

Foundations of Software Testing

Chapter 2: Test Generation: Requirements

Aditya P. Mathur
Purdue University



These slides are copyrighted. They are for use with the **Foundations of Software Testing** book by Aditya Mathur. Please use the slides but do not remove the copyright notice.

Last update: September 3, 2007

Learning Objectives

- Equivalence class partitioning
- Boundary value analysis
- Test generation from predicates

Essential *black-box* techniques for generating tests for functional testing.

© Aditya P. Mathur 2006

2

Applications of test generation techniques

Test generation techniques described in this chapter belong to the **black-box** testing category.

These techniques are useful during **functional testing** where the objective is to test whether or not an application, unit, system, or subsystem, correctly implements the functionality as per the given requirements

© Aditya P. Mathur 2006

3

The test selection problem



4

Requirements and test generation

Requirements serve as the starting point for the generation of tests. During the initial phases of development, requirements may exist only in the minds of one or more people.

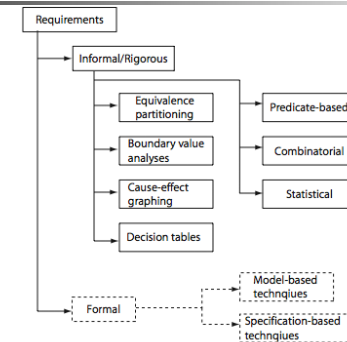
These requirements, more aptly ideas, are then specified rigorously using modeling elements such as *use cases*, *sequence diagrams*, and *statecharts in UML*.

Rigorously specified requirements are often transformed into formal requirements using *requirements specification languages* such as *Z*, *S*, and *RSML*.

© Aditya P. Mathur 2006

5

Test generation techniques



© Aditya P. Mathur 2006

6

Test selection problem

Let D denote the *input domain* of a program P . The test selection problem is to *select a subset T of tests* such that execution of P against each element of T will *reveal all errors in P* .

In general there *does not exist any algorithm* to construct such a test set. However, there are *heuristics* and model based methods that can be used to generate tests that will reveal certain type of faults.

© Aditya P. Mathur 2006

7


Test selection problem (contd.)

The challenge is to construct a test set $T \subseteq D$ that will *reveal as many errors in P as possible*.

The problem of test selection is difficult due primarily to the *size* and *complexity* of the input domain of P .

© Aditya P. Mathur 2006

8




Exhaustive testing

The *large size* of the input domain *prevents* a tester from *exhaustively testing* the program under test against all possible inputs. By "exhaustive" testing we mean testing the given program against every element in its input domain.

The *complexity* makes it harder to *select individual tests*.

© Aditya P. Mathur 2006 9




Large input domain

Consider program P that is required to *sort a sequence of integers* into ascending order. Assuming that P will be executed on a machine in which integers range from -32768 to 32767, the input domain of pr consists of *all possible sequences of integers* in the range [-32768, 32767].

If there is *no limit* on the size of the sequence that can be input, then the input domain of P is *infinitely large* and P can never be tested exhaustively. If the size of the input sequence is limited to, say $N_{max} > 1$, then the size of the input domain depends on the value of N.

© Aditya P. Mathur 2006 10




Complex input domain

Consider a procedure P in a payroll processing system that takes an employee record as input and computes the weekly salary. For simplicity, assume that the employee *record consists of the following items with their respective types and constraints:*

ID: int;	ID is 3-digits long from 001 to 999.
name: string;	name is 20 characters long; each character belongs to the set of 26 letters and a space character.
rate: float;	rate varies from \$5 to \$10 per hour; rates are in multiples of a quarter.
hoursWorked: int;	hoursWorked varies from 0 to 60.

© Aditya P. Mathur 2006 11



Equivalence class partitioning

12

Equivalence partitioning

Test selection using *equivalence partitioning* allows a tester to *subdivide the input domain* into a relatively small number of sub-domains (say N).

In strict mathematical terms, the sub-domains by definition are *disjoint*. Each subset is known as an *equivalence class*.

© Aditya P. Mathur 2006

13

Program behavior and equivalence classes

The N equivalence classes are created assuming that the program under test exhibits the *same behavior* on all elements, i.e. tests, within a class.

This assumption allows the tester to *select exactly one test from each equivalence class* resulting in a test suite of exactly N tests.

© Aditya P. Mathur 2006

14

Faults targeted

The entire set of inputs to any application can be divided into at least two subsets: one containing *all the expected, or legal, inputs* (E) and the other containing *all unexpected, or illegal, inputs* (U).

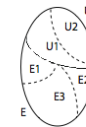
Each of the two subsets, can be *further subdivided into subsets* on which the application is required to *behave differently* (e.g. E1, E2, E3, and U1, U2).

© Aditya P. Mathur 2006

15

Faults targeted (contd.)

Equivalence class partitioning selects tests that target any faults in the application that cause it to behave incorrectly when the input is in either of the two classes or their subsets.



© Aditya P. Mathur 2006

16

Example 1

Consider an application A that takes an integer denoted by **age** as input. Let us suppose that the only legal values of **age** are in the range [1..120]. The set of input values is now divided into a set E containing all integers in the range [1..120] and a set U containing the remaining integers.

The diagram shows a large irregular shape representing 'All integers'. A smaller region within it is labeled '[1..120]'. The remaining area is labeled 'Other integers'.

© Aditya P. Mathur 2006 17

Example 1 (contd.)

Further, assume that the application is required to process all values in the range [1..61] in accordance with requirement R1 and those in the range [62..120] according to requirement R2. Thus E is further subdivided into two regions depending on the expected behavior.

Similarly, it is expected that all **invalid inputs less than or equal to 1 are to be treated in one way while all greater than 120 are to be treated differently**. This leads to a subdivision of U into two categories.

