SOFTWARE TESTING

Test Generation from Requirements – II

- Cause –Effect Graphs
- Predicates

Limitations of the other Methods

Boundary Value Analysis and Equivalent Partitioning based methods

- Result in too many test cases since they allow the

Selection of a large number of input combinations

- Many of these test cases are infeasible

Cause-Effect Graphing is basically a hardware testing technique adapted to software testing. The CEG technique is a black-box method, i.e, it considers only the desired external behavior of a system.

In CEG analysis, first, identify ***causes***1, ***effects***2 and ***constraints***3 in the (natural language) specification. Second, construct a CEG as a combinational logic network which consists of nodes, called *causes* and *effects, arcs* with Boolean operators (*and, or, not*) between causes and effects, and *constraints*. Finally, trace this graph to build a decision table which may be subsequently converted into use cases and, eventually, test cases.

Cause-effect Graphing also known as dependency modeling.

- Visual representation of a logical relationship among inputs and outputs that can be expressed as a Boolean expression

Focuses on modeling dependency relationships among

• Program input conditions (causes), and

• Output conditions (effects)

Allows selecting only few relevant test cases and thereby helping us to overcome the problem of too many test cases. Cause-Effect Graphing (CEG) is a model used to help identify productive test cases by using a simplified digital-logic circuit (combinatorial

logic network) graph. It's origin is in hardware engineering but it has been adapted for use in software engineering. The CEG technique is a Black-Box method, (i.e. it considers the external behavior of a system with respect to how that system has been specified). It takes into consideration the combinations of causes that result in effecting the system's behavior.

Cause-Effect Graphing (CEG) is used to derive test cases from a given natural language specification to validate its corresponding implementation. The CEG technique is a lack-box method, i.e, it considers only the desired external behavior of a system Cause-Effect Graphing technique derives the minimum number of test cases to cover 100% of the functional requirements to improve the quality of test coverage.

A cause-effect graph is "a graphical representation of inputs or stimuli (causes) with their associated outputs (effects), which can be used to design test cases".

Furthermore, cause-effect graphs contain directed arcs that represent logical relationships between causes and effects.

Each arc can be influenced by Boolean operators.

Such graphs can be used to design test cases, which can directly be derived from the graph , or to visualize and measure the completeness and the clearness of a test model for the tester.

Cause-Effect Graphing is very similar to Decision Table-Based Testing, where logical relationships of the inputs produce outputs; this is shown in the form of a graph.

The graph used is similar to that of a Finite State Machine (FSM). Symbols are used to show the relationships between input conditions, those symbols are similar to the symbols used in propositional logic.

Cause-effect graphs are directed graphs with causes and effects represented as nodes and connections between them represented as arcs.

Each node is labeled with a unique number or letter referencing a certain condition.

Arcs can be negated and connected to other arcs with Boolean operators.

Cause-effect graphs are often used because they are easy to understand and intuitive to use.

The CEG technique is a Black-Box method, (i.e. it considers the external behavior of a system with respect to how that system has been specified).

It takes into consideration the combinations of causes that result in effecting the system's behavior.

Although there are some issues with CEG such as the difficulty in discerning causes and effects from some natural language specifications, CEG remains a good way to analyze specification completeness and represent logic relationships from which productive test cases can be identified.

The starting point for the Cause-Effect Graph is the requirements document.

The requirements describe "what" the system is intended to do.

The requirements can describe real time systems, events, data driven systems, state transition diagrams, object oriented systems, graphical user interface standards, etc.
Any type of logic can be modeled using a Cause-Effect diagram.

Each cause (or input) in the requirements is expressed in the cause-effect graph as a condition, which is either true or false.

Each effect (or output) is expressed as a condition, which is either true or false.
In CEG analysis, first, identify ***causes***, ***Effects*** and ***constraints*** in the (natural language) specification.

Second, construct a CEG as a combinational logic network which consists of nodes, called *causes* and *effects, arcs* with Boolean operators (*and, or, not*) between causes and effects, and *constraints*.

Finally, trace this graph to build a decision table which may be subsequently converted into use cases and eventually, test cases.

A cause represents a distinct input condition or an equivalence class of input conditions.

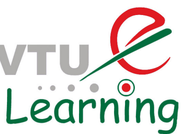
A cause can be interpreted as an entity which brings about an internal change in the system. In a CEG, a cause is always positive and atomic.

An effect represents an output condition or a system transformation which is observable.

An effect can be a state or a message resulting from a combination of causes.

Constraints represent external constraints on the system.
Procedure for Test Generation using Cause-Effect
Graphing

- Identify causes and effects by reading the requirements.
- Assign unique identifier to each cause and effect



- Express the relationship between causes and effects using cause effect graph
- Transform the cause effect graph into a limited entry decision table
- Generate tests from the decision table

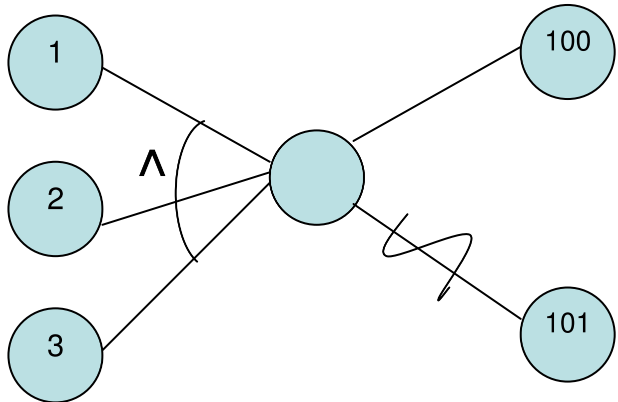
Example - Sendfile Command

In a given network, the sendfile command is used to send a file to a user on a different file server. The sendfile command takes three arguments: the first argument should be an existing file in the sender's home directory, the second argument should be the name of thereceiver's file server, and the third argument should be the receiver's userid. If all the arguments are correct, then the file is successfully sent; otherwise the sender obtains an error message.

The above informal specification is first traced to derive causes and effects. Each one of these causes and effects are given a unique identification number.

Causes	**Effects**
1. The first argument is the name of an existing file in the sender's home directory. 2. The second argument is the name of receiver's file server. 3. The third argument is the receiver's userid.	100. The file is successfully sent. 101. The sender obtains an error message.

Cause-Effect Graph:



Decision Table:

	1	2	3	4	5	6	7	8
Causes 1 2 3	 1 1 1	 0 0 0	 1 0 0	 0 1 0	 0 0 1	 1 1 0	 1 0 1	 0 1 1
Effects 100 101	 1 0	 0 1	 0 1	 0 1	 0 1	 0 1	 0 1	 0 1

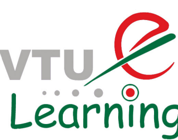
Cause-Effect Graphing Example: 2

I have a requirement that says: "If A OR B, then C."

The following rules hold for this requirement:

- If A is true and B is true, then C is true.
- If A is true and B is false, then C is true.
- If A is false and B is true, then C is true.
- If A is false and B is false, then C is false.

The cause-effect graph that represents this requirement is provided in Figure.



The cause-effect graph shows the relationship between the causes and effects.

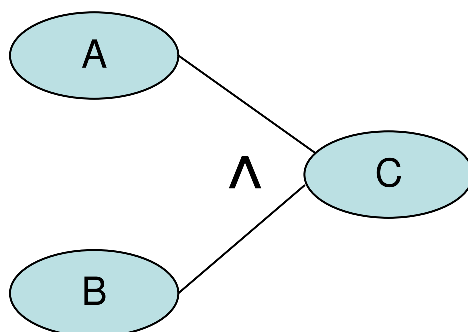


Figure – A Cause-Effect Graph

In Figure, A, B and C are called nodes. Nodes A and B are the causes, while Node C is an effect. Each node can have a true or false condition.

