



# Test Generation – Predicate Analysis

Andrea Polini

Advanced Topics on Software Engineering – Software Testing  
MSc in Computer Science  
University of Camerino

# Dependency modeling

## CEG aka Dependency Modeling

The very general idea is to make explicit, also **through a graphical representation**, the relation among input conditions (**causes**) and output conditions (**effects**) and to exploit such relations for testing purposes.

In any case the relation can be fruitfully represented by a **boolean expression**

## Cause and effects

A **cause** is any condition in the requirements that may effect the program output. An **effect** is the response of the program to some combination of input conditions. An effect is **not necessarily visible** to the external user, while it can be retrieved introducing suitable probes (**test points**)

## Exercise

The LED close to the product description should be switched on when the credit becomes greater then the price of the snack

# Dependency modeling

## CEG aka Dependency Modeling

The very general idea is to make explicit, also **through a graphical representation**, the relation among input conditions (**causes**) and output conditions (**effects**) and to exploit such relations for testing purposes.

In any case the relation can be fruitfully represented by a **boolean expression**

## Cause and effects

A **cause** is any condition in the requirements that may effect the program output. An **effect** is the response of the program to some combination of input conditions. An effect is **not necessarily visible** to the external user, while it can be retrieved introducing suitable probes (**test points**)

## Exercise

The LED close to the product description should be switched on when the credit becomes greater then the price of the snack

# Dependency modeling

## CEG aka Dependency Modeling

The very general idea is to make explicit, also **through a graphical representation**, the relation among input conditions (**causes**) and output conditions (**effects**) and to exploit such relations for testing purposes.

In any case the relation can be fruitfully represented by a **boolean expression**

## Cause and effects

A **cause** is any condition in the requirements that may effect the program output. An **effect** is the response of the program to some combination of input conditions. An effect is **not necessarily visible** to the external user, while it can be retrieved introducing suitable probes (**test points**)

## Exercise

The LED close to the product description should be switched on when the credit becomes greater then the price of the snack

# Test generation from CEG

## CEG and test generation

- ▶ Identify cause and effects **reading the requirements**. Assign a **unique identifier**
- ▶ Express the relationship between causes and effects **using a CEG**
- ▶ Transform the **CEG into a decision table**
- ▶ Generate **tests from the decision table**

## Are we checking iff relations?

The strategy reported above checks that if the conditions happens the effect should be visible. In general effects could be produces as consequence of additional behaviour so it may possible that you are not interested in checking the other way arund i.e. **an effect is present only if those conditions holds**

# Test generation from CEG

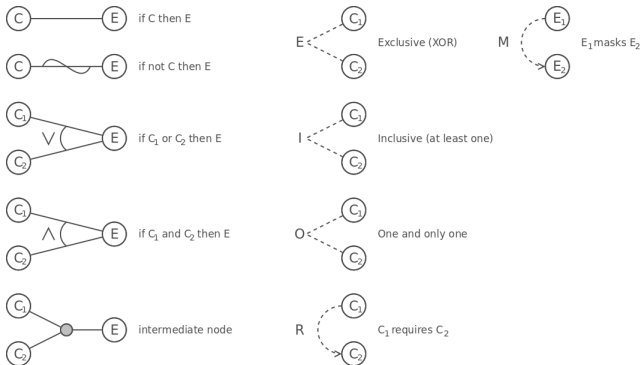
## CEG and test generation

- ▶ Identify cause and effects **reading the requirements**. Assign a **unique identifier**
- ▶ Express the relationship between causes and effects **using a CEG**
- ▶ Transform the **CEG into a decision table**
- ▶ Generate **tests from the decision table**

## Are we checking iff relations?

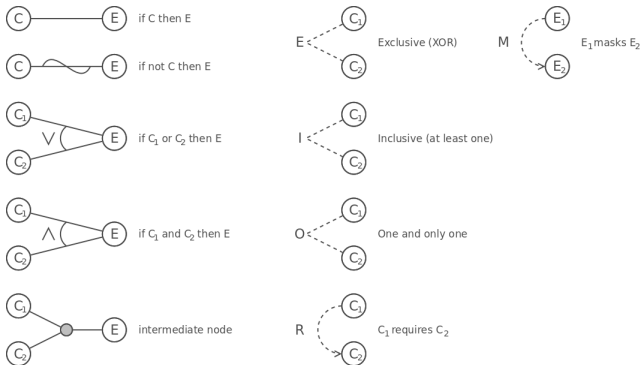
The strategy reported above checks that if the conditions happens the effect should be visible. In general effects could be produces as consequence of additional behaviour so it may possible that you are not interested in checking the other way arund i.e. **an effect is present only if those conditions holds**

# CEG Notation



- The mask relation can be expressed as additional relations on the causes
- What if I would like to check iff relations among causes and effects?