© Aditya P. Mathur 2006 18

Example 1 (contd.)

The diagram shows a large irregular shape representing 'All integers'. It is partitioned into four regions: '<1', '[62-120]', '[1..61]', and '>120'.

© Aditya P. Mathur 2006 19

Example 1 (contd.)

Tests selected using the equivalence partitioning technique aim at targeting faults in the application under test with respect to **inputs in any of the four regions**, i.e. two regions containing expected inputs and two regions containing the unexpected inputs.

It is expected that **any single test selected from the range [1..61] will reveal any fault with respect to R1**. Similarly, **any test selected from the region [62..120] will reveal any fault with respect to R2**. A similar expectation applies to the two regions containing the unexpected inputs.

© Aditya P. Mathur 2006 20

Effectiveness

The *effectiveness* of tests generated using equivalence partitioning for testing application A, is judged by the *ratio of the number of faults these tests are able to expose to the total faults lurking in A*.

As is the case with any test selection technique in software testing, the *effectiveness of tests selected using equivalence partitioning is less than 1* for most practical applications. The effectiveness can be improved through an unambiguous and complete *specification of the requirements* and carefully selected tests using the *equivalence partitioning* technique described in the following sections.

© Aditya P. Mathur 2006

21

Example 2

This example shows a few ways to define equivalence classes based on the knowledge of *requirements* and the *program text*.

Consider that `wordCount` method takes a word `w` and a filename `f` as input and returns the number of occurrences of `w` in the text contained in the file named `f`. An exception is raised if there is no file with name `f`.

© Aditya P. Mathur 2006

22

Example 2 (contd.)

begin

String w, f

Input w, f

if (not exists(f) {raise exception; return(0);}

if(length(w)==0)return(0);

if(empty(f))return(0);

return(getCount(w,f));

end

Using the partitioning method described in the examples above, we obtain the following equivalence classes.

© Aditya P. Mathur 2006

23

Example 2 (contd.)

Equivalence class	w	f
E1	non-null	exists, not empty
E2	non-null	does not exist
E3	non-null	exists, empty
E4	null	exists, not empty
E5	null	does not exist
E6	null	exists, empty

© Aditya P. Mathur 2006

24

Example 2 (contd.)

Note that the number of equivalence classes *without any knowledge of the program code may be fewer* (e.g., 2 for file exists and file does not exist), whereas the *number of equivalence classes derived with the knowledge of partial code is 6*.

Of course, an *experienced tester will likely derive the six equivalence classes* given above, and perhaps more, even before the code is available

Equivalence classes based on program output

In some cases the equivalence classes are based on the *output* generated by the program. For example, suppose that a program outputs an integer.

It is worth asking: ``*Does the program ever generate a 0? What are the maximum and minimum possible values of the output?*''

These two questions lead to two the following *equivalence classes based on outputs*:

Equivalence classes based on program output (contd.)

- E1: Output value v is 0.
- E2: Output value v is the maximum possible.
- E3: Output value v is the minimum possible.
- E4: All other output values.

Based on the *output equivalence* classes one may now *derive equivalence classes for the inputs*. Thus each of the four classes given above might lead to one equivalence class consisting of inputs.

Equivalence classes for variables: range

Eq. Classes	Example	
	Constraints	Classes
One class with values inside the range and two with values outside the range.	speed $\in [60..90]$	{50}, {75}, {92}
	area: float area ≥ 0.0	{{-1.0}, {15.52}}
	age: int	{{-1}, {56}, {132}}
	letter: char	{{J}, {3}}

Equivalence classes for variables: strings

Eq. Classes	Example	
	Constraints	Classes
At least one containing all legal strings and one all illegal strings based on any constraints.	firstname: string	{{ε}, {Sue}, {Looong Name}}

© Aditya P. Mathur 2006

29

Equivalence classes for variables: enumeration

Eq. Classes	Example	
	Constraints	Classes
Each value in a separate class	autocolor: {red, blue, green}	{{red}, {blue}, {green}}
	up: boolean	{{true}, {false}}

© Aditya P. Mathur 2006

30

Equivalence classes for variables: arrays

Eq. Classes	Example	
	Constraints	Classes
One class containing all legal arrays, one containing the empty array, and one containing a larger than expected array.	int [] aName: new int{3};	{{[]}, {[-10, 20]}, {[-9, 0, 12, 15]}}

© Aditya P. Mathur 2006

31

Equivalence classes for variables: compound data type

Arrays in Java and structures in C++ are *compound types*. Such input types may arise while testing components of an application such as a function or an object.

While generating equivalence classes for such inputs, one must *consider legal and illegal values for each component* of the structure. The next example illustrates the derivation of *equivalence classes for an input variable that has a compound type*.

© Aditya P. Mathur 2006

32

Equivalence classes for variables: compound data type: Example

```
struct transcript
{
    string fName; // First name.
    string lName; // Last name.
    string cTitle [200]; // Course titles.
    char grades [200]; // Letter grades corresponding
                       // to course titles.
}
```

© Aditya P. Mathur 2006

33

Unidimensional partitioning

One way to partition the input domain is to *consider one input variable at a time*. Thus each input variable leads to a partition of the input domain. We refer to this style of partitioning as *unidimensional* equivalence partitioning or simply unidimensional partitioning.

This type of partitioning is commonly used.

© Aditya P. Mathur 2006

34

Multidimensional partitioning

Another way is to consider the input domain I as the *set product of the input variables* and define a relation on I. This procedure creates one partition consisting of several equivalence classes. We refer to this method as *multidimensional* equivalence partitioning or simply multidimensional partitioning.