The lines, called vectors, connect the cause nodes A and B to the effect node C.

All requirements are translated into nodes and relationships on the cause-effect graph.

There are only four possible relationships among nodes, and they are indicated by the following symbols:

Where A always leads to C, a straight line ------------.
Where A or B lead to C, a V at the intersection means "or".
Where A and B lead to C, an inverted V at the intersection means "and".
A tilde ~ means "not" as in "If not A, then C".

The Decision Table

The cause-effect graph is then converted into a decision table or "truth table" representing the logical relationships between the causes and effects.

Each column of the decision table is a test case. Each test case corresponds to a unique possible combination of inputs that are either in a true state, a false state, or a masked state (the masked state will be described in Figure below).

Since there are 2 inputs to this example, there are 2 ** 2 = 4 combinations of inputs from which test cases can be selected.

Explores combinations of input conditions

Consists of 2 parts: Condition section and Action section

- Condition Section - Lists conditions and their combinations

- Action Section - Lists responses to be produced

Exposes errors in specification

Columns in decision table are converted to test cases

Similar to Condition Coverage used in White Box Testing

Decision Table from Cause-Effect Graph

	Test1	Test2	Test3
Causes			
A	T	F	F
B	F	T	F
Effect			
C	T	T	F

A Simple Insurance Model:

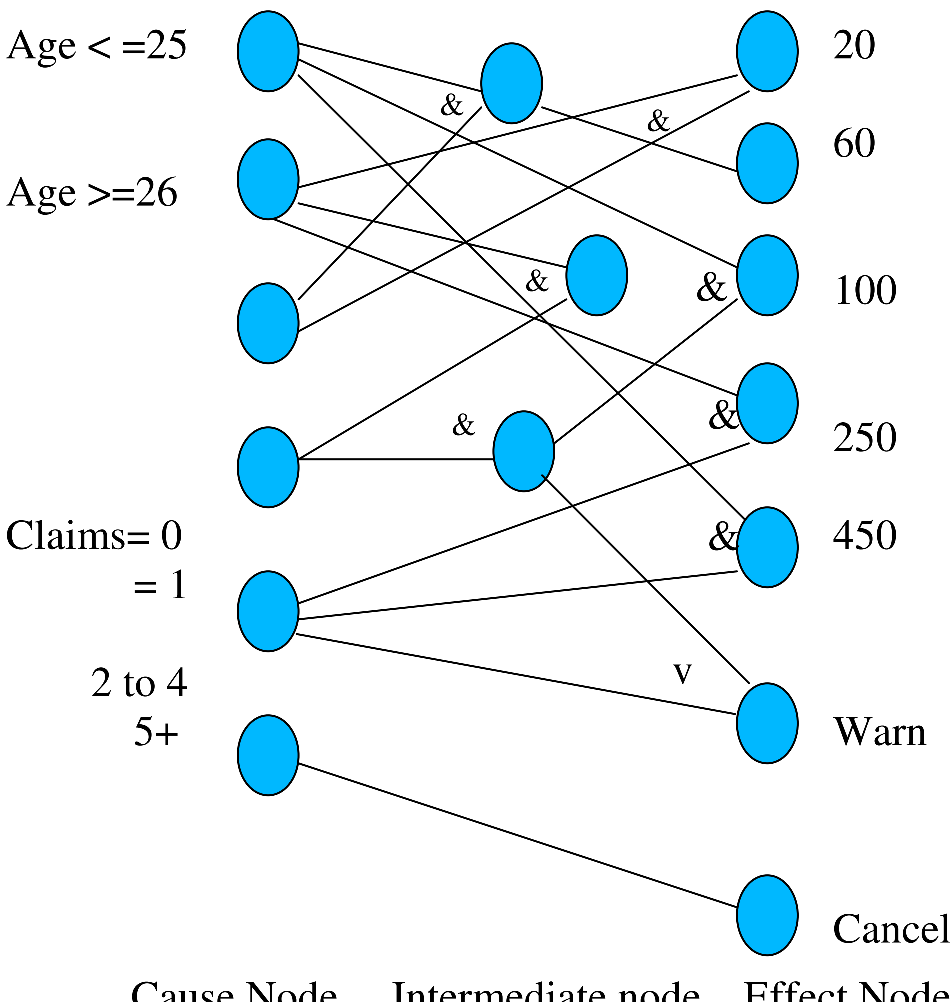
Cause Variables :

1. Age of Client
2. No.of claims in last year

Effects:

1. Premium rise
2. Issue warning letter
3. Cancel policy

CEG (Cause-Effect Graph) for Car insurance Model:



CEG:

It has the following characteristics:

- Graphical representation
- Same Information as in Decision Table
- Can be mapped to a Decision Table
- Uses decision variables , conditions , and resulting actions
- Visual clues for missing or incorrect relationships
- Constraints on causes are supported
  Exclusive Causes (mutually exclusive)
  Inclusive Causes (at least one true)
  Singleton Causes (exactly one true)
- Large Models -> Visually Intractable

- One node for each decision variable and
- One node for each effect (output action)
- Line from cause node to effect node: the cause is a necessary condition for the effect
- An effect needs more than one cause : show relationship between causes using
  V : *logical or* ; & : *logical and* ;
  ~ : *logical not*
- Intermediate nodes may be used to simplify the graph.
- Directly Boolean Expressions can be derived or Converted to Decision Tables.

<u>Constraints on Effects</u>

Consider the following two effects in the inventory example:

Ef1: Generate “Shipping invoice”

Ef2: Generate an “Order not shipped” regret letter

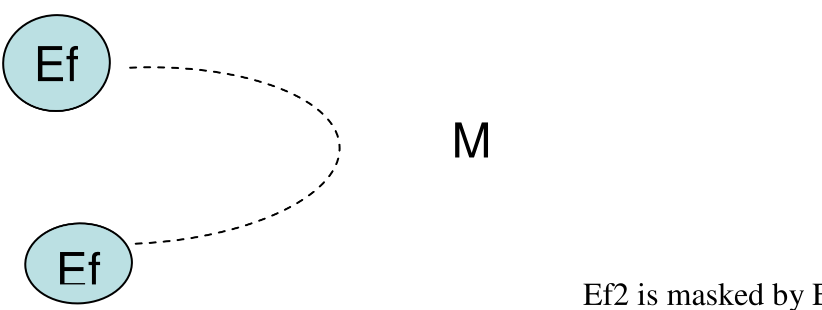
Effect Ef1 occurs when the order can be met from the inventory.

Effect Ef2 occurs when the order placed cannot be met from the Inventory or when the order item has been discontinued after the order was placed.

However, Ef2 is masked by Ef1 for the same order, that is both effects cannot occur for the same order.

A condition that is false is said to be in 0-state, and a condition that is true is said to be in 1-state.

An effect present is 1-state, and absent is in 0-state.



Ef2 is masked by Ef1

Constraints	Possible Values		
	C1	C2	C3
E (C1, C2, C3)	0 1 0 0	0 0 1 0	0 0 0 1
I (C1, C2)	1 0 1	0 1 1	- - -
R (C1, C2)	1 1 0 0	1 1 0 1	- - - -
O (C1, C2, C3)	1 0 0	0 1 0	0 0 1

Another Example for CEG

❑ Verbal specification

The character in column 1 must be an "A" or a "B".
The character in column 2 must be a digit.
In this situation, the file update is made.
If the first character is incorrect, message X12 is issued.
If the second character is not a digit, message X13 is issued.

