseprice = 400 €</td>
</tr>
<tr>
<td>BIG_VAN</td>
<td>Type = 2 &amp; HP &gt;= 100</td>
<td>Baseprice = 600 €</td>
</tr>
<tr>
<td>OLD_PERSON</td>
<td>Age &gt; 65</td>
<td>Extracharge = 20%</td>
</tr>
<tr>
<td>YOUNG_MALE</td>
<td>Gender = 0 &amp; Age &lt; 25</td>
<td>Extracharge = 20%</td>
</tr>
<tr>
<td>YOUNG_FEMALE</td>
<td>Gender = 1 &amp; Age &lt; 21</td>
<td>Extracharge = 20%</td>
</tr>
<tr>
<td>NORMAL_MALE</td>
<td>Gender = 0 &amp; Age &gt;= 25 &amp; Age &lt;= 65</td>
<td>Extracharge = 0%</td>
</tr>
<tr>
<td>NORMAL_FEMALE</td>
<td>Gender = 1 &amp; Age &gt;= 21 &amp; Age &lt;= 65</td>
<td>Extracharge = 0%</td>
</tr>
<tr>
<td>I_TOO_YOUNG</td>
<td>Age &lt; 18</td>
<td>Invalid</td>
</tr>
<tr>
<td>I_VAN TOO YOUNG</td>
<td>Type = 2 &amp; Age &lt; 21</td>
<td>Invalid</td>
</tr>
<tr>
<td>I_VAN TOO OLD</td>
<td>Type = 2 &amp; Age &gt; 65</td>
<td>Invalid</td>
</tr>
</tbody>
</table>
### 3.3. Task 3: Derive test data

Each cause can be interpreted as a predicate \( \varphi \) defining a domain (an \( n \)-dimensional space where \( n \) is the number of input variables, see Section 2.2). As already stated above, causes that affect the same effect variable have to be mutually exclusive, which means that their domains do not intersect. However, the domains of causes that belong to different sets \( C_1 \ldots k \), \( C_{\text{Invalid}} \) can intersect.

A valid input domain is defined by picking one arbitrary cause from each set \( C_1 \ldots k \) and creating the intersection of their domains:

\[
D_v = \{(x_1, \ldots, x_n) \in I \mid (x_1, \ldots, x_n) \text{ satisfies } \varphi_1 \in C_1 \land \ldots \land (x_1, \ldots, x_n) \text{ satisfies } \varphi_k \in C_k \land (x_1, \ldots, x_n) \text{ does not satisfy any } \varphi_{\text{Invalid}} \in C_{\text{Invalid}}\}
\]

Obviously, the number of valid input domains corresponds to the number of possible cause combinations.

An invalid input domain is defined by any cause from \( C_{\text{Invalid}} \):

\[
D_i = \{(x_1, \ldots, x_n) \in I \mid (x_1, \ldots, x_n) \text{ satisfies } \varphi_{\text{Invalid}} \in C_{\text{Invalid}}\}
\]

Note that a domain may be empty or consist of several subspaces that are not necessarily connected to each other. According to our definition, only closed subspaces represent \( n \)-dimensional equivalence partitions, so we have to handle each closed subspace separately. Our goal is to find effective test data by determining boundary points for each equivalence partition using the following algorithm:

1. In order to test valid input data, choose one cause for each effect variable and combine the Boolean expressions of these causes with ‘\( \land \)’ operators.

   For generating invalid data, choose exactly one cause that has the effect Invalid. This cause may be combined with valid effects, but not with other invalid ones (to avoid defect masking).

2. To make sure that no unwanted invalid data is generated, exclude all invalid causes that have not been used in the expression from step 1 in the following way:

\[
\text{expression} \land \neg(\varphi_{\text{Invalid}, 1}) \land \ldots \land \neg(\varphi_{\text{Invalid}, m})
\]

3. Convert the resulting Boolean expression into a form that can be processed by a linear programming algorithm:
   - Convert all inequalities into the form \( c_1 x_1 + c_2 x_2 + \ldots + c_n x_n + c_{n+1} \leq 0 \) where \( c_1, \ldots, c_{n+1} \) are constant numbers and \( x_1, \ldots, x_n \) variables.
   - Convert the entire expression into DNF (Disjunctive Normal Form - all OR (\( \lor \)) operators are brought to the top of the syntax tree).