Multidimensional partitioning *leads to a large number of equivalence classes* that are difficult to manage manually. Many classes so created might be *infeasible*. Nevertheless, equivalence classes so created offer an increased variety of tests as is illustrated in the next section.

© Aditya P. Mathur 2006

35

Partitioning Example

Consider an application that requires two integer inputs *x* and *y*. Each of these inputs is expected to lie in the following ranges:
 $3 \leq x \leq 7$ and $5 \leq y \leq 9$.

For *unidimensional partitioning* we apply the partitioning guidelines to *x* and *y* individually. This leads to the following *six equivalence classes*.

© Aditya P. Mathur 2006

36

Partitioning Example (contd.)

E1: $x < 3$ E2: $3 \leq x \leq 7$ E3: $x > 7$ — y ignored.

E4: $y < 5$ E5: $5 \leq y \leq 9$ E6: $y > 9$ — x ignored.

For *multidimensional partitioning* we consider the input domain to be the set product $X \times Y$. This *leads to 9 equivalence classes*.

© Aditya P. Mathur 2006 37

Partitioning Example (contd.)

E1: $x < 3, y < 5$ E2: $x < 3, 5 \leq y \leq 9$ E3: $x < 3, y > 9$

E4: $3 \leq x \leq 7, y < 5$ E5: $3 \leq x \leq 7, 5 \leq y \leq 9$ E6: $3 \leq x \leq 7, y > 9$

E7: $x > 7, y < 5$ E8: $x > 7, 5 \leq y \leq 9$ E9: $x > 7, y > 9$

© Aditya P. Mathur 2006 38

Systematic procedure for equivalence partitioning

1. Identify the input domain: Read the requirements carefully and identify all input and output variables, their types, and any conditions associated with their use.

Environment variables, such as class variables used in the method under test and environment variables in Unix, Windows, and other operating systems, also serve as input variables. Given the set of values each variable can assume, an approximation to the input domain is the product of these sets.

© Aditya P. Mathur 2006 39

Systematic procedure for equivalence partitioning (contd.)

2. Equivalence classing: Partition the set of values of each variable into disjoint subsets. Each subset is an equivalence class. Together, the equivalence classes based on an input variable partition the input domain. Partitioning the input domain using values of one variable, is done based on the expected behavior of the program.

Values for which the program is expected to behave in the "same way" are grouped together. Note that "same way" needs to be defined by the tester.

© Aditya P. Mathur 2006 40



Systematic procedure for equivalence partitioning (contd.)

3. Combine equivalence classes: This step is usually omitted and the equivalence classes defined for each variable are directly used to select test cases. However, by not combining the equivalence classes, one misses the opportunity to generate useful tests.

The equivalence classes are combined *using the multidimensional partitioning* approach described earlier.



Systematic procedure for equivalence partitioning (contd.)

4. Identify infeasible equivalence classes: An infeasible equivalence class is one that contains a combination of input data that cannot be generated during test. Such an equivalence class might arise due to several reasons.

For example, suppose that an application is tested via its GUI, i.e. data is input using commands available in the GUI. The GUI might disallow invalid inputs by offering a palette of valid inputs only. There might also be constraints in the requirements that render certain equivalence infeasible.



Boiler control example (BCS)

The control software of **BCS**, abbreviated as **CS**, is required to offer several options. One of the options, **C** (for control), is used by a human operator to give one of three commands (**cmd**): change the boiler temperature (**temp**), shut down the boiler (**shut**), and cancel the request (**cancel**).

Command **temp** causes **CS** to ask the operator to enter the amount by which the temperature is to be changed (**tempch**). Values of **tempch** are in the range -10..10 in increments of 5 degrees Fahrenheit. A temperature change of 0 is not an option.

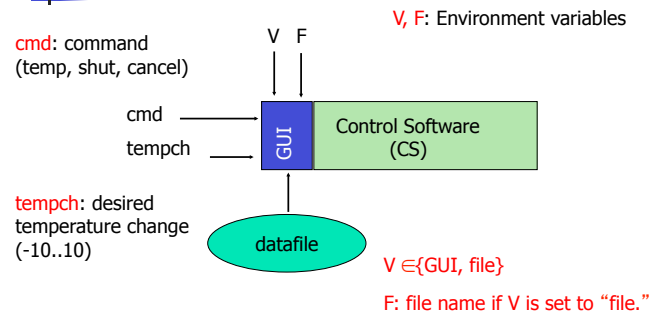


BCS: example (contd.)

Selection of option **C** forces the **BCS** to examine variable **V**. If **V** is set to **GUI**, the operator is asked to enter one of the three commands via a GUI. However, if **V** is set to **file**, **BCS** obtains the command from a command file.

The command file may contain any one of the three commands, together with the value of the temperature to be changed if the command is **temp**. The **file name** is obtained from variable **F**.

BCS: example (contd.)



© Aditya P. Mathur 2006

45

BCS: example (contd.)

Values of **V** and **F** can be altered by a *different module* in BCS.

In response to **temp** and **shut** commands, the control software is required to generate appropriate signals to be sent to the boiler heating system.

© Aditya P. Mathur 2006

46

BCS: example (contd.)

We assume that the control software is to be tested in a simulated environment. The tester takes on the role of an operator and interacts with the CS via a GUI.

The *GUI forces the tester to select from a limited set of values* as specified in the requirements. For example, the only options available for the value of **tempch** are -10, -5, 5, and 10. We refer to these four values of **tempch** as **tvalid** while all other values as **tinvalid**.

© Aditya P. Mathur 2006

47

BCS: 1. Identify input domain

The first step in generating equivalence partitions is to *identify the (approximate) input domain*. Recall that the domain identified in this step will likely be a superset of the complete input domain of the control software.

First we *examine the requirements*, identify input variables, their types, and values. These are listed in the following table.

