2024 Spring

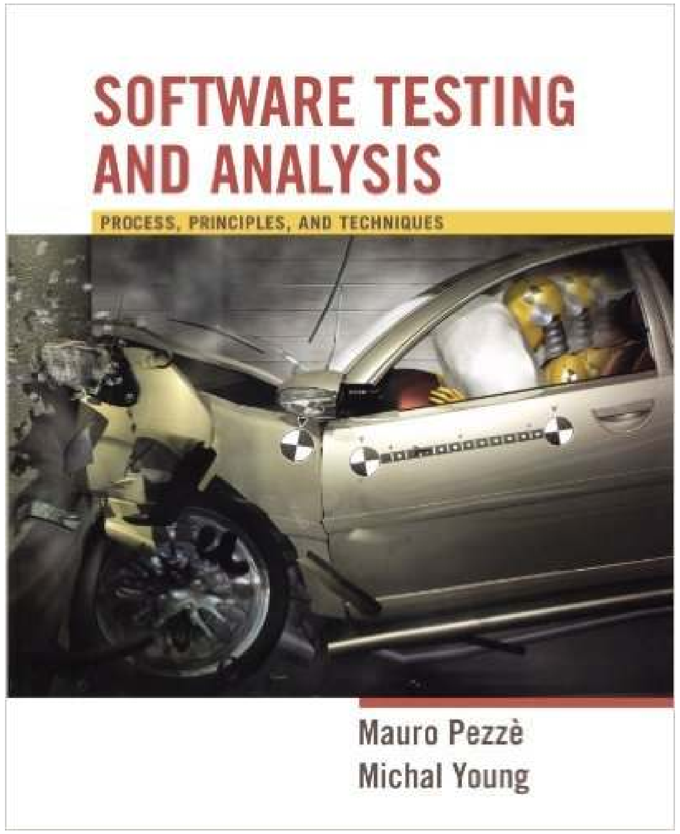
Software Verification & Validation

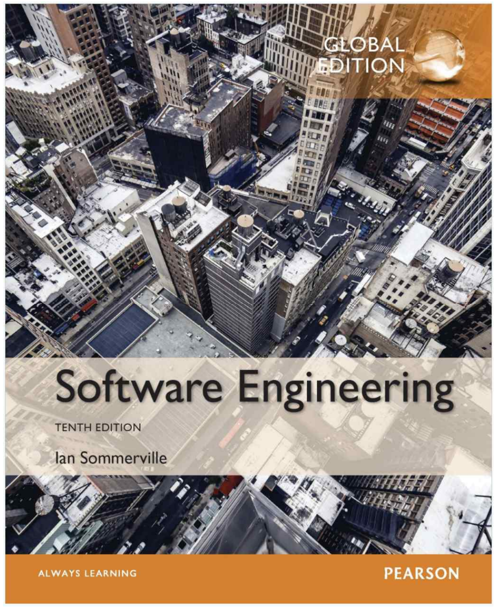
JUNBEOM YOO

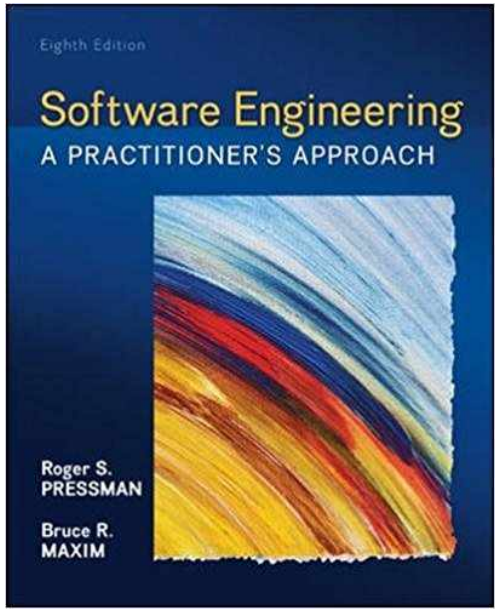
KONKUK University

http://dslab.konkuk.ac.kr

Text and References


SOFTWARE TESTING AND ANALYSIS
PROCESS, PRINCIPLES, AND TECHNIQUES
Mauro Pezzè
Michal Young





Contents

Code Review

Clean Code

Refactoring

코드 기반 테스트

테스트 계획서

TDD

구조 기반 테스트

소프트웨어 테스트

Boundary Value 테스트

3점 접합

TFD

블랙박스 테스트

구조 테스트

테스트 실행

CI / CD

모델 기반 테스트

테스트 결과 보고서

N-Pairwise 테스트

테스트 Stub

기능 테스트

화이트박스 테스트

Static Code Analysis

스펙 기반 테스트

CTIP(Continuous Test & Integration Platform)

테스트 케이스 자동 생성

테스트 오라클

Code Review

테스트 케이스

Category-Partitioning 테스트

테스트 데이터

요구공학

Coverage Criteria

테스트 자동화 도구

xUnit

테스트 데이터 자동 생성

테스트 Scaffolding

시스템 테스트 계획서

테스트 드라이버

Coverage 측정

테스트 명세서

gTest

QAS

HW/SW Co-테스트

비기능/품질 테스트

Cyclomatic Complexity

시뮬레이션 기반 테스트

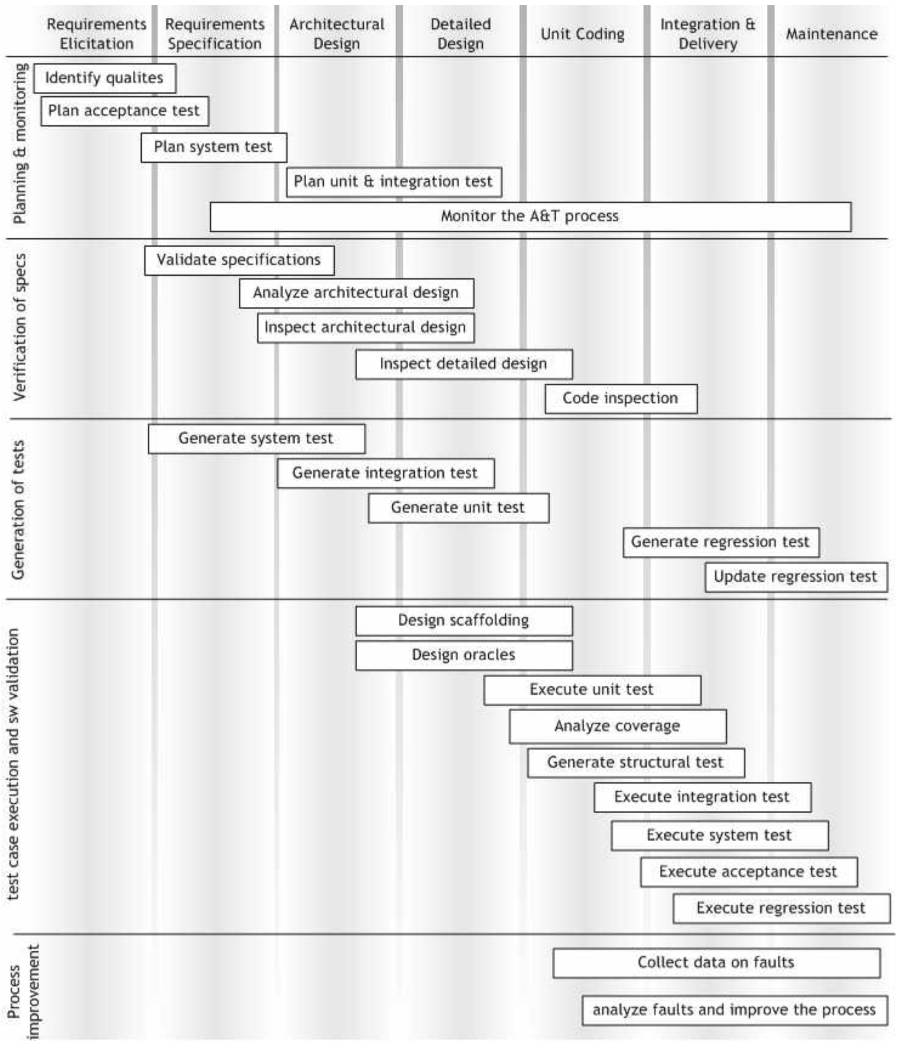
FUNDAMENTALS OF SOFTWARE TESTING & ANALYSIS

Engineering Processes

- **All engineering processes** have two common activities.
  - Construction (개발)
  - Checking (검수, 감리)

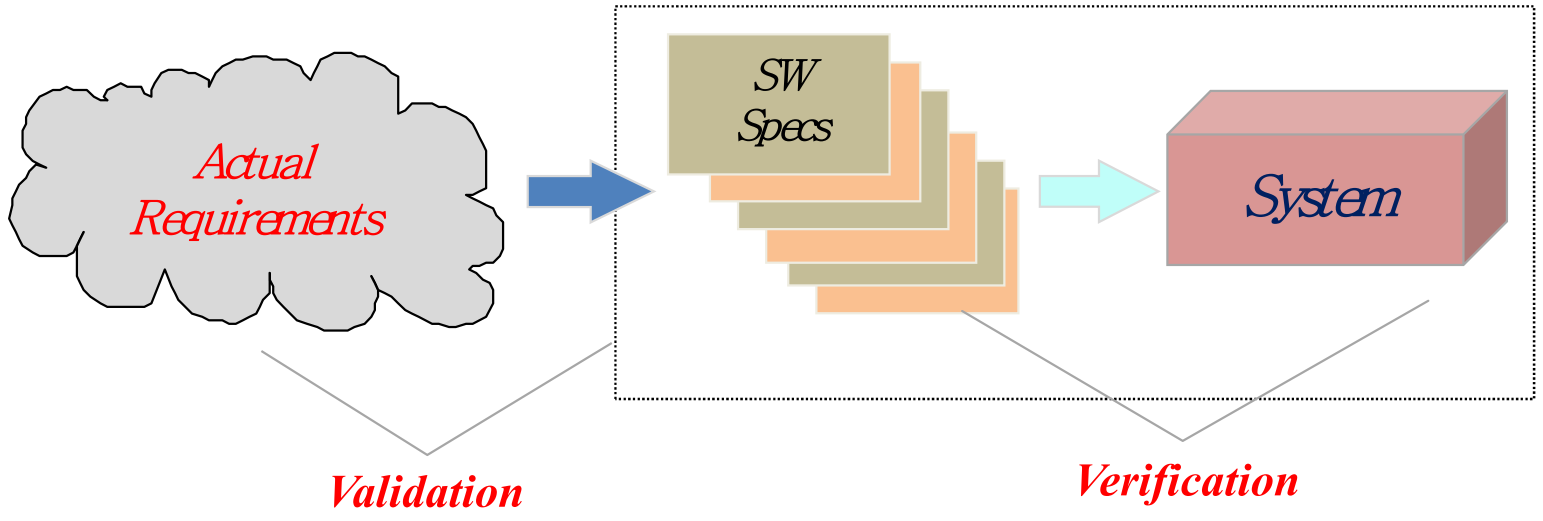
- **Software engineering** (purpose: construction of high-quality software)
  - Construction (= Development)
  - **Verification** $\leftarrow$ **Our Concern !**

Verification Activities : An Example of Testing Activities



Verification and Validation (V&V)

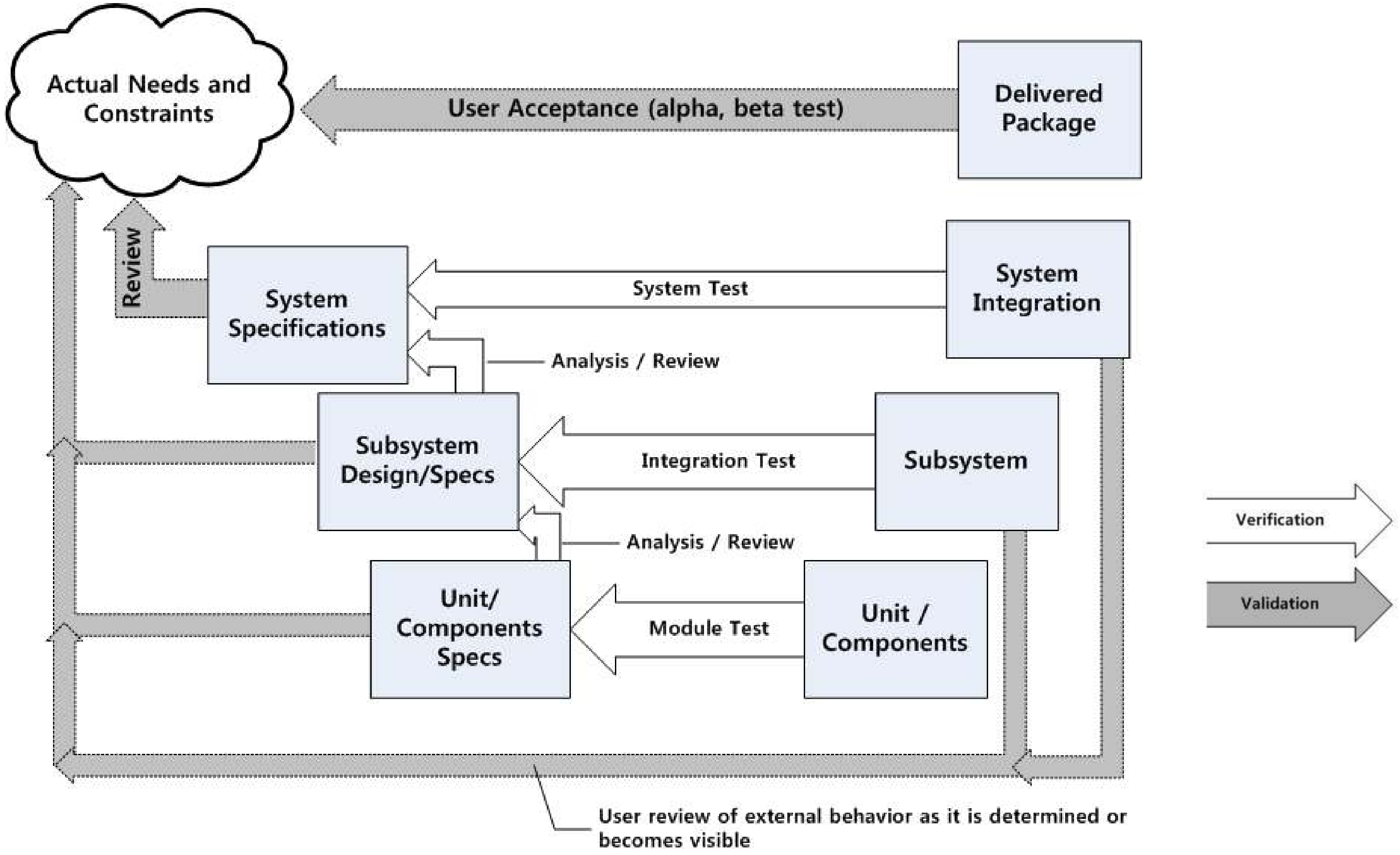
- **Validation**: Are we building the right software?
  - *"Does the software system meet the user's real needs?"*
- **Verification**: Are we building the software, right? *(with respect to requirements specification)*
  - *"Does the software system meet the requirements specifications?"*



V&V Depends on Specifications

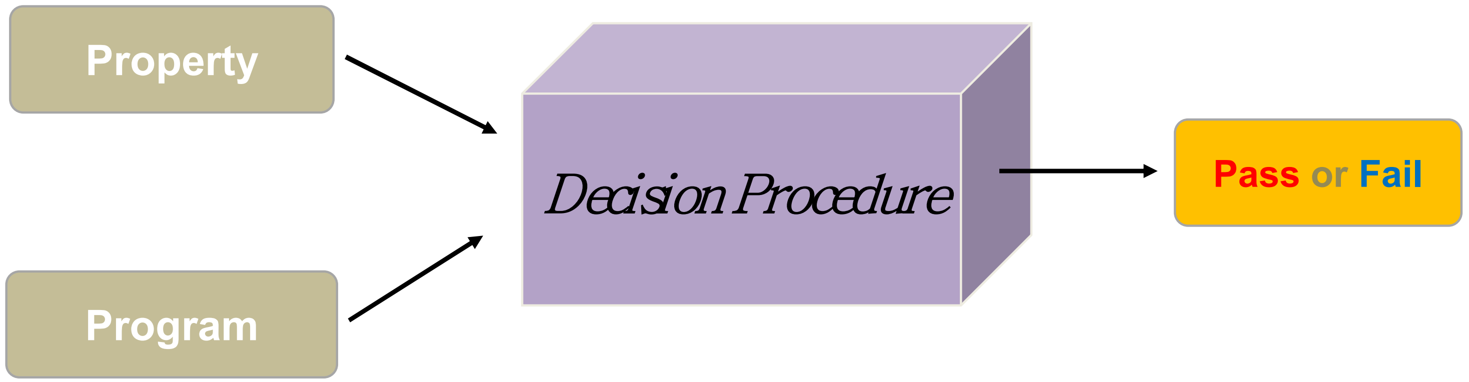
- Unverifiable (but validatable) specification
  - *"If a user presses a request button at floor i, an available elevator must arrive at floor i soon."*
- Verifiable specification:
  - *"If a user presses a request button at floor i, an available elevator must arrive at floor i within 30 seconds."*

V-Model of V&V Activities



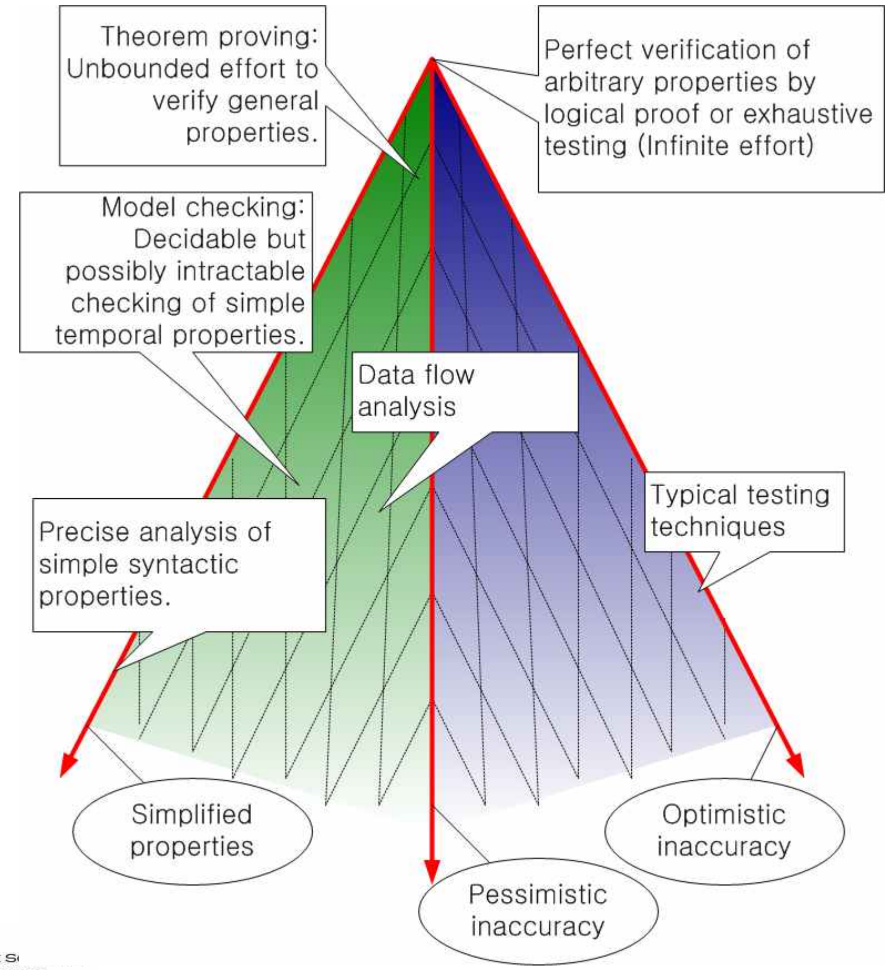
Undecidability of Correctness Properties

- **Correctness properties** are <u>not decidable</u>.
  - **Halting problem** is embedded in almost every property of interest.



In computability theory, the ***halting problem*** *is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running, or continue to run forever. Alan Turing proved in 1936 that a general algorithm to solve the halting problem for all possible program–input pairs cannot exist.*

3 Dimensions of STA Activities


Theorem proving: Unbounded effort to verify general properties.
Perfect verification of arbitrary properties by logical proof or exhaustive testing (Infinite effort)
Model checking: Decidable but possibly intractable checking of simple temporal properties.
Data flow analysis
Typical testing techniques
Precise analysis of simple syntactic properties.
Simplified properties
Pessimistic inaccuracy
Optimistic inaccuracy

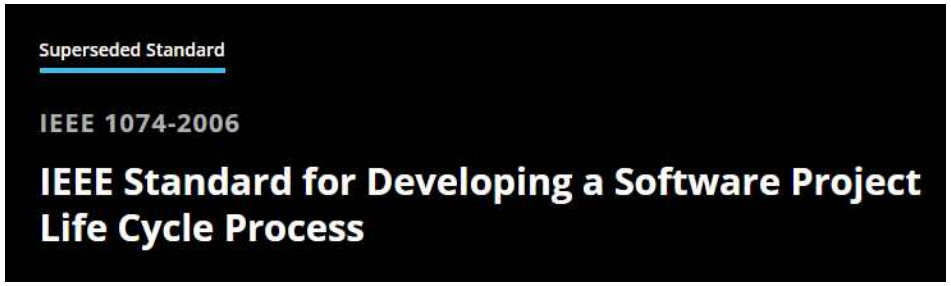
- **Optimistic Inaccuracy**
  - We may accept some programs that do not possess the property.
  - It may not detect all violations.
  - **Testing**
- **Pessimistic Inaccuracy**
  - Not guaranteed to accept a program even if the program does possess the property being analyzed, because of false alarms
  - **Static Code Analysis**
- **Simplified Properties**
  - It reduces the degree of freedom by simplifying the property to check.
  - Theorem Proving, **Model Checking**

Software Quality

- Qualities cannot be added after development.
  - **Quality** results from a set of inter-dependent activities.
    - Quality depends on every part of the software process.
  - **Quality assurance** is not a phase, but a life-style.
    - Testing and analysis activities occur from early in requirements engineering through delivery and subsequent evolution.

- An essential feature of software development processes is

"Software test and analysis is thoroughly integrated into development processes."


Superseded Standard

IEEE 1074-2006

IEEE Standard for Developing a Software Project Life Cycle Process

Software Quality Process

- **Quality process**
  - **A set of activities and responsibilities** focusing on ensuring adequate dependability concerned with project schedule or with product usability
  - **A&T planning** is Integral to the quality process.
    - **Quality goals** can be achieved only through careful A&T planning.
    - **Selects and arranges** STA activities to be as cost-effective as possible
    - Should balance several STA activities across the whole development process

- Quality process provides **a framework** for
  - Selecting and arranging *STA activities* and considering interactions and trade-offs *with other important goals.*

Active Standard

IEEE 730-2014

IEEE Standard for Software Quality Assurance Processes

An Example of Other Important Goals

- *High Dependability* vs. *Time to Market*

- **Critical medical devices**
  - Better to achieve ultra-high dependability on a much longer schedule than a reasonably high degree of dependability on a tight schedule

- **Mass market products**
  - Better to achieve a reasonably high degree of dependability on a tight schedule than to achieve ultra-high dependability on a much longer schedule

A&T Plan

- A comprehensive description of the quality process that includes:
  - Objectives, goals and scope of A&T activities
  - Documents and other items that must be available
  - Items to be tested
  - Features to be tested and not to be tested
  - *Analysis and test activities*
  - *Staff* involved in A&T
  - Constraints
  - Pass and fail criteria for Test
  - *Schedule*
  - Deliverables
  - Hardware and software requirements
  - Risks and contingencies

Quality Goals

- Goal must be further refined into a clear and reasonable set of objectives.

- **Product quality** : goals of software quality engineering
- **Process quality** : means to achieve the goals (*i.e.*, product quality)

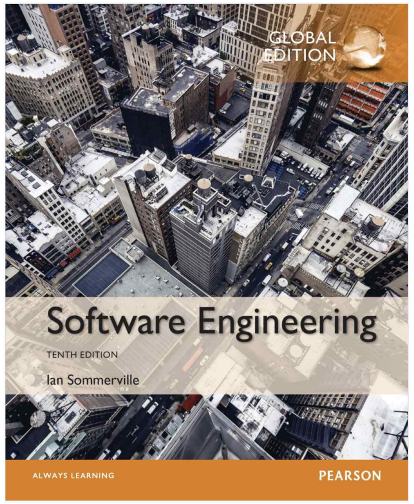
- Product qualities
  - Internal qualities: invisible to clients
    - Maintainability, Flexibility, Reparability, Changeability
  - External qualities: directly visible to clients
    - Usefulness
      - Usability, Performance, Security, Portability, Interoperability
    - **Dependability**
      - Correctness, Reliability, Safety, Robustness
      - Availability, Reliability, Safety, Security

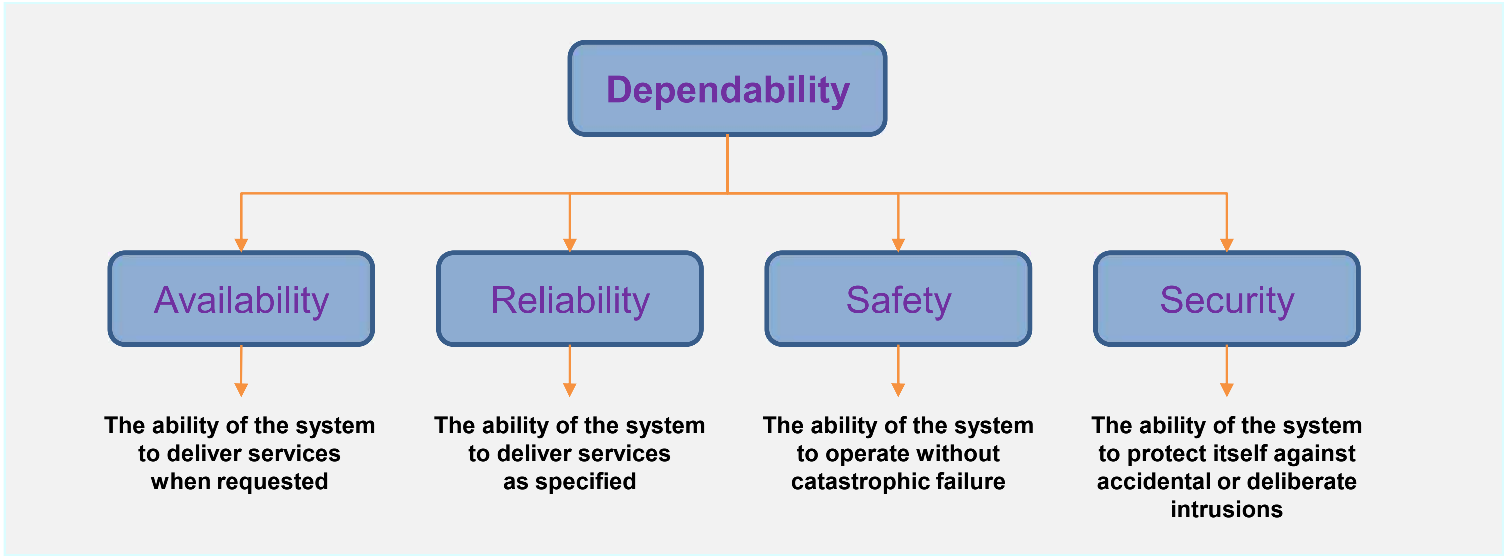
ISO/IEC 25010:2011
Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models

Dependability

- **Dependability** = trustworthiness to a system
  - Dependable system is a system that is trusted by its users.

- Principal dimensions of dependability
  - Availability, Reliability, Safety, Security
  - Others are Reparability, Maintainability, Survivability, Error tolerance, etc.

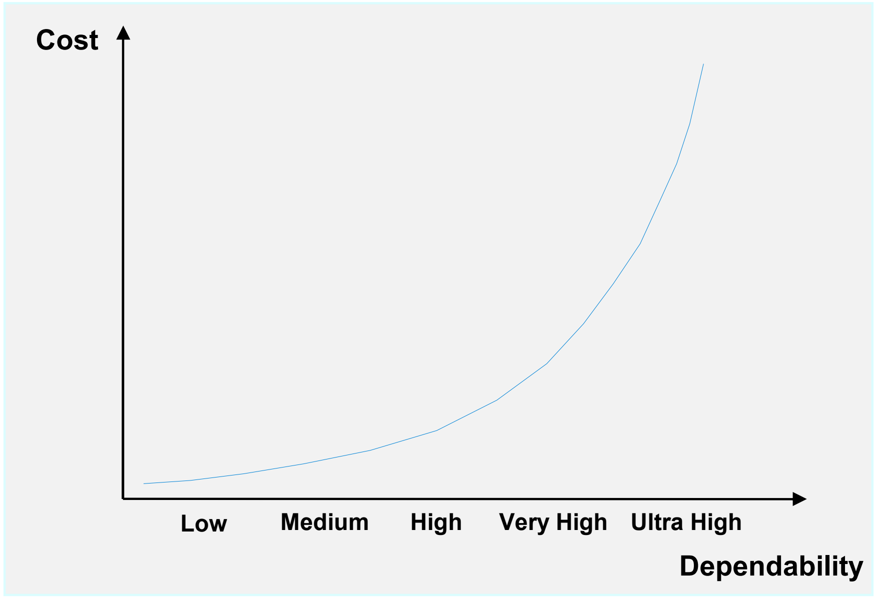



Dependability
Availability
Reliability
Safety
Security
The ability of the system to deliver services when requested
The ability of the system to deliver services as specified
The ability of the system to operate without catastrophic failure
The ability of the system to protect itself against accidental or deliberate intrusions

DEPENDABLE SOFTWARE LABORATORY

Dependability Costs

- **Dependability costs**
  - Cost to achieve the required dependability
  - Tend to increase exponentially as <u>required levels of dependability</u> increase
    - More expensive development techniques and hardware are required.
    - Increased testing and system validation are also required.



Dependability Economics

- **Dependability Economics**
  - *"It may be <u>more cost effective</u> to accept untrustworthy systems and pay for failure costs, because of very high costs of dependability achievement."*

- However, it depends on
  - Social and political factors
    - Poor reputation for products may lose future business.
  - System types
    - For business systems (custom SW), modest levels of dependability may be adequate.

Dependability Properties

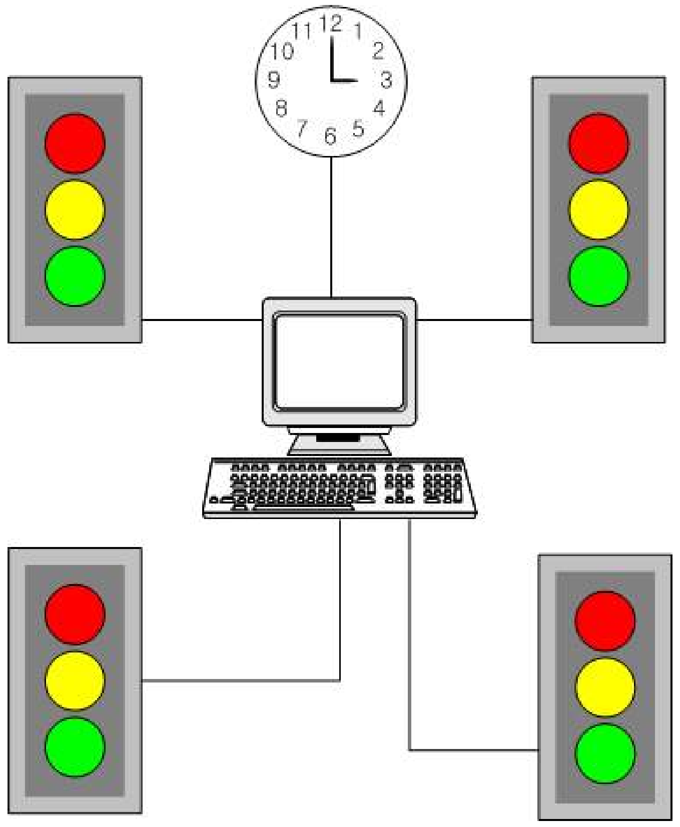
- **Correctness**
  - A program is correct if it is consistent with its specification.
  - Seldom practical for non-trivial systems
- **Reliability**
  - Likelihood of correct function for some “unit” of behavior
  - Statistical approximation to correctness (100% reliable = correct)

Normal Operation

- **Safety**
  - Concerned with preventing certain undesirable behavior, called hazards
  - “Catastrophes should never happen.”
- **Robustness**
  - Providing acceptable (degraded) behavior under extreme conditions
  - Fail softly

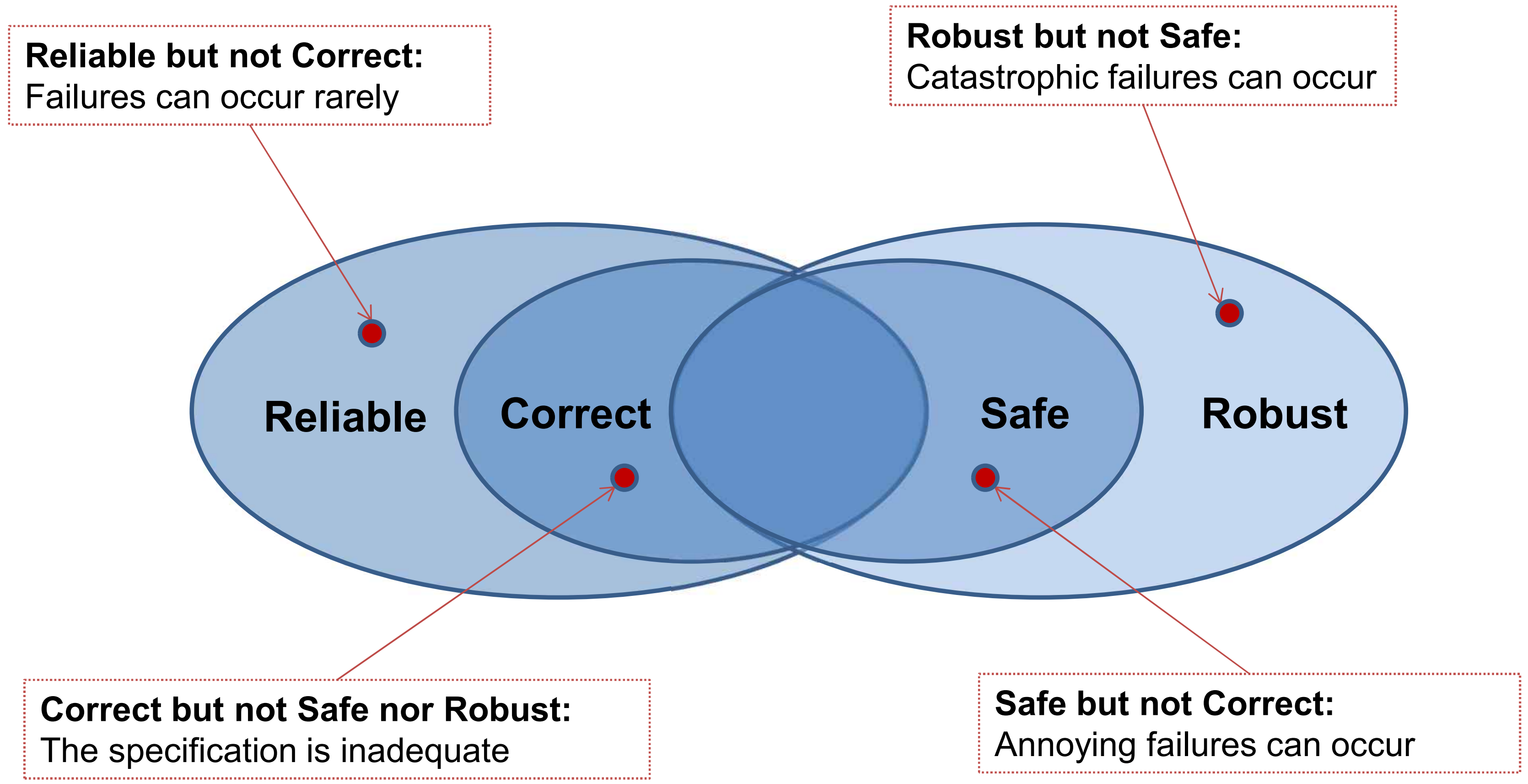
Abnormal Operation & Situation

An Example of Dependability Properties



- **Correctness, Reliability :**
  - Let traffic pass according to correct pattern and central scheduling
- **Robustness, Safety :**
  - Provide degraded function when it fails
  - Never signal conflicting greens
    - Blinking red / blinking yellow is better than no lights.
    - No lights is better than conflicting greens.

Relationship among Dependability Properties


Reliable but not Correct:
Failures can occur rarely
Robust but not Safe:
Catastrophic failures can occur
Reliable
Correct
Safe
Robust
Correct but not Safe nor Robust:
The specification is inadequate
Safe but not Correct:
Annoying failures can occur

SQA Engineers

- **SQA** : Software Quality Assurance

- A pretty important engineer for assuring SW quality consistently
  - Managing quality process
  - **Selecting appropriate activities for each project/organization**
    - Preparing, Monitoring, Evaluating, Improving
  - Keeping balance between quality and other goals (time to market)
  - Experienced well
  - **Working on the rock of deep/solid/accurate knowledge on STA activities**
    - Testing
    - Static Analysis
    - Model Checking
    - + Review

Basic Questions on Software Verification

1. When do verification and validation start and end?

2. What techniques should be applied?

3. How can we assess the readiness of a product?

4. How can we ensure the quality of successive releases?

5. How can the development process be improved?

1. When Do Verification and Validation Start and End?

- For an example, **Test**
  - A widely-used V&V activity
  - Usually known as a last activity in software development process, but not the last activity is “test execution”
  - Test execution is a small part of V&V process

- V&V start as soon as we decide to build a software product, or even before.

- V&V last far beyond the product delivery as long as the software is in use, to cope with evolution and adaptations to new conditions.

Early Start: From Feasibility Study

- Feasibility study of a new project must take into account required qualities and their impact on the overall cost.

- Quality related activities include
  - **Risk analysis**
  - **Measures** needed to assess and control quality at each stage of development
  - **Assessment** of the impact of new features and new quality requirements
  - Contribution of quality control activities to development cost and schedule

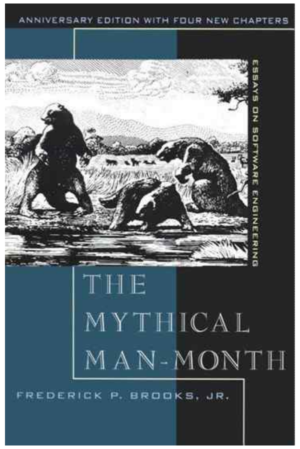
Long Lasting: Beyond Maintenance

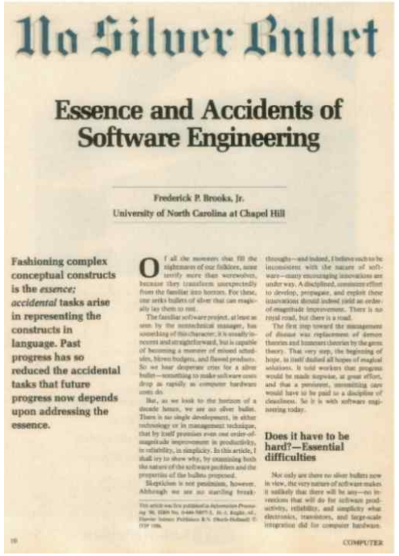
- Maintenance activities include
  - Analysis of changes and extensions
  - Generation of new test suites for the added functionalities
  - Re-executions of tests to check for non regression of software functionalities after changes and extensions
  - Fault tracking and analysis

2. What Techniques Should be Applied?

- **No single A&T technique can serve all purposes.**

- The primary reasons for combining techniques are:
  - Effectiveness for different classes of faults
    - analysis instead of testing for race conditions
  - Applicability at different points in a project
    - inspection for early requirements validation
  - Differences in purpose
    - statistical testing to measure reliability
  - Tradeoffs in cost and assurance
    - expensive technique for key properties


ANNIVERSARY EDITION WITH FOUR NEW CHAPTERS
ESSAYS ON SOFTWARE ENGINEERING
THE MYTHICAL MAN-MONTH
FREDERICK P. BROOKS, JR.



No Silver Bullet

Essence and Accidents of Software Engineering

Frederick P. Brooks, Jr.
University of North Carolina at Chapel Hill

Fashioning complex conceptual constructs is the *essence;* *accidental* tasks arise in representing the constructs in language. Past progress has so reduced the accidental tasks that future progress now depends upon addressing the essence.

Does it have to be hard?—Essential difficulties

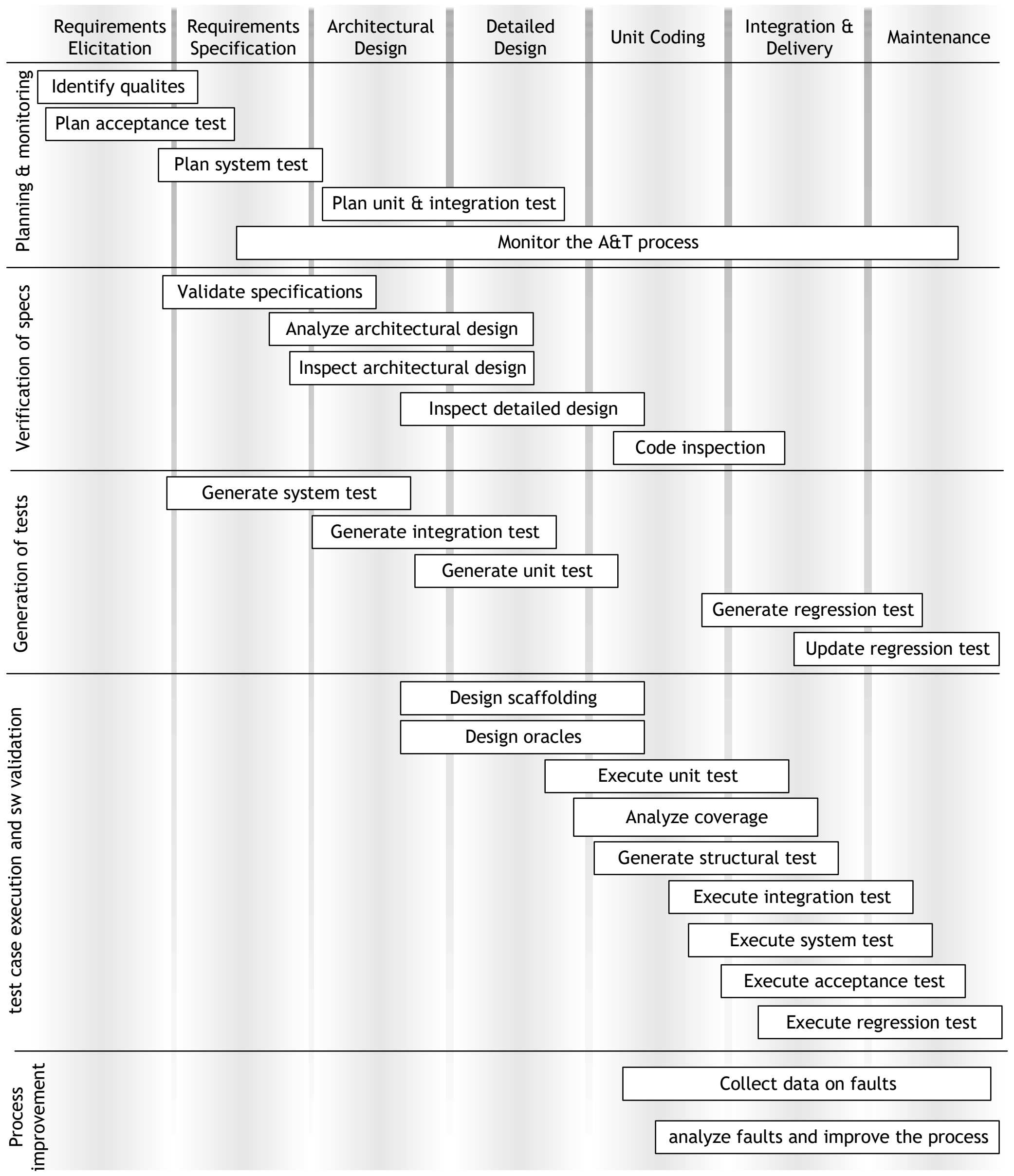
currents — trends. people. projects

No Silver Bullet: Software Engineering Reloaded

Steven Fraser and Dennis Mancl

Opening statements

"No single software engineering development would produce an order-of-magnitude improvement to programming productivity within10 years."
Fredrick Brooks 1986



Requirements Elicitation
Requirements Specification
Architectural Design
Detailed Design
Unit Coding
Integration & Delivery
Maintenance
Planning & monitoring
Identify qualites
Plan acceptance test
Plan system test
Plan unit & integration test
Monitor the A&T process
Verification of specs
Validate specifications
Analyze architectural design
Inspect architectural design
Inspect detailed design
Code inspection
Generation of tests
Generate system test
Generate integration test
Generate unit test
Generate regression test
Update regression test
test case execution and sw validation
Design scaffolding
Design oracles
Execute unit test
Analyze coverage
Generate structural test
Execute integration test
Execute system test
Execute acceptance test
Execute regression test
Process improvement
Collect data on faults
analyze faults and improve the process

3. How Can We Assess the Readiness of a Product?

- A&T activities aim at revealing faults during development.
  - We cannot reveal or remove all faults.
  - A&T cannot last infinitely.
- One day all A&T activities must stop.
- We have to know whether products meet the quality requirements or not.
  - We must specify the required level of dependability.
    → **Metric & Measurement**
  - We can determine when that level has been attained.
    → **Assessment**

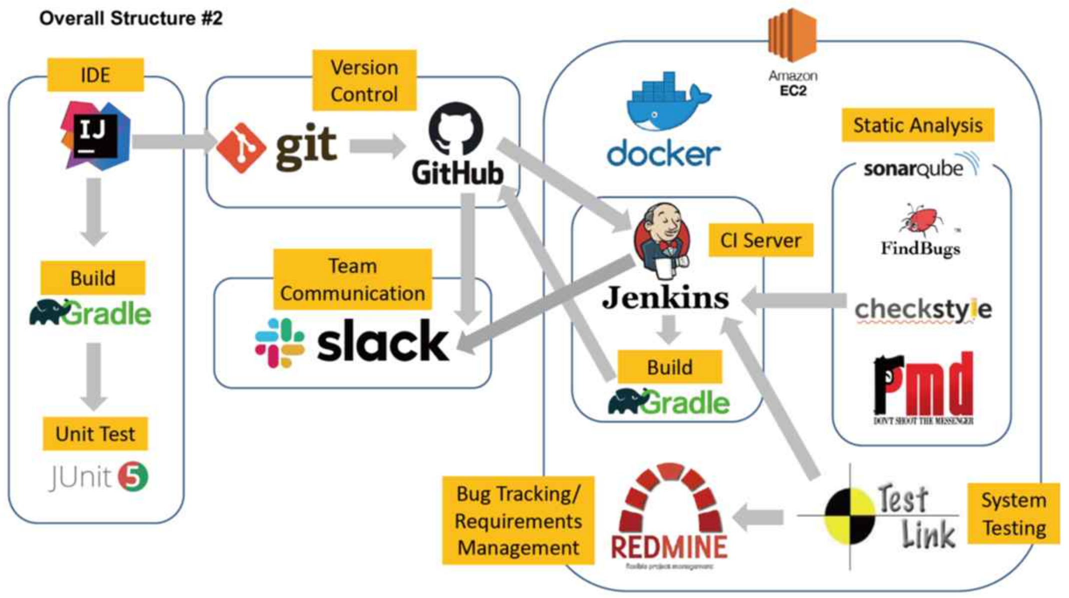
4. How Can We Ensure the Quality of Successive Releases?

- Software products operate for many years and undergo many changes.
  - To adapt to environment changes
  - To serve new and changing user requirements

- A&T activities does not stop at the first release.

- Quality tasks after delivery include
  - Test and analysis of new and modified code
  - Re-execution of system tests
  - Extensive record-keeping

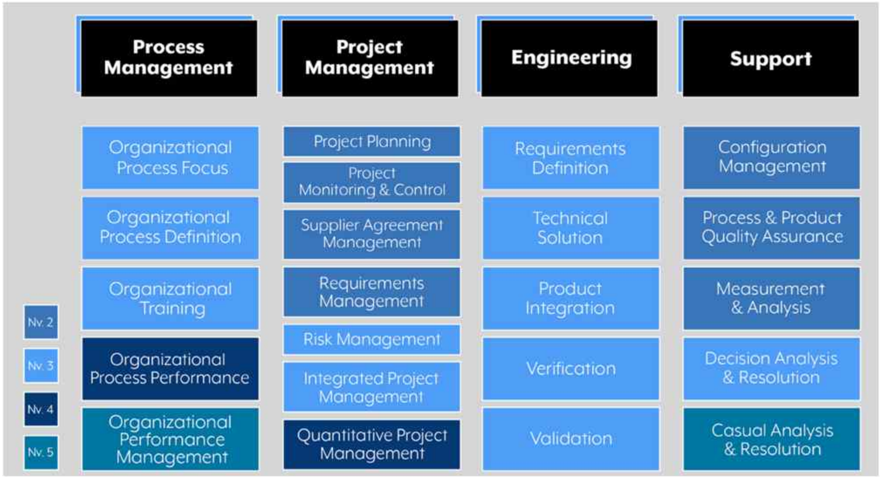
- CTIP helps a lot.



5. How Can the Development Process be Improved?

- The same defects are encountered in project after project.
- We can improve the quality through identifying and removing weaknesses
  - in **development process**
  - in **A&T process (quality process)**

- **SPI (Software Process Improvement)**



Process Management	Project Management	Engineering	Support
Organizational Process Focus	Project Planning	Requirements Definition	Configuration Management
Organizational Process Definition	Project Monitoring & Control	Technical Solution	Process & Product Quality Assurance
Organizational Training	Supplier Agreement Management	Product Integration	Measurement & Analysis
Organizational Process Performance	Requirements Management	Verification	Decision Analysis & Resolution
Organizational Performance Management	Risk Management	Validation	Casual Analysis & Resolution
	Integrated Project Management		
	Quantitative Project Management		

- **CMMi** tries to evaluate quantitatively process quality of an organization (Lv. 1 ~ 5).
  1. **Initial** : Essentially uncontrolled
  2. **Repeatable** : Product (Project) management procedures are defined and used.
  3. **Defined** : Process management procedures and strategies are defined and used.
  4. **Managed** : Quality management strategies are defined and used.
  5. **Optimizing** : Process improvement strategies are defined and used.

BASIC SOFTWARE TESTING & ANALYSIS TECHNIQUES

- **Finite Models**
- **Data Dependency and Data Flow Models**
- **Symbolic Execution and Proof of Properties**
- **Finite State Verification**

Finite Models

- CFG
- Call Graph
- FSM

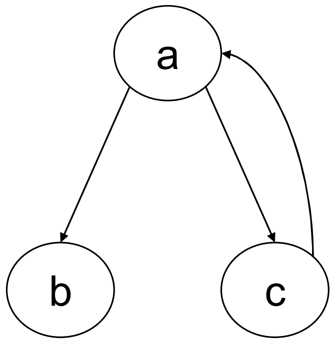
Model

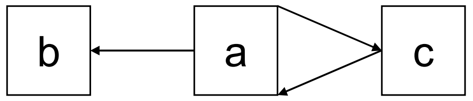
- A **model** is a representation that is simpler than the artifact it represents.
  - While preserves some important attributes of the actual artifact

- Our concern is with ***models*** of ***program execution***.

Directed Graph

- Directed graph:
  - N : set of nodes
  - E : set of edges (relation on the set of nodes)

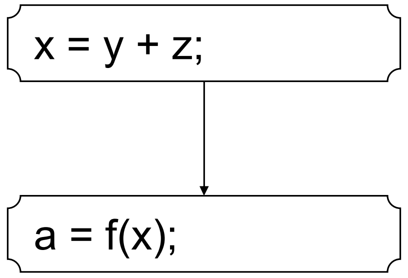




N = { a, b, c }
E = { (a, b), (a, c), (c, a) }

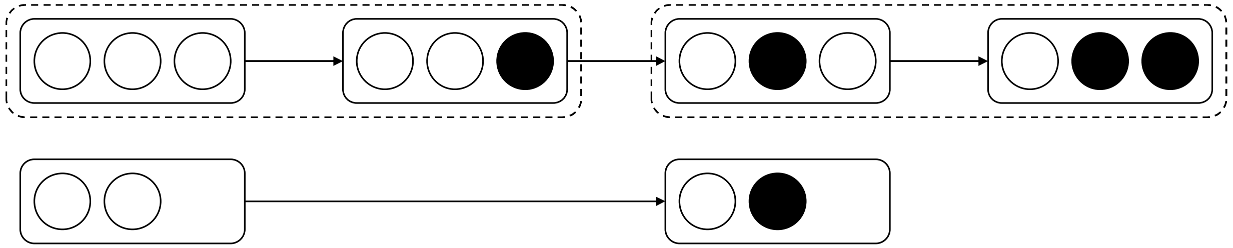
Directed Graph with Labels

- We can label nodes with the names or descriptions of the entities they represent.
  - If nodes a and b represent program regions containing assignment statements, we might draw the two nodes and an edge (a, b) connecting them in this way:

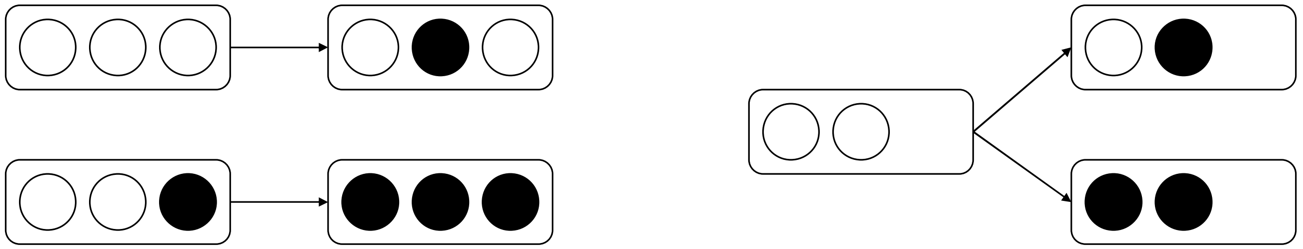


Finite Abstractions of Behavior

- Two (side) effects of abstraction
  - **Coarsening of execution model**



  - **Introduction of nondeterminism**



Intraprocedural Control Flow Graph

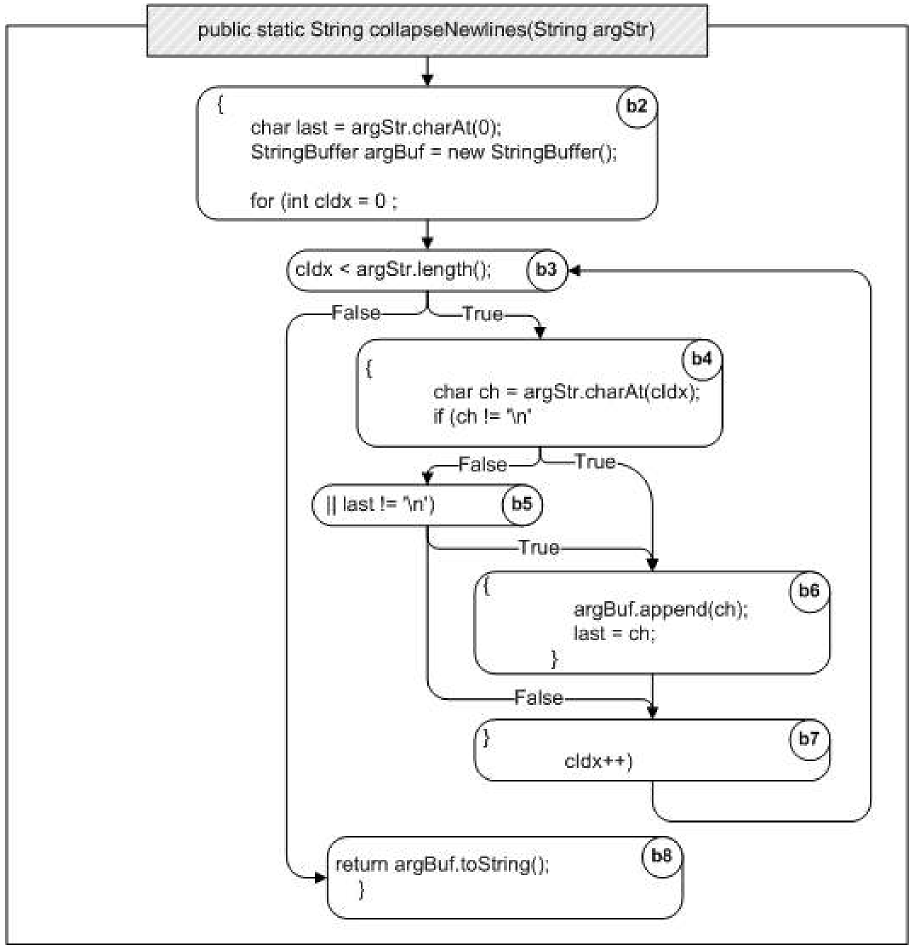
- Called “Control Flow Graph” or “**CFG**”
  - A directed graph (N, E)

- Nodes
  - Regions of source code (basic blocks)
  - Basic block = maximal program region with a single entry and single exit point
  - Statements are often grouped in single regions to get a compact model.
  - Sometime single statements are broken into more than one node to model control flow within the statement.

- Directed edges
  - Possibility that program execution proceeds from the end of one region directly to the beginning of another

An Example of CFG

```
public static String collapseNewlines(String argStr)
  {
      char last = argStr.charAt(0);
      StringBuffer argBuf = new StringBuffer();

      for (int cIdx = 0 ; cIdx < argStr.length(); cIdx++)
      {
          char ch = argStr.charAt(cIdx);
          if (ch != '\n' || last != '\n')
          {
              argBuf.append(ch);
              last = ch;
          }
      }

      return argBuf.toString();
  }
```


public static String collapseNewlines(String argStr)
{ char last = argStr.charAt(0); StringBuffer argBuf = new StringBuffer(); for (int cIdx = 0 ;
b2
cIdx < argStr.length();
b3
False
True
{ char ch = argStr.charAt(cIdx); if (ch != '\n'
b4
False
True
|| last != '\n')
b5
True
{ argBuf.append(ch); last = ch; }
b6
False
} cIdx++)
b7
return argBuf.toString(); }
b8

The Use of CFG

- CFG may be used directly to define thoroughness criteria for testing.
  - *Test Case Selection and Adequacy*
  - *Structural Testing*

- CFG is often used to define another model which is used to define a thoroughness criterion.
  - Data Flow Graph
  - Data Dependency Graph
  - Control Dependency Graph

Call Graph

- "Interprocedural Control Flow Graph" = **Call Graph**
  - A directed graph (N, E)

- Nodes
  - Represent procedures, methods, functions, etc.