4. We now have a Boolean expression whose syntax tree has the following structure:
   - At the bottom we have linear inequalities of the form \( c_1 x_1 + c_2 x_2 + \ldots + c_n x_n + c_{n+1} \leq 0 \). Each inequality defines one boundary of an \( n \)-dimensional equivalence partition.
   - The inequalities may be linked by AND (\( \land \)) operators, thus defining the set of all boundaries for one \( n \)-dimensional equivalence partition.
   - At the top, there may be OR (\( \lor \)) operators that divide the intersecting space into several \( n \)-dimensional equivalence partitions that are not necessarily connected to each other. Boundaries have to be found for each of these partitions.

For each equivalence partition, determine boundary states by using a linear programming algorithm as shown in [13]. Note that the definition ranges of the variables have to be considered even if they are not defined explicitly in the causes.

Various cost functions (objective functions) are conceivable. For our studies we chose to minimize and maximize each input variable once.

If no boundaries were found (which means that the intersection is empty), go back to step 1 and try a different combination of causes.

5. (Optional) To reduce the number of test data, discard boundary states that include no unique variable values.

6. The expected result for each test data set can easily be determined from the effect variable values.

7. Repeat for other cause combinations until each cause is covered at least once or no further combinations are possible.

**Example:**

To illustrate the generation algorithm, we have included a practical example. Although the effort involved is considerable, the procedure can fortunately be fully automated.

1. We start by choosing the cause SMALL_BIKE for the effect variable Baseprice and the cause OLD_PERSON for the effect variable Extracharge. By combining the causes, we get: \((\text{Type} = 0) \land (\text{HP} < 25) \land (\text{Age} > 65)\)
2. We now have to exclude all invalid causes:

\( (\text{Type} = 0) \land (\text{HP} < 25) \land (\text{Age} > 65) \land (\neg (\text{Age} < 18)) \land (\neg (\text{Type} = 2) \land (\text{Age} < 21)) \land (\neg (\text{Type} = 2) \land (\text{Age} > 65)) \)

3. After performing the transformations described in step 3, we get a complicated Boolean expression containing not less than 8 ‘∨’ and 45 ‘∧’ operators.

4. The ‘∨’ operators divide the expression into 9 parts, each defining an equivalence partition. For each partition, we use a linear programming algorithm to search for boundary states. Each variable is minimized once and maximized once in consideration of its definition range.

In this example, all partition boundaries are orthogonal, so we get a huge number of identical (and rather trivial) solutions. On the whole, the algorithm produces only four different boundary states: \( (\text{HP}, \text{Type}, \text{Age}, \text{Gender}) = \{(0, 0, 66, 0), (24, 0, 999, 1), (24, 0, 999, 1)\} \).

5. (Optional) To further reduce the amount of test data, we discard the boundary states that include no unique variable values. Thus, our set of solutions is reduced to \( (\{0, 0, 66, 0\}, (24, 0, 999, 1)\}) \).

6. For calculating the expected result, we use the effect variable values for the two causes: Baseprice = 50€, Extracharge = 20% => Insurance Premium = 60€

7. The algorithm is repeated for other cause combinations until all causes are covered.

Finally, we get 19 valid and 6 invalid solutions:

| Comb. #1 (valid): SMALL_BIKE ∧ OLD_PERSON |
| Sol. V1: HP = 0, Type = 0, Age = 66, Gender = 0 |
| Sol. V2: HP = 24, Type = 0, Age = 999, Gender = 1 |

| Comb. #2 (valid): MEDIUM_BIKE ∧ YOUNG_MALE |
| Sol. V3: HP = 49, Type = 0, Age = 18, Gender = 0 |
| Sol. V4: HP = 49, Type = 0, Age = 24, Gender = 0 |
| Sol. V5: HP = 25, Type = 0, Age = 21, Gender = 0 |

| Comb. #3 (valid): BIG_BIKE ∧ YOUNG_FEMALE |
| Sol. V6: HP = 50, Type = 0, Age = 18, Gender = 1 |
| Sol. V7: HP = 9999, Type = 0, Age = 20, Gender = 1 |

| Comb. #4 (valid): SMALL_CAR ∧ NORMAL_MALE |
| Sol. V8: HP = 0, Type = 1, Age = 25, Gender = 0 |
| Sol. V9: HP = 59, Type = 1, Age = 65, Gender = 0 |

| Comb. #5 (valid): MEDIUM_CAR ∧ NORMAL_FEMALE |
| Sol. V10: HP = 60, Type = 1, Age = 21, Gender = 1 |
| Sol. V11: HP = 99, Type = 1, Age = 65, Gender = 1 |

| Comb. #6 (valid): BIG_CAR ∧ OLD_PERSON |
| Sol. V12: HP = 100, Type = 1, Age = 66, Gender = 0 |
| Sol. V13: HP = 9999, Type = 1, Age = 999, Gender = 1 |