© Aditya P. Mathur 2006

48

BCS: Variables, types, values

Variable	Kind	Type	Value(s)
V	Environment	Enumerated	File, GUI
F	Environment	String	A file name
cmd	Input via GUI/ File	Enumerated	{temp, cancel, shut}
tempch	Input via GUI/ File	Enumerated	{-10, -5, 5, 10}

© Aditya P. Mathur 2006

49

BCS: Input domain

Input domain $\subseteq S = V \times F \times \text{cmd} \times \text{tempch}$

Sample values in the input domain (--: don't care):

(GUI, --, shut, --), (file, cmdfile, shut, --)

(file, cmdfile, temp, 5)

© Aditya P. Mathur 2006

50

BCS: 2. Equivalence classing

Variable	Partition
V	{{GUI}, {file}, {undefined}}
F	{{fvalid}, {finvalid}}
cmd	{{temp}, {cancel}, {shut}, {cinvalid}}
tempch	{{tvalid}, {tinvalid}}

© Aditya P. Mathur 2006

51

BCS: 3. Combine equivalence classes (contd.)

Note that tinvalid, tvalid, finvalid, and fvalid denote sets of values. "undefined" denotes one value.

There is a total of $3 \times 4 \times 2 \times 5 = 120$ equivalence classes.

Sample equivalence class: {(GUI, fvalid, temp, -10)}

Note that each of the classes listed above represents an infinite number of input values for the control software. For example, {(GUI, fvalid, temp, -10)} denotes an infinite set of values obtained by replacing fvalid by a string that corresponds to the name of an existing file. Each value is a potential input to the BCS.

© Aditya P. Mathur 2006

52

BCS: 4. Discard infeasible equivalence classes

Note that the amount by which the boiler temperature is to be changed is only considered when the operator selects **temp** for **cmd**. Thus all equivalence classes that match the following template are infeasible.

$\{(V, F, \{\text{cancel, shut, cinvalid}\}, \text{tvalid} \cup \text{tinvalid})\}$

This parent-child relationship between **cmd** and **tempch** renders *infeasible* a total of $3 \times 2 \times 3 \times 5 = 90$ equivalence classes.

© Aditya P. Mathur 2006

53

BCS: 4. Discard infeasible equivalence classes (contd.)

After having discarded all infeasible equivalence classes, we are left with the following *testable (or feasible) equivalence classes*.

```
{{(GUI,_,temp,tvalid)} {(GUI,_,shut,NA)} {(GUI,_,cancel,NA)}  
{(file,fvalid,temp,tvalid+tinvalid)} {(file,fvalid,shut,_)}  
{(file,fvalid,cancel,_)} {(file,fvalid,cinvald,_)}  
{(file,finvalid,NA,NA)}  
{(undefined,NA,NA,NA)}
```

_ Input Not Used
NA Input Not Allowed

© Aditya P. Mathur 2006

54

Selecting test data

Given a set of equivalence classes that form a partition of the input domain, it is *relatively straightforward to select tests*. However, options exist in the presence of _ values.

In the most general case, *a tester simply selects one test that serves as a representative of each equivalence class*.

© Aditya P. Mathur 2006

55

Boundary value analysis



56



Errors at the boundaries

Experience indicates that programmers make mistakes in processing values at and near the *boundaries of equivalence classes*.

For example, suppose that method M is required to **compute a function f1 when $x \leq 0$ is true and function f2 otherwise**. However, M has an **error** due to which it **computes f1 for $x < 0$ and f2 otherwise**.

Obviously, this **fault is revealed**, though not necessarily, when **M is tested against $x=0$** but not if the input test set is, for example, $\{-4, 7\}$ derived using equivalence partitioning. In this example, the value $x=0$, lies at the boundary of the equivalence classes $x \leq 0$ and $x > 0$.

© Aditya P. Mathur 2006

57



Boundary value analysis (BVA)

Boundary value analysis is a test selection technique that targets faults in applications at the boundaries of equivalence classes.

While equivalence partitioning selects tests from within equivalence classes, boundary value analysis focuses on tests at and near the boundaries of equivalence classes.

Certainly, tests derived using either of the two techniques may **overlap**.

© Aditya P. Mathur 2006

58



BVA: Procedure

- 1 **Partition the input domain using unidimensional partitioning.** This leads to as many partitions as there are input variables. Alternately, a single partition of an input domain can be created using multidimensional partitioning. We will generate several sub-domains in this step.
- 2 **Identify the boundaries for each partition.** Boundaries may also be identified using special relationships amongst the inputs.
- 3 **Select test data such that each boundary value occurs in at least one test input.**

© Aditya P. Mathur 2006

59



BVA: Example: 1. Create equivalence classes

Assuming that an item **code** must be in the range 99..999 and **quantity** in the range 1..100,

Equivalence classes for code:

- E1: Values less than 99.
- E2: Values in the range.
- E3: Values greater than 999.

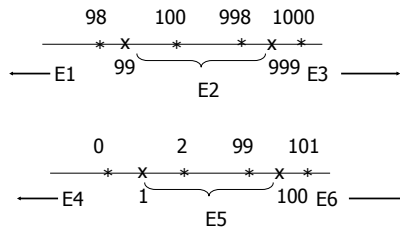
Equivalence classes for qty:

- E4: Values less than 1.
- E5: Values in the range.
- E6: Values greater than 100.

© Aditya P. Mathur 2006

60

BVA: Example: 2. Identify boundaries



Equivalence classes and boundaries for `findPrice`. Boundaries are indicated with an x. Points near the boundary are marked *.