# CEG Notation



- The mask relation can be expressed as additional relations on the causes
- What if I would like to check iff relations among causes and effects?



# Creating a CFG

## Process

To create a CFG follow the process below:

- ▶ carefully identify causes and effects from a thoughtful **analysis of the requirements**.
- ▶ assign to each cause and each effect a **unique identifier**
- ▶ represent the **identified relations** in a CFG

# Example

## Computer purchase system

A web based company sells computers (CPU), printers (PR), monitors (M), and additional memories (RAM). The Web GUI will include 4 windows for driving the buyer through the selection process. A final window with the free giveaway items is then displayed. For simplicity only one item per category can be purchased.

**Conditions:** For each order the buyer may select from 3 CPU, 2 PR, 3M. RAM is available only as a “free” upgrade. M20 and M23 any CPU or as stand alone. M30 only with CPU 3. PR 1 is available free with CPU 2 or 3. M and PR can be purchased as stand alone. Not M30. CPU 1 gets RAM 256 upgrade. CPU 2 o 3 gets RAM 512 upgrade. RAM 1G upgrade and free PR2 available if CPU 3 purchased with M30. There is a window to make selection with menus in particular a widget displaying the free item available and a “Price” widget reports the calculation related to prices. The strategy asks you to:

- ▶ Read carefully the requirements
- ▶ Identify causes and effects
- ▶ Identify relations among them

# Example - The CFG for the Computer Purchase System

Construct the CFG for the Computer Purchase System

- ▶ causes?
- ▶ effects?
- ▶ relations?
- ▶ Do we need to consider relations as iff?

# Decision Tables from a CEG

CEG models relations among different aspects of the system. The derivation of test requires the definition of the corresponding **decision table**

## Decision tables

For each **cause and effect** use a row and **put test as columns** of the matrix. Each entry in the decision table is a 0 or a 1 depending on whether or not **the corresponding condition is false or true**, respectively.

# How to derive a DT

**Input:** A CEG containing causes  $C_1, C_2, \dots, C_p$  and effects  $Ef_1, Ef_2, \dots, Ef_q$

**Output:** A decision table containing  $p + q$  rows and  $M$  columns where  $M$  depends on relationship between causes and effects.

## Procedure CFG2DT

**Step1:** Initialize DT to an empty DT

**Step2:** Execute the following steps for  $i=1$  to  $q$

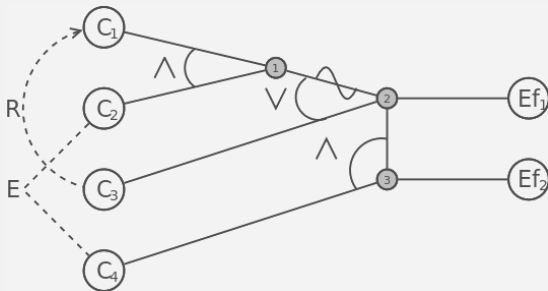
**2.1** Select the next effect  $e$

**2.2** Find combinations of conditions that cause  $e$  to be present and store the  $m$  generated vector. Avoid combinatorial explosion.

**2.3** update the decision table adding the generated vectors

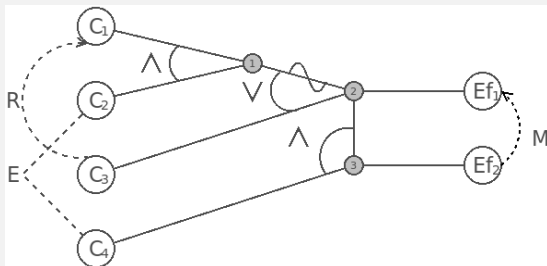
# DT derivation

Consider the following CEG and derive the corresponding decision table:



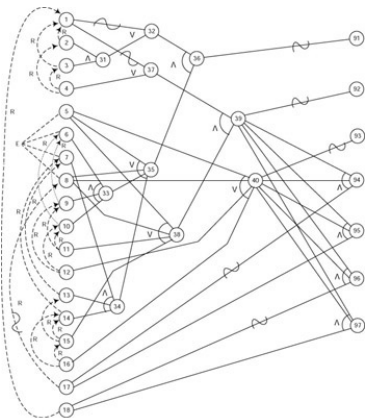
# DT derivation

How can we deal with Mask relations?



