

Content

1	Organisation	3
1.1	Topics.....	3
1.2	PÜ & PL.....	3
1.3	Schedule.....	3
2	Introduction.....	4
2.1	Verification and Validation (V&V)	4
2.2	Systematic View-based Approach.....	5
3	Basic Concepts of Testing.....	5
3.1	Testing Levels.....	6
3.2	Software Faults, Errors and Failures	6
3.3	Controllability vs. Observability	6
3.4	Software Testability.....	7
3.5	Coverage criteria.....	7
4	Graph Coverage	7
4.1	Control flow graphs	7
4.2	Testing and Covering Graphs.....	9
4.3	Summary	12
5	Logic Coverage.....	13
5.1	Overview	13
5.2	Active Clause Coverage.....	14
5.3	Inactive Clause Coverage	15
5.4	When is a clause active?	16
5.5	Summary	17
6	Input Domain Characterization	18
6.1	Input Domain Model (IDM)	18
6.2	Approaches to IDM.....	18
6.3	Identifying Characteristics & Partitioning.....	19
6.4	Example: Sides of a triangle.....	19

6.5	Defining Coverage Criteria.....	21
6.6	Constraints	23
6.7	Summary	23
7	Syntactic Coverage.....	24
7.1	Formal Grammar: BNF	24
7.2	Coverage Criteria	25
7.3	Mutations	25
7.4	Program Mutation	26
7.5	Summary	27
8	KobrA.....	28
8.1	Primary Specification Views	28
8.2	Auxiliary Specification Artifacts	29
8.3	Operation Specification.....	29
8.4	Behavioral Model.....	30
8.5	Consistency	30
9	Specification-based testing.....	32
9.1	KobrA-based Test Design Method	32
9.2	Mapping Behavior Model to a formal graph.....	33

1 Organisation

1.1 Topics

„The model-based specification, validation and quality assurance of software systems and components.“

- Testing Overview
- Graph Coverage
- Logic Coverage
- Input Partitioning
- Syntax Coverage
- KobrA Method
- System/Component Specification

1.2 PÜ & PL

Exam

- 13th December
- Based on the material in the lecture notes and tutorials

1.3 Schedule

Lecture

- Friday, 10:15 –11:45, A5, B144
- Video-only lecture 16.09.2022
- No lecture 07.10.2022

Tutorial

- Wednesday, 12:00-13:45, A5, B144
- First tutorial: **28.09.2022**

2 Introduction

2.1 Verification and Validation (V&V)

Verification

- „Are we building the product right?“
- The software should conform to its specification

Validation

- “Are we building the right product?“
- The software should do what the user really requires

💡 V&V is a whole life-cycle process and must be applied at each stage in the software process. It has two principal objectives:

- The discovery of defects in a system
- The assessment of whether the system is usable in an operational situation

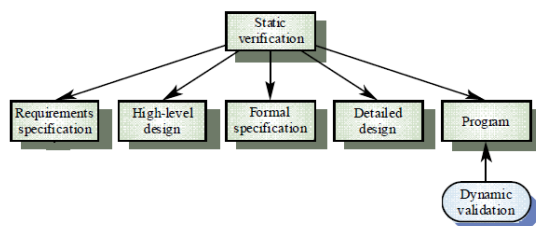
! V&V should establish confidence that the software is fit for purpose.

Software Inspections (static):

- Concerned with analysis of the static system representation to discover problems
- May be supported by tool-based document and code analysis

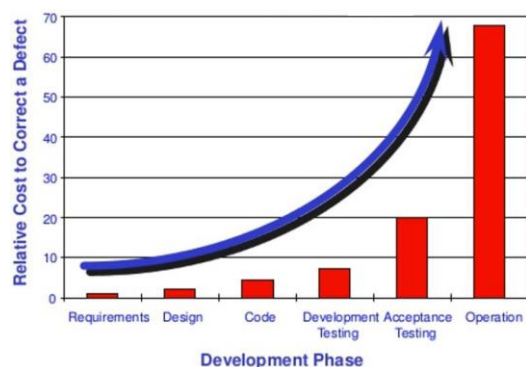
Software testing (dynamic):

- Concerned with exercising and observing system behavior
- The system is executed with test data and its operational behavior is observed

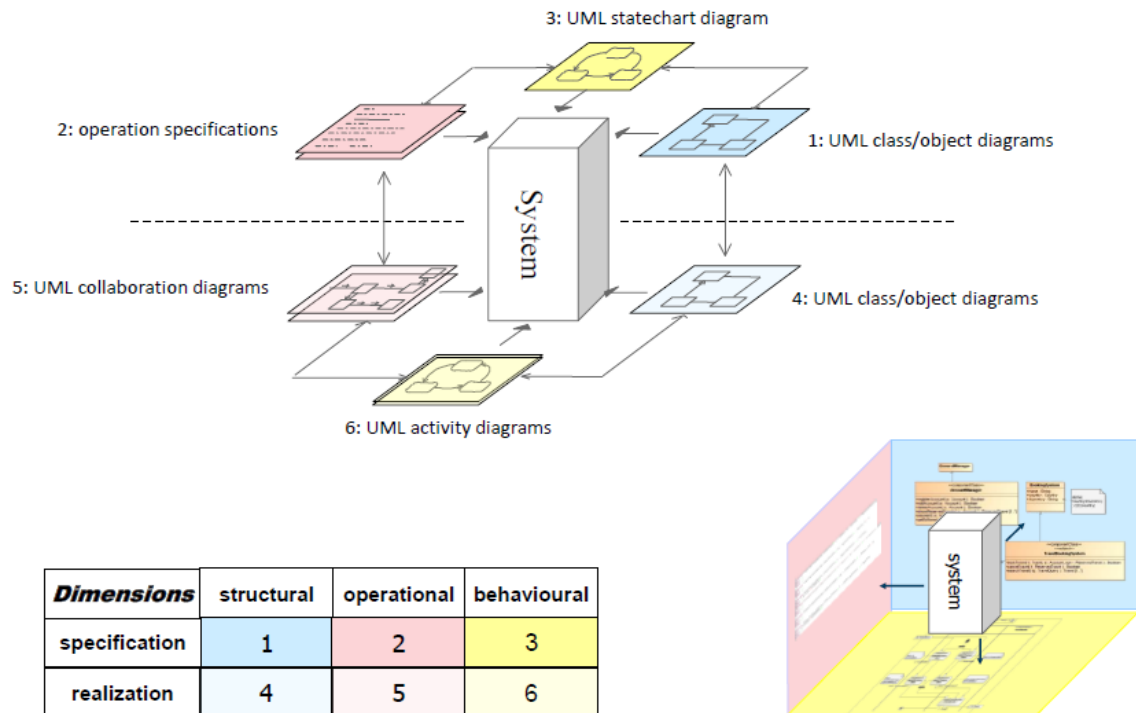


Relative cost of fixing defects increases dramatically the later they are discovered. This has four important implications on how software is developed:

- Acceptance tests as early and often as possible (→ agile)
- Inspections are as important as tests (→ Model-driven development)
- Write specifications unambiguously and understandably (→ MD-development)
- Define tests as early as possible (→ **Model-driven testing**)




2.2 Systematic View-based Approach



KobrA-Approach developed by the Chair of Prof. Atkinson

3 Basic Concepts of Testing

 Software testing is the process of *executing a software system* to determine whether it *matches its specification* and *executes in its intended environment*.

Acceptance testing

Tests designed to establish whether a system fulfills users' and customers' expectations

Reliability testing

Tests designed to reflect the frequency of user inputs → used for reliability estimation

Usability testing

Checks the system from a usability point of view → is all functionality efficiently accessible?

Defect testing

Tests designed to discover system defects. They can reveal the presence of errors not their absence. Only exhaustive testing could show a program is free from defects, but exhaustive testing is impossible

Performance testing

Is used to evaluate and understand the application's scalability when more users are added or the volume of data increases. It is important for identifying bottlenecks in high usage applications

Compatibility testing

Ensures that the application works with differently configured systems based on what the users have or may have

Stress testing

Exercises the system beyond its maximum designed load → checks for unacceptable loss of service or data

3.1 Testing Levels

Component (unit) testing

- Testing of individual program components
- Is usually the responsibility of the component developer (except sometimes for critical systems)


Integration testing

- Testing of groups of components integrated to create a system or subsystem
- The responsibility of an independent testing team
- Usually includes regression testing where tests are re-executed


System testing


- Testing of the entire software in its expected environment (OS, hardware, environment)

3.2 Software Faults, Errors and Failures

 Software fault: A static defect in the software (bug)

- Are equivalent to design mistakes in hardware; an error made by a programmer
- They are there when a system is created and do not “appear” over time when a part wears out

 Software error: An incorrect internal state that is the manifestation of some fault

 Software failure: External, incorrect behavior with respect to the requirements or other description of the expected behavior. Three conditions are necessary for a failure to be observed:


- *Reachability*: the location or locations in the program that contain the fault must be reached during execution
- *Infection*: a software error must occur
- *Propagation*: the error must propagate to cause a failure (i.e., some output of the program to be incorrect)

3.3 Controllability vs. Observability

 Software Controllability


How easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviors

- Easy to control software with inputs from keyboard
- Inputs from hardware sensors or distributed software is harder

 Software Observability

How easy it is to observe the behavior of a program in terms of its outputs, effects on the environment and other hardware and software components

- Software that affects hardware devices, databases or remote files for example have low observability


 Data abstraction (high level programming language, data models, etc.) reduces controllability and observability.