© Aditya P. Mathur 2006

61

BVA: Example: 3. Construct test set

Test selection based on the boundary value analysis technique requires that tests must include, for each variable, values at and around the boundary. Consider the following test set:

$$T = \{ \begin{array}{l} t1: (\text{code}=98, \text{qty}=0), \\ t2: (\text{code}=99, \text{qty}=1), \\ t3: (\text{code}=100, \text{qty}=2), \\ t4: (\text{code}=998, \text{qty}=99), \\ t5: (\text{code}=999, \text{qty}=100), \\ t6: (\text{code}=1000, \text{qty}=101) \end{array} \}$$

© Aditya P. Mathur 2006

62

Testing predicates



63

Where do predicates arise?

Predicates arise from requirements in a variety of applications. Here is an example from Paradkar, Tai, and Vouk, "Specification based testing using cause-effect graphs, *Annals of Software Engineering*," V 4, pp 133-157, 1997.

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

© Aditya P. Mathur 2006

64

Boiler shutdown conditions

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)

Boiler in degraded mode
when either is true.

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.

© Aditya P. Mathur 2006

65

Boiler shutdown conditions

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**.

© Aditya P. Mathur 2006

66

Another example

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.

$p_f: (\text{printerstatus}=\text{ON}) \wedge (\text{printertray} \neq \text{empty})$

© Aditya P. Mathur 2006

67

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.

© Aditya P. Mathur 2006

68

Predicates

Relational operators (relop): $\{<, \leq, >, \geq, =, \neq\}$

= and == are equivalent.

Boolean operators (bop): $\{\!, \wedge, \vee, \text{xor}\}$ also known as
 $\{\text{not, AND, OR, XOR}\}$.

Relational expression: $e1 \text{ relop } e2$. (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**. ($\text{gender} = \text{"female"} \wedge \text{age} > 65$)

© Aditya P. Mathur 2006

69

Boolean expressions

Boolean expression: one or more Boolean variables joined
by **bop**. ($a \wedge b \vee !c$)

a , b , and c are also known as **literals**. Negation is also denoted by
placing a bar over a Boolean expression such as in $\overline{a \wedge b}$. We
also write \overline{ab} for $\overline{a \wedge b}$ and $\overline{a+b}$ for $\overline{a \vee b}$ when there is no
confusion.

Singular Boolean expression: When each literal appears
only once, e.g. ($a \wedge b \vee !c$)

© Aditya P. Mathur 2006

70

Boolean expressions (contd.)

Disjunctive normal form (DNF): Sum of products:
e.g. $(p \wedge q) \vee (r \wedge s) \vee (a \wedge c)$.

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

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

e.g., CNF: $(p \vee !r)(p \vee s)(q \vee !r)(q \vee s)$ is equivalent to DNF: $(pq \vee !rs)$

© Aditya P. Mathur 2006

71

Boolean expressions (contd.)

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

If expression E contains components e_1, e_2, \dots
then e_i is considered a **singular component** only if
it is singular &
it is mutually singular with the remaining elements of E .

© Aditya P. Mathur 2006

72

Fault model for predicate testing

What faults are we targeting when testing for the correct implementation of predicates?

Boolean operator fault: Suppose that the specification of a software module requires that an action be performed when the condition $(a < b) \vee (c > d) \wedge e$ is true.

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

© Aditya P. Mathur 2006

73

Boolean operator faults

Correct predicate: $(a < b) \vee (c > d) \wedge e$

$(a < b) \wedge (c > d) \wedge e$

Incorrect Boolean operator

$(a < b) \vee \neg (c > d) \wedge e$

Incorrect negation operator

$(a < b) \wedge (c > d) \vee e$

Incorrect Boolean operators
(multiple faults).

© Aditya P. Mathur 2006

74

Relational operator faults

Correct predicate: $(a < b) \vee (c > d) \wedge e$

$(a == b) \vee (c > d) \wedge e$

Incorrect relational operator

$(a == b) \vee (c \leq d) \wedge e$

Two relational operator faults

$(a == b) \vee (c > d) \vee e$

Incorrect Boolean & relational
operators

© Aditya P. Mathur 2006

75

Goal of predicate testing

Given a *correct predicate* p_c , the goal of predicate testing is to generate a **test set T** such that there is **at least one test case** $t \in T$ for which p_c and its **faulty version** p_f , **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.

© Aditya P. Mathur 2006

76

Goal of predicate testing (contd.)

As an example, suppose that $p_c: a < b + c$ and $p_i: a > b + c$. Consider a test set $T = \{t1, t2\}$ where $t1: \langle a=0, b=0, c=0 \rangle$ and $t2: \langle a=0, b=1, c=1 \rangle$.

The fault in p_i is **not revealed** by $t1$ as both p_c and p_i evaluate to false when evaluated against $t1$.

However, the fault **is revealed** by $t2$ as p_c evaluates to true and p_i to false when evaluated against $t2$.

© Aditya P. Mathur 2006

77

Missing or extra Boolean variable faults

Correct predicate: $a \vee b$

Missing Boolean variable fault: a

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

© Aditya P. Mathur 2006

78

Predicate constraints: BR symbols

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

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**.

© Aditya P. Mathur 2006

79

Infeasible constraints

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

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

© Aditya P. Mathur 2006

80

Predicate constraints

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

A **predicate constraint** C for predicate p_r is a **sequence of $(n + 1)$ BR symbols**, one for each Boolean variable or relational expression in p_r .

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

© Aditya P. Mathur 2006

81

True and false constraints

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

C is referred to as a **true constraint** when $p_r(C)$ is true and a **false constraint** otherwise.

A set of constraints S is partitioned into subsets S^t and S^f , such that for each C in S^t , $p_r(C) = \text{true}$, and for each C in S^f , $p_r(C) = \text{false}$.
 $S = S^t \cup S^f$.