| Comb. #7 (valid): SMALL_VAN ∧ YOUNG_MALE |
| Sol. V14: HP = 0, Type = 2, Age = 21, Gender = 0 |
| Sol. V15: HP = 59, Type = 2, Age = 24, Gender = 0 |

| Comb. #8 (valid): MEDIUM_VAN ∧ YOUNG_MALE |
| Sol. V16: HP = 60, Type = 2, Age = 21, Gender = 0 |
| Sol. V17: HP = 99, Type = 2, Age = 24, Gender = 0 |

| Comb. #9 (valid): BIG_VAN ∧ YOUNG_MALE |
| Sol. V18: HP = 100, Type = 2, Age = 21, Gender = 0 |
| Sol. V19: HP = 9999, Type = 2, Age = 24, Gender = 0 |

| Comb. #10 (invalid): I_TOO_YOUNG ∧ SMALL_BIKE |
| Sol. I1: HP = 0, Type = 0, Age = 0, Gender = 0 |
| Sol. I2: HP = 24, Type = 0, Age = 17, Gender = 1 |

| Comb. #11 (invalid): I_VAN_TOO_YOUNG |
| Sol. I3: HP = 0, Type = 2, Age = 18, Gender = 0 |
| Sol. I4: HP = 9999, Type = 2, Age = 20, Gender = 1 |

| Comb. #12 (invalid): I_VAN_TOO_OLD |
| Sol. I5: HP = 0, Type = 2, Age = 66, Gender = 0 |
| Sol. I6: HP = 9999, Type = 2, Age = 999, Gender = 1 |

This set of test data covers all valid and invalid causes plus all boundary states with at least one unique variable value.

4. Practical experience

4.1. Implementation of the CECIL method

While the first two tasks of the CECIL method have to be performed by a human tester who enters the necessary information into a suitable computer program, the third task can be fully automated. Thus, the considerable effort required to manually convert complex Boolean expressions and solve linear programs can be avoided. Also, the testers only need basic skills in how to formulate Boolean expressions rather than a deeper mathematical understanding of n-dimensional spaces and linear programs.

We have incorporated the CECIL method into our test generation tool set IDATG (Integrating Design and Automated Test case Generation) [17][18] and will now show its application from the user's perspective.
Firstly, the user specifies the input variables and their definition ranges (the data structure is displayed on the left side of Figure 1). The effect variables are then defined. It is now possible to specify cause/effect pairs in IDATG’s Effect Editor that is shown on the right side of Figure 1. The syntax used for Boolean expressions differs only slightly from that used in this paper, as can be seen in the following example:

\[ \#\text{Type}\# = 1 \text{ AND } \#\text{HP}\# \geq 60 \text{ AND } \#\text{HP}\# < 100 \]

To enhance usability, variables can be selected from a list. It is even possible to formulate expressions that refer to other causes, e.g.

\[ \#\text{Age}\# > 65 \text{ AND } (\%\text{BIG BIKE} \text{ OR } \%\text{BIG CAR}) \]

As soon as all valid and invalid cause/effect pairs have been defined, the generation algorithm can be started. For our example, it automatically produces the results shown in Table 3 in less than 1 second on an average laptop computer.

The resulting test data can be exported in XML format or be used to generate complete test cases with IDATG. Whereas a detailed description would clearly be outside the scope of this paper, we merely want to give the reader a basic idea of the procedure: IDATG can be used to define test sequences in which the generated test data and the corresponding GUI objects are referenced. The steps in a test sequence can represent user actions such as entering a value or clicking a button, but also a comparison between the expected and actual results.

A test case generation algorithm tries to find valid paths through the sequence graphs while considering the specified semantic conditions. Conditions may refer to the contents of a GUI element or to the value of an effect variable of the current data set.

Finally, the generated test cases can be exported in various formats such as complete scripts for HP WinRunner® and Borland SilkTest®, or XML. IDATG is used for testing the Spacecraft Operating System of the European Space Agency ESA, for example.

4.2. Efficiency of the CECIL method

Several case studies have been conducted to evaluate the efficiency of the CECIL method. One of them is the familiar insurance example. For this purpose, we implemented an application that calculates the insurance premium but introduced several types of defects (logical defects, mutated operators and operands, modified error messages etc.).

After applying the CECIL method using IDATG and automatically executing the resulting test cases (see Table 3) with HP WinRunner®, ALL defects were detected (including some that we introduced unintentionally).

By comparison, with ordinary, manually applied cause-effect analysis only 70% of the defects were found while most of the boundary-related defects were overlooked.

