1 Black Box Test Data Generation Techniques

1.1 Equivalence Partitioning

Introduction

Equivalence partitioning is based on the premise that the inputs and outputs of a component can be partitioned into classes that, according to the component's specification, will be treated similarly by the component. Thus the result of testing a single value from an equivalence partition is considered representative of the complete partition.

Example

Consider a component, *generate_grading,* with the following specification:

> *The component is passed an exam mark (out of 75) and a coursework (c/w) mark (out of 25), from which it generates a grade for the course in the range 'A' to 'D'. The grade is calculated from the overall mark which is calculated as the sum of the exam and c/w marks, as follows:*
>
> | | | |
> |---|---|---|
> | *greater than or equal to 70* | - | *'A'* |
> | *greater than or equal to 50, but less than 70* | - | *'B'* |
> | *greater than or equal to 30, but less than 50* | - | *'C'* |
> | *less than 30* | - | *'D'* |
>
> *Where a mark is outside its expected range then a fault message ('FM') is generated. All inputs are passed as integers.*

Initially the equivalence partitions are identified and then test cases derived to exercise the partitions. Equivalence partitions are identified from both the inputs and outputs of the component and both valid and invalid inputs and outputs are considered.

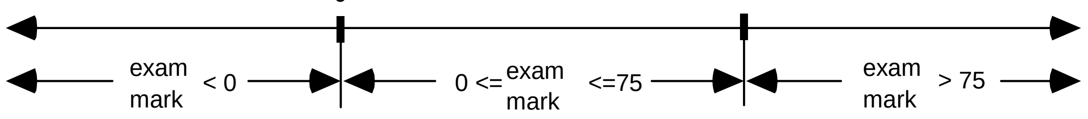
The partitions for the two inputs are initially identified. The *valid* partitions can be described by:

$0 \le$ exam mark ≤ 75
$0 \le$ coursework mark ≤ 25

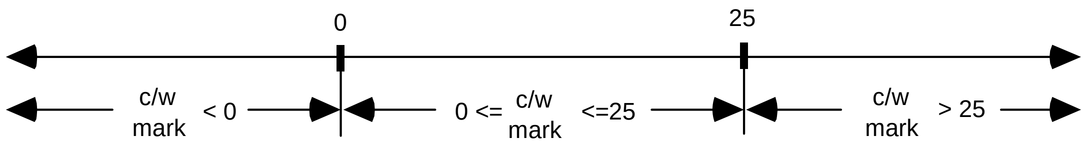
The most obvious *invalid* partitions based on the inputs can be described by:

exam mark > 75
exam mark < 0
coursework mark > 25
coursework mark < 0

Partitioned ranges of values can be represented pictorially, therefore, for the input, exam mark, we get:



And for the input, coursework mark, we get:



Less obvious invalid input equivalence partitions would include any other inputs that can occur not so far included in a partition, for instance, non-integer inputs or perhaps non-numeric inputs. So, we could generate the following invalid input equivalence partitions:

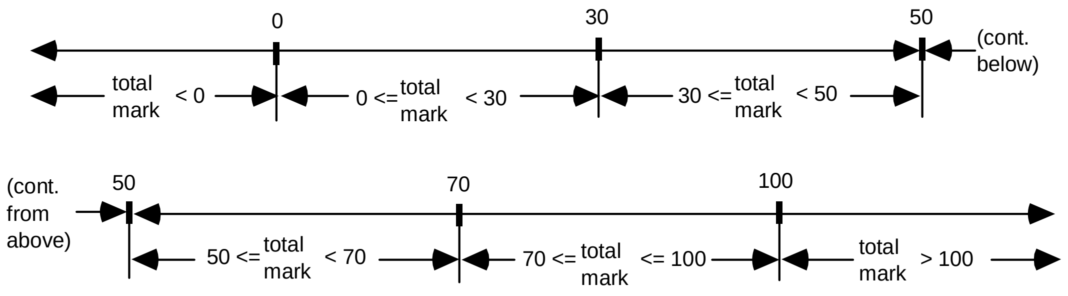
exam mark = real number (a number with a fractional part)
exam mark = alphabetic
coursework mark = real number
coursework mark = alphabetic

Next, the partitions for the outputs are identified. The *valid* partitions are produced by considering each of the valid outputs for the component:

'A'	is induced by	$70 \leq$ total mark ≤ 100
'B'	is induced by	$50 \leq$ total mark < 70
'C'	is induced by	$30 \leq$ total mark < 50
'D'	is induced by	$0 \leq$ total mark < 30
'Fault Message'	is induced by	total mark > 100
'Fault Message'	is induced by	total mark < 0

where total mark = exam mark + coursework mark. Note that 'Fault Message' is considered as a valid output as it is a *specified* output.