© Aditya P. Mathur 2006

82

Predicate constraints: Example

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

$\langle b = \text{true}, r = 1, s = 1, u = 1, v = 0 \rangle$

The following test case **does not** satisfy C for p_r .

$\langle b = \text{true}, r = 1, s = 2, u = 1, v = 2 \rangle$

© Aditya P. Mathur 2006

83

Predicate testing: criteria

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

- T is **minimal** and
- T **guarantees the detection of any fault** in implementation of p_r ;

We will discuss two such criteria named **BOR** and **BRO**.

© Aditya P. Mathur 2006

84

Predicate testing: BOR testing criterion

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

T is referred to as a *BOR-adequate test set* and sometimes written as T_{BOR} .

Predicate testing: BRO testing criterion

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

T is referred to as a *BRO-adequate test set* and sometimes written as T_{BRO} .

Predicate testing: guaranteeing fault detection

Let T_x , $x \in \{BOR, BRO\}$, be a test set derived from predicate p_r .

Let p_f be another predicate obtained from p_r by *injecting* single or multiple *faults*: Boolean operator fault and relational operator fault.

T_x is said to guarantee the detection of faults in p_f if for some $t \in T_x$, $p_r(t) \neq p_f(t)$.

Guaranteeing fault detection: example

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

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

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

t1: $\langle a=1, b=2, c=1, d=0 \rangle$; Satisfies (t, t), i.e. $a < b$ is true and $c > d$ is also true.

t2: $\langle a=1, b=2, c=1, d=2 \rangle$; Satisfies (t, f)

t3: $\langle a=1, b=0, c=1, d=0 \rangle$; Satisfies (f, t)

Guaranteeing fault detection

Generate single Boolean operator faults in $p_r: a < b \wedge c > d$ and show that T guarantees the detection of each fault.

Example:

$a < b \wedge c > d$ $a < b \wedge !c > d$
 $t, f, f \leftarrow$ outcomes of test cases $\rightarrow f, t, f$

See Table 2.6. page 158

© Aditya P. Mathur 2006

89

Algorithms for generating BOR and BRO 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:

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

Note that $A \otimes B$ is a minimal set.

© Aditya P. Mathur 2006

90

Set products: Example

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

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

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

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

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

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

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

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

© Aditya P. Mathur 2006

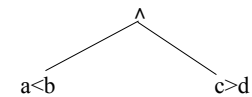
91

Generation of BOR constraint set

See page 160 for a formal algorithm. An illustration follows.

We want to generate T_{BOR} for: $p_r: a < b \wedge c > d$

First, generate syntax tree of p_r .



© Aditya P. Mathur 2006

92

Generation of the BOR constraint set

We will use the following notation:

S_N is the constraint set for node N in the syntax tree for p_r .

S_N^t is the true constraint set for node N in the syntax tree for p_r .

S_N^f is the false constraint set for node N in the syntax tree for p_r .

$S_N = S_N^t \cup S_N^f$.

© Aditya P. Mathur 2006 93

Generation of the BOR constraint set (contd.)

Second, label each leaf node with the constraint set $\{(t), (f)\}$. We label the nodes as N_1, N_2 , and so on for convenience.

© Aditya P. Mathur 2006 94

Generation of the BOR constraint set (contd.)

Third, compute the constraint set for the next higher node in the syntax tree (i.e., constraint set for N_3 from those of its descendants). Three possibilities for N_3 : AND; OR; and NOT.

<p>AND:</p> $S_{N_3}^t = S_{N_1}^t \otimes S_{N_2}^t$ $S_{N_3}^f = (S_{N_1}^f \times \{t_2\}) \cup (\{t_1\} \times S_{N_2}^f)$	<p>OR:</p> $S_{N_3}^f = S_{N_1}^f \otimes S_{N_2}^f$ $S_{N_3}^t = (S_{N_1}^t \times \{f_2\}) \cup (\{f_1\} \times S_{N_2}^t)$
<p>NOT:</p> $S_{N_3}^f = S_{N_1}^t$ $S_{N_3}^t = S_{N_1}^f$	

© Aditya P. Mathur 2006 95

Generation of the BOR constraint set (contd.)

$$S_{N_3}^t = S_{N_1}^t \otimes S_{N_2}^t = \{(t)\} \otimes \{(t)\} = \{(t, t)\}$$

$$S_{N_3}^f = (S_{N_1}^f \times \{t_2\}) \cup (\{t_1\} \times S_{N_2}^f)$$

$$= (\{(f)\} \times \{(t)\}) \cup (\{(t)\} \times \{(f)\})$$

$$= \{(f, t)\} \cup \{(t, f)\}$$

$$= \{(f, t), \{(t, f)\}$$

© Aditya P. Mathur 2006 96

Generation of T_{BOR}

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

S_{N_3} 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.

$T_{BOR} = \{t_1, t_2, t_3\}$
 $t_1 = \langle a=1, b=2, c=6, d=5 \rangle (t, t)$
 $t_2 = \langle a=1, b=0, c=6, d=5 \rangle (f, t)$
 $t_3 = \langle a=1, b=2, c=1, d=2 \rangle (t, f)$

$S_{N_3} = \{(t, t), (f, t), (t, f)\}$

© Aditya P. Mathur 2006 97

Generation of BRO constraint set

See page 163 for a formal algorithm. An illustration follows.