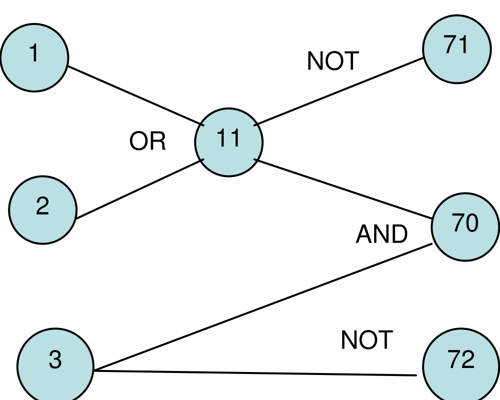
❑ Causes

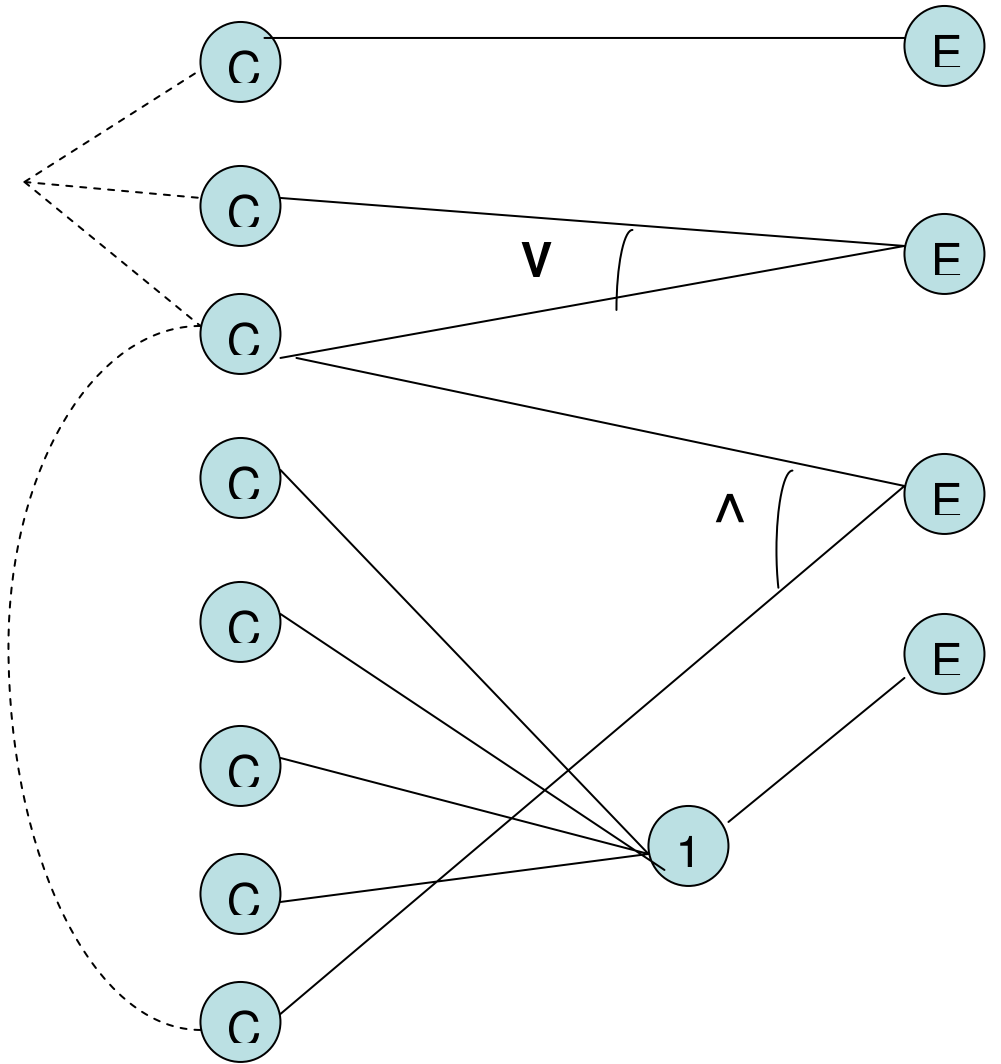
1 character in column 1 is "A"

2 character in column 1 is "B"
3 character in column 2 is a digit

❑ Effects

70 file update, update message -> effect1
71 message X12 is issued-> effect2
72 message X13 is issued-> effect3





Creating Cause-Effect Graph (Step-by Step Procedure)

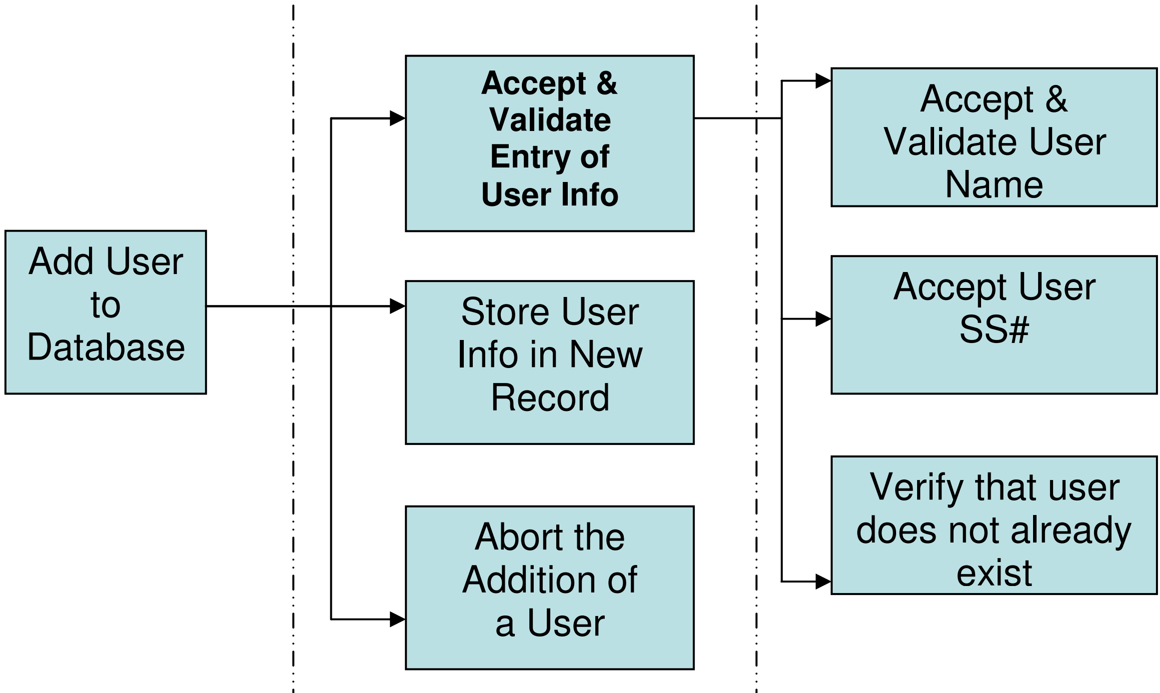
Step one: Break the specification down into workable pieces.

First, the functional requirements are decomposed and analyzed.

To do this, the functional requirements are partitioned into logical groupings, for example, commands, actions, and menu options.

Each logical grouping is then further analyzed and decomposed into a list of detailed functions, sub-functions, and so forth.

Decomposed Function List



Step two: Identify the causes and effects.

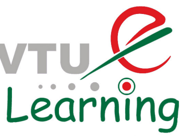
a) Identify the causes (the distinct or equivalence classes of input conditions) and assign each one a unique number.

A cause can also be referred to as an input, as a distinct input condition, or as an equivalence class of input conditions. In equivalence class partitioning, each input or output is divided into subset domains that represent both valid and invalid values.

For example, if an input has a specified range of 1 to 100 there are three equivalency classes, one valid class containing all values from 1 to 100 inclusive, and two invalid classes, values less than one and values greater than 100. Examining the specification, or other similar artifact, word-by-word and underlining words or phrases that describe inputs helps to identify the causes. An input (cause) is an event that is generated outside an application that the application must react to in some fashion. Examples of inputs include hardware events (e.g. keystrokes, pushed buttons, mouse clicks, and sensor activations), API calls, return codes, and so forth.

b) Identify the effects or system transformation and assign each one a unique number.

An effect can also be referred to as an output action, as a distinct output condition, as an



equivalence class of output conditions or as an output such as a confirmation message or error message. An output (effect) is an event that an application generates that is sent outside the application. Examples of output include a message printed on the screen, a string sent to a database, a command sent to the hardware, a request to the operating system, and so forth. System transformations such as file or database record updates are considered effects as well. As with causes, examining the specification, or other similar artifact, word-by-word and underlining words or phrases that describe outputs or system transformations helps to identify the effects.

Consider the following set of requirements as an example:

Requirements for Calculating Car Insurance Premiums:
R00101 For females less than 65 years of age, the premium is $500
R00102 For males less than 25 years of age, the premium is $3000
R00103 For males between 25 and 64 years of age, the premium is $1000
R00104 For anyone 65 years of age or more, the premium is $1500

When examining these requirements, we see that there are two variables that will determine what the premium will be: sex and age. Therefore there are 5 distinct *causes* or input conditions: Sex is either male or female, and age is either less than 25, between 25 and 64 inclusive, or 65 or more. There are also 4 distinct *effects* or ouput conditions: the premium is either $500, $1000, $1500, or $3000. As shown in Table 1 below, each cause and each effect is assigned an arbitrary unique number as part of this process step.

Causes (input conditions) Effects (output conditions)
1. Sex is Male
2. Sex is Female
3. Age is <25
4. Age is >=25 and < 65
5. Age is >= 65
100. Premium is $1000
101. Premium is $3000
102. Premium is $1500
103. Premium is $500

Step three: The semantic content of the specification is analyzed and transformed into a Boolean graph linking the causes and effects. This is the cause-effect graph.

Semantics, in this step's instructions, reflect the meaning of the programs or functions. This meaning is discerned from the specification and transformed into a boolean graph that maps the causes to the resulting effects. It is easier to derive the boolean function for each effect from their separate CEGs, then these can be overlaid to produce a single decision table for all effects (see step 5). For example, the following separate CEGs (see Table 2) can be derived from the Calculating Car Insurance Premiums example above.

CEG	Interpretation
CEG #1:	Causes: 1. Sex is Male and (^) 4. Age is >=25 and < 65 Effect: 100: Premium is $1000
CEG #2:	Causes: 1. Sex is Male and (^) 3. Age is <25 Effect: 101: Premium is $3000
CEG #3:	Causes: 1. Sex is Male and (^) 5. Age is >= 65 or (v) 2. Sex is Female and (^) 5. Age is >= 65 Effect: 102: Premium is $1500
CEG #4:	Causes: 2. Sex is Female and (^) 3. Age is <25 or (v) 2. Sex is Female and (^) 4. Age is >=25 and < 65 Effect: 103: Premium is $500

Table 2 – Cause-Effect Graphs

Step four: Annotate the graph with constraints describing combinations of causes and/or effects that are impossible because of syntactic or environmental constraints or considerations.

In most software programs, certain combinations of causes are impossible because of syntactic or environmental considerations. For example, for the purpose of calculating insurance premium in the above example, a person cannot be both a “Male” and a “Female” simultaneously. To show this, the CEG is annotated, as appropriate; with the following constraint symbols (see Table 3):

Constraint Symbol	Definition
E (a, b)	The "E" (Exclusive) constraint states that both causes *a* and *b* cannot be true simultaneously.
I (a, b, c)	The "I" (Inclusive (at least one)) constraint states that at least one of the causes *a*, *b* and *c* must always be true (*a*, *b*, and *c* cannot be false simultaneously).
O (a, b)	The "O" (One and Only One) constraint states that one and only one of the causes *a* and *b* can be true.
R (a, b)	The "R" (Requires) constraint states that for cause *a* to be true, than cause *b* must be true. In other words, it is impossible for cause *a* to be true and cause *b* to be false.
M (x, y)	The "M" (mask) constraint states that if effect *x* is true; effect *y* is forced to false. (Note that the mask constraint relates to the effects and not the causes like the other constraints.

Table 3 – Constraint Symbols

Step five: Methodically trace state conditions in the graphs, converting them into a limited-entry decision table. Each column in the table represents a test case.

The ones (1) in the limited entry decision table column indicate that the cause (or effect) is true in the CEG and zeros (0) indicate that it is false. Table 4 below illustrates the limited-entry decision table created by converting the CEG from the Calculating Car Insurance Premiums example. For example, the CEG #1, from Table 2 in step 3, converts into test case column 1 in the table below. From CEG #1, causes 1 and 3 being true result in effect 101 being true.

Test Case	1	2	3	4	5	6
Causes:						
1 (male)	1	1	1	0	0	0
2 (female)	0	0	0	1	1	1
3 (<25)	1	0	0	0	1	0
4 (>=25 and < 65)	0	1	0	0	0	1
5 (>= 65)	0	0	1	1	0	0
Effects:						
100 (Premium is $1000)	0	1	0	0	0	0
101 (Premium is $3000)	1	0	0	0	0	0
102 (Premium is $1500)	0	0	1	1	0	0
103 (Premium is $500)	0	0	0	0	1	1

Table 4 – Limited-Entry Decision Table

Some CEGs may result in more than one test case being created. For example, because of the one and only one constraint in the annotated CGE #3 from step 4, this CEG results in test cases 3 and 4 in the decision table above.

Step six: The columns in the decision table are converted into test cases.
Converting the decision table above would result in the following test cases (see Table 5):

Test Case #	Inputs (Causes)		Expected Output (Effects)
	Sex	Age	Premium
1	Male	<25	$3000
2	Male	>=25 and < 65	$1000
3	Male	>= 65	$1500
4	Female	>= 65	$1500
5	Female	<25	$500
6	Female	>=25 and < 65	$500

Table 5 – Test Cases

Testing predicates

Predicates arise from requirements in a variety of applications.

A boiler needs to be to be shut down when the following conditions hold:

1. The water level in the boiler is below X lbs. (a)
2. The water level in the boiler is above Y lbs. (b)
3. A water pump has failed. (c)
4. A pump monitor has failed. (d)
5. Steam meter has failed. (e)

The boiler is to be shut down when a or b is true or the boiler is in degraded mode and the steam meter fails. We combine these five conditions to form a compound condition (predicate) for boiler shutdown.

Denoting the five conditions above as a through e, we obtain the following Boolean expression E that when true must force a boiler shutdown:

$$E=a+b+(c+d)e$$

where the + sign indicates “OR” and a multiplication indicates “AND.”

The goal of predicate-based test generation is to generate tests from a predicate p that guarantee the detection of any error that belongs to a class of errors in the coding of p.

A condition is represented formally as a predicate, also known as a Boolean expression. For example, consider the requirement

``*if the printer is ON and has paper then send document to printer*."

This statement consists of a condition part and an action part. The following predicate represents the condition part of the statement.

pr: (printerstatus=ON) ∧ (printertray!= empty)

Test generation from predicates

We will now examine two techniques, named BOR and BRO for generating tests that are guaranteed to detect certain faults in the coding of conditions.

The conditions from which tests are generated might arise from requirements or might be embedded in the program to be tested.

Conditions guard actions.

For example,

if condition then action

Is a typical format of many functional requirements.

Predicates

Relational operators (relop): {<, ≤, >, ≥, =, ≠.} = and == are equivalent.

Boolean operators (bop): {!,∧,∨, xor} also known as {not, AND, OR, XOR}.

Relational expression: e1 relop e2. (e.g. a+b<c)

e1 and e2 are expressions whose values can be compared using relop.

Simple predicate: A Boolean variable or a relational expression. (x<0)

Compound predicate:

Join one or more simple predicates using bop. (gender==“female”∧age>65)

Boolean expressions

Boolean expression: one or more Boolean variables joined by bop. (a∧b∨!c)

a, b, and c are also known as literals. Negation is also denoted by placing a bar over a Boolean expression such as in (a∧b). We also write ab for a∧b and a+b for a∨b when there is no confusion.

Singular Boolean expression: When each literal appears only once, e.g. (a∧b∨!c)

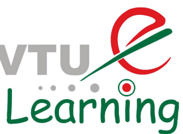
Disjunctive normal form (DNF): Sum of product terms:
e.g. (p q) +(rs) + (a c).

Conjunctive normal form (CNF): Product of sums:
e.g.: (p+q)(r+s)(a+c)

Any Boolean expression in DNF can be converted to an equivalent CNF and vice versa.

e.g.CNF: (p+!r)(p+s)(q+!r)(q+s) is equivalent to DNF: (pq+!rs)

Mutually singular: Boolean expressions e1 and e2 are mutually singular when they do not share any literal.

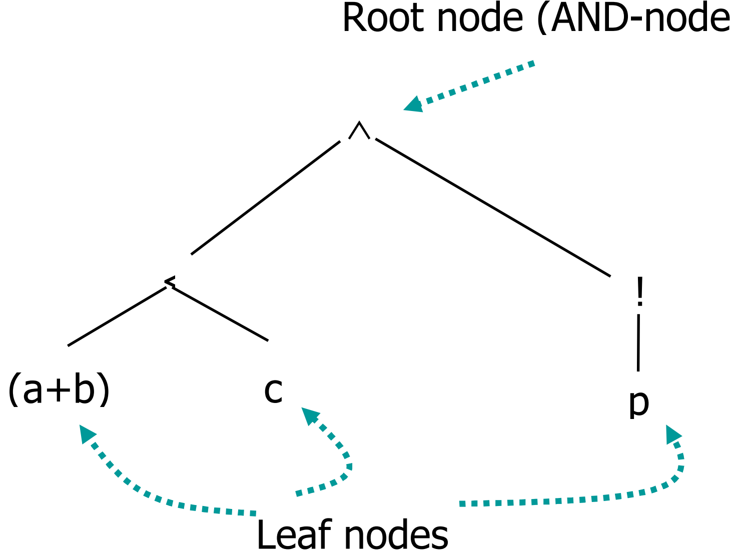


If expression E contains components e1, e2,.. then ei is considered singular only if it is non-singular and mutually singular with the remaining elements of E.

Boolean expressions: Syntax tree representation

Abstract syntax tree (AST) for: (a+b)<c ∧!p.

Notice that internal nodes are labeled by Boolean and relational operators



Fault model for predicate testing

<u>Boolean operator fault</u>: Suppose that the specification of a software module requires that an action be performed when the condition (a<b) ∨ (c>d) ∧e is true.

Here a, b, c, and d are integer variables and e is a Boolean variable.

Correct predicate: (a<b) ∨ (c>d) ∧e

(a<b) ∧ (c>d) ∧e	Incorrect Boolean operator
(a<b) ∨ ! (c>d) ∧e	Incorrect negation operator
(a<b) ∧(c>d) ∨ e	Incorrect Boolean operators
(a<b) ∨ (e>d) ∧c	Incorrect Boolean variable.

Relational operator faults

Correct predicate: (a<b) ∨ (c>d) ∧e

(a==b) ∨ (c>d) ∧e	Incorrect relational operator
(a==b) ∨ (c≤d) ∧e	Two relational operator faults
(a==b) ∨ (c>d) ∨ e	Incorrect Boolean operators

Arithmetic expression faults

Correct predicate:Ec: e1 relop1 e2. Incorrect predicate: Ei: : e3 relop2 e4. Assume that Ec and Ei use the same set of variables.

Ei has an off-by-ε fault if |e3-e4|= ε for any test case for which e1=e2.

Ei has an off-by-ε* fault if |e3-e4|≥ ε for any test case for which e1=e2.

Ei has an off-by-ε+ fault if |e3-e4|> ε for any test case for which e1=e2.

Arithmetic expression faults: Examples

Correct predicate: Ec: a<(b+c). Assume ε=1.

Ei: a<b. Given c=1, Ei has an off-by-1 fault as |a-b|= 1 for a test case for which a=b+c, e.g. <a=2, b=1, c=1>.

Ei: a<b+1. Given c=2, Ei has an off-by-1* fault as |a-(b+1)|≥ 1 for any test case for which a=b+c; <a=4, b=2, c=2>

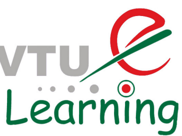
Ei: a<b-1. Given c>0, Ei has an off-by-1+ fault as |a-(b-1)|>1 for any test case for which a=b+c; <a=3, b=2, c=1>.

Goal of predicate testing

Given a correct predicate pc, the goal of predicate testing is to generate a test set T such that there is at least one test case t∈ T for which pc and its faulty version pi, evaluate to different truth values.

Such a test set is said to guarantee the detection of any fault of the kind in the fault model introduced above.

As an example, suppose that pc: a<b+c and pi: a>b+c. Consider a test set T={t1, t2} where t1: <a=0, b=0, c=0> and t2: <a=0, b=1, c=1>.



The fault in pi is not revealed by t1 as both pc and pi evaluate to false when evaluated against t1.

However, the fault is revealed by t2 as pc evaluates to true and pi to false when evaluated against t2.

Missing or extra Boolean variable faults

Correct predicate: $a \vee b$

Missing Boolean variable fault: a

Extra Boolean variable fault: $a \vee b \wedge c$

Predicate constraints: BR symbols

Consider the following Boolean-Relational set of BR-symbols:
BR={t, f, <, =, >, +ε, -ε}

A BR symbol is a constraint on a Boolean variable or a relational expression.

For example, consider the predicate E: a<b and the constraint “>” . A test case that satisfies this constraint for E must cause E to evaluate to false.

<u>Infeasible constraints</u>

A constraint C is considered infeasible for predicate pr if there exists no input values for the variables in pr that satisfy c.

For example, the constraint **t** is infeasible for the predicate $a>b \wedge b>d$ if it is known that d>a.

<u>Predicate constraints</u>

Let pr denote a predicate with n, n>0, $\vee$ and $\wedge$ operators.

A predicate constraint C for predicate pr is a sequence of (n+1) BR symbols, one for each Boolean variable or relational expression in pr. When clear from context, we refer to ``predicate constraint" as simply constraint.

Test case t satisfies C for predicate pr, if each component of pr satisfies the corresponding constraint in C when evaluated against t. Constraint C for predicate pr guides the development of a test for pr, i.e. it offers hints on what the values of the variables should be for pr to satisfy C.

True and false constraints

pr(C) denotes the value of predicate pr evaluated using a test case that satisfies C.

C is referred to as a true constraint when pr(C) is true and a false constraint otherwise.

A set of constraints S is partitioned into subsets St and Sf, respectively, such that for each C in St, pr(C) =true, and for any C in Sf, pr(C) =false. S= St ∪ Sf.

Predicate constraints: Example

Consider the predicate pr: $b\wedge (r<s) \vee (u\geq v)$ and a constraint C: (t, =, >). The following test case satisfies C for pr.

<b=true, r=1, s=1, u=1, v=0>

The following test case does not satisfy C for pr.

<b=true, r=1, s=2, u=1, v=2>

Predicate testing: criteria

Given a predicate pr, we want to generate a test set T such that

- T is minimal and
- T guarantees the detection of any fault in the implementation of pr; faults correspond to the fault model we discussed earlier.

We will discuss three such criteria named BOR, BRO, and BRE.

Predicate testing: BOR testing criterion

A test set T that satisfies the BOR testing criterion for a compound predicate pr, guarantees the detection of single or multiple Boolean operator faults in the implementation of pr.

T is referred to as a BOR-adequate test set and sometimes written as TBOR.

Predicate testing: BRO testing criterion

A test set T that satisfies the BRO testing criterion for a compound predicate pr, guarantees the detection of single or multiple Boolean operator and relational operator faults in the implementation of pr.

T is referred to as a BRO-adequate test set and sometimes written as TBRO.

Predicate testing: BRE testing criterion

A test set T that satisfies the BRE testing criterion for a compound predicate pr, guarantees the detection of single or multiple Boolean operator, relational expression, and arithmetic expression faults in the implementation of pr.

T is referred to as a BRE-adequate test set and sometimes written as TBRE.

Predicate testing: guaranteeing fault detection

Let Tx, $x \in \{BOR, BRO, BRE\}$, be a test set derived from predicate pr. Let pf be another predicate obtained from pr by injecting single or multiple faults of one of three kinds: Boolean operator fault, relational operator fault, and arithmetic expression fault.

Tx is said to guarantee the detection of faults in pf if for some $t \in Tx$, $p(t) \neq pf(t)$.

<u>Guaranteeing fault detection: example</u>

Let $pr=a<b \wedge c>d$

Constraint set S={(t, t), (t,f), (f, t)}

Let TBOR={t1, t2, t3} is a BOR adequate test set that satisfies S.

t1: <a=1, b=2, c=1, d=0 >; Satisfies (t, t), i.e. a<b is true and
c<d is also true.
t2: <a=1, b=2, c=1, d=2 >; Satisfies (t, f)
t3: <a=1, b=0, c=1, d=0 >; Satisfies (f, t)

Algorithms for generating BOR, BRO, and BRE adequate tests

Review of a basic definition: The cross product of two sets A and B is defined as:

$A \times B=\{(a,b) | a \in A \text{ and } b \in B\}$

The onto product of two sets A and B is defined as:

$A \otimes B=\{(u,v) | u \in A, v \in B$, such that each element of A appears at least once as u and each element of B appears once as v.$\}$

<u>Set products: Example</u>

Let A={t, =, >} and B={f, <}

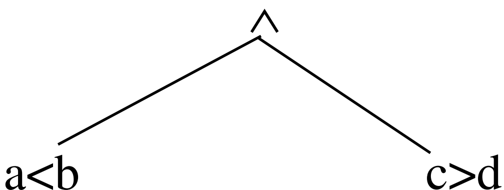
$A \times B=\{(t, f), (t, <), (=, f), (=, <), (>,f), (>,<)\}$

A⊗B ={(t, f), (=,<), (>,<)}

Generation of BOR constraint set

We want to generate TBOR for: pr: a<b ∧ c>d

First, generate syntax tree of pr.

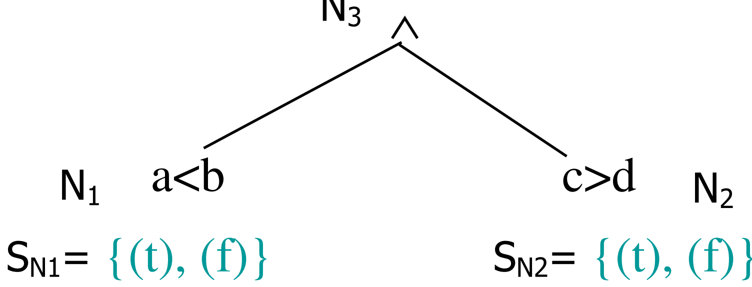


1. Generation of BOR constraint set

We will use the following notation:

SN is the constraint set for node N in the syntax tree for pr.
SNt is the true constraint set for node N in the syntax tree for pr.
SNf is the false constraint set for node N in the syntax tree for pr.
SN= SNt ∪ SNf .

Second, label each leaf node with the constraint set {(t), (f)}.
We label the nodes as N1, N2, and so on for convenience.

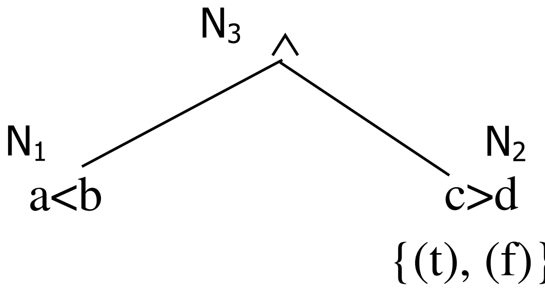


Notice that N1 and N2 are direct descendents of N3 which is an AND-node.

Third, compute the constraint set for the next higher node in the syntax tree, in this case N3. For an AND node, the formulae used are the following.

SN3t = SN1t ⊗ SN2t ={(t)} ⊗ {(t)}={(t, t)}

$$
\begin{aligned}
SN3f &= (SN1f \times \{t2\}) \cup (\{t1\} \times SN2f \\
&= (\{(f)\} \times \{(t)\}) \cup (\{(t)\} \times \{(f)\}) \\
&= \{(f, t)\} \cup \{(t, f)\} \\
&= \{(f, t), \{(t, f)\}
\end{aligned}
$$

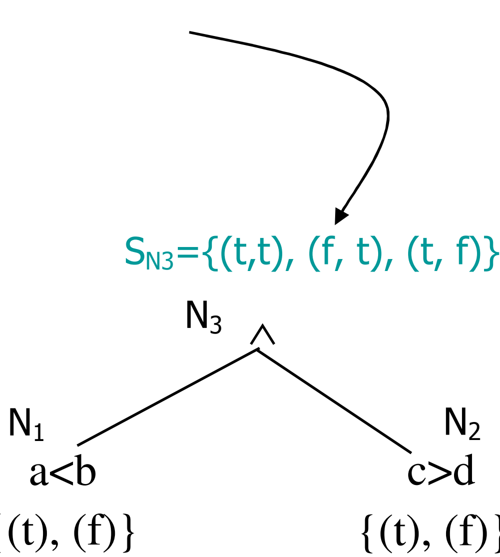


Generation of T_{BOR}

As per our objective, we have computed the BOR constraint set for the root node of the AST(pr). We can now generate a test set using the BOR constraint set associated with the root node.

SN3 contains a sequence of three constraints and hence we get a minimal test set consisting of three test cases. Here is one possible test set.

TBOR ={t1, t2, t3}
t1=<a=1, b=2, c=6, d=5> (t, t)
t2=<a=1, b=0, c=6, d=5> (f, t)
t3=<a=1, b=2, c=1, d=2> (t, f)



2. Generation of BRO constraint set

Recall that a test set adequate with respect to a BRO constraint set for predicate pr, guarantees the detection of all combinations of single or multiple Boolean operator and relational operator faults.