The equivalence partitions and boundaries for total mark are shown pictorially below:



An invalid output would be any output from the component other than one of the five specified. It is difficult to identify unspecified outputs, but obviously they must be considered as if we can cause one then we have identified a flaw with either the component, its specification, or both. For this example three unspecified outputs were identified and are shown below. This aspect of equivalence partitioning is very subjective and different testers will inevitably identify different partitions which *they* feel could possibly occur.

output	= 'E'
output	= 'A+'
output	= 'null'

Thus the following nineteen equivalence partitions have been identified for the component (remembering that for some of these partitions a certain degree of subjective choice was required, and so a different tester would not necessarily duplicate this list exactly):

0 ≤ exam mark ≤ 75
exam mark > 75
exam mark < 0
0 ≤ coursework mark ≤ 25
coursework mark > 25
coursework mark < 0
exam mark = real number
exam mark = alphabetic
coursework mark = real number
coursework mark = alphabetic
70 ≤ total mark ≤ 100
50 ≤ total mark < 70
30 ≤ total mark < 50
0 ≤ total mark < 30
total mark > 100
total mark < 0
output = 'E'
output = 'A+'
output = 'null'

Having identified all the partitions then test cases are derived that 'hit' each of them. Two distinct approaches can be taken when generating the test cases. In the first a test case is generated for each identified partition on a one-to-one basis, while in the second a minimal set of test cases is generated that cover all the identified partitions.

The one-to-one approach will be demonstrated first as it can make it easier to see the link between partitions and test cases. For each of these test cases only the single partition being targetted is stated explicitly. Nineteen partitions were identified leading to nineteen test cases.

The test cases corresponding to partitions derived from the input exam mark are:

Test Case	1	2	3
Input (exam mark)	44	-10	93
Input (c/w mark)	15	15	15
total mark (as calculated)	59	5	108
Partition tested (of exam mark)	0 ≤ e ≤ 75	e < 0	e > 75
Exp. Output	'B'	'FM'	'FM'

Note that the input coursework (c/w) mark has been set to an arbitrary valid value of 15.

The test cases corresponding to partitions derived from the input coursework mark are:

Test Case	4	5	6
Input (exam mark)	40	40	40
Input (c/w mark)	8	-15	47
total mark (as calculated)	48	25	87
Partition tested (of c/w mark)	0 ≤ c ≤ 25	c < 0	c > 25
Exp. Output	'C'	'FM'	'FM'

Note that the input exam mark has been set to an arbitrary valid value of 40.

The test cases corresponding to partitions derived from possible invalid inputs are:

Test Case	7	8	9	10
Input (exam mark)	48.7	q	40	40
Input (c/w mark)	15	15	12.76	g
total mark (as calculated)	63.7	not applicable	52.76	not applicable
Partition tested	exam mark = real number	exam mark = alphabetic	c/w mark = real number	c/w mark = alphabetic
Exp. Output	'FM'	'FM'	'FM'	'FM'

The test cases corresponding to partitions derived from the valid outputs are:

Test Case	11	12	13
Input (exam mark)	-10	12	32
Input (c/w mark)	-10	5	13
total mark (as calculated)	-20	17	45
Partition tested (of total mark)	$t < 0$	$0 \leq t < 30$	$30 \leq t < 50$
Exp. Output	'FM'	'D'	'C'

Test Case	14	15	16
Input (exam mark)	44	60	80
Input (c/w mark)	22	20	30
total mark (as calculated)	66	80	110
Partition tested (of total mark)	$50 \leq t < 70$	$70 \leq t \leq 100$	$t > 100$
Exp. Output	'B'	'A'	'FM'

The input values of exam mark and coursework mark have been derived from the total mark, which is their sum.

The test cases corresponding to partitions derived from the invalid outputs are:

Test Case	17	18	19
Input (exam mark)	-10	100	null
Input (c/w mark)	0	10	null
total mark (as calculated)	-10	110	null+null
Partition tested (output)	'E'	'A+'	'null'
Exp. Output	'FM'	'FM'	'FM'

It should be noted that where invalid input values are used (as above, in test cases 2, 3, 5-11, and 16-19) it may, depending on the implementation, be impossible to actually execute the test case. For instance, in Ada, if the input variable is declared as a positive integer then it will not be possible to assign a negative value to it. Despite this, it is still worthwhile *considering* all the test cases for completeness.