3.4 Software Testability

How easy it is to write effective tests for a software component. Factors that contribute to it:

- *Controllability* – how difficult is it to (re-) configure a program to a certain state?
- *Observability* – how easy is it to observe output values? Are there hidden states?
- *Understandability* – how good is the documentation?
- *Traceability* – are there log mechanisms that allow tracing function calls etc.?
- *Test Support Capability* – common test management and test creation technologies?

3.5 Coverage criteria


 Coverage criteria give systematic ways to search the input space (possible inputs). The goal is to find the fewest inputs that will find the most problems.

Test Requirements

Specific things that must be satisfied or covered during testing.

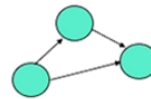
Test Criteria

A Collection of rules and a process that define test requirements.

 In principle a tester's job is simple: to define a model of the software, then find ways to cover it. In practice, however, testers need to find the best "return on investment" in terms of effort spent and faults discovered.

Basic Structures for Defining Test Criteria:

1. Graphs
2. Logical Expressions:
3. Input Domain Characterization:
4. Syntactic Structures




(not X or not Y) and A and B
 A: {0, 1, >1}, B: {600, 700, 800}, C: {swe, isa}
 if (x>y): z=x-y else: z=2*x

4 Graph Coverage

Graphs are the most commonly used structure for testing. They can come from many sources:

- Control flow graphs
- Design structure
- Interaction diagrams
- State diagrams

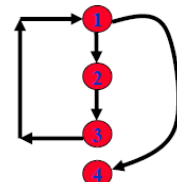
 Tests usually are intended to "cover" the graph in some way.

4.1 Control flow graphs

Useful for visualizing the structure of source code. Is created by numbering all the statements (not declarations or methods) of a program and represent them by nodes in the control flow graph. An edge from one node to another node exists if execution of the statement representing the first node can result in transfer of control to the other node.

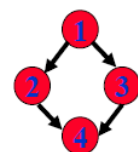
Iteration:

- 1 while(a>b){
- 2 b=b*a;
- 3 b=b-1;}
- 4 c=b+d;



Selection:

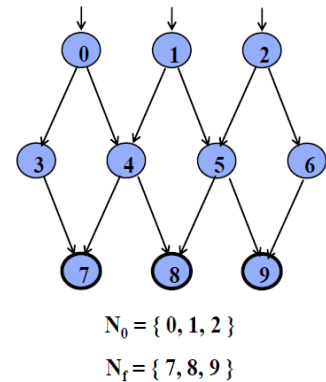
- 1 if(a>b) then
- 2 c=3;
- 3 else c=5;
- 4 c=c*c;



4.1.1 Definition of a Graph

Consists of four sets:

- A set N of nodes, N is not empty
- A set N_0 of initial nodes, N_0 is non-empty subset of N
- A set N_f of final nodes, N_f is non-empty subset of N
- A set E of edges, each edge from one node to another (n_i, n_j) means an edge where i is the predecessor and j is the successor



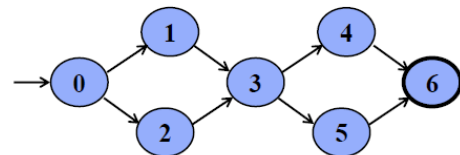
4.1.2 Paths

A *path* p is a sequence of nodes where each pair of nodes is an edge in the graph $[n_1, n_2, \dots, n_m]$. The *length* of a path in a graph is the number of edges. A subsequence of nodes in p is a sub path of p .

A *test path* starts at an initial node and ends at a final node. It represents execution of test cases. Some test paths can be executed by many tests while some cannot be executed by any tests.

SESE (single entry, single exit) graphs are all test paths that start at a single node and end at another node (N_0 and N_f have exactly one node).

- A test path p visits node n if n is in p
- A test path p visits edge e if e is in p
- A test path p tours subpath q if q is a subpath of p



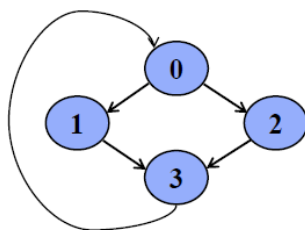
Double-diamond graph

Four test paths

$[0, 1, 3, 4, 6]$
 $[0, 1, 3, 5, 6]$
 $[0, 2, 3, 4, 6]$
 $[0, 2, 3, 5, 6]$

A path from node n_i to n_j is *simple* if no node appears more than once, except the first and last nodes are the same (no internal loops, any path can be composed of simple paths).

A simple path that does not appear as a proper subpath of any other simple path is a *prime path*.



Simple Paths : $[0], [1], [2], [3], [0, 1], [0, 2], [1, 3], [2, 3], [3, 0], [0, 1, 3], [0, 2, 3], [1, 3, 0], [2, 3, 0], [3, 0, 1], [3, 0, 2], [0, 1, 3, 0], [0, 2, 3, 0], [1, 3, 0, 1], [2, 3, 0, 2], [3, 0, 1, 3], [3, 0, 2, 3], [1, 3, 0, 2], [2, 3, 0, 1]$

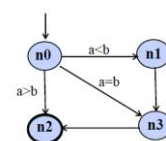
Prime Paths : $[0, 1, 3, 0], [0, 2, 3, 0], [1, 3, 0, 1], [2, 3, 0, 2], [3, 0, 1, 3], [3, 0, 2, 3], [1, 3, 0, 2], [2, 3, 0, 1]$

4.1.3 Tests

- $path(t)$ is the test path executed by test t
- $path(T)$ is the set of test paths executed by the set of tests T

Each test executes one and only one test path. A location in a graph (node or edge) can be *reached* from another location if there is a sequence of edges from the first location to the second.

- Syntactic reach: A sub path exists in the graph
- Semantic reach: A test exists that can execute that subpath





Test t1: $(a=0, b=1) \rightarrow path(t1) = [n0, n1, n3, n2]$
 Test t2: $(a=1, b=1) \rightarrow path(t2) = [n0, n3, n2]$
 Test t3: $(a=2, b=1) \rightarrow path(t3) = [n0, n2]$


$T = \{t1, t2, t3\}$


$path(T) = \{path(t1), path(t2), path(t3)\}$

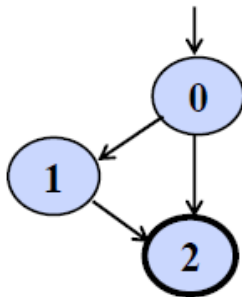
4.2 Testing and Covering Graphs

 **Test requirements (TR)** describe properties of test paths. Test criterion are rules that define test requirements.

 Given a set TR of test requirements for a criterion C , a set of tests T satisfies C on a graph if and only if for every test requirement in TR , there is a test path in $path(T)$ that meets the test requirement TR .

 **Node Coverage (NC)**: Test set T satisfies node coverage on graph G if for every syntactically reachable node n in N , there is some path p in $path(T)$ such that p visits n . $\rightarrow TR$ contains each reachable node in G .

 **Edge Coverage (EC)**: TR contains each reachable path of length up to 1, inclusive, in G .




Node Coverage : $TR = \{ 0, 1, 2 \}$

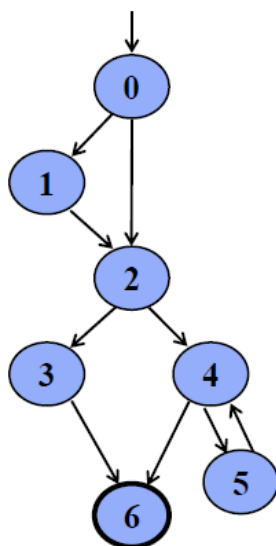
Test Path = $[0, 1, 2]$

Edge Coverage : $TR = \{ (0,1), (0, 2), (1, 2) \}$

Test Paths = $[0, 1, 2], [0, 2]$

 **Edge-Pair Coverage (EPC)**: TR contains each reachable path of lengths up to 2, inclusive, in G .

 **Complete Path Coverage (CPC)**: TR contains all paths in G .



Node Coverage

$TR = \{ 0, 1, 2, 3, 4, 5, 6 \}$

Test Paths: $[0, 1, 2, 3, 6] [0, 1, 2, 4, 5, 4, 6]$

Edge Coverage

$TR = \{ (0,1), (0,2), (1,2), (2,3), (2,4), (3,6), (4,5), (4,6), (5,4) \}$

Test Paths: $[0, 1, 2, 3, 6] [0, 2, 4, 5, 4, 6]$

Edge-Pair Coverage

$TR = \{ [0,1,2], [0,2,3], [0,2,4], [1,2,3], [1,2,4], [2,3,6], [2,4,5], [2,4,6], [4,5,4], [5,4,5], [5,4,6] \}$


Test Paths: $[0, 1, 2, 3, 6] [0, 1, 2, 4, 6] [0, 2, 3, 6] [0, 2, 4, 5, 4, 5, 4, 6]$

Complete Path Coverage

Test Paths: $[0, 1, 2, 3, 6] [0, 1, 2, 4, 6] [0, 1, 2, 4, 5, 4, 6] [0, 1, 2, 4, 5, 4, 5, 4, 6] [0, 1, 2, 4, 5, 4, 5, 4, 5, 4, 6] \dots$

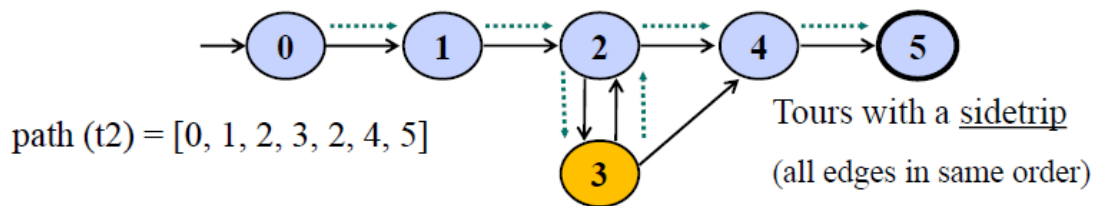
Through the loop between node 4 and 5 there are infinite possible paths, which makes complete path coverage impossible.