# Example

Apply the procedure to the following CEG:

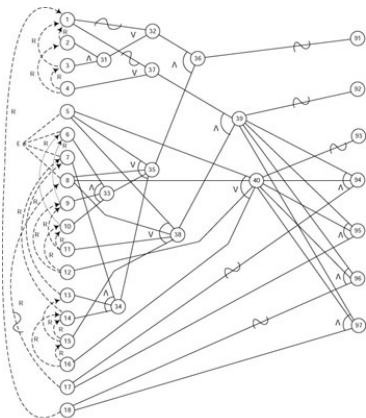


You need to automatize the process. In your opinion which should be the main component included in a supporting tool?



# Example

Apply the procedure to the following CEG:



You need to automatize the process. In your opinion which should be the main component included in a supporting tool?

# Heuristic to avoid combinatorial explosion

The described approach could lead to **exponential generation** on the number of tests with respect to causes. Indeed having an effect depending on  $n$  causes can lead to the generation of a **number of vectors in the order of  $2^n$**

## Reduction strategies

- ▶ For **or** relations: enumerate just those situations in which two causes are both false (0) or one of them true (1)
- ▶ For **and** relations: enumerate those situations for which causes assume different values (0), and those in which all of them are true (1)

# Test generation

## Tests from a decision table

Each **column** of the decision table constitutes the source for generating tests. Consider that each condition could be satisfied by **more assignment** to the variable leading to the generation of more than one test for each column.

To derive a test case you need then to **identify the inputs correlated to the cause** and the **value that makes the cause valid**. On such sets you can apply previously considered approaches. Let's go back to the LED example.

# Test generation from predicates

Techniques aiming at finding bugs in the **coding of conditions**

## Predicate testing

*if the printer is ON and has paper then send the document for printing*

```
pr: (printer_status=ON) ^ printer_tray!=empty
```

We are interested in generating test cases from predicates such that any fault belonging to a class is detected.

Consider the following predicate:

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

The following test:

$$t = (a = 1, b = 2, c = 4, d = 2, e = true)$$

results in

$$p(t) = true$$

# Fault model

Which kind of faults are generally targetted:

- **Boolean operator fault**
  - incorrect boolean operator used
  - negation missing or placed incorrectly
  - parentheses are incorrect
  - incorrect Boolean variable used
- **relational operator fault**
  - incorrect relational operator is used
- **arithmetic expression fault**
  - arithmetic expression is off by an amount equal to  $\epsilon$   
(off-by- $\epsilon$ , off-by- $\epsilon^+$ , off-by- $\epsilon^*$ )

## Objective of predicate testing

To generate a test set  $\mathcal{T}$  such that there is at least one test cast  $t \in \mathcal{T}$  for which  $p_c$  and its faulty version  $p_f$  are distinguishable

# Fault model

Which kind of faults are generally targetted:

- **Boolean operator fault**
  - incorrect boolean operator used
  - negation missing or placed incorrectly
  - parentheses are incorrect
  - incorrect Boolean variable used
- **relational operator fault**
  - incorrect relational operator is used
- **arithmetic expression fault**
  - arithmetic expression is off by an amount equal to  $\epsilon$   
(off-by- $\epsilon$ , off-by- $\epsilon^+$ , off-by- $\epsilon^*$ )

## Objective of predicate testing

To generate a test set  $\mathcal{T}$  such that there is at least one test cast  $t \in \mathcal{T}$  for which  $p_c$  and its faulty version  $p_i$  are distinguishable

## Missing or extra boolean faults

Missing or extra boolean faults corresponds to situation in which the programmer **forgot to include** a variable in a condition or **added a not needed** variable

The following approaches do not provide any guarantee on the possible identification of such kind of faults

## Predicate constraints

Let  $BR$  denote the following set of symbols  $\{\mathbf{t}, \mathbf{f}, <, =, >, +\epsilon, -\epsilon\}$ . A  $BR$  specifies a **constraint** on a Boolean variable of a relational expression.