It can be seen above that several of the test cases are similar, such as test cases 1 and 14, where the main difference between them is the partition targetted. As the component has two inputs and one output, each test case actually 'hits' three partitions; two input partitions and one output partition.. Thus it is possible to generate a smaller 'minimal' test set that still 'hits' all the identified partitions by deriving test cases that are designed to exercise more than one partition. The following test case suite of eleven test cases corresponds to the minimised test case suite approach where each test case is designed to hit as many new partitions as possible rather than just one. Note that here all three partitions are explicitly identified for each test case.

Test Case	1	2	3	4
Input (exam mark)	60	40	25	15
Input (c/w mark)	20	15	10	8
total mark (as calculated)	80	55	35	23
Partition (of exam mark)	$0 \leq e \leq 75$	$0 \leq e \leq 75$	$0 \leq e \leq 75$	$0 \leq e \leq 75$
Partition (of c/w mark)	$0 \leq c \leq 25$	$0 \leq c \leq 25$	$0 \leq c \leq 25$	$0 \leq c \leq 25$
Partition (of total mark)	$70 \leq t \leq 100$	$50 \leq t < 70$	$30 \leq t < 50$	$0 \leq t < 30$
Exp. Output	'A'	'B'	'C'	'D'

Test Case	5	6	7	8
Input (exam mark)	-10	93	60.5	q
Input (c/w mark)	-15	35	20.23	g
total mark (as calculated)	-25	128	80.73	-
Partition (of exam mark)	$e < 0$	$e > 75$	e = real number	e = alphabetic
Partition (of c/w mark)	$c < 0$	$c > 25$	c = real number	c = alphabetic
Partition (of total mark)	$t < 0$	$t > 100$	$70 \leq t \leq 100$	-
Exp. Output	'FM'	'FM'	'FM'	'FM'

Test Case	9	10	11
Input (exam mark)	-10	100	'null'
Input (c/w mark)	0	10	'null'
total mark (as calculated)	-10	110	null+null
Partition (of exam mark)	$e < 0$	$e > 75$	-
Partition (of c/w mark)	$0 \leq c \leq 25$	$0 \leq c \leq 25$	-
Partition (of total mark)	$t < 0$	$t > 100$	-
Partition (of output)	'E'	'A+'	'null'
Exp. Output	'FM'	'FM'	'FM'

The one-to-one and minimised approaches represent the two approaches to equivalence partitioning. The disadvantage of the one-to-one approach is that it requires more test cases and if this causes problems a more minimalist approach can be used. Normally, however, the identification of partitions is far more time consuming than the generation and execution of test cases themselves and so any savings made by reducing the size of the test case suite are relatively small compared with the overall cost of applying the technique. The disadvantage of the minimalist approach is that in the event of a test failure it can be difficult to identify the cause due to several new partitions being exercised at once. This is a debugging problem rather than a testing problem, but there is no reason to make debugging more difficult than it is already.

Some testers would say that a variable's input domain equivalence partitions must be combined with every other variables' input domains equivalence partitions. This is an extreme view as it leads to an explosion of number of tests. It hard however to argue against it on the ground of completeness.

Too many tests are a problem since deriving the expected output can be time consuming and error prone (on the other hand actual test execution and checking against the expected results can be done using tools and is therefore not a problem in itself).

1.2 Boundary Value Analysis

Introduction

Boundary Value Analysis is based on the following premise. Firstly, that the inputs and outputs of a component can be partitioned into classes that, according to the component's specification, will be treated similarly by the component (in the same way as in equivalence partitioning) and, secondly, that developers are prone to making errors in their treatment of the boundaries of these classes. Thus test cases are generated to exercise these boundaries.

Example

Consider a component, *generate_grading,* with the following specification:

> *The component is passed an exam mark (out of 75) and a coursework (c/w) mark (out of 25), from which it generates a grade for the course in the range 'A' to 'D'. The grade is calculated from the overall mark which is calculated as the sum of the exam and c/w marks, as follows:*

greater than or equal to 70	-	*'A'*
greater than or equal to 50, but less than 70	-	*'B'*
greater than or equal to 30, but less than 50	-	*'C'*
less than 30	-	*'D'*

> *Where a mark is outside its expected range then a fault message ('FM') is generated. All inputs are passed as integers.*