 **Prime Path Coverage (CPC)**: TR contains each prime path in G .

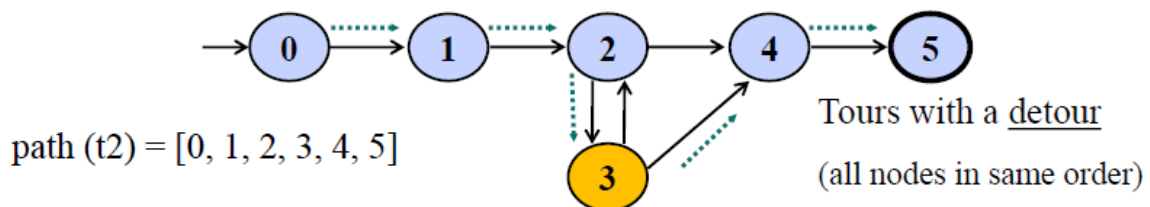
 Prime paths do not have internal loops, test paths might. A test path can make an excursion when touring a subpath either as *sidetrip* or a *detour*.

Excursions

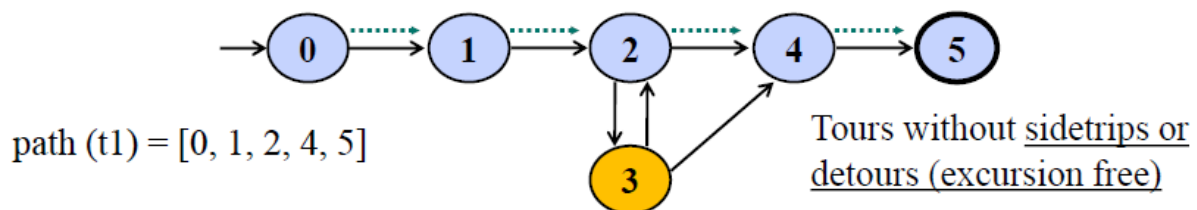
A test path p tours subpath q with **sidetrips** if every **edge** in q is also in p in the same order. The tour can include a sidetrip, as long as it comes back to the same node.



A test path p tours subpath q with **detours** if every **node** in q is also in p in the same order. The tour can include a detour from node n_i , as long as it comes back to the prime path at a successor of n_i .



A test path that tours a path without a sidetrip or a detour is referred to as *excursion-free* and is said to tour the path *directly*.



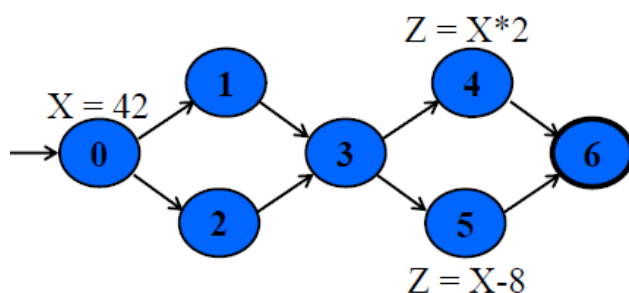
Data Flow Criteria - Goal: Try to ensure that values are computed and used correctly.

Definition (def): an assignment of a value to a variable at a node

Use: a use of a value in a variable at a node

def(n): the set of variables that are defined at node n

use(n): the set of variables that are used at node n



Defs: def (0) = {X} def (4) = {Z} def (5) = {Z} Uses: use (4) = {X} use (5) = {X}

An *infeasible* test requirement cannot be satisfied. Most test criteria have some infeasible TR. Best Effort Touring: satisfy as many TRs as possible without excursions but allow excursions to try to satisfy TR that otherwise would be unsatisfiable.

DU Pairs & Paths

- du pair: A pair of nodes (n_i, n_j) such that a variable v is defined at n_i and used at n_j
- def-clear: A path from n_i to n_j is *def-clear* with respect to variable v if v is not given another value on any of the nodes or edges in the path
- du path: A simple subpath that is def-clear with respect to v from a def of v to a use of v
- $\text{du}(n_i, n_j, v)$ – the set of du-paths from n_i to n_j with respect to v
- $\text{du}(n_i, v)$ – the set of du-paths that start at n_i with respect to v

du pairs for X

$\{0, 4\}$

$\{0, 5\}$

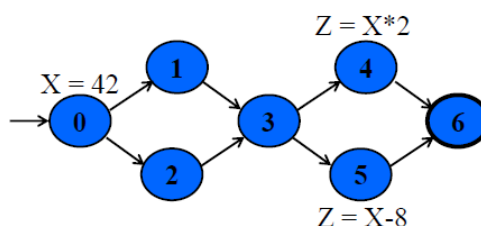
du paths for X

$[0, 1, 3, 4]$

$[0, 2, 3, 4]$

$[0, 1, 3, 5]$

$[0, 2, 3, 5]$



Defs: $\text{def}(0) = \{X\}$

$\text{def}(4) = \{Z\}$

$\text{def}(5) = \{Z\}$

Uses: $\text{use}(4) = \{X\}$

$\text{use}(5) = \{X\}$

All-defs coverage (ADC): For each set of du-paths $S = \text{du}(n_i, v)$, TR contains at least one path d in S . → make sure every def reaches a use.

All-uses coverage (AUC): For each set of du-paths $S = \text{du}(n_i, n_j, v)$, TR contains at least one path d in S . → make sure every def reaches all possible uses.

All-du-paths coverage (ADUPC): For each set $S = \text{du}(n_i, n_j, v)$, TR contains every path d in S . → make sure every path between *defs* and *uses* is covered.

du pairs for X

$\{0, 4\}$

$\{0, 5\}$

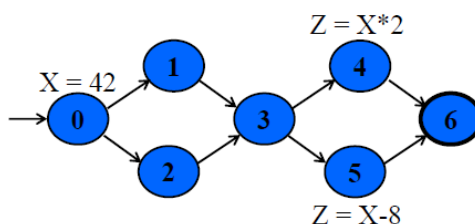
du paths for X

$[0, 1, 3, 4]$

$[0, 2, 3, 4]$

$[0, 1, 3, 5]$

$[0, 2, 3, 5]$



All-defs for X

$[0, 1, 3, 4]$

All-uses for X

$[0, 1, 3, 4]$

$[0, 1, 3, 5]$

All-du-paths for X

$[0, 1, 3, 4]$

$[0, 2, 3, 4]$

$[0, 1, 3, 5]$

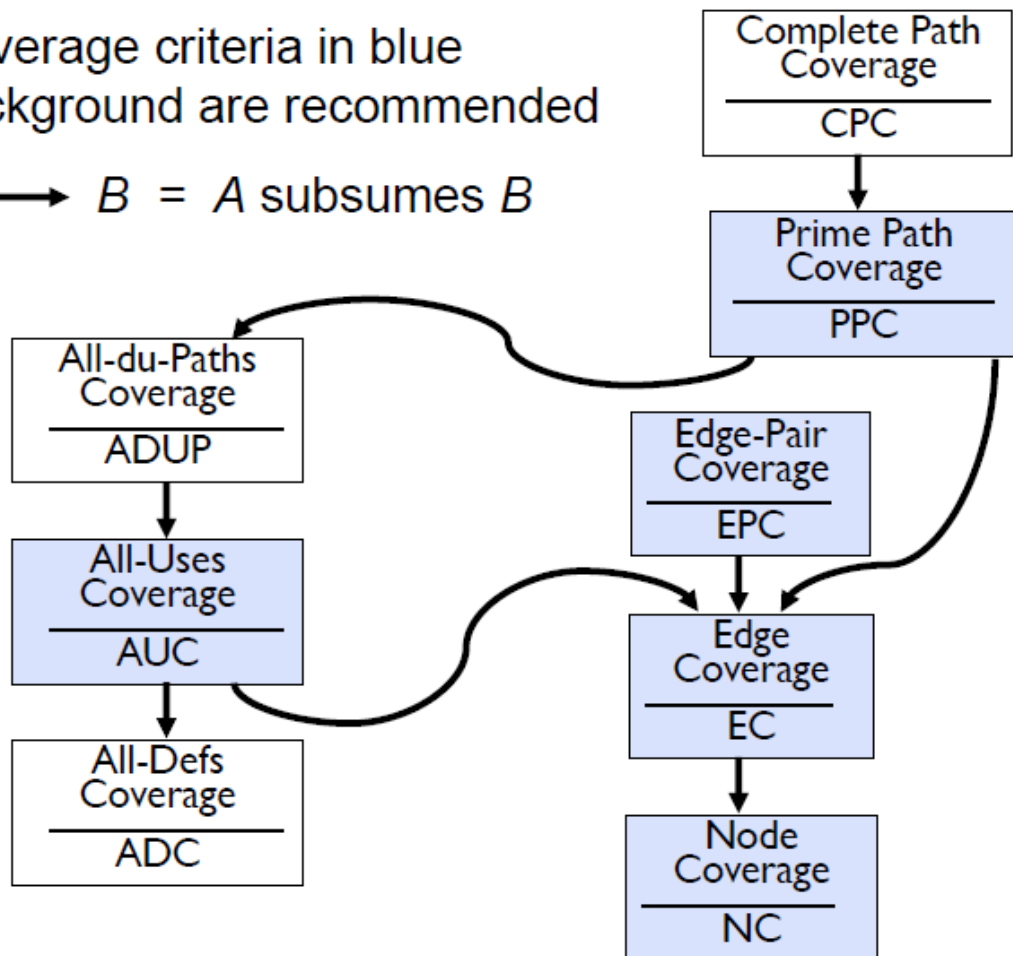
$[0, 2, 3, 5]$

Coverage criteria

Excursions can be used, just as with previous touring, but test paths have to du-tour subpaths. A test path p du-tours subpath d with respect to v if p tours d and the subpath taken is *def-clear* with respect to v .

4.3 Summary

- Coverage criteria in blue background are recommended
- $A \longrightarrow B = A$ subsumes B



5 Logic Coverage

5.1 Overview

A *predicate* is an expression that evaluates to a Boolean value. Predicates can contain Boolean variables, non-boolean variables related by (logical) operators (>, <, ==, not, and, or, ...). A *clause* is a predicate with no logical operators.

$$(a < b) \vee f(z) \wedge b \wedge (m \geq n)$$

four clauses:

- $(a < b)$ – relational expression
- $f(z)$ – Boolean-valued function
- b – Boolean variable
- $(m \geq n)$ – relational expression

Abbreviations

P is the set of predicates

p is a single predicate in P

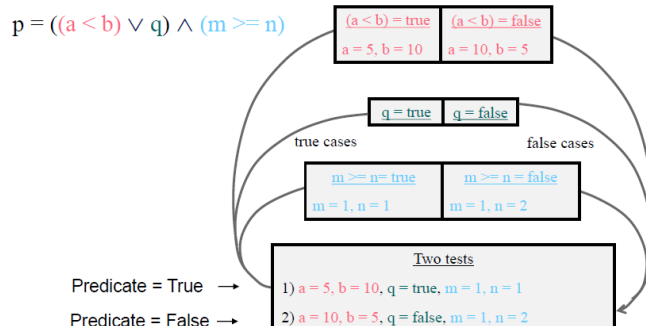
C is the set of clauses in P

C_p is the set of clauses in predicate p

c is a single clause in C

Predicate Coverage (PC): For each p in P , TR contains two requirements: p evaluates to true, and p evaluates to false.

Clause Coverage (CC): For each c in C , TR contains two requirements: c evaluates to true, and c evaluates to false.



$$\text{Predicate} = ((a < b) \vee q) \wedge (m \geq n)$$

Predicate = true

$$\begin{aligned}
 &a = 5, b = 10, q = \text{true}, m = 1, n = 1 \\
 &= ((5 < 10) \vee \text{true}) \wedge (1 \geq 1) \\
 &= (\text{true} \vee \text{true}) \wedge \text{true} \\
 &= \text{true}
 \end{aligned}$$

Predicate = false

$$\begin{aligned}
 &a = 10, b = 5, q = \text{false}, m = 1, n = 1 \\
 &= ((10 < 5) \vee \text{false}) \wedge (1 \geq 1) \\
 &= (\text{false} \vee \text{false}) \wedge \text{true} \\
 &= \text{false}
 \end{aligned}$$

! CC does not necessarily guarantee PC.

Combinatorial Coverage (CoC): For each predicate p in P , TR has test requirements for every clause in the predicate C_p to evaluate to each possible combination of truth values. → CoC requires every possible combination

This is simple but very expensive as it requires 2^n tests, where n is the number of clauses.

	$a < b$	q	$m \geq n$	$((a < b) \vee q) \wedge (m \geq n)$
1	T	T	T	T
2	T	T	F	F
3	T	F	T	T
4	T	F	F	F
5	F	T	T	T
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

A clause being tested independently from other clauses makes it *active*. When it is the focus of the attempt to execute it with a value of true and false.

5.2 Active Clause Coverage

A clause c_i in predicate p (called the *major clause*) determines p if and only if the values of the remaining *minor clauses* c_j are such that changing c_i changes the value of p .

$$p = a \vee b$$

if $b = \text{true}$, p is always true.
so if $b = \text{false}$, a determines p .
if $a = \text{false}$, b determines p .

$$p = a \wedge b$$

if $b = \text{false}$, p is always false.
so if $b = \text{true}$, a determines p .
if $a = \text{true}$, b determines p .

Active Clause Coverage (ACC): For each p in P and each major clause c_i in C_p , choose minor clauses $c_j, j \neq i$ so that c_i determines p . TR has two requirements for each c_i : c_i evaluates to true and c_i evaluates to false.

$$p = a \vee b$$

1) $a = \text{true}$, $b = \text{false}$

2) $a = \text{false}$, $b = \text{false}$

3) $a = \text{false}$, $b = \text{true}$

4) $a = \text{false}$, $b = \text{false}$

? Ambiguity: Do the minor clauses have to have the same values when the major clause is true and false?

- Minor clauses do not need to be the same (GACC)
- Minor clauses need to be the same (RACC)
- Minor clauses force the predicate to become both true and false (CACC)

General Active Clause Coverage (GACC): ACC, but the values chosen for the minor clauses c_j do not need to be the same when c_i is true as when c_i is false.

Major clause (a) is evaluated to true and false and determines P

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Minor clauses (b and c) don't have to be the same

GACC w.r.t. clause - $a: \{1,2,3\} \times \{5,6,7\} = (1,5), (1,6), (1,7), (2,5), (2,6), (2,7), (1,5), (3,6), (3,7).$

Major clause (b) is evaluated to true and false and determines P

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Minor clauses (a and c) don't have to be the same

GACC w.r.t. clause - $a: \{1,2,3\} \times \{5,6,7\}$
 $b: \{2,4\}$

Major clause (c) is evaluated to true and false and determines P

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Minor clauses (a and b) don't have to be the same

GACC w.r.t. clause - $a: \{1,2,3\} \times \{5,6,7\}$
 $b: \{2,4\}$
 $c: \{3,4\}$

! This is the most relaxed but its possible to satisfy GACC without satisfying predicate coverage.

Correlated Active Clause Coverage (CACC): ACC, but the values chosen for the minor clauses c_j must cause p to be true for one value of the major clause c_i and false for the other.

Major clause (a) is evaluated to true and false and determines P

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Minor clauses (b and c) must cause p to be true for one value of the major clause and false for the other

CACC w.r.t. clause - $a: \{1,2,3\} \times \{5,6,7\}$

Major clause (b) is evaluated to true and false and determines P

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Minor clauses (a and c) must cause p to be true for one value of the major clause and false for the other

CACC w.r.t. clause - $a: \{1,2,3\} \times \{5,6,7\}$
 $b: \{2,4\}$


Major clause (c) is evaluated to true and false and determines P

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Minor clauses (a and b) must cause p to be true for one value of the major clause and false for the other

CACC w.r.t. clause - $a: \{1,2,3\} \times \{5,6,7\}$
 $b: \{2,4\}$
 $c: \{3,4\}$

! This is a compromise between the others and satisfies predicate coverage.

 **Restricted Active Clause Coverage (RACC):** ACC, but the values chosen for the minor clauses c_i must be the same when c_i is true as when c_i is false.

Major clause (a) is evaluated to true and false and **determines P**

Minor clauses (b and c) have to be the same

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

RACC w.r.t. clause - a: (1,5) (2,6) (3,7)

Major clause (b) is evaluated to true and false and **determines P**

Minor clauses (a and c) have to be the same

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

RACC w.r.t. clause - a: (1,5) (2,6) (3,7)
b: (2,4)

Major clause (c) is evaluated to true and false and **determines P**


Minor clauses (a and b) have to be the same

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

RACC w.r.t. clause - a: (1,5) (2,6) (3,7)
b: (2,4)
c: (3,4)


! This is the strictest variant and often leads to infeasible test requirements.

5.3 Inactive Clause Coverage

 **Inactive Clause Coverage (ICC):** For each p in P and each major clause c_i in C_p , choose minor clauses $c_j, j \neq i$, so that c_i does not determine p . TR has four requirements for each c_i :

- c_i evaluates to true with p true
- c_i evaluates to true with p false
- c_i evaluates to false with p true
- c_i evaluates to false with p false

💡 Predicate needs to evaluate to true and false!

 **General Inactive Clause Coverage (GICC):** The values chosen for the minor clauses c_j do not need to be the same when c_i is true as when c_i is false.

Major clause (a) is evaluated to true and false and **does not determine P**

Minor clauses (b and c) don't have to be the same

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

GICC w.r.t. clause - a: P_T = infeasible, P_F = (4,8)

Major clause (b) is evaluated to true and false and **does not determine P**

Minor clauses (a and c) don't have to be the same

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F


GICC w.r.t. clause - a: P_T = infeasible, P_F = (4,8)
b: P_T = (1, 3), P_F = {5,6} x {7,8}

Major clause (c) is evaluated to true and false and **does not determine P**

Minor clauses (a and b) don't have to be the same

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

GICC w.r.t. clause - a: P_T = infeasible, P_F = (4,8)
b: P_T = (1, 3), P_F = {5,6} x {7,8}
c: P_T = (1, 2), P_F = {5,7} x {6,8}

 **Restricted Inactive Clause Coverage (RICC):** The values chosen for the minor clauses c_j must be the same when c_i is true as when c_i is false.

Major clause (a) is evaluated to true and false and **does not determine P**

Minor clauses (b and c) **have** to be the same

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

RICC w.r.t. clause - a: P_T = infeasible, P_F = (4,8)

Major clause (b) is evaluated to true and false and **does not determine P**

Minor clauses (a&c) **have** to be the same

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

RICC w.r.t. clause - a: P_T = infeasible, P_F = (4,8)
b: P_T = (1, 3), P_F = (5,7), (6,8)

Major clause (c) is evaluated to true and false and **does not determine P**

Minor clauses (a and b) **have** to be the same

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

RICC w.r.t. clause - a: P_T = infeasible, P_F = (4,8)
b: P_T = (1, 3), P_F = (5,7), (6,8)
c: P_T = (1, 2), P_F = (5,6), (7,8)

5.4 When is a clause active?

Let $p_{c=true}$ be the original predicate p but with every occurrence of c replaced by true and $p_{c=false}$ be the original predicate p but with every occurrence of c replaced by false.

To find values for the minor clauses, connect $p_{c=true}$ and $p_{c=false}$ with exclusive OR:

$$p_c = p_{c=true} \oplus p_{c=false}$$

After solving, p_c describes exactly the values needed for c to determine p .

5.4.1 De Morgan's Laws & Logical XOR

- $\neg(a \vee b) = \neg a \wedge \neg b$
- $\neg(a \wedge b) = \neg a \vee \neg b$

a	b	$a \wedge b$	$a \wedge \neg b$	$a \oplus (a \wedge b)$
T	T	T	F	F
T	F	F	T	T
F	T	F	F	F
F	F	F	F	F

$$a \oplus (a \wedge b) = a \wedge \neg b$$

a	b	$a \vee b$	$\neg a \wedge b$	$a \oplus (a \vee b)$
T	T	T	F	F
T	F	T	F	F
F	T	T	T	T
F	F	F	F	F

$$a \oplus (a \vee b) = \neg a \wedge b$$

5.4.2 Examples

$$\begin{aligned} p &= a \vee b \\ p_a &= p_{a=true} \oplus p_{a=false} \\ &= (\text{true} \vee b) \oplus (\text{false} \vee b) \\ &= \text{true} \oplus b \\ &= \neg b \end{aligned}$$

To make a active, the value of b needs to be false

$$\begin{aligned} p &= a \wedge b \\ p_a &= p_{a=true} \oplus p_{a=false} \\ &= (\text{true} \wedge b) \oplus (\text{false} \wedge b) \\ &= b \oplus \text{false} \\ &= b \end{aligned}$$

To make a active, the value of b needs to be true

$$\begin{aligned} p &= (a \wedge b) \vee (a \wedge \neg b) \\ p_a &= p_{a=true} \oplus p_{a=false} \\ &= ((\text{true} \wedge b) \vee (\text{true} \wedge \neg b)) \oplus ((\text{false} \wedge b) \vee (\text{false} \wedge \neg b)) \\ &= (b \vee \neg b) \oplus \text{false} \\ &= \text{true} \oplus \text{false} \\ &= \text{true} \end{aligned}$$

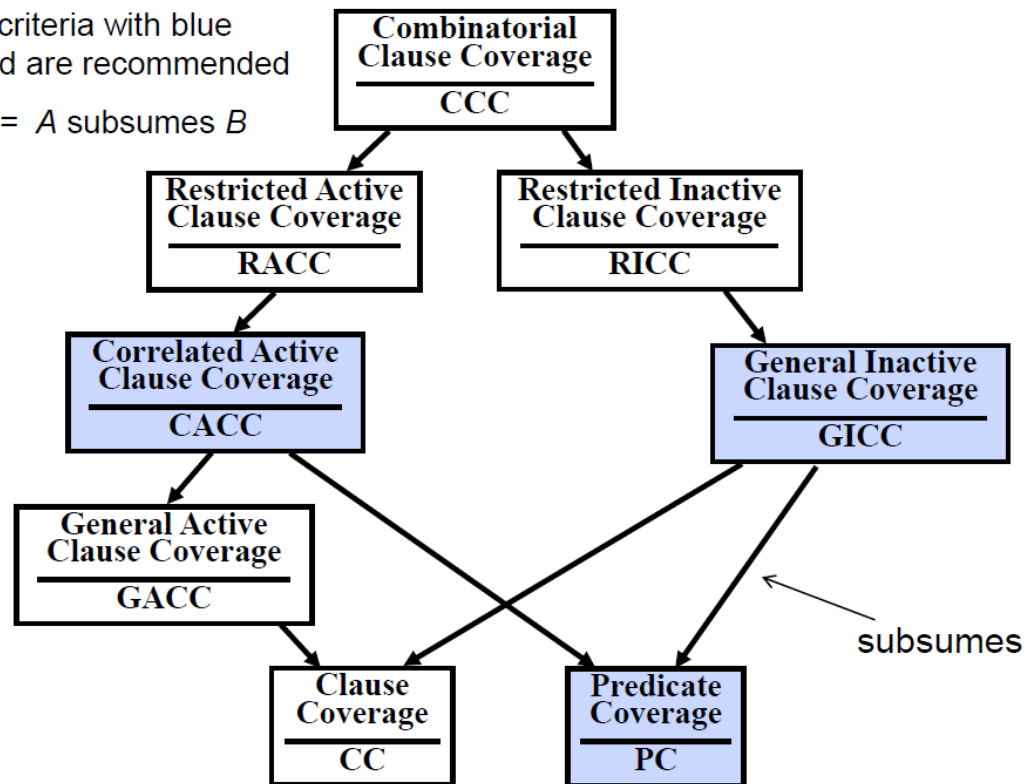
$$\begin{aligned} p &= (a \wedge b) \vee (a \wedge \neg b) \\ p_b &= p_{b=true} \oplus p_{b=false} \\ &= ((a \wedge \text{true}) \vee (a \wedge \neg \text{true})) \oplus ((a \wedge \text{false}) \vee (a \wedge \neg \text{false})) \\ &= (a \vee \text{false}) \oplus (\text{false} \vee a) \\ &= a \oplus a \\ &= \text{false} \end{aligned}$$

a always determines the value of this predicate, b never does.

5.5 Summary

Coverage criteria with blue background are recommended

$A \rightarrow B = A$ subsumes B




6 Input Domain Characterization

The *input domain* to a program contains all the possible inputs (combinations) to that program. For even small programs, the input domain is so large that it might as well be infinite.

Input parameters define the scope of the input domain, they can be:

- Parameters to a method
- Data read from a file
- Global variables
- User level inputs

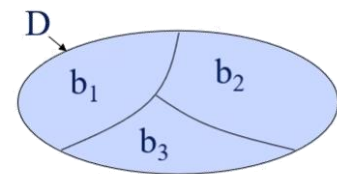
The basic goal of this approach is to identify regions of the input domain which are expected to have similar behavior when it comes to uncovering faults in the software.


 The input domain for each input parameter is *partitioned into regions* that have similar behavior with regard to uncovering faults. Then criteria are defined on how values should be selected from these regions/blocks to test the software. The very minimum is to pick at least one value from each region.

6.1 Input Domain Model (IDM)

Used to define coverage criteria based on the input domain.

- Domain D is a set that contains all possible combinations of all possible values of all the inputs to the software system
- Partition (scheme) q (of D), also called question q , defines a set of blocks $B_q = b_1, b_2, \dots, b_Q$ which the whole domain is divided into
- The partitioning into these blocks must satisfy two properties:
 - blocks must be *pairwise disjoint* (no overlap): $b_i \cap b_j = \emptyset$
 - together the blocks *cover* the domain D (complete): $b_1 \cup \dots \cup b_Q = D$



 Assumption: Each value from each partition is *equally useful* for testing.

6.2 Approaches to IDM

Approaches to identifying questions that can lead to partitions of the input domain.

6.2.1 Interface-based approach

Develops characteristics directly from individual input parameters. Meaning and semantics of the software that should be tested are not known. It's the simplest to apply and can be partially automated in some situations.

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
// else return true if element is in the list, false otherwise
```

Interface-Based Approach

Two parameters : list, element

Characteristics :