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

© Aditya P. Mathur 2006 98

BRO constraint set

The BRO constraint set S for relational expression $e_1 \text{ relop } e_2$:
 $S = \{(>), (=), (<)\}$

Separation of S into its true (S^t) and false (S^f) components:

relop: $>$	$S^t = \{(>)\}$	$S^f = \{ (=), (<) \}$
relop: \geq	$S^t = \{(>), (=)\}$	$S^f = \{ (<) \}$
relop: $=$	$S^t = \{ (=) \}$	$S^f = \{ (<), (>) \}$
relop: $<$	$S^t = \{ (<) \}$	$S^f = \{ (=), (>) \}$
relop: \leq	$S^t = \{ (<), (=) \}$	$S^f = \{ (>) \}$

Note: t_N denotes an element of S_N^t , f_N denotes an element of S_N^f

© Aditya P. Mathur 2006 99

BRO constraint set: Example

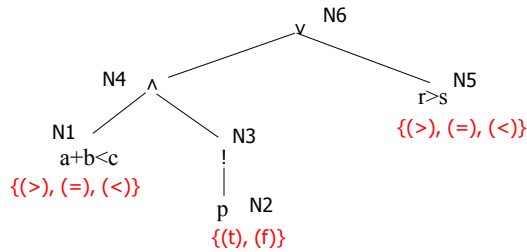
$p_r: (a+b < c) \wedge !p \vee (r > s)$

Step 1: Construct the AST for the given predicate.

© Aditya P. Mathur 2006 100

BRO constraint set: Example (contd.)

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



© Aditya P. Mathur 2006

101

BRO constraint set: Example (contd.)

Step 2: Traverse the tree and compute constraint set for each internal node.

$$S_{N3}^f = S_{N2}^f = \{(f)\} \quad S_{N3}^t = S_{N2}^t = \{(t)\}$$

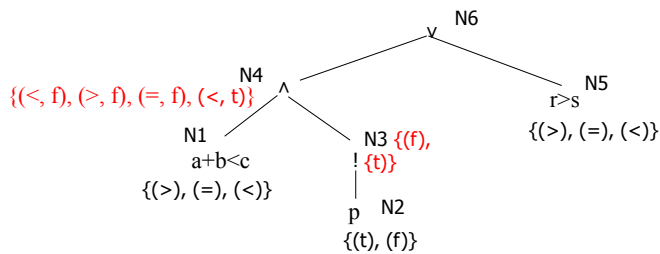
$$S_{N4}^t = S_{N1}^t \otimes S_{N3}^t = \{(<)\} \otimes \{(f)\} = \{(<, f)\}$$

$$\begin{aligned} S_{N4}^f &= (S_{N1}^f \times \{(t_{N3})\}) \cup (\{(t_{N1})\} \times S_{N3}^f) \\ &= (\{(>, =)\} \times \{(f)\}) \cup \{(<)\} \times \{(t)\} \\ &= \{(>, f), (=, f)\} \cup \{(<, t)\} \\ &= \{(>, f), (=, f), (<, t)\} \end{aligned}$$

© Aditya P. Mathur 2006

102

BRO constraint set: Example (contd.)



© Aditya P. Mathur 2006

103

BRO constraint set: Example (contd.)

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

$$\begin{aligned} S_{N6}^f &= S_{N4}^f \otimes S_{N5}^f \\ &= \{(>, f), (=, f), (<, t)\} \otimes \{(<)\} \\ &= \{(>, f, <), (=, f, <), (<, t, <)\} \end{aligned}$$

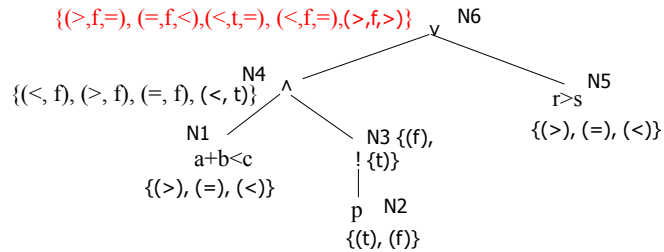
$$\begin{aligned} S_{N6}^t &= (S_{N4}^t \times \{(f_{N5})\}) \cup (\{(f_{N4})\} \times S_{N5}^t) \\ &= (\{(<, f)\} \times \{(=)\}) \cup \{(>, f)\} \times \{(>)\} \\ &= \{(<, f, =)\} \cup \{(>, f, >)\} \\ &= \{(<, f, =), (>, f, >)\} \end{aligned}$$

© Aditya P. Mathur 2006

104

BRO constraint set: Example (contd.)

Constraint set for $p_r: (a+b < c) \wedge !p \vee (r > s)$



© Aditya P. Mathur 2006

105

BRO constraint set

Given the constraint set for $p_r: (a+b < c) \wedge !p \vee (r > s)$, construct T_{BRO} .

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

$\langle a=1, b=1, c=1, p=true, r=1, s=1 \rangle$
 $\langle a=1, b=0, c=1, p=true, r=1, s=2 \rangle$
 $\langle a=1, b=1, c=3, p=false, r=1, s=1 \rangle$
 $\langle a=0, b=2, c=3, p=true, r=0, s=0 \rangle$
 $\langle a=1, b=1, c=0, p=true, r=2, s=0 \rangle$

© Aditya P. Mathur 2006

106

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.

© Aditya P. Mathur 2006

107

Non-singular expressions and DNF: Examples

Predicate (p_r)	DNF	Mutually singular components in p_r
$ab(b+c)$	$abb+abc$	$a; b(b+c)$
$a(bc+bd)$	$abc+abd$	$a; (bc+bd)$
$a(bc+!b+de)$	$abc+a!b+ade$	$a; bc+!b; de$

© Aditya P. Mathur 2006

108

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**.

© Aditya P. Mathur 2006

109

Meaning Impact (MI) procedure

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

The MI procedure is on pages 168-169. We illustrate it with an example.

© Aditya P. Mathur 2006

110

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 $r < s$.

© Aditya P. Mathur 2006

111

MI procedure: Example (contd.)