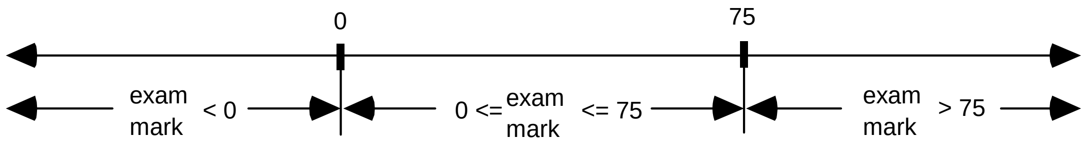
Initially the equivalence partitions are identified, then the boundaries of these partitions are identified, and then test cases are derived to exercise the boundaries. Equivalence partitions are identified from both the inputs and outputs of the component and both valid and invalid inputs and outputs are considered.

$0 \leq$ exam mark ≤ 75
$0 \leq$ coursework mark ≤ 25

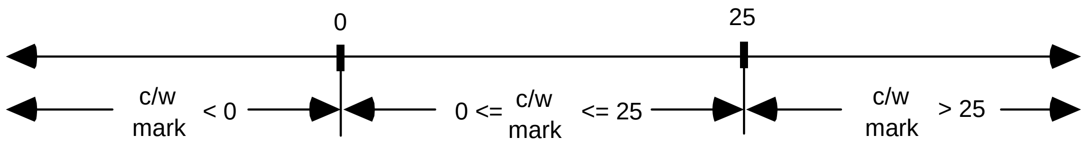
The most obvious *invalid* partitions can be described by:

exam mark > 75
exam mark < 0
coursework mark > 25
coursework mark < 0

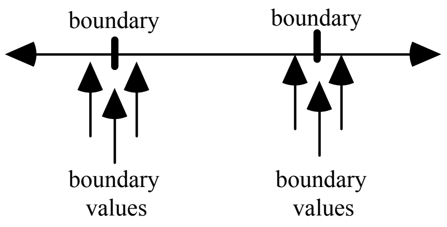
Partitioned ranges of values can be represented pictorially, therefore, for the input, exam mark, the same notation leads to:



And for the input, coursework mark, we get:



For each boundary three values are used, one on the boundary itself and one either side of it, the smallest significant distance away, as shown below:



Thus the six test cases derived from the input exam mark are:

Test Case	1	2	3	4	5	6
Input (exam mark)	-1	0	1	74	75	76
Input (c/w mark)	15	15	15	15	15	15
Boundary tested (exam mark)		0			75	
Exp. Output	'FM'	'D'	'D'	'A'	'A'	'FM'

Note that the input coursework (c/w) mark has been set to an arbitrary valid value of 15.

The test cases derived from the input coursework mark are thus:

Test Case	7	8	9	10	11	12
Input (exam mark)	40	40	40	40	40	40
Input (c/w mark)	-1	0	1	24	25	26
Boundary tested (c/w mark)		0			25	
Exp. Output	'FM'	'C'	'C'	'B'	'B'	'FM'

Note that the input exam mark has been set to an arbitrary valid value of 40.

Less obvious invalid input equivalence partitions would include any other inputs that can occur not so far included in a partition, for instance, non-integer inputs or perhaps non-numeric inputs. In order to be considered an equivalence partition those values within it must be expected, from the specification, to be treated in an equivalent manner by the component. Thus we could generate the following invalid input equivalence partitions:

exam mark = real number
exam mark = alphabetic
coursework mark = real number
coursework mark = alphabetic
etc.

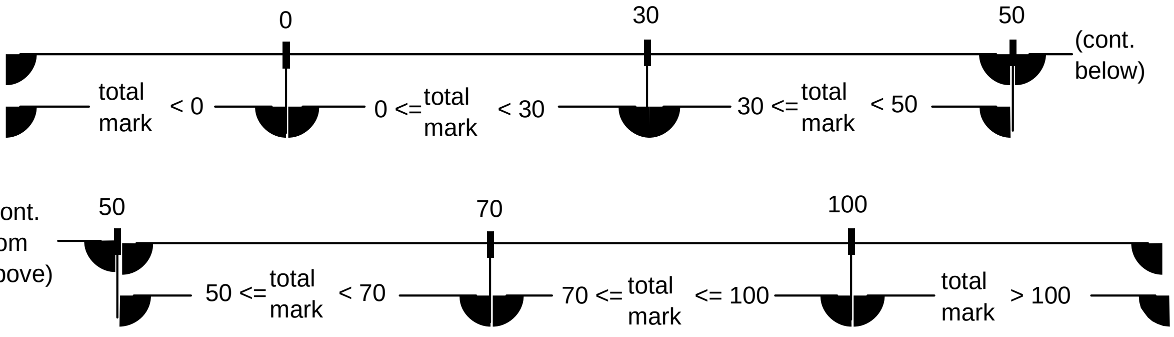
Although these are valid equivalence partitions they have no identifiable boundaries and so no test cases are derived.