BRO constraint set

The BRO constraint set S for relational expression *e1 relop e2*:

S={(>), (=), (<)}

Separation of S into its true (St) and false (Sf)components:

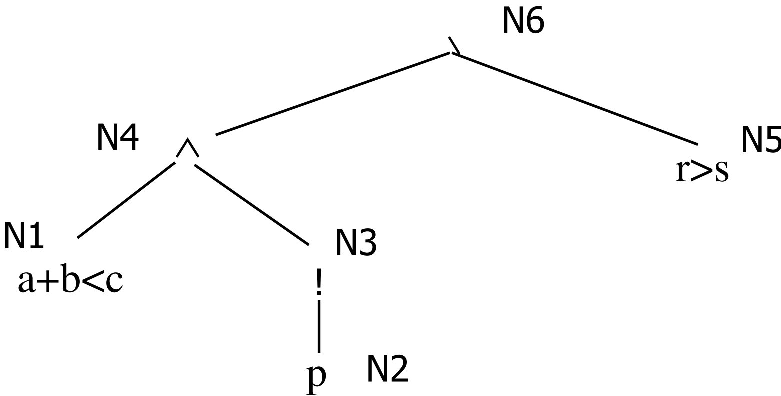
relop	St	Sf
relop: >	St={(>)}	Sf={(=), (<)}
relop: ≥	St={(>), (=)}	Sf={(<)}
relop: =	St={(=)}	Sf={(<), (>)}
relop: <	St={(<)}	Sf={(=), (>)}
relop: ≤	St={(<), (=)}	Sf={(>)}

Note: tN denotes an element of StN. fN denotes an element of SfN

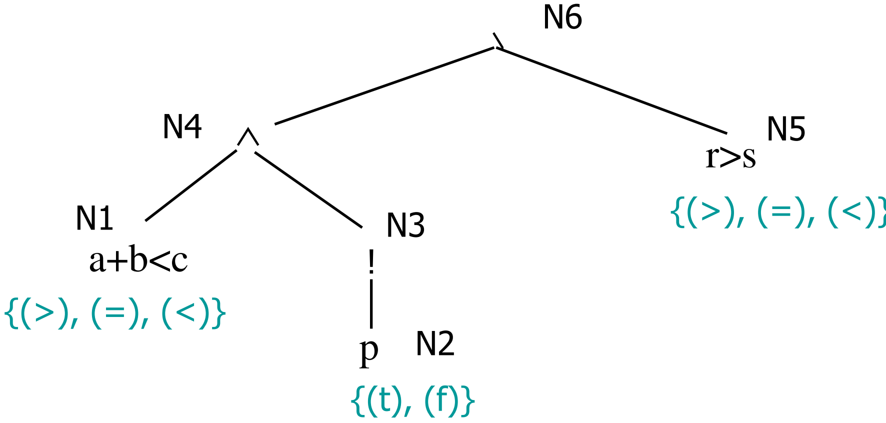
<u>BRO constraint set: Example</u>

pr: (a+b<c)∧!p ∨ (r>s)

Step 1: Construct the AST for the given predicate.



Step 2: Label each leaf node with its constraint set S.

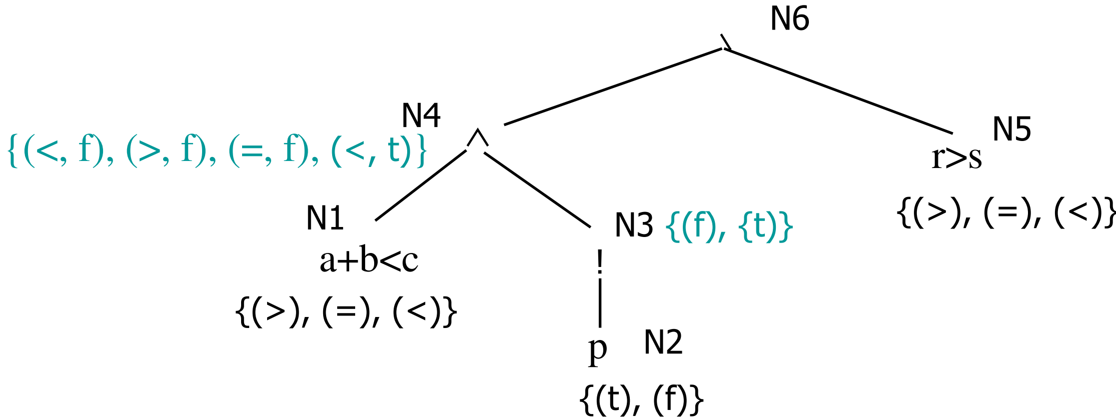


Step 3: Traverse the tree and compute constraint set for each internal node.

StN3=SN2f={(f)} SfN3=SN2t= {(t)}

StN4=SN1t ⊗ SN3t={(<)} ⊗{(f)}={(<, f)}

SfN4= (SfN1 × {(tN3)}) ∪ ({(tN1)} × SfN3)
=({(>,=)} ×{(f)}) ∪ {(<)} ×{(t)})
={(>, f), (=, f)} ∪ {(<, t)}
={(>, f), (=, f), (<, t)}

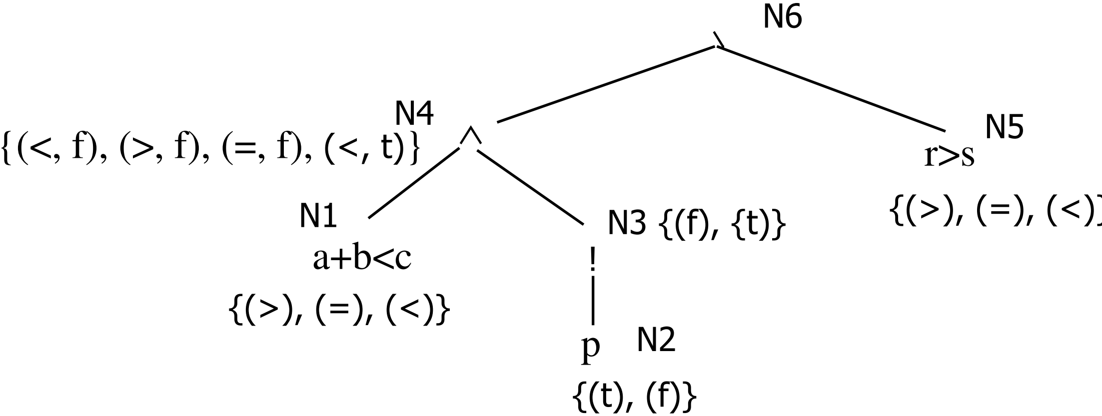


Next compute the constraint set for the root node (this is an OR-node).

SfN6=SfN4 ⊗ SfN5
=\{(>,f),(=,f),(<,t)\} ⊗\{(=),(<)\}=\{(<, f)\}
=\{(>,f,=), (=,f,<),(<,t,=)\}

StN6= (StN4 × \{(fN5)\})∪ (\{(fN4)\} × StN5)
=(\{(<,f)\} ×\{(=)\}) ∪ \{(>,f)\} ×\{(>)\})
=\{(<,f,=)\} ∪ \{(>,f,>)\}
=\{(<,f,=),(>,f,>)\}

Constraint set for pr: (a+b<c)∧!p ∨ (r>s)



BOR constraints for non-singular expressions

Test generation procedures described so far are for singular predicates. Recall that a singular predicate contains only one occurrence of each variable.

We will now learn how to generate BOR constraints for non-singular predicates.

First, let us look at some non-singular expressions, their respective disjunctive normal forms (DNF), and their mutually singular components.

Non-singular expressions and DNF: Examples

Predicate (pr)	**DNF**	**Mutually singular components in pr**
ab(b+c)	**abb+abc**	**a; b(b+c)**

a(bc+ bd)	abc+abd	a; (bc+bd)
a(!b+!c)+cde	a!ba +a!c+cde	a; !b+!c+ cde
a(bc+!b+de)	abc+a!b+ade	a; bc+!b; de

Generating BOR constraints for non-singular expressions

We proceed in two steps.