```
element is null? (block1 = true, block2 = false)
list is empty? (block1 = true, block2 = false)
```

6.2.2 Functionality-based approach

Meaning and semantics of the software that should be tested are known. It is harder to develop as it requires more design effort. It may result in better tests, or fewer tests that are as effective.

Functionality-Based Approach

Two parameters : list, element

Characteristics :

number of occurrences of element in list?

(0, 1, >1)

element occurs first in list?

(true, false)

element occurs last in list?

(true, false)

! For each separate characteristic, the input values that are partitioned are comprised of every input parameter that the function has. If the characteristic partitions the data only based on one parameter, the “individual dots” are still combinations of all parameters with regard to the partitioning.

6.3 Identifying Characteristics & Partitioning

To apply IDM in practical projects identifying characteristics that are used to divide the input domain into blocks is necessary. This is a highly creative engineering step as there is no cookbook or such thing to follow. Although, there are some guidelines:

- Parameter types (for interface-based approach)
- Preconditions and postconditions
- Relationships among variables
- Relationship of variables with special values (one, zero, null, blank, ...)

The program source (code) should not be used → black box; the characteristics should be based on the input domain. The program source should be used with graph or logic criteria.

The next step is to map characteristics into blocks. Strategies for identifying blocks are:

- Include valid, invalid and special values
- Explore boundaries of domains
- Sub-partition some blocks
- Include values that represent “normal use”

Finally checking for completeness and disjointness is advised.

6.4 Example: Sides of a triangle

Consider a function that classifies a triangle based on the length of its each sides. The result is an integer value r :

If $r = 1$, all sides are of different length → scalene; if $r = 2$, two sides are of equal length → isosceles; if $r = 3$, all three sides are of equal length → equilateral; if $r = 4$, it is not a valid triangle.

6.4.1 Interface-based IDM

Characteristics are defined without considering what method actually does. Below example results in a maximum of $4 * 4 * 4 = 64$ tests.

Characteristic	b_1	b_2	b_3	b_4
q_1 = "length of side 1"	greater than 1	equal to 1	equal to 0	less than 0
q_2 = "length of side 2"	greater than 1	equal to 1	equal to 0	less than 0
q_3 = "length of side 3"	greater than 1	equal to 1	equal to 0	less than 0

Example input values for partition q_1

Characteristic	b_1	b_2	b_3	b_4
Side1	(2 , 3, 1)	(1 , 1, 1)	(0 , 3, 4)	(-1 , 2, 6)

6.4.2 Functionality-based IDM

A functionality-based characterization could use the fact that the three integers represent a triangle. β -Alanin

Geometric Characterization (version 1)

Characteristic	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	scalene	isosceles	equilateral	invalid

example inputs	(4, 5, 6)	(3, 3, 4)	(3, 3, 3)	(3, 4, 8)
----------------	-----------	-----------	-----------	-----------

! Equilateral also includes isosceles.

Geometric Characterization (version 2)

Characteristic	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	scalene	isosceles, not equilateral	equilateral	invalid


A different approach would be to break the geometric characterization into four separate characteristics:

Four Characteristics for TriType

Characteristic	b_1	b_2
q_1 = "Scalene"	True	False
q_2 = "Isosceles"	True	False
q_3 = "Equilateral"	True	False
q_4 = "Valid"	True	False


6.5 Defining Coverage Criteria

Once characteristics and partitions are defined, the next step is to choose test values. *Criteria* are used to choose effective subsets.

 *All Combinations (ACoC): All combinations of blocks from all characteristics must be used.*

The number of tests is the product of the number of blocks in each characteristic: $\prod_{i=1} Q(B_i)$

! For the TriTyp (version 2) example this leads to 64 total tests.

 *Each Choice (EC): One value from each block for each characteristic must be used in at least one test case.*

This means that each block of all the questions is represented by a test once.

The number of tests is the number of blocks in the largest characteristic: $\max_{i=1} Q(B_i)$

This criterion yields fewer tests (is cheaper) than ACoC but perhaps not that effective.

Characteristic	b_1	b_2	b_3	b_4
q_1 = "length of side 1"	greater than 1	equal to 1	equal to 0	less than 0
q_2 = "length of side 2"	greater than 1	equal to 1	equal to 0	less than 0
q_3 = "length of side 3"	greater than 1	equal to 1	equal to 0	less than 0

Test set 1.
(2, 2, 2),
(1, 1, 1),
(0, 0, 0),
(-1, -1, -1)

Test set 2.
(2, 1, 0),
(-1, 2, 1),
(0, -1, 2),
(1, 0, -1)

Test set 3.
(2, 2, -1),
(-1, 1, 0),
(0, 0, 2),
(1, -1, 1)

Many ways satisfy this criterion

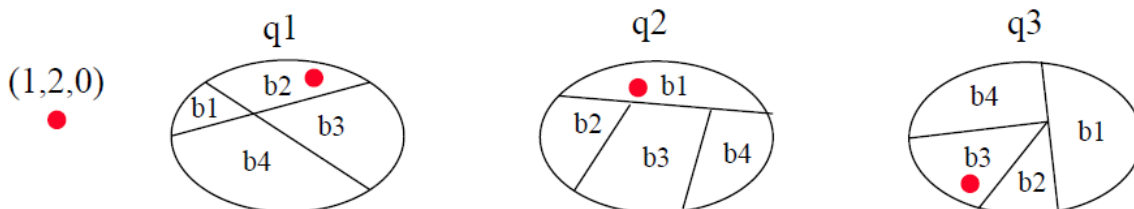
 Pairwise (PWC): A value from each block for each characteristic must be combined with a value from every block for each other characteristic.

The number of tests is at least the product of two largest characteristics:

$$Max_{i=1} Q(B_i) * Max_{j=1, j \neq i} Q(B_i)$$


TriTyp - second interface-based characterization

Characteristic	b_1	b_2	b_3	b_4
q_1 = “length of side 1”	greater than 1	equal to 1	equal to 0	less than 0
q_2 = “length of side 2”	greater than 1	equal to 1	equal to 0	less than 0
q_3 = “length of side 3”	greater than 1	equal to 1	equal to 0	less than 0



2, 2, 2	2, 1, 1	2, 0, 0	2, -1, -1
1, 2, 1	1, 1, 0	1, 0, -1	1, -1, 2
0, 2, 0	0, 1, -1	0, 0, 2	0, -1, 1
-1, 2, -1	-1, 1, 2	-1, 0, 1	-1, -1, 0


💡 Testers sometimes recognize that certain values are important. One block is chosen from each characteristic, that is supposed to be the most important one (supposedly most likely to uncover faults in the system). This uses domain knowledge of the program.


 **Base Choice (BC):** A base choice block is chosen for each characteristic, and a base test is formed by using the base choice for each characteristic. Subsequent tests are chosen by holding all but one base choice constant and using each non-base choice in each other characteristic.

The number of tests is one base test + one test for each other block in each partition:

$$1 + \sum_{i=1}^Q (B_i - 1)$$

Only one solution	(2, 2, 2), (2, 2, 1), (2, 1, 2), (1, 2, 2), (2, 2, 0), (2, 0, 2), (0, 2, 2), (2, 2, -1), (2, -1, 2), (-1, 2, 2)
-------------------	--

 **Multiple Base Choice (MBC):** One or more base choice blocks are chosen for each characteristic, and base tests are formed by using each base choice for each characteristic. Subsequent tests are chosen by holding all but one base choice constant for each base test and using each non-base choices in each other characteristic.

 Always keep a pair of base choices constant and only set values for non-base choices of the other partition.

If there are M base tests and m_i base choices for each characteristic:

$$M + \sum_{i=1}^Q (M * (B_i - m_i))$$

For TriTyp (second interface-based characterization):

Base:	2, 2, 2	2, 2, 0	2, 0, 2	0, 2, 2
		2, 2, -1	2, -1, 2	-1, 2, 2
Base:	1, 1, 1	1, 1, 0	1, 0, 1	0, 1, 1
		1, 1, -1	1, -1, 1	-1, 1, 1

6.6 Constraints

Some combinations of blocks are infeasible. Generally, there are two types of constraints:

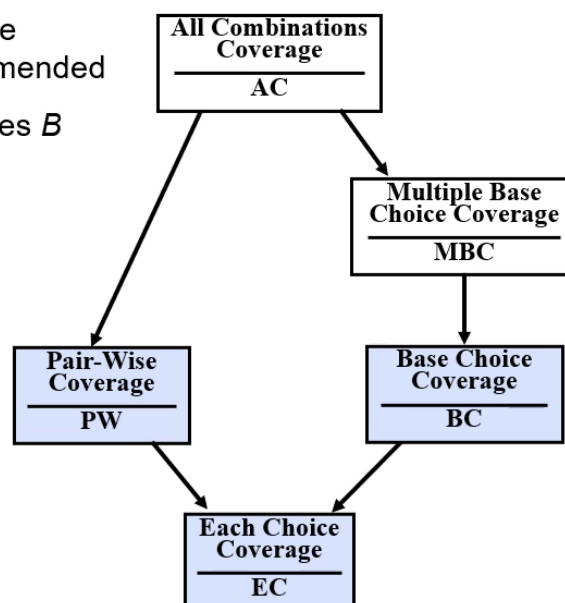
- A block from one characteristic *cannot* be combined with a specific block from another
- A block from one characteristic *can only* be combined with a specific block from another

Handling constraints depends on the criterion used. For *ACoC* and *PW*: drop the infeasible pairs; for *BC* and *MBC*: change a value to another non-base choice to find a feasible combination.

6.7 Summary

Coverage criteria in blue background are recommended

$A \longrightarrow B = A$ subsumes B



7 Syntactic Coverage

Software artifacts often follow strict syntax rules. The syntax is often expressed as a grammar in a meta-language such as BNF. Tests are created with two general goals: *cover* the syntax in some way and *violate* the syntax.

Allowable inputs are defined by grammars such as regular expressions. For example:

$$(G\ s\ n\ |\ B\ t\ n)^*$$

- G and B could represent commands, methods, or events
- S , t and n can represent arguments, parameters, individual values or sets of values

* = closure operator
(zero or more occurrences)

$|$ = choice operator
(either one can be selected)

💡 Grammars, such as the expression above, define how to create acceptable strings by applying the operators (closure “*” and choice “|”). They essentially define derivation rules that can be used to create acceptable strings. When a string is acceptable and satisfies these rules or can be created using these rules, it is set to be *in the grammar*.

When trying to cover the grammar in some way, a test case is a string, or sequence of strings, that is *in the grammar*.

7.1 Formal Grammar: BNF

The symbol “ $::=$ ” can be read as “can be rewritten as”.

Stream $::=$ action*

action $::=$ actG | actB

actG $::=$ “G” s n

actB $::=$ “B” t n

s $::=$ digit¹⁻³

t $::=$ digit¹⁻³

n $::=$ digit² “.” digit² “.” digit²

digit $::=$ “0” | “1” | “2” | “3” | “4” | “5” | “6” |
“7” | “8” | “9”

Start symbol

Non-terminal symbol

Terminal symbol

Production Rule

💡 These grammars are useful for:

Recognizer: determine when strings are in the grammar → parsing

Generator: derive new strings that are in the grammar

Stream $::=$ action *

$::=$ action action *

$::=$ actG action*


$::=$ G s n action*

$::=$ G digit¹⁻³ digit² . digit² . digit² action*


$::=$ G digitdigit digitdigit.digitdigit.digitdigit action*

$::=$ G 26 08.01.90 action*

7.2 Coverage Criteria


 *Terminal Symbol Coverage (TSC): TR contains each terminal symbol t (at least once) in the grammar G .*

💡 Each terminal symbol must appear in at least one of the strings constituting the set of tests. The maximum number of tests for TSC is the number of terminal symbols in the grammar → 13 in the stream grammar.

 *Production Derivation Coverage (PDC): TR contains each production p (at least once) in the grammar G .*

💡 Each production rule *choice* must be used in the derivation of at least one of the strings constituting the set of tests. PDC subsumes TSC. The maximum number of tests for PDC is bound by the number of productions → 18 in the stream grammar.

The most comprehensive criteria requires the test set to contain every string in the grammar:

 *Derivation Coverage (DC): TR contains every possible string that can be derived from the grammar G .*

DC subsumes all other coverage criteria

! Attention:

- The maximum number of tests required for TSC is the number of terminal symbols
 - 13 in the stream grammar
- The number of PDC tests is bound by the number of productions
 - 18 in the stream grammar
- The number of DC tests depends on the details of the grammar
 - in the stream grammar the number is infinite !

7.3 Mutations


Grammars describe both valid and invalid strings. The notion of *mutants* can be used to systematically cover strings that violate the grammar. A mutant is a *variation* of a valid string. Mutants may be valid or invalid strings. Mutations are created by means of *mutation operators*.


- So called *ground strings* are (valid) strings in the grammar
- A mutation operator is a rule that specifies syntactic variations of strings generated from a grammar

- A *mutant* is a string that is created by one application of a mutation operator. A mutant can either be in the grammar (valid) or very close to being in the grammar (invalid).

<u>Ground String</u>	<u>Valid Mutant</u>	<u>Invalid Mutant</u>
G 26 08.01.90	B 26 08.01.90	7 26 08.01.90
B 22 06.27.94	B 45 06.27.94	B 22 06.27.0

The key to mutation testing is the mutation operators. Well-designed operators lead to effective tests. This results in two simple coverage criteria:

 **Terminal Symbol Coverage (MOC):** For each mutation operator, TR contains exactly one requirement, to create a mutated string m that is derived using the mutation operator.

 **Mutation Production Coverage (MPC):** For each mutation operator, and each production that the operator can be applied to, TR contains exactly the requirement to create a mutated string from that production.


<u>Ground Strings</u>	<u>Mutation Operators</u>
G 25 08.01.90	• Exchange <i>actG</i> and <i>actB</i>
B 21 06.27.94	• Replace a digit with "G"


<u>MOC</u>	<u>MPC</u>
B 25 08.01.90	B 25 08.01.90 G 21 06.27.94
B 2G 06.27.94	G 25 08.01.9G B 2G 06.27.94
	G G5 08.01.90 B 21 06.27.9G
	G 2G 08.01.90 B 21 0G.27.94

7.4 Program Mutation

A program is essentially a string and thus can also be subject to mutation. Program-based mutation testing aims to mutate programs to see if a test will detect a difference in behavior to the original.

- Grammar = programming language syntax
- Ground strings = original program (compilation unit)
- Mutated strings = alternative program (must be in the grammar)

 **Killing mutants:** Given a mutant m of a derivation D and a test t , t is said to kill m if and only if the output of t on D is different from the output of t on m .

 **Mutation Coverage (MC):** For each mutation m of a derivation D , which is not behaviorally equivalent to D , TR contains exactly one requirement, to kill m .

💡 Mutation Coverage requires enough tests to kill all non-equivalent mutants. If MC is not possible, the proportion of non-equivalent mutants killed is called the *mutation score*.

The diagram shows a code snippet for a `CalculatorTest` class. The code is as follows:

```

6 public class CalculatorTest {
7
8     private static class Calculator {
9
10        int add(int a, int b) {
11            // apply arithmetic mutation operator
12            // '+' => '-' : a - b
13            return a + b;
14        }
15    }
16
17    @Test
18    public void test_add() {
19        Calculator cut = new Calculator();
20
21        int actual = cut.add(1, 1);
22
23        // if mutation operator applied, actual == 0
24        assertEquals(2, actual);
25    }
26 }

```

Annotations and arrows:

- A box labeled **Calculator** adding two Integers points to the `Calculator` class.
- A box labeled **Mutation Operator** Change '+' to '-' points to the comment in the `add` method.
- An arrow labeled **1) mutate** points to the `return a + b;` line.
- An arrow labeled **2) exhibit different behavior i.e. test failed → mutant killed** points to the `assertEquals(2, actual);` line.

📄 When the system creates a new instance of the calculator in line 19, it uses the mutation.

The number of tests requires for mutation coverage depends on the number of mutants that have to be killed, which in turn depends on two things:

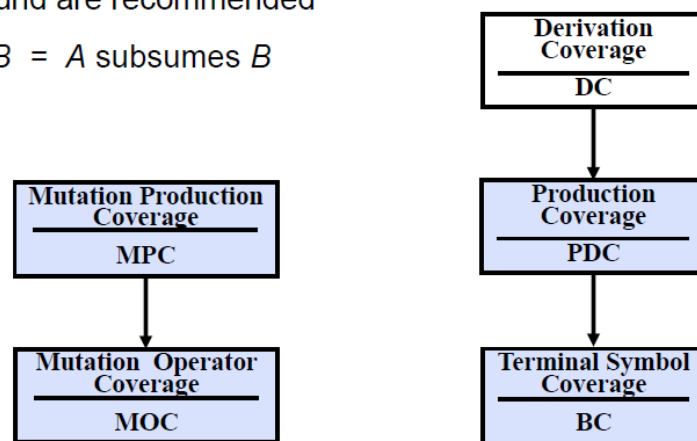
- The syntax of the artifact being mutated
- The mutation operators

❗ Mutation testing is very difficult to apply by hand. It is very effective, though. It can also be used to evaluate the quality of a set of tests as well as the quality of programs.


7.5 Summary

Coverage criteria in blue background are recommended

$A \rightarrow B = A$ subsumes B



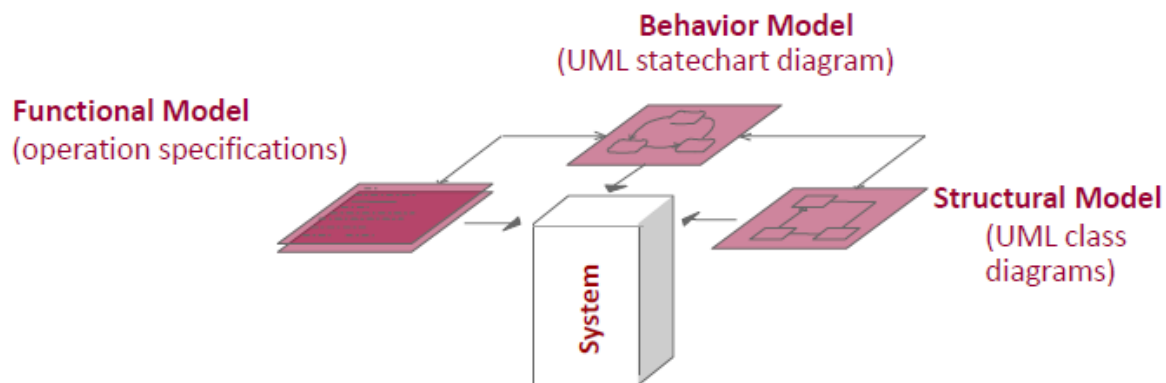
8 Kobra

 The Kobra-approach applies common methods that are used in most *model-based software specification techniques*. The basic goal of a system specifications in the Kobra-method is describing the externally visible properties of a system. Said specification defines the systems'

- Supplied services (interface)
- Imported services (used components)

These interface descriptions can be thought as a contract between the system and its clients. Ultimately, the system specification defines the requirements the realization of the system has to fulfill.

$view \Leftrightarrow model$
 $system \Leftrightarrow component$



*Top-part of the model constellation diagram, only including the **primary specification views/models***


Kobra very strictly and systematically applies the principle that *models are views* of a software system, while most other methods take a more relaxed view about models. In Kobra the content and form of a view should be as precisely defined as possible


- Strong concept of completeness
- Strong concept of correctness


This allows that diagrams can be checked against each other to check their mutual correctness and completeness.

8.1 Primary Specification Views

Present the most important information about the system:





 **Structural view:** describes the data manipulated by the component, its environment, and any externally visible structure. → via ≥ 1 class diagrams and ≥ 0 object diagrams

 **Functional view:** describes the computations performed by the component. → via 1 operation specification for each operation.

 **Behavioral view:** describes the states exhibited by the component and the events that change them → via ≥ 1 state chart diagrams

8.2 Auxiliary Specification Artifacts

Additional and supporting artifacts:

-  *non-functional requirements specification*: describes the quality characteristics
-  *non-functional requirements specification*: describes the desired quality thresholds and related quality factors.
-  *(data-) dictionary: tables of model entities and their role*
-  *test cases: support functional testing*

8.3 Operation Specification

The functional model of a Kobra-component is composed of multiple operation specifications. Each operation specification defines the behavior of the corresponding operation in terms of state changes and output events. The whole point of operation specifications is that they specify the behavior of system operations declaratively (“what are the effects of this operation?” not “how are they achieved?” → realization). The state of the system is represented by the set of objects and the relationships between them. A system operation may

- Create and/or delete instances of a class
- Change an attribute value of an existing object
- Add or delete objects from relationships (create/destroy links between objects)
- Send an event to a component

Operations are specified in the functional model by means of pre- and post-conditions. For specifications, operations are treated as black boxes. Nothing is said about an operation’s intermediate states.



Name	Name of the operation.
Description	Identification of the purpose of the operation, followed by an informal description of the normal and exceptional effects.
Rules	Properties that constrain the realization and implementation of the system.
Receives	Information input to the operation by the invoker.
Returns	Information returned to the invoker by the operation.
Sends	Signals which the operation sends to imported systems. These can be events or operation invocations.
Reads	Externally visible information accessed by the operation.
Changes	Externally visible information changed by the operation.
Assumes	Precondition on the externally visible state of the system and on the inputs (in receives clause) that must be true for the system to guarantee the post condition (result clause).
Result	Strongest post condition on the externally visible properties of the system and the returned entities (returns clause) that become true after execution of the operation with true assumes clause.

Operation specification template: “Name” and “Result” are mandatory clauses !