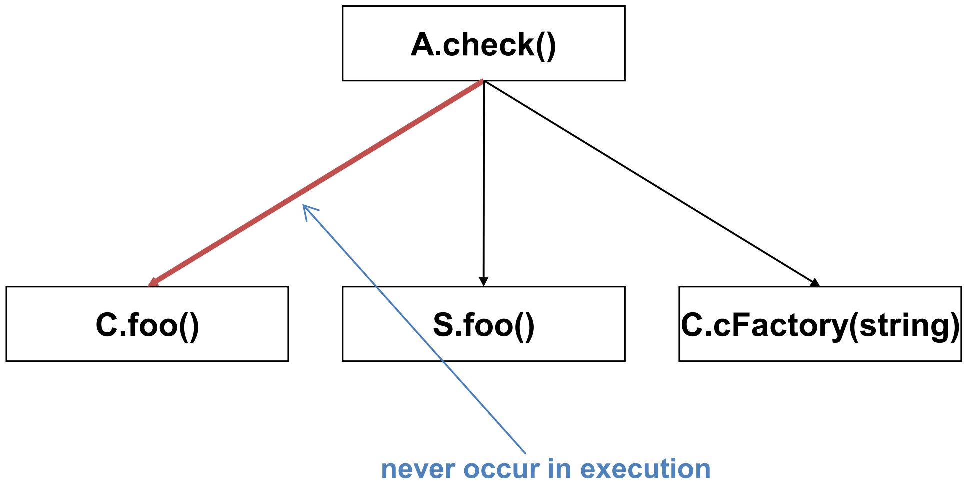
- Edges
  - Represent 'call' relation

- **Call graph** presents many more design issues and trade-off than CFG.
  - Overestimation of call relation
  - Context sensitive/insensitive

Overestimation in a Call Graph

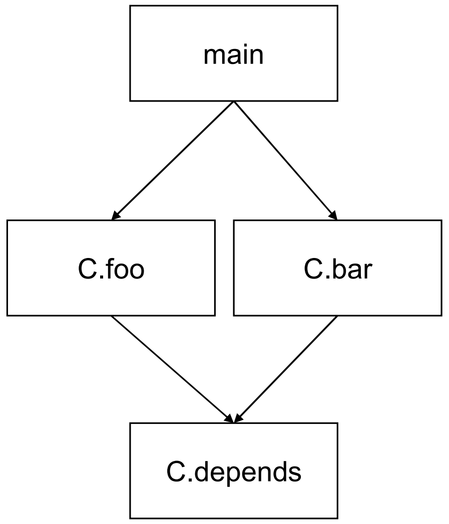
- The static call graph includes calls through dynamic bindings that never occur in execution.

```
public class C {
   public static C cFactory(String kind) {
        if (kind == "C") return new C();
        if (kind == "S") return new S();
        return null;
   }
   void foo() {
       System.out.println("You called the parent's method");
   }
   public static void main(String args[]) {
        (new A()).check();
   }
}
class S extends C {
   void foo() {
        System.out.println("You called the child's method");
   }
}
class A {
   void check() {
        C myC = C.cFactory("S");
        myC.foo();
   }
}
```

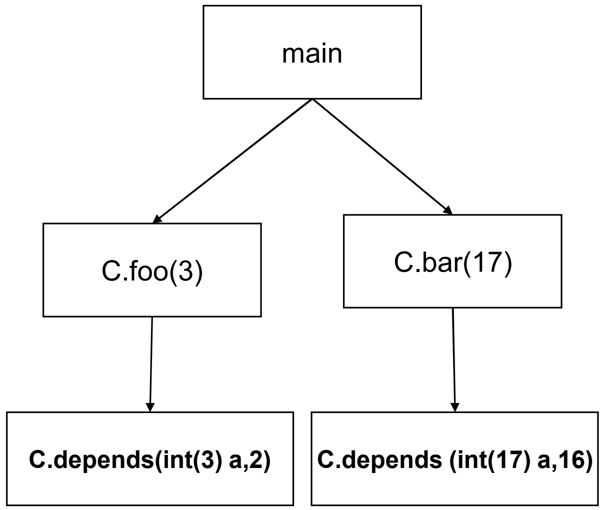


Context Sensitive/Insensitive Call Graphs

```
public class Context {
  public static void main(String args[]) {
      Context c = new Context();
      c.foo(3);
      c.bar(17);
  }

  void foo(int n) {
      int[]  myArray = new int[ n ];
      depends( myArray, 2) ;
  }

  void bar(int n) {
      int[]  myArray = new int[ n ];
      depends( myArray, 16) ;
  }

  void depends( int[] a, int n ) {
      a[n] = 42;
  }
}
```



< Context Insensitive >



< Context Sensitive >

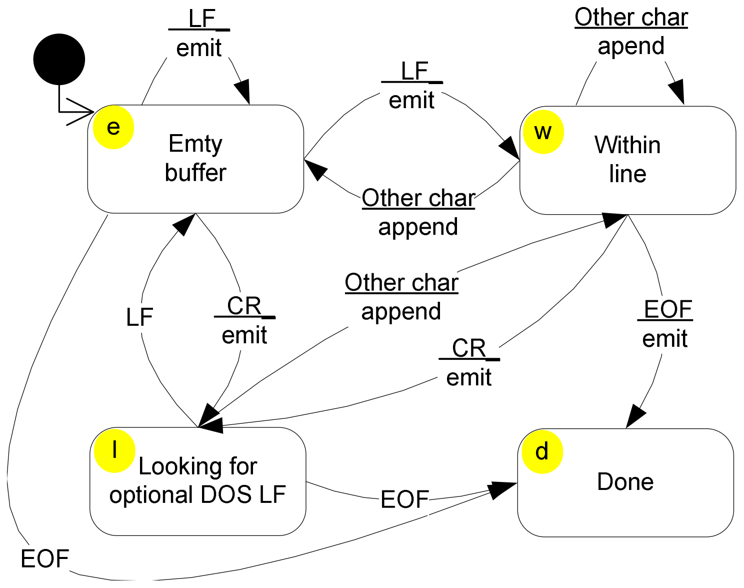
Finite State Machine

- **FSM**s are constructed prior to source code and serve as specifications.
  - While CFGs can be extracted from programs.
  - A directed graph (N, E)
  - CFG and FSM are duals.

- Nodes
  - A finite set of states

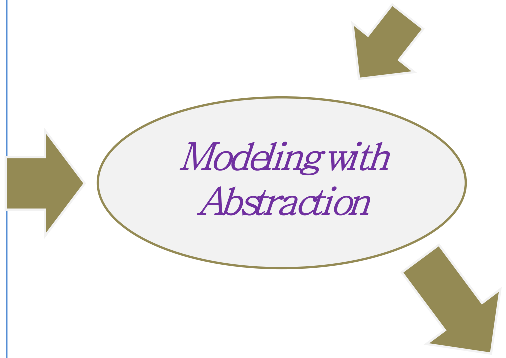
- Edges
  - A set of transitions among states

	LF	CR	EOF	other char
e	e / emit	l / emit	d / -	w / append
w	e / emit	l / emit	d / emit	w / append
l	e / -		d / -	w / append



Abstract Function for Modeling FSMs

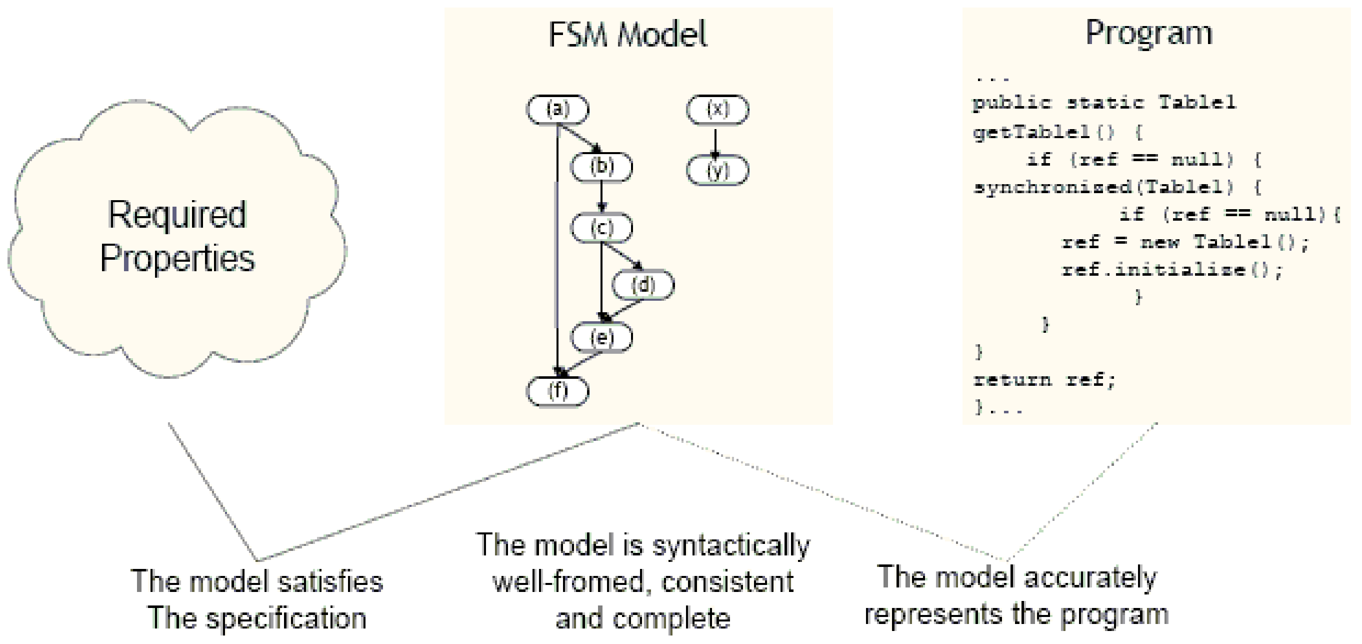
```
1   /** Convert each line from standard input */
2   void  transduce() {
3
4     #define BUFLEN 1000
5     char buf[BUFLEN];   /* Accumulate line into this buffer   */
6     int    pos = 0;            /* Index for next character in buffer */
7
8     char inChar; /* Next character from input */
9
10    int atCR = 0; /* 0="within line", 1="optional DOS LF" */
11
12    while ((inChar = getchar()) != EOF ) {
13      switch (inChar) {
14      case LF:
15        if (atCR) {    /* Optional DOS LF */
16          atCR = 0;
17        } else {         /* Encountered CR within line */
18          emit(buf, pos);
19          pos = 0;
20        }
21        break;
22      case CR:
23        emit(buf, pos);
24        pos = 0;
25        atCR = 1;
26        break;
27      default:
28        if (pos >= BUFLEN-2) fail("Buffer overflow");
29        buf[pos++] = inChar;
30      } /* switch */
31    }
32    if (pos > 0) {
33      emit(buf, pos);
34    }
35  }
```



Abstract state	*Concrete state*		
	Lines	atCR	pos
e (Empty buffer)	3 – 13	0	0
w (Within line)	13	0	> 0
l (Looking for LF)	13	1	0
d (Done)	36	–	–

	LF	CR	EOF	other
e	e / emit	l / emit	d / –	w / append
w	e / emit	l / emit	d / emit	w / append
l	e / –	l / emit	d / –	w / append

Correctness Relations for FSM Models



Required Properties
FSM Model
(a)
(b)
(c)
(d)
(e)
(f)
(x)
(y)
Program
The model satisfies The specification
The model is syntactically well-fromed, consistent and complete
The model accurately represents the program

```
...
public static Table1
getTable1() {
    if (ref == null) {
synchronized(Table1) {
          if (ref == null){
       ref = new Table1();
       ref.initialize();
          }
     }
}
return ref;
}...
```

Dependable Software Laboratory

Data and Control Dependence

- Data Dependency Graph
- Control Dependency

Why Data Flow Models Need?

- **The 3 Finite models** emphasize **control flow** only.
  - Control flow graph
  - Call graph
  - Finite state machine

- We also need to reason about **data dependence** to reason about transmission of information through program variables.
  - ***"Where does this value of x come from?"***
  - ***"What would be affected by changing this?"***

- Many program analyses and test design techniques use **data flow information** and **dependences**, and often **in combination with control flows**.

Definition-Use Pairs

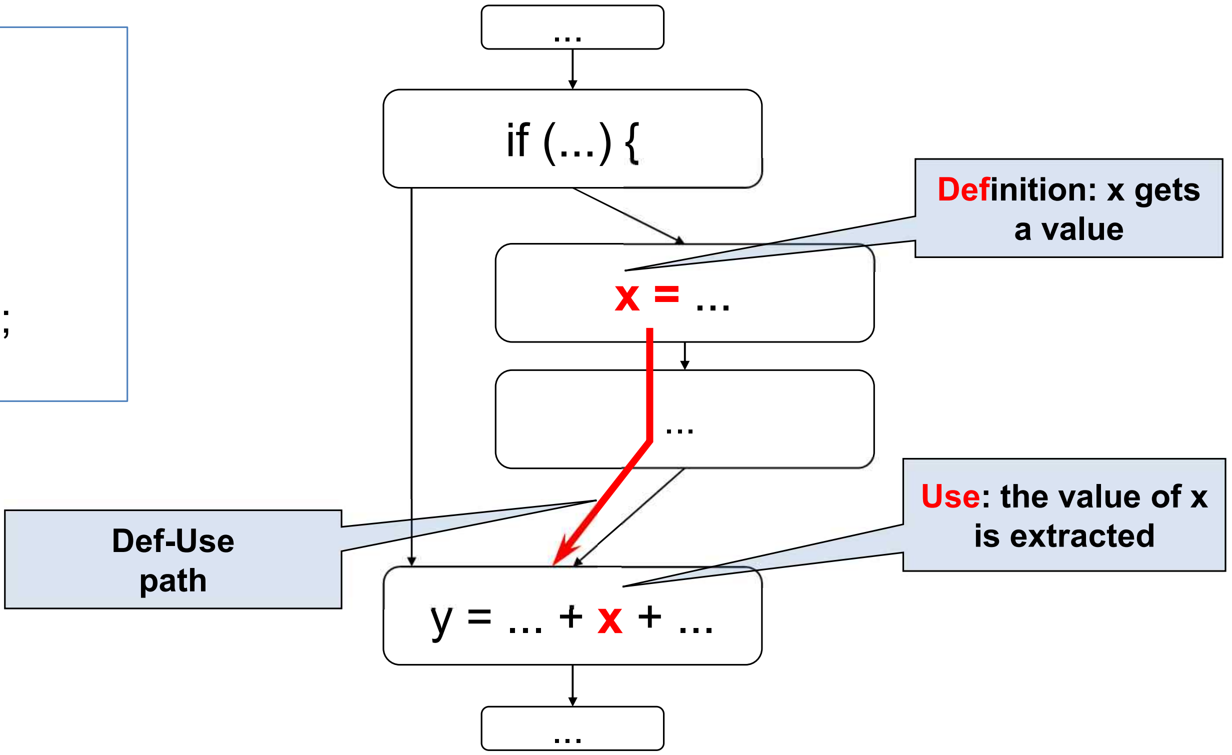
- **Def-use (du) pair** associates a point in a program where a value is produced with a point where it is used.

- **Definition**: where a variable gets a value
  - Variable declaration
  - Variable initialization
  - Assignment
  - Values received by a parameter

- **Use**: extraction of a value from a variable
  - Expressions
  - Conditional statements
  - Parameter passing
  - Returns

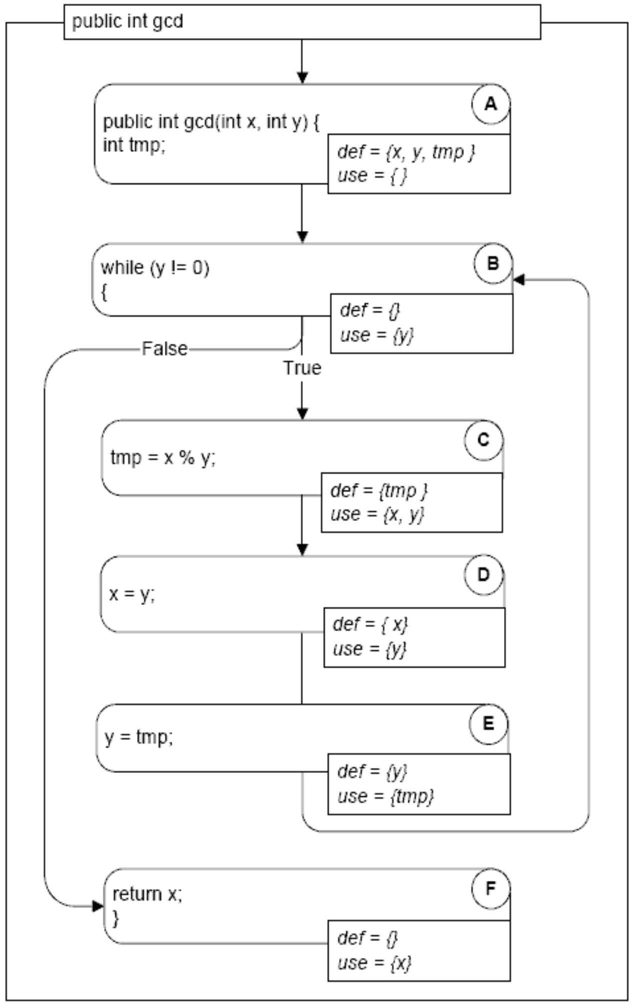
Def-Use Pairs and Def-Use Paths

```
...
if (...) {
    x = ... ;
...
}
y = ... + x + ... ;
...
```



Def-Use Pairs

```
/**  Euclid's algorithm */

public int gcd(int x, int y) {
        int tmp;                // A: def x, y, tmp
        while (y != 0) {     // B: use y
           tmp = x % y;     // C: def tmp; use x, y
           x = y;                // D: def x; use y
           y = tmp;            // E: def y; use tmp
        }
     return x;                // F: use x
 }
```



DEPENDABLE SOFTWARE LABORATORY

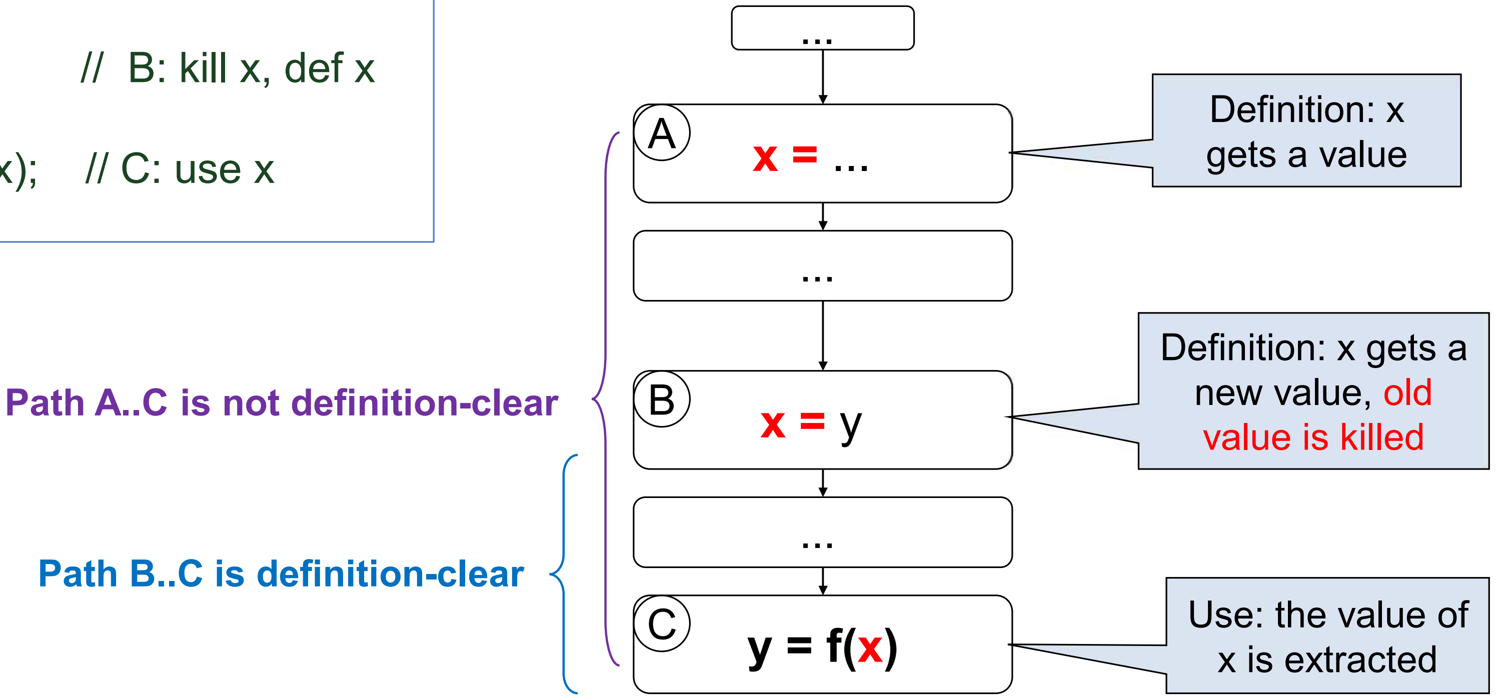
Definition-Clear & Killing

- A **definition-clear path** is a path along the CFG from a definition to a use of the same variable without another definition of the variable between.
  - If, instead, another definition is present on the path, then the latter definition **kills** the former.

- A **def-use pair** is formed if and only if there is a definition-clear path between the definition and the use.

Definition-Clear & Killing

```
x = ...       // A: def x
q = ...
x = y;        //  B: kill x, def x
z = ...
y = f(x);     // C: use x
```

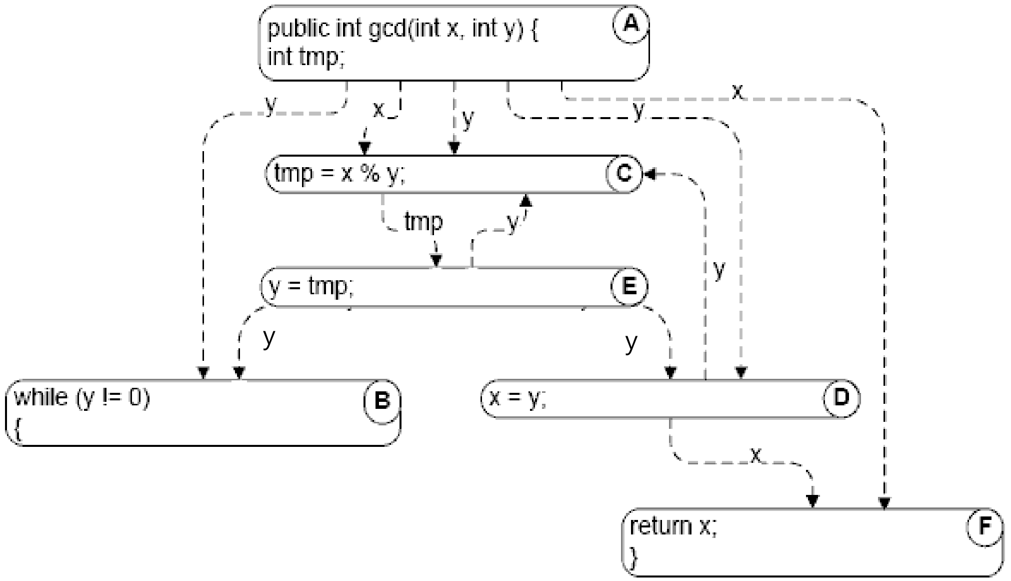


...
A
x = ...
Definition: x gets a value
...
B
x = y
Definition: x gets a new value, old value is killed
Path A..C is not definition-clear
...
Path B..C is definition-clear
C
y = f(x)
Use: the value of x is extracted

(Direct) Data Dependence Graph

- **Direct data dependence graph**
  - "*Where did these values come from?*"
  - A direct graph (N, E)
    - Nodes: as in the control flow graph (CFG)
    - Edges: def-use (du) pairs, labelled with the variable name

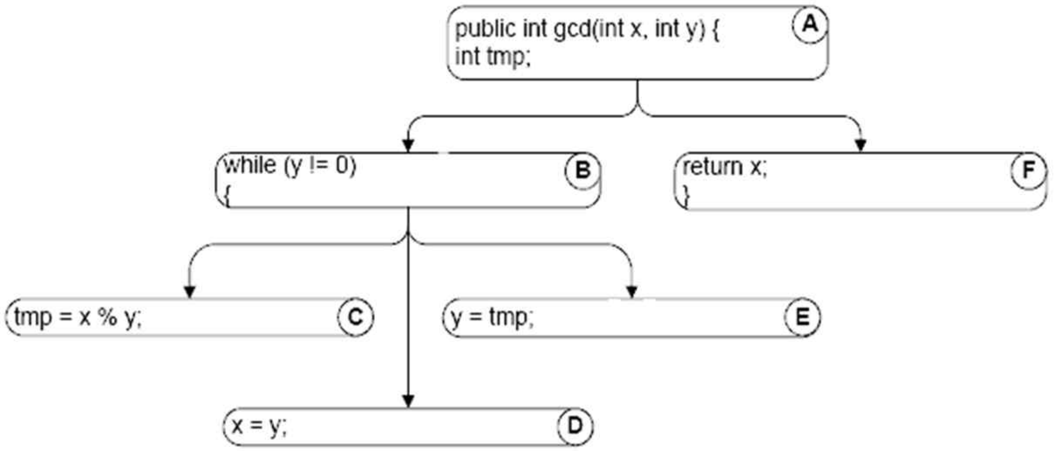
```
/**  Euclid's algorithm */

public int gcd(int x, int y) {
        int tmp;              // A: def x, y, tmp
        while (y != 0) {      // B: use y
            tmp = x % y;      // C: def tmp; use x, y
            x = y;            // D: def x; use y
            y = tmp;          // E: def y; use tmp
        }
      return x;               // F: use x
}
```



Control Dependence

- **Control dependence**
  - "*Which statement controls whether this statement executes?*"
  - A (rooted) directed graph
    - Nodes: as in the CFG
    - Edges: unlabelled, from entry/branching points to controlled blocks

```
/**  Euclid's algorithm */

public int gcd(int x, int y) {
       int tmp;             // A: def x, y, tmp
       while (y != 0) {     // B: use y
          tmp = x % y;     // C: def tmp; use x, y
          x = y;           // D: def x; use y
          y = tmp;         // E: def y; use tmp
       }
     return x;             // F: use x
 }
```



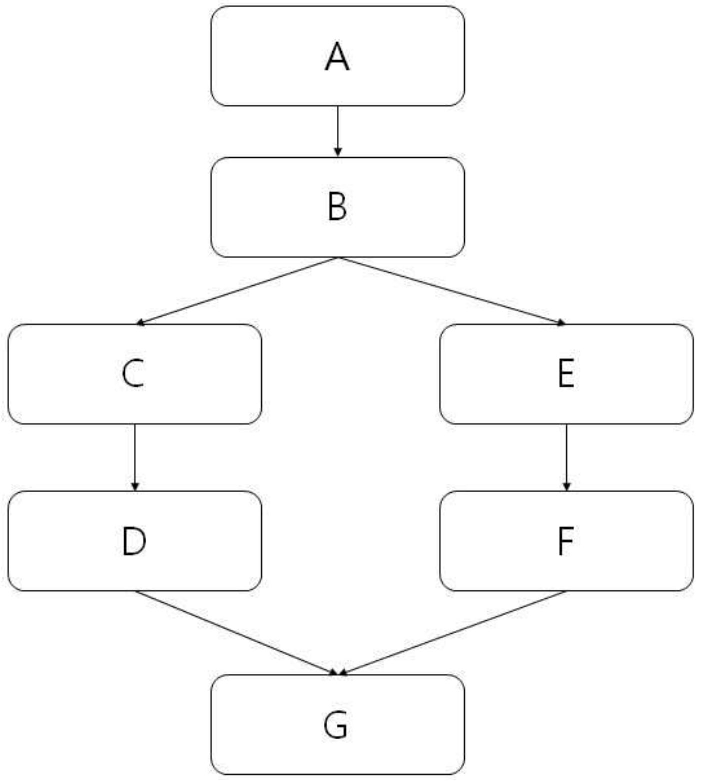
Dominator

- **Pre-dominators** are used to make this intuitive notion of "*controlling decision*" precise.

- *Node M **dominates** node N, if every path from the root to N passes through M.*
  - A node will typically have many dominators, but except for the root, there is a unique immediate dominator of node N which is closest to N on any path from the root, and which is in turn dominated by all the other dominators of N.
  - Because each node (except the root) has a unique immediate dominator, the immediate dominator relation forms a tree.

- **Post-dominators** are calculated in the reverse of the control flow graph, using a special "exit" node as the root.

An Example of Dominators



- A pre-dominates all nodes.
- G post-dominates all nodes.

- F and G post-dominate E.
- G is the immediate post-dominator of B.

- C does not post-dominate B.

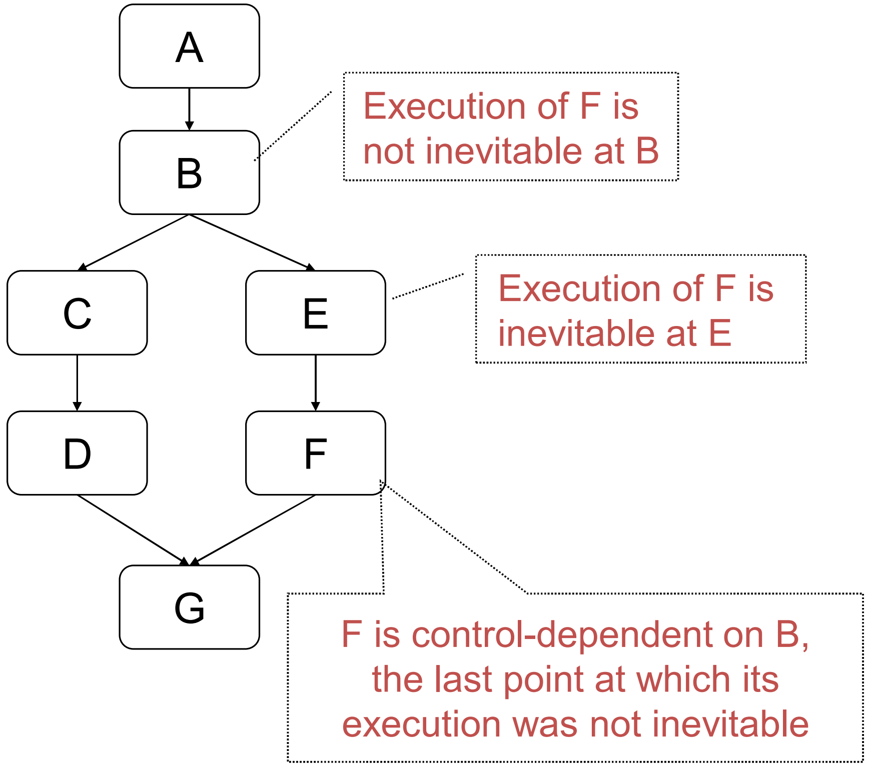
- B is the immediate pre-dominator of G.
- F does not pre-dominate G.

More Precise Definition of Control Dependence

- We can use post-dominators to give a more precise definition of control dependence
  - Consider again a node N that is reached on some but not all execution paths.
  - There must be some node C with the following property:
    - C has at least two successors in the control flow graph (i.e., it represents a control flow decision).
    - C is not post-dominated by N.
    - There is a successor of C in the control flow graph that is post-dominated by N.
  - When these conditions are true, we say node N is control-dependent on node C.

- Intuitively, if C is the last decision that controls whether N executes or not, we say that N is control-dependent on C.

An Example of Control Dependence


A
B
C
E
D
F
G
Execution of F is not inevitable at B
Execution of F is inevitable at E
F is control-dependent on B, the last point at which its execution was not inevitable

Symbolic Execution and Proof of Properties

Symbolic Execution

- Symbolic execution builds predicates that characterize conditions for executing paths and effects of the execution on program state.
  - ***Bridges program behavior to logic***

- Finds important applications in
  - Program analysis
  - Test data generation
  - Formal verification (proofs) of program correctness
    - Rigorous proofs of properties of critical subsystems
      - Example: safety kernel of a medical device
    - Formal verification of critical properties particularly resistant to dynamic testing
      - Example: security properties
    - Formal verification of algorithm descriptions and logical designs
      - less complex than implementations

Symbolic Execution

- Tracing execution with **symbolic** values and expressions
  – Values are expressions over symbols.
  – Executing statements computes new expressions with the symbols.

Execution with concrete values

(before)

low	12
high	15
mid	-

mid = (high + low) / 2

(after)

low	12
high	15
mid	13

Execution with symbolic values

(before)

low	L
high	H
mid	-

mid = (high + low) / 2

(after)

Low	L
high	H
mid	(L+H) / 2

Tracing Execution with Symbolic Executions

```
char *binarySearch( char *key, char *dictKeys[ ],
        char *dictValues[ ],  int dictSize) {

  int low = 0;
  int high = dictSize - 1;
  int mid;
  int comparison;

  while (high >= low) {
    mid = (high + low) / 2;
    comparison = strcmp( dictKeys[mid], key );
    if (comparison < 0) {
      low = mid + 1;
    } else if ( comparison > 0 ) {
      high = mid - 1;
    } else {
      return dictValues[mid];
    }
  }
  return 0;

}
```

∧ ∀k, 0 ≤ k < size : dictKeys[k] = key → L ≤ k ≤ H
∧ H ≥ M ≥ L

Execution with symbolic values

```
            (before)
            low = 0
∧           high = H-1
∧           mid = (H-1)/2        ← supposed

while (high >= low) {

            (after)
            low = 0
∧           high = H-1
∧           mid = (H-1)/2        ← when true
∧           (H-1)/2 - key >= 0
...
∧           not((H-1)/2 - key >= 0)   ← when false
```

Summary Information

- Symbolic representation of paths may become extremely complex.
- We can simplify the representation by replacing a complex condition P with **a weaker condition W** such that
  - P => W
    - W describes the path with less precision.
  - W is a **summary** of P.

An Example of Summary Information

- If we are reasoning about the correctness of the binary search algorithm,
  - "mid = (high+low) / 2 "

Complete condition:

$$
\begin{aligned}
& \mathit{low} = L \\
\wedge\ & \mathit{high} = H \\
\wedge\ & \mathit{mid} = M \\
\wedge\ & M = (L+H) / 2
\end{aligned}
$$

Weaker condition:

$$
\begin{aligned}
& \mathit{low} = L \\
\wedge\ & \mathit{high} = H \\
\wedge\ & \mathit{mid} = M \\
\wedge\ & L <= M <= H
\end{aligned}
$$

- The weaker condition contains less information, but still ***enough*** to reason about correctness.

Weaker Precondition

- The weaker predicate "*L <= mid <= H*" is chosen based on what must be true for the program to execute correctly.
  - It cannot be derived automatically from source code.
  - It depends on our understanding of the code and our rationale for believing it to be correct.
- A predicate stating what should be true at a given point can be expressed in the form of an **assertion**.
- Weakening the predicate has a cost for testing.
  - Satisfying the predicate is no longer sufficient to find data that forces program execution along that path.
    - Test data satisfying a weaker predicate W is necessary to execute the path, but it may not be sufficient.
    - Showing that W cannot be satisfied shows path infeasibility.

Loops and Assertions

- The number of execution paths through a program with loops is potentially infinite.
- To reason about program behavior in a loop, we can place within the loop an **invariant**.
  - Assertion that states a predicate that is expected to be true each time execution reaches that point
- Each time program execution reaches the invariant assertion, we can weaken the description of program state.
  - If predicate P represents the program state and the assertion is W
  - We must first ascertain P => W
  - And then we can substitute W for P

Precondition and Postcondition

- Supposed that
  - Every loop contains an assertion **(Loop Invariant)**
  - There is an assertion at the beginning of the program **(Precondition)**
  - There is a final assertion at the end **(Postcondition)**

- Then
  - Every possible execution path would be a sequence of segments from one assertion to the next.

- **Precondition** : the assertion at the beginning of a segment
- **Postcondition** : the assertion at the end of the segment

Verification of Program Correctness

- For each program segment, if we can verify that
  - ***Starting from the precondition,***
  - ***Executing the program segment,***
  - ***And postcondition holds at the end of the segment.***

- Then, we can verify the correctness of *an infinite number of program paths*.

An Example of Verification with Assertions

```
char *binarySearch( char *key, char *dictKeys[ ],
      char *dictValues[ ],  int dictSize) {

  int low = 0;
  int high = dictSize - 1;
  int mid;
  int comparison;

  while (high >= low) {
    mid = (high + low) / 2;
    comparison = strcmp( dictKeys[mid], key );
    if (comparison < 0) {
      low = mid + 1;
    } else if ( comparison > 0 ) {
      high = mid - 1;
    } else {
      return dictValues[mid];
    }
  }
  return 0;

}
```

Precondition: "should be sorted"
$\forall i,j,\ 0 \le i < j < size : dictKeys[i] \le dictKeys[j]$

Invariant: "should be in range"
$\forall i,\ 0 \le i < size : dictKeys[i] = key \rightarrow low \le i \le high$

When Executing the Loop

Initial values:

$low = L$
$\wedge\ high = H$

Precondition
$\forall i,j,\ 0 \le i < j < size : dictKeys[i] \le dictKeys[j]$

Instantiated invariant:

$\forall i, j,\ 0 \le i < j < size : dictKeys[i] \le dictKeys[j]$
$\wedge\ \forall k,\ 0 \le k < size : dictKeys[k] = key \rightarrow L \le k \le H$

After executing:

`mid = (high + low) / 2`

Invariant
$\forall i,\ 0 \le i < size :$
$dictKeys[i] = key \rightarrow low \le i \le high$

$low = L$
$\wedge\ high = H$
$\wedge\ mid = M$
$\wedge\ \forall i, j,\ 0 \le i < j < size : dictKeys[i] \le dictKeys[j]$
$\wedge\ \forall k,\ 0 \le k < size : dictKeys[k] = key \rightarrow L \le k \le H$
$\wedge\ H \ge M \ge L$

After executing the Loop

In case of M < key < H

After executing the loop:

$low = M+1$
$\land\ high = H$
$\land\ mid = M$
$\land\ \forall i, j, 0 \le i < j < size : dictKeys[i] \le dictKeys[j]$
$\land\ \forall k, 0 \le k < size : dictKeys[k] = key \rightarrow L \le k \le H$
$\land\ H \ge M \ge L$
$\land\ dictkeys[M] < key$

The new instance of the invariant:

$\forall i, j, 0 \le i < j < size : dictKeys[i] \le dictKeys[j]$
$\land\ \forall k, 0 \le k < size : dictKeys[k] = key \rightarrow M+1 \le k <= H$

- *If the invariant is satisfied, then the loop is correct with respect to the preconditions and the invariant.*

At the End of the Loop

- *Even the invariant is satisfied, but the postcondition is false:*

$$
\begin{aligned}
&\ \ \mathbf{low = L} \\
&\wedge\ \mathbf{high = H} \\
&\wedge\ \forall i, j,\ 0 \le i < j < size : dictKeys[i] \le dictKeys[j] \\
&\wedge\ \forall k,\ 0 \le k < size : dictKeys[k] = key \rightarrow L \le k \le H \\
&\wedge\ L > H
\end{aligned}
$$

- ***If the condition satisfies the post-condition, then the program is correct with respect to the pre-condition and post-condition.***

Compositional Reasoning

- Follow the hierarchical structure of a program
  - at a small scale (within a single procedure)
  - at larger scales (across multiple procedures)

- **Hoare triple: [pre] block [post]**
  - ***If the program is in a state satisfying the precondition pre at entry to the block, then after execution of the block, it will be in a state satisfying the postcondition post.***

(Not "it should be")

Reasoning about Hoare Triples: Inference

While loops:

I : invariant
C : loop condition
S : body of the loop

premise

$$\frac{[\,I \wedge C\,]\; S\; [\,I\,]}{[\,I\,]\ \text{while}(C)\ \{\,S\,\}\ [I \wedge \neg C]}$$

conclusion

Inference rule says:
"if we can verify the premise (top), then we can infer the conclusion (bottom)"

Other Inference Rule

if statement:

$$\frac{[P \wedge C]\ \text{thenpart}\ [Q] \qquad [P \wedge \neg C]\ \text{elsepart}\ [Q]}{[P]\ \text{if}\ (C)\ \{\text{thenpart}\}\ \text{else}\ \{\text{elsepart}\}\ [Q]}$$

Reasoning Style

- Summarize the effect of a block of program code by a " **contract = precondition + postcondition** "
  - We can then use the contract wherever the procedure is called.

- Summarizing binarySearch:

(∀i,j, 0≤i<j<size : keys[i]≤keys[j]) ← precondition

s = binarySearch(k, keys, vals, size)

(s=v and ∃i , 0≤i≤size : keys[i]=k ∧ vals[i]=v) ← postcondition
∨ (s=v ∧ ¬∃i , 0≤i≤size : keys[i]=k)

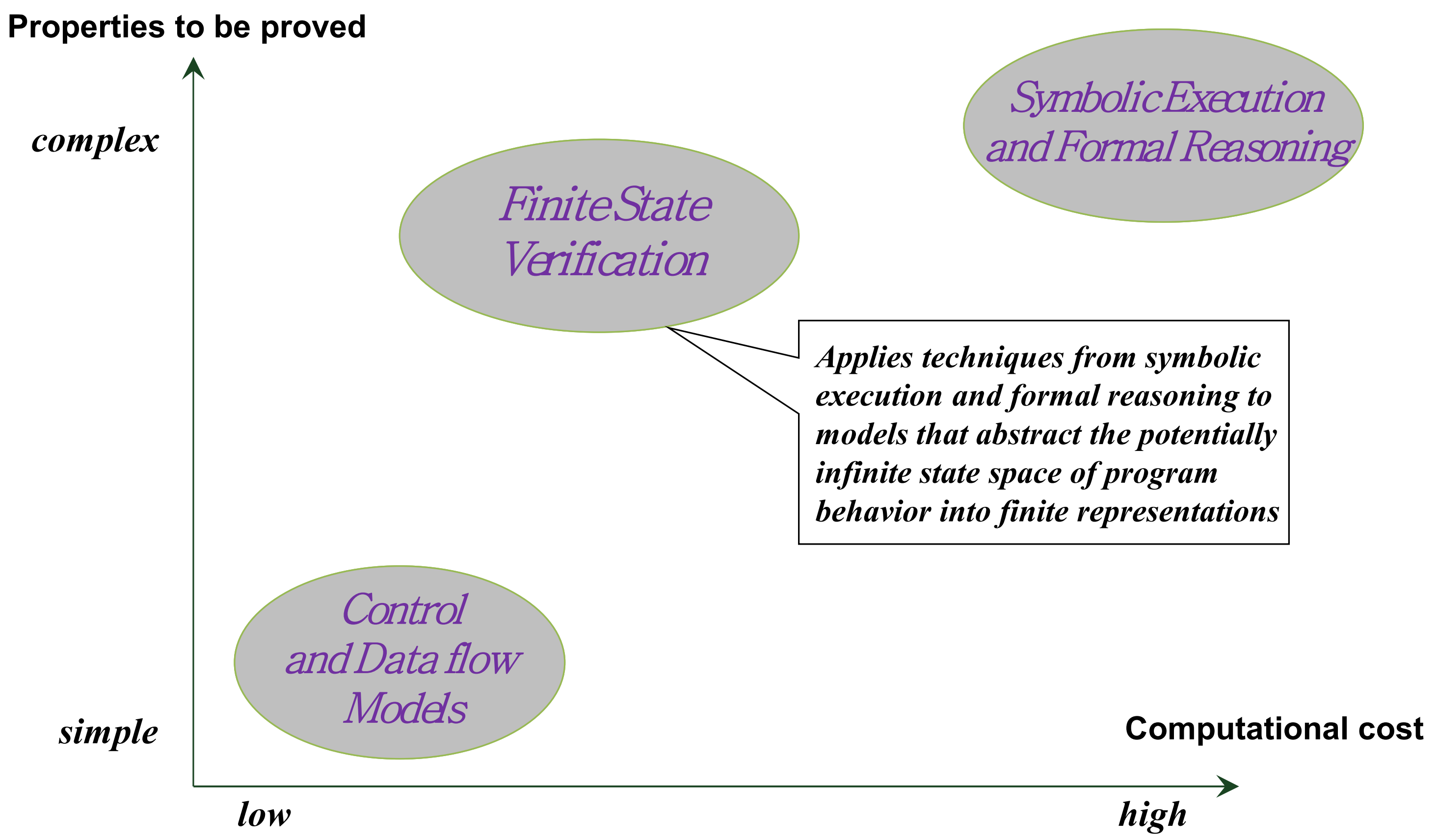
Finite State Verification

Finite State Verification (FSV)

- **Finite state verification** can **automatically prove** some significant properties of a finite model of the infinite execution space.
  - Most important properties of program execution are not decidable.

- Need to balance trade-offs among
  - Generality of properties to be checked
  - Class of programs or models that can be checked
  - Computational effort in checking
  - Human effort in producing models and specifying properties

Resources and Results

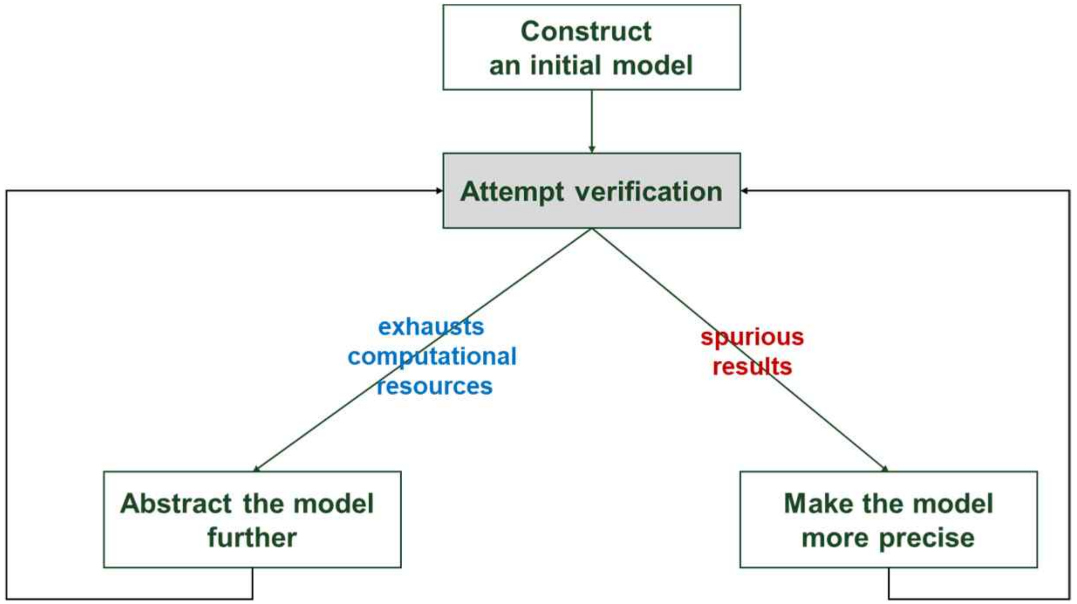

Properties to be proved
complex
simple
Finite State Verification
Symbolic Execution and Formal Reasoning
Control and Data flow Models
Computational cost
low
high

Applies techniques from symbolic execution and formal reasoning to models that abstract the potentially infinite state space of program behavior into finite representations

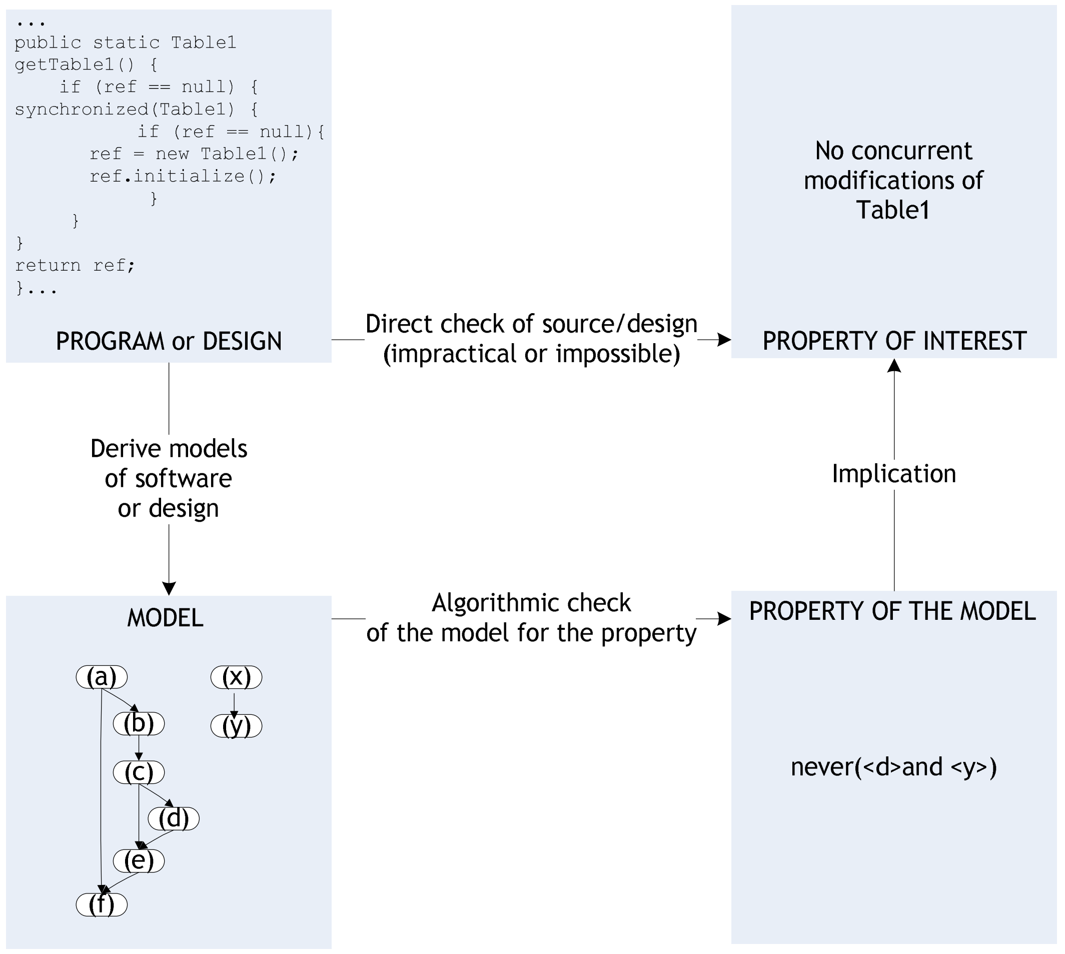
Cost of FSV

- **Human effort and skill** are required.
  - to prepare a finite state model
  - to prepare a suitable specification (property) for automated analysis

- **Iterative process of FSV**
  - Prepare a model and specify properties
  - Attempt verification
  - Receive reports of impossible or unimportant faults
  - Refine the specification or the model


Construct an initial model
Attempt verification
exhausts computational resources
spurious results
Abstract the model further
Make the model more precise

Finite State Verification Framework



```
...
public static Table1
getTable1() {
    if (ref == null) {
synchronized(Table1) {
            if (ref == null){
        ref = new Table1();
        ref.initialize();
            }
      }
}
return ref;
}...
```

PROGRAM or DESIGN
Direct check of source/design (impractical or impossible)
No concurrent modifications of Table1
PROPERTY OF INTEREST
Derive models of software or design
Implication
MODEL
(a)
(b)
(c)
(d)
(e)
(f)
(x)
(y)
Algorithmic check of the model for the property
PROPERTY OF THE MODEL
never(<d>and <y>)

Applications for Finite State Verifications

- Concurrent (multi-threaded, distributed, parallel, etc.) system
  - First and most well-developed application of FSV
  - ***Difficult to test thoroughly*** (apparent non-determinism based on scheduler)
  - Sensitive to differences between development environment and field environment

- Data models
  - Difficult to identify “corner cases” and interactions among constraints, or to thoroughly test them

- ***Security***
  - Some threats depend on unusual (and untested) use

Modeling Concurrent System

- **Deriving a good finite state model is hard.**

- Example: FSM model of a program with multiple threads of control
  - Simplifying assumptions
    - We can determine in advance the number of threads.
    - We can obtain a finite state machine model of each thread.
    - We can identify the points at which processes can interact.
  - State of the whole system model
    - Tuple of states of individual process models
  - Transition
    - Transition of one or more of the individual processes, acting individually or in concert

An Example : On-line Purchasing System

- Specification
  - In-memory data structure initialized by reading configuration tables at system start-up
  - Initialization of the data structure must appear atomic.
  - The system must be reinitialized on occasion.
  - The structure is kept in memory.

- Implementation (with bugs)
  - No monitor (e.g., *Java synchronized*), because it's too expensive.
  - But use *double-checked locking idiom** for a fast system
    - *Bad decision, broken idiom ... but extremely hard to find the bug through testing. (before JVM 1.4)

On-line Purchasing System - Implementation

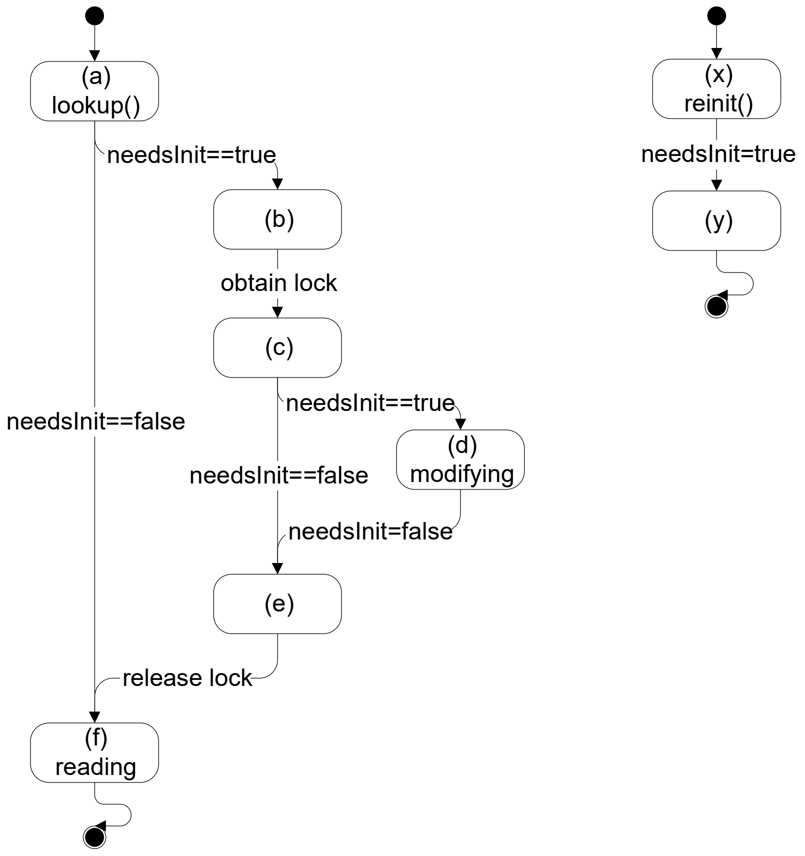
```
class Table1 {
        private static Table1 ref = null;
        private boolean needsInit = true;
        private ElementClass [ ] theValues;
        private Table1() { }

public static Table1 getTable1() {
        if (ref == null)
          { synchedInitialize(); }
        return ref;
}

private static synchronized void synchedInitialize() {
         if (ref == null) {
                  ref = new Table1();
                  ref.initialize();
         }
}
```

```
public void reinit()  { needsInit = true; }

private synchronized void initialize() {
     . . .
      needsInit = false;
}

public int lookup(int i) {
   if (needsInit) {
      synchronized(this) {
         if (needsInit) {
            this.initialize();
         }
      }
   }
   return theValues[i].getX()  + theValues[i].getY();
}

   . . .
}
```

Analysis on On-line Purchasing System

- Start from models of individual threads
  - Systematically trace all the possible interleaving of threads
  - Like hand-executing all possible sequences of execution, but automated

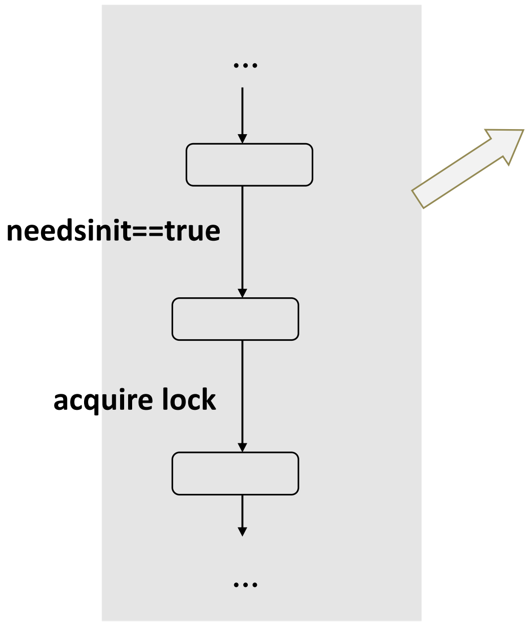
- Analysis begins by constructing an **FSM model** of each individual thread.



Analysis (Continued)

- Java threading rules:
  - "When one thread has obtained a monitor lock, the other thread cannot obtain the same lock."
- Locking prevents threads from concurrently calling initialize
  - But does not prevent possible race condition between threads executing the lookup method
- However, tracing possible executions by hand is completely impractical.
- Use a finite state verification using the **SPIN model checker**

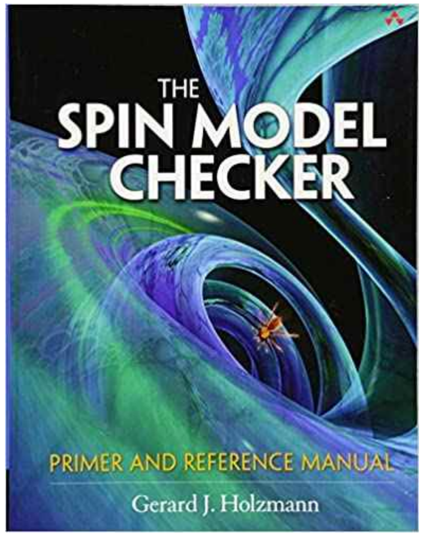
Modeling the System in PROMELA



```
proctype Lookup(int id) {
  if :: (needsInit) ->
    atomic { ! locked  -> locked = true; };
    if  :: (needsInit) ->
      assert (! modifying);
      modifying = true;
      /*  Initialization happens here */
      modifying = false ;
      needsInit = false;
     :: (! needsInit) ->
      skip;
     fi;
     locked = false ;
   fi;
   assert  (! modifying);}
```

Run SPIN and Output

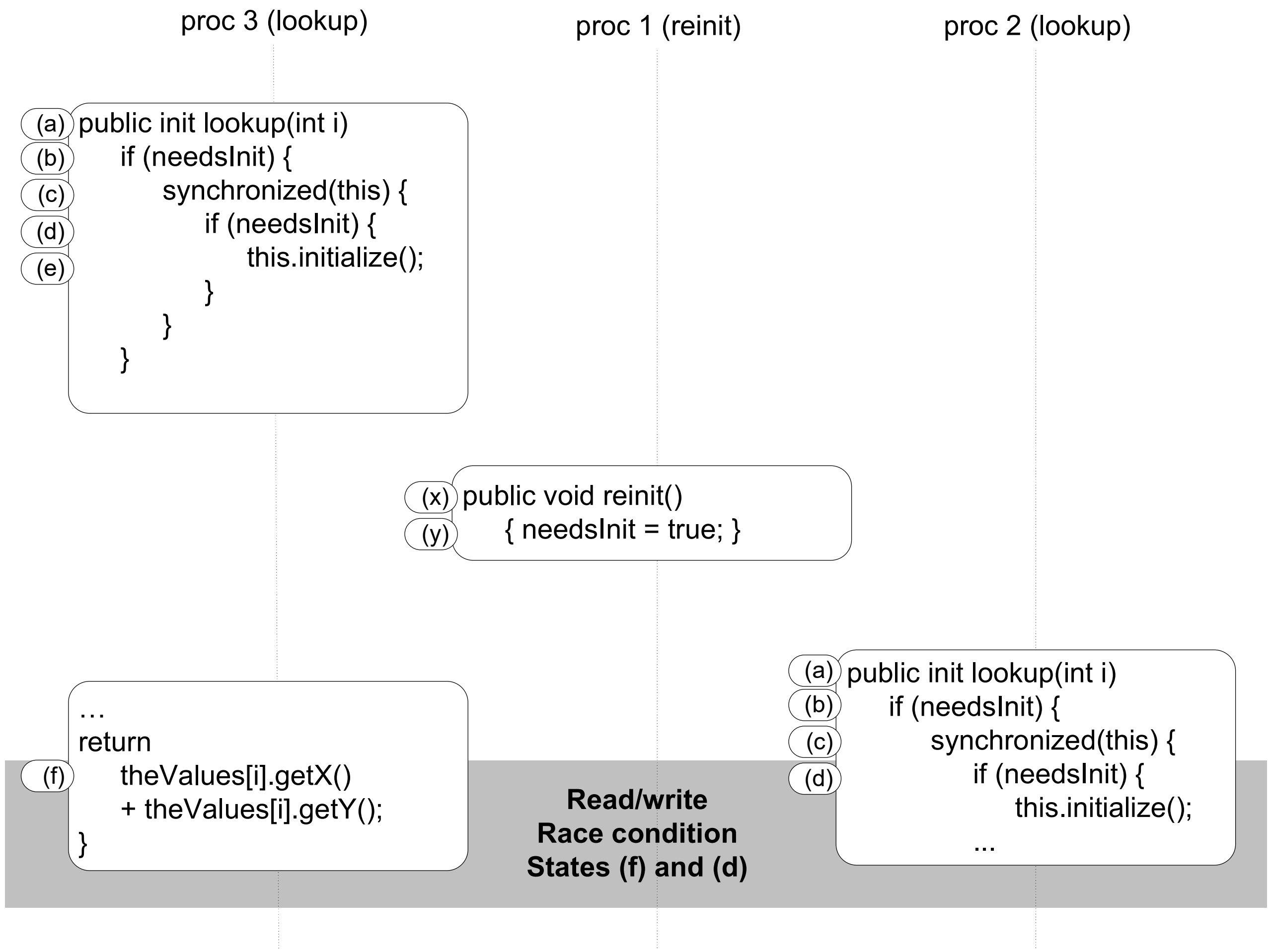
- **Spin**
  - Depth-first search of possible executions of the model
  - Explores 51 states and 92 state transitions in 0.16 seconds
  - Finds a sequence of 17 transitions from the initial state of the model to a state in which one of the assertions in the model evaluates to false



```
Depth=10 States=51 Transitions=92 Memory=2.302
pan: assertion violated  !(modifying) (at depth 17)
pan: wrote pan_in.trail
(Spin Version 4.2.5 -- 2 April 2005)
…
0.16 real          0.00 user          0.03 sys
```

DEPENDABLE SOFTWARE LABORATORY

Counterexample: Interpret the Output



proc 3 (lookup):

```
(a) public init lookup(int i)
(b)     if (needsInit) {
(c)         synchronized(this) {
(d)             if (needsInit) {
(e)                 this.initialize();
                }
            }
        }
```

proc 1 (reinit):

```
(x) public void reinit()
(y)     { needsInit = true; }
```

proc 3 (lookup):

```
…
return
(f)     theValues[i].getX()
        + theValues[i].getY();
}
```

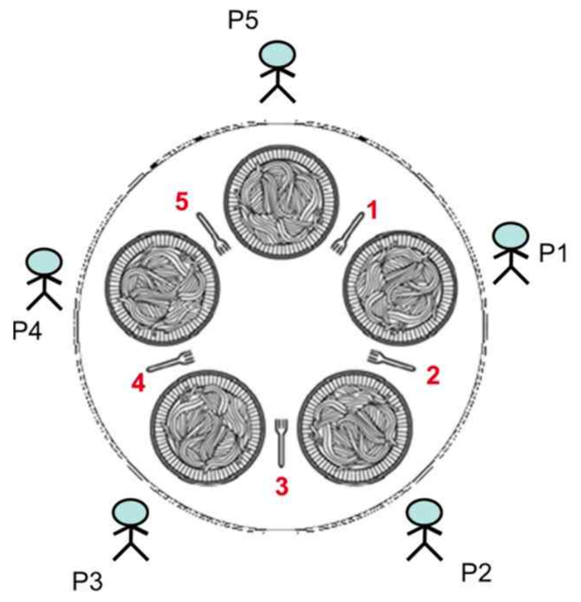
proc 2 (lookup):

```
(a) public init lookup(int i)
(b)     if (needsInit) {
(c)         synchronized(this) {
(d)             if (needsInit) {
                    this.initialize();
                ...
```

The State Space Explosion Problem

- **Dining Philosophers** - looking for deadlock with SPIN

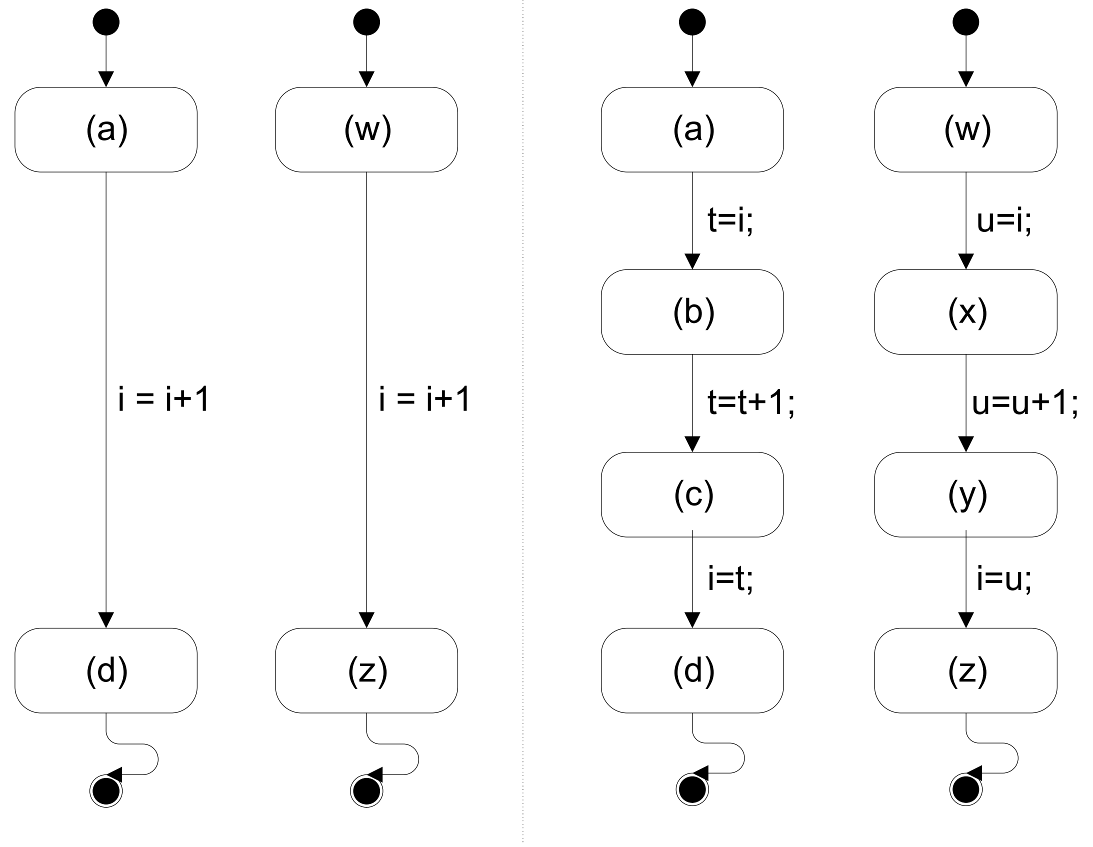
5 phils+forks	145 states Deadlock found
10 phils+forks	18,313 states Error trace too long to be useful
15 phils+forks	148,897 states Error trace too long to be useful



The Model Correspondence Problem

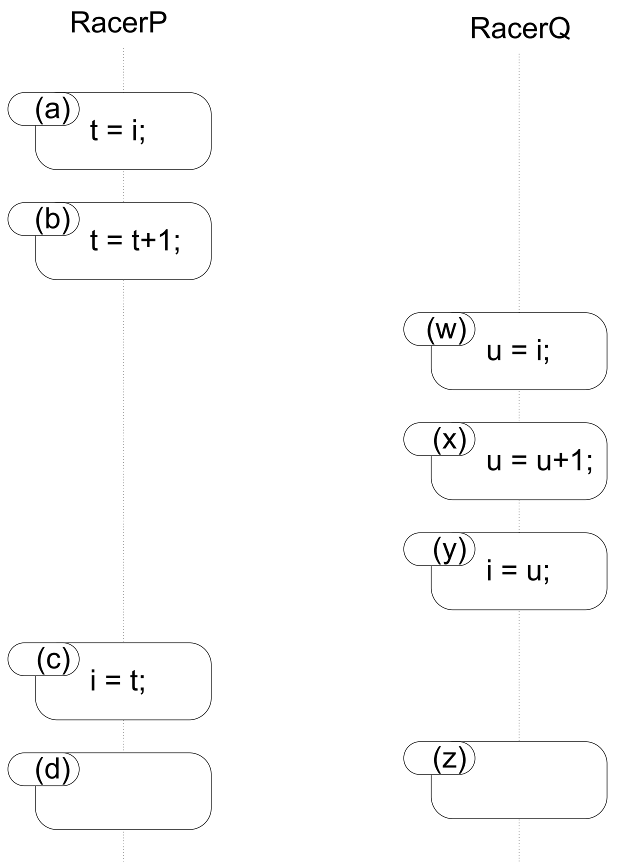
- Verifying correspondence between model and program
  - **Extract the model from the source code with verified procedures**
    - Blindly mirroring all details → state space explosion
    - Omitting crucial detail → "false alarm" reports
  - **Conformance testing**
    - Combination of FSV and testing is a good tradeoff.
- **Produce the source code automatically from the model**
  - Most applicable within well-understood domains
  - A motivation of **MBD** (Model-Based Development)

Granularity of Modeling



Analysis of Different Models

- We can find the race only with fine-grain models.



Looking for Appropriate Granularity

- **Compilers** may rearrange the order of instruction.
  - A simple store of a value into a memory cell may be compiled into a store into a local register, with the actual store to memory appearing later.
  - Two loads or stores to different memory locations may be reordered for reasons of efficiency.
  - Parallel computers may place values initially in the cache memory of a local processor, and only later write into a memory area.
  - Even representing each memory access as an individual action is not always sufficient.

- Example: Double-check idiom only for lazy initialization
  - Spin assumes that memory accesses occur in the order given in the PROMELA program, and we code them in the same order as the Java program.
  - But Java does not guarantee that they will be executed in that order.
  - And SPIN would find a flaw.

Intentional Models

- Enumerating all reachable states is a limiting factor of finite state verification.
- We can reduce the space by using intentional (**symbolic**) representations.
  - Describing sets of reachable states without enumerating each one individually
  - Intentional models do not necessarily grow with the size of the set they represent.
- Example: a set of Integers
  - **Enumeration** : {2, 4, 6, 8, 10, 12, 14, 16, 18}
  - **Intentional representation** : {x∈N | x mod 2 =0 and 0<x<20} ← "***characteristic function***"

OBDD: A Useful Intentional Model

- **OBDD** **(Ordered Binary Decision Diagram)**
  - A compact representation of Boolean functions

- Characteristic function for transition relations
  - Transitions = pairs of states
  - Function from pairs of states to Booleans is true, if there is a transition between the pair.
  - Built iteratively by breadth-first expansion of the state space:
    - Create a representation of the whole set of states reachable in k+1 steps from the set of states reachable in k steps
    - OBDD stabilizes when all the transitions that can occur in the next step are already represented in the OBDD.

From OBDD to Symbolic Checking

- Intentional representation itself is not enough.
  - We must have an algorithm for determining whether it satisfies the property we are checking.

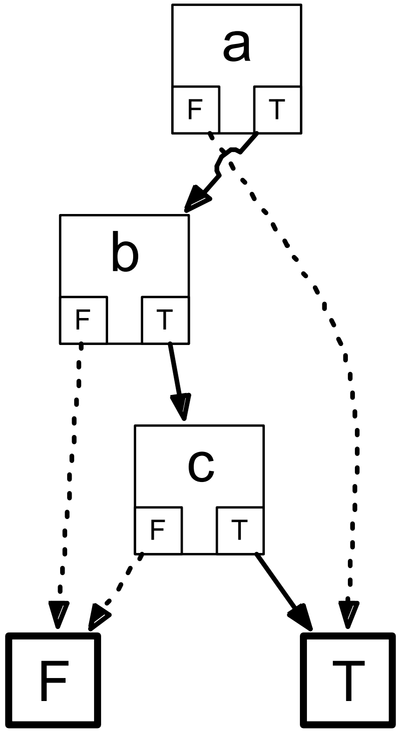
- Example: A set of communicating state machines using OBDD
  - Representing the transition relation of a set of communicating state machines (Model)
  - Modeling a class of temporal logic specification formulas (Specification)

- We going to combine OBDD representations of model and specification to produce a representation of just the set of transitions leading to a violation of the specification.
  - If the set is empty, the property has been verified.

Representing Transition Relations as Boolean Functions

- BDD is a decision tree that has been transformed into an acyclic graph by merging nodes leading to identical sub-trees.

- $a \Rightarrow$ b and c
  not(a) or (b and c)



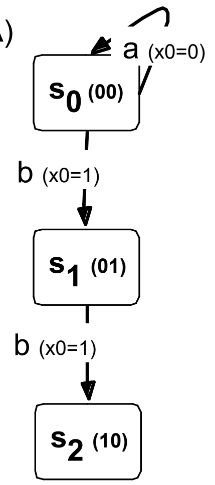
Representing Transition Relations as Boolean Functions : Steps

A. Assign a label to each state

B. Encode transitions

C. The transition tuples correspond to paths leading to true, and all other paths lead to false.

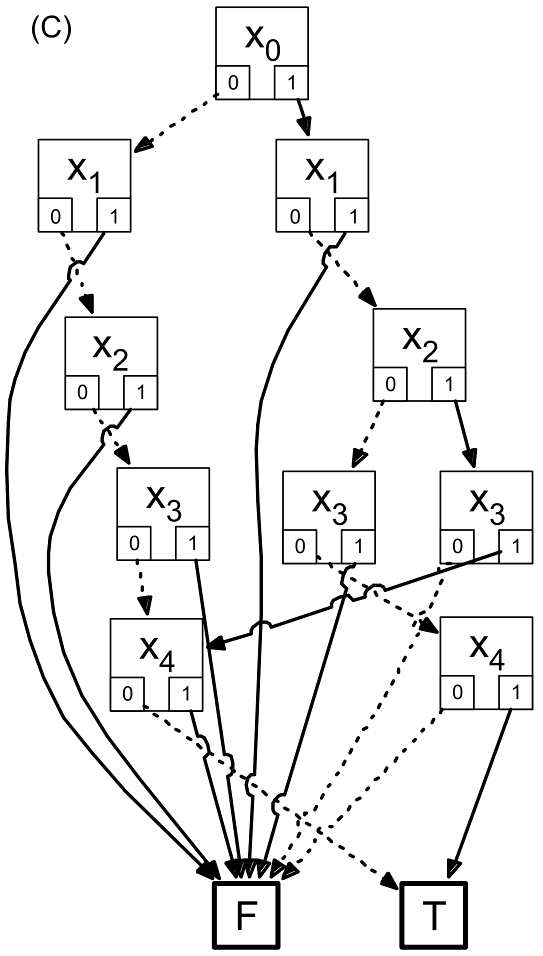
(A)



(B)

x_0	x_1 x_2	x_3 x_4
0	0 0	0 0
1	0 0	0 1
1	0 1	1 0
sym	from state	to state

(C)



Intentional vs. Explicit Representations

- Worst case:
  - Given a large set S of states,
  - A representation capable of distinguishing each subset of S cannot be more compact on average than the representation that simply lists elements of the chosen subset.

- Intentional representations work well when they exploit structure and regularity of the state space.

Model Refinement

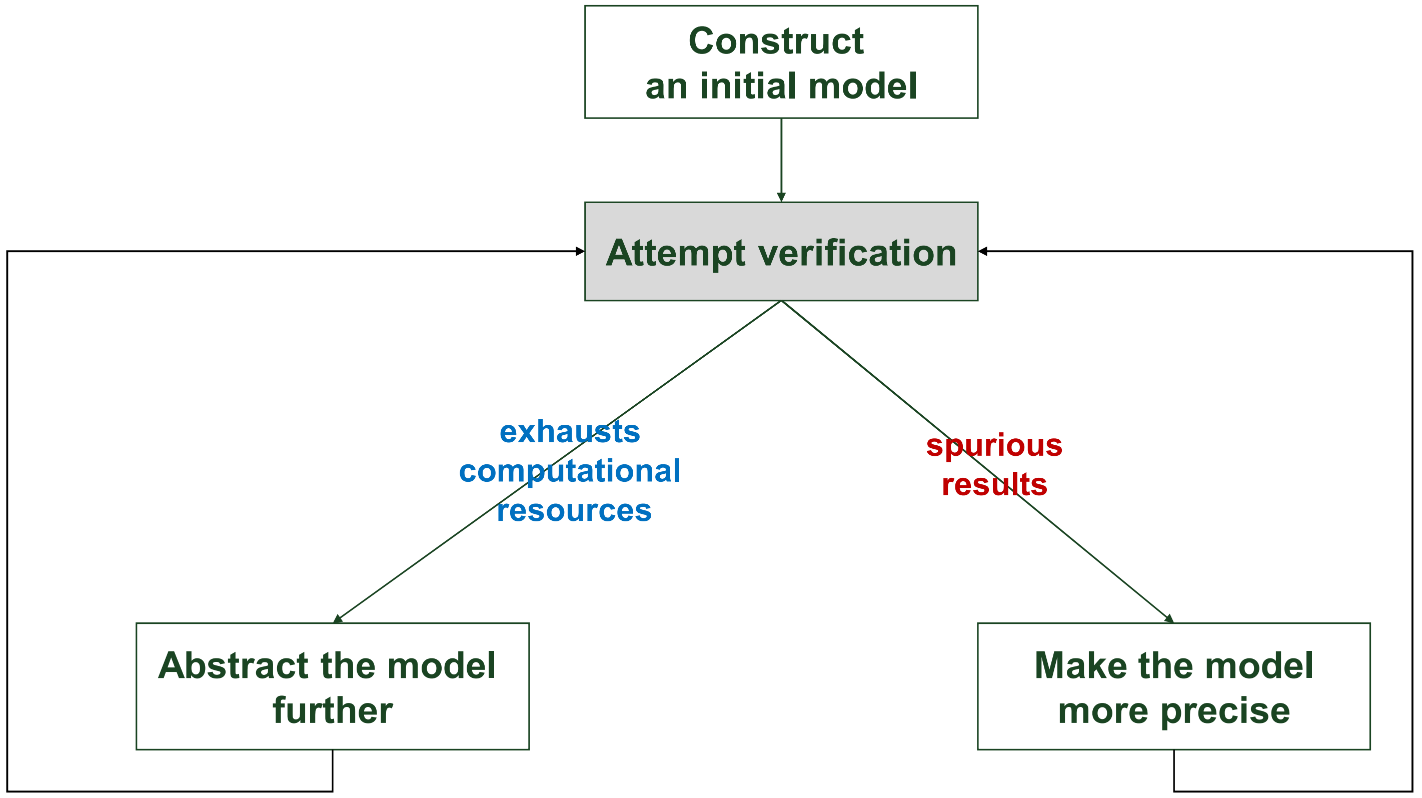
- Construction of finite state models should balance precision and efficiency.

- Often the first model is unsatisfactory.
  - Case 1: Report potential failures that are obviously impossible
  - Case 2: Exhaust resources before producing any result

- Minor differences in the model can have large effects on tractability of the verification procedure.

- **Finite state verification as iterative process** is required.

Iteration Verification Process


Construct an initial model
Attempt verification
exhausts computational resources
spurious results
Abstract the model further
Make the model more precise

Refinement 1: Adding Details to the Model

M_1 |= P Initial (coarse grain) model
(The counter example that violates P is possible in M_1,
but does not correspond to an execution of the real program.)

M_2 |= P Refined (more detailed) model
(the counterexample above is not possible in M_2, but a new
counterexamples violates M_2, and does not correspond to an
execution of the real program too.)

....

M_k |= P Refined (final) model
(the counter example that violates P in M_k corresponds to an
execution in the real program.)

Refinement 2: Add Premises to the Property

Initial (coarse grain) model

$M \models P$

Add a constraint C_1 that eliminates the bogus behavior

$M \models C_1 \Rightarrow P$

$M \models (C_1 \text{ and } C_2) \Rightarrow P$

....

Until the verification succeeds or produces a valid counter example

SOFTWARE TESTING TECHNIQUES

- Test Case Selection and Adequacy
- Functional Testing
- Combinatorial Testing
- Structural Testing
- Data-Flow Testing
- Model-Based Testing
- Fault-Based Testing
- Test Execution

Code Review

Clean Code

Refactoring

코드 기반 테스트

테스트 계획서

TDD

구조 기반 테스트

소프트웨어 테스트

Boundary Value 테스트

3점 접합

TFD

블랙박스 테스트

구조 테스트

테스트 실행

CI / CD

모델 기반 테스트

테스트 결과 보고서

N-Pairwise 테스트

테스트 Stub

기능 테스트

화이트박스 테스트

Static Code Analysis

스펙 기반 테스트

CTIP(Continuous Test & Integration Platform)

테스트 케이스 자동 생성

테스트 오라클

Code Review

테스트 케이스

Category-Partitioning 테스트

요구공학

테스트 데이터

Coverage Criteria

xUnit

테스트 자동화 도구

테스트 데이터 자동 생성

테스트 Scaffolding

시스템 테스트 계획서

테스트 드라이버

Coverage 측정

gTest

QAS

테스트 명세서

HW/SW Co-테스트

비기능/품질 테스트

Cyclomatic Complexity

시뮬레이션 기반 테스트

Test Case Selection and Adequacy

Overview

- What we want to know is a real way of **measuring effectiveness of testing**.
  - ***"If the system passes an adequate suite of test cases, then it must be correct."***

- But that's **impossible**.
  - The adequacy of test suites is provably undecidable.

- Therefore, we'll have to settle on **weaker proxies** for **adequacy**.

Terminologies in Testing

Terms	Descriptions
Test case	a set of inputs, execution conditions, and a pass/fail criterion
Test case specification (Test specification)	a requirement to be satisfied by one or more test cases
Test obligation	a partial test case specification, requiring some property deemed important to thorough testing
Test suite	a set of test cases
Test (**Test execution)**	the activity of executing test cases and evaluating their results
Adequacy criterion	a predicate that is true (satisfied) or false of a ⟨program, test suite⟩ pair

Source of Test Specification

Testing	Other similar names (but not the same exactly)	Source of test specification / Examples
Functional Testing	Black box testing Specification-based testing	Software specification
		If specification requires robust recovery from power failure, test obligations should include simulated power failure.
Structural Testing	White box testing Code-based testing	Source code
		Traverse each program loop one or more times
Model-based Testing		Models of system • Models used in specification or design • Models derived from source code
		Exercise all transitions in communication protocol model
Fault-basedTesting		Hypothesized faults, Common bugs
		Check for buffer overflow handling (common vulnerability) by testing on very large inputs

Adequacy Criteria

- Adequacy criterion = Set of test obligations

- **A test suite satisfies an adequacy criterion,** ***iff***
  - *All the tests succeed (pass), and*
  - *Every test obligation in the criterion is satisfied by at least one of the test cases in the test suite.*

  - Example:
    - ***"The statement coverage adequacy criterion is satisfied by test suite S for program P, if each executable statement in P is executed by at least one test case in S, and the outcome of each test execution was pass."***

Satisfiability

- Often no test suite can satisfy a criterion for a given program.
  - Example:
    - Defensive programming style includes “can’t happen” sanity checks.
      - if (z < 0) {
          throw new LogicError (“z must be positive here!”)
        }
    - For this program, no test suite can satisfy statement coverage.

- **Two ways** of coping with the **unsatisfiability of adequacy criteria**
  - **A** : Exclude any unsatisfiable obligation from the criterion
  - **B** : Measure the extent to which a test suite approaches an adequacy criterion

Coping with the Unsatisfiability

- Approach A
  - Exclude any unsatisfiable obligation from the criterion
  - Example:
    - Modify statement coverage to require execution only of statements which can be executed
  - But we can't know for sure which are executable or not.

- **Approach B**
  - Measure the extent to which a test suite approaches an adequacy criterion
  - Example
    - If a test suite satisfies 85 of 100 obligations, we have reached 85% coverage.
  - Terms:
    - An adequacy criterion is satisfied or not.
    - **A coverage measure is the fraction of satisfied obligations.**

Coverage

- Measuring **coverage** **(% of satisfied test obligations)** can be a useful indicator of
  - Progress toward a thorough test suite (thoroughness of test suite), and
  - Trouble spots requiring more attention in testing.

- But coverage is only a proxy for thoroughness or adequacy.
  - It's easy to improve coverage without improving a test suite (much easier than designing good test cases)
  - The only measure that really matters is (cost-) effectiveness.

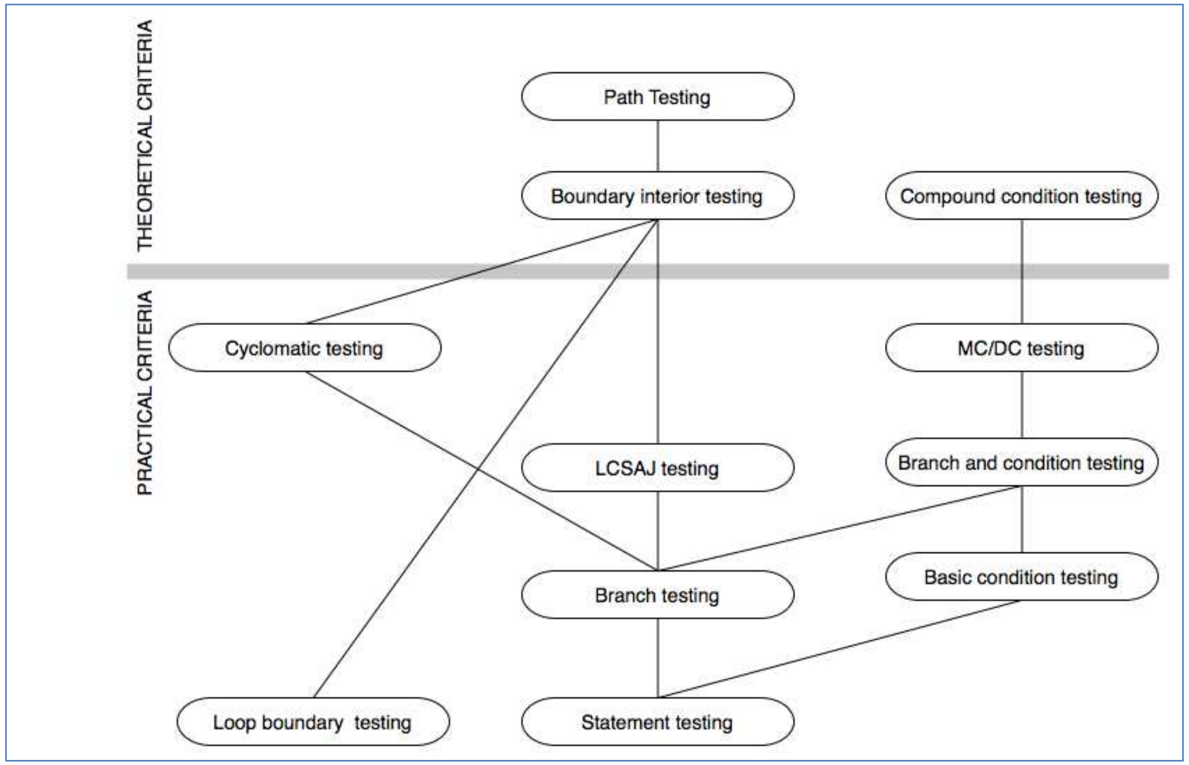
Comparing Criteria

- Can we distinguish stronger from weaker adequacy criteria?

- Analytical approach
  - Describe conditions under which one adequacy criterion is provably stronger than another
  - Just a piece of the overall “effectiveness” question
  - Stronger = gives stronger guarantees → **Subsumes** relation
    - Working from easier to harder levels of coverage, but not a direct indication of quality.

Subsumes Relation

- Test adequacy criterion A subsumes test adequacy criterion B iff, for every program P, every test suite satisfying A with respect to P also satisfies B with respect to P.



Use of Adequacy Criteria

- Test selection approaches (**Selection**)
  - Guidance in devising a thorough test suite
    - E.g., A specification-based testing criterion may suggest test cases covering representative combinations of values.
- Revealing missing tests (**Measurement**)
  - Post hoc analysis: What might I have missed with this test suite?
- **Often in combination**
  - **Design test suite from specifications, then use structural criterion (e.g., coverage of all branches) to highlight missed logic**

Functional Testing

Functional Testing

- **Functional testing**
  - Deriving test cases **from program specifications**
  - '*Functional*' refers to the source of information used in test case design, not to what is tested.
  - Functional specification is a formal or informal description of intended program behavior.

- Also known as:
  - **Specification-based testing** (from specifications)
  - **Black-box testing** (no view of source code)

Systematic Testing vs. Random Testing

- **Random (uniform) testing**
  - Pick possible inputs uniformly
  - Avoids designer's bias
  - But treats all inputs as equally valuable.

- **Systematic (non-uniform) testing**
  - ***Try to select inputs that are especially valuable***
  - Usually by choosing representatives of classes that are apt to fail often or not at all

- **Functional testing is a systematic (partition-based) testing strategy.**

Why Not Random Testing?

- Due to non-uniform distribution of faults
  - Example:
    - Java class “roots” applies quadratic equation

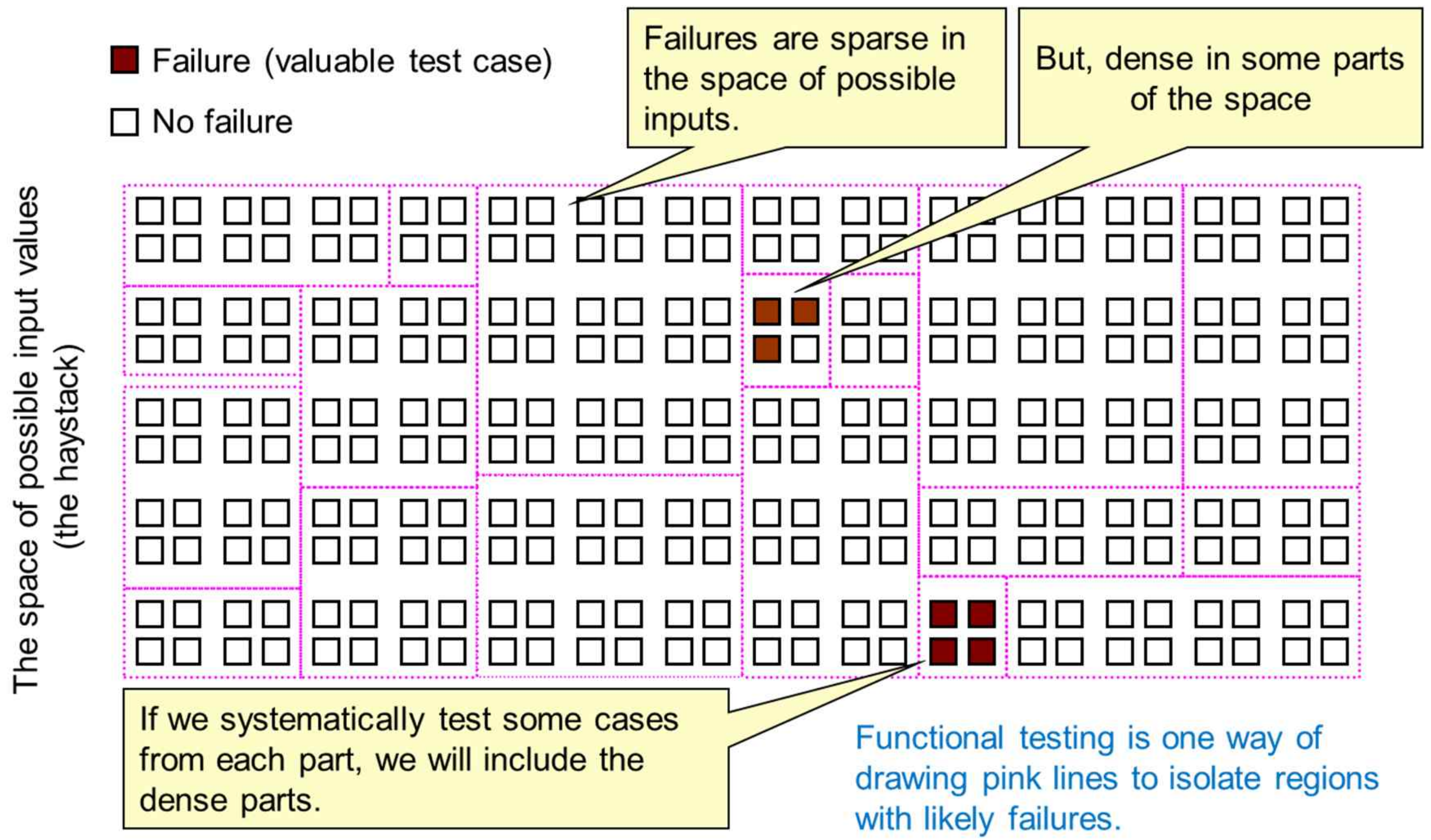
$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

  - Supposed an incomplete implementation logic:
    - Program does not properly handle the case in which b^2 - 4ac = 0 and a = 0
  - Failing values are sparse in the input space: needles in a very big haystack
  - Random sampling is unlikely to choose a=0 and b=0.

Purpose of Testing

- Our goal is to find needles and remove them from hay.
  - Look systematically (non-uniformly) for needles.
  - We need to use everything we know about needles.
    - E.g., Are they heavier than hay? Do they sift to the bottom?
- To estimate the proportion of needles to hay, **sample randomly(uniformly)**.
  - Reliability estimation requires unbiased samples for valid statistics, but that's not our goal.

Systematic Partition Testing



■ Failure (valuable test case)
□ No failure
Failures are sparse in the space of possible inputs.
But, dense in some parts of the space
The space of possible input values (the haystack)
If we systematically test some cases from each part, we will include the dense parts.

Functional testing is one way of drawing pink lines to isolate regions with likely failures.

Principles of Systematic Partitioning

- Exploit some knowledge to choose samples that are more likely to include "*special*" or "*trouble-prone*" regions of the input space
  - Failures are sparse in the whole input space.
  - But we may find regions in which they are dense.

- (Quasi-) Partition testing: separates the input space into classes whose union is the entire space
  - Sampling each class in the quasi-partition selects at least one input that leads to a failure, revealing the fault.
  - Seldom guaranteed; We depend on experience-based heuristics.

A Systematic Approach: Functional Testing

- Functional testing uses the **specification** (formal or informal) to partition the input space.
  - For example, the specification of “roots” program suggests division between cases with zero, one, and two real roots.
  - Test each category and boundaries between categories
    - No guarantees, but experience suggests failures often lie at the boundaries. (as in the “roots” program)

- **Functional Testing is a base-line technique for designing test cases.**
  - Timely
    - Often useful in refining specifications and assessing testability before code is written
  - Effective
    - Find some classes of fault (e.g., missing logic) that can elude other approaches
  - Widely applicable
    - To any description of program behavior serving as specification
    - At any level of granularity from module to system testing
  - Economical
    - Typically, less expensive to design and execute than structural (code-based) test cases

Functional Test vs. Structural Test

- Different testing strategies are most effective for different classes of faults.

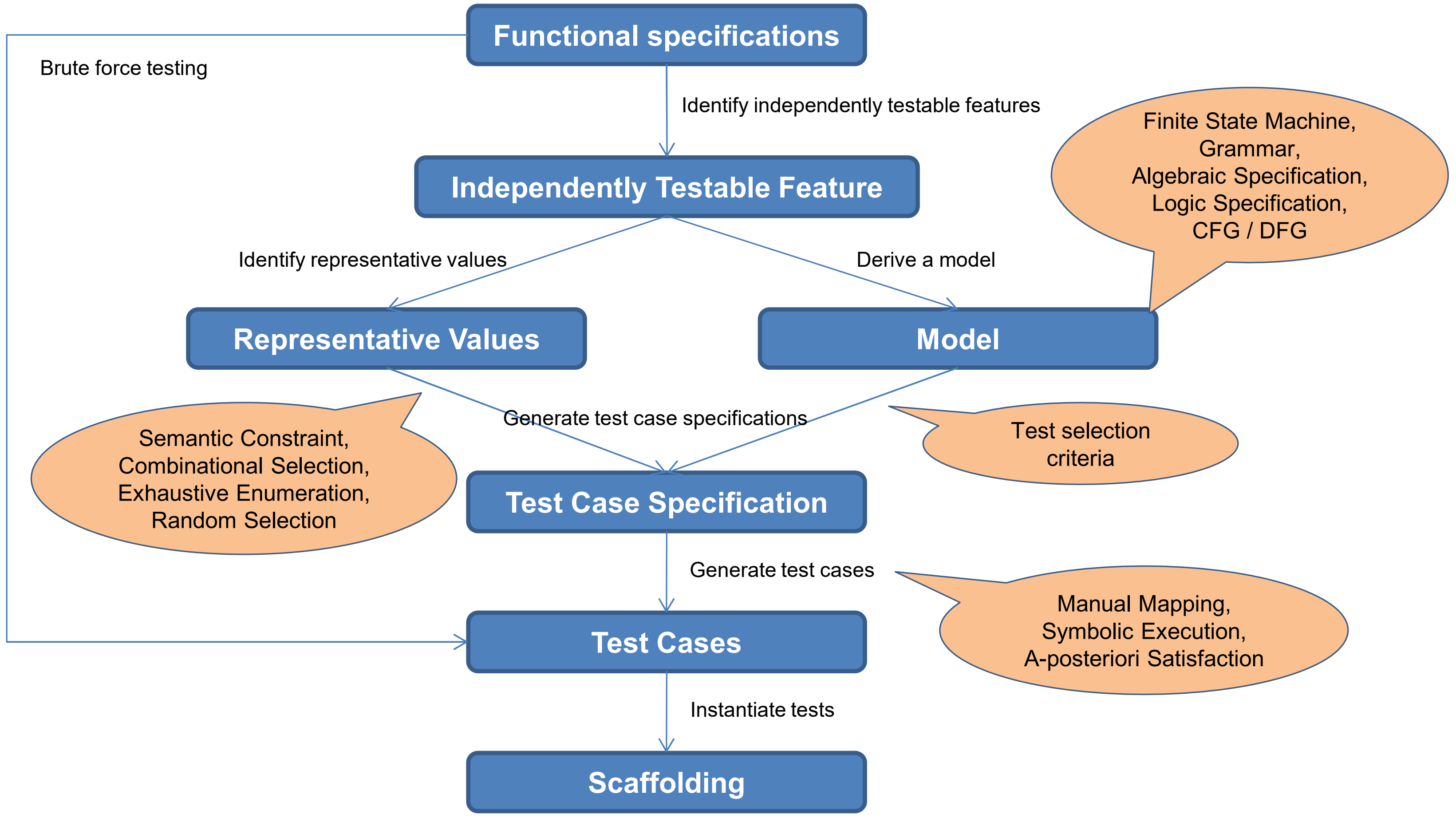
- **Functional testing** is best for missing logic faults.
  - A common problem: Some program logic was simply forgotten.
  - Structural testing will never focus on code that isn't there.

- Functional test applies at all granularity levels.

– Unit	(from module interface spec)
– Integration	(from API or subsystem spec)
– System	(from system requirements spec)
– Regression	(from system requirements + bug history)

- **Structural test** design applies to relatively small parts of a system.
  - Unit and integration testing

Main Steps of Functional Program Testing


Functional specifications
Brute force testing
Identify independently testable features
Independently Testable Feature
Finite State Machine, Grammar, Algebraic Specification, Logic Specification, CFG / DFG
Identify representative values
Derive a model
Representative Values
Model
Semantic Constraint, Combinational Selection, Exhaustive Enumeration, Random Selection
Generate test case specifications
Test selection criteria
Test Case Specification
Generate test cases
Manual Mapping, Symbolic Execution, A-posteriori Satisfaction
Test Cases
Instantiate tests
Scaffolding

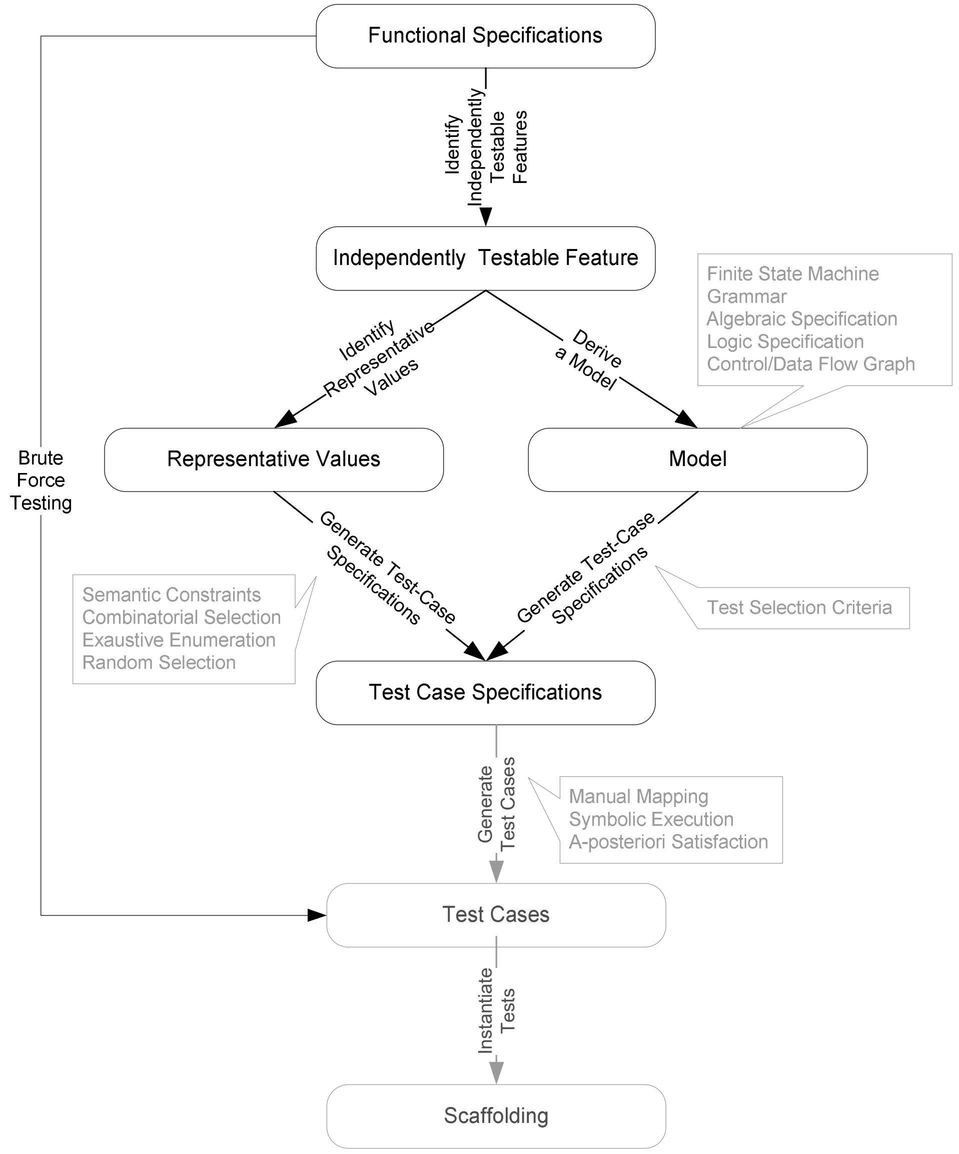
From Specifications to Test Cases

1. Identify independently testable features (categories)
   - If the specification is large, break it into independently testable features.

2. Identify representative classes of values, or derive a model of behavior
   - Often simple input/output transformations don't describe a system.
   - We use models in program specification, in program design, and in test design too.

3. Generate test case specifications
   - Typically, combinations of input values or model behaviors

4. Generate test cases and instantiate tests



Functional Specifications
Identify Independently Testable Features
Independently Testable Feature
Identify Representative Values
Derive a Model
Finite State Machine
Grammar
Algebraic Specification
Logic Specification
Control/Data Flow Graph
Representative Values
Model
Brute Force Testing
Generate Test-Case Specifications
Generate Test-Case Specifications
Semantic Constraints
Combinatorial Selection
Exaustive Enumeration
Random Selection
Test Selection Criteria
Test Case Specifications
Generate Test Cases
Manual Mapping
Symbolic Execution
A-posteriori Satisfaction
Test Cases
Instantiate Tests
Scaffolding

Combinatorial Testing

Overview

- **Combinatorial testing** identifies distinct attributes that can be varied In data, environment or configuration.
  - Example:
    - Browser could be "IE" or "Firefox"
    - Operating system could be "Vista", "XP" or "OSX"

- Combinatorial testing systematically generates combinations to be tested.
  - Example:
    - IE on Vista, IE on XP, Firefox on Vista, Firefox on OSX, etc.
  - Rationale: Test cases should be varied and include possible "corner cases".

Key Ideas in Combinatorial Approaches

- **Category-partition testing**
  - Separate (manual) identification of values that characterize the input space from (automatic) generation of combinations for test cases

- **Pairwise testing**
  - Systematically test interactions among attributes of the program input space with a relatively small number of test cases

- Catalog-based testing
  - Aggregate and synthesize the experience of test designers in a particular organization or application domain, to aid in identifying attribute values

1. Category-Partition Testing

1. Decompose the specification into independently testable features
   - for each feature, identify parameters and environment elements
   - for each parameter and environment element, identify elementary characteristics (→ categories)

2. Identify representative (classes of) values
   - for each characteristic(category), identify classes of values
     - normal values
     - boundary values
     - special values
     - error values

3. Generate test case specifications

An Example: “Check Configuration”

- In the Web site of a computer manufacturer, ‘checking configuration’ checks the validity of a computer configuration.
  - Two parameters:
    - Model
    - Set of Components

Model: A model identifies a specific product and determines a set of constraints on available components. Models are characterized by logical slots for components, which may or may not be implemented by physical slots on a bus. Slots may be required or optional. Required slots must be assigned with a suitable component to obtain a legal configuration, while optional slots may be left empty or filled depending on the customer’s needs.

Example: The required “slots” of the Chipmunk C20 laptop computer include a screen, a processor, a hard disk, memory, and an operating system. (Of these, only the hard disk and memory are implemented using actual hardware slots on a bus.) The optional slots include external storage devices such as a CD/DVD writer.

Set of Components: A set of (slot, component) pairs, correspond to the required and optional slots of the model. A component is a choice that can be varied within a model, and which is not designed to be replaced by the end user. Available components and a default for each slot is determined by the model. The special value empty is allowed (and may be the default selection) for optional slots. In addition to being compatible or incompatible with a particular model and slot, individual components may be compatible or incompatible with each other.

Example: The default configuration of the Chipmunk C20 includes 20 gigabytes of hard disk; 30 and 40 gigabyte disks are also available. (Since the hard disk is a required slot, empty is not an allowed choice.) The default operating system is RodentOS 3.2, personal edition, but RodentOS 3.2 mobile server edition may also be selected. The mobile server edition requires at least 30 gigabytes of hard disk.

Step 1: Identify Independently Testable Features and Parameter Characteristics

- **Choosing categories**
  - No hard-and-fast rules for choosing categories!
  - Not a trivial task

- Categories reflect test designer's judgment.
  - Which classes of values may be treated differently by an implementation.

- Choosing categories well requires experience and knowledge of the application domain and product architecture.

Identify Independently Testable Units

Parameters	Categories
Model	Model number
	Number of required slots for selected model (#SMRS)
	Number of optional slots for selected model (#SMOS)
Components	Correspondence of selection with model slots
	Number of required components with selection ≠ empty
	Required component selection
	Number of optional components with selection ≠ empty
	Optional component selection
Product Database	Number of models in database (#DBM)
	Number of components in database (#DBC)

Step 2: Identify Representative Values

- Identify representative classes of values for each of the categories

- Representative values may be identified by applying
  - Boundary value testing
    - Select extreme values within a class
    - Select values outside but as close as possible to the class
    - Select interior (non-extreme) values of the class
  - Erroneous condition testing
    - Select values outside the normal domain of the program

Representative Values: Model

- **Model number**
  - Malformed
  - Not in database
  - Valid
- **Number of required slots for selected model (#SMRS)**
  - 0
  - 1
  - Many
- **Number of optional slots for selected model (#SMOS)**
  - 0
  - 1
  - Many

Representative Values: Components

- **Correspondence of selection with model slots**
  - Omitted slots
  - Extra slots
  - Mismatched slots
  - Complete correspondence

- **Number of required components with non-empty selection**
  - 0
  - < number required slots
  - = number required slots

- **Required component selection**
  - Some defaults
  - All valid
  - ≥ 1 incompatible with slots
  - ≥ 1 incompatible with another selection
  - ≥ 1 incompatible with model
  - ≥ 1 not in database

Representative Values: Components

- **Number of optional components with non-empty selection**
  - 0
  - < #SMOS
  - = #SMOS

- **Optional component selection**
  - Some defaults
  - All valid
  - ≥ 1 incompatible with slots
  - ≥ 1 incompatible with another selection
  - ≥ 1 incompatible with model
  - ≥ 1 not in database

Representative Values: Product Database

- **Number of models in database (#DBM)**
  - 0
  - 1
  - Many

- **Number of components in database (#DBC)**
  - 0
  - 1
  - Many

  - Note 0 and 1 are unusual (special) values.
    - They might cause unanticipated behavior alone or in combination with particular values of other parameters.

Step 3: Generate Test Case Specifications

- A combination of values for each category corresponds to a test case specification.
  - In the example, we have **314,928 test cases**.
  - Most of which are impossible.
    - Example: zero slots and at least one incompatible slot

- Need to introduce **constraints** in order to rule out impossible combinations and reduce the size of the test suite.
  - **Error constraints**
  - **Property constraints**
  - **Single constraints**

Error Constraints

- **[error]** indicates a value class that corresponds to an erroneous values.
  - Need to be tried only once

- Error value class
  - No need to test all possible combinations of errors, and one test is enough.

Model number

Malformed	**[error]**
Not in database	**[error]**
Valid	

Correspondence of selection with model slots

Omitted slots	**[error]**
Extra slots	**[error]**
Mismatched slots	**[error]**
Complete correspondence	

Number of required comp. with non-empty selection

0	**[error]**
< number of required slots	**[error]**

Required comp. selection

≥ 1 not in database	**[error]**

Number of models in database (#DBM)

0	**[error]**

Number of components in database (#DBC)

0	**[error]**

Error constraints reduce test suite from 314,928 to 2,711 test cases

Property Constraints

- **Constraint [property] [if-property]** rule out invalid combinations of values.
  - [property] groups values of a single parameter to identify subsets of values with common properties.
  - [if-property] bounds the choices of values for a category that can be combined with a particular value selected for a different category.

Number of required slots for selected model (#SMRS)	
1	**[property RSNE]**
Many	**[property RSNE] [property RSMANY]**
Number of optional slots for selected model (#SMOS)	
1	**[property OSNE]**
Many	**[property OSNE] [property OSMANY]**
Number of required comp. with non-empty selection	
0	**[if RSNE] [error]**
< number required slots	**[if RSNE] [error]**
= number required slots	**[if RSMANY]**
Number of optional comp. with non-empty selection	
< number required slots	**[if OSNE]**
= number required slots	**[if OSMANY]**

from 2,711 to 908 test cases

Single Constraints

- **[single]** indicates a value class that test designers choose to test only once to reduce the number of test cases.

- Example
  - Value some default for required component selection and optional component selection may be tested only once despite not being an erroneous condition.

- Note
  - Single and error have the same effect but differ in rationale.
  - Keeping them distinct is important for documentation and regression testing.

Number of required slots for selected model (#SMRS)

– 0	[single]
– 1	[property RSNE] [single]

Number of optional slots for selected model (#SMOS)

– 0	[single]
– 1	[single] [property OSNE]

Required component selection

– Some default	[single]

Optional component selection

– Some default	[single]

Number of models in database (#DBM)

– 1	[single]

Number of components in database (#DBC)

– 1	[single]

from 908 to 69 test cases

Check Configuration – Summary of Categories

Parameter Model

- Model number

Malformed	[error]
Not in database	[error]
Valid	

- Number of required slots for selected model (#SMRS)

0	[single]
1	[property RSNE] [single]
Many	[property RSNE] [property RSMANY]

- Number of optional slots for selected model (#SMOS)

0	[single]
1	[property OSNE] [single]
Many	[property OSNE] [property OSMANY]

Environment Product data base

- Number of models in database (#DBM)

0	[error]
1	[single]
Many	

- Number of components in database (#DBC)

0	[error]
1	[single]
Many	

Parameter Component

- Correspondence of selection with model slots

Omitted slots	[error]
Extra slots	[error]
Mismatched slots	[error]
Complete correspondence	

- # of required components (selection $\neq$ empty)

0	[if RSNE] [error]
< number required slots	[if RSNE] [error]
= number required slots	[if RSMANY]

- Required component selection

Some defaults	[single]
All valid	
$\geq$ 1 incompatible with slots	
$\geq$ 1 incompatible with another selection	
$\geq$ 1 incompatible with model	
$\geq$ 1 not in database	[error]

- # of optional components (selection $\neq$ empty)

0	
< #SMOS	[if OSNE]
= #SMOS	[if OSMANY]

- Optional component selection

Some defaults	[single]
All valid	
$\geq$ 1 incompatible with slots	
$\geq$ 1 incompatible with another selection	
$\geq$ 1 incompatible with model	
$\geq$ 1 not in database	[error]

TSL : Test Specification Language

SPECIAL ARTICLE

THE CATEGORY-PARTITION METHOD FOR SPECIFYING AND GENERATING FUNCTIONAL TESTS

A method for creating functional test suites has been developed in which a test engineer analyzes the system specification, writes a series of formal test specifications, and then uses a generator tool to produce test descriptions from which test scripts are written. The advantages of this method are that the tester can easily modify the test specification when necessary, and can control the complexity and number of the tests by annotating the tests specification with constraints.

THOMAS J. OSTRAND and MARC J. BALCER

The goal of functional testing of a software system is to find discrepancies between the actual behavior of the implemented system's functions and the desired behavior as described in the system's functional specification. To achieve this goal requires first, that tests be executed for all of the system's functions, and second, that the tests be designed to maximize the chances of finding errors in the software. Although a particular method or testing group may emphasize one or the other, these two aspects of testing are mutually complementary, and both are necessary for maximally productive testing. It is not enough merely to "cover all the functionality"; the tests must be aimed at the most vulnerable parts of the implementation. For functional testing, this implies testing boundary conditions, special cases, error handlers, and cases where inputs and the system environment interact in potentially dangerous ways. Conversely, it is not enough to write excellent, error-exposing tests for only some of a system's functions, or to write tests aimed only at certain types of errors or certain characteristics of a specification or implementation.

Functional tests can be derived from the software's specifications, from design information, or from the code itself. All three test sources provide useful information, and none of them should be ignored. Code-based tests relate to the modular structure, logic, control flow, and data flow of the software. They have the particular advantage that a program is a formal object, and it is therefore easy to make precise statements about the adequacy or thoroughness of code-based tests. Design-based tests relate to the programming abstractions, data structures, and algorithms used to construct the software. Specification-based tests relate directly to what the software is supposed to do, and therefore are probably the most intuitively appealing type of functional tests.

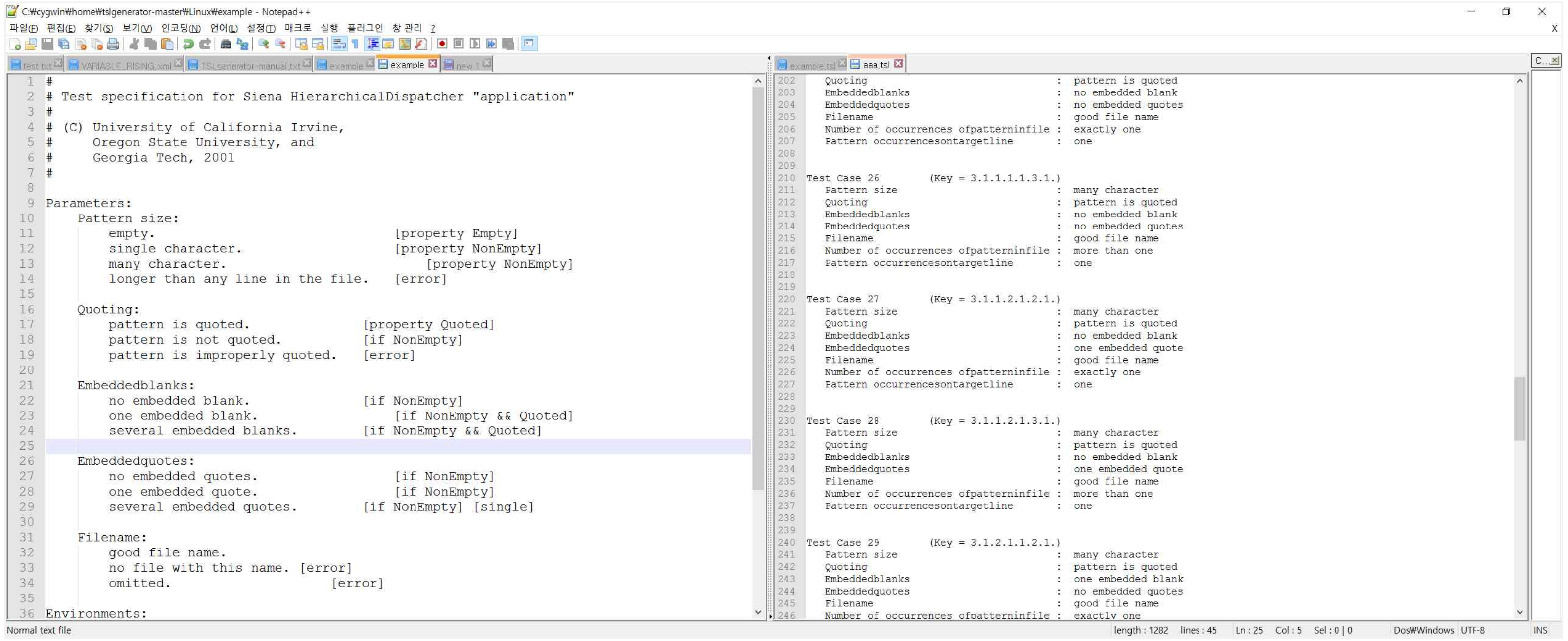
A standard approach to generating specification-based functional tests is first to partition the input domain of a function being tested, and then to select test data from each class of the partition. The idea is that all elements within an equivalence class are essentially the same for the purposes of testing. If the testing's main emphasis is to attempt to show the presence of errors, then the assumption is that any element of a class will expose the error as well as any other one. If the testing's main emphasis is to attempt to give confidence in the software's correctness, then the assumption is that correct results for a single element in a class will provide confidence that all elements in the class would be processed correctly.

A preliminary version of this paper appeared as "Machine-Aided Production of Software Tests," in the Proceedings of the Fifth Annual Pacific Northwest Software Quality Conference, Portland, Oregon, October 19–20, 1987.

© 1988 ACM 0001-0782/88/0600-0676 $1.50

676 Communications of the ACM June 1988 Volume 31 Number 6

- **TSL** **(Test Specification Language)**
  - Category
  - Property List
  - Selector Expression
  - https://github.com/alexorso/tslgenerator



C:₩cygwin₩home₩tslgenerator-master₩Linux₩example - Notepad++
파일(F) 편집(E) 찾기(S) 보기(V) 인코딩(N) 언어(L) 설정(T) 매크로 실행 플러그인 창 관리 ?
test.txt VARIABLE_RISING.xml TSLgenerator-manual.txt example example new 1
example.tsl aaa.tsl
Normal text file
length : 1282 lines : 45 Ln : 25 Col : 5 Sel : 0 | 0 Dos₩Windows UTF-8 INS

```
#
# Test specification for Siena HierarchicalDispatcher "application"
#
# (C) University of California Irvine,
#     Oregon State University, and
#     Georgia Tech, 2001
#

Parameters:
    Pattern size:
        empty.                               [property Empty]
        single character.                    [property NonEmpty]
        many character.                          [property NonEmpty]
        longer than any line in the file.    [error]

    Quoting:
        pattern is quoted.               [property Quoted]
        pattern is not quoted.           [if NonEmpty]
        pattern is improperly quoted.    [error]

    Embeddedblanks:
        no embedded blank.               [if NonEmpty]
        one embedded blank.                  [if NonEmpty && Quoted]
        several embedded blanks.         [if NonEmpty && Quoted]

    Embeddedquotes:
        no embedded quotes.                  [if NonEmpty]
        one embedded quote.                  [if NonEmpty]
        several embedded quotes.         [if NonEmpty] [single]

    Filename:
        good file name.
        no file with this name. [error]
        omitted.                    [error]

Environments:
```

```
   Quoting                               :  pattern is quoted
   Embeddedblanks                        :  no embedded blank
   Embeddedquotes                        :  no embedded quotes
   Filename                              :  good file name
   Number of occurrences ofpatterninfile :  exactly one
   Pattern occurrencesontargetline       :  one


Test Case 26        (Key = 3.1.1.1.1.3.1.)
   Pattern size                          :  many character
   Quoting                               :  pattern is quoted
   Embeddedblanks                        :  no embedded blank
   Embeddedquotes                        :  no embedded quotes
   Filename                              :  good file name
   Number of occurrences ofpatterninfile :  more than one
   Pattern occurrencesontargetline       :  one


Test Case 27        (Key = 3.1.1.2.1.2.1.)
   Pattern size                          :  many character
   Quoting                               :  pattern is quoted
   Embeddedblanks                        :  no embedded blank
   Embeddedquotes                        :  one embedded quote
   Filename                              :  good file name
   Number of occurrences ofpatterninfile :  exactly one
   Pattern occurrencesontargetline       :  one


Test Case 28        (Key = 3.1.1.2.1.3.1.)
   Pattern size                          :  many character
   Quoting                               :  pattern is quoted
   Embeddedblanks                        :  no embedded blank
   Embeddedquotes                        :  one embedded quote
   Filename                              :  good file name
   Number of occurrences ofpatterninfile :  more than one
   Pattern occurrencesontargetline       :  one


Test Case 29        (Key = 3.1.2.1.1.2.1.)
   Pattern size                          :  many character
   Quoting                               :  pattern is quoted
   Embeddedblanks                        :  one embedded blank
   Embeddedquotes                        :  no embedded quotes
   Filename                              :  good file name
   Number of occurrences ofpatterninfile :  exactly one
```

TSL → **40 Test cases**

Category-Partitioning Testing, in Summary

- Category partition testing gives us systematic approaches to
  - Identify characteristics and values (the creative step)
  - Generate combinations (the mechanical step).

- But test suite size grows very rapidly with number of categories.
  - Pairwise (and n-way) combinatorial testing is a non-exhaustive approach.
    - Combine values systematically but not exhaustively.

2. Pairwise Combination Testing

- Category partition works well when intuitive constraints reduce the number of combinations to *a small amount of test cases*.
  - Without many constraints, the number of combinations may be unmanageable.

- **Pairwise combination**
  - Generate combinations that efficiently cover all pairs (triples,…) of classes, instead of exhaustive combinations
  - Rationale:
    - Most failures are triggered by single values or combinations of a few values.
    - Covering pairs (triples,…) reduces the number of test cases, but reveals most faults.

An Example: Display Control

- No constraints reduce the total number of combinations **432 (3x4x3x4x3) test cases**, if we consider all combinations.

Display Mode	Language	Fonts	Color	Screen size
full-graphics	English	Minimal	Monochrome	Hand-held
text-only	French	Standard	Color-map	Laptop
limited-bandwidth	Spanish	Document-loaded	16-bit	Full-size
	Portuguese		True-color	

Pairwise Combination: 17 Test Cases

Language	Color	Display Mode	Fonts	Screen Size
English	Monochrome	Full-graphics	Minimal	Hand-held
English	Color-map	Text-only	Standard	Full-size
English	16-bit	Limited-bandwidth	-	Full-size
English	True-color	Text-only	Document-loaded	Laptop
French	Monochrome	Limited-bandwidth	Standard	Laptop
French	Color-map	Full-graphics	Document-loaded	Full-size
French	16-bit	Text-only	Minimal	-
French	True-color	-	-	Hand-held
Spanish	Monochrome	-	Document-loaded	Full-size
Spanish	Color-map	Limited-bandwidth	Minimal	Hand-held
Spanish	16-bit	Full-graphics	Standard	Laptop
Spanish	True-color	Text-only	-	Hand-held
Portuguese	-	-	Monochrome	Text-only
Portuguese	Color-map	-	Minimal	Laptop
Portuguese	16-bit	Limited-bandwidth	Document-loaded	Hand-held
Portuguese	True-color	Full-graphics	Minimal	Full-size
Portuguese	True-color	Limited-bandwidth	Standard	Hand-held

Adding Constraints

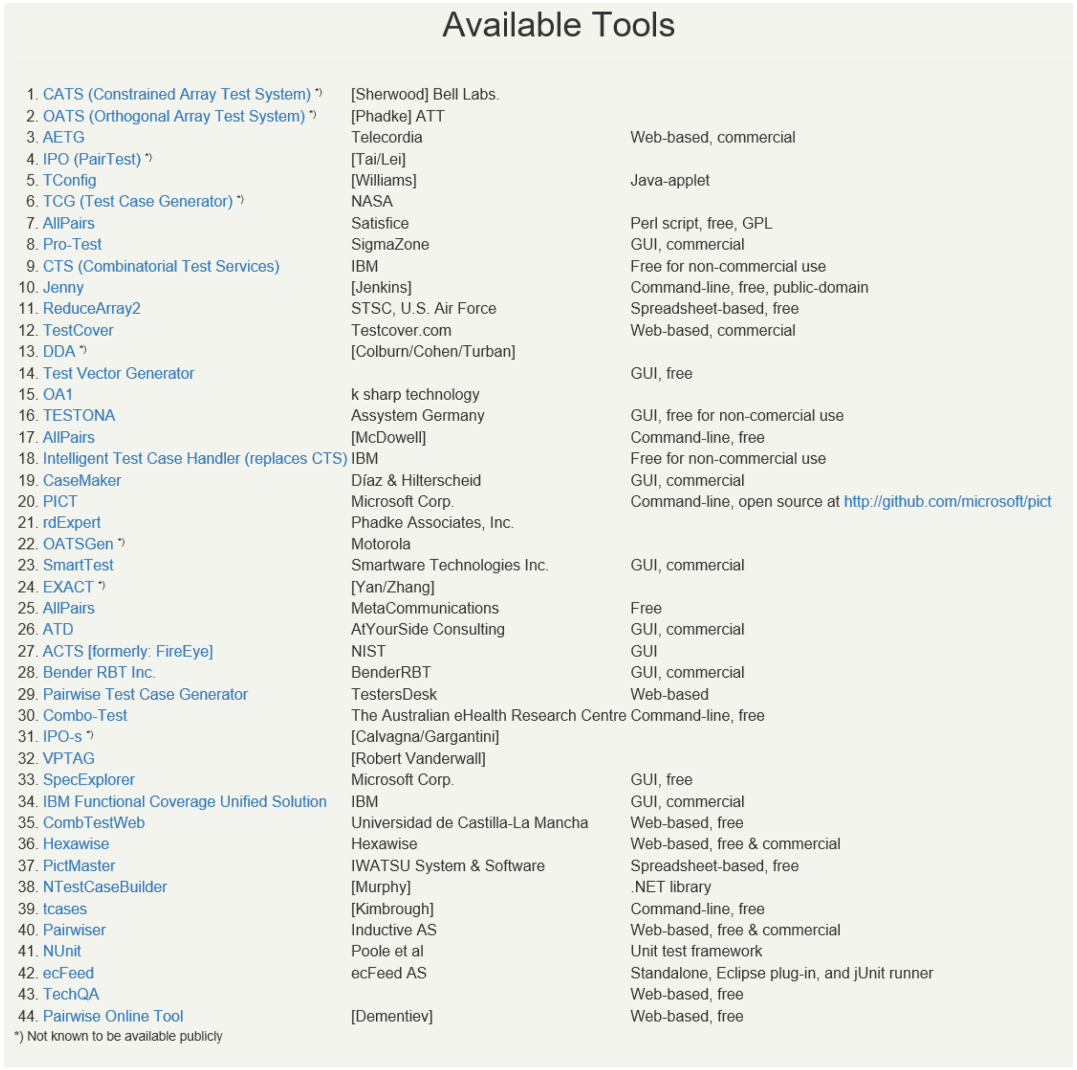
- Simple constraints
  - Example: "*Color monochrome not compatible with screen laptop and full size*" can be handled by considering the case in separate tables.

Display Mode	Language	Fonts	Color	Screen size
full-graphics	English	Minimal	Monochrome	Hand-held
text-only	French	Standard	Color-map	
limited-bandwidth	Spanish	Document-loaded	16-bit	
	Portuguese		True-color	

Display Mode	Language	Fonts	Color	Screen size
full-graphics	English	Minimal		
text-only	French	Standard	Color-map	Laptop
limited-bandwidth	Spanish	Document-loaded	16-bit	Full-size
	Portuguese		True-color	

Pairwise Testing Tools

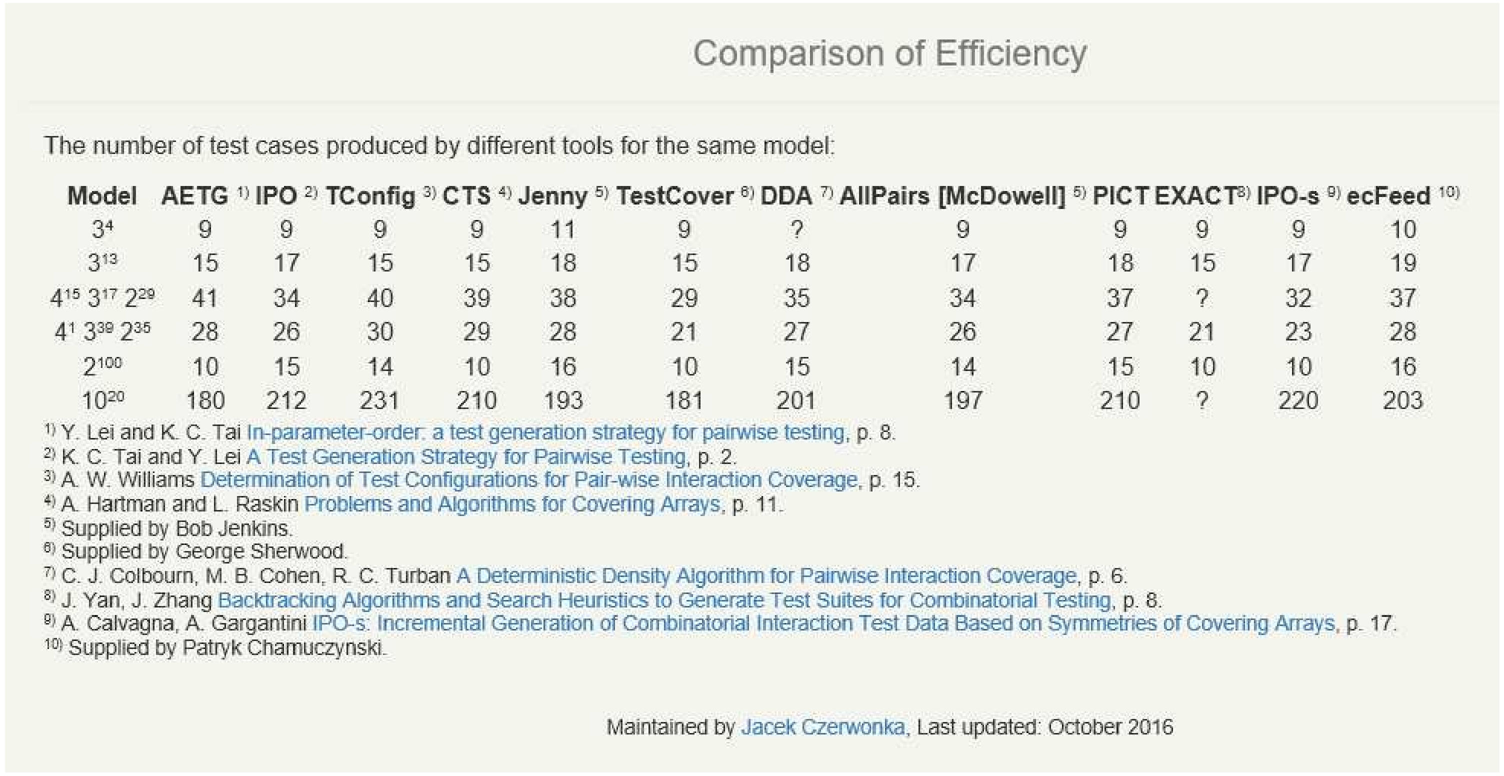
- www.pairwise.org



Available Tools

1. CATS (Constrained Array Test System) *)	[Sherwood] Bell Labs.	
2. OATS (Orthogonal Array Test System) *)	[Phadke] ATT	
3. AETG	Telecordia	Web-based, commercial
4. IPO (PairTest) *)	[Tai/Lei]	
5. TConfig	[Williams]	Java-applet
6. TCG (Test Case Generator) *)	NASA	
7. AllPairs	Satisfice	Perl script, free, GPL
8. Pro-Test	SigmaZone	GUI, commercial
9. CTS (Combinatorial Test Services)	IBM	Free for non-commercial use
10. Jenny	[Jenkins]	Command-line, free, public-domain
11. ReduceArray2	STSC, U.S. Air Force	Spreadsheet-based, free
12. TestCover	Testcover.com	Web-based, commercial
13. DDA *)	[Colburn/Cohen/Turban]	
14. Test Vector Generator		GUI, free
15. OA1	k sharp technology	
16. TESTONA	Assystem Germany	GUI, free for non-comercial use
17. AllPairs	[McDowell]	Command-line, free
18. Intelligent Test Case Handler (replaces CTS)	IBM	Free for non-commercial use
19. CaseMaker	Díaz & Hilterscheid	GUI, commercial
20. PICT	Microsoft Corp.	Command-line, open source at http://github.com/microsoft/pict
21. rdExpert	Phadke Associates, Inc.	
22. OATSGen *)	Motorola	
23. SmartTest	Smartware Technologies Inc.	GUI, commercial
24. EXACT *)	[Yan/Zhang]	
25. AllPairs	MetaCommunications	Free
26. ATD	AtYourSide Consulting	GUI, commercial
27. ACTS [formerly: FireEye]	NIST	GUI
28. Bender RBT Inc.	BenderRBT	GUI, commercial
29. Pairwise Test Case Generator	TestersDesk	Web-based
30. Combo-Test	The Australian eHealth Research Centre	Command-line, free
31. IPO-s *)	[Calvagna/Gargantini]	
32. VPTAG	[Robert Vanderwall]	
33. SpecExplorer	Microsoft Corp.	GUI, free
34. IBM Functional Coverage Unified Solution	IBM	GUI, commercial
35. CombTestWeb	Universidad de Castilla-La Mancha	Web-based, free
36. Hexawise	Hexawise	Web-based, free & commercial
37. PictMaster	IWATSU System & Software	Spreadsheet-based, free
38. NTestCaseBuilder	[Murphy]	.NET library
39. tcases	[Kimbrough]	Command-line, free
40. Pairwiser	Inductive AS	Web-based, free & commercial
41. NUnit	Poole et al	Unit test framework
42. ecFeed	ecFeed AS	Standalone, Eclipse plug-in, and jUnit runner
43. TechQA		Web-based, free
44. Pairwise Online Tool	[Dementiev]	Web-based, free

*) Not known to be available publicly



Comparison of Efficiency

The number of test cases produced by different tools for the same model:

Model	AETG [1]	IPO [2]	TConfig [3]	CTS [4]	Jenny [5]	TestCover [6]	DDA [7]	AllPairs [McDowell] [5]	PICT	EXACT[8]	IPO-s [9]	ecFeed [10]
3^4	9	9	9	9	11	9	?	9	9	9	9	10
3^{13}	15	17	15	15	18	15	18	17	18	15	17	19
$4^{15} 3^{17} 2^{29}$	41	34	40	39	38	29	35	34	37	?	32	37
$4^1 3^{39} 2^{35}$	28	26	30	29	28	21	27	26	27	21	23	28
2^{100}	10	15	14	10	16	10	15	14	15	10	10	16
10^{20}	180	212	231	210	193	181	201	197	210	?	220	203

1) Y. Lei and K. C. Tai In-parameter-order: a test generation strategy for pairwise testing, p. 8.
2) K. C. Tai and Y. Lei A Test Generation Strategy for Pairwise Testing, p. 2.
3) A. W. Williams Determination of Test Configurations for Pair-wise Interaction Coverage, p. 15.
4) A. Hartman and L. Raskin Problems and Algorithms for Covering Arrays, p. 11.
5) Supplied by Bob Jenkins.
6) Supplied by George Sherwood.
7) C. J. Colbourn, M. B. Cohen, R. C. Turban A Deterministic Density Algorithm for Pairwise Interaction Coverage, p. 6.
8) J. Yan, J. Zhang Backtracking Algorithms and Search Heuristics to Generate Test Suites for Combinatorial Testing, p. 8.
9) A. Calvagna, A. Gargantini IPO-s: Incremental Generation of Combinatorial Interaction Test Data Based on Symmetries of Covering Arrays, p. 17.
10) Supplied by Patryk Chamuczynski.

Maintained by Jacek Czerwonka, Last updated: October 2016

Pairwise Combination Testing, in Summary

- Category-partition approach gives us
  - Separation between (manual) identification of parameter characteristics and values, and (automatic) generation of test cases that combine them
  - Constraints to reduce the number of combinations

- Pairwise (or n-way) testing gives us
  - Much smaller test suites, even without constraints
  - But we can still use constraints.

- We still need help to make the manual step more systematic.

3. Catalog-based Testing

- Deriving value classes requires human judgment. Therefore, gathering experience in a systematic collection can
  - Speed up the test design process,
  - Routinize many decisions, better focusing human effort,
  - Accelerate training, and
  - Reduce human error

- **Catalogs** capture the experience of test designers by listing important cases for each possible type of variable.
  - Example: If the computation uses an integer variable, a catalog might indicate the following relevant cases
    - The element immediately preceding the lower bound
    - The lower bound of the interval
    - A non-boundary element within the interval
    - The upper bound of the interval
    - The element immediately following the upper bound

Catalog-based Testing Process

1. Identify elementary items of the specification
   - Pre-conditions
   - Post-conditions
   - Definitions
   - Variables
   - Operations

2. Derive a first set of test case specifications from pre-conditions, post-conditions and definitions

3. Complete the set of test case specifications using test catalogs

What Have We Got from Three Methods?

- **Category partition testing**
  - Division into a (manual) step of identifying categories and values, with constraints, and an (automated) step of generating combinations

- **Pairwise testing**
  - Systematic generation of smaller test suites

- Catalog-based testing
  - Improving the manual step by recording and using standard patterns for identifying significant values

- Three ideas can be combined.

Structural Testing

Structural Testing

- Judging **test suite thoroughness** based on the **structure of the program** itself
  - Also known as
    - **White-box testing**
    - **Glass-box testing**
    - **Code-based testing**
  - Distinguish from functional (requirements-based, “black-box”) testing

- Structural testing is still testing product **functionality** against its specification.
  - Only the measure of thoroughness has changed.

Rationale of Structural Testing

- One way of answering the question ***"What is missing in our test suite?"***
  - If a part of a program is not executed by any test case in the suite, faults in that part cannot be exposed.
  - But what's the '**part**'?
    - Typically, a control flow element or combination
      - Statements (CFG nodes)
      - Branches (CFG edges)
      - Fragments and combinations: Conditions, paths

- Structural testing complements functional testing.
  - Another way to recognize cases that are treated differently

- Recalling fundamental rationale
  - ***"Prefer test cases that are treated differently over cases treated the same."***

No Guarantee

- Executing all control flow elements does not guarantee finding all faults.
  - Execution of a faulty statement may not always result in a failure.
    - The state may not be corrupted when the statement is executed with some data values.
    - Corrupt state may not propagate through execution to eventually lead to failure.

- What is the value of structural coverage?
  - **Increases confidence in thoroughness of testing**

Structural Testing Complements Functional Testing

- Control flow-based testing includes cases that may not be identified from specifications alone.
  - Typical case: Implementation of a single item of the specification by multiple parts of the program

- Test suites that satisfy control flow adequacy criteria could fail in revealing faults that can be caught with functional criteria.
  - Typical case: Missing path faults

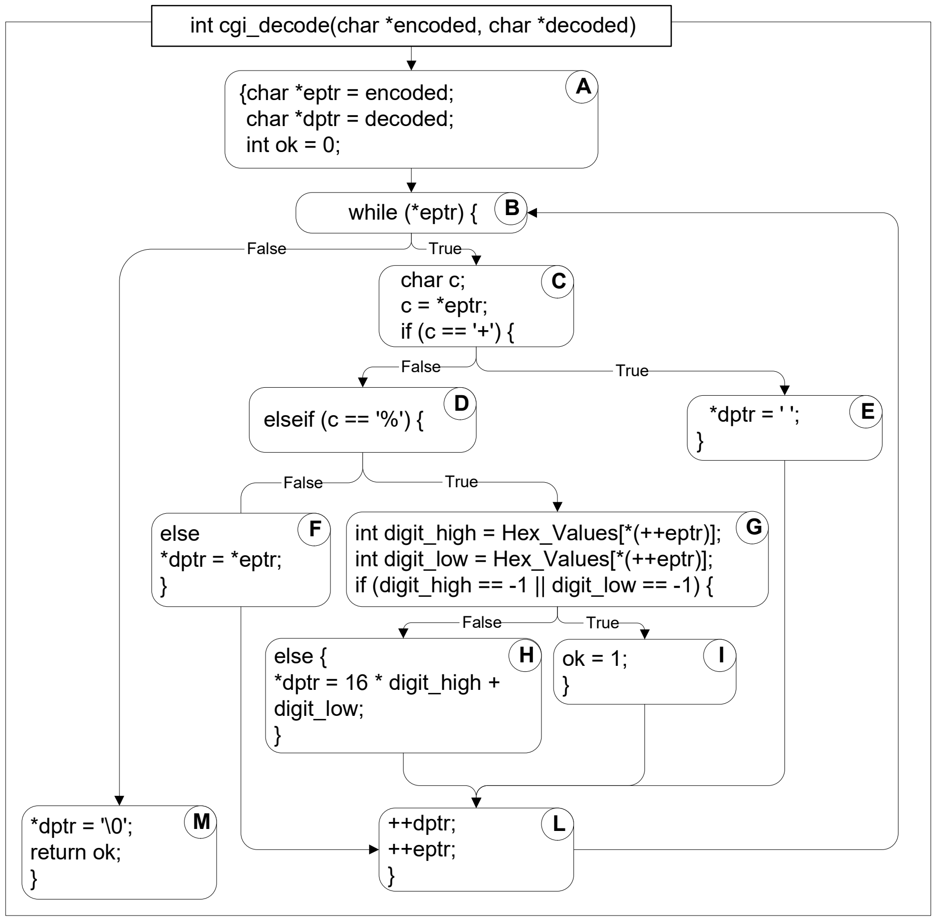
Structural Testing, in Practice

- **Create functional test suite first, then measure structural coverage to identify and see what is missing.**
  - May interpret unexecuted elements due to natural differences between specification and implementation.
  - May reveal flaws in the software or development process
    - Inadequacy of specifications that do not include cases present in the implementation
    - Coding practice that radically diverges from the specification
    - Inadequate functional test suites

- Attractive because **structural testing is automated**.
  - Coverage measurements are convenient progress indicators.
  - Sometimes used as a criterion of completion of testing
    - Use with caution: does not ensure effective test suites

An Example Program: ‘cgi_decode’ and CFG

```
#include  “hex_values.h”

int cgi_decode(char* encoded, char* decoded) {
  char *eptr = encoded;
  char *dptr = decoded;
  int ok = 0;
  while (*eptr) {
    char c;
    c = *eptr;
    if (c ==  ‘+’  ) {
      *dptr =  ‘  ’;
    } else if (c ==  ‘%’  ) {
     int digit_high = Hex_Values[*(++eptr)];
     int digit_low = Hex_Values[*(++eptr)];

     if (digit_high == -1 || digit_low == -1) {
        ok = 1;
      } else {
        *dptr = 16 * digit_high + digit_low;
      }
    } else {
      *dptr = *eptr;
    }
    ++dptr;
    ++eptr;
  }
  *dptr =  ‘\0’ ;
  return ok;
}
```



Structural Testing Techniques

1. **Statement (Coverage-based structural) Testing**

2. **Branch (Coverage-based structural) Testing**

3. **Condition (Coverage-based structural) Testing**
   - Basic
   - Compounded
   - MC/DC

4. **Path Testing**
   - Bounded interior
   - Loop boundary
   - LCSAJ
   - Cyclomatic

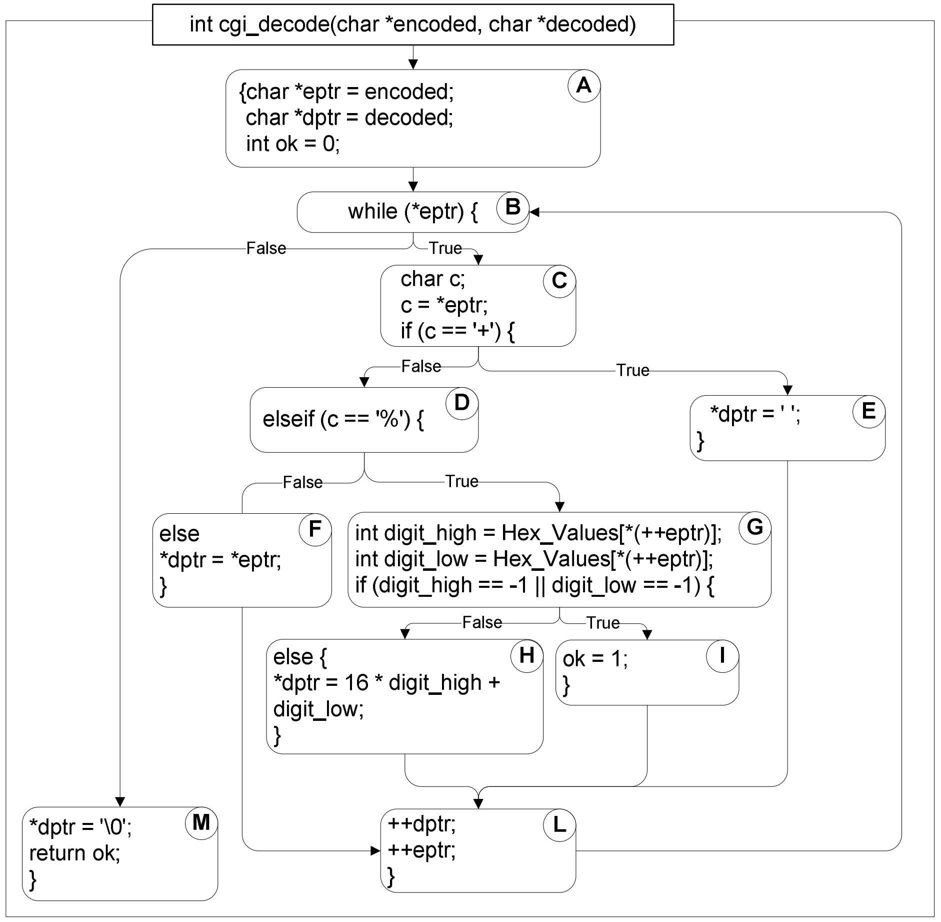
1. Statement Testing

- Adequacy criterion:
  - **Each statement (or node in the CFG) must be executed at least once.**
- Coverage:

$$\frac{\text{number of executed statements}}{\text{number of statements}}$$

- Rationale:
  - *A fault in a statement can only be revealed by executing the faulty statement.*
- Nodes in a CFG often represent basic blocks of multiple statements.
  - Some standards refer to 'basic block coverage' or 'node coverage'.
  - Difference in granularity, but not in concept

An Example: for Function “cgi_decode”



< Test cases >

T_0 =
{“”, “test”, “test+case%1Dadequacy”}
17/18 = 94% Statement coverage

T_1 =
{“adequate+test%0Dexecution%7U”}
18/18 = 100% Statement coverage

T_2 = {“%3D”, “%A”, “a+b”, “test”}
18/18 = 100% Statement coverage

T_3 = {“ ”, “+%0D+%4J”}
…

T4 = {“first+test%9Ktest%K9”}
…

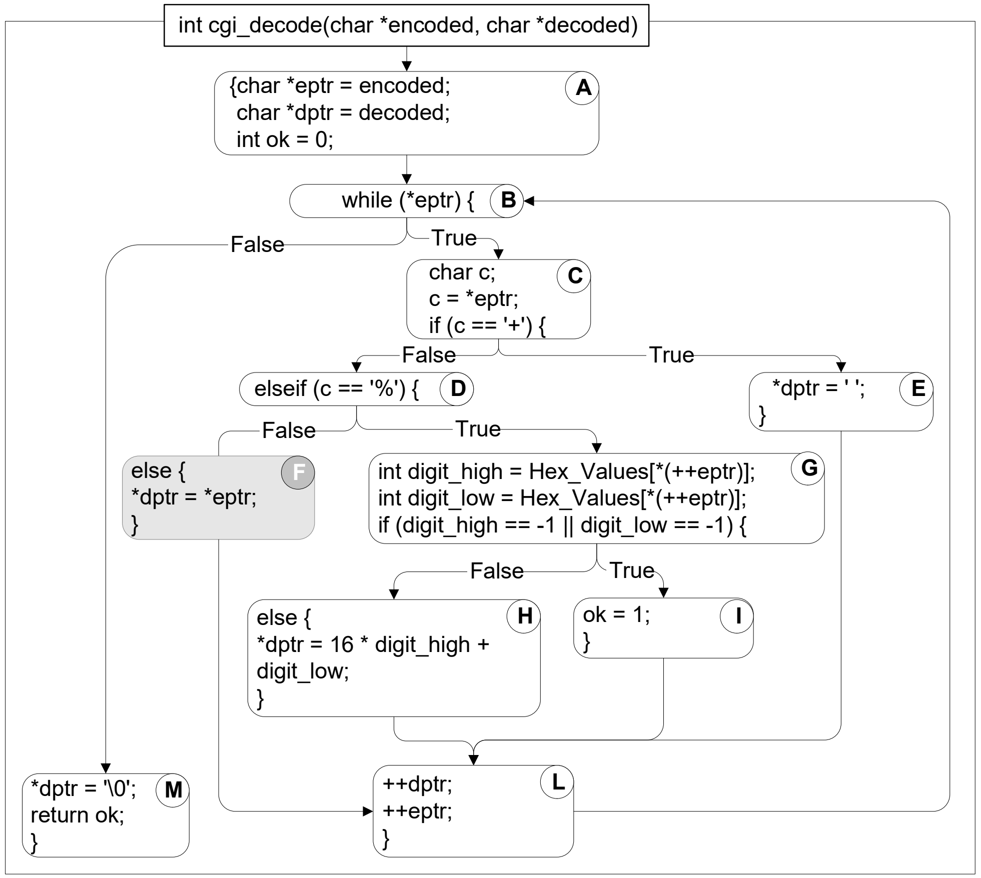
Coverage Is Not a Matter of Size

- Coverage does not depend on the number of test cases.
  - T_0 , T_1 : $T_1 >_{coverage} T_0$ $T_1 <_{cardinality} T_0$
  - T_1 , T_2 : $T_2 =_{coverage} T_1$ $T_2 >_{cardinality} T_1$

- Minimizing test suite size is not the goal.
  - Small test cases make failure diagnosis easier.
  - But a failing test case in T_2 gives more information for fault localization than a failing test case in T_1

Complete Statement Coverage

- Complete statement coverage may not imply executing all branches in a program.
- Example:
  - Suppose block F were missing
  - But statement adequacy would not require false branch from D to L
- T3 = {“ ”, “+%0D+%4J”}
  - 100% statement coverage
  - No false branch from D



int cgi_decode(char *encoded, char *decoded)
{char *eptr = encoded; char *dptr = decoded; int ok = 0;
A
while (*eptr) {
B
False
True
char c; c = *eptr; if (c == '+') {
C
False
True
elseif (c == '%') {
D
*dptr = ' '; }
E
False
True
else { *dptr = *eptr; }
F
int digit_high = Hex_Values[*(++eptr)]; int digit_low = Hex_Values[*(++eptr)]; if (digit_high == -1 || digit_low == -1) {
G
False
True
else { *dptr = 16 * digit_high + digit_low; }
H
ok = 1; }
I
*dptr = '\0'; return ok; }
M
++dptr; ++eptr; }
L

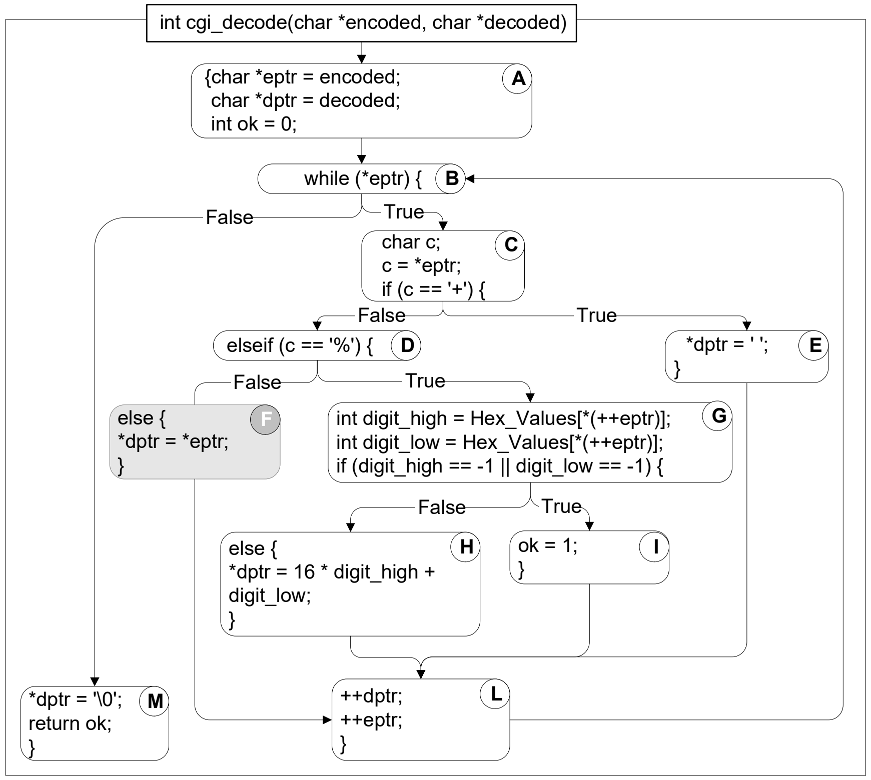
2. Branch Testing

- Adequacy criterion:
  - **Each branch (edge in the CFG) must be executed at least once.**

- Coverage:

$$\frac{\text{number of executed branches}}{\text{number of branches}}$$

- Example:
  - T_3 = {“”, “+%0D+%4J”}
    - 100% Stmt Cov.
    - 88% Branch Cov. (7/8 branches)
  - T_2 = {“%3D”, “%A”, “a+b”, “test”}
    - 100% Stmt Cov.
    - 100% Branch Cov. (8/8 branches)


int cgi_decode(char *encoded, char *decoded)
{char *eptr = encoded; char *dptr = decoded; int ok = 0;
A
while (*eptr) {
B
False
True
char c; c = *eptr; if (c == '+') {
C
False
True
elseif (c == '%') {
D
*dptr = ' '; }
E
False
True
else { *dptr = *eptr; }
F
int digit_high = Hex_Values[*(++eptr)]; int digit_low = Hex_Values[*(++eptr)]; if (digit_high == -1 || digit_low == -1) {
G
False
True
else { *dptr = 16 * digit_high + digit_low; }
H
ok = 1; }
I
*dptr = '\0'; return ok; }
M
++dptr; ++eptr; }
L

Statements vs. Branches

- Traversing all edges causes all nodes to be visited.
  - Therefore, test suites that satisfy the branch adequacy also satisfy the statement adequacy criterion for the same program.
  - Branch adequacy subsumes statement adequacy.

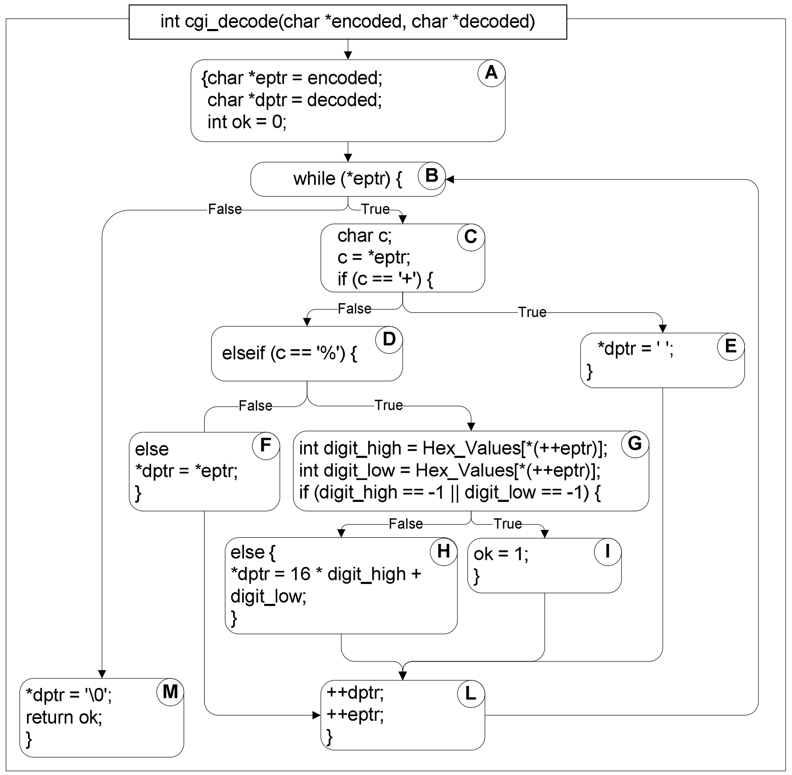
- The converse is not true (see T_3).
  - A statement-adequate test suite may not be branch-adequate.

All Branches Coverage

- "All branches coverage" can still miss conditions.
- Example:
  - Supposed that we missed the negation operator of "digit_high == -1",

    digit_high == 1 || digit_low == -1

- Branch adequacy criterion can be satisfied by varying only 'digit_low'.
  - The faulty sub-expression might never determine the result.
  - We might never really test the faulty condition, even though we tested both outcomes of the branch.



3. Condition Testing

- **Branch coverage** exposes faults in how a computation has been decomposed into cases.
  - Intuitively attractive: checking the programmer's case analysis
  - But only roughly: grouping cases with the same outcome

- **Condition coverage** considers case analysis in more detail.
  - Consider 'individual conditions' in a compound Boolean expression
    - E.g., both parts of '"igit_high == 1 || digit_low == -1"

- Adequacy criterion:
  - **Each basic condition must be executed at least once.**

- **Basic condition testing coverage** :

$$\frac{\text{number of truth values taken by all basic conditions}}{2 * \text{number of basic conditions}}$$

DEPENDABLE SOFTWARE LABORATORY

Basic Conditions vs. Branches

- Basic condition adequacy criterion can be satisfied without satisfying branch coverage.

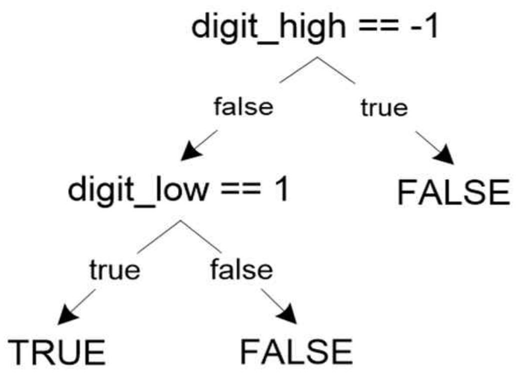
- T4 = {“first+test%9Ktest%K9”}
  - Satisfies basic condition adequacy
  - But does not satisfy branch condition adequacy

- Branch and basic condition are not comparable.
  - Neither implies the other.

Covering Branches and Conditions

- Branch and condition adequacy:
  - Cover all conditions and all decisions

- **Compound condition adequacy** :
  - Cover all possible evaluations of compound conditions.
  - Cover all branches of a decision tree.



Compounded Conditions

- Compound conditions often have exponential complexity.
- Example: (((a || b) && c) || d) && e

Test Case	a	b	c	d	e	Outcome
(1)	true	-	true	-	true	true
(2)	false	true	true	-	true	true
(3)	true	-	false	true	true	true
(4)	false	true	false	true	true	
(5)	false	false	-	true	true	
(6)	true	-	true	-	false	
(7)	false	true	true	-	false	
(8)	true	-	false	true	false	
(9)	false	true	false	true	false	
(10)	false	false	-	true	false	
(11)	true	-	false	false	-	
(12)	false	true	false	false	-	
(13)	false	false	-	false	-	

Modified Condition/Decision (MC/DC)

- Motivation
  - Effectively test **important combinations of conditions**, without exponential blowup in test suite size
  - “Important” combinations means:
    - **Each basic condition shown to independently affect the outcome of each decision.**
    - Requires
      - For each basic condition C, two test cases,
      - Values of all ‘evaluated’ conditions except C are the same.
      - Compound condition as a whole evaluates to ‘true’ for one and ‘false’ for the other.

Complexity of MC/DC

- MC/DC has a linear complexity.

- Example: (((a || b) && c) || d) && e
  - Underlined values independently affect the output of the decision.

Test Case	a	b	c	d	e	Outcome
(1)	true	-	true	-	true	true
(2)	false	true	true	-	true	true
(3)	true	-	false	true	true	true
(6)	true	-	true	-	false	false
(11)	true	-	false	false	-	false
(13)	false	false	-	false	-	false

Comments on MC/DC

- **MC/DC**
  - Basic condition coverage (C)
  - Branch coverage (DC)
  - + one additional condition (M)
    - Every condition must independently affect the decision's output.

  - Subsumed by compound conditions
  - Subsumes all other criteria discussed so far.

  - Stronger than statement and branch coverage

- Widely used as a good balance of thoroughness and test size.
  - Required by various international standard for functional safety
    - **DO-178B/C**
    - **ISO 26262**
    - **IEC 61508**

4. Path Testing

- There are many more paths than branches.
  - Decision and condition adequacy criteria consider individual decisions only.

- **Path testing** focuses combinations of decisions along paths.

- Adequacy criterion:
  - **Each path must be executed at least once.**

- Coverage:

$$\frac{\text{number of executed paths}}{\text{number of paths}}$$

Path Coverage Criteria in Practice

- The number of paths in a program with loops is ***unbounded***.
  - Usually impossible to satisfy

- To be a feasible criterion, we should partition infinite set of paths into a finite number of classes.

- Useful criteria can be obtained by limiting
  - Number of traversals of loops
  - Length of the paths to be traversed
  - Dependencies among selected paths

LCSAJ Adequacy

- **Linear Code Sequence And Jumps (LCSAJ)**
  - Sequential subpath in the CFG starting and ending in a branch
    - TER_1 = statement coverage
    - TER_2 = branch coverage
    - TER_{n+2} = coverage of n consecutive LCSAJs
  - Essentially considering full path coverage of (short) sequences of decisions

- Data flow criteria considered in a later chapter provide a more principled way of choosing some particular sub-paths as important enough to cover in testing.
  - But, neither LCSAJ nor data flow criteria are much used in current practice.

Cyclomatic Adequacy

- **Cyclomatic number**
  - Number of independent paths in the CFG
  - A path is representable as a bit vector, where each component of the vector represents an edge.
  - “Dependence” is ordinary linear dependence between (bit) vectors

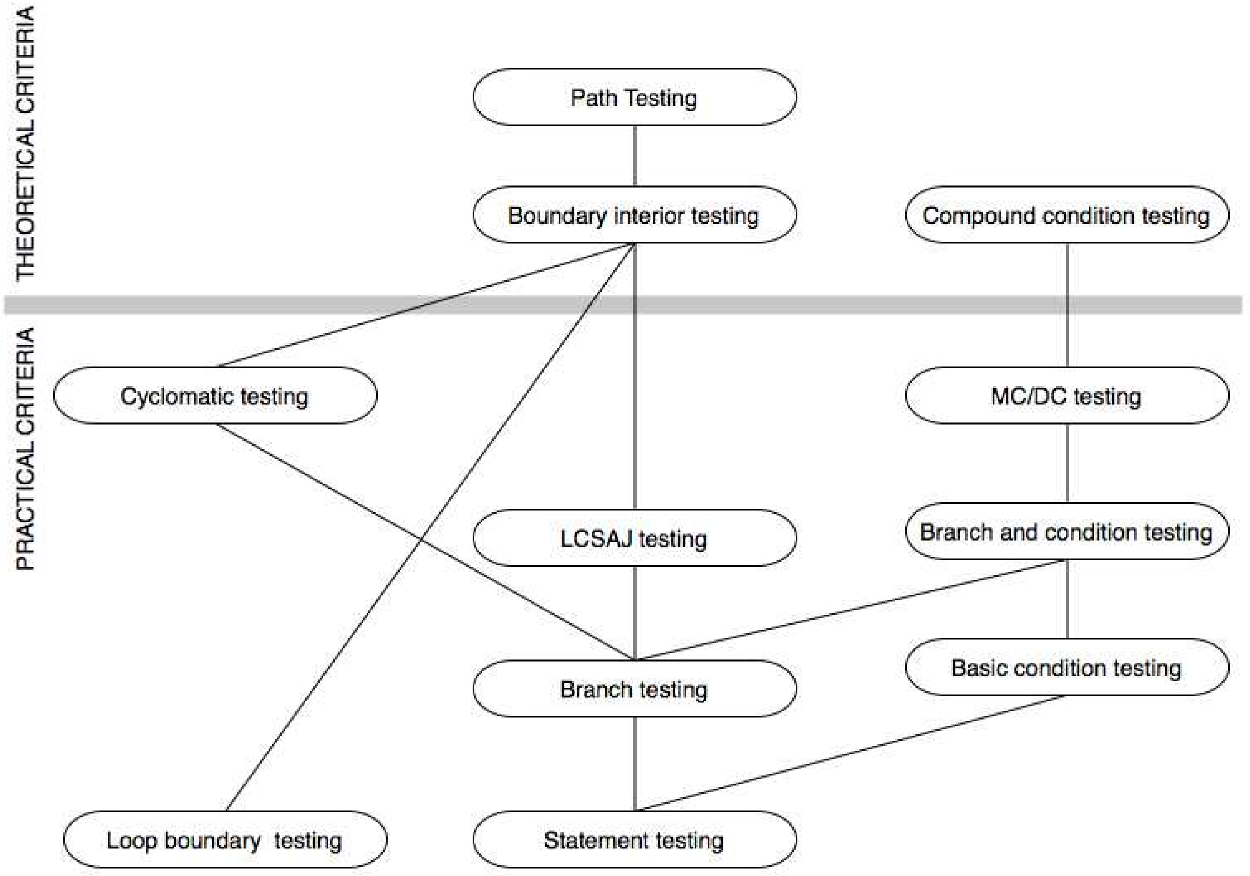
- If e = #edges, n = #nodes, c = #connected components of a graph,
  - e - n + c for an arbitrary graph
  - **e - n + 2** for a CFG ← **Cyclomatic complexity**

- Cyclomatic coverage counts the number of independent paths that have been exercised, relative to cyclomatic complexity.

Procedure Call Testing

- Measuring coverage of control flow within individual procedure is not well-straightly suited to integration or system testing.
- Choose a coverage granularity commensurate with the granularity of testing
  - If unit testing has been effective, then faults that remain to be found in integration testing will be primarily interface faults, and testing effort should focus on interfaces between units rather than their internal details.
- **Procedure entry and exit testing**
  - Procedure may have multiple entry points (e.g., Fortran) and multiple exit points.
- **Call coverage**
  - The same entry point may be called from many points.

Comparing Structural Testing Criteria



Subsumption Relation among Structural Test Adequacy Criteria

Satisfying Structural Criteria

- Large amounts of ‘fossil’ code may indicate serious maintainability problems.
- But some unreachable code is common even in well-designed and well-maintained systems.
- Solutions:
  1. **Make allowances by setting a coverage goal less than 100%**
  2. **Require justification of elements left uncovered**
     - As RTCA-DO-178B/C and EUROCAE ED-12B for modified MC/DC

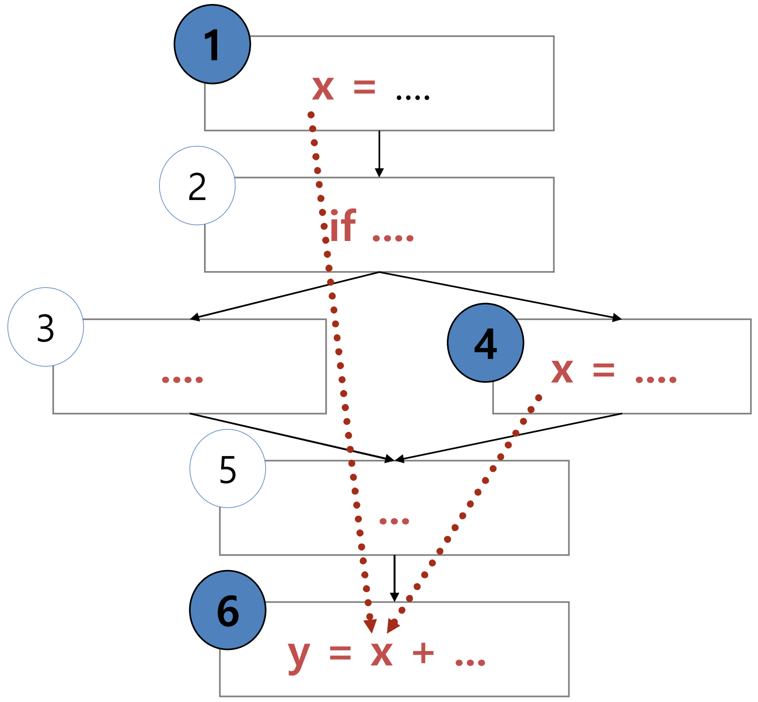
Data Flow Testing

Motivation

- Middle ground in structural testing
  - Node and edge coverage don't test interactions.
  - Path-based criteria require impractical number of test cases.
    - Only a few paths uncover additional faults, anyway.
  - **Need to distinguish "important" paths**

- Intuition: **Statements interact through data flow.**
  - Value computed in one statement, is used in another.
  - **Bad value computation can be revealed only when it is used.**

Def-Use Pairs



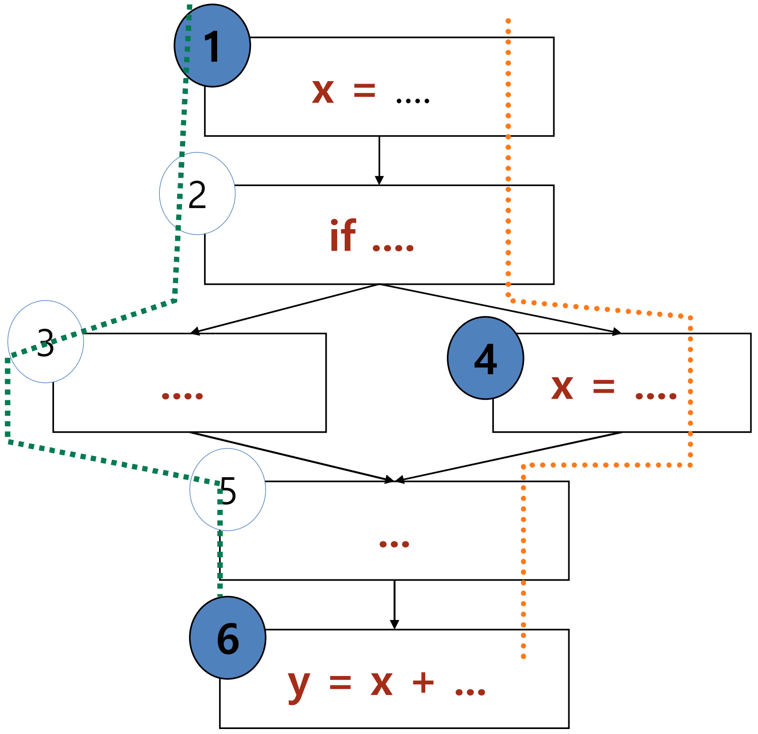
- Value of x at 6 could be computed at 1 or at 4.
- Bad computation at 1 or 4 could be revealed only if they are used at 6.
- (1, 6) and (4, 6) are **def-use (DU) pairs**.
  - *defs* at 1, 4
  - *use* at 6

Terminology

- **DU pair**
  - A pair of definition and use for a variable, such that at least one DU path exists from the definition to the use.
  - “x = ...” is a definition of x
  - “= ... x ...” is a use of x

- **DU path**
  - A **definition-clear path** on the CFG starting from a definition to a use of a same variable
  - Definition clear: Value is not replaced on path.
  - Note: Loops could create infinite DU paths between a def and a use.

Definition-Clear Path



- 1,2,3,5,6 is a definition-clear path from 1 to 6.
  - x is not re-assigned between 1 and 6.
- 1,2,4,5,6 is not a definition-clear path from 1 to 6.
  - the value of x is “killed” (reassigned) at node 4.
- (1, 6) is a DU pair because 1,2,3,5,6 is a definition-clear path.

Adequacy Criteria

- **All DU pairs**
  - Each DU pair is exercised by at least one test case.
- **All DU paths**
  - Each simple (non looping) DU path is exercised by at least one test case.
- **All definitions**
  - For each definition, there is at least one test case which exercises a DU pair containing it.
  - Because every computed value is used somewhere.

- Corresponding coverage fractions can be defined similarly.

Difficult Cases

- x[i] = ... ; ... ; y = x[j]
  - DU pair (only) if i == j
- p = &x ; ... ; *p = 99 ; ... ; q = x
  - *p is an alias of x
- m.putFoo(...); ... ; y=n.getFoo(...);
  - Are m and n the same object?
  - Do m and n share a "foo" field?
- Problem of aliases:
  - Which references are (always or sometimes) the same?

Data Flow Coverage with Complex Structures

- Arrays and pointers are critical for data flow analysis.
  - **Under-estimation** of aliases may fail to include some DU pairs.
  - **Over-estimation** may introduce unfeasible test obligations.

- For testing, it may be preferable to accept under-estimation of alias set rather than over-estimation or expensive analysis.
  - Alias analysis may rely on external guidance or other global analysis to calculate good estimates.
  - Undisciplined use of dynamic storage, pointer arithmetic, etc. may make the whole analysis infeasible.
  - But, in other applications (e.g., compilers), a conservative over-estimation of aliases is usually required.

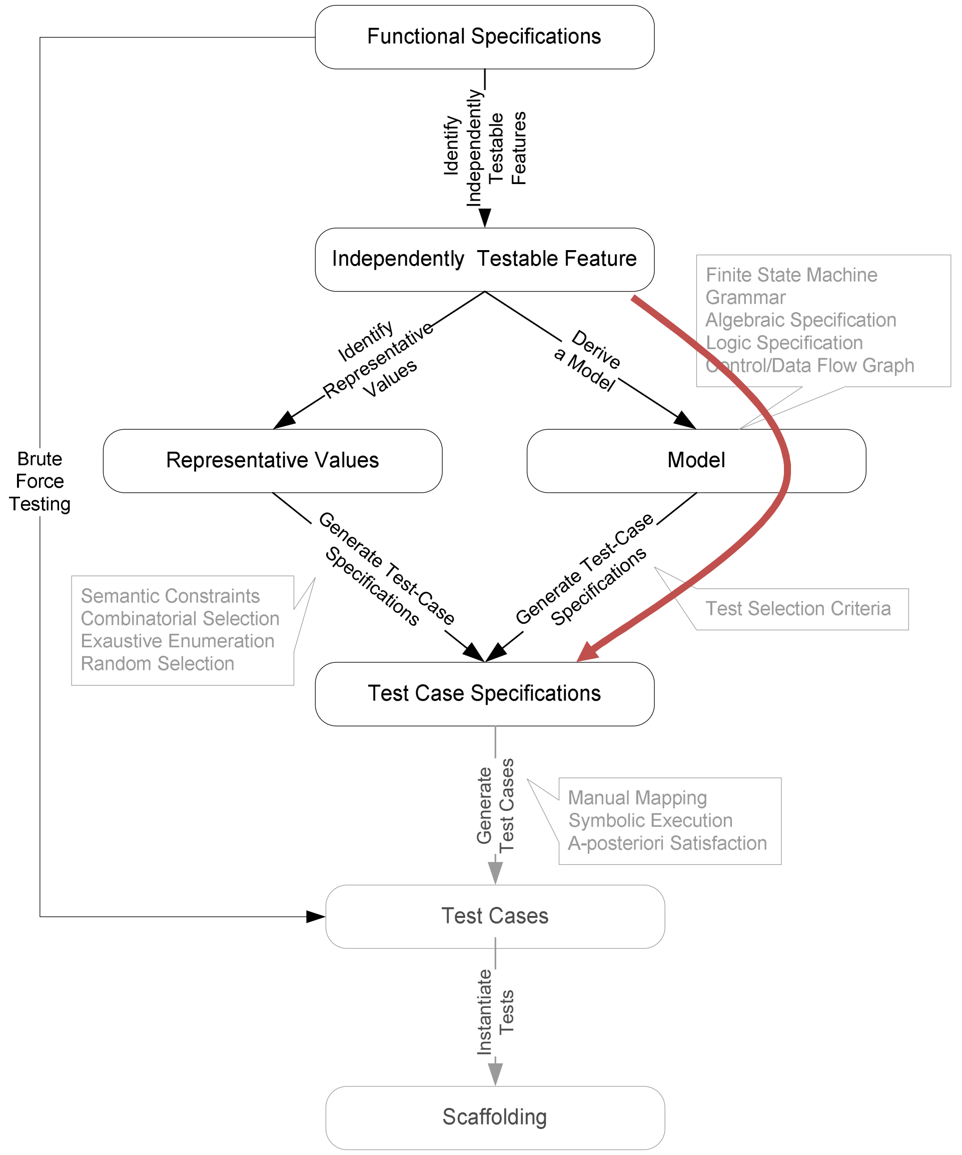
Data Flow Coverage in Practice

- The path-oriented nature of data flow analysis makes the infeasibility problem especially relevant.
  - Combinations of elements matter.
  - Impossible to distinguish feasible from infeasible paths.
  - More paths = More work to check manually

- In practice, reasonable coverage is (often, not always) achievable.
  - Number of paths is exponential in worst case, but often linear.
  - All DU paths is more often impractical.

Dependable Software Laboratory

Model-Based Testing

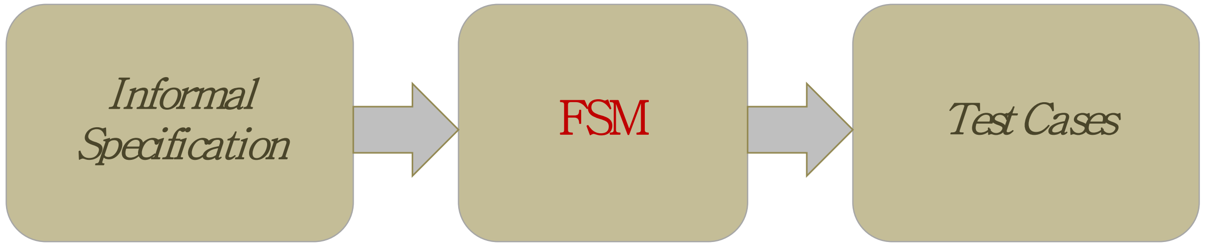

Functional Specifications
Identify Independently Testable Features
Independently Testable Feature
Identify Representative Values
Derive a Model
Finite State Machine
Grammar
Algebraic Specification
Logic Specification
Control/Data Flow Graph
Representative Values
Model
Brute Force Testing
Generate Test-Case Specifications
Generate Test-Case Specifications
Semantic Constraints
Combinatorial Selection
Exaustive Enumeration
Random Selection
Test Selection Criteria
Test Case Specifications
Generate Test Cases
Manual Mapping
Symbolic Execution
A-posteriori Satisfaction
Test Cases
Instantiate Tests
Scaffolding

Overview

- **Models** used in specification or design have structure.
  - Useful information for selecting representative classes of behavior
  - Behaviors that are treated differently with respect to the model should be tried by a thorough test suite.
  - In combinatorial testing, it is difficult to capture that structure clearly and correctly in constraints.

- **We can devise test cases to check actual behavior against behavior specified by the model.**
  - "Coverage" similar to structural testing, but applied to specification or design models

Deriving Test Cases from Finite State Machines



Informal Specification: Feature “Maintenance” of the Chipmunk Web Site and an FSM model

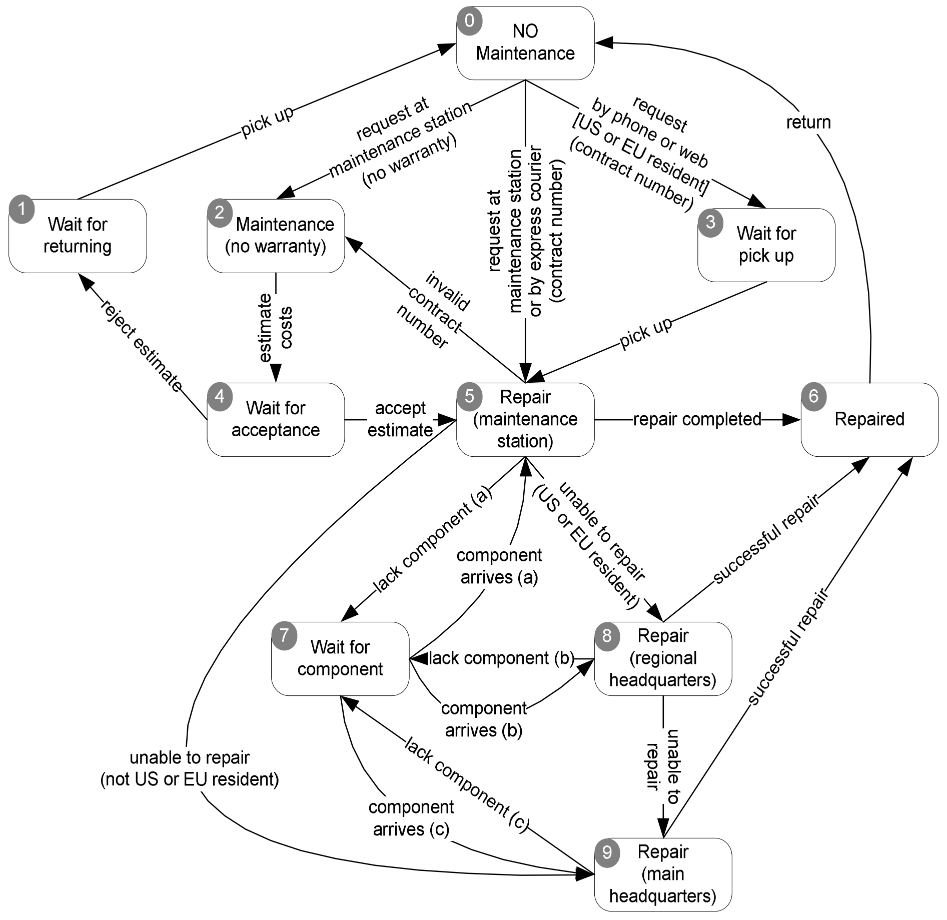
Maintenance: The Maintenance function records the history of items undergoing maintenance.

If the product is covered by warranty or maintenance contract, maintenance can be requested either by calling the maintenance toll free number, or through the web site, or by bringing the item to a designated maintenance station. If the maintenance is requested by phone or web site and the customer is a US or EU resident, the item is picked up at the customer site, otherwise, the customer shall ship the item with an express courier.

If the maintenance contract number provided by the customer is not valid, the item follows the procedure for items not covered by warranty.
If the product is not covered by warranty or maintenance contract, maintenance can be requested only by bringing the item to a maintenance station. The maintenance station informs the customer of the estimated costs for repair. Maintenance starts only when the customer accepts the estimate.

If the customer does not accept the estimate, the product is returned to the customer. Small problems can be repaired directly at the maintenance station. If the maintenance station cannot solve the problem, the product is sent to the maintenance regional headquarters (if in US or EU) or to the maintenance main headquarters (otherwise).

If the maintenance regional headquarters cannot solve the problem, the product is sent to the maintenance main headquarters.
Maintenance is suspended if some components are not available.
Once repaired, the product is returned to the customer.



Test Cases Generated from the FSM

- FSM can be used both to
  1. Guide test selection (checking each state transition)
  2. Constructing an oracle that judge whether each observed behavior is correct

TC1	0	2	4	1	0					
TC2	0	5	2	4	5	6	0			
TC3	0	3	5	9	6	0				
TC4	0	3	5	7	5	8	7	8	9	6	0

- Questions:
  - *"Is this a thorough test suite?"*
  - *"How can we judge?"*

  → Coverage criteria require

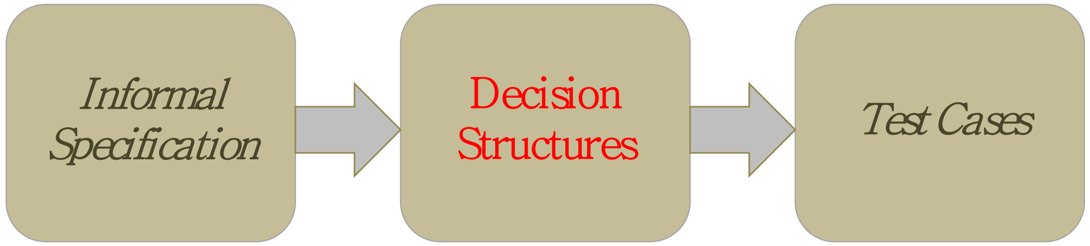
Transition Coverage Criteria

- **All state coverage**
  - Every state in the model should be visited by at least one test case.

- **All transition coverage**
  - Every transition between states should be traversed by at least one test case.
  - Most commonly used criterion
  - A transition can be thought of as a (precondition, postcondition) pair

Deriving Test Cases from Decision Structures

- Some specifications are structured as **decision tables**, decision trees, or flow charts.



Informal Specification: Feature "Price" of the Chipmunk Web Site and a Decision Table Model

Pricing: The pricing function determines the adjusted price of a configuration for a particular customer.

The scheduled price of a configuration is the sum of the scheduled price of the model and the scheduled price of each component in the configuration. The adjusted price is either the scheduled price, if no discounts are applicable, or the scheduled price less any applicable discounts.
There are three price schedules and three corresponding discount schedules, Business, Educational, and Individual.
….
Educational prices: The adjusted price for a purchase charged to an educational account in good standing is the scheduled price from the educational price schedule. No further discounts apply.
…
Special-price non-discountable offers: Sometimes a complete configuration is offered at a special, non-discountable price. When a special, non-discountable price is available for a configuration, the adjusted price is the non-discountable price or the regular price after any applicable discounts, whichever is less.

	Education		Individual					
EduAc	T	T	F	F	F	F	F	F
BusAc	-	-	F	F	F	F	F	F
CP > CT1	-	-	F	F	T	T	-	-
YP > YT1	-	-	-	-	-	-	-	-
CP > CT2	-	-	-	-	F	F	T	T
YP > YT2	-	-	-	-	-	-	-	-
SP < Sc	F	T	F	T	-	-	-	-
SP < T1	-	-	-	-	F	T	-	-
SP < T2	-	-	-	-	-	-	F	T
Out	Edu	SP	ND	SP	T1	SP	T2	SP

Constraints

at-most-one (EduAc, BusAc)
YP > YT2 → YP > YT1
CP > CT2 → CP > CT1
SP > T2 → SP > T1

at-most-one (YP < YT1, YP > YT2)
at-most-one (CP < CT1, CP > CT2)
at-most-one (SP < T1, SP > T2

Test Cases Generated from the Decision Table

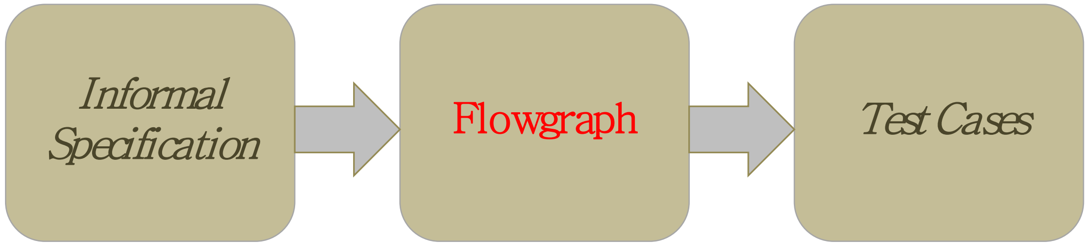
- **Basic condition coverage**
  - A test case specification for each column in the table

- **Compound condition adequacy criterion**
  - A test case specification for each combination of truth values of basic conditions

- **Modified condition/decision adequacy criterion (MC/DC)**
  - Each column in the table represents a test case specification.
  - We add columns that differ in one input row and in outcome, then merge compatible columns.

Deriving Test Cases from Control and Data Flow Graph

- If the specification or model has both decisions and sequential logic, we can cover it like program source code with **flowgraph**.



Informal Specification: Feature “Process Shipping Order” of the Chipmunk Web Site and an CFG Model

Process shipping order: The Process shipping order function checks the validity of orders and prepares the receipt.

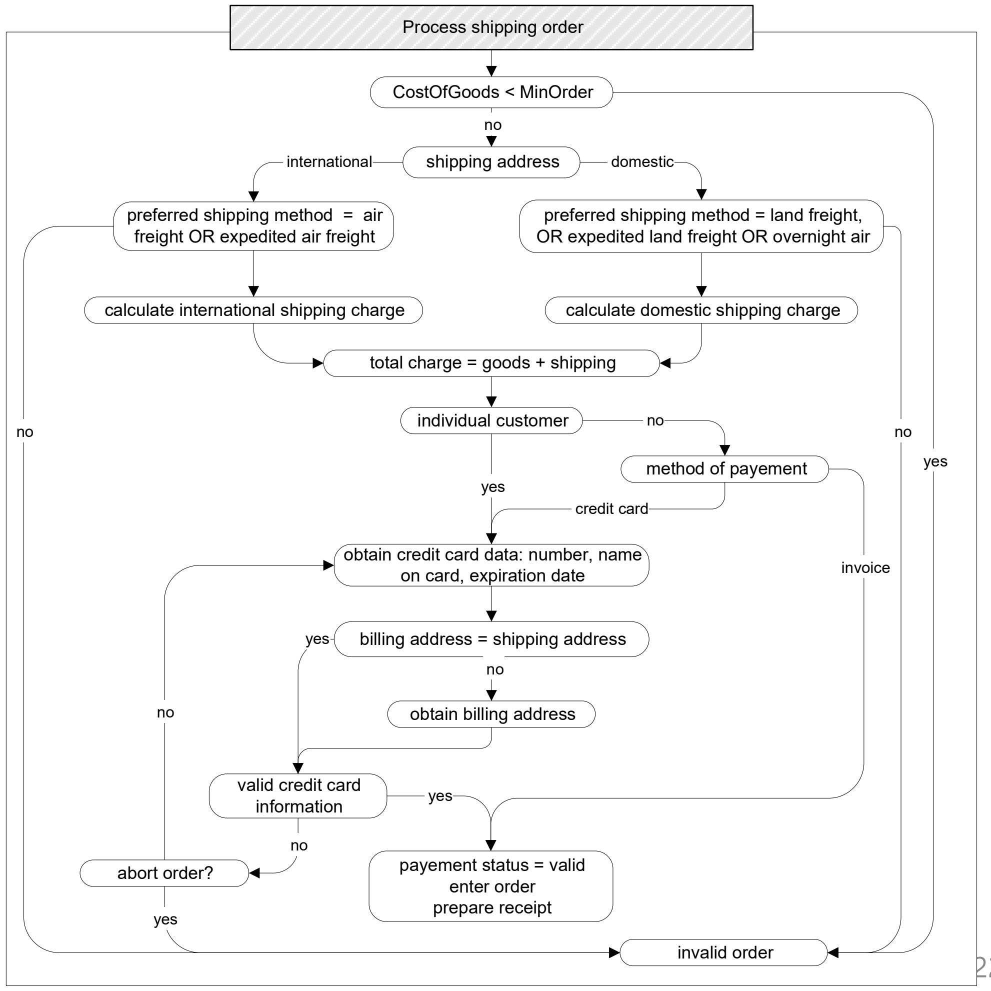
A valid order contains the following data:

cost of goods: If the cost of goods is less than the minimum processable order (MinOrder) then the order is invalid.
shipping address: The address includes name, address, city, postal code, and country.
preferred shipping method: If the address is domestic, the shipping method must be either land freight, expedited land freight, or overnight air; If the address is international, the shipping method must be either air freight, or expedited air freight.

type of customer which can be individual, business, educational
preferred method of payment. Individual customers can use only credit cards, business and educational customers can choose between credit card and invoice

card information: if the method of payment is credit card, fields credit card number, name on card, expiration date, and billing address, if different than shipping address, must be provided. If credit card information is not valid the user can either provide new data or abort the order.

The outputs of Process shipping order are
validity: Validity is a boolean output which indicates whether the order can be processed.
total charge: The total charge is the sum of the value of goods and the computed shipping costs (only if validity = true).
payment status: if all data are processed correctly and the credit card information is valid or the payment is invoice, payment status is set to valid, the order is entered, and a receipt is prepared; otherwise, validity = false.


Process shipping order
CostOfGoods < MinOrder
no
shipping address
international
domestic
preferred shipping method = air freight OR expedited air freight
preferred shipping method = land freight, OR expedited land freight OR overnight air
calculate international shipping charge
calculate domestic shipping charge
total charge = goods + shipping
individual customer
no
method of payement
yes
credit card
obtain credit card data: number, name on card, expiration date
invoice
yes
billing address = shipping address
no
obtain billing address
no
valid credit card information
yes
no
abort order?
payement status = valid
enter order
prepare receipt
yes
invalid order
no
no
yes

Test Cases Generated from the CFG

- **Node adequacy criteria**

Case	Too Small	Ship Where	Ship Method	Cust Type	Pay Method	Same Address	CC valid
TC-1	No	Int	Air	Bus	CC	No	Yes
TC-2	No	Dom	Air	Ind	CC	-	No (abort)

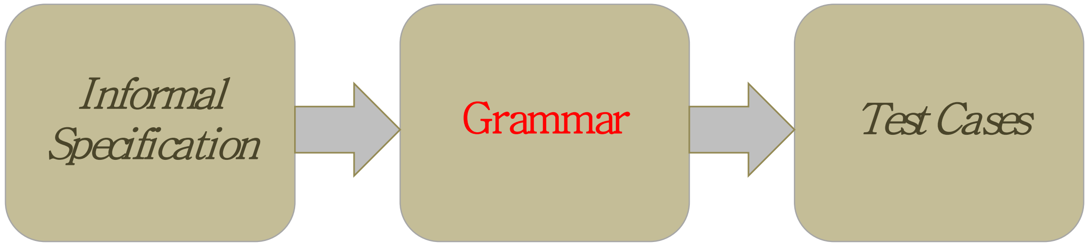
- **Branch adequacy criteria**

Case	Too Small	Ship Where	Ship Method	Cust Type	Pay Method	Same Address	CC valid
TC-1	No	Int	Air	Bus	CC	No	Yes
TC-2	No	Dom	Land	-	-	-	-
TC-3	Yes	-	-	-	-	-	-
TC-4	No	Dom	Air	-	-	-	-
TC-5	No	Int	Land	-	-	-	-
TC-6	No	-	-	Edu	Inv	-	-
TC-7	No	-	-	-	CC	Yes	-
TC-8	No	-	-	-	CC	-	No (abort)
TC-9	No	-	-	-	CC	-	No (no abort)

Deriving Test Cases from Grammars

- **Grammars** are good at representing inputs of varying and unbounded size with recursive structure and boundary conditions.

- Examples:
  - Complex textual inputs
  - Trees (search trees, parse trees, ...)
    - Example: XML and HTMl are trees in textual form
  - Program structures
    - Which are also tree structures in textual format



DEPENDABLE SOFTWARE LABORATORY

Grammar-Based Testing

- Test cases are ‘strings’ generated from the grammar.

- Coverage criteria:
  - **Production coverage:**
    - Each production must be used to generate at least one (section of) test case.
  - **Boundary condition:**
    - Annotate each recursive production with minimum and maximum number of application, then generate:
      - Minimum
      - Minimum + 1
      - Maximum - 1
      - Maximum

Informal Specification: Feature “Check Configuration” of the Chipmunk Web Site and a Grammar Model

Check configuration: The Check-configuration function checks the validity of a computer configuration.

Model: A model identifies a specific product and determines a set of constraints on available components. Models are characterized by logical slots for components, which may or may not be implemented by physical slots on a bus. Slots may be required or optional. Required slots must be assigned with a suitable component to obtain a legal configuration, while optional slots may be left empty or filled depending on the customers' needs

Example: The required ``slots'' of the Chipmunk C20 laptop computer include a screen, a processor, a hard disk, memory, and an operating system. (Of these, only the hard disk and memory are implemented using actual hardware slots on a bus.) The optional slots include external storage devices such as a CD/DVD writer.

Set of Components: A set of [slot,component] pairs, which must correspond to the required and optional slots associated with the model. A component is a choice that can be varied within a model, and which is not designed to be replaced by the end user. Available components and a default for each slot is determined by the model. The special value empty is allowed (and may be the default selection) for optional slots. In addition to being compatible or incompatible with a particular model and slot, individual components may be compatible or incompatible with each other.

Example: The default configuration of the Chipmunk C20 includes 20 gigabytes of hard disk; 30 and 40 gigabyte disks are also available. (Since the hard disk is a required slot, empty is not an allowed choice.) The default operating system is RodentOS 3.2, personal edition, but RodentOS 3.2 mobile server edition may also be selected. The mobile server edition requires at least 30 gigabytes of hard disk.

Model	<Model>	::= <modelNumber> <compSequence> <optCompSequence>
compSeq1 [0, 16]	<compSequence>	::= <Component> <compSequence>
compSeq2	<compSequence>	::= empty
optCompSeq1 [0, 16]	<optCompSequence>	::= <OptionalComponent> <optCompSequence>
optCompSeq2	<optCompSequence>	::= empty
Comp	<Component>	::= <ComponentType> <ComponentValue>
OptComp	<OptionalComponent>	::= <ComponentType>
modNum	<modelNumber>	::= string
CompTyp	<ComponentType>	::= string
CompVal	<ComponentValue>	::= string

Test Cases Generated from the Grammar

- "Mod000"
  - Covers Model, compSeq1[0], compSeq2, optCompSeq1[0], optCompSeq2, modNum
- "Mod000 (Comp000, Val000) (OptComp000)"
  - Covers Model, compSeq1[1], compSeq2, optCompSeq2[0], optCompSeq2, Comp, OptComp, modNum, CompTyp, CompVal
- Etc.
- Comments:
  - By first applying productions with nonterminals on the right side, we obtain few, large test cases.
  - By first applying productions with terminals on the right side, we obtain many, small test cases.

Grammar Testing vs. Combinatorial Testing

- Combinatorial specification-based testing is good for "mostly independent" parameters.
  - We can incorporate a few constraints, but complex constraints are hard to represent and use.
  - We must often "factor and flatten."
    - E.g., separate "set of slots" into characteristics "number of slots" and predicates about what is in the slots (all together)

- Grammar describes sequences and nested structure naturally.
  - But, some relations among different parts may be difficult to describe and exercise systematically,
    - E.g., compatibility of components with slots.

Fault-Based Testing

Estimating Test Suite Quality

- Supposed that I have a program with bugs.

- Add 100 new bugs
  - Assume they are exactly like real bugs in every way
  - I make 100 copies of my program, each with one of my 100 new bugs.

- Run my test suite on the programs with seeded bugs
  - And the tests revealed 20 of the bugs.
  - The other 80 program copies do not fail.

- What/How can I infer about **my test suite's quality**?

Basic Assumptions

- We want to judge effectiveness of a test suite in finding real faults,
  - by measuring how well it finds **seeded fake faults**.

- Valid to the extent that the seeded bugs are representative of real bugs
  - Not necessarily identical
  - But the differences should not affect the selection.

Mutation Testing

- A **mutant** is a copy of a program with a mutation.

- A **mutation** is a syntactic change (a seeded bug).
  - Example: change (i < 0) to (i <= 0)

- Run test suite on all the mutant programs.
- **A mutant is killed**, if it fails on at least one test case. → The bug is found.

- If many mutants are killed, infer that the test suite is also effective at finding real bugs.

Assumptions on Mutation Testing

- ***Competent programmer hypothesis***
  - Programs are nearly correct.
    - Real faults are small variations from the correct program.
    - Therefore, mutants are reasonable models of real buggy programs.

- ***Coupling effect hypothesis***
  - Tests that find simple faults also find more complex faults.
  - Even if mutants are not perfect representatives of real faults, a test suite that kills mutants is good at finding real faults too.

Mutant Operators

- Syntactic changes from legal program to illegal program
  - Specific to each programming language

- Examples:
  - **crp**: constant for constant replacement
    - E.g., from (x < 5) to (x < 12)
    - Select constants found somewhere in program text
  - **ror**: relational operator replacement
    - E.g., from (x <= 5) to (x < 5)
  - **vie**: variable initialization elimination
    - E.g., change int x =5; to int x;

Fault-based Adequacy Criteria

- Mutation analysis consists of the following steps:
  - Select mutation operators
  - Generate mutants
  - Distinguish mutants

- **Live mutants**
  - Mutants not killed by a test suite

- *Given a set of mutants SM and a test suite T, the fraction of nonequivalence mutants killed by T measures the adequacy of T with respect to SM.*

Variations on Mutation Analysis

- Problem:
  - There are lots of mutants.
  - Running each test case to completion on every mutant is expensive.
  - Number of mutants grows with the square of program size.

- Solutions:
  - Weak mutation:
    - Execute meta-mutant (with many seeded faults) together with original program
  - Statistical mutation
    - Just create a random sample of mutants

Summary

- **Fault-based testing** is a widely used in semiconductor manufacturing.
  - With good fault models of typical manufacturing faults, e.g., “stuck-at-one” for a transistor
  - But fault-based testing for design errors is more challenging (as in software).

- **Mutation testing** is not widely used in industry.
  - But plays a role in software testing research, to compare effectiveness of testing techniques.

Test Execution

Automating Test Execution

- Designing test cases and test suites is creative.
  - Demanding intellectual activity
  - Requiring human judgment

- Executing test cases should be automatic.
  - Design once, execute many times

- **Test automation** separates the creative human process from the mechanical process of test execution.

From Test Case Specifications to Test Cases

- Test design often yields test case specifications, rather than concrete data.
  - E.g., “a large positive number”, not 420,023
  - E.g., “a sorted sequence, length > 2”, not “Alpha, Beta, Chi, Omega”
  - Other details for execution may be omitted.

- **Test Generation** creates concrete/executable test cases from test case specifications.

- A Tool chain for test case generation and execution
  - A combinatorial test case generation to create test data
    - Optional: Constraint-based data generator to “concretize” individual values, e.g., from “positive integer” to 42
  - ‘DDSteps’ to convert from spreadsheet data to ‘JUnit’ test cases
  - ‘**JUnit**’ to execute concrete test cases

Scaffolding

- Code produced to support development activities
  - Not part of the "product" as seen by the end user
  - May be temporary (like scaffolding in construction of buildings)

- **Scaffolding** includes
  - **Test harnesses**
  - **Drivers**
  - **Stubs**

Scaffolding

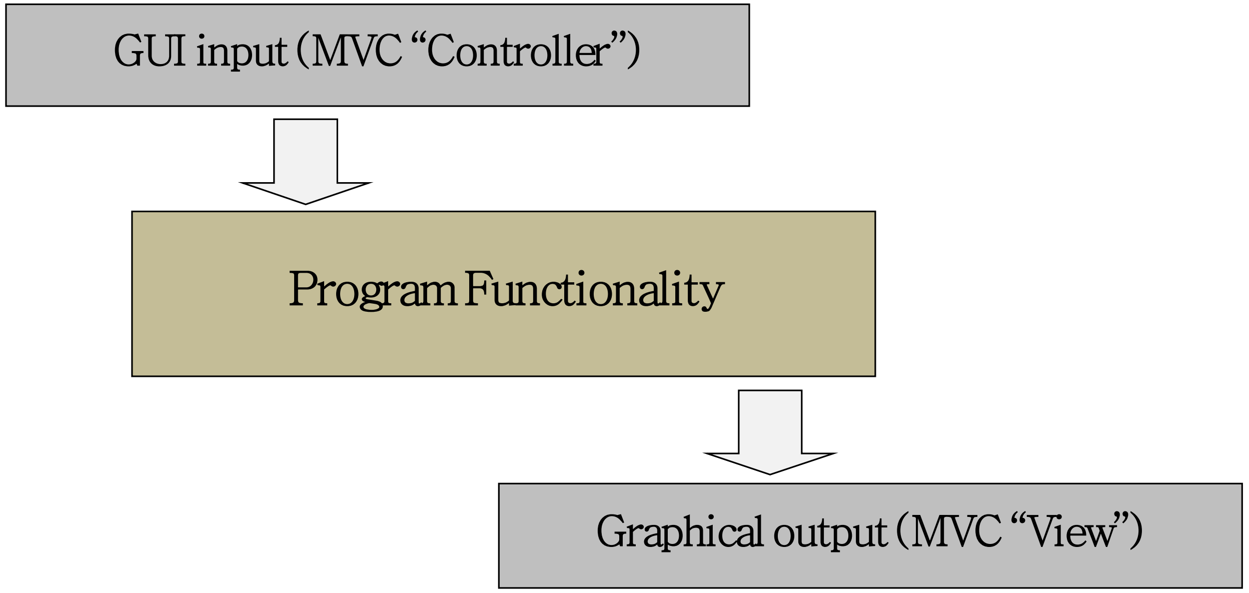
- **Test driver**
  - A “main” program for running a test
    - May be produced before a “real” main program
    - Provide more control than the “real” main program
  - To drive program under test through test cases

- **Test stub**
  - Substitute for called functions/methods/objects

- **Test harness**
  - Substitutes for other parts of the deployed environment
  - E.g., Software simulation of a hardware device

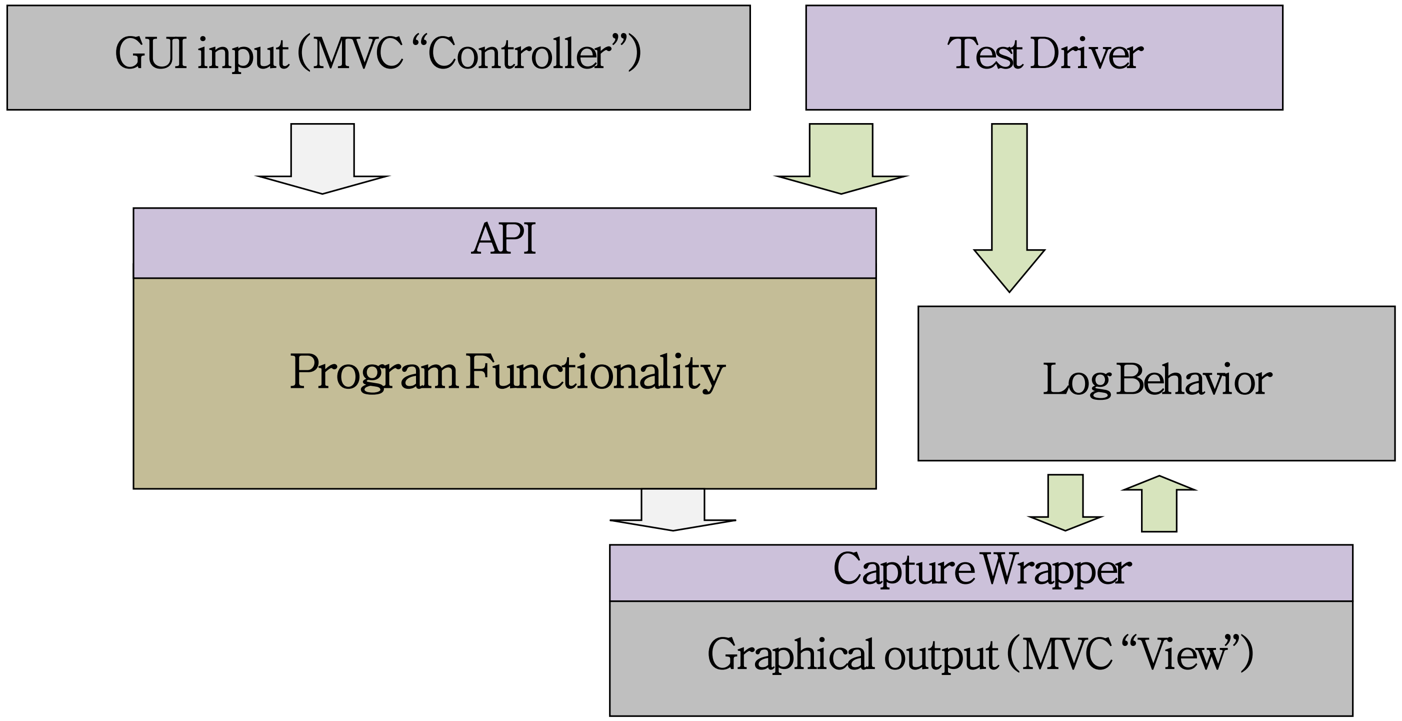
Controllability & Observability

- Example: We want to automate tests.
  - Interactive input provides limited control.
  - Graphical output provides limited observability.



Controllability & Observability

- Solution:
  - A design for automated test provides interfaces for control (API) and observation (wrapper on output)



Generic vs. Specific Scaffolding

- How general should scaffolding be?
  - We could build a driver and stubs for each test case.
  - Or at least factor out some common code of the driver and test management (e.g. JUnit)
  - Or further factor out some common support code, to drive a large number of test cases from data (as in DDSteps)
  - Or further generate the data automatically from a more abstract model (e.g. network traffic model)

- It's a question of **costs and re-use**, just as for other kinds of software.

Test Oracles

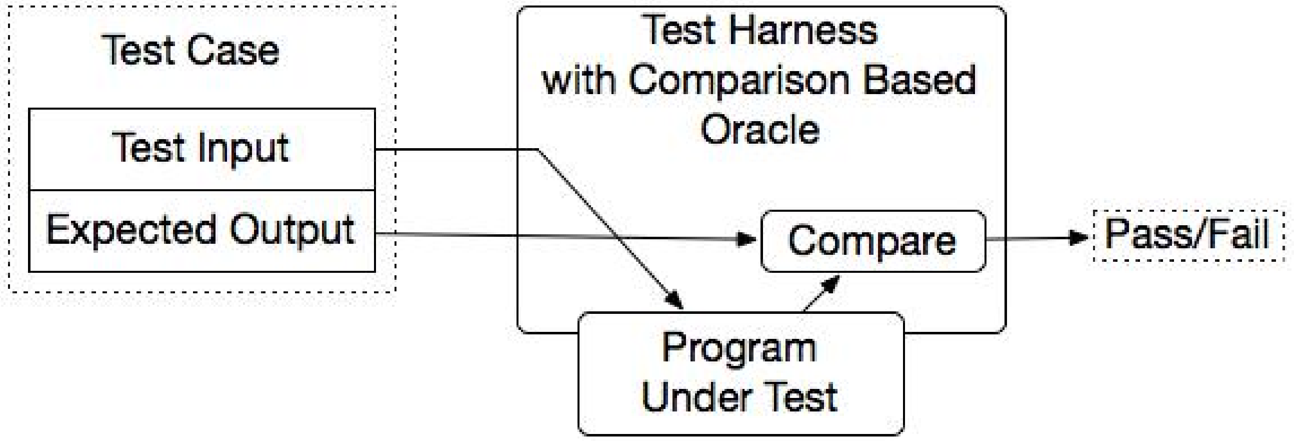
- No use running 10,000 test cases automatically, if the results must be checked by hand.

- It's a problem of 'range of specific to general', again
  - E.g., JUnit: Specific oracle ("assert") should be coded by hand in each test case.

- Typical approaches
  - Comparison-based oracle with predicted output value
  - Self-checks

Comparison-based Oracle

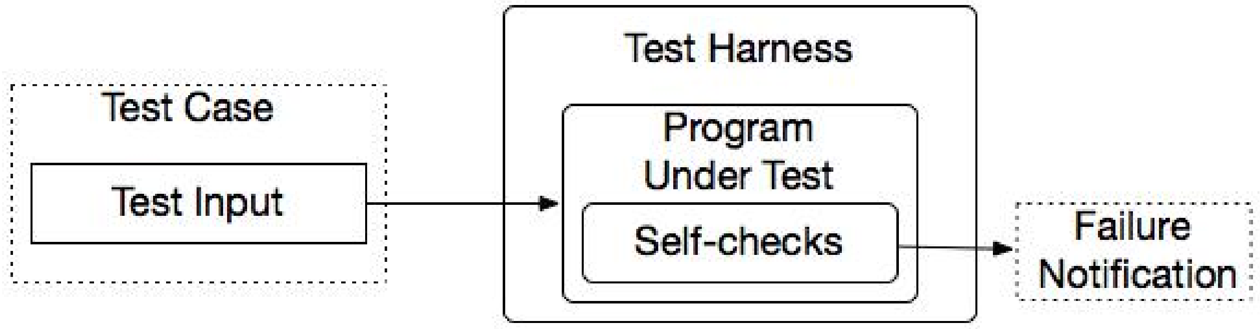
- With a comparison-based oracle, we need predicted output for each input.
  - Oracle compares actual to predicted output, and reports failure if they differ.
  - Fine for a small number of hand-generated test cases
  - E.g., for hand-written JUnit test cases



Self-Checks as Oracles

- An oracle can also be written as self-checks.
  - Often possible to judge correctness without predicting results

- Advantages and limits: Usable with large, automatically generated test suites, but often only a partial check
  - E.g., structural invariants of data structures
  - Recognize many or most failures, but not all



TESTING IN FUNCTIONAL SAFETY STANDARDS (IEC 61508, ISO-26262)

Contents and Pages of the Standard ISO-26262

I. ISO 26262

■ ISO 26262는

ISO 26262 is the adaptation of IEC 61508 to comply with needs specific to the application sector of electrical and/or electronic (E/E) systems within road vehicles.

ISO 26262는 **전장 E/E시스템**에 적합하게 **IEC 61508**를 **특화** 한 표준

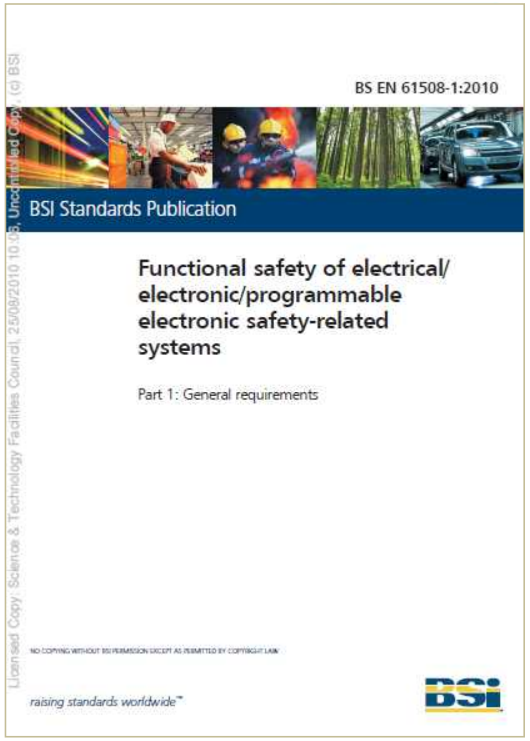
With the trend of increasing technological complexity, software content and mechatronic implementation, there are increasing risks from systematic failures and random hardware failures. ISO 26262 includes guidance to avoid these risks by providing appropriate requirements and processes.

전장시스템의 발전과 함께, **Systematic Failures**와 **Random Failures**로 인한 **Risk 증가**
+ **ISO 26262**는 이런 risk를 줄일 수 있는 **요구사항(Requirements)**과 **프로세스(processes)**를 제공한다.

Safety Lifecycle
+
Functional Safety
+
ASIL

1.130
systematic failure
failure (1.39), related in a deterministic way to a certain cause, that can only be eliminated by a change of the design or of the manufacturing process, operational procedures, documentation or other relevant factors

1.92
random hardware failure
failure (1.39) that can occur unpredictably during the lifetime of a hardware **element** (1.32) and that follows a probability distribution



BS EN 61508-1:2010

BSI Standards Publication

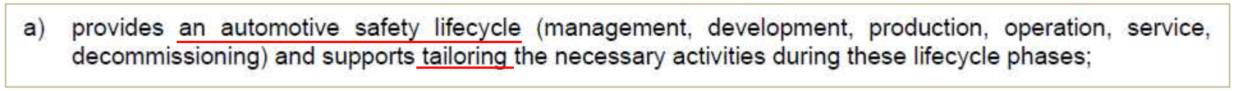
Functional safety of electrical/electronic/programmable electronic safety-related systems

Part 1: General requirements

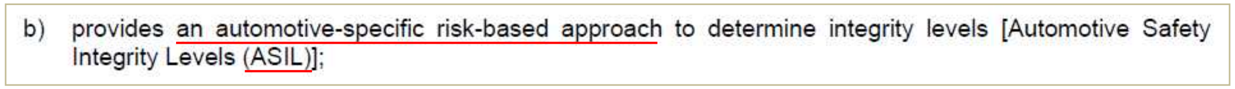
raising standards worldwide™

BSI

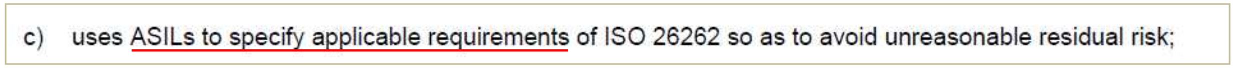
I. ISO 26262


a) provides an automotive safety lifecycle (management, development, production, operation, service, decommissioning) and supports tailoring the necessary activities during these lifecycle phases;

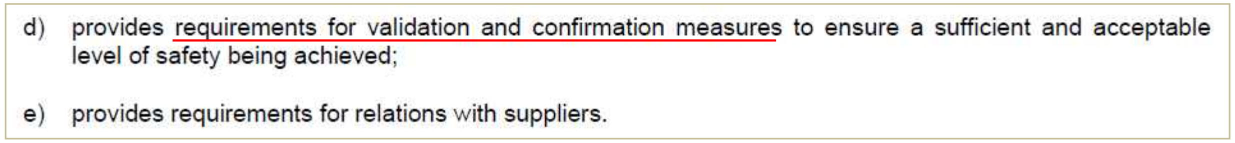
+ **전장시스템**을 위한 **Safety Lifecycle**을 제공한다.
+ Safety Lifecycle을 적절하게 **수정(Tailoring)**할 수 있는 방법을 제공한다.


b) provides an automotive-specific risk-based approach to determine integrity levels [Automotive Safety Integrity Levels (ASIL)];

+ 전장시스템에 특화된 **리스크** 기반 **SIL(Safety Integrity Level)**을 제공한다.
→ **ASIL(Automotive SIL)**


c) uses ASILs to specify applicable requirements of ISO 26262 so as to avoid unreasonable residual risk;

+ **ASIL을 이용**해서 적용해야 하는 ISO 26262 모든 요구사항을 **분류**한다.


d) provides requirements for validation and confirmation measures to ensure a sufficient and acceptable level of safety being achieved;

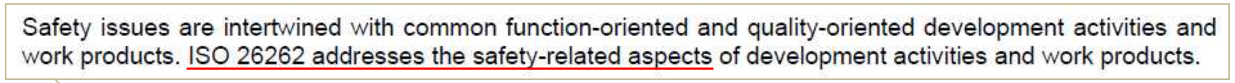
e) provides requirements for relations with suppliers.

+ 요구되는 수준의 안전성(즉, ASIL)이 잘 만족되었는지 확인할 수 있는
Validation 및 Confirmation 방법(Measures)에 대한 요구사항을 제공한다.

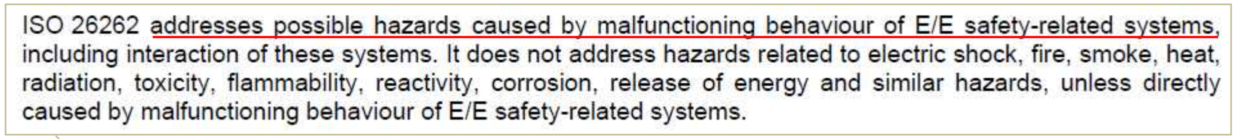
+ V&C 방법들이 가져야 할 요건들을 제공
+ 구체적인 V&C 방법을 제시 X

I. ISO 26262

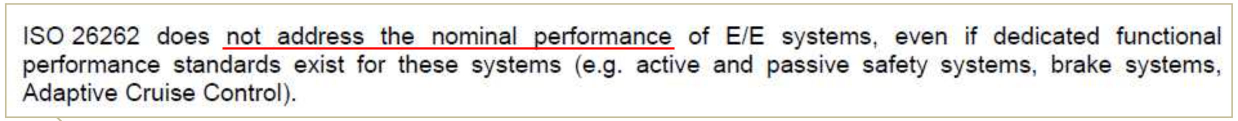
■ ISO 26262 기본 가정 (Assumptions)


Safety issues are intertwined with common function-oriented and quality-oriented development activities and work products. ISO 26262 addresses the safety-related aspects of development activities and work products.

시스템을 구성하는 일반 기능(Non-Safety Functions)의 개발에 대해서는 고려하지 않는다.
+ **Safety-Related Functions**의 개발만을 다룬다.


ISO 26262 addresses possible hazards caused by malfunctioning behaviour of E/E safety-related systems, including interaction of these systems. It does not address hazards related to electric shock, fire, smoke, heat, radiation, toxicity, flammability, reactivity, corrosion, release of energy and similar hazards, unless directly caused by malfunctioning behaviour of E/E safety-related systems.

E/E Safety-Related System의 **오작동(Malfunctioning)으로 인한 Hazards**만을 고려한다.
+ electric shock, fire, smoke, heat 등은 고려 X


ISO 26262 does not address the nominal performance of E/E systems, even if dedicated functional performance standards exist for these systems (e.g. active and passive safety systems, brake systems, Adaptive Cruise Control).

성능(Performance) 관련 이슈는 고려 X

I. ISO 26262

■ V-Model 기반 개발 방법론

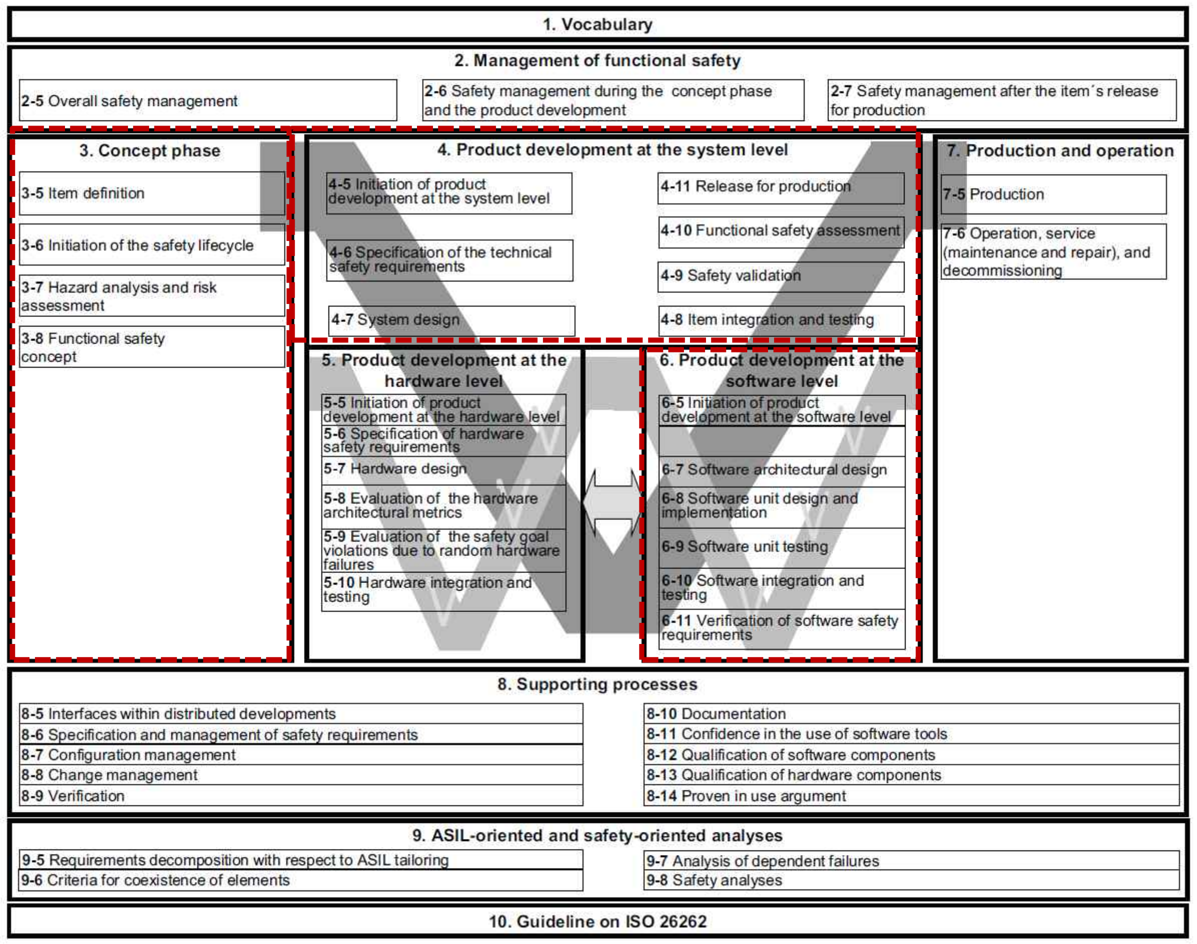

1. Vocabulary
2. Management of functional safety
2-5 Overall safety management
2-6 Safety management during the concept phase and the product development
2-7 Safety management after the item´s release for production
3. Concept phase
3-5 Item definition
3-6 Initiation of the safety lifecycle
3-7 Hazard analysis and risk assessment
3-8 Functional safety concept
4. Product development at the system level
4-5 Initiation of product development at the system level
4-6 Specification of the technical safety requirements
4-7 System design
4-11 Release for production
4-10 Functional safety assessment
4-9 Safety validation
4-8 Item integration and testing
7. Production and operation
7-5 Production
7-6 Operation, service (maintenance and repair), and decommissioning
5. Product development at the hardware level
5-5 Initiation of product development at the hardware level
5-6 Specification of hardware safety requirements
5-7 Hardware design
5-8 Evaluation of the hardware architectural metrics
5-9 Evaluation of the safety goal violations due to random hardware failures
5-10 Hardware integration and testing
6. Product development at the software level
6-5 Initiation of product development at the software level
6-6 Specification of software safety requirements
6-7 Software architectural design
6-8 Software unit design and implementation
6-9 Software unit testing
6-10 Software integration and testing
6-11 Verification of software safety requirements
8. Supporting processes
8-5 Interfaces within distributed developments
8-6 Specification and management of safety requirements
8-7 Configuration management
8-8 Change management
8-9 Verification
8-10 Documentation
8-11 Confidence in the use of software tools
8-12 Qualification of software components
8-13 Qualification of hardware components
8-14 Proven in use argument
9. ASIL-oriented and safety-oriented analyses
9-5 Requirements decomposition with respect to ASIL tailoring
9-6 Criteria for coexistence of elements
9-7 Analysis of dependent failures
9-8 Safety analyses
10. Guideline on ISO 26262

Figure 1 — Overview of ISO 26262

III. ISO 26262-2

■ Safety Lifecycle

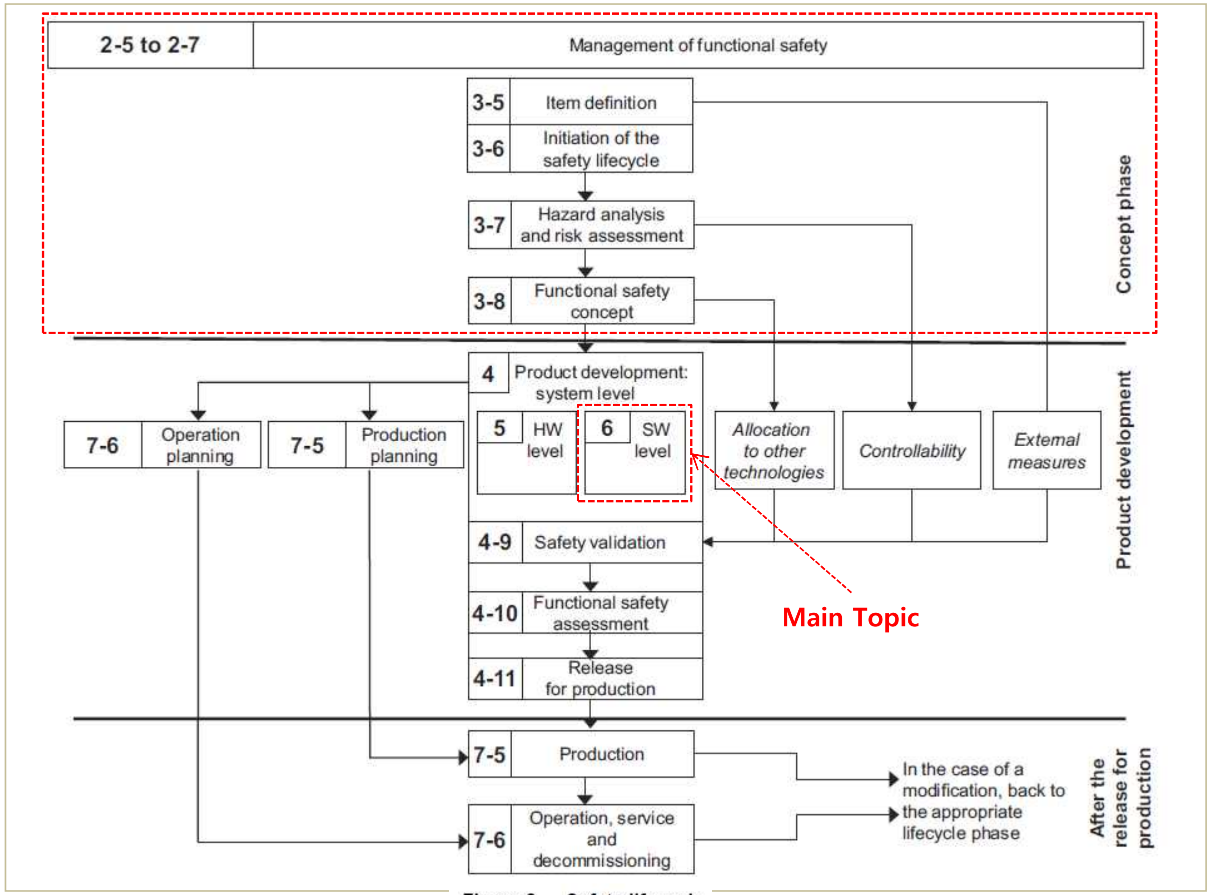
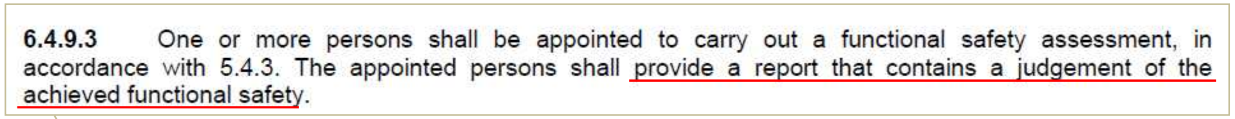

2-5 to 2-7
Management of functional safety
3-5 Item definition
3-6 Initiation of the safety lifecycle
3-7 Hazard analysis and risk assessment
3-8 Functional safety concept
Concept phase
4 Product development: system level
5 HW level
6 SW level
7-6 Operation planning
7-5 Production planning
Allocation to other technologies
Controllability
External measures
Product development
4-9 Safety validation
4-10 Functional safety assessment
4-11 Release for production
Main Topic
7-5 Production
7-6 Operation, service and decommissioning
In the case of a modification, back to the appropriate lifecycle phase
After the release for production

Figure 2 — Safety lifecycle

KU KONKUK UNIVERSITY

III. ISO 26262-2

■ Functional Safety Assessment



6.4.9.3 One or more persons shall be appointed to carry out a functional safety assessment, in accordance with 5.4.3. The appointed persons shall provide a report that contains a judgement of the achieved functional safety.

각 **Item**이 요구되는 **기능안전성(Functional Safety)**을 **확보**했는지 **판정(Judgment)**한다.

+ 합격
+ 불합격
+ 조건부 합격

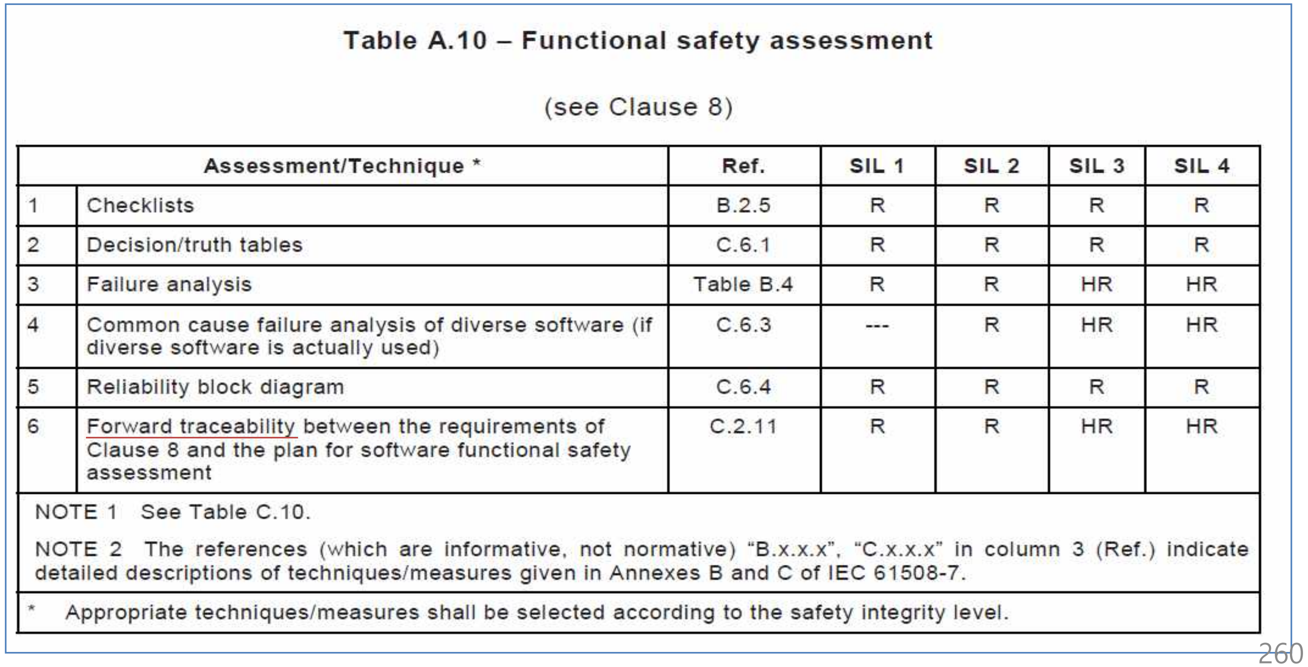


Table A.10 – Functional safety assessment

(see Clause 8)

Assessment/Technique *		Ref.	SIL 1	SIL 2	SIL 3	SIL 4
1	Checklists	B.2.5	R	R	R	R
2	Decision/truth tables	C.6.1	R	R	R	R
3	Failure analysis	Table B.4	R	R	HR	HR
4	Common cause failure analysis of diverse software (if diverse software is actually used)	C.6.3	---	R	HR	HR
5	Reliability block diagram	C.6.4	R	R	R	R
6	Forward traceability between the requirements of Clause 8 and the plan for software functional safety assessment	C.2.11	R	R	HR	HR

NOTE 1 See Table C.10.

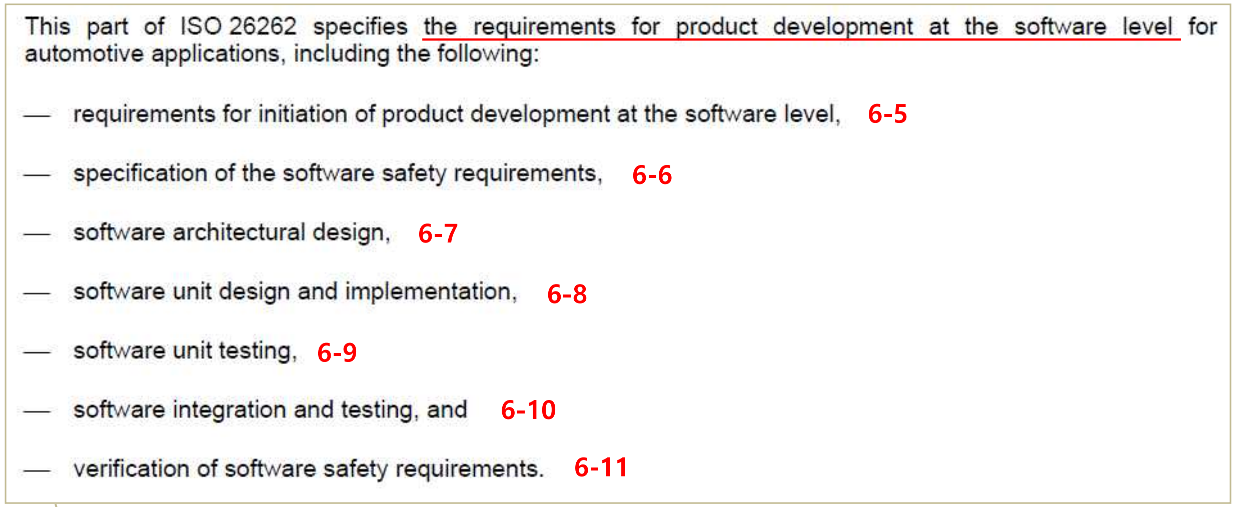
NOTE 2 The references (which are informative, not normative) "B.x.x.x", "C.x.x.x" in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7.

* Appropriate techniques/measures shall be selected according to the safety integrity level.

DEPENDABLE SOFTWARE LABORATORY

V. ISO 26262-6

■ Part 6: Product Development at the Software Level



This part of ISO 26262 specifies the requirements for product development at the software level for automotive applications, including the following:

— requirements for initiation of product development at the software level, 6-5

— specification of the software safety requirements, 6-6

— software architectural design, 6-7

— software unit design and implementation, 6-8

— software unit testing, 6-9

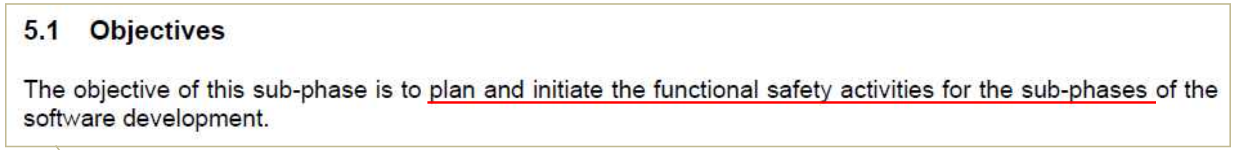
— software integration and testing, and 6-10

— verification of software safety requirements. 6-11

ISO 26262 Part 6는 **Software의 개발**에 대한 요구사항을 정의한다.
+ 6-5 Requirements for Initiation of Product Development at the Software Level : 소프트웨어 개발 프로세스를 시작
+ 6-6 Specification of the Software Safety Requirements : 요구사항 분석 및 명세
+ 6-7 Software Architecture Design : 상위 설계
+ 6-8 Software Unit Design and Implementation : 유닛 설계 및 구현
+ 6-9 Software Unit Testing : 단위 시험
+ 6-10 Software Integration and Testing : 모듈 통합 및 통합 시험
+ 6-11 Verification of Software Safety Requirements : 안전 요구사항 검증

6-5 Initiation of Product Development at the Software Level

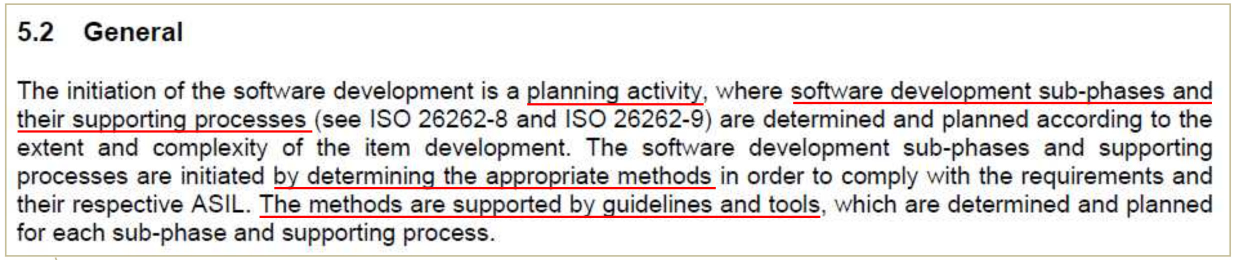
■ 6-5 Initiation of Product Development at the Software Level



5.1 Objectives

The objective of this sub-phase is to plan and initiate the functional safety activities for the sub-phases of the software development.

소프트웨어 개발을 위한 **계획**을 세우고, **주요 Activities**를 정한다.
→ 소프트웨어 개발 프로세스를 준비한다.



5.2 General

The initiation of the software development is a planning activity, where software development sub-phases and their supporting processes (see ISO 26262-8 and ISO 26262-9) are determined and planned according to the extent and complexity of the item development. The software development sub-phases and supporting processes are initiated by determining the appropriate methods in order to comply with the requirements and their respective ASIL. The methods are supported by guidelines and tools, which are determined and planned for each sub-phase and supporting process.

소프트웨어 개발 계획은
+ 소프트웨어 개발 **각 단계들(Sub-Phases)**과 **지원 프로세스들(Supporting Processes)**를 결정한다.
+ **적절한 방법론(Methods)**에 의해 결정된다.
+ **가이드라인(Guidelines)**과 **도구(Tools)**에 의해 지원받는다.

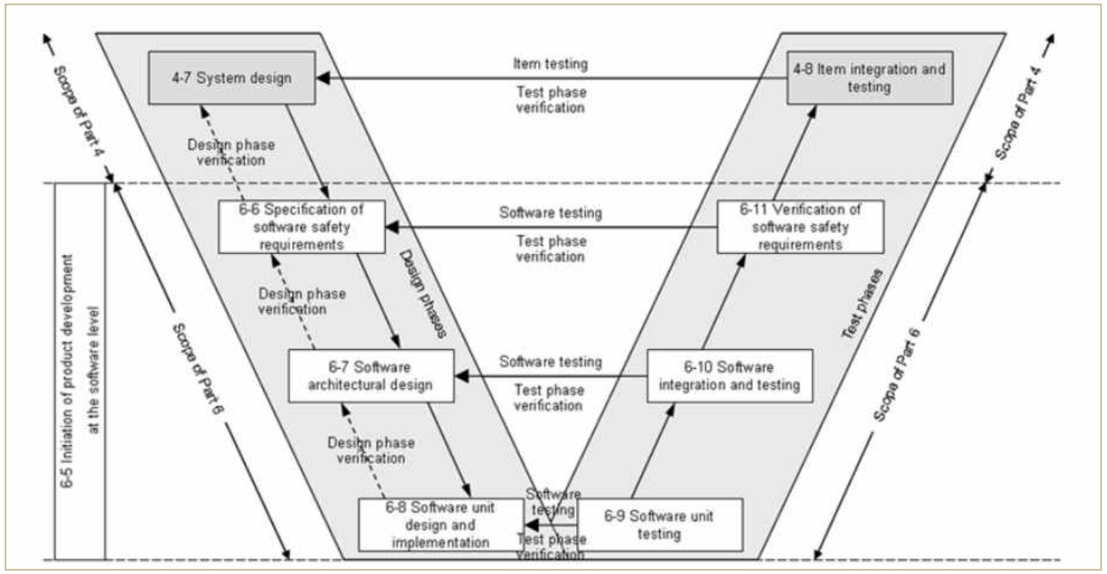
적절한 방법론(Appropriate Methods)은 **(Technical Safety Requirements + ASIL)**을 효과적으로 처리할 수 있어야 한다.

6-5 Initiation of Product Development at the Software Level

5.4.1 The activities and the determination of appropriate methods for the product development at the software level shall be planned.

5.4.2 The tailoring of the lifecycle for product development at the software level shall be performed in accordance with ISO 26262-2:2011, 6.4.5, and based on the reference phase model given in Figure 2.

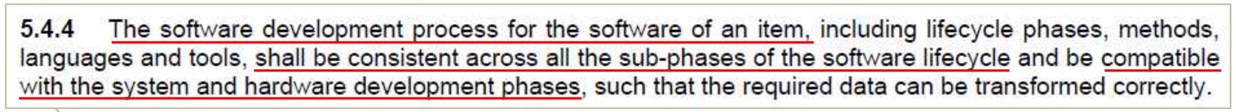
소프트웨어 개발을 위한 **프로세스 Activities**와 사용할 적절한 **방법론(Methods)**를 계획한다.
+ **SDLC(Software Development Life-Cycle)**을 **수정(Tailoring)**할 수 있다.
+ **V-Model** 사용**(Figure 2)**



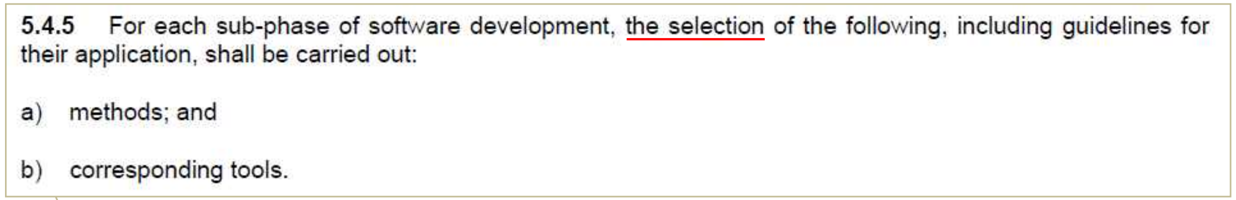
Scope of Part 4
6-5 Initiation of product development at the software level
4-7 System design
Item testing
Test phase verification
4-8 Item integration and testing
Scope of Part 4
Design phase verification
6-6 Specification of software safety requirements
Software testing
Test phase verification
6-11 Verification of software safety requirements
Design phases
Test phases
Design phase verification
Scope of Part 6
Scope of Part 6
6-7 Software architectural design
Software testing
Test phase verification
6-10 Software integration and testing
Design phase verification
6-8 Software unit design and implementation
Software testing
Test phase verification
6-9 Software unit testing

Figure 2 — Reference phase model for the software development

6-5 Initiation of Product Development at the Software Level


5.4.4 The software development process for the software of an item, including lifecycle phases, methods, languages and tools, shall be consistent across all the sub-phases of the software lifecycle and be compatible with the system and hardware development phases, such that the required data can be transformed correctly.

소프트웨어 개발 프로세스(+ Lifecycle phases, 방법론, 가이드라인(언어) 및 도구)는 전체적으로 **일관(Consistent)**되어야 한다.
+ 시스템 및 하드웨어 개발 프로세스와 **호환(Compatible)**되어야 한다.

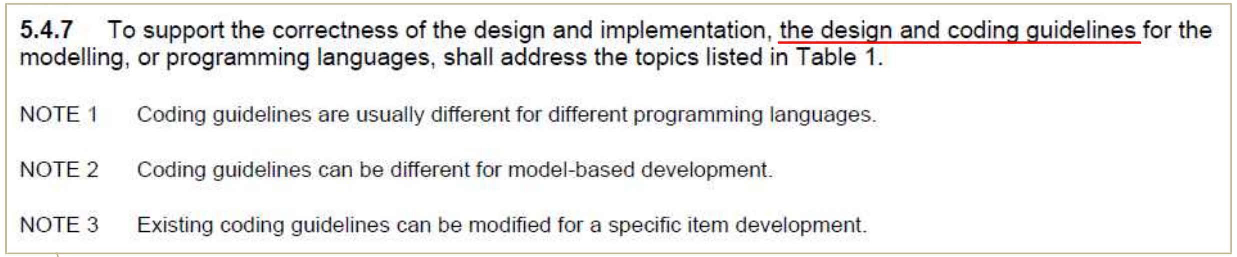

5.4.5 For each sub-phase of software development, the selection of the following, including guidelines for their application, shall be carried out:

a) methods; and

b) corresponding tools.

SDLC 각 단계에서 사용할, **적절한(Appropriate)**
  + **방법론(Methods)**
  + **도구(Tools)** : 방법론의 사용을 돕는 (자동화) 도구
  + **가이드라인(Guidelines)** : 방법론+도구의 사용 설명서 및 예제(Best Practice)
을 **선정**해야 한다.

6-5 Initiation of Product Development at the Software Level 5/16


5.4.7 To support the correctness of the design and implementation, the design and coding guidelines for the modelling, or programming languages, shall address the topics listed in Table 1.

NOTE 1 Coding guidelines are usually different for different programming languages.

NOTE 2 Coding guidelines can be different for model-based development.

NOTE 3 Existing coding guidelines can be modified for a specific item development.

모델링 언어 및 **프로그래밍 언어**를 정확하게 사용하기 위한 **가이드라인(Guideline)**을 제공해야 한다.
- + 사용 언어에 따라 상이
- + 모델 기반 방법론에서는 다르게 적용됨
- + 기존에 있는 가이드라인을 적용 대상(A Specific Item)에 따라 적절하게 수정 가능

가이드라인은 Table 1.의 항목(Topics)들을 포함해야 한다.

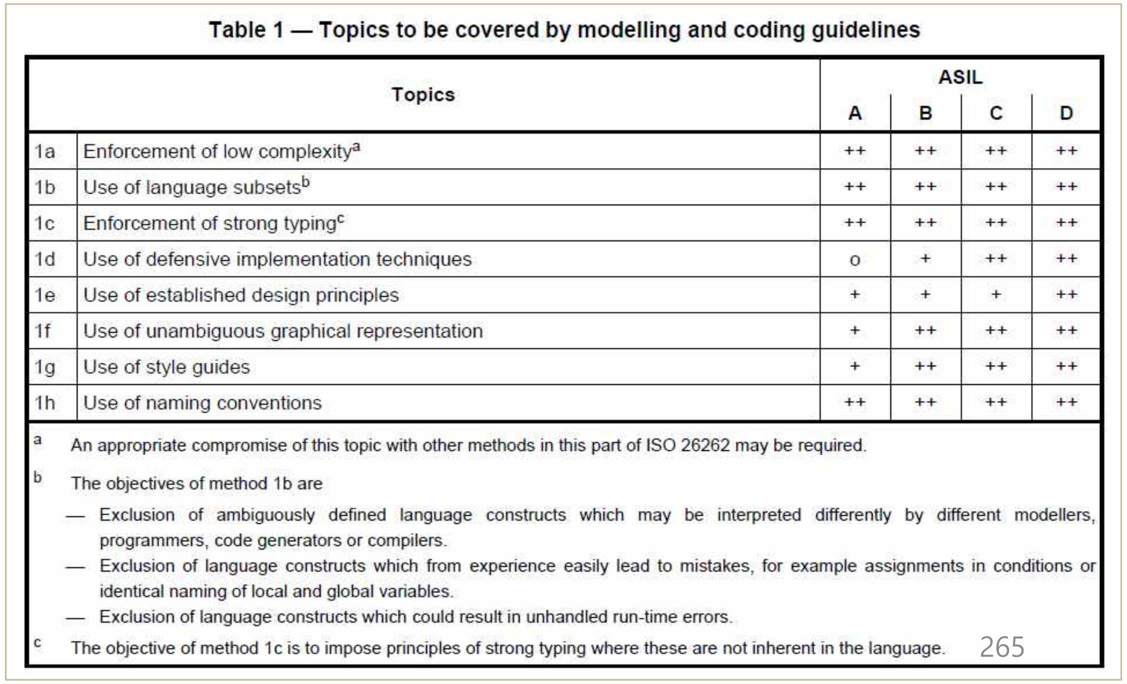


Table 1 — Topics to be covered by modelling and coding guidelines

Topics		ASIL			
		A	B	C	D
1a	Enforcement of low complexity[a]	++	++	++	++
1b	Use of language subsets[b]	++	++	++	++
1c	Enforcement of strong typing[c]	++	++	++	++
1d	Use of defensive implementation techniques	o	+	++	++
1e	Use of established design principles	+	+	+	++
1f	Use of unambiguous graphical representation	+	++	++	++
1g	Use of style guides	+	++	++	++
1h	Use of naming conventions	++	++	++	++

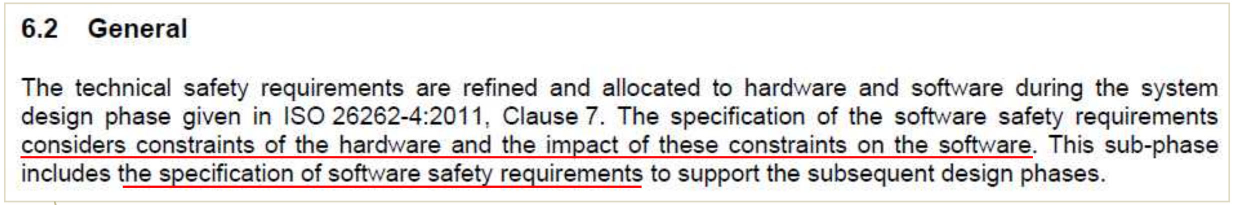
[a] An appropriate compromise of this topic with other methods in this part of ISO 26262 may be required.

[b] The objectives of method 1b are
— Exclusion of ambiguously defined language constructs which may be interpreted differently by different modellers, programmers, code generators or compilers.
— Exclusion of language constructs which from experience easily lead to mistakes, for example assignments in conditions or identical naming of local and global variables.
— Exclusion of language constructs which could result in unhandled run-time errors.

[c] The objective of method 1c is to impose principles of strong typing where these are not inherent in the language.

6-6 Specification of Software Safety Requirements

■ 6-6 Specification of Software Safety Requirements

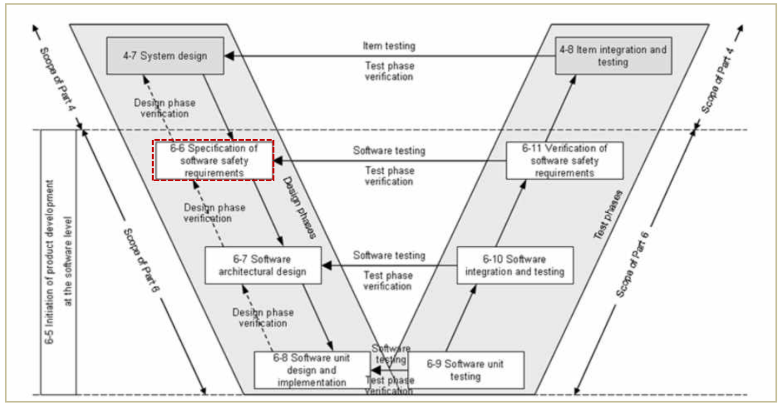


6.2 General

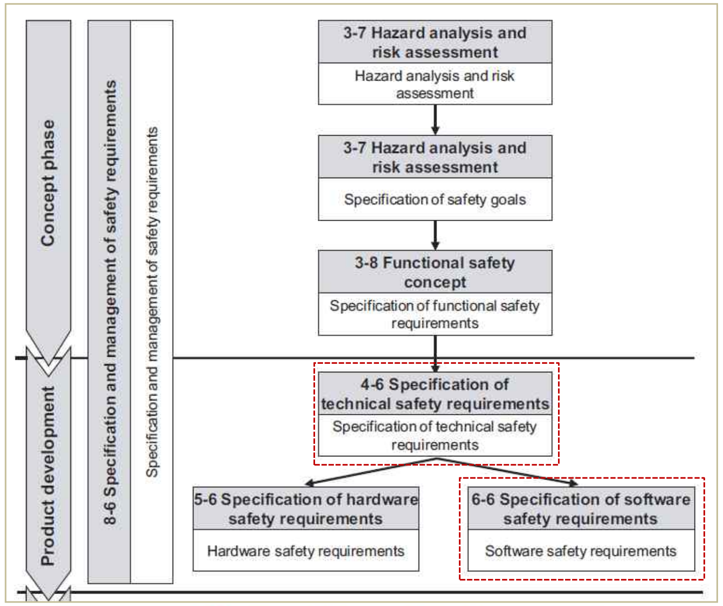
The technical safety requirements are refined and allocated to hardware and software during the system design phase given in ISO 26262-4:2011, Clause 7. The specification of the software safety requirements considers constraints of the hardware and the impact of these constraints on the software. This sub-phase includes the specification of software safety requirements to support the subsequent design phases.

Technical Safety Requirements는 **26262-4**에서
+ **명세(Specification)**되고
+ **Hardware/Software**에 **할당(Allocation)**된다.

6-6에서는
+ Hardware로 인한 **제약사항(Constraints)**과
+ Software에 미치는 **영향(Impact)**을 고려하여
Software Safety Requirements를 작성한다.



4-7 System design
Item testing
Test phase verification
4-8 Item integration and testing
Design phase verification
6-6 Specification of software safety requirements
Software testing
Test phase verification
6-11 Verification of software safety requirements
Design phases
Test phases
Design phase verification
6-7 Software architectural design
Software testing
Test phase verification
6-10 Software integration and testing
Design phase verification
6-8 Software unit design and implementation
Software testing
Test phase verification
6-9 Software unit testing
Scope of Part 4
Scope of Part 6
Scope of Part 4
Scope of Part 6
6-5 Initiation of product development at the software level

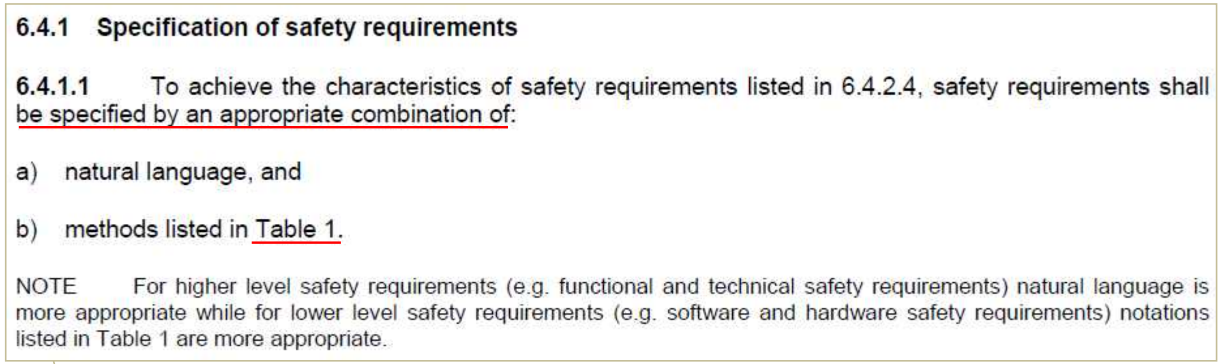


Concept phase
Product development
8-6 Specification and management of safety requirements
Specification and management of safety requirements
3-7 Hazard analysis and risk assessment
Hazard analysis and risk assessment
3-7 Hazard analysis and risk assessment
Specification of safety goals
3-8 Functional safety concept
Specification of functional safety requirements
4-6 Specification of technical safety requirements
Specification of technical safety requirements
5-6 Specification of hardware safety requirements
Hardware safety requirements
6-6 Specification of software safety requirements
Software safety requirements

Figure 3 — Structure of the safety requirements (26262-3)

6-6 Specification of Software Safety Requirements

(26262-8 Clause 6)



6.4.1 Specification of safety requirements

6.4.1.1 To achieve the characteristics of safety requirements listed in 6.4.2.4, safety requirements shall be specified by an appropriate combination of:

a) natural language, and

b) methods listed in Table 1.

NOTE For higher level safety requirements (e.g. functional and technical safety requirements) natural language is more appropriate while for lower level safety requirements (e.g. software and hardware safety requirements) notations listed in Table 1 are more appropriate.

26262-8: Clause 6
+ **Safety Requirements Specification**은 **자연어**나 **비정형·준정형·정형명세**의 **조합**으로 **명세** 되어야 한다.

Safety Requirements가 가져야 할 **속성**
+ Unambiguous and Comprehensive
+ Atomic
+ Internally Consistent
+ Feasible
+ Verifiable

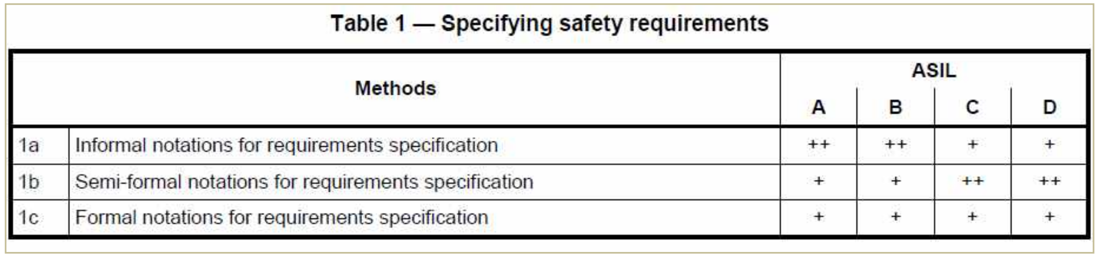


Table 1 — Specifying safety requirements

Methods		ASIL			
		A	B	C	D
1a	Informal notations for requirements specification	++	++	+	+
1b	Semi-formal notations for requirements specification	+	+	++	++
1c	Formal notations for requirements specification	+	+	+	+

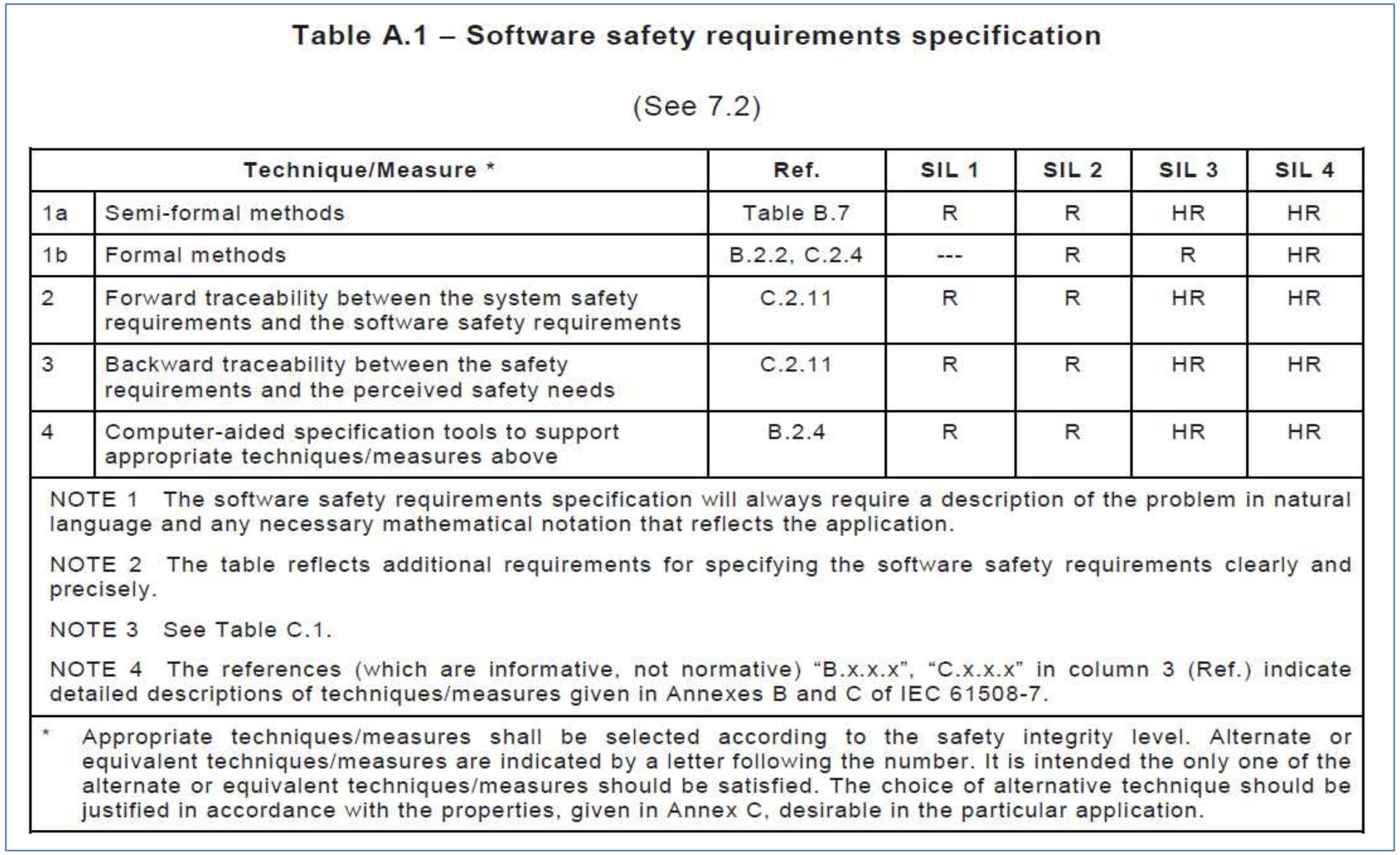


Table A.1 – Software safety requirements specification

(See 7.2)

Technique/Measure *		Ref.	SIL 1	SIL 2	SIL 3	SIL 4
1a	Semi-formal methods	Table B.7	R	R	HR	HR
1b	Formal methods	B.2.2, C.2.4	---	R	R	HR
2	Forward traceability between the system safety requirements and the software safety requirements	C.2.11	R	R	HR	HR
3	Backward traceability between the safety requirements and the perceived safety needs	C.2.11	R	R	HR	HR
4	Computer-aided specification tools to support appropriate techniques/measures above	B.2.4	R	R	HR	HR

NOTE 1 The software safety requirements specification will always require a description of the problem in natural language and any necessary mathematical notation that reflects the application.

NOTE 2 The table reflects additional requirements for specifying the software safety requirements clearly and precisely.

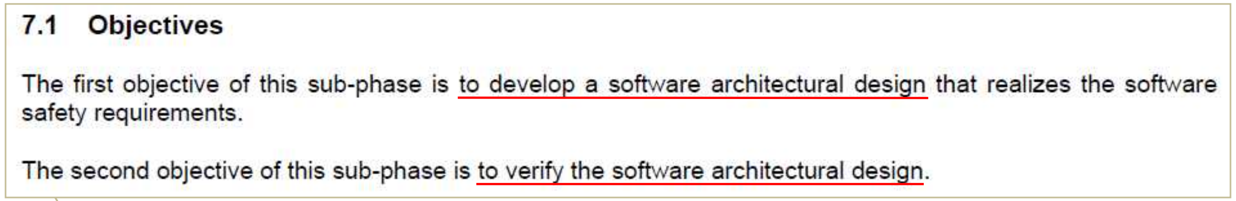
NOTE 3 See Table C.1.

NOTE 4 The references (which are informative, not normative) "B.x.x.x", "C.x.x.x" in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7.

* Appropriate techniques/measures shall be selected according to the safety integrity level. Alternate or equivalent techniques/measures are indicated by a letter following the number. It is intended the only one of the alternate or equivalent techniques/measures should be satisfied. The choice of alternative technique should be justified in accordance with the properties, given in Annex C, desirable in the particular application.

6-7 Software Architectural Design

■ 6-7. Software Architectural Design



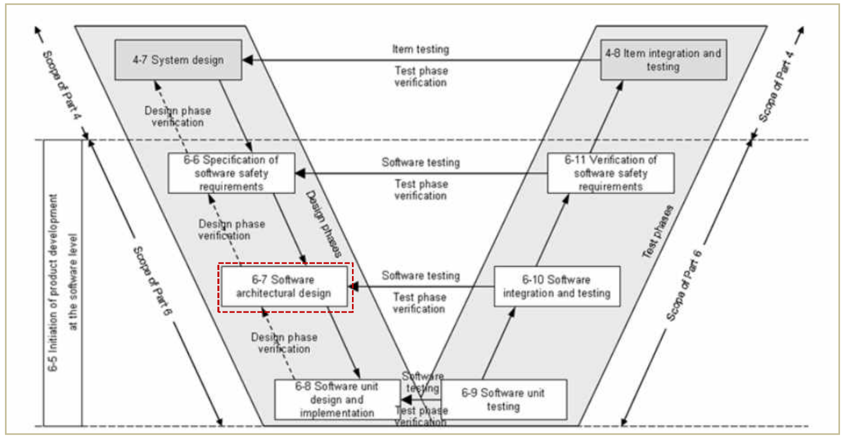
7.1 Objectives

The first objective of this sub-phase is to develop a software architectural design that realizes the software safety requirements.

The second objective of this sub-phase is to verify the software architectural design.

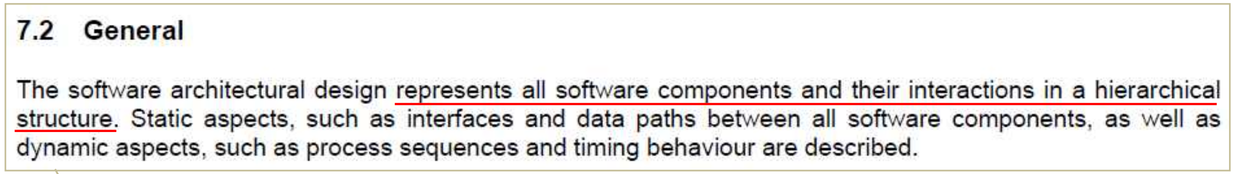
Software Safety Requirements를 구현(실재화: Realization)할 수 있는 **Software Architectural Design**을 **개발**한다.

개발한 **Software Architectural Design**을 **검증(Verify)**한다.



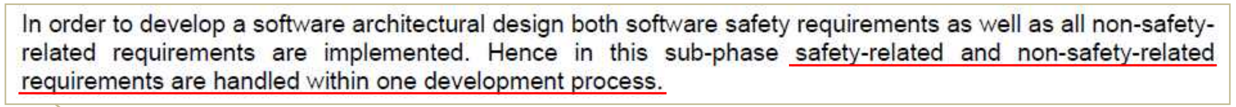
4-7 System design
Item testing
Test phase verification
4-8 Item integration and testing
Scope of Part 4
Scope of Part 4
Design phase verification
6-6 Specification of software safety requirements
Software testing
Test phase verification
6-11 Verification of software safety requirements
Design phases
Test phases
6-5 Initiation of product development at the software level
Scope of Part 6
Scope of Part 6
Design phase verification
6-7 Software architectural design
Software testing
Test phase verification
6-10 Software integration and testing
Design phase verification
6-8 Software unit design and implementation
Software testing
Test phase verification
6-9 Software unit testing

6-7 Software Architectural Design

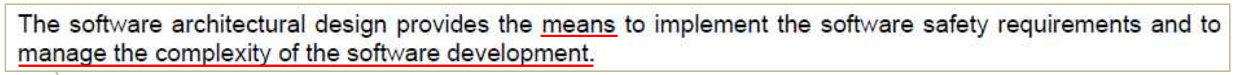

7.2 General

The software architectural design represents all software components and their interactions in a hierarchical structure. Static aspects, such as interfaces and data paths between all software components, as well as dynamic aspects, such as process sequences and timing behaviour are described.

Software Architectural Design은
- + **모든 Software Components**와 이들 간의 **Interactions**을 **계층적으로(in a Hierarchical Structure)**로 표현한다.
- + 정적 요소: interface / data paths (모든 software components의)
- + 동적 요소: process sequences , timing behavior


In order to develop a software architectural design both software safety requirements as well as all non-safety-related requirements are implemented. Hence in this sub-phase safety-related and non-safety-related requirements are handled within one development process.

Safety / Non-Safety Requirements를 모두 함께 고려한다.


The software architectural design provides the means to implement the software safety requirements and to manage the complexity of the software development.

Software Architectural Design은
- + Software Safety Requirements를 구현하고
- + 소프트웨어 개발의 **복잡도(Complexity)**를 **관리**할 수 있는 **방법들(Means)**을 제공한다.

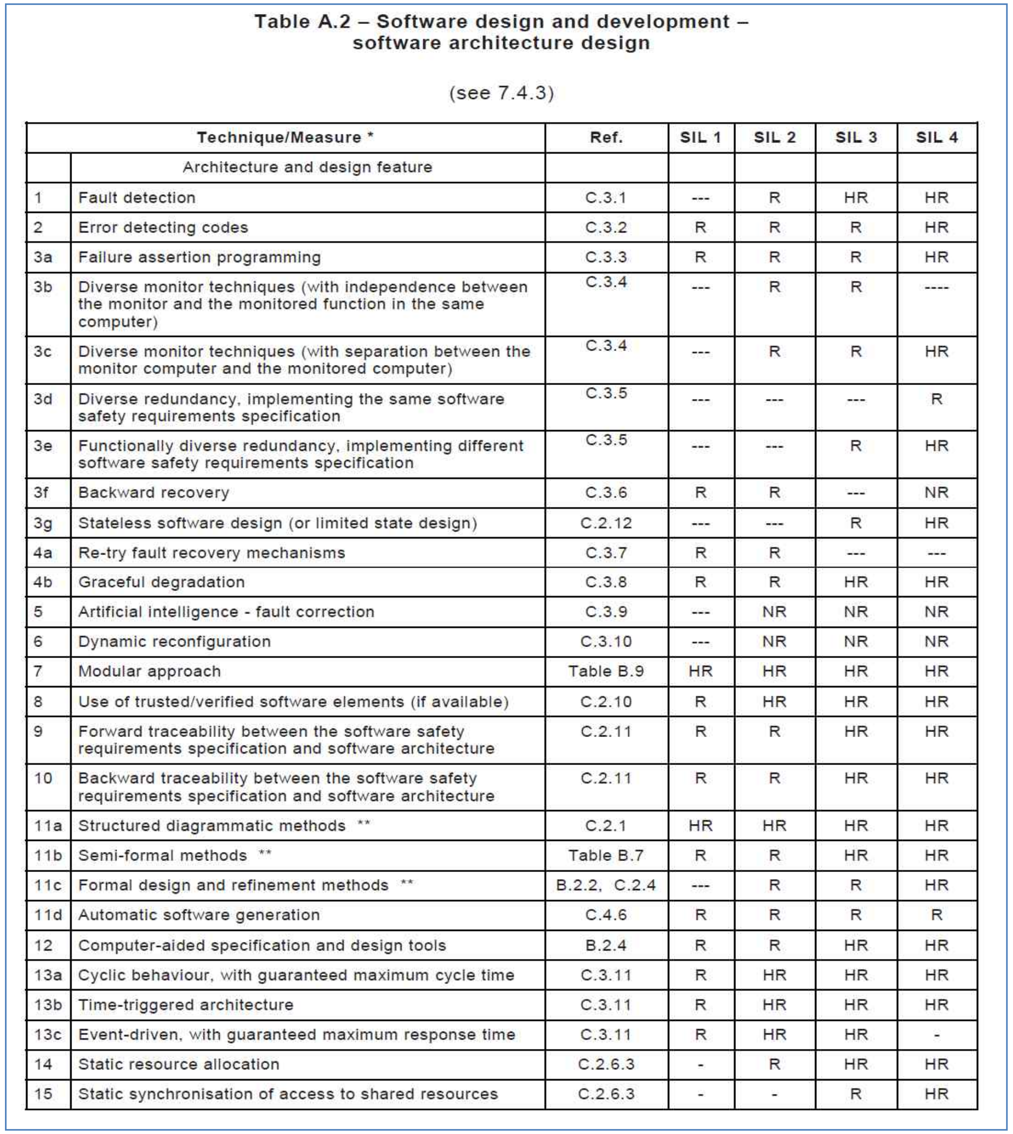
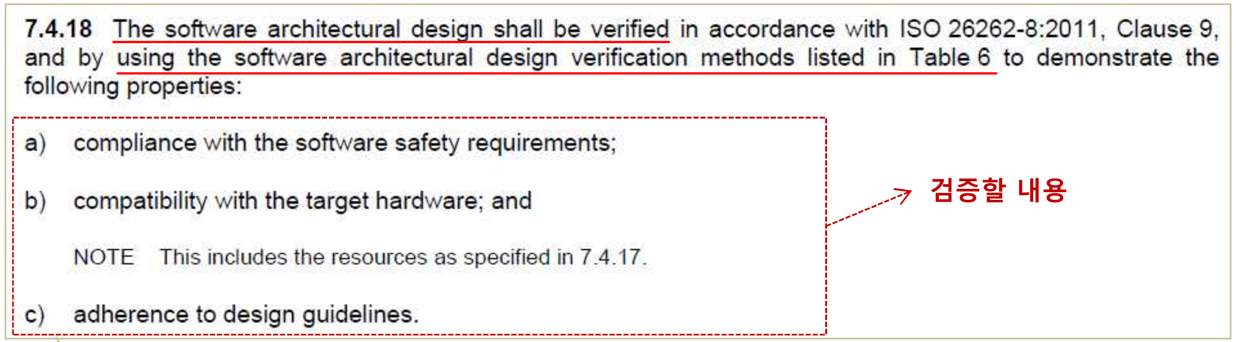


Table A.2 – Software design and development – software architecture design

(see 7.4.3)

	Technique/Measure *	Ref.	SIL 1	SIL 2	SIL 3	SIL 4
	Architecture and design feature					
1	Fault detection	C.3.1	---	R	HR	HR
2	Error detecting codes	C.3.2	R	R	R	HR
3a	Failure assertion programming	C.3.3	R	R	R	HR
3b	Diverse monitor techniques (with independence between the monitor and the monitored function in the same computer)	C.3.4	---	R	R	----
3c	Diverse monitor techniques (with separation between the monitor computer and the monitored computer)	C.3.4	---	R	R	HR
3d	Diverse redundancy, implementing the same software safety requirements specification	C.3.5	---	---	---	R
3e	Functionally diverse redundancy, implementing different software safety requirements specification	C.3.5	---	---	R	HR
3f	Backward recovery	C.3.6	R	R	---	NR
3g	Stateless software design (or limited state design)	C.2.12	---	---	R	HR
4a	Re-try fault recovery mechanisms	C.3.7	R	R	---	---
4b	Graceful degradation	C.3.8	R	R	HR	HR
5	Artificial intelligence - fault correction	C.3.9	---	NR	NR	NR
6	Dynamic reconfiguration	C.3.10	---	NR	NR	NR
7	Modular approach	Table B.9	HR	HR	HR	HR
8	Use of trusted/verified software elements (if available)	C.2.10	R	HR	HR	HR
9	Forward traceability between the software safety requirements specification and software architecture	C.2.11	R	R	HR	HR
10	Backward traceability between the software safety requirements specification and software architecture	C.2.11	R	R	HR	HR
11a	Structured diagrammatic methods **	C.2.1	HR	HR	HR	HR
11b	Semi-formal methods **	Table B.7	R	R	HR	HR
11c	Formal design and refinement methods **	B.2.2, C.2.4	---	R	R	HR
11d	Automatic software generation	C.4.6	R	R	R	R
12	Computer-aided specification and design tools	B.2.4	R	R	HR	HR
13a	Cyclic behaviour, with guaranteed maximum cycle time	C.3.11	R	HR	HR	HR
13b	Time-triggered architecture	C.3.11	R	HR	HR	HR
13c	Event-driven, with guaranteed maximum response time	C.3.11	R	HR	HR	-
14	Static resource allocation	C.2.6.3	-	R	HR	HR
15	Static synchronisation of access to shared resources	C.2.6.3	-	-	R	HR

6-7 Software Architectural Design



7.4.18 The software architectural design shall be verified in accordance with ISO 26262-8:2011, Clause 9, and by using the software architectural design verification methods listed in Table 6 to demonstrate the following properties:

a) compliance with the software safety requirements;

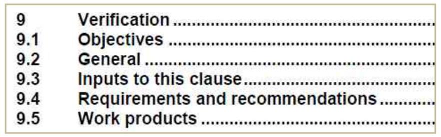
b) compatibility with the target hardware; and

NOTE This includes the resources as specified in 7.4.17.

c) adherence to design guidelines.

검증할 내용

26262-8 Clause 9를 이용하여
- + Software Architectural Design을 **검증(Verification)** 해야 한다.
- + **Table 6** 의 검증방법 사용



(26262-8 Clause 9)

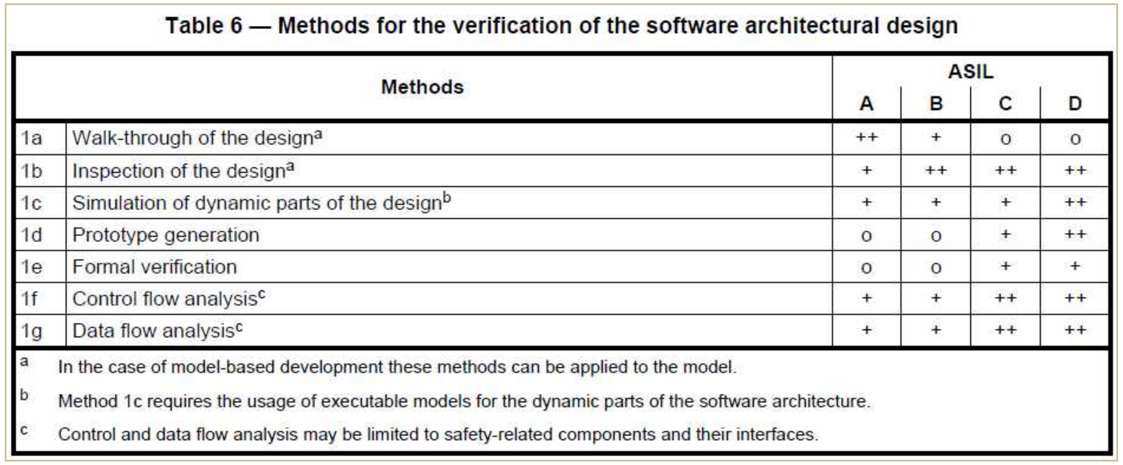


Table 6 — Methods for the verification of the software architectural design

	Methods	ASIL A	ASIL B	ASIL C	ASIL D
1a	Walk-through of the design[a]	++	+	o	o
1b	Inspection of the design[a]	+	++	++	++
1c	Simulation of dynamic parts of the design[b]	+	+	+	++
1d	Prototype generation	o	o	+	++
1e	Formal verification	o	o	+	+
1f	Control flow analysis[c]	+	+	++	++
1g	Data flow analysis[c]	+	+	++	++

[a] In the case of model-based development these methods can be applied to the model.

[b] Method 1c requires the usage of executable models for the dynamic parts of the software architecture.

[c] Control and data flow analysis may be limited to safety-related components and their interfaces.

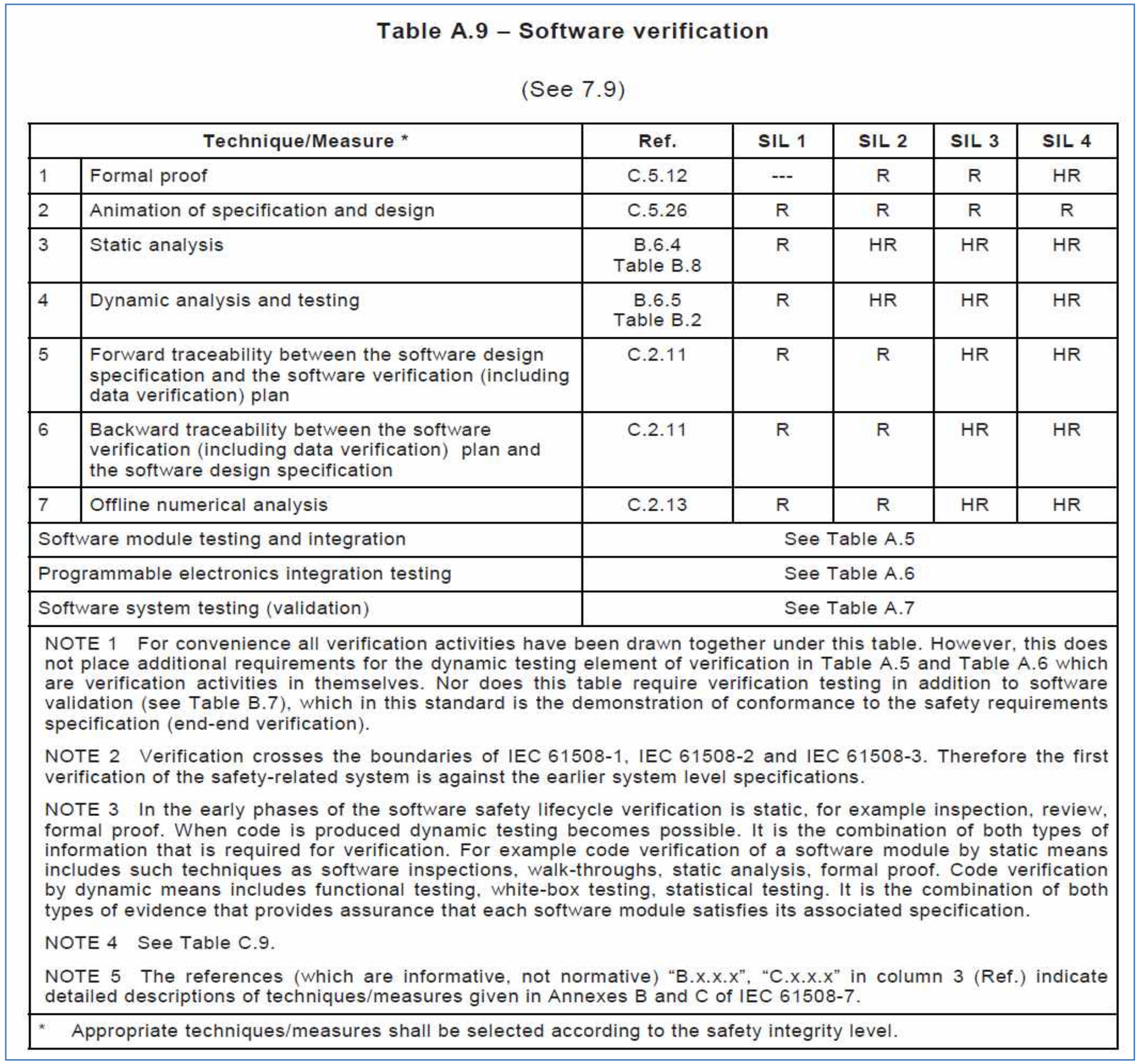


Table A.9 – Software verification

(See 7.9)

	Technique/Measure *	Ref.	SIL 1	SIL 2	SIL 3	SIL 4
1	Formal proof	C.5.12	---	R	R	HR
2	Animation of specification and design	C.5.26	R	R	R	R
3	Static analysis	B.6.4 Table B.8	R	HR	HR	HR
4	Dynamic analysis and testing	B.6.5 Table B.2	R	HR	HR	HR
5	Forward traceability between the software design specification and the software verification (including data verification) plan	C.2.11	R	R	HR	HR
6	Backward traceability between the software verification (including data verification) plan and the software design specification	C.2.11	R	R	HR	HR
7	Offline numerical analysis	C.2.13	R	R	HR	HR
Software module testing and integration		See Table A.5				
Programmable electronics integration testing		See Table A.6				
Software system testing (validation)		See Table A.7				

NOTE 1 For convenience all verification activities have been drawn together under this table. However, this does not place additional requirements for the dynamic testing element of verification in Table A.5 and Table A.6 which are verification activities in themselves. Nor does this table require verification testing in addition to software validation (see Table B.7), which in this standard is the demonstration of conformance to the safety requirements specification (end-end verification).

NOTE 2 Verification crosses the boundaries of IEC 61508-1, IEC 61508-2 and IEC 61508-3. Therefore the first verification of the safety-related system is against the earlier system level specifications.

NOTE 3 In the early phases of the software safety lifecycle verification is static, for example inspection, review, formal proof. When code is produced dynamic testing becomes possible. It is the combination of both types of information that is required for verification. For example code verification of a software module by static means includes such techniques as software inspections, walk-throughs, static analysis, formal proof. Code verification by dynamic means includes functional testing, white-box testing, statistical testing. It is the combination of both types of evidence that provides assurance that each software module satisfies its associated specification.

NOTE 4 See Table C.9.

NOTE 5 The references (which are informative, not normative) "B.x.x.x", "C.x.x.x" in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7.

* Appropriate techniques/measures shall be selected according to the safety integrity level.

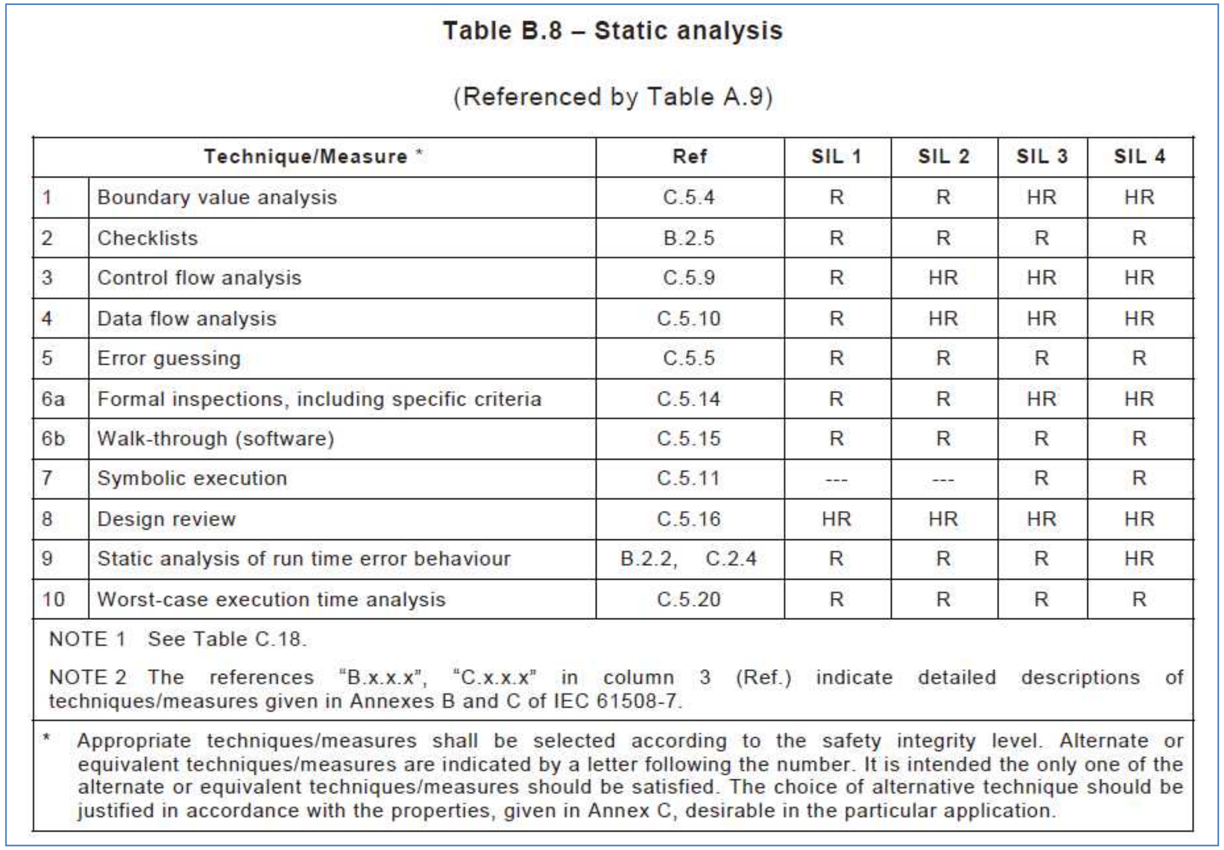


Table B.8 – Static analysis

(Referenced by Table A.9)

	Technique/Measure *	Ref	SIL 1	SIL 2	SIL 3	SIL 4
1	Boundary value analysis	C.5.4	R	R	HR	HR
2	Checklists	B.2.5	R	R	R	R
3	Control flow analysis	C.5.9	R	HR	HR	HR
4	Data flow analysis	C.5.10	R	HR	HR	HR
5	Error guessing	C.5.5	R	R	R	R
6a	Formal inspections, including specific criteria	C.5.14	R	R	HR	HR
6b	Walk-through (software)	C.5.15	R	R	R	R
7	Symbolic execution	C.5.11	---	---	R	R
8	Design review	C.5.16	HR	HR	HR	HR
9	Static analysis of run time error behaviour	B.2.2, C.2.4	R	R	R	HR
10	Worst-case execution time analysis	C.5.20	R	R	R	R

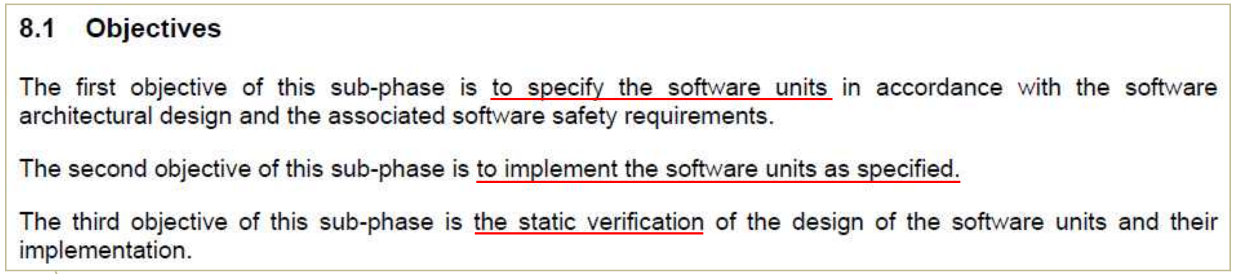
NOTE 1 See Table C.18.

NOTE 2 The references "B.x.x.x", "C.x.x.x" in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7.

* Appropriate techniques/measures shall be selected according to the safety integrity level. Alternate or equivalent techniques/measures are indicated by a letter following the number. It is intended the only one of the alternate or equivalent techniques/measures should be satisfied. The choice of alternative technique should be justified in accordance with the properties, given in Annex C, desirable in the particular application.

6-8 Software Unit Design and Implementation

■ 6-8 Software Unit Design and Implementation



8.1 Objectives

The first objective of this sub-phase is to specify the software units in accordance with the software architectural design and the associated software safety requirements.

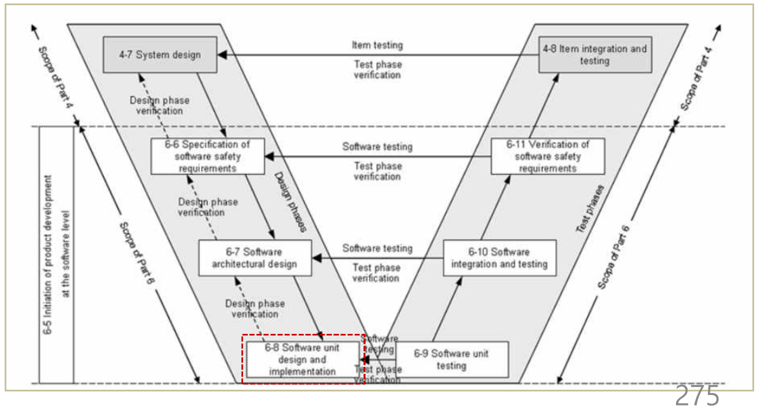
The second objective of this sub-phase is to implement the software units as specified.

The third objective of this sub-phase is the static verification of the design of the software units and their implementation.

Software Architectural Design과 **Software Safety Requirements**에 따라 **Software Unit**을 **상세설계(Detailed Design)**한다.

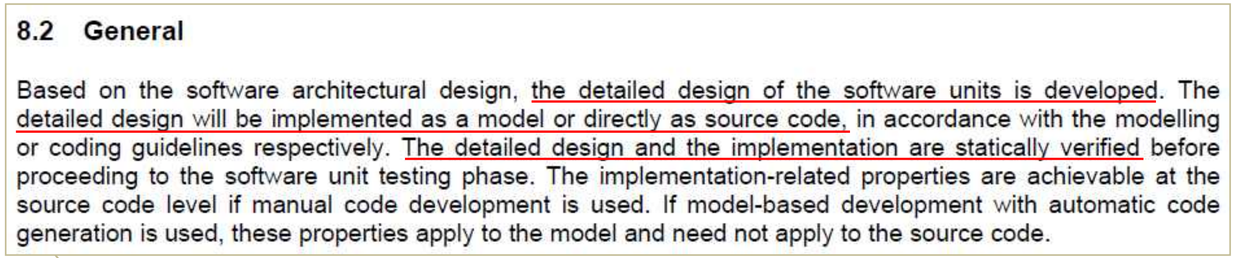
상세설계한 **Software Unit**을 **구현(Implementation)**한다.

Software Unit의 상세설계와 구현을 **정적검증(Static Verification)** 한다.



4-7 System design
Item testing
Test phase verification
4-8 Item integration and testing
Scope of Part 4
Design phase verification
6-6 Specification of software safety requirements
Software testing
Test phase verification
6-11 Verification of software safety requirements
Design phases
Test phases
Design phase verification
6-7 Software architectural design
Software testing
Test phase verification
6-10 Software integration and testing
Design phase verification
6-8 Software unit design and implementation
Software testing
Test phase verification
6-9 Software unit testing
6-5 Initiation of product development at the software level
Scope of Part 6
Scope of Part 4
Scope of Part 6

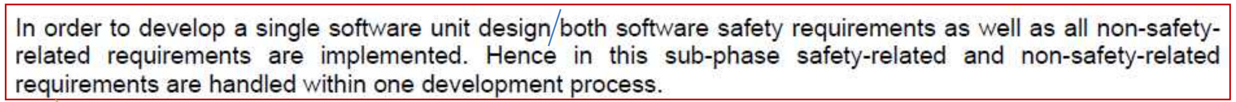
6-8 Software Unit Design and Implementation



8.2 General

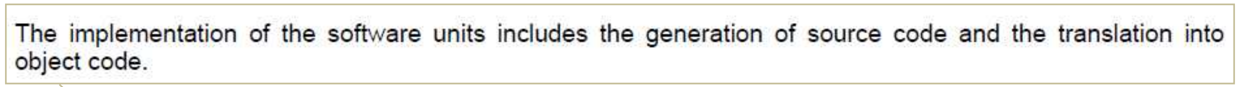
Based on the software architectural design, the detailed design of the software units is developed. The detailed design will be implemented as a model or directly as source code, in accordance with the modelling or coding guidelines respectively. The detailed design and the implementation are statically verified before proceeding to the software unit testing phase. The implementation-related properties are achievable at the source code level if manual code development is used. If model-based development with automatic code generation is used, these properties apply to the model and need not apply to the source code.

Software Architectural Design에 따라 **Software Unit**을 **상세설계**하고 **구현**한다.
+ 모델 기반 방법론일 경우, 최종 모델을 생성하는 것을 구현단계로 간주
+ **정적검증(Static Verification)**을 수행한다.



In order to develop a single software unit design/both software safety requirements as well as all non-safety-related requirements are implemented. Hence in this sub-phase safety-related and non-safety-related requirements are handled within one development process.

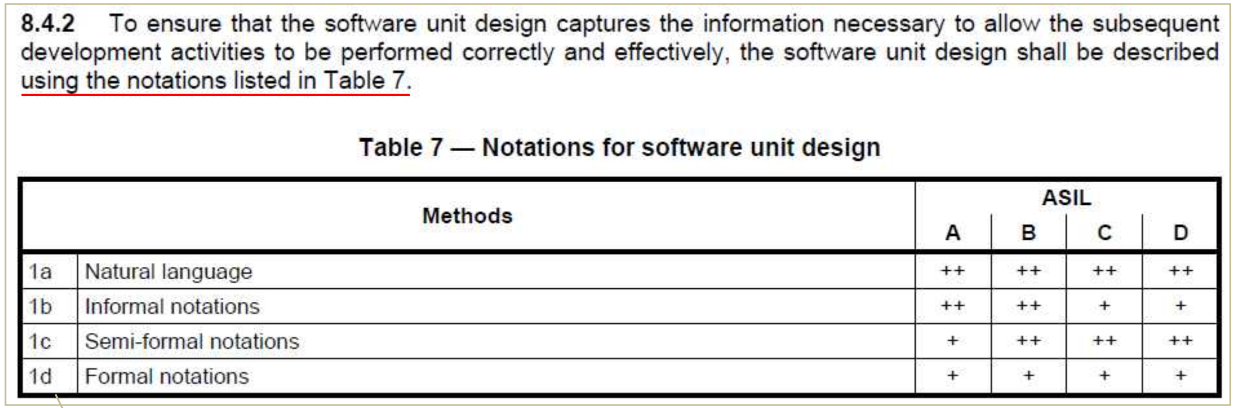
Safety / Non-Safety Requirements를 모두 함께 고려한다.



The implementation of the software units includes the generation of source code and the translation into object code.

Software Unit 구현은 **소스코드 생성**과 **오브젝트 코드 변환**을 포함한다.

6-8 Software Unit Design and Implementation



8.4.2 To ensure that the software unit design captures the information necessary to allow the subsequent development activities to be performed correctly and effectively, the software unit design shall be described using the notations listed in Table 7.

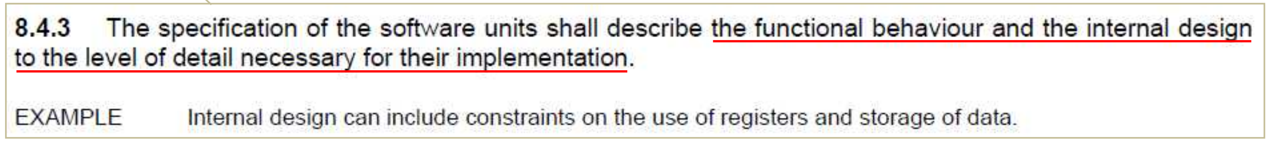
Table 7 — Notations for software unit design

	Methods	**ASIL**			
		A	**B**	**C**	**D**
1a	Natural language	++	++	++	++
1b	Informal notations	++	++	+	+
1c	Semi-formal notations	+	++	++	++
1d	Formal notations	+	+	+	+

Software Unit의 상세설계는 **표기법(Notations)**을 사용해서 잘 명세해야 한다. **(Table 7)**

+ 기능(Functional Behavior)
+ 내부 설계

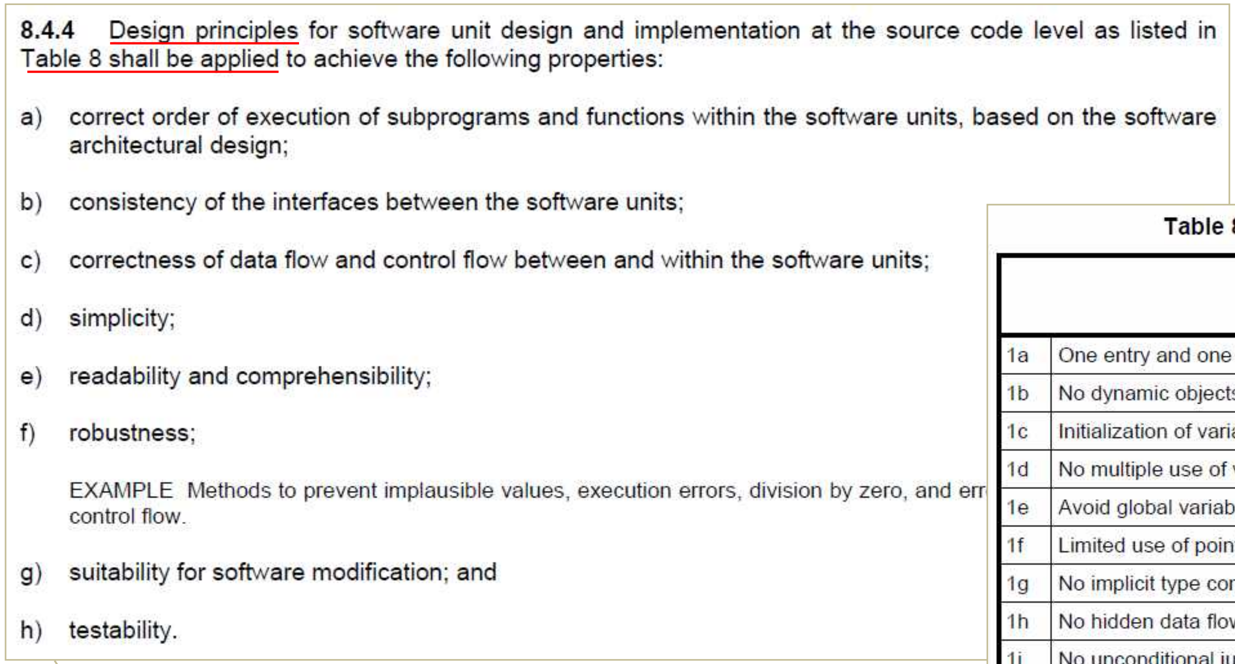
→ **보고 바로 구현할 수 있을 정도로 자세하게 작성한다.**



8.4.3 The specification of the software units shall describe the functional behaviour and the internal design to the level of detail necessary for their implementation.

EXAMPLE Internal design can include constraints on the use of registers and storage of data.

6-8 Software Unit Design and Implementation



8.4.4 Design principles for software unit design and implementation at the source code level as listed in Table 8 shall be applied to achieve the following properties:

a) correct order of execution of subprograms and functions within the software units, based on the software architectural design;

b) consistency of the interfaces between the software units;

c) correctness of data flow and control flow between and within the software units;

d) simplicity;

e) readability and comprehensibility;

f) robustness;

EXAMPLE Methods to prevent implausible values, execution errors, division by zero, and err
control flow.

g) suitability for software modification; and

h) testability.

설계원칙(Design Principles)을 적용해야 한다.
+ **Table 8**
+ **상세설계**와 **구현된 소스코드**에 모두 적용

원칙은 나중에 검증(Verification)해야 한다.

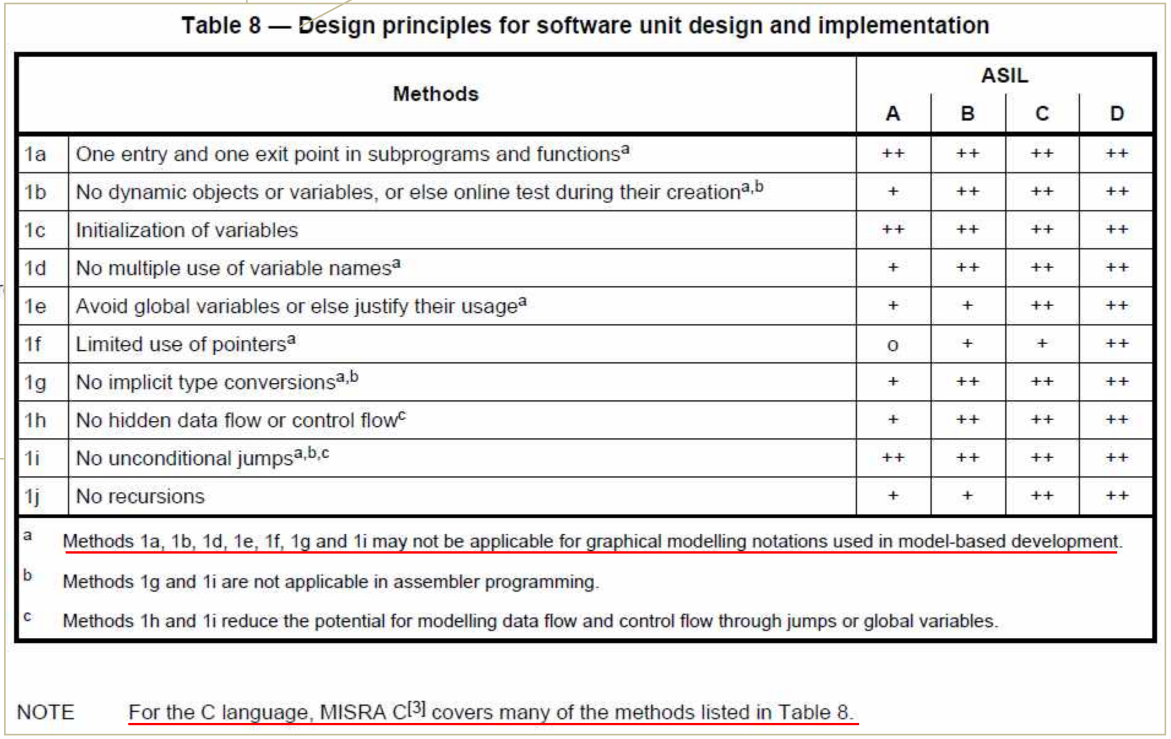


Table 8 — Design principles for software unit design and implementation

	Methods	ASIL A	ASIL B	ASIL C	ASIL D
1a	One entry and one exit point in subprograms and functions[a]	++	++	++	++
1b	No dynamic objects or variables, or else online test during their creation[a,b]	+	++	++	++
1c	Initialization of variables	++	++	++	++
1d	No multiple use of variable names[a]	+	++	++	++
1e	Avoid global variables or else justify their usage[a]	+	+	++	++
1f	Limited use of pointers[a]	o	+	+	++
1g	No implicit type conversions[a,b]	+	++	++	++
1h	No hidden data flow or control flow[c]	+	++	++	++
1i	No unconditional jumps[a,b,c]	++	++	++	++
1j	No recursions	+	+	++	++

[a] Methods 1a, 1b, 1d, 1e, 1f, 1g and 1i may not be applicable for graphical modelling notations used in model-based development.

[b] Methods 1g and 1i are not applicable in assembler programming.

[c] Methods 1h and 1i reduce the potential for modelling data flow and control flow through jumps or global variables.

NOTE For the C language, MISRA C[3] covers many of the methods listed in Table 8.

+ MISRA-C를 적용하면 Table 8의 많은 부분을 만족시킬 수 있다.
+ 1a, 1b, 1d, 1e, 1f, 1g, 1i는 모델 기반 방법론에는 적용되지 않는다.

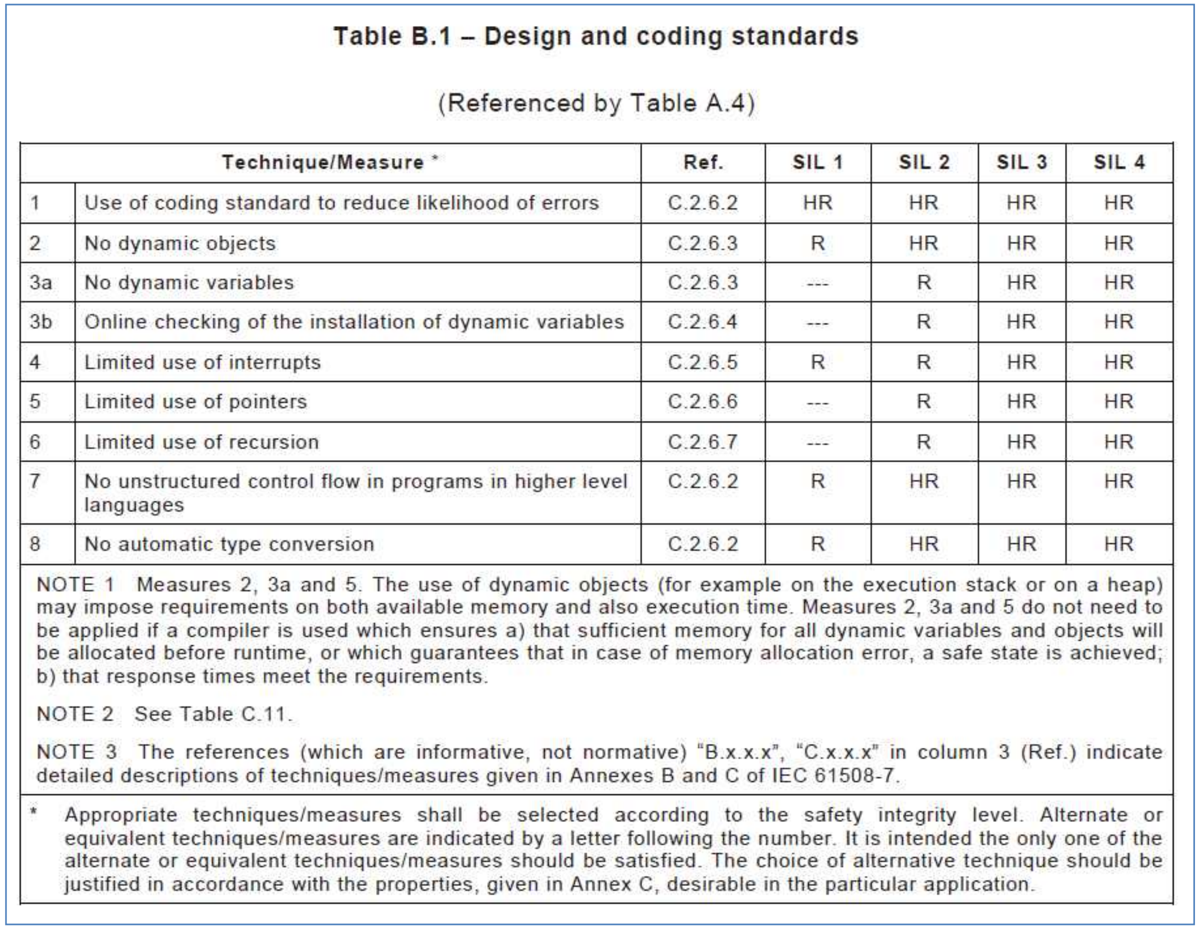


Table B.1 – Design and coding standards

(Referenced by Table A.4)

Technique/Measure *		Ref.	SIL 1	SIL 2	SIL 3	SIL 4
1	Use of coding standard to reduce likelihood of errors	C.2.6.2	HR	HR	HR	HR
2	No dynamic objects	C.2.6.3	R	HR	HR	HR
3a	No dynamic variables	C.2.6.3	---	R	HR	HR
3b	Online checking of the installation of dynamic variables	C.2.6.4	---	R	HR	HR
4	Limited use of interrupts	C.2.6.5	R	R	HR	HR
5	Limited use of pointers	C.2.6.6	---	R	HR	HR
6	Limited use of recursion	C.2.6.7	---	R	HR	HR
7	No unstructured control flow in programs in higher level languages	C.2.6.2	R	HR	HR	HR
8	No automatic type conversion	C.2.6.2	R	HR	HR	HR

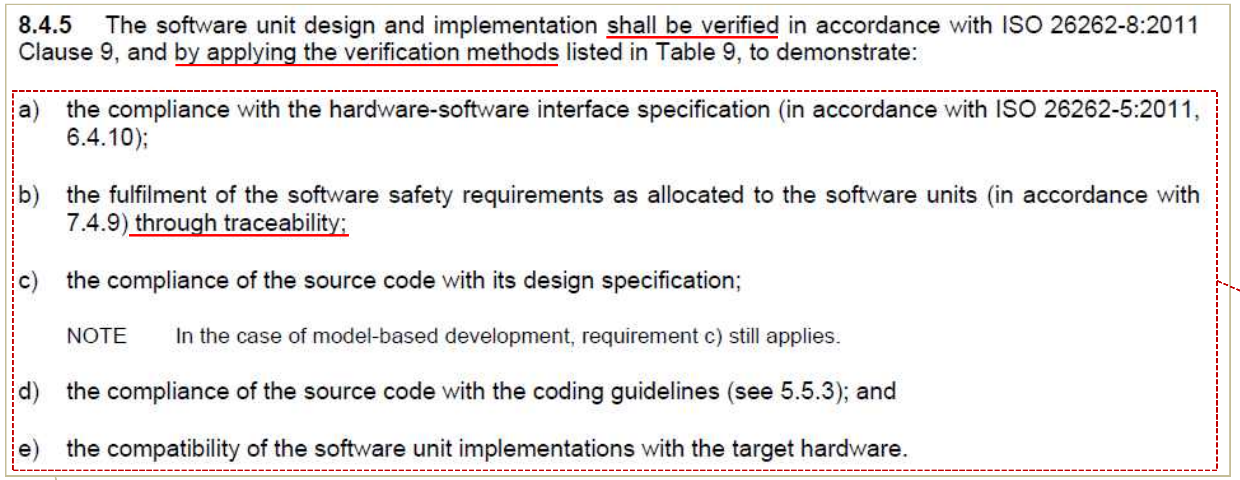
NOTE 1 Measures 2, 3a and 5. The use of dynamic objects (for example on the execution stack or on a heap) may impose requirements on both available memory and also execution time. Measures 2, 3a and 5 do not need to be applied if a compiler is used which ensures a) that sufficient memory for all dynamic variables and objects will be allocated before runtime, or which guarantees that in case of memory allocation error, a safe state is achieved; b) that response times meet the requirements.

NOTE 2 See Table C.11.

NOTE 3 The references (which are informative, not normative) "B.x.x.x", "C.x.x.x" in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7.

* Appropriate techniques/measures shall be selected according to the safety integrity level. Alternate or equivalent techniques/measures are indicated by a letter following the number. It is intended the only one of the alternate or equivalent techniques/measures should be satisfied. The choice of alternative technique should be justified in accordance with the properties, given in Annex C, desirable in the particular application.

6-8 Software Unit Design and Implementation



8.4.5 The software unit design and implementation shall be verified in accordance with ISO 26262-8:2011 Clause 9, and by applying the verification methods listed in Table 9, to demonstrate:

a) the compliance with the hardware-software interface specification (in accordance with ISO 26262-5:2011, 6.4.10);

b) the fulfilment of the software safety requirements as allocated to the software units (in accordance with 7.4.9) through traceability;

c) the compliance of the source code with its design specification;

NOTE In the case of model-based development, requirement c) still applies.

d) the compliance of the source code with the coding guidelines (see 5.5.3); and

e) the compatibility of the software unit implementations with the target hardware.

검증할 내용

Software Unit Design과 **Implementation**에 대한 **검증(Verification)**을 수행해야 한다.
+ **Table 9**의 방법들을 사용

검증 내용
+ **Hardware-Software Interface Specification의 준수 여부**
+ **할당된 Software Safety Requirements를 모두 구현하였는지 여부**
+ **소스 코드의 Design Specification 준수 여부**
+ **소스 코드의 Coding Guideline 준수 여부**
+ **구현과 Target Hardware와의 호환성 여부**

→ 어떤 방법을 어떻게 사용해서 어느 내용을 효과적으로 검증할 것인지 결정해야 한다.

DEPENDABLE SOFTWARE LABORATORY

6-8 Software Unit Design and Implementation

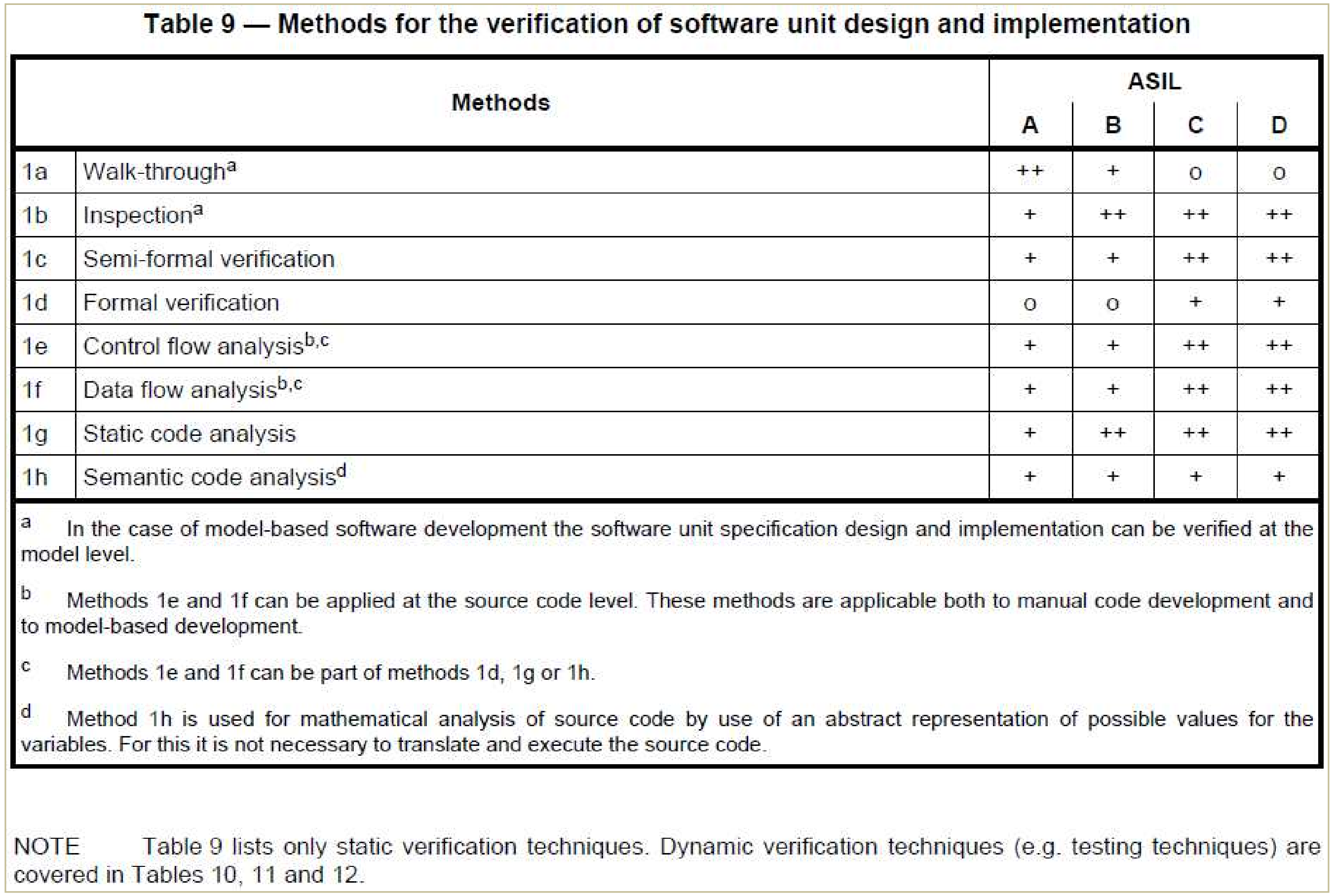


Table 9 — Methods for the verification of software unit design and implementation

	Methods	ASIL			
		A	B	C	D
1a	Walk-through[a]	++	+	o	o
1b	Inspection[a]	+	++	++	++
1c	Semi-formal verification	+	+	++	++
1d	Formal verification	o	o	+	+
1e	Control flow analysis[b,c]	+	+	++	++
1f	Data flow analysis[b,c]	+	+	++	++
1g	Static code analysis	+	++	++	++
1h	Semantic code analysis[d]	+	+	+	+

[a] In the case of model-based software development the software unit specification design and implementation can be verified at the model level.

[b] Methods 1e and 1f can be applied at the source code level. These methods are applicable both to manual code development and to model-based development.

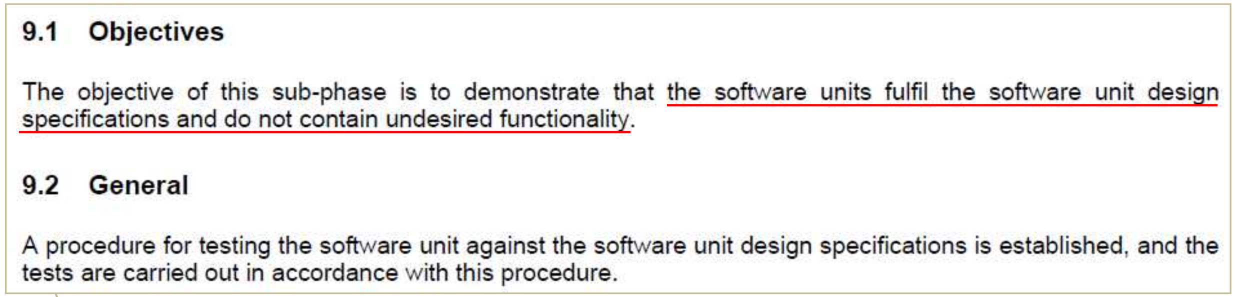
[c] Methods 1e and 1f can be part of methods 1d, 1g or 1h.

[d] Method 1h is used for mathematical analysis of source code by use of an abstract representation of possible values for the variables. For this it is not necessary to translate and execute the source code.

NOTE Table 9 lists only static verification techniques. Dynamic verification techniques (e.g. testing techniques) are covered in Tables 10, 11 and 12.

6-9 Software Unit Testing

■ 6-9 Software Unit Testing



9.1 Objectives

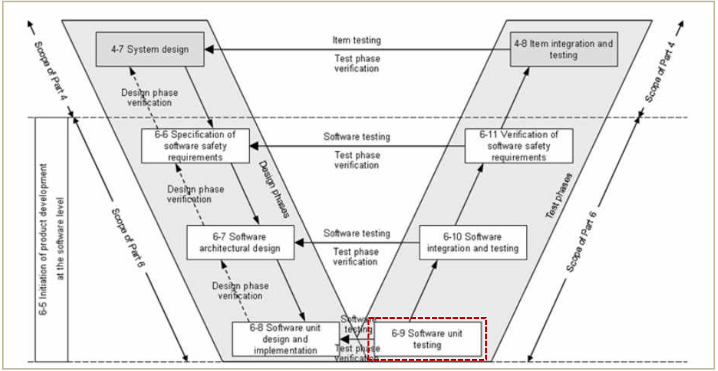
The objective of this sub-phase is to demonstrate that the software units fulfil the software unit design specifications and do not contain undesired functionality.

9.2 General

A procedure for testing the software unit against the software unit design specifications is established, and the tests are carried out in accordance with this procedure.

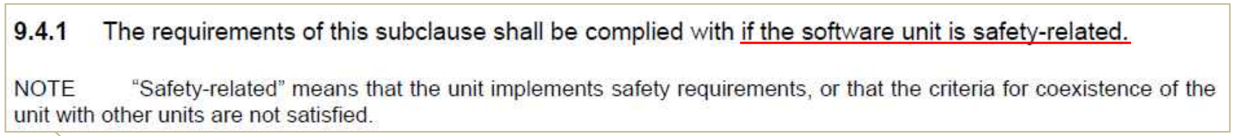
Software Unit에 대한 **테스팅(Software Unit Testing)**을 수행한다.
- **Software Unit**이 **Software Unit Design Specification**을 만족함을 보인다.
- **Software Unit**이 **원치 않는 기능(Undesired Functionalities)**을 수행하지 않음을 보인다.
  → 일반적인 소프트웨어 테스팅의 목적 X

Software Unit Testing을 위한 **절차(Procedure)**를 수립하고,
이에 따라 테스팅을 **수행**한다.



4-7 System design
Item testing
Test phase verification
4-8 Item integration and testing
Design phase verification
6-6 Specification of software safety requirements
Software testing
Test phase verification
6-11 Verification of software safety requirements
Design phase verification
Design phases
Test phases
6-7 Software architectural design
Software testing
Test phase verification
6-10 Software integration and testing
Design phase verification
6-8 Software unit design and implementation
Software testing
Test phase verification
6-9 Software unit testing
Scope of Part 4
Scope of Part 6
6-5 Initiation of product development at the software level

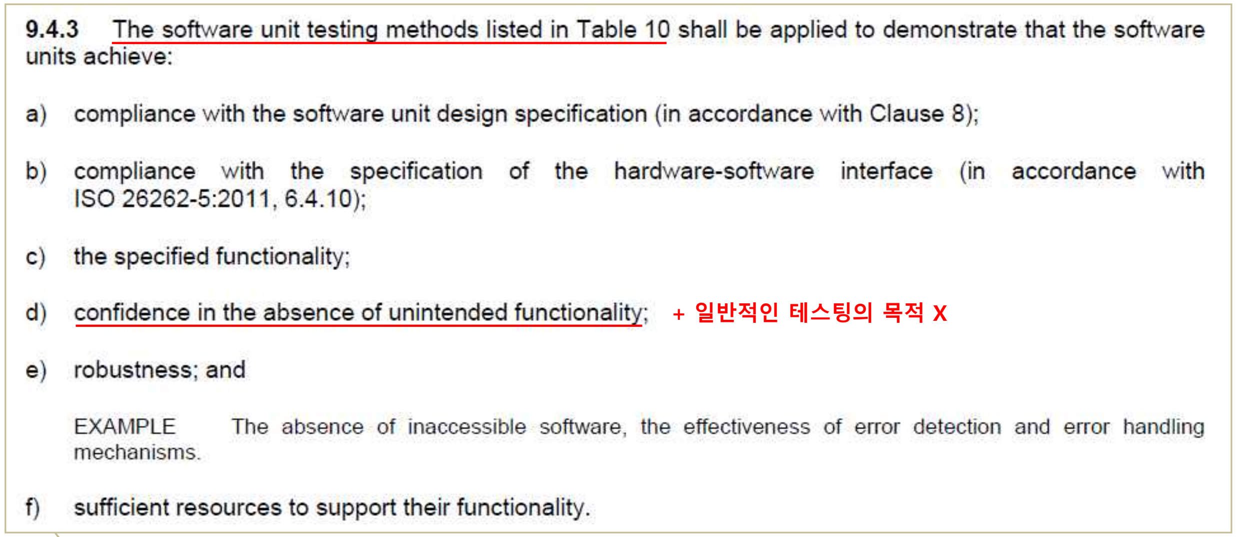
6-9 Software Unit Testing



9.4.1 The requirements of this subclause shall be complied with if the software unit is safety-related.

NOTE "Safety-related" means that the unit implements safety requirements, or that the criteria for coexistence of the unit with other units are not satisfied.

Safety-Related Functions을 구현한 **Software Unit**에 대한 **테스팅**을 수행한다.



9.4.3 The software unit testing methods listed in Table 10 shall be applied to demonstrate that the software units achieve:

a) compliance with the software unit design specification (in accordance with Clause 8);

b) compliance with the specification of the hardware-software interface (in accordance with ISO 26262-5:2011, 6.4.10);

c) the specified functionality;

d) confidence in the absence of unintended functionality; **+ 일반적인 테스팅의 목적 X**

e) robustness; and

EXAMPLE The absence of inaccessible software, the effectiveness of error detection and error handling mechanisms.

f) sufficient resources to support their functionality.

Table 10의 **테스팅 방법론(Testing Methods)**을 사용하여 Software Unit Testing을 수행한다.
+ **테스팅 목적(a~f)**에 따라 사용하는 테스팅 방법론 상이하다.

6-9 Software Unit Testing

일반적으로 수행하는 Unit Testing 방법들

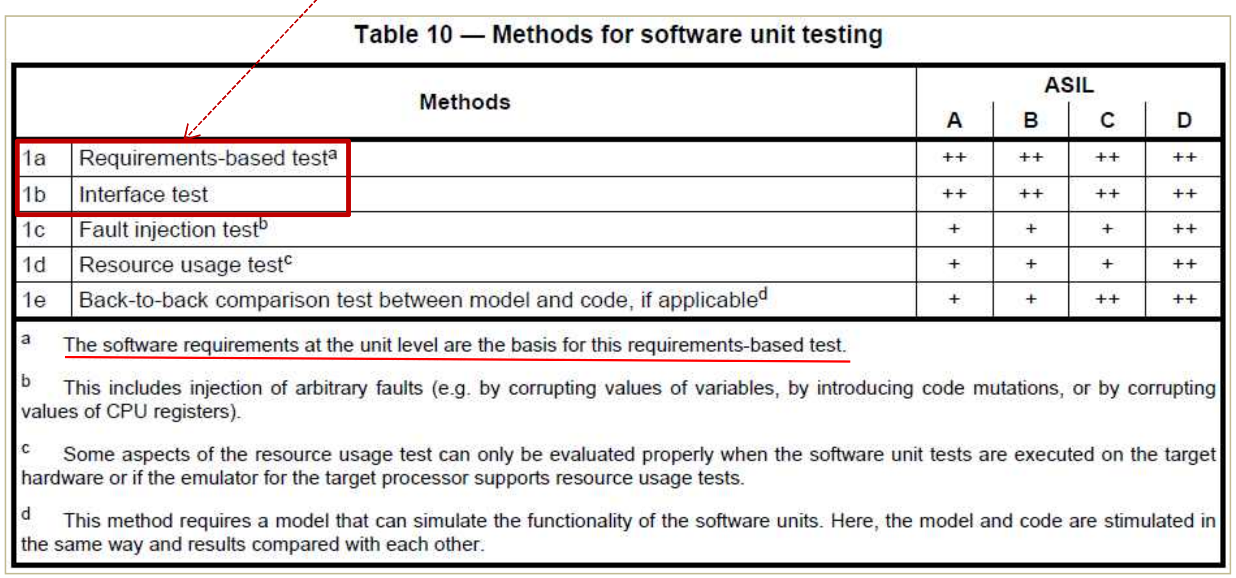


Table 10 — Methods for software unit testing

	Methods	ASIL A	ASIL B	ASIL C	ASIL D
1a	Requirements-based test[a]	++	++	++	++
1b	Interface test	++	++	++	++
1c	Fault injection test[b]	+	+	+	++
1d	Resource usage test[c]	+	+	+	++
1e	Back-to-back comparison test between model and code, if applicable[d]	+	+	++	++

[a] The software requirements at the unit level are the basis for this requirements-based test.

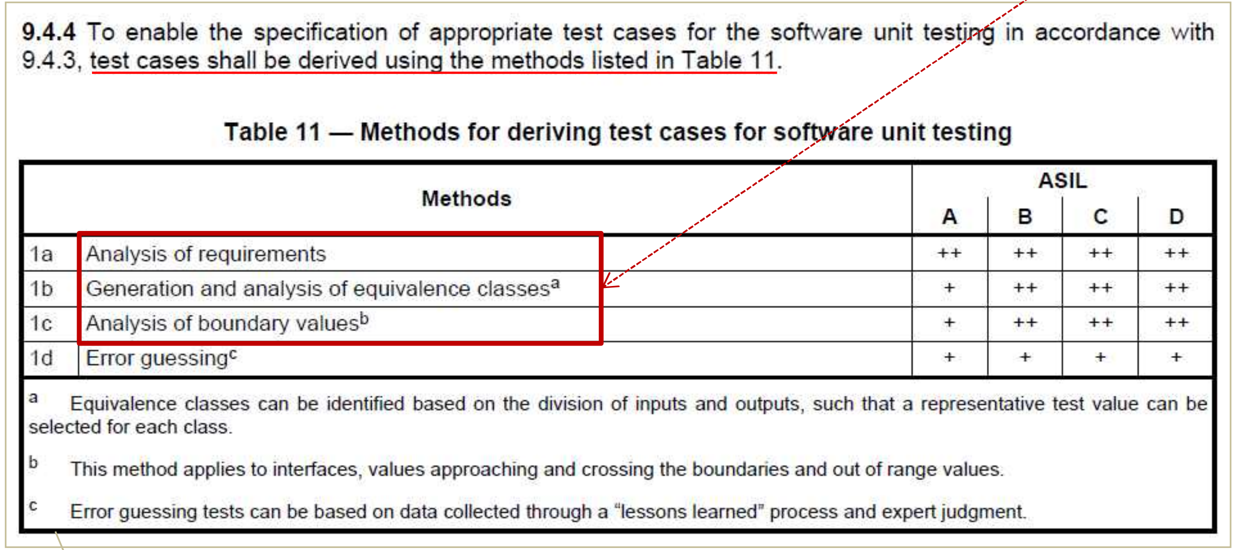
[b] This includes injection of arbitrary faults (e.g. by corrupting values of variables, by introducing code mutations, or by corrupting values of CPU registers).

[c] Some aspects of the resource usage test can only be evaluated properly when the software unit tests are executed on the target hardware or if the emulator for the target processor supports resource usage tests.

[d] This method requires a model that can simulate the functionality of the software units. Here, the model and code are stimulated in the same way and results compared with each other.

6-9 Software Unit Testing

일반적인 Functional Test Cases 생성 방법



9.4.4 To enable the specification of appropriate test cases for the software unit testing in accordance with 9.4.3, test cases shall be derived using the methods listed in Table 11.

Table 11 — Methods for deriving test cases for software unit testing

	Methods	ASIL A	ASIL B	ASIL C	ASIL D
1a	Analysis of requirements	++	++	++	++
1b	Generation and analysis of equivalence classes[a]	+	++	++	++
1c	Analysis of boundary values[b]	+	++	++	++
1d	Error guessing[c]	+	+	+	+

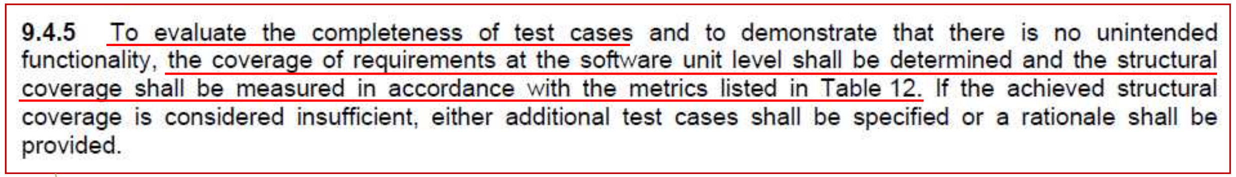
[a] Equivalence classes can be identified based on the division of inputs and outputs, such that a representative test value can be selected for each class.

[b] This method applies to interfaces, values approaching and crossing the boundaries and out of range values.

[c] Error guessing tests can be based on data collected through a "lessons learned" process and expert judgment.

Software Unit Testing의 **Test Cases**를 **개발(Derivation)**하기 위한 방법론 **(Table 11)**
+ Software Unit Design Specification을 잘 분석한다.
+ 동일 클래스(Equivalence Classes) 개념을 이용해서 분석한다.
+ 경계값(Boundary Values) 분석을 통해 개발한다.
+ 자주 발생했던 오류들을 테스트 케이스로 활용한다.

6-9 Software Unit Testing


9.4.5 To evaluate the completeness of test cases and to demonstrate that there is no unintended functionality, the coverage of requirements at the software unit level shall be determined and the structural coverage shall be measured in accordance with the metrics listed in Table 12. If the achieved structural coverage is considered insufficient, either additional test cases shall be specified or a rationale shall be provided.

Unit Test Cases의 **Software Unit Design Specification**에 대한 **Requirements Coverage**를 **측정**
+ UTC가 SUDS를 얼마나 커버하는지 계산한다. (100%: 전체를 테스트함, 50%: SSRS의 항목 중 50%만 테스트 수행)

→ 개발된 UTC의 **성능(Completeness)**을 평가하는 **절대적인 지표 O** (항상 100% 要)

Unit Test Cases의 **Structural Coverage**를 측정
+ 추가적으로, **UTC**가 **Unit Code**를 **얼마나 실행**하는지 확인한다.
+ **Structural Coverage Criteria** 사용 **(Table 12)**
  + Statement Coverage , Branch Coverage , MC/DC 등

+ **Testing Tools**을 이용해서 **측정** 및 **생성** 가능

→ 개발된 UTC의 성능을 평가하는 **절대적인 지표 X**
  + Requirements Analysis(즉, Functional Test)를 통해 개발된 UTC를 보완 가능

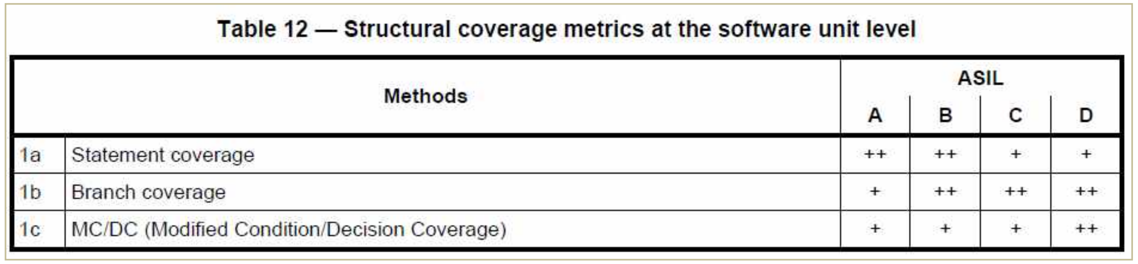


Table 12 — Structural coverage metrics at the software unit level

Methods		ASIL			
		A	B	C	D
1a	Statement coverage	++	++	+	+
1b	Branch coverage	+	++	++	++
1c	MC/DC (Modified Condition/Decision Coverage)	+	+	+	++

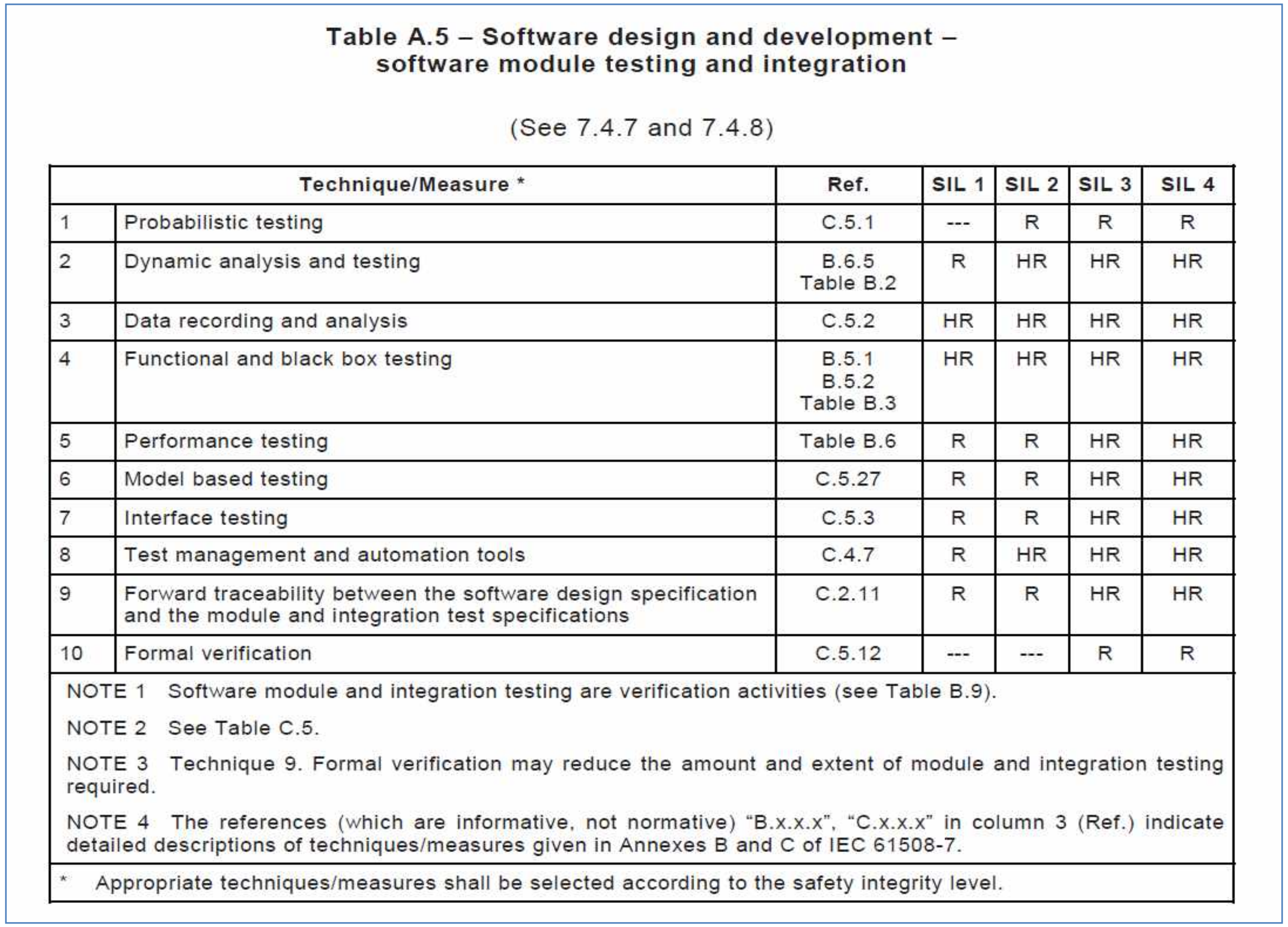


Table A.5 – Software design and development – software module testing and integration

(See 7.4.7 and 7.4.8)

Technique/Measure *		Ref.	SIL 1	SIL 2	SIL 3	SIL 4
1	Probabilistic testing	C.5.1	---	R	R	R
2	Dynamic analysis and testing	B.6.5 Table B.2	R	HR	HR	HR
3	Data recording and analysis	C.5.2	HR	HR	HR	HR
4	Functional and black box testing	B.5.1 B.5.2 Table B.3	HR	HR	HR	HR
5	Performance testing	Table B.6	R	R	HR	HR
6	Model based testing	C.5.27	R	R	HR	HR
7	Interface testing	C.5.3	R	R	HR	HR
8	Test management and automation tools	C.4.7	R	HR	HR	HR
9	Forward traceability between the software design specification and the module and integration test specifications	C.2.11	R	R	HR	HR
10	Formal verification	C.5.12	---	---	R	R

NOTE 1 Software module and integration testing are verification activities (see Table B.9).

NOTE 2 See Table C.5.

NOTE 3 Technique 9. Formal verification may reduce the amount and extent of module and integration testing required.

NOTE 4 The references (which are informative, not normative) "B.x.x.x", "C.x.x.x" in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7.

* Appropriate techniques/measures shall be selected according to the safety integrity level.

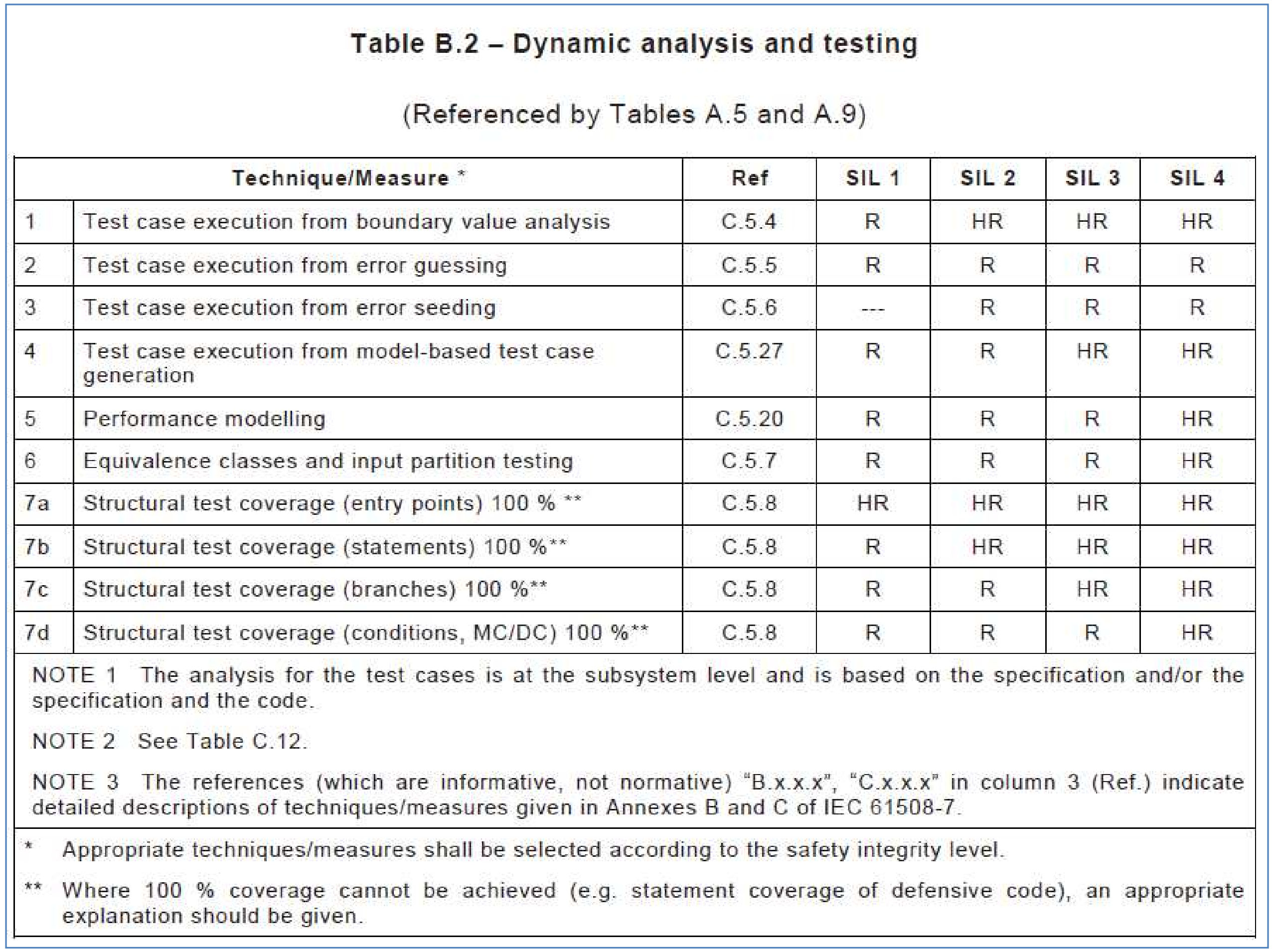


Table B.2 – Dynamic analysis and testing

(Referenced by Tables A.5 and A.9)

Technique/Measure *		Ref	SIL 1	SIL 2	SIL 3	SIL 4
1	Test case execution from boundary value analysis	C.5.4	R	HR	HR	HR
2	Test case execution from error guessing	C.5.5	R	R	R	R
3	Test case execution from error seeding	C.5.6	---	R	R	R
4	Test case execution from model-based test case generation	C.5.27	R	R	HR	HR
5	Performance modelling	C.5.20	R	R	R	HR
6	Equivalence classes and input partition testing	C.5.7	R	R	R	HR
7a	Structural test coverage (entry points) 100 % **	C.5.8	HR	HR	HR	HR
7b	Structural test coverage (statements) 100 %**	C.5.8	R	HR	HR	HR
7c	Structural test coverage (branches) 100 %**	C.5.8	R	R	HR	HR
7d	Structural test coverage (conditions, MC/DC) 100 %**	C.5.8	R	R	R	HR

NOTE 1 The analysis for the test cases is at the subsystem level and is based on the specification and/or the specification and the code.