First we will examine the Meaning Impact (MI) procedure for generating a minimal set of constraints from a possibly non-singular predicate.

Next, we will examine the procedure to generate BOR constraint set for a non-singular predicate.

Meaning Impact (MI) procedure

Given Boolean expression E in DNF, the MI procedure produces a set of constraints SE that guarantees the detection of missing or extra NOT (!) operator faults in the implementation of E.

MI procedure: An Example

Consider the non-singular predicate: a(bc+!bd). Its DNF equivalent is:
E=abc+a!bd.

Note that a, b, c, and d are Boolean variables and also referred to as literals. Each literal represents a condition. For example, a could represent r<s.

Step 0: Express E in DNF notation. Clearly, we can write E=e1+e2, where e1=abc and e2=a!bd.

Step 1: Construct a constraint set Te1 for e1 that makes e1 true. Similarly construct Te2 for e2 that makes e2 true.

Te1 ={(t,t,t,t), (t,t,t,f)} Te2 ={(t,f,t,t), (t,f,f,t)}

Note that the four t's in the first element of Te1 denote the values of the Boolean variables a, b,c, and d, respectively. The second element, and others, are to be interpreted similarly.

Step 2: From each Tei , remove the constraints that are in any other Tej. This gives us TSei and TSej. Note that this step will lead TSei ∩TSej =∅.

There are no common constraints between Te1 and Te2 in our example. Hence we get:

TSe1 ={(t,t,t,t), (t,t,t,f)} TSe2 ={(t,f,t,t), (t,f,f,t)}

Step 3: Construct StE by selecting one element from each Te.

StE ={(t,t,t,t), (t,f,f,f)}

Step 4: For each term in E, obtain terms by complementing each literal, one at a time.

e11= !abc	e21= a!bc	e31= ab!c
e12= !a!bd	e22= abd	e32= a!b!d

From each term e above, derive constraints Fe that make e true. We get the following six sets.

Fe11= {(f,t,t,t), (f,t,t,f)}
Fe21= {(t,f,t,t), (t,f,t,f)}
Fe31= {(t,t,f,t), (t,t,f,f)}

Fe12= {(f,f,t,t), (f,f,f,t)}
Fe22= {(t,t,t,t), (t,t,f,t)}
Fe32= {(t,f,t,f), (t,f,f,f)}

Step 5: Now construct FSe by removing from Fe any constraint that appeared in any of the two sets Te constructed earlier.

FSe11= FSe11
FSe21= {(t,f,t,f)}
FSe31= FSe13

FSe12= FSe12
FSe22= {(t,t,f,t)}
FSe32= FSe13

Step 6: Now construct SfE by selecting one constraint from each Fe

SfE ={(f,t,t,f), (t,f,t,f), (t,t,f,t), (f,f,t,t)}

Step 7: Now construct SE= StE ∪SfE

SE={{(t,t,t,t), (t,f,f,f), (f,t,t,f), (t,f,t,f), (t,t,f,t), (f,f,t,t)}

Note: Each constraint in StE makes E true and each constraint in SfE makes E false.

BOR-MI-CSET procedure

The BOR-MI-CSET procedure takes a non-singular expression E as input and generates a constraint set that guarantees the detection of Boolean operator faults in the implementation of E.

The BOR-MI-CSET procedure using the MI procedure described earlier.

We illustrate it with an example.

BOR-MI-CSET: Example

Consider a non-singular Boolean expression:
E= a(bc+!bd)

Mutually non-singular components of E:

e1=a
e2=bc+!bd

We use the BOR-CSET procedure to generate the constraint set for e1 (singular component) and MI-CSET procedure for e2 (non-singular component).

For component e1 we get:
Ste1={t}. Sfe1={f}
Recall that Ste1 is true constraint set for e1 and Sfe1 is false constraint set for e1.

Component e2 is a DNF expression. We can write e2=u+v where u=bc and v=!bd.

Let us now apply the MI-CSET procedure to obtain the BOR constraint set for e2.

As per Step 1 of the MI-CSET procedure we obtain:

Tu={(t,t,t), (t,t,f)} Tv={(f,t,t), (f,f,t)}

Applying Steps 2 and 3 to Tu and Tv we obtain:

TSu=Tu TSv=Tv

Ste2={(t,t,f), (f, t, t)}

Next we apply Step 4 to u and v. We obtain the following complemented expressions from u and v:

u1=!bc u2=b!c
v1=bd v2=!b!d

Continuing with Step 4 we obtain:

Fu1={(f,t,t), (f,t,f)} Fu2=(t,f,t), (t,f,f)}
Fv1={(t,t,t), (t,f,t)} Fv2={(f,t,f), (f,f,f)}

Next we apply Step 5 to the F constraint sets to obtain:

FSu1={(f,t,f)} FSu2=(t,f,t), (t,f,f)}
FSv1={(t,f,t)} FSv2={(f,t,f), (f,f,f)}

Applying Step 6 to the FS sets leads to the following

Sfe2={(f,t,f), (t,f,t)}

Combing the true and false constraint sets for e2 we get:

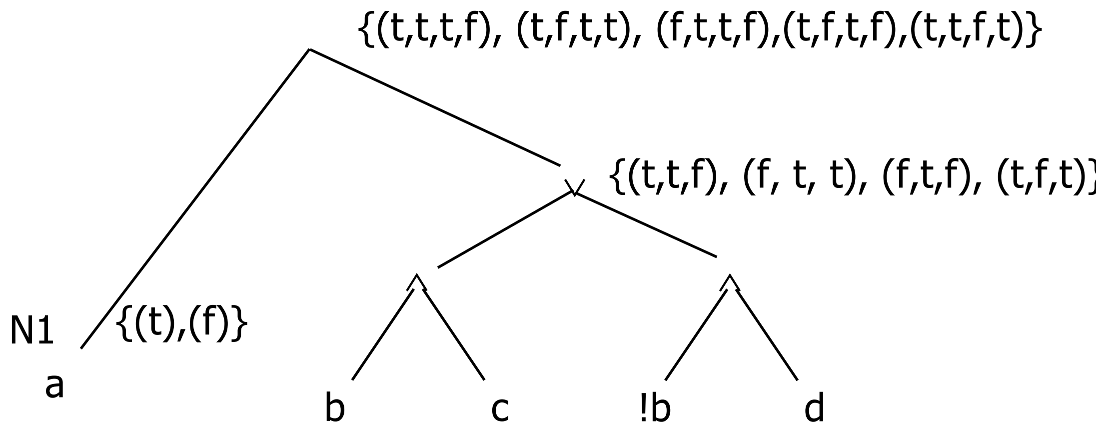
Se2={(t,t,f), (f, t, t), {(f,t,f), (t,f,t)}

Summary:

Ste1={(t)} Sfe1={(f)} from BOR-CSET procedure.
Ste2={(t,t,f), (f, t, t)} Sfe2={(f,t,f), (t,f,t)} from MI-CSET procedure.

We now apply Step 2 of the BOR-CSET procedure to obtain the constraint set for the entire expression E.

StN3=StN1 ⊗ StN22
SfN3=(SfN1 × {t2})∪({t1} × SfN2)



Summary

Equivalence partitioning and boundary value analysis are the most commonly used methods for test generation while doing functional testing.

Given a function *f* to be tested in an application, one can apply these techniques to generate tests for *f.*

Most requirements contain conditions under which functions are to be executed. Predicate testing procedures covered are excellent means to generate tests to ensure that each condition is tested adequately

Usually one would combine equivalence partitioning, boundary value analysis, and predicate testing procedures to generate tests for a requirement of the following type:if condition then action 1, action 2, …action n;