Next, the partitions and boundaries for the outputs are identified. The *valid* partitions are produced by considering each of the valid outputs for the component thus:

'A'	is induced by	$70 \leq$ total mark ≤ 100
'B'	is induced by	$50 \leq$ total mark < 70
'C'	is induced by	$30 \leq$ total mark < 50
'D'	is induced by	$0 \leq$ total mark < 30
'Fault Message'	is induced by	total mark > 100
'Fault Message'	is induced by	total mark < 0

where total mark = exam mark + coursework mark.

'Fault Message' is considered here as it is a specified output. The equivalence partitions and boundaries for total mark are shown below:



Thus the test cases derived from the valid outputs are:

Test Case	13	14	15	16	17	18	19	20	21
Input (exam mark)	-1	0	0	29	15	6	24	50	26
Input (c/w mark)	0	0	1	0	15	25	25	0	25
total mark (as calculated)	-1	0	1	29	30	31	49	50	51
Boundary tested (total mark)		0			30			50	
Exp. Output	'FM'	'D'	'D'	'D'	'C'	'C'	'C'	'B'	'B'

Test Case	22	23	24	25	26	27
Input (exam mark)	49	45	71	74	75	75
Input (c/w mark)	20	25	0	25	25	26
total mark (as calculated)	69	70	71	99	100	101
Boundary tested (total mark)		70			100	
Exp. Output	'B'	'A'	'A'	'A'	'A'	'FM'

The input values of exam mark and coursework mark have been derived from the total mark, which is their sum.

An invalid output would be any output from the component other than one of the five specified. It is difficult to identify unspecified outputs, but obviously they must be considered as if we can cause one then we have identified a flaw with either the component, its specification, or both. For this example three unspecified outputs were identified ('E', 'A+', and 'null'), but it is not possible to group these possible outputs into ordered partitions from which boundaries can be identified and so no test cases are derived.

So far several partitions have been identified that appear to be bounded on one side only. These are:

exam mark > 75
exam mark < 0
coursework mark > 25
coursework mark < 0
total mark > 100
total mark < 0

In fact these partitions are bounded on their other side by implementation-dependent maximum and minimum values. For integers held in sixteen bits these would be 32767 and -32768 respectively.

Thus, the above partitions can be more fully described by:

75 < exam mark ≤ 32767
-32768 ≤ exam mark < 0
25 < coursework mark ≤ 32767
-32768 ≤ coursework mark < 0
100 < total mark ≤ 32767
-32768 ≤ total mark < 0

It can be seen that by bounding these partitions on both sides a number of additional boundaries are identified, which must be tested. This leads to the following additional test cases:

Test Case	28	29	30	31	32	33
Input (exam mark)	32766	32767	32768	-32769	-32768	-32767
Input (c/w mark)	15	15	15	15	15	15
Boundary tested (exam mark)		32767			-32768	
Exp. Output	'FM'	'FM'	'FM'	'FM'	'FM'	'FM'

Test Case	34	35	36	37	38	39
Input (exam mark)	40	40	40	40	40	40
Input (c/w mark)	32766	32767	32768	-32769	-32768	-32767
Boundary tested (c/w mark)		32767			-32768	
Exp. Output	'FM'	'FM'	'FM'	'FM'	'FM'	'FM'

Test Case	40	41	42	43	44	45
Input (exam mark)	16383	32767	1	0	-16384	-32768
Input (c/w mark)	16383	0	32767	-32767	-16384	-1
total mark (as calculated)	32766	32767	32768	-32767	-32768	-32769
Boundary tested (total mark)		32767			-32768	
Exp. Output	'FM'	'FM'	'FM'	'FM'	'FM'	'FM'

It should be noted that where invalid input values are used (as above, in test cases 1, 6, 7, 12, 13, and 27-45) it may, depending on the implementation, be impossible to actually execute the test case. For instance, in Ada, if the input variable is declared as a positive integer then it will not be possible to assign a negative value to it. Despite this, it is still worthwhile *considering* all the test cases for completeness.

2 White Box Test Data Generation Techniques

2.1 Statement Testing and Coverage

Introduction

This structural test technique is based upon the decomposition of the component into constituent statements.

Example

The two principal questions to answer are:

- what is a statement?
- which statements are executable?

In general a statement should be an atomic action, that is a statement should be executed completely or not at all. For instance:

```
IF a THEN b ENDIF
```

is considered as more than one statement because `b` may or may not be executed depending upon the condition `a`. The definition of *statement* used for statement testing need not be the one used in the language definition.

We would expect statements which are associated with machine code to be regarded as executable. For instance, we would expect all of the following to be regarded as executable:

- assignments;
- loops and selections;
- procedure and function calls;
- variable declarations with explicit initialisations;
- dynamic allocation of variable storage on a heap.

However, most other variable declarations can be regarded as non executable.

Consider, the following C code:

```
a;
if (b) {
       c;
       }
d;
```

Any test case with b TRUE will achieve full statement coverage. Note that full statement coverage can be achieved without exercising with b FALSE.

2.2 Branch (AKA Decision) Testing and Coverage

Consider the example:

> *The component shall determine the position of a word in a table of words ordered alphabetically. Apart from the word and table, the component shall also be passed the number of words in the table to be searched. The component shall return the position of the word in the table (starting at zero) if it is found, otherwise it shall return "-1".*

1.3 Decision Table Testing

Exercise

Use the decision table below to answer the following questions:

Delivery Charges	Rules					
	1	**2**	**3**	**4**	**5**	**6**
Purchase Amount	>=$500	<500 & >100	<500 & >100	<500 & >100	<=100	<=100
Preferred Customer?	--	Y	N	N	--	--
Paid Cash?	--	--	Y	N	Y	N
Free Delivery	X	X				
Charge $5			X		X	
Charge $10				X		X

1. How much is delivery if a customer paid by credit card for a bill totaling $1005.49?
2. How much is delivery if a preferred customer paid cash for a $99.95 bill?
3. How much is delivery if a customer paid cash for a bill totaling $1250.00, if they are not a preferred customer?
4. How much is delivery if the customer's bill comes out to $455.00, they paid by credit card, and they're a preferred customer?

Reading Decision Tables

1. Free delivery
2. delivery is $5
3. Free delivery
4. Free delivery

2. Medical Insurance Policy:

Patients with medical insurance may or may not receive reimbursements for medical visits, depending on a couple of factors... First of all, no charges are reimbursed to the patient until the deductible has been met. If the deductible has been met, then reimbursement depends on the type of visit: Doctor's office visits are reimbursed at 50%, Hospital visits are reimbursed at 80% and Lab visits are reimbursed at 70%.

Pick out actions and conditions in policy description:

1. Introduces the conditions (a couple of factors) and actions (patients receiving some kind of reimbursement)
2. Condition: Until the deductible has been met. Action: no reimbursement to the patient.
3. Condition: If the deductible has been met...
4. Conditions: type of visit is doctor's office; type of visit is hospital, type of visit is lab; Actions: Reimburse 80%, reimburse 70%, reimburse 50%

Conditions:
Deductible has been met (Values = Y, N)
Type of visit(Values = (D)octor, (H)ospital, (L)ab)

Actions:
No reimbursement
Reimburse 80%
Reimburse 70%
Reimburse 50%

Number of rules: 2 values * 3 values = 6 rules.

Medical Insurance Policy	**Rules**					
	1	**2**	**3**	**4**	**5**	**6**
Deductible has been met	Y	Y	Y	N	N	N
Type of visit	D	H	L	D	H	L
No reimbursement				X	X	X
Reimburse 80%		X				
Reimburse 70%			X			
Reimburse 50%	X					

D = Doctor's visit, H = Hospital visit, L = Lab visit

We can reduce this table by combining rules 1 to 3 since they all have the same action and the "Deductible has been met" is Yes for all, and the "Type of Visit" covers all possible options (D, H, L):

Medical Insurance Policy	**Rules**			
	1	**2**	**3**	**4**
Deductible has been met	Y	Y	Y	N
Type of visit	D	H	L	--
No reimbursement				X
Reimburse 80%		X		
Reimburse 70%			X	
Reimburse 50%	X			

D = Doctor's visit, H = Hospital visit, L = Lab visit

Extended Entry Version:

Medical Insurance Policy	**Rules**			
	1	**2**	**3**	**4**
Deductible Status	Met	Met	Met	Not Met
Type of visit	Doctor	Hospital	Lab	--
Amt. of Reimbursement	50%	80%	70%	0%