The effort required to formalize the textual specification, specify the variables, causes and effects with the IDATG editors and generate the test data was about 2 hours. Another 2 hours were necessary to capture the GUI and define the test sequences and other details required by the IDATG test case generator. The duration of the automated test generation and execution process was negligible (< 5 min.).
The CECIL method has also been applied to test a real application - the working time management system of Siemens Austria. This system allows the employees to enter the time they arrived at work and the time they went home on each day of the week. Based on a set of rather complex rules, the system calculates parameters such as the total working hours and the hours that count as 50% or 100% overtime. In this example, the equivalence partitions can be represented as 2-dimensional areas:

<table>
<thead>
<tr>
<th>Day</th>
<th>00:00 – 07:30</th>
<th>07:31-12:59</th>
<th>13:00-16:14</th>
<th>16:15-18:59</th>
<th>19:00-23:59</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Tue</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Wed</td>
<td>Invalid</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thu</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fri</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sat</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sun</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3 - Equivalence partitions (Example)

After the rules were entered in IDATG in the form of causes and effects, the CECIL algorithm produced a set of test data that covered all causes. Each generated boundary state is marked with an 'x' in Figure 3 - note that each boundary has at least one 'x' on each side. Once again, the effort required to apply the method was minimal (2 hours for specifying the problem and generating the data) while the resulting tests found all known boundary-related defects.

It has also been proved that the method can be applied to problems whose boundaries are not orthogonal. An example is the triangle problem introduced in Section 2.2, even if it has only one effect variable (the type of the triangle). For this example, IDATG generated 23 valid and 6 invalid test data sets.

4.3. Possible variations of the CECIL method

The basic method described in Section 3 can be modified in several ways:

Instead of covering each cause only once, it would be possible to cover each combination of causes provided that:

a) the causes affect different effect variables and

b) their domains have a non-empty intersection.

For instance, it would be possible to combine each of the nine causes that affect the Baseprice with each of the five causes that affect the Extracharge, thus resulting in 45 combinations. Naturally, this would increase the number of test data significantly but would also improve the error detection potential. This "cause combination coverage" is recommended for testing safety-critical systems but may be too costly for other projects.

Similar considerations hold for the optional deletion of boundary states that include no unique variable values (task 3/step 5). In our experience, although this reduction of test data only affects the error detection potential very slightly, it nevertheless seems prudent to omit this step for safety-critical projects.

Further variations could be introduced by modifying the cost function for the linear programming algorithm. Various possibilities have been analysed in [14].

5. Conclusion

Systematic test case design methods play an important role when it comes to improving defect detection and reducing quality costs.

The crucial questions that have to be answered are:

- How can the required coverage be achieved?
- How many test cases are necessary?
- How effective are the test cases?

Standards such as IEC61508 permit the use of equivalence partitions/boundary values to reduce the number of test cases. Special attention must be given to multi-dimensional problems in which the partitioning does not depend on a single input but on a combination of different variables that are interrelated by logical dependencies. As was shown in the triangle example [14], problems with three or more dimensions are difficult to handle without the support of an appropriate tool.

However, the application of the CECIL method is limited when dealing with multi-dimensional problems. The principal contribution of this paper is to provide a basis for generating more efficient test data. The CECIL method was introduced by applying it step-by-step to the example of "Vehicle Insurance". It combines the benefits of equivalence partitioning, boundary value analysis and cause-effect analysis and therefore has an exceptional defect detection potential while limiting the number of test data to a reasonable amount.

The usability problem could be solved by providing a simple-to-use specification technique while automating the mathematically challenging steps of the method. It has been proved that this technique is applicable to a large number of applications such as the working time management system of Siemens Austria.

Applying mutation analysis in the course of various case studies has shown the effectiveness of the CECIL method. Further research is needed to explore the effectiveness of the technique with regard to different defect types.

Whereas the CECIL method can be applied to a huge variety of problems, it cannot be used for
predicates that include non-linear expressions (see Section 3.2). Since it makes little sense to search for boundary values for non-linear problems, the ordinary cause-effect analysis method has to be used in such cases.

While this is certainly a limitation, it only concerns the dependencies between the input variables but not the calculation rules themselves. Also, in most domains non-linear input dependencies are far less common than linear ones. It is still an open question whether the CECIL method can be adapted to non-linear problems and whether it would lead to similar improvements of the defect detection potential.

6. Acknowledgement

The research work reported here was partially conducted within the Softnet Austria competence network (www.soft-net.at) and was funded by the Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirische Wirtschaftsförderungsgesellschaft mbH. (SFG) and the city of Vienna within the scope of the Centre for Innovation and Technology (ZIT).

7. References