In particular  $+\epsilon$  and  $-\epsilon$  are constraints on an expression  $e_1 < e_2$  that can be respectively satisfied by tests such that  $0 < e_1 - e_2 \leq \epsilon$  and  $-\epsilon \leq e_1 - e_2 < 0$

A constraint is considered **infeasible** if there exist no input values for the variable in the predicate that can satisfy the constraint.



## Predicate constraints

Let  $BR$  denote the following set of symbols  $\{\mathbf{t}, \mathbf{f}, <, =, >, +\epsilon, -\epsilon\}$ . A  $BR$  specifies a **constraint** on a Boolean variable of a relational expression. In particular  $+\epsilon$  and  $-\epsilon$  are constraints on an expression  $e_1 < e_2$  that can be respectively satisfied by tests such that  $0 < e_1 - e_2 \leq \epsilon$  and  $-\epsilon \leq e_1 - e_2 < 0$

A constraint is considered **infeasible** if there exist no input values for the variable in the predicate that can satisfy the constraint.

## Predicate constraints

Let  $BR$  denote the following set of symbols  $\{\mathbf{t}, \mathbf{f}, <, =, >, +\epsilon, -\epsilon\}$ . A  $BR$  specifies a **constraint** on a Boolean variable of a relational expression. In particular  $+\epsilon$  and  $-\epsilon$  are constraints on an expression  $e_1 < e_2$  that can be respectively satisfied by tests such that  $0 < e_1 - e_2 \leq \epsilon$  and  $-\epsilon \leq e_1 - e_2 < 0$

A constraint is considered **infeasible** if there exist no input values for the variable in the predicate that can satisfy the constraint.

# Predicate testing criteria

## Three common criteria:

- ▶ **BOR (Boolean Operator)**: A test set  $\mathcal{T}$  that satisfied the BOR-testing criterion for a compound predicate  $p_r$ , guarantees the detection of single or multiple Boolean operator faults in the implementation of  $p_r$ .  $\mathcal{T}$  is referred to as a BOR-adequate test set and sometimes written as  $\mathcal{T}_{BOR}$ .
- ▶ **BRO (Boolean and relational Operator)**: A test set  $\mathcal{T}$  that satisfied 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  $p_r$ .  $\mathcal{T}$  is referred to as a BRO-adequate test set and sometimes written as  $\mathcal{T}_{BRO}$ .
- ▶ **BRE (Boolean and relational expression)**: A test set  $\mathcal{T}$  that satisfied the BRE-testing criterion for a compound predicate  $p_r$ , guarantees the detection of single or multiple Boolean operator, relational operator and arithmetic expression faults in the implementation of  $p_r$ .  $\mathcal{T}$  is referred to as a BRO-adequate test set and sometimes written as  $\mathcal{T}_{BRE}$ .

# Predicate testing criteria

## Three common criteria:

- ▶ **BOR (Boolean Operator)**: A test set  $\mathcal{T}$  that satisfied the BOR-testing criterion for a compound predicate  $p_r$ , guarantees the detection of single or multiple Boolean operator faults in the implementation of  $p_r$ .  $\mathcal{T}$  is referred to as a BOR-adequate test set and sometimes written as  $\mathcal{T}_{BOR}$ .
- ▶ **BRO (Boolean and relational Operator)**: A test set  $\mathcal{T}$  that satisfied 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  $p_r$ .  $\mathcal{T}$  is referred to as a BRO-adequate test set and sometimes written as  $\mathcal{T}_{BRO}$ .
- ▶ **BRE (Boolean and relational expression)**: A test set  $\mathcal{T}$  that satisfied the BRE-testing criterion for a compound predicate  $p_r$ , guarantees the detection of single or multiple Boolean operator, relational operator and arithmetic expression faults in the implementation of  $p_r$ .  $\mathcal{T}$  is referred to as a BRO-adequate test set and sometimes written as  $\mathcal{T}_{BRE}$ .

# Predicate testing criteria

## Three common criteria:

- ▶ **BOR (Boolean Operator)**: A test set  $\mathcal{T}$  that satisfied the BOR-testing criterion for a compound predicate  $p_r$ , guarantees the detection of single or multiple Boolean operator faults in the implementation of  $p_r$ .  $\mathcal{T}$  is referred to as a BOR-adequate test set and sometimes written as  $\mathcal{T}_{BOR}$ .
- ▶ **BRO (Boolean and relational Operator)**: A test set  $\mathcal{T}$  that satisfied 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  $p_r$ .  $\mathcal{T}$  is referred to as a BRO-adequate test set and sometimes written as  $\mathcal{T}_{BRO}$ .
- ▶ **BRE (Boolean and relational expression)**: A test set  $\mathcal{T}$  that satisfied the BRE-testing criterion for a compound predicate  $p_r$ , guarantees the detection of single or multiple Boolean operator, relational operator and arithmetic expression faults in the implementation of  $p_r$ .  $\mathcal{T}$  is referred to as a BRO-adequate test set and sometimes written as  $\mathcal{T}_{BRE}$ .

## BOR example

Let  $p_r : a < b \wedge c > d$  and  $\mathcal{S}$  constraints on  $p_r$  where  $\mathcal{S} = \{(\mathbf{t}, \mathbf{t}), (\mathbf{t}, \mathbf{f}), (\mathbf{f}, \mathbf{t})\}$  the following test set  $\mathcal{T}$  satisfies constraint set  $\mathcal{S}$  and the BOR-testing criterion:

$$\mathcal{T} = \{t_1 : \langle a = 1, b = 2, c = 1, d = 0 \rangle ; \\ t_2 : \langle a = 1, b = 2, c = 1, d = 2 \rangle ; \\ t_3 : \langle a = 1, b = 0, c = 1, d = 0 \rangle ; \\ \}$$

# BOR example

## Covered faults

To discover the covered faults lets modify the proposition introducing one or more operational fault

Predicate	t1	t2	t3
$a < b \wedge c > d$	true	false	false
$a < b \vee c > d$	true	<b>true</b>	<b>true</b>
$a < b \wedge \neg c > d$	<b>false</b>	<b>true</b>	false
$\neg a < b \wedge c > d$	<b>false</b>	false	<b>true</b>
$a < b \vee \neg c > d$	true	<b>true</b>	false
$\neg a < b \vee c > d$	true	false	<b>true</b>
$\neg a < b \wedge \neg c > d$	<b>false</b>	false	false
$\neg a < b \vee \neg c > d$	true	<b>true</b>	<b>true</b>

# Generating BOR, BRO, BRE adequate tests

A **predicate constraint**  $C$  for predicate  $p_r$  is a sequence of  $n + 1$  boolean and relational symbols.

A **test case**  $t$  **satisfies**  $C$  for predicate  $p_r$ , if each component of  $p_r$  satisfies the corresponding constraint in  $C$  when evaluated against  $t$ .  
e.g.: given  $p_r = b \wedge r < s \vee u \geq v$  and  $C : (t, =, >)$  the following test case satisfies  $C$ :  $\langle b = true, r = 1, s = 1, u = 1, v = 0 \rangle$

There exist algorithms for the generation of adequate tests given constraints on the predicate. They are based on the definition of:

- Cartesian product of sets
- *onto* set product operator
- $AST(p_r)$



# Onto Operator

## Onto Operator

Given two sets  $A$  and  $B$  the onto operator constructs the minimal set of pairs  $\langle a, b \rangle$  where  $a \in A$  and  $b \in B$  and each element of the two sets is used in at least one of the pairs in the onto set  $A \otimes B$ .

Which is the cardinality of the onto set?

Let  $A = \{t, 0, >\}$  and  $B = \{f, <\}$  lets derive the cartesian product and some examples of onto product sets

# Abstract Syntax Tree for a Predicate p

## AST

The abstract syntax tree provides a tree based representation of a predicate that captures the syntactic relations among the predicate constituents, and that is typically useful for associating meaning to the predicate itself.

Leaves of the tree are atomic propositions while nodes are boolean operators

## AST

Let's build the AST for the proposition:

$$a < b \vee q \wedge \neg p \vee (a == c \wedge p)$$

# Generating the BOR-constraint set

Let  $p_r$  be a predicate and  $AST(P_r)$  its abstract syntax tree,  $S_N$  the constraint set attached to a node  $N$  (where  $S_N^t$  and  $S_N^f$  are the true and false constraints associated with the node). The following alg. generates the BOR-constraint set for  $p_r$

**Input:**  $AST(p_r)$  (only singular expressions)

**Output:** BOR-Constraint set attached to the root node

- 1 Label each leaf node  $N$  of  $AST(p_r)$  with its constraint set  $S_N = \{t, f\}$
- 2 Visit the  $AST$  bottom-up. Let  $N_1$  and  $N_2$  direct descendants of node  $N$  and  $S_{N_1}$  and  $S_{N_2}$  the corresponding BOR-constraint set.  $S_N$  is computed as follows:

## 2.1 $N$ is an OR-node:

- $S_N^f = S_{N_1}^f \otimes S_{N_2}^f$
- $S_N^t = (S_{N_1}^t \times \{f_2\}) \cup (\{f_1\} \times S_{N_2}^t)$  where  $f_1 \in S_{N_1}^f$  and  $f_2 \in S_{N_2}^f$

## 2.2 $N$ is an AND-node:

- $S_N^t = S_{N_1}^t \otimes S_{N_2}^t$
- $S_N^f = (S_{N_1}^f \times \{t_2\}) \cup (\{t_1\} \times S_{N_2}^f)$  where  $t_1 \in S_{N_1}^t$  and  $t_2 \in S_{N_2}^t$

## 2.3 $N$ is NOT-node:

- $S_N^t = S_{N_1}^f$
- $S_N^f = S_{N_1}^t$

## BOR-constraint set example

Let's apply the BOR-constraint procedure to:

$$\blacktriangleright (a + b < c) \wedge \neg p \vee (r > s)$$

# Generating the BRO-constraint set

**Input:**  $AST(p_r)$  (only singular expressions)

**Output:** BRO-Constraint set attached to the root node

- 1 Label each leaf node  $N$  of  $AST(p_r)$  with its constraint set  $S_N$ . For each leaf node that represents a Boolean variable  $S_N = \{t, f\}$ . For each leaf node that is a relational expression  $S_N = \{(>), (=), (<)\}$ .
- 2 Visit the  $AST$  bottom-up. Let  $N_1$  and  $N_2$  direct descendants of node  $N$  and  $S_{N_1}$  and  $S_{N_2}$  the corresponding BRO-constraint set.  $S_N$  is computed as done for the BOR procedure.

## Generating the BRO-constraint set

Let's apply the BRO-constraint procedure to:

$$\blacktriangleright (a + b < c) \wedge \neg p \vee (r > s)$$

# BRE constraint sets

Constraint	Satisfying condition
$+\epsilon$	$0 < e_1 - e_2 \leq +\epsilon$
$-\epsilon$	$-\epsilon \leq e_1 - e_2 < 0$

True and false components for a relational operator are defined as follows:

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

# Generating the BRE-constraint set

**Input:**  $AST(p_r)$  (only singular expressions)

**Output:** BRE-Constraint set attached to the root node

- 1 Label each leaf node  $N$  of  $AST(p_r)$  with its constraint set  $S_N$ . For each leaf node that represents a Boolean variable  $S_N = \{t, f\}$ . For each leaf node that is a relational expression  $S_N = \{(+\epsilon), (=), (-\epsilon)\}$ .
- 2 Visit the  $AST$  bottom-up. Let  $N_1$  and  $N_2$  direct descendants of node  $N$  and  $S_{N_1}$  and  $S_{N_2}$  the corresponding BRE-constraint set.  $S_N$  is computed as done for the BOR procedure.

Which are the relations among the test suites generated by the different methods?



# Generating the BRE-constraint set

**Input:**  $AST(p_r)$  (only singular expressions)

**Output:** BRE-Constraint set attached to the root node

- 1 Label each leaf node  $N$  of  $AST(p_r)$  with its constraint set  $S_N$ . For each leaf node that represents a Boolean variable  $S_N = \{t, f\}$ . For each leaf node that is a relational expression  $S_N = \{(+\epsilon), (=), (-\epsilon)\}$ .
- 2 Visit the  $AST$  bottom-up. Let  $N_1$  and  $N_2$  direct descendants of node  $N$  and  $S_{N_1}$  and  $S_{N_2}$  the corresponding BRE-constraint set.  $S_N$  is computed as done for the BOR procedure.

Which are the relations among the test suites generated by the different methods?

## Generating the BRE-constraint set

Let's apply the BRE-constraint procedure to:

$$\blacktriangleright (a + b < c) \wedge \neg p \vee (r > s)$$

# Generating test sets for non singular expressions

## MI-CSET Procedure

**Input:** A boolean expression  $E = e_1 + e_2 + \dots + e_n$  in minimal DNF containing  $n$  terms. Term  $e_i$  contains  $l_i > 0$  literals.

**Output:** A set of constraints  $S_E$  that guarantees the detection of missing or extra NOT operator fault in a faulty version of  $E$ .

- 1 For each term  $e_i$ ,  $1 \leq i \leq n$  construct  $T_{e_i}$  as the set of constraints making  $e_i$  true
- 2 Let  $TS_{e_i} = T_{e_i} - \bigcup_{j=1, j \neq i}^n T_{e_j}$ . Note that for  $i \neq j$ ,  $TS_{e_i} \cap TS_{e_j} = \emptyset$
- 3 Construct  $S_E^t$  by including one constraint from each  $TS_{e_i}$
- 4 Let  $e_i^j$  denotes the term obtained by complementing the  $j^{th}$  literal in  $e_i$ , for  $1 \leq i \leq n$  and  $1 \leq j \leq l_i$ . Construct  $F_{e_i^j}$  as the set of constraints that make  $e_i^j$  true
- 5 Let  $FS_{e_i^j} = F_{e_i^j} - \bigcup_{k=1}^n T_{e_k}$ . Thus for any constraint  $c \in FS_{e_i^j}$ ,  $E(c) = \text{false}$
- 6 Construct  $S_E^f$  that is minimal and covers each  $FS_{e_i^j}$  a least once.
- 7 Construct the desired constraint set for  $E$  as  $S_E = S_E^t \cup S_E^f$ .

# MI-CSET

Let's apply the procedure to:  $a \wedge ((b \wedge c) \vee (\neg b \wedge d))$

# Generating test sets for non singular expressions

## BOR-MI-CSET Procedure

**Input:** A boolean expression  $E$

**Output:** A set of constraints  $S_E$  that guarantees the detection of boolean operator faults in  $E$

- 1 Partition  $E$  in a set of  $n$  mutually singular components  $E = \{E_1, E_2, \dots, E_n\}$
- 2 Generate the BOR constraint set for each singular component in  $E$  using the BOR-CSET procedure.
- 3 Generate the MI-constraint set for each non-singular component in  $E$  using the MI-CSET procedure
- 4 Combine the constraints generated in the previous two steps as indicated in step 2 of the BOR-CSET procedure. The result of the combination is the constraint set for  $E$

## BOR-MI-CSET procedure - an example

Let's apply the procedure to:  $a \wedge ((b \wedge c) \vee (\neg b \wedge d))$

# CEG and Predicate testing

## CEG

- CEG strategy to define relations among causes and effects (“oracles”)
- Decision table technique to identify test cases

## Predicate testing

- Strategies for deriving test from predicates, fault coverage guarantees

“Better together”

# CEG and Predicate testing

## CEG

- CEG strategy to define relations among causes and effects (“oracles”)
- Decision table technique to identify test cases

## Predicate testing

- Strategies for deriving test from predicates, fault coverage guarantees

“Better together”



# CEG and Predicate testing

## CEG

- CEG strategy to define relations among causes and effects (“oracles”)
- Decision table technique to identify test cases

## Predicate testing

- Strategies for deriving test from predicates, fault coverage guarantees

“Better together”

# Usage of predicate testing techniques

Approaches to test set derivation from predicates can be applied considering different starting points:

- ▶ Specification based testing
- ▶ Program based testing

The different settings have different consequences

## Exercise

Consider the BOR, BRO, BRE criteria for testing predicates including expressions and relational operator, and shortly introduce their objectives and differences. Use the most appropriate criteria for singular expressions to generate a test set, able to discover logical, and relational faults, for the following compound predicates (possibly transforming them):

$$\neg((x \cdot z) \geq (y + z) \wedge \neg p) \wedge ((z = w) \vee p)$$

$$((x(y^2 + 1) \geq x) \wedge a \wedge (x^2y = 7)) \vee ((x^2y - 7 = 0) \wedge xy^2 < 0 \wedge d)$$