NOTE 2 See Table C.12.

NOTE 3 The references (which are informative, not normative) "B.x.x.x", "C.x.x.x" in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7.

* Appropriate techniques/measures shall be selected according to the safety integrity level.

** Where 100 % coverage cannot be achieved (e.g. statement coverage of defensive code), an appropriate explanation should be given.

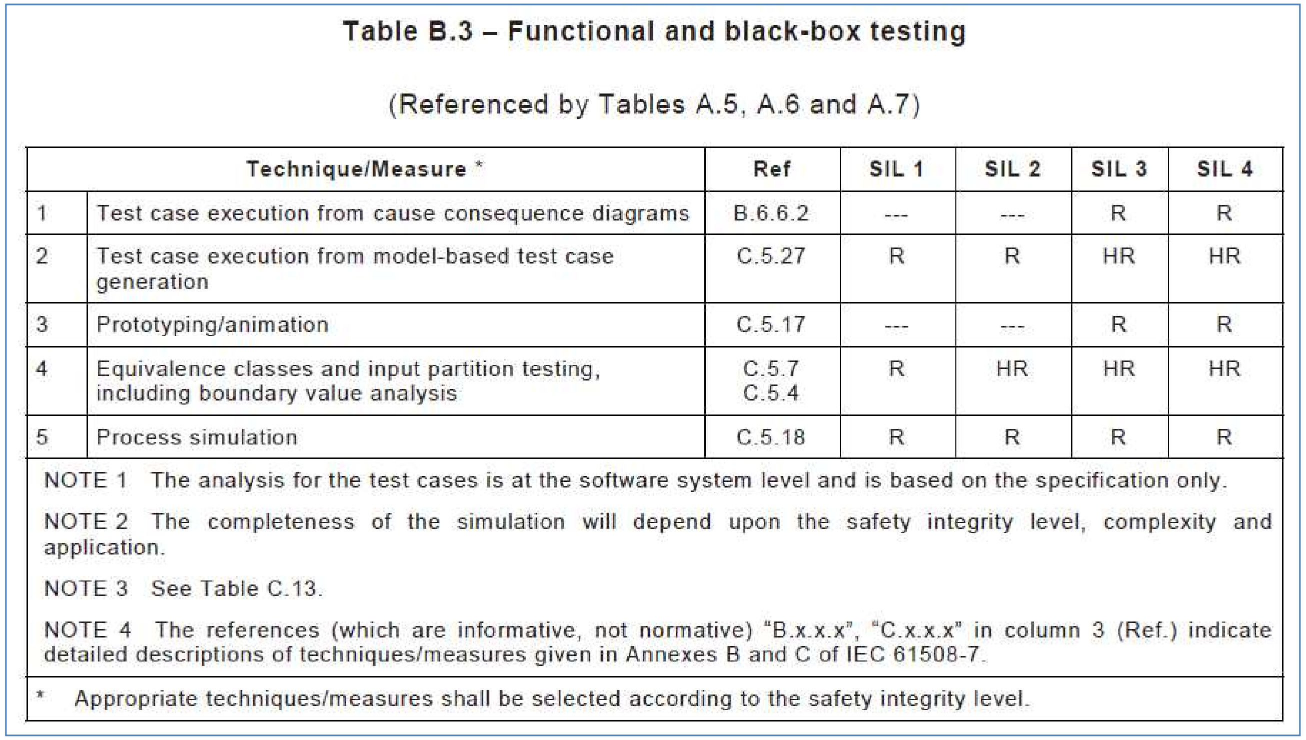


Table B.3 – Functional and black-box testing

(Referenced by Tables A.5, A.6 and A.7)

	Technique/Measure *	Ref	SIL 1	SIL 2	SIL 3	SIL 4
1	Test case execution from cause consequence diagrams	B.6.6.2	---	---	R	R
2	Test case execution from model-based test case generation	C.5.27	R	R	HR	HR
3	Prototyping/animation	C.5.17	---	---	R	R
4	Equivalence classes and input partition testing, including boundary value analysis	C.5.7 C.5.4	R	HR	HR	HR
5	Process simulation	C.5.18	R	R	R	R

NOTE 1 The analysis for the test cases is at the software system level and is based on the specification only.

NOTE 2 The completeness of the simulation will depend upon the safety integrity level, complexity and application.

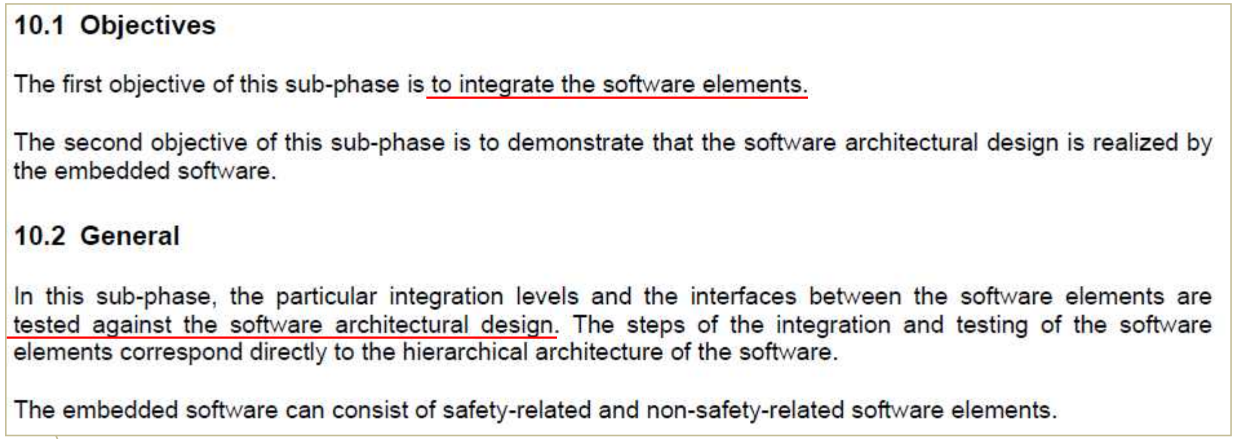
NOTE 3 See Table C.13.

NOTE 4 The references (which are informative, not normative) "B.x.x.x", "C.x.x.x" in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7.

* Appropriate techniques/measures shall be selected according to the safety integrity level.

6-10 Software Integration and Testing

■ 6-10 Software Integration and Testing



10.1 Objectives

The first objective of this sub-phase is to integrate the software elements.

The second objective of this sub-phase is to demonstrate that the software architectural design is realized by the embedded software.

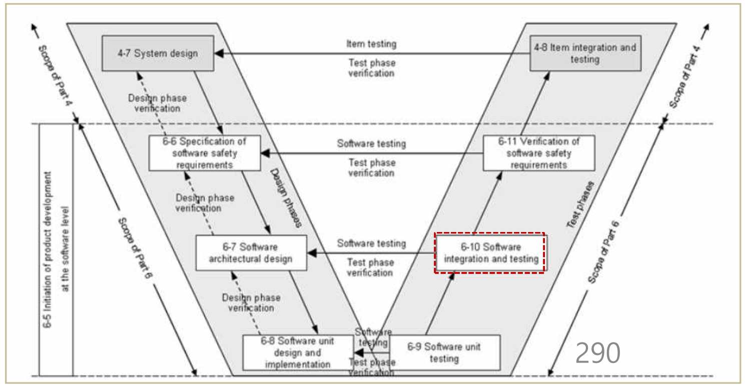
10.2 General

In this sub-phase, the particular integration levels and the interfaces between the software elements are tested against the software architectural design. The steps of the integration and testing of the software elements correspond directly to the hierarchical architecture of the software.

The embedded software can consist of safety-related and non-safety-related software elements.

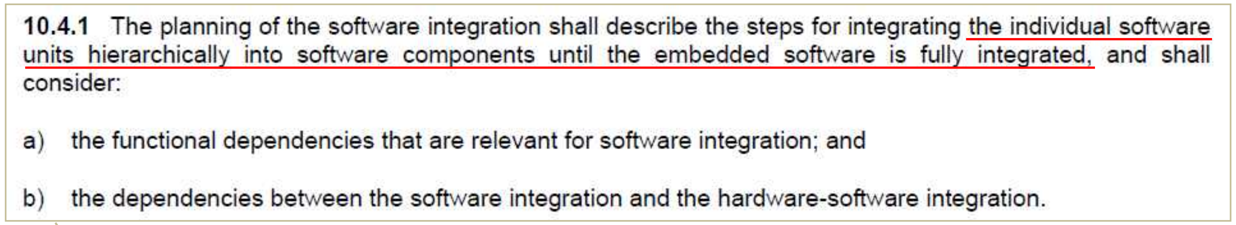
Software Elements에 대한 **통합(Integration)**을 수행한다.
+ Non-Safety-Related Software Elements도 포함할 수 있다.

Software Integration(+Interfaces)에 대한 **테스팅(Integration Testing)**을 계획하고 수행한다.
+ Software Integration이 Software Architecture Design을 잘 구현했는지 확인한다.



Scope of Part 4
4-7 System design
Item testing
Test phase verification
4-8 Item integration and testing
Scope of Part 4
Design phase verification
6-6 Specification of software safety requirements
Software testing
Test phase verification
6-11 Verification of software safety requirements
Design phases
Test phases
Design phase verification
6-7 Software architectural design
Software testing
Test phase verification
6-10 Software integration and testing
Design phase verification
6-8 Software unit design and implementation
Software testing
Test phase verification
6-9 Software unit testing
6-5 Initiation of product development at the software level
Scope of Part 6
Scope of Part 6

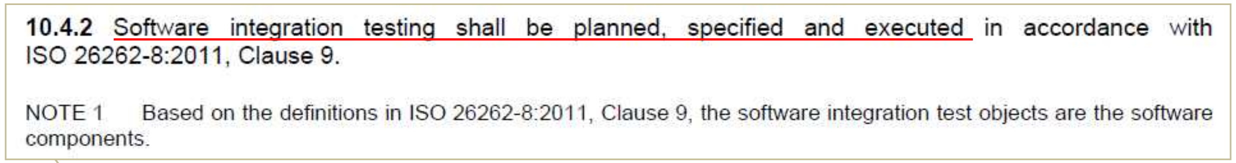
6-10 Software Integration and Testing


10.4.1 The planning of the software integration shall describe the steps for integrating the individual software units hierarchically into software components until the embedded software is fully integrated, and shall consider:

a) the functional dependencies that are relevant for software integration; and

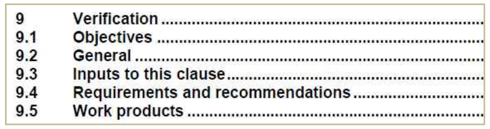
b) the dependencies between the software integration and the hardware-software integration.

전체 Embedded Software가 완성될 때까지, **Software Units**을 **계층적(Hierarchically)**으로 **통합(Integration)**한다.
+ 기능적 의존관계
+ Hardware-Software Integration 과의 의존관계 고려 要


10.4.2 Software integration testing shall be planned, specified and executed in accordance with ISO 26262-8:2011, Clause 9.

NOTE 1 Based on the definitions in ISO 26262-8:2011, Clause 9, the software integration test objects are the software components.

Software Integration Testing은 **26262-8 Clause 9**에 따라 계획되어야 한다.
+ SIT의 대상 : **Software Components**



(26262-8 Clause 9)

6-10 Software Integration and Testing

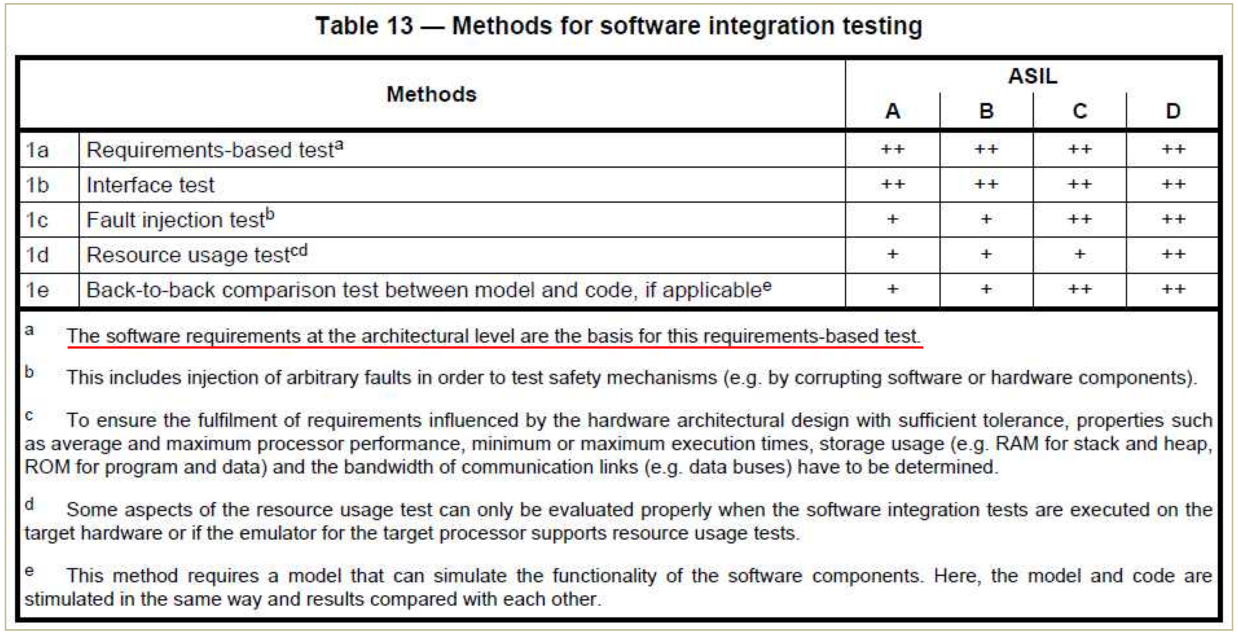


Table 13 — Methods for software integration testing

Methods		ASIL			
		A	B	C	D
1a	Requirements-based test[a]	++	++	++	++
1b	Interface test	++	++	++	++
1c	Fault injection test[b]	+	+	++	++
1d	Resource usage test[cd]	+	+	+	++
1e	Back-to-back comparison test between model and code, if applicable[e]	+	+	++	++

[a] The software requirements at the architectural level are the basis for this requirements-based test.

[b] This includes injection of arbitrary faults in order to test safety mechanisms (e.g. by corrupting software or hardware components).

[c] To ensure the fulfilment of requirements influenced by the hardware architectural design with sufficient tolerance, properties such as average and maximum processor performance, minimum or maximum execution times, storage usage (e.g. RAM for stack and heap, ROM for program and data) and the bandwidth of communication links (e.g. data buses) have to be determined.

[d] Some aspects of the resource usage test can only be evaluated properly when the software integration tests are executed on the target hardware or if the emulator for the target processor supports resource usage tests.

[e] This method requires a model that can simulate the functionality of the software components. Here, the model and code are stimulated in the same way and results compared with each other.

6-10 Software Integration and Testing

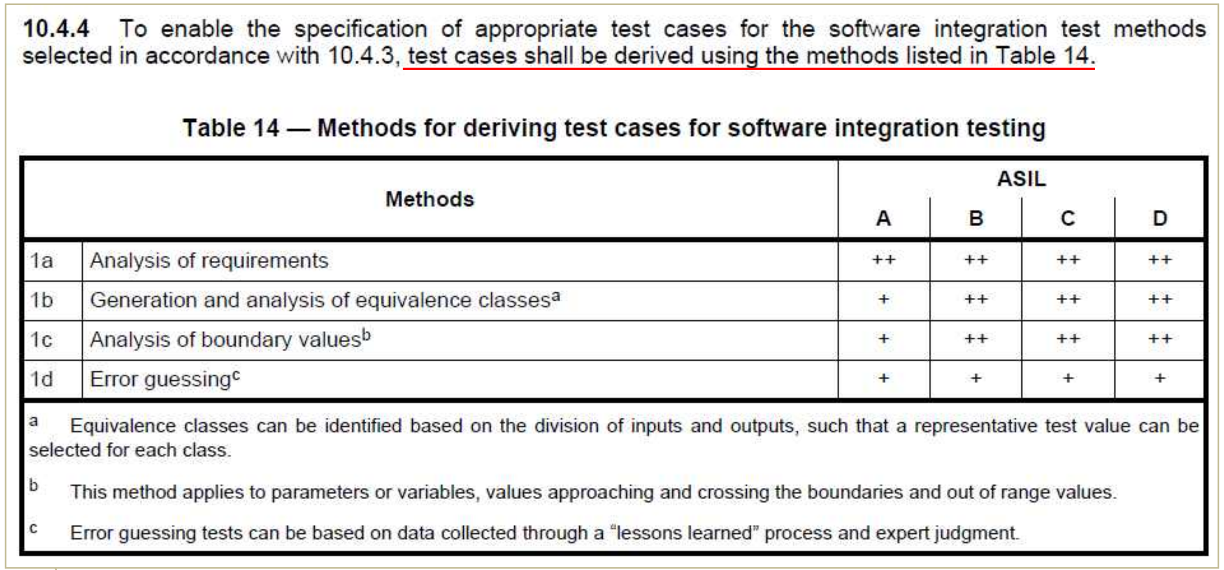

10.4.4 To enable the specification of appropriate test cases for the software integration test methods selected in accordance with 10.4.3, test cases shall be derived using the methods listed in Table 14.

Table 14 — Methods for deriving test cases for software integration testing

Methods		ASIL A	ASIL B	ASIL C	ASIL D
1a	Analysis of requirements	++	++	++	++
1b	Generation and analysis of equivalence classes[a]	+	++	++	++
1c	Analysis of boundary values[b]	+	++	++	++
1d	Error guessing[c]	+	+	+	+

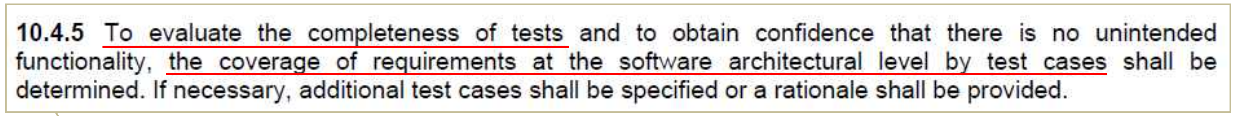
[a] Equivalence classes can be identified based on the division of inputs and outputs, such that a representative test value can be selected for each class.

[b] This method applies to parameters or variables, values approaching and crossing the boundaries and out of range values.

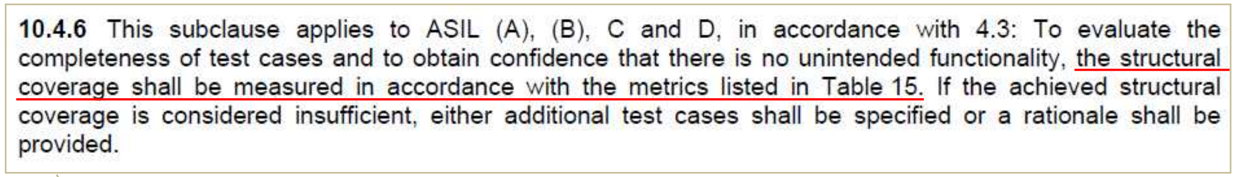
[c] Error guessing tests can be based on data collected through a "lessons learned" process and expert judgment.

Software Integration Testing의 **Test Cases**를 **개발(Derivation)**하기 위한 방법론 **(Table 11)**
+ Software Architectural Design을 잘 분석한다.
+ 동일 클래스(Equivalence Classes) 개념을 이용해서 분석한다.
+ 경계값(Boundary Values) 분석을 통해 개발한다.
+ 자주 발생했던 오류들을 테스트 케이스로 활용한다.

6-10 Software Integration and Testing


10.4.5 To evaluate the completeness of tests and to obtain confidence that there is no unintended functionality, the coverage of requirements at the software architectural level by test cases shall be determined. If necessary, additional test cases shall be specified or a rationale shall be provided.

Integration Test Cases의 **Software Architectural Design**에 대한 **Requirements Coverage**를 **측정**
+ ITC가 SAD을 얼마나 커버하는지 계산한다. (100%: 전체를 테스트함, 50%: SAD의 항목 중 50%만 테스트 수행)
→ 개발된 ITC의 **성능(Completeness)**을 평가하는 **절대적인 지표**(항상 100% 要)


10.4.6 This subclause applies to ASIL (A), (B), C and D, in accordance with 4.3: To evaluate the completeness of test cases and to obtain confidence that there is no unintended functionality, the structural coverage shall be measured in accordance with the metrics listed in Table 15. If the achieved structural coverage is considered insufficient, either additional test cases shall be specified or a rationale shall be provided.

Integration Test Cases의 **Structural Coverage**를 측정함으로써, ITC의 **성능(Completeness)**을 판단한다.
+ **Structural Coverage Criteria** 사용 **(Table 15)**
  + Function Coverage , Call Coverage
+ **디버깅·정적분석 도구**를 이용해서 **측정** 가능

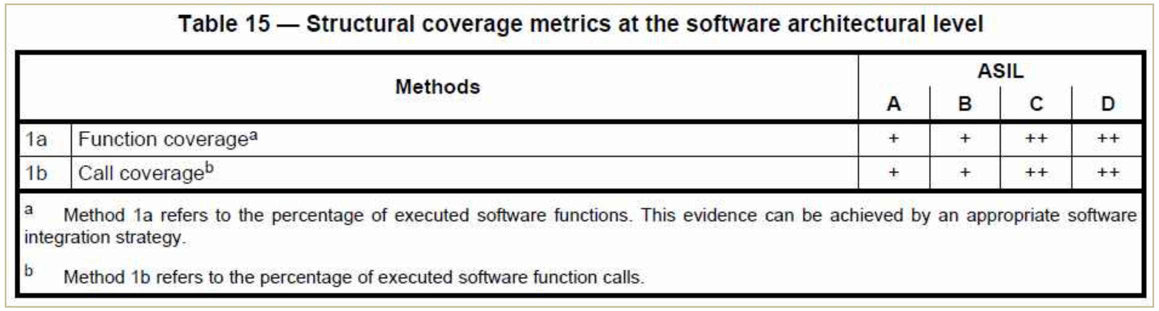


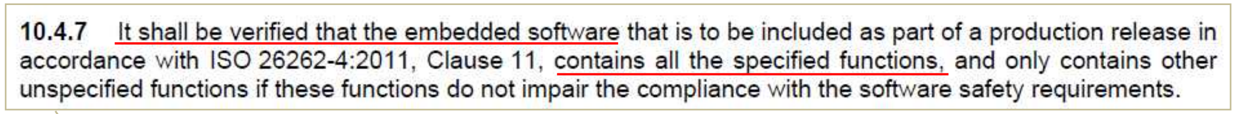
Table 15 — Structural coverage metrics at the software architectural level

Methods		ASIL			
		A	B	C	D
1a	Function coverage[a]	+	+	++	++
1b	Call coverage[b]	+	+	++	++

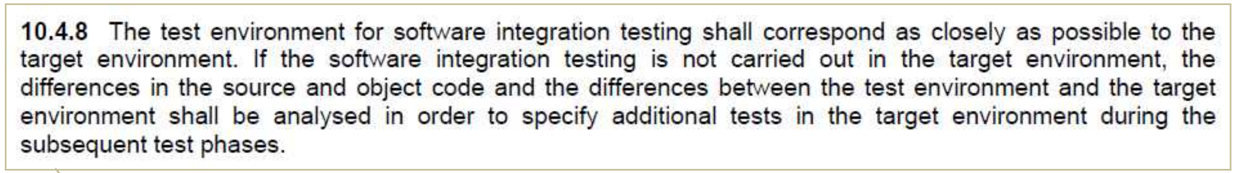
[a] Method 1a refers to the percentage of executed software functions. This evidence can be achieved by an appropriate software integration strategy.

[b] Method 1b refers to the percentage of executed software function calls.

6-10 Software Integration and Testing


10.4.7 It shall be verified that the embedded software that is to be included as part of a production release in accordance with ISO 26262-4:2011, Clause 11, contains all the specified functions, and only contains other unspecified functions if these functions do not impair the compliance with the software safety requirements.

최종 통합된 **Embedded Software**가 **Product Release에 필요한 기능**(코드)**만 포함**하는지 확인한다.
+ 디버깅이나 테스팅, 에뮬레이션 등을 위한 코드·기능은 삭제되어야 한다.
+ **필요한 기능이 잘 구현되었는지는 6-11 Verification of Software Safety Requirements에서 수행**


10.4.8 The test environment for software integration testing shall correspond as closely as possible to the target environment. If the software integration testing is not carried out in the target environment, the differences in the source and object code and the differences between the test environment and the target environment shall be analysed in order to specify additional tests in the target environment during the subsequent test phases.

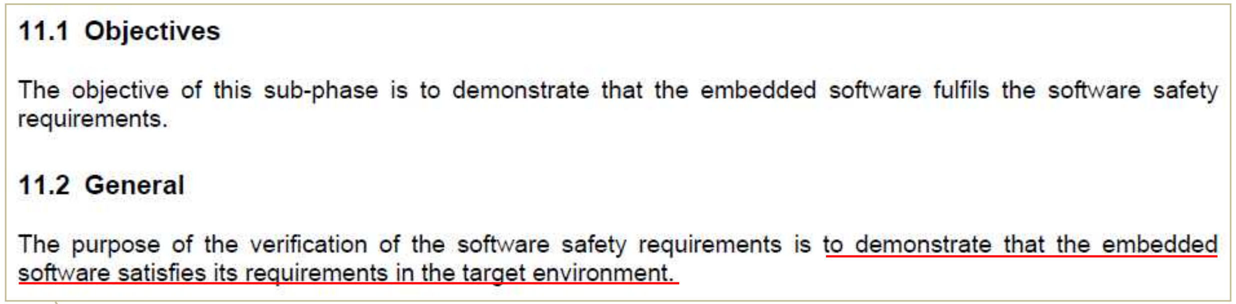
Software Integration Testing을 위한 **테스팅 환경**은 **실제 환경**과 가능한 유사해야 한다.
+ 차이점이 있을 경우, 다음 테스팅 단계를 위해 정확한 분석을 수행해야 한다.

6-10 Software Integration and Testing

Test Level		Embedded SW	Processor	Rest of embedded System	Plant
One-way simulation	MT (Model Test)	Simulated	-	-	-
Back-to-back simulation	MiL (Model-in-the-Loop)	Simulated	-	-	Simulated
Rapid prototyping	RP	Experimental	Experimental	Experimental	Real
SW Unit, SW integration (1)	SiL (SW-in-the-Loop)	Experimental (host)	Host	Simulated	Simulated
SW Unit, SW integration (2)	SiL	Real (target)	Emulator	Simulated	Simulated
HW/SW integration	HiL (HW-in-the-Loop)	Real (target)	Real (target)	Experimental	Simulated
System integration	HiL	Real (target)	Real (target)	Prototype	Simulated
Environmental	HiL/ST (System Test)	Real (target)	Real (target)	Real	Simulated
Pre-Production	ST	Real (target)	Real (target)	Real	Real

6-11 Verification of Software Safety Requirements

■ 6-11 Verification of Software Safety Requirements



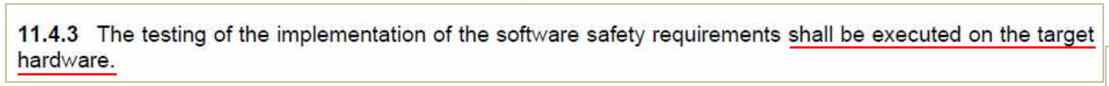
11.1 Objectives

The objective of this sub-phase is to demonstrate that the embedded software fulfils the software safety requirements.

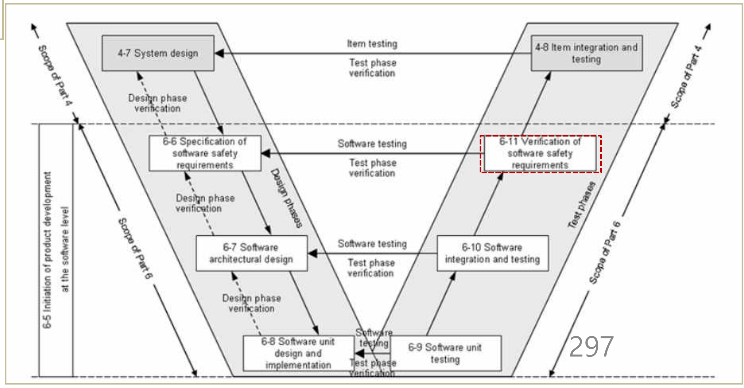
11.2 General

The purpose of the verification of the software safety requirements is to demonstrate that the embedded software satisfies its requirements in the target environment.

최종 통합된 **Embedded Software**가 **Software Safety Requirements**를 **만족**하는지 **확인(Verification)**한다.
+ Embedded Software가 실제로 구동되는 **Target Environment 고려**해서 **테스팅**을 수행
+ Embedded Software를 **Hardware에 탑재**해서 테스트 해야 한다.

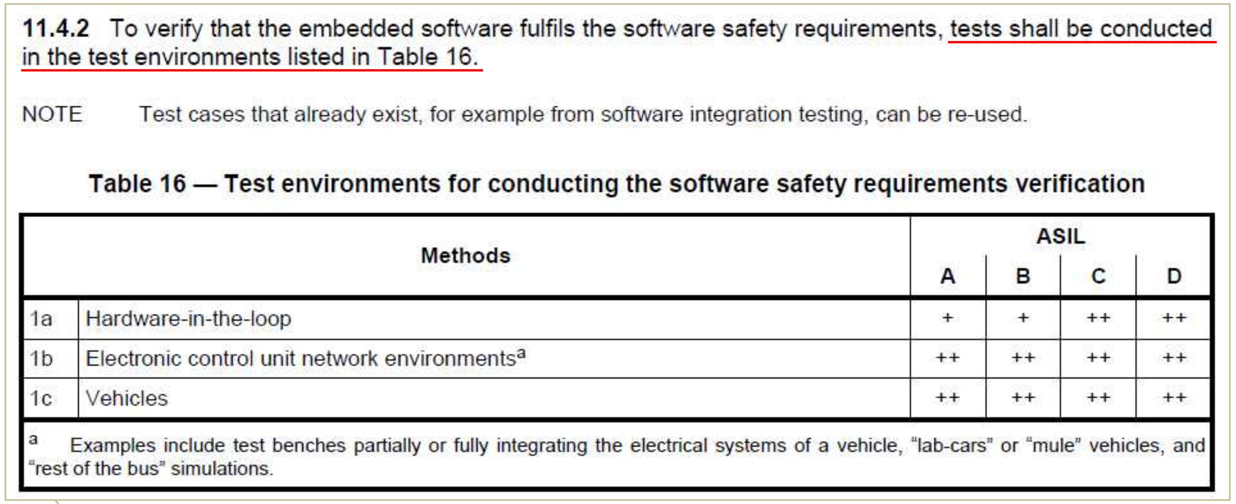


11.4.3 The testing of the implementation of the software safety requirements shall be executed on the target hardware.



Scope of Part 4
4-7 System design
Item testing
Test phase verification
4-8 Item integration and testing
Scope of Part 4
Design phase verification
6-6 Specification of software safety requirements
Software testing
Test phase verification
6-11 Verification of software safety requirements
6-5 Initiation of product development at the software level
Scope of Part 6
Design phase verification
Design phases
Test phases
6-7 Software architectural design
Software testing
Test phase verification
6-10 Software integration and testing
Scope of Part 6
Design phase verification
6-8 Software unit design and implementation
Software testing
Test phase verification
6-9 Software unit testing

6-11 Verification of Software Safety Requirements



11.4.2 To verify that the embedded software fulfils the software safety requirements, tests shall be conducted in the test environments listed in Table 16.

NOTE Test cases that already exist, for example from software integration testing, can be re-used.

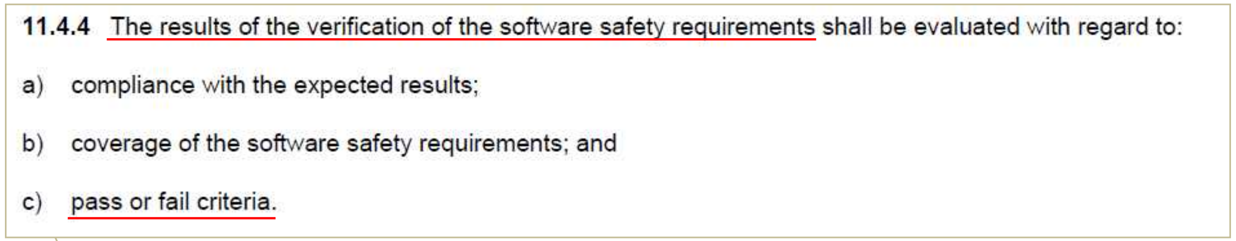
Table 16 — Test environments for conducting the software safety requirements verification

	Methods	**ASIL**			
		A	**B**	**C**	**D**
1a	Hardware-in-the-loop	+	+	++	++
1b	Electronic control unit network environments[a]	++	++	++	++
1c	Vehicles	++	++	++	++

[a] Examples include test benches partially or fully integrating the electrical systems of a vehicle, “lab-cars” or “mule” vehicles, and “rest of the bus” simulations.

Verification of SSR은 **Embedded Software**를 **타겟 하드웨어 환경**에서 **테스트**함으로써 수행된다.
+ **Table 16**

6-11 Verification of Software Safety Requirements



11.4.4 The results of the verification of the software safety requirements shall be evaluated with regard to:

a) compliance with the expected results;

b) coverage of the software safety requirements; and

c) pass or fail criteria.

Verification of Software Safety Requirements은
+ 테스팅 수행(100% 통과 要)
+ SSR에 대한 Requirements Coverage 계산
+ **Pass/Fail 판정**

을 포함한다.

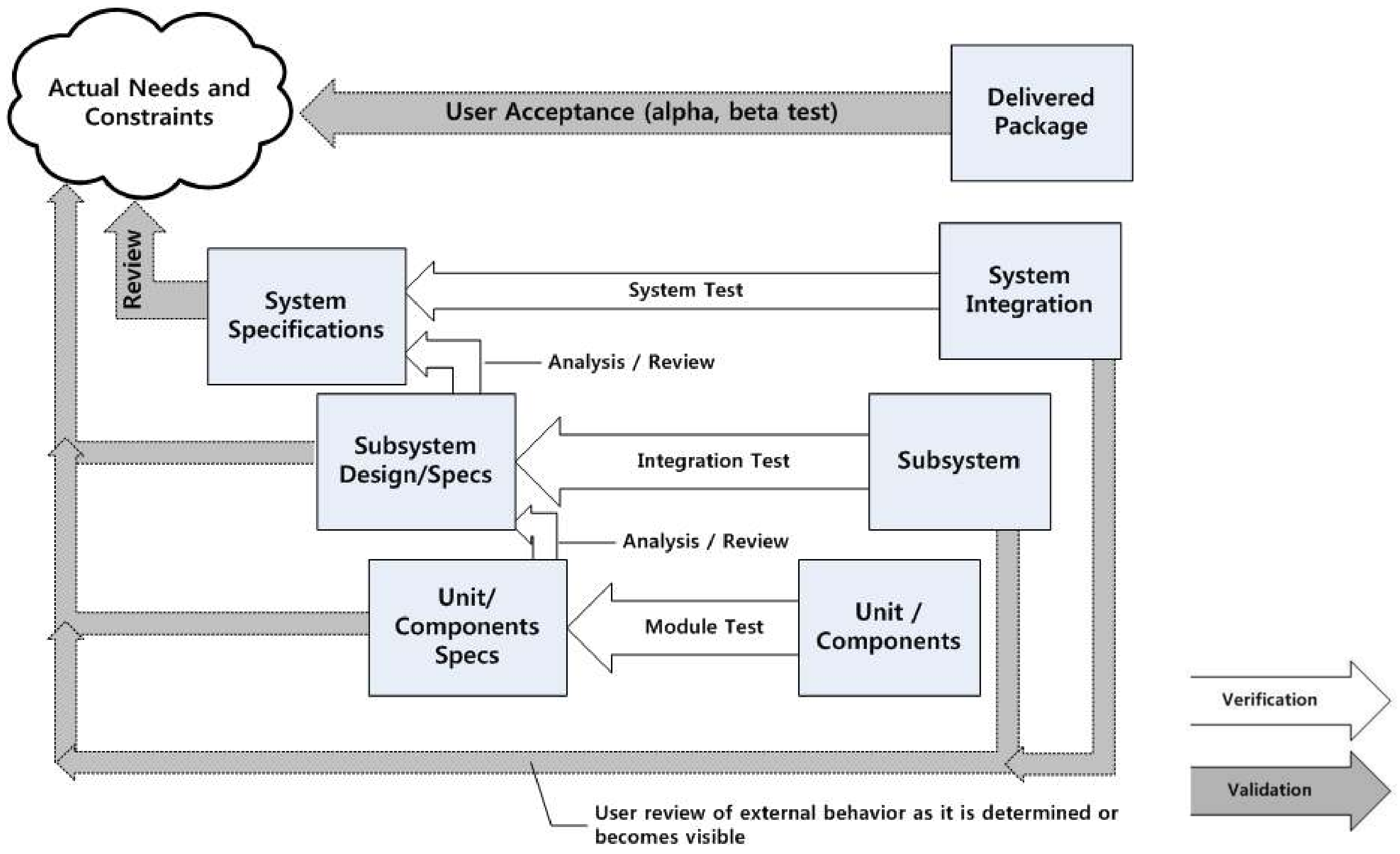
SUMMARY

Software Quality Process

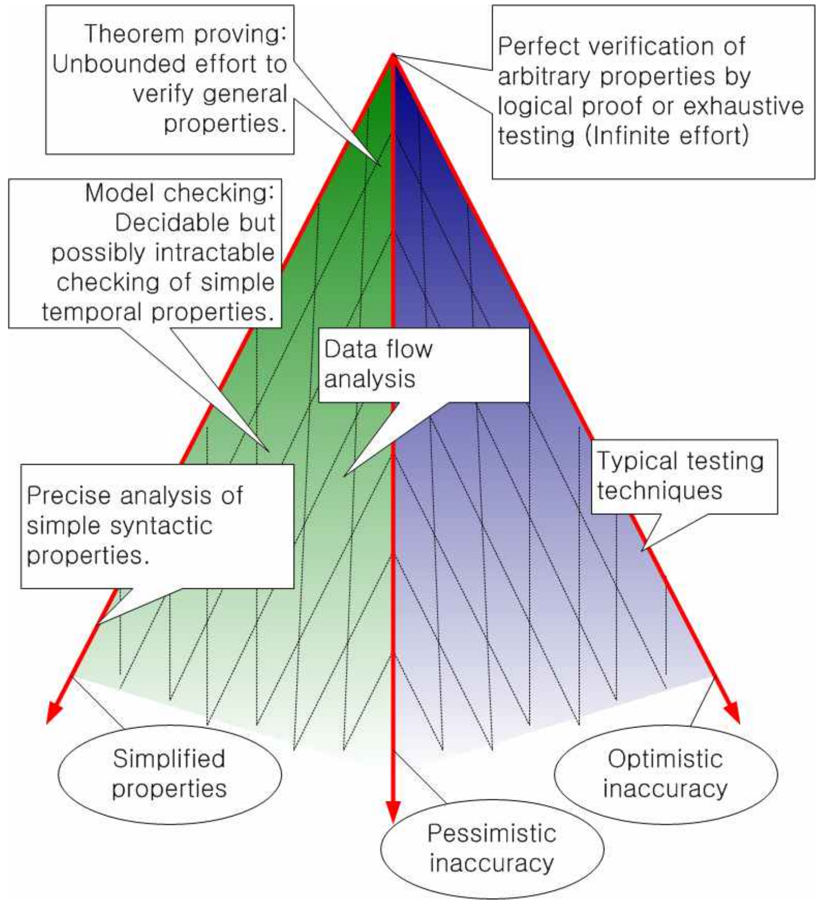
- **Quality process**
  - **A set of activities and responsibilities** focusing on ensuring adequate dependability concerned with project schedule or with product usability
  - **A&T planning** is Integral to the quality process.
    - **Quality goals** can be achieved only through careful A&T planning.
    - **Selects and arranges** STA activities to be as cost-effective as possible
    - Should balance several STA activities across the whole development process

- Quality process provides **a framework** for
  - Selecting and arranging STA activities, and also
  - Considering interactions and trade-offs *with other important goals*.

V-Model of V&V Activities



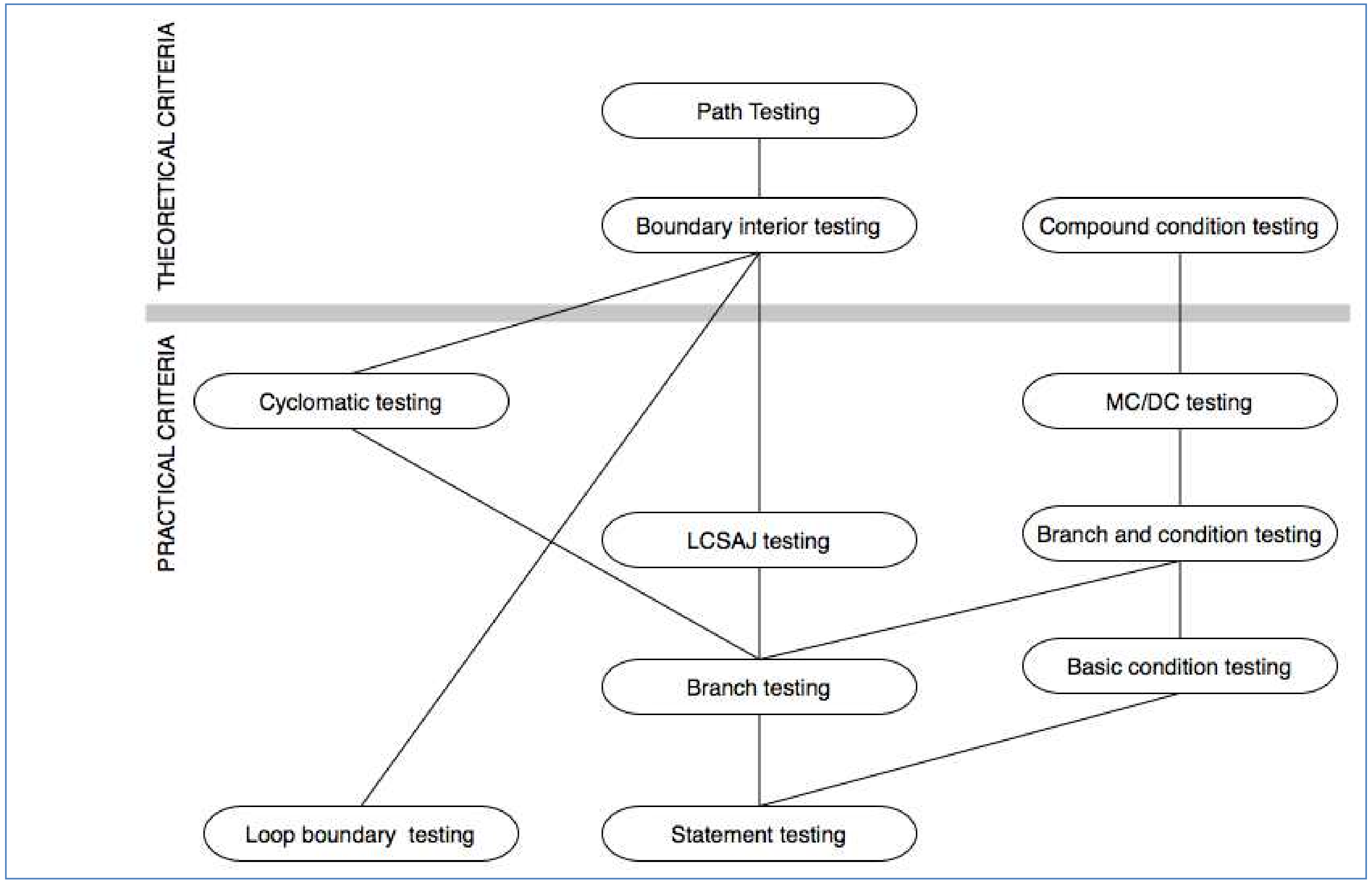
3 Dimensions of STA Activities



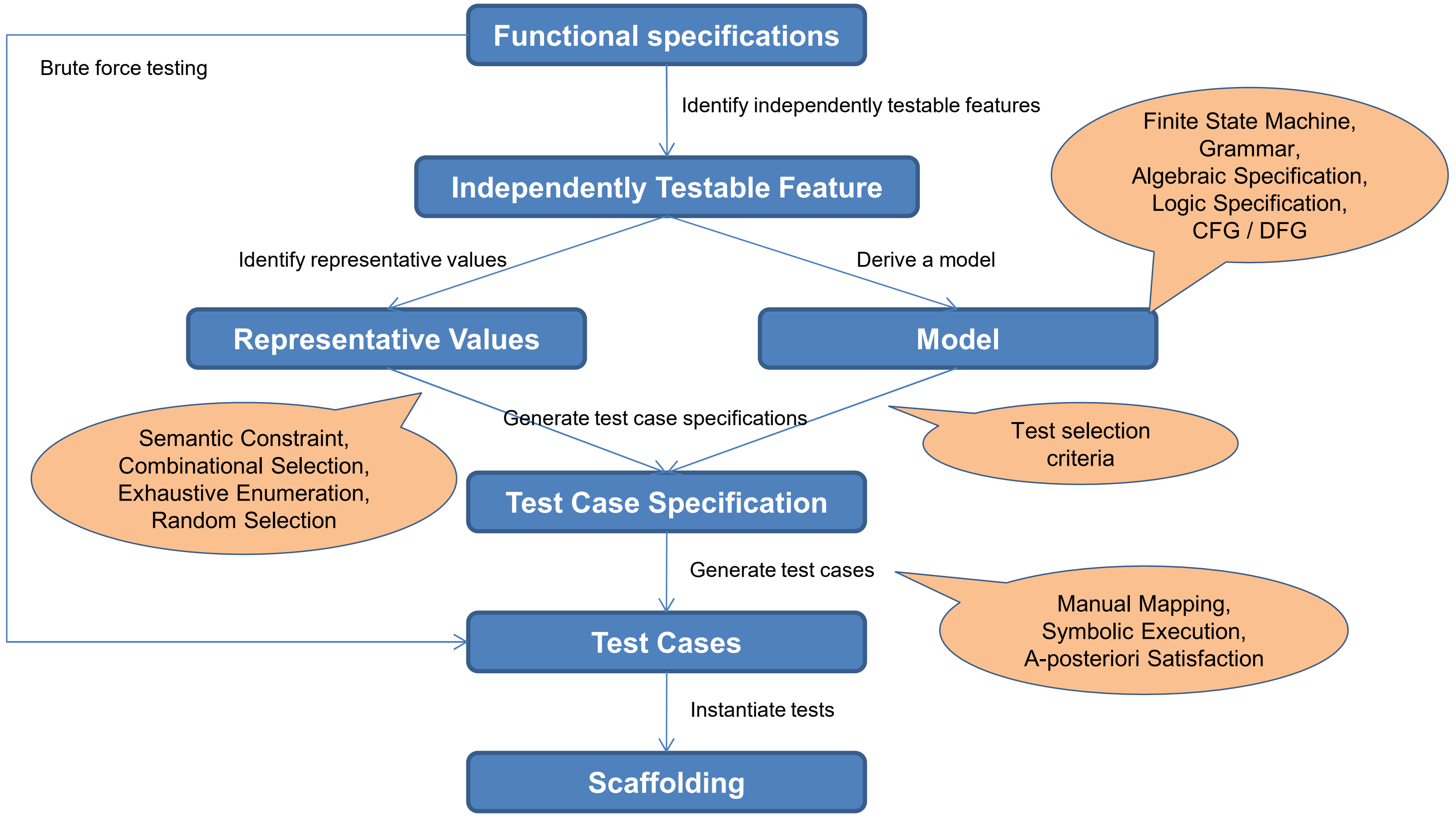
Theorem proving: Unbounded effort to verify general properties.
Perfect verification of arbitrary properties by logical proof or exhaustive testing (Infinite effort)
Model checking: Decidable but possibly intractable checking of simple temporal properties.
Data flow analysis
Typical testing techniques
Precise analysis of simple syntactic properties.
Simplified properties
Optimistic inaccuracy
Pessimistic inaccuracy

- **Optimistic Inaccuracy**
  - We may accept some programs that do not possess the property.
  - It may not detect all violations.
  - Testing
- **Pessimistic Inaccuracy**
  - Not guaranteed to accept a program even if the program does possess the property being analyzed, because of false alarms
  - Static Code Analysis
- **Simplified Properties**
  - It reduces the degree of freedom by simplifying the property to check.
  - Theorem Proving, Model Checking

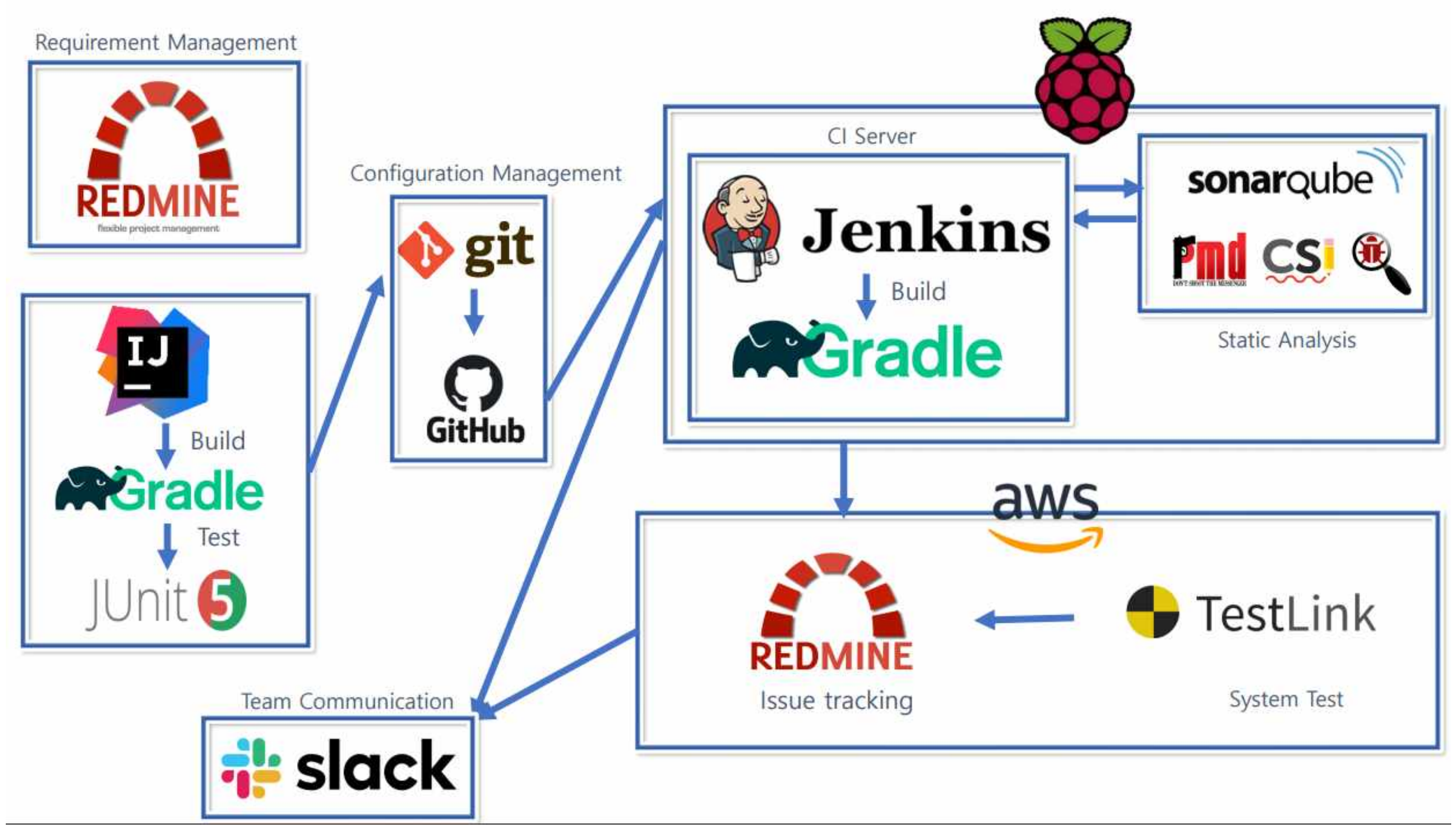
Testing Coverages



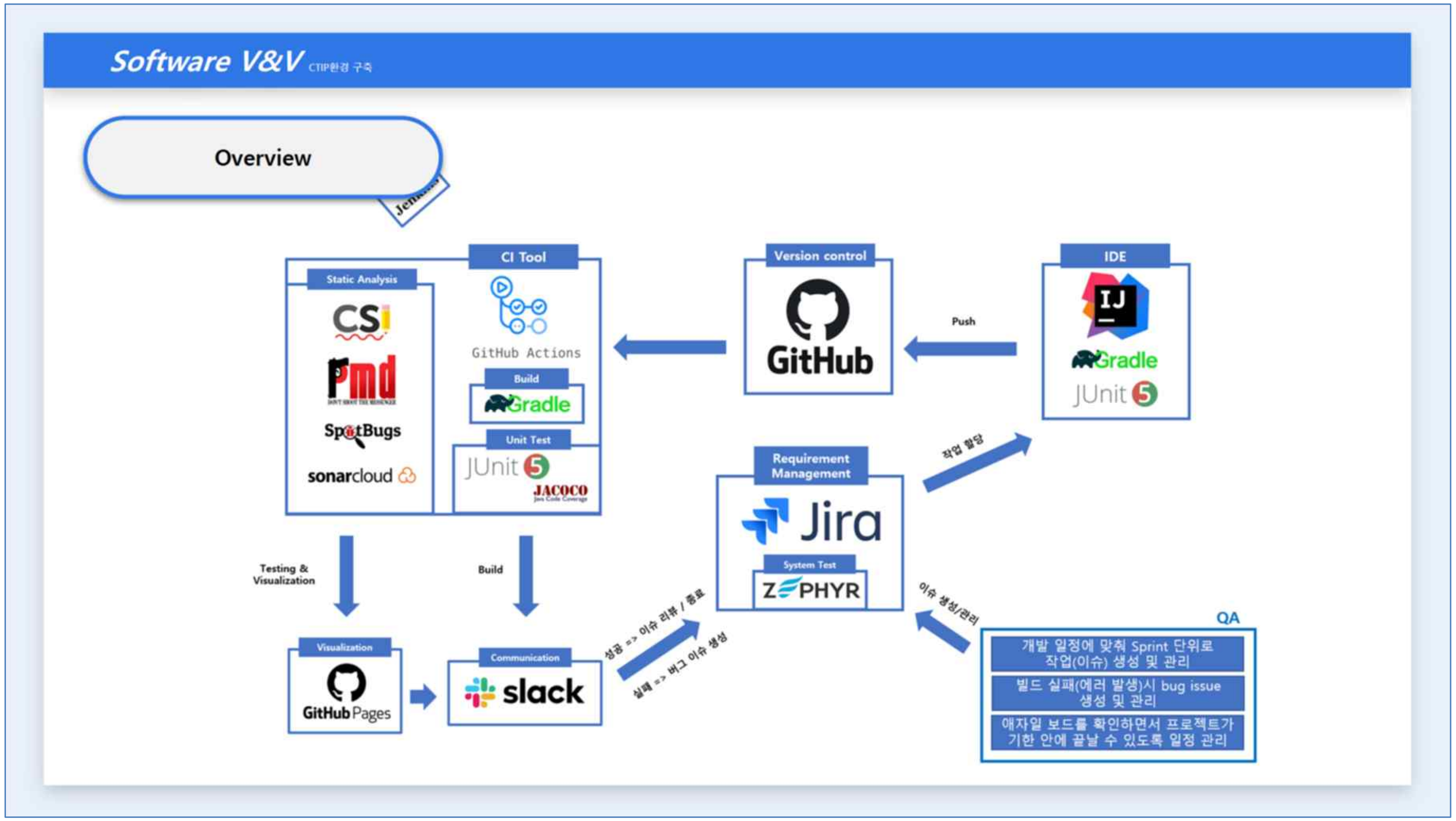
Functional Program Testing


Functional specifications
Brute force testing
Identify independently testable features
Independently Testable Feature
Finite State Machine, Grammar, Algebraic Specification, Logic Specification, CFG / DFG
Identify representative values
Derive a model
Representative Values
Model
Generate test case specifications
Semantic Constraint, Combinational Selection, Exhaustive Enumeration, Random Selection
Test selection criteria
Test Case Specification