Step 0: Express E in DNF notation. Clearly, we can write $E=e_1+e_2$, where $e_1=abc$ and $e_2=a!bd$.

Step 1: Construct a constraint set T_{e_1} for e_1 that makes e_1 true. Similarly construct T_{e_2} for e_2 that makes e_2 true.

$$T_{e_1} = \{(t,t,t,t), (t,t,t,f)\} \quad T_{e_2} = \{(t,f,t,t), (t,f,f,t)\}$$

Note that the four t 's in the first element of T_{e_1} denote the values of the Boolean variables a , b , c , and d , respectively. The second element, and others, are to be interpreted similarly.

© Aditya P. Mathur 2006

112

MI procedure: Example (contd.)

Step 2: From each T_{ei} , remove the constraints that are in any other T_{ej} . This gives us TS_{ei} and TS_{ej} . Note that this step will lead $TS_{ei} \cap TS_{ej} = \emptyset$.

There are no common constraints between T_{e1} and T_{e2} in our example. Hence we get:

$$TS_{e1} = \{(t,t,t,t), (t,t,t,f)\} \quad TS_{e2} = \{(t,f,t,t), (t,f,f,t)\}$$

© Aditya P. Mathur 2006 113

MI procedure: Example (contd.)

Step 3: Construct S^E by selecting one element from each TS_e .

$$S^E = \{(t,t,t,f), (t,f,f,t)\}$$

Note that for each constraint x in S^E we get $E(x)=\text{true}$. Also, S^E is minimal.

© Aditya P. Mathur 2006 114

MI procedure: Example (contd.)

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

$$e^1_1 = !abc \quad e^2_1 = a!bc \quad e^3_1 = ab!c$$

$$e^1_2 = !a!bd \quad e^2_2 = abd \quad e^3_2 = a!b!d$$

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

© Aditya P. Mathur 2006 115

MI procedure: Example (contd.)

$$F_{e^1_1} = \{(f,t,t,t), (f,t,t,f)\}$$

$$F_{e^2_1} = \{(t,f,t,t), (t,f,t,f)\}$$

$$F_{e^3_1} = \{(t,t,f,t), (t,t,f,f)\}$$

$$F_{e^1_2} = \{(f,f,t,t), (f,f,f,t)\}$$

$$F_{e^2_2} = \{(t,t,t,t), (t,t,f,t)\}$$

$$F_{e^3_2} = \{(t,f,t,f), (t,f,f,f)\}$$

© Aditya P. Mathur 2006 116

MI procedure: Example (contd.)

Step 5: Now construct FS_e by removing from F_e any constraint that appeared in any of the two sets T_e constructed earlier.

$$\begin{aligned}
 FSe^1_1 &= Fe^1_1 \\
 FSe^2_1 &= \{(t,f,t,f)\} \\
 FSe^3_1 &= Fe^1_3 \\
 \\
 FSe^1_2 &= Fe^1_2 \\
 FSe^2_2 &= \{(t,t,f,t)\} \\
 FSe^3_2 &= Fe^1_3
 \end{aligned}$$

Constraints common with T_{e1} and T_{e2} are removed.

© Aditya P. Mathur 2006

117

MI procedure: Example (contd.)

Step 6: Now construct S^f_E by selecting one constraint from each FS_e

$$S^f_E = \{(f,t,t,f), (t,f,t,f), (t,t,f,t), (f,f,t,t)\}$$

Step 7: Now construct $S_E = S^t_E \cup S^f_E$

$$S_E = \{(t,t,t,t), (t,f,f,f), (f,t,t,f), (t,f,t,f), (t,t,f,t), (f,f,t,t)\}$$

Note: Each constraint in S^t_E makes E true and each constraint in S^f_E makes E false. Check it out!

We are now done with the MI procedure.

© Aditya P. Mathur 2006

118

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 entire procedure is described on page 171. We illustrate it with an example.

© Aditya P. Mathur 2006

119

BOR-MI-CSET: Example

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

Mutually singular components of E :

$$\begin{aligned}
 e1 &= a \\
 e2 &= bc + !bd
 \end{aligned}$$

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

© Aditya P. Mathur 2006

120

BOR-MI-CSET: Example (contd.)

Summary:

$S_{e1}^t = \{(t)\}$ $S_{e1}^f = \{(f)\}$ from BOR-CSET procedure.

$S_{e2}^t = \{(t,t,f), (f, t, t)\}$ $S_{e2}^f = \{(f,t,f), (t,f,t)\}$ from MI-CSET procedure.

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

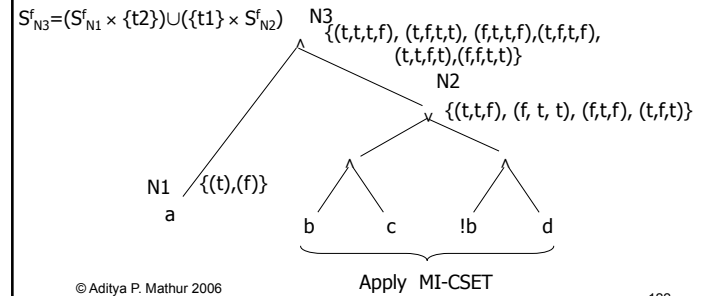
© Aditya P. Mathur 2006

121

BOR-MI-CSET: Example (contd.)

$$S_{N3}^t = S_{N1}^t \otimes S_{N22}^t$$

Obtained by applying Step 2 of BOR-CSET to an AND node.



© Aditya P. Mathur 2006

122

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 .

© Aditya P. Mathur 2006

123

Summary (contd.)

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.

© Aditya P. Mathur 2006

124



Summary (contd.)

Usually one would combine equivalence partitioning, boundary value analysis, and predicate testing procedures to generate tests for a requirement of the following type:

