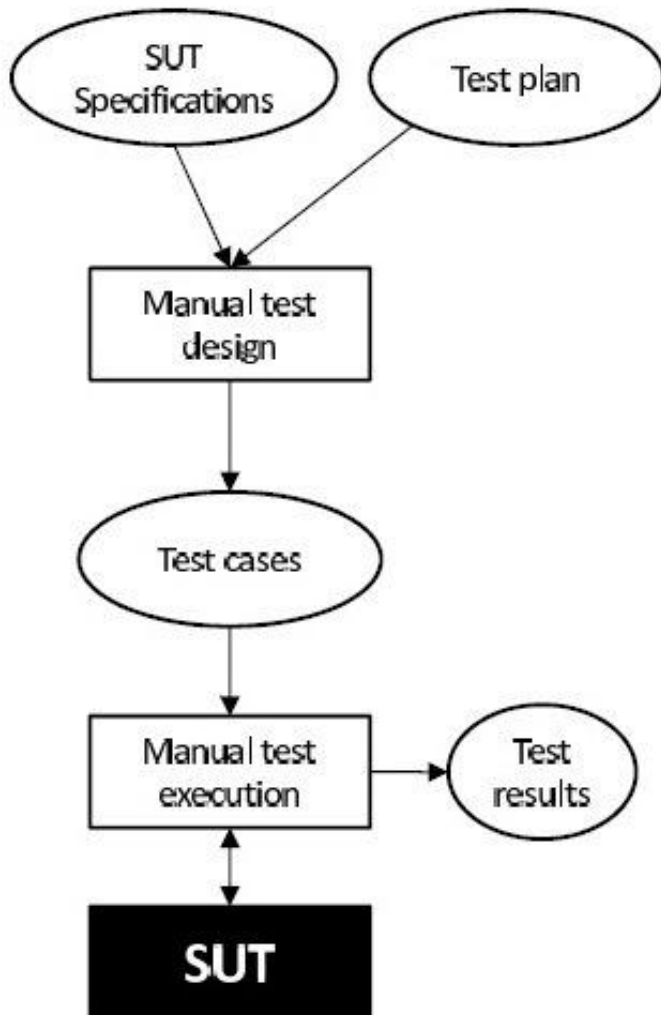


# MBT – MODEL-BASED TESTING

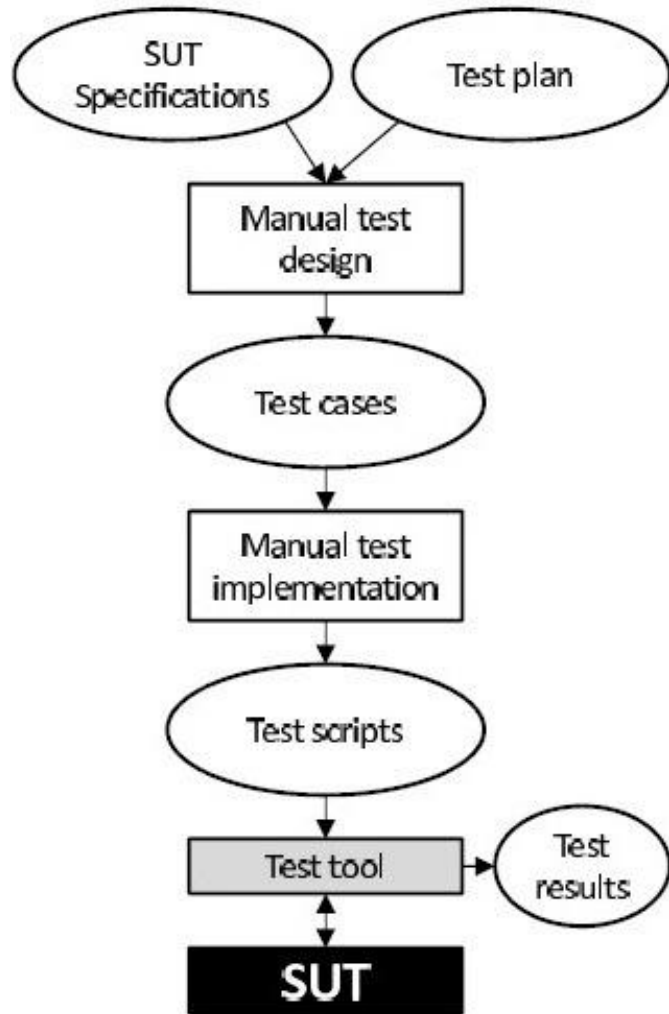
Gusztáv Adamis  
adamis@tmit.bme.hu

# MANUAL TESTING



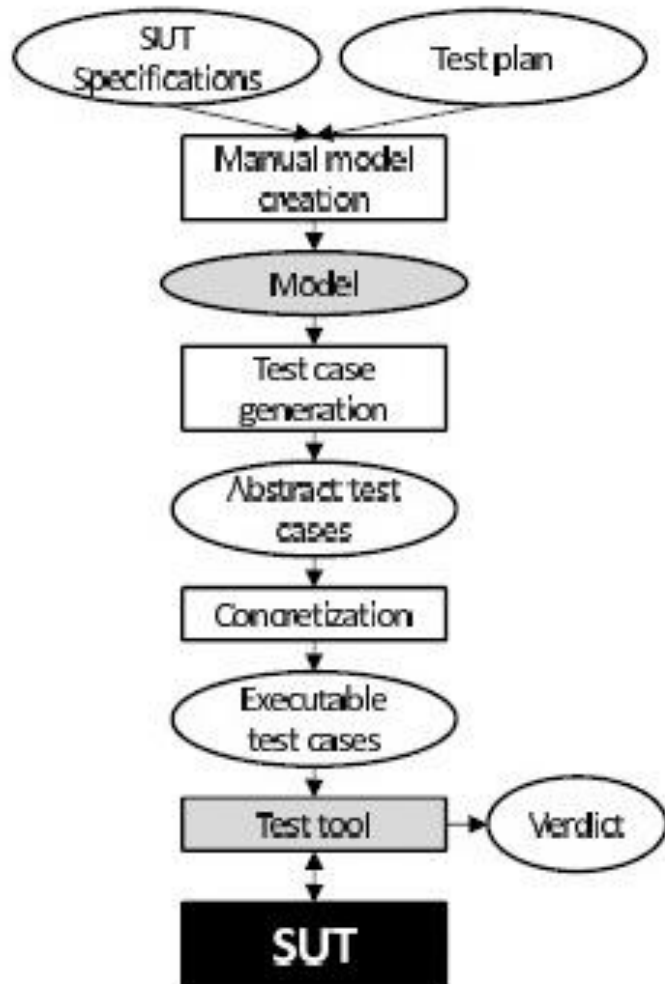
- › Manual execution
  - Slow
  - Time consuming
  - Documentation
  - E.g. GUI test
    - › Though several solutions of partly automating (Selenium, Jasmine)
- › In most of the cases automation is necessary to achieve acceptable coverage in acceptable time frame

# AUTOMATED TESTING (TEST SCRIPTS)



- › Test programs (scripts) to write
  - Time
- › Test execution tool to develop
  - Time
- › But when they are ready, fast execution
- › Problems:
  - Test development
  - Test validation
  - Test maintenance

# MODEL-BASED TESTING



- › Automatic (abstract) test case generation
  - Different strategies, goals
  - Model shall be verified, generated tests are correct
  - Any change in specification -> Re-generate the tests
    - › No need to touch the test scripts
    - › Easier maintenance
- Problem:
  - › Test Harness
  - › Good only for ‘green field’ test development

# TEST GENERATION ALGORITHMS

- › Measuring the adequacy of a test suite
  - Coverage
- › Deciding when to stop test generation
  
- › Typically 100% coverage is not possible
  - Time consuming
  - Non-reachable states
    - › Theoretically – wrong model
    - › Practically – complex guards, lot of variables

# CLASSIFICATION OF TEST SELECTION CRITERIA

- › Most of them can also be applied to code, but now for MODEL
  - Not the same
  - Both shall be tested
- › Structural Model Coverage
- › Data Coverage
- › Fault-Model
- › Requirements-Based
- › Explicit Test Case Specification

# CONTROL-FLOW-ORIENTED COVERAGE CRITERIA

- › For models in (E)FSM, UML, OCL, etc.
  - Decisions, Pre/Post conditions

```
context SmartCard::verifyPin(p:PIN_CODES):MESSAGES
  post:
    if pinTry = 0 then
      result = MESSAGES:NO_MORE_TRIES
    else
      if (p=pin or statusPin=PINSTATUS::DISABLE) then
        result = MESSAGES::SUCCESS
      else
        result = MESSAGES::ERROR
      endif
    endif
  endif
```

# CONTROL-FLOW-ORIENTED COVERAGE CRITERIA

- › Statement Coverage (SC)
  - execute every reachable statement
- › Decision (or Branch) Coverage (DC)
  - each reachable *decision* is made true by some tests and false by other tests
- › Condition Coverage
  - each *condition* in the program is tested with a true result and also with a false result
- › Path Coverage (PC)
  - execute every satisfiable path through the control-flow graph
  - generally impossible to reach (loops!)

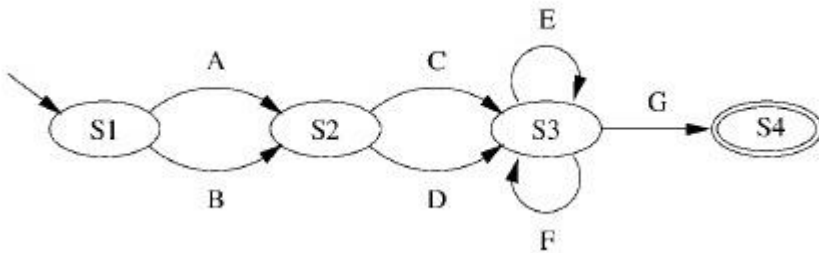


# DATA-FLOW-ORIENTED COVERAGE CRITERIA

- › Definition (assignment to) and use of variables
  - Data Flow Graphs
  - Definition-use paths
- › All-Defs
  - test at least one def-use pair  $(d_v, u_v)$  for every definition  $d_v$ , that is, at least one path from each definition to one of its feasible uses
- › All-Uses
  - test all def-use pairs  $(d_v, u_v)$ . (testing all feasible uses of all defs)
- › All-Def-Use-Paths
  - test all def-use pairs  $(d_v, u_v)$  and to test all paths from  $d_v$ , to  $u_v$
  - Practically unrealistic

# TRANSITION-BASED COVERAGE CRITERIA

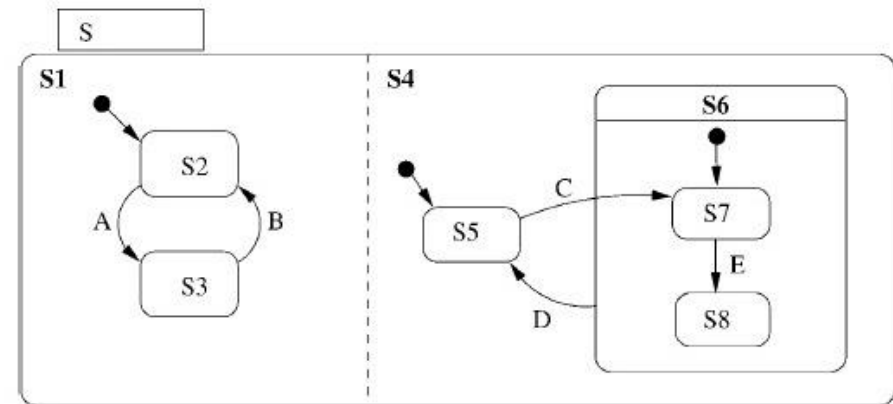
- › For (E)FSM, LTS, UML state charts, etc. models



- › All-states Coverage

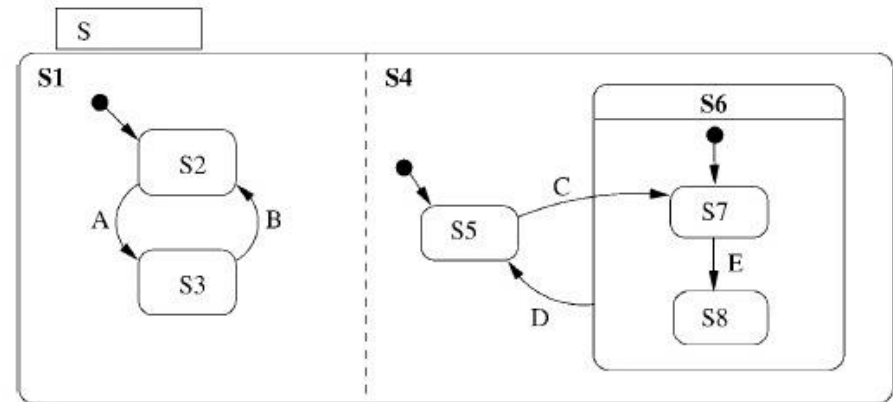
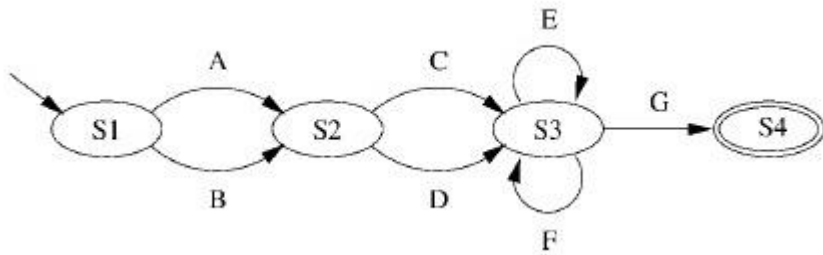
- Every state of the model is visited at least once

- › ACG



- › ACE

# TRANSITION-BASED COVERAGE CRITERIA



## > All-transitions Coverage

– Every transition of the model must be traversed at least once

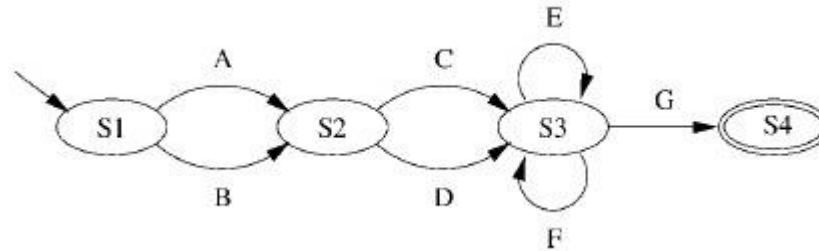
– ACEFG + BD

– ACEFG + BDG (if shall end at final state)

ABCED or ABCEDCD

D is one or two transactions?

# TRANSITION-BASED COVERAGE CRITERIA



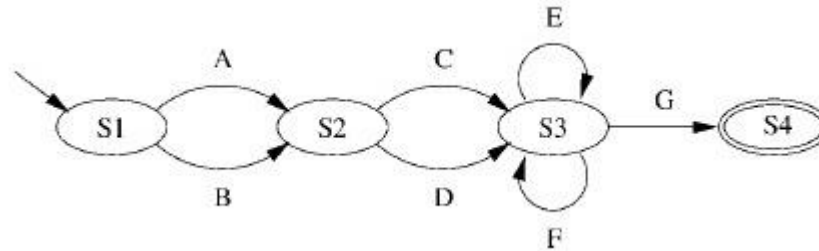
## › All-transition-pairs Coverage

- Every pair of adjacent transitions in the FSM or statechart model must be traversed at least once

## › For S2:

- $AC+AD+BC+BD$

# TRANSITION-BASED COVERAGE CRITERIA



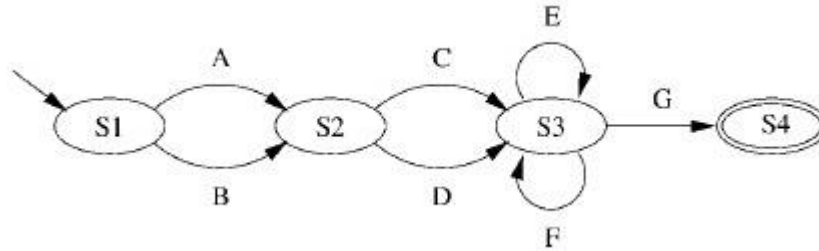
## > All-loop-free-paths Coverage:

- Every loop-free path must be traversed at least once
- ACG + ADG + BCG + BDG
  - > Does not cover all transitions or even all states!

## > All-one-loop-paths Coverage

- Every path containing at most two repetitions of one (and only one) state must be traversed at least once
- All the loop-free paths and all the paths that loop once
  - >  $4 \cdot 3 = 12$  test cases

# TRANSITION-BASED COVERAGE CRITERIA



## > All-round-trips Coverage

- Similar to previous, but requires each loop only once and not with all possible preceding/following paths
- Weaker, but more realistic to achieve in practice
  - > E.g.: ACE + ACF + ACG + AD + B

## > All-paths Coverage

- Every path must be traversed at least once (exhaustive testing)
- If loop: infinite number of paths
- ACG+ADG+BCG+BDG+ACEG+ACEEG+....

# TRANSITION-BASED COVERAGE CRITERIA

- › All-loop-free-paths, All-one-loop-paths, All-round-trips are inadequate on their own, since they do not guarantee to cover all transitions or even all states
  - Extreme example: in the first state we have to take a loop at least twice, nothing else than the first state is reachable...
- › In practice, the All-transition Coverage is the minimum to reach

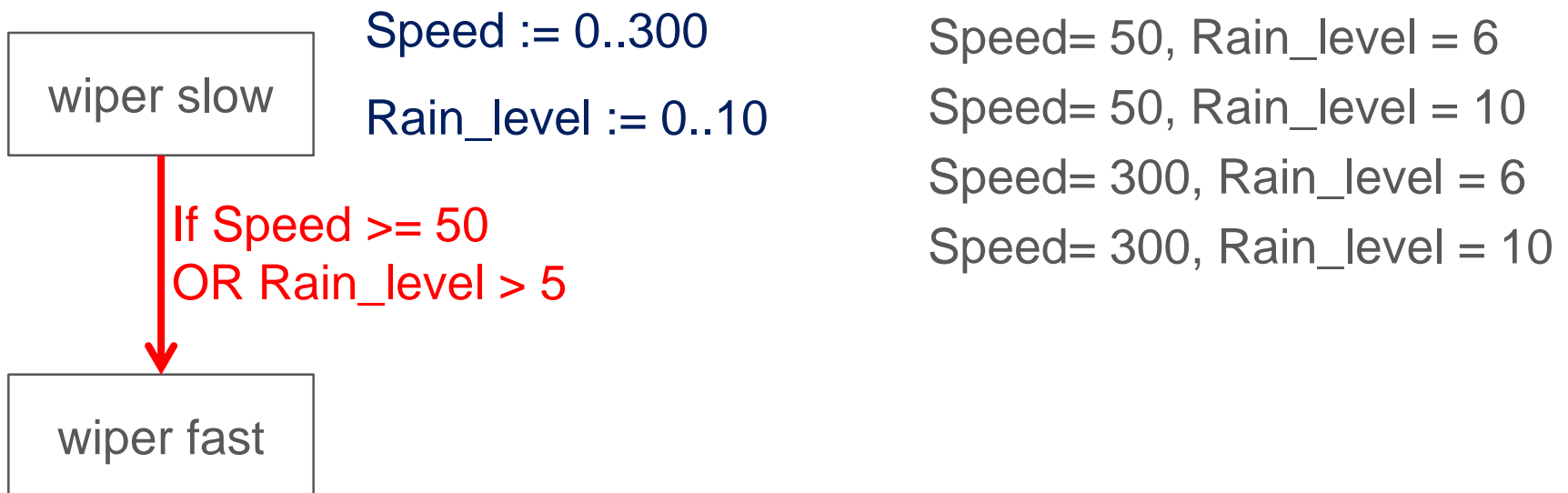
# DATA COVERAGE CRITERIA

- › For choosing a few good data values to use as test inputs when there is a huge number of possible input values
- › Two extremes:
  - **One-value:** simply requires to test at least one value from the domain  $D$ . Often too simple.
  - **All-values:** requires to test every value in the domain  $D$ .
    - › This is not practical if  $D$  is large (e.g., 0. . 999999), but when  $D$  is small, such as an enumerated type, it can be useful to test all possibilities



# BOUNDARY VALUE TESTING

- › Choose test input values at the boundaries of the input domains
  - Lots of faults in the SUT are located at the boundary between two functional behaviours



# STATISTICAL DATA COVERAGE

- › Random value generation can be acceptable
  - If all with the same probability
  - Random value generation ***according to a given distribution***
- › Partition-based testing
- › Experiences show that the efficiency of both is ~same, but random is much easier to implement
- › Boundary testing is enough if we want to have the minimal number of tests, but if we want more
  - Suitable also for non-ordered types, e.g. enumeration
    - › Car colour: silver 24%, black 17%, ...

# PAIRWISE TESTING

- › Pairwise testing is based on the assumption that most defects are created as a result of no more than two test parameters (test values) being in a certain combination
- › E.g:
  - Destinations: Canada, Mexico, USA
  - Class: Coach, Business, First
  - Seat: Aisle, Window
- › Exhaustive testing:
  - $3*3*2 = 18$  combinations

# PAIRWISE TESTING

Test	Destination	Class	Seat Preference
1	Canada	Coach	Aisle
2	Mexico	Coach	Aisle
3 (defect!)	USA	Coach	Aisle
4	Canada	Business Class	Aisle
5	Mexico	Business Class	Aisle
6	USA	Business Class	Aisle
7	Canada	First Class	Aisle
8	Mexico	First Class	Aisle
9	USA	First Class	Aisle
10	Canada	Coach	Window
11	Mexico	Coach	Window
12 (defect!)	USA	Coach	Window
13	Canada	Business Class	Window
14	Mexico	Business Class	Window
15	USA	Business Class	Window
16	Canada	First Class	Window
17	Mexico	First Class	Window
18	USA	First Class	Window

- › Assume: USA, Coach causes the problem
- › Pairwise test generation:
  - Test 18:
    - › USA, First: in 9,
    - › USA, Window: in 15,
    - › First, Window: in 17
  - Test 18 is redundant, etc.

# PAIRWISE TESTING

Number	Destination	Class	Seat Preference
1	Canada	Coach	Aisle
3 (defect!)	USA	Coach	Aisle
5	Mexico	Business Class	Aisle
8	Mexico	First Class	Aisle
9	USA	First Class	Aisle
11	Mexico	Coach	Window
13	Canada	Business Class	Window
15	USA	Business Class	Window
16	Canada	First Class	Window

- › 9 tests instead of 18
- › Still finds the problem

# PAIRWISE TESTING

- › More effective if much higher combinations
  - If 10 variables with 5 values each
    - ›  $5^{10}=9\ 765\ 625$  exhaustive tests
    - › Only 44 pairwise tests
  - If 75 binary variables
    - ›  $2^{75} = 37\ 778\ 931\ 862\ 957\ 161\ 709\ 568$  exhaustive tests
    - › Only 28 pairwise tests
- › Complicated test generation algorithms
- › N-wise coverage
  - If suppose that the problem depends on N values instead 2
  - Number of tests rapidly grows as N increases
  - All-triples can still be practical, but all-quadruples are ~not

# FAULT-BASED CRITERIA

## › Pre-specified faults

- Typically frequently occurred ones
- Mutation operators: e.g. substitute + with – in expressions
- Generate tests for each mutant of the original program,
  - › Design a test that distinguishes that mutant from the original program
  - › The resulting test suite is therefore able to show, which faults are NOT in the SUT
    - Fault-finding Power

# REQUIREMENTS-BASED CRITERIA

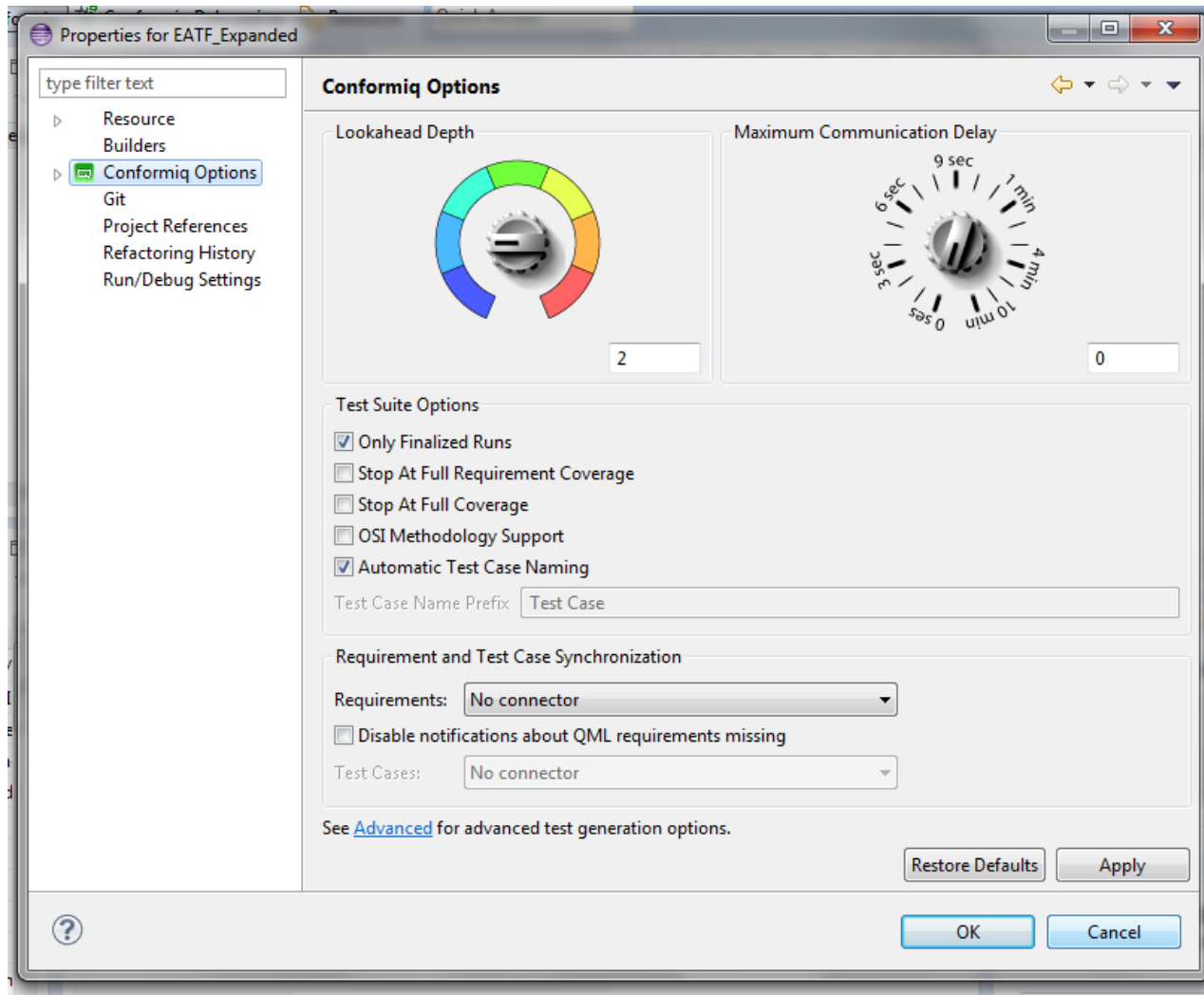
- › High-level, testable statements of functionalities
- › Each requirement shall be tested (e.g. in acceptance tests)
- › In MBT two typical solutions:
  - 1. Record the requirements inside the behavior model (as annotations on various parts of the model) so that the test generation process can ensure that all requirements have been tested
  - 2. Formalize each requirement and then use that formal expression as a test selection criterion to drive the automated generation of one or more tests from the behavior model
    - › Explicit test case specifications



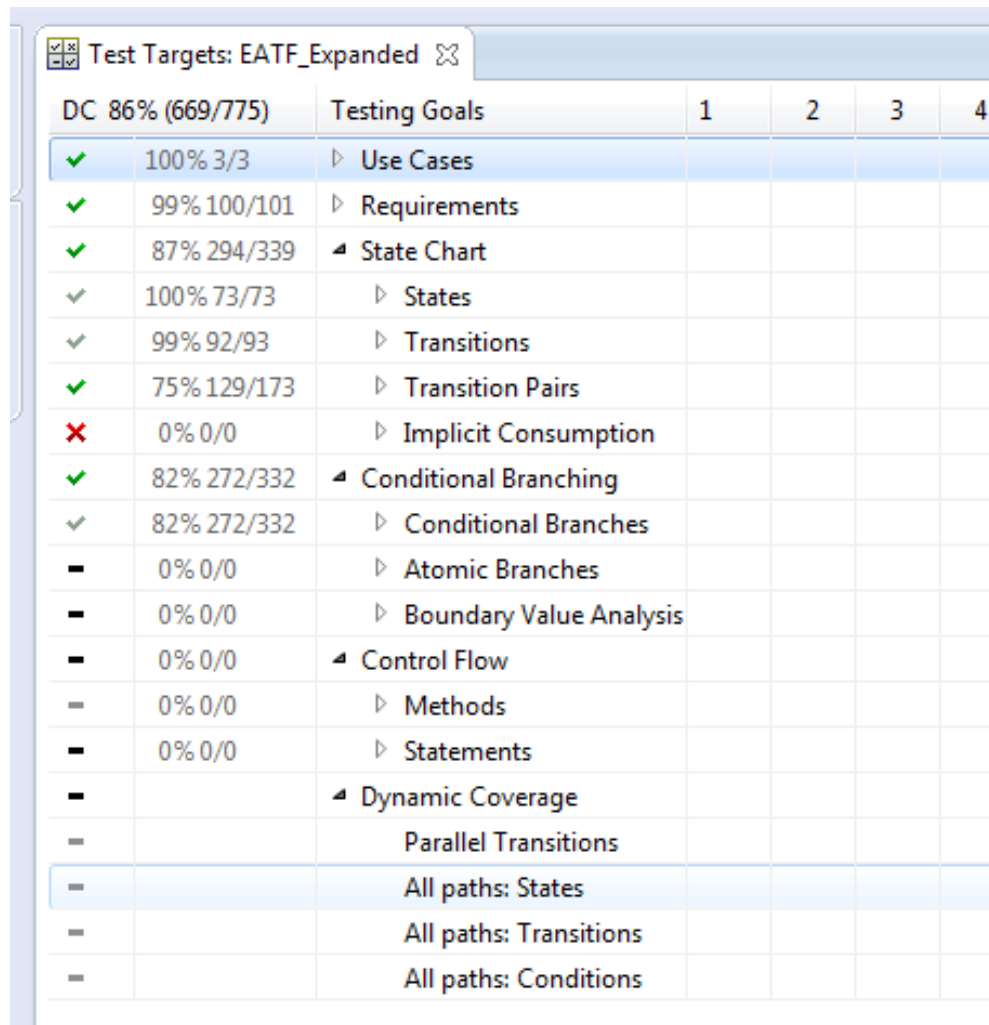
# EXPLICIT TEST CASE SPECIFICATION

- › Some explicit requirements are given in the model
  - E.g: Test shall contain this state
- › Generate test
  - For typical or for less typical cases
  - Just for a given service
  - Etc.

# TEST SELECTION IN AN MBT TOOL (CONFORMIQ)



# TEST SELECTION IN AN MBT TOOL (CONFORMIQ)



The screenshot shows the 'Test Targets: EATF\_Expanded' window in the Conformiq tool. It displays a table of testing goals with their respective coverage percentages and counts. The table has four columns for testing goals (1, 2, 3, 4) and a total coverage of 86% (669/775).

DC 86% (669/775)	Testing Goals	1	2	3	4
✓ 100% 3/3	▸ Use Cases				
✓ 99% 100/101	▸ Requirements				
✓ 87% 294/339	▾ State Chart				
✓ 100% 73/73	▸ States				
✓ 99% 92/93	▸ Transitions				
✓ 75% 129/173	▸ Transition Pairs				
✗ 0% 0/0	▸ Implicit Consumption				
✓ 82% 272/332	▾ Conditional Branching				
✓ 82% 272/332	▸ Conditional Branches				
- 0% 0/0	▸ Atomic Branches				
- 0% 0/0	▸ Boundary Value Analysis				
- 0% 0/0	▾ Control Flow				
- 0% 0/0	▸ Methods				
- 0% 0/0	▸ Statements				
-	▾ Dynamic Coverage				
-	Parallel Transitions				
-	All paths: States				
-	All paths: Transitions				
-	All paths: Conditions				

NX - eantwuh@esekits1059.rnd.ki.sw.ericsson.se/2:1160 - SLED 11 32 TS  
 Conformiq - MaxDia\_TestModel\_Impl/model/SystemBlock.cqa - Conformiq

**Project Explorer**  
 MaxDia\_TestModel\_Impl  
 DC 1  
 model  
 CQ\_TestHarnessTemplate.ttcn3  
 CQ\_TestSuite.ttcn3  
 CQ\_TestSystem.ttcn3  
 CQ\_Types.ttcn3  
 SIPClientPrototype

```

    /** Declaration of the external interface of the system being modeled. This is
    specific to system modeling; a similar construct does not appear usually in
    programming languages. In this "system block", we initially declare one inbound
    interface (in) and one outbound interface (out). The identifiers 'in' and
    'out' are the names for the interfaces in the model. After the colon we
    list the types of records that can possibly go through the interface in
    question. */
    system
    {
      Inbound in : Conn_CER, Send_Message, Stop, Disc, Start, Message,
                Conn_Ack, Conn_Nack, CER, CEA, DPR, DPA, DWR, DWA;
      Outbound out : Message, Disc,
                  Conn_Req, CER, CEA, DPR, DPA, DWR, DWA;
    }

    /** Declaration of a message type, which is technically presented as a "record
    type". It is a record of pure data. This record type 'MyMessage' is empty,
    i.e. it does not contain any actual data fields. */
    record Message { }
    record Send_Message { }
    record Start { }
    record Stop { }
    record Disc { }
    record Conn_CER { }
    record Conn_Req { }
  
```

**Model Browser**  
 Diameter, Diameter.cqa, Main.cqa, SystemBlock.cqa

**Test Case 3**  
 Search:   
 1 Test Case 1  
 2 Test Case 2  
 3 Test Case 3  
 4 Test Case 4  
 5 Test Case 5  
 6 Test Case 6  
 7 Test Case 7  
 8 Test Case 8  
 9 Test Case 9  
 10 Test Case 10

Steps: Test Case 3  
 Trace: Test Case 3  
 Console

```

    main()
    Diameter.Diameter()
    Diameter.run()
    Diameter.initial-state-0
    Diameter.Closed
    Diameter.Send_CEA()
    Diameter.R_Open
    Diameter.R_Open
    Diameter.Send_DPA()
    Diameter.Disc()
    Diameter.final-state-1
  
```

Computer Conformiq - MaxDia... Wed Jul 20, 11:08 AM

wuhen@adatbanyas... Inbox - Microsoft O... xDIA Project EricCOL... Gmail - Inbox - awu... MaxDIA DEMO test ... model\_level\_testing... Total Commander 7... EN  
 NX - eantwuh@esek... Wednesday 2011-07-20

# MBT AT DIFFERENT LEVELS