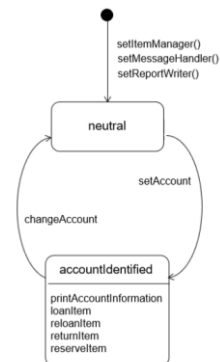
Well-formed Operation Specifications

In general, the result clause may have several subclauses. Each subclause must also be purely declarative. The meaning of the result clause must not depend on the order of the subclauses (no flow of control). A specification is satisfiable if, for all initial values satisfying the assumes clause, there exist final values satisfying the result clause.

8.4 Behavioral Model

It takes form as a UML state chart diagram and describes how a system behaves in response to external stimuli. Three important concepts are related in the behavioral model:

- *Events* are invisible occurrences that have a location in space and time
- *Operations* are encapsulated units of functionality associated with systems
- *States* represent a period of time in between events that determine how the system will respond to the subsequent event.

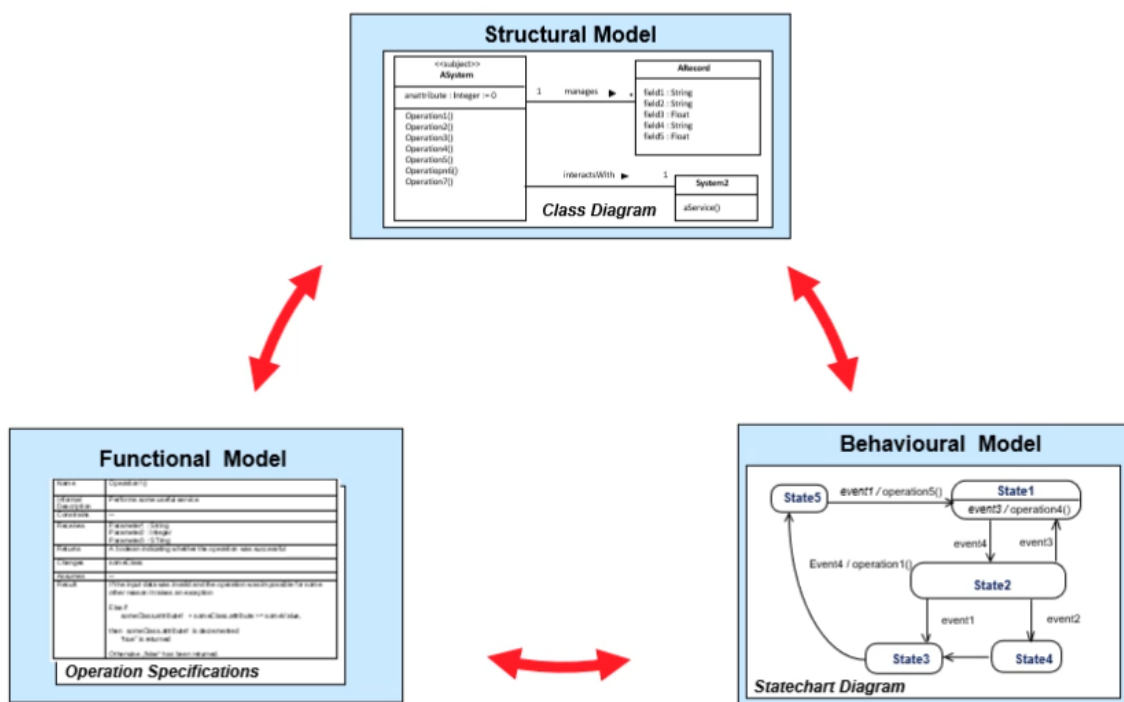


In a Kobra behavioral view the events are the invocations of the systems operations. The set of transitions from a state define the operations that are possible in that state

- External transitions move the system into a new state
- Internal transitions leave the system in the same state

8.5 Consistency

There is a set of very strict and well-defined intermodel constraints/consistency rules which have to be satisfied for the overall specification as a whole to be well-formed.



8.5.1 Intra-Diagram Consistency Rules

! Class Diagrams

- Only components can have an operation compartment
- If the subject component has an operation compartment, it must list all of the operations of the system
- If an acquired component has an operation compartment it should only list the operations that the subject component invokes

! State chart diagrams

- The outgoing transitions of each state must be disjoint (i.e., the behavioral model must be deterministic)

! Operation Specifications

- There must be one operation specification for every operation belonging to the component under specification
- Assumes and result clauses must be Boolean expressions
- Terms in sends, receives, returns and changes clauses should refer to items mentioned in the assumes and result clause.

8.5.2 Inter-diagram Consistency Rules

Class Diagrams \Leftrightarrow Operation specifications

These models must basically agree on the nature of the entities that exist in order for operations to do their work. In particular this means:

- All classes, attributes and relationship referred to in an operation specification must be in the class diagram.
- Every class in the class diagram should appear in at least one operation specification
- Every attribute appearing in the class diagram should appear in at least one operation specification
- Every operation listed in the operation compartment of a system other than the subject must appear in at least one operation specification

Objects Diagrams \Leftrightarrow Class Diagrams

Object diagrams can only define instances of entities defined in the class diagram of the same system specification

Class Diagrams \Leftrightarrow State chart Diagrams

- Attributes referred to in a state chart diagram must appear in the class representing the subject and must match in name and type
- conditions within a state chart diagram may be defined only in terms

Object Diagrams \Leftrightarrow Operation Specifications

Instance names used in the operation specifications must match those in the object diagram.

Operation Specifications \Leftrightarrow State chart Diagrams

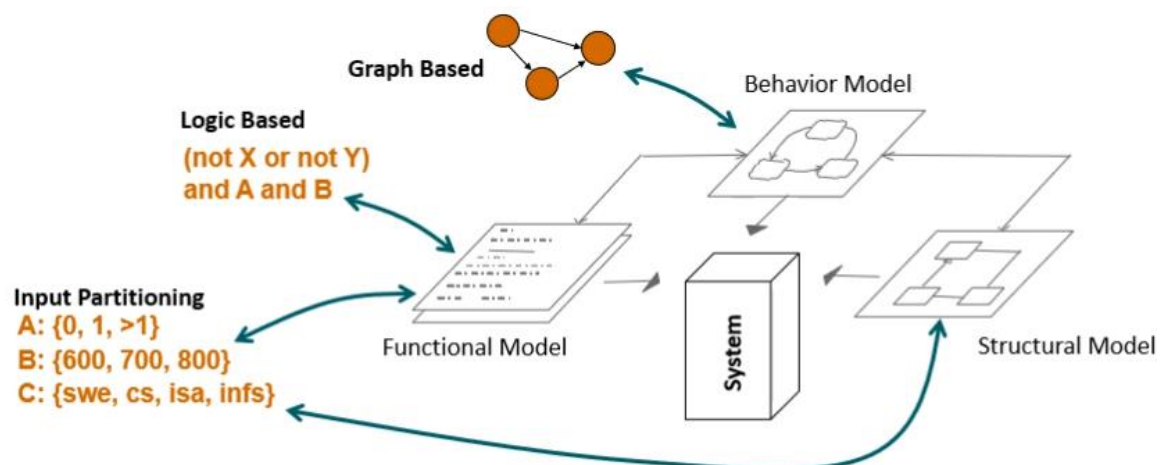
These two models must basically agree on the states, events and operations of the system. In particular, this means-

- operation specifications may only mention states and events appearing in the state chart diagram
- state chart diagrams may only mention operations specified in the functional model (i.e., which have an operation specification)

9 Specification-based testing

Software specifications need to be written in readable, precise, and easy-to-understand way. The need to be accompanied by well-defined and comprehensive test definitions. Kobra specifications provide an ideal basis for designing black-box tests based on the provided and required properties of components.

- Behavioral model \Rightarrow graph-based criteria
- Functional model \Rightarrow logic-based criteria and input space partitioning
- Structural model \Rightarrow input space partitioning



9.1 Kobra-based Test Design Method

1. Apply graph-based criteria
2. Apply logic-based criteria
3. Apply data partitioning criteria
4. Consolidate all requirements and define test sets to fulfill them
5. Define concrete tests to realize the test sets in a minimal way

💡 This will result in a large number of tests. A simplification of these tests is possible that does not significantly reduce the effectiveness of these tests. This simplification is based on:

- Changer operations
 - Sometimes change the state of the system after execution
 - Also known as constructor operations or setter operations (in programming l.)
- Inspector operations
 - Never change the state of the system after execution
 - Never have a changes clause in their operation specification
 - Also known as getter operations (in programming languages)

9.2 Mapping Behavior Model to a formal graph

Each state is mapped to a node, S_i , called a state node. Each external transition is mapped to a node, T_i , called a transition node, plus two edges from the initial state to T_i and from T_i to the final state.



Each internal transition caused by a *changer operation* is mapped to a node (a transition node T_i) plus two edges, one from the initial state to T_i and from T_i back to the final state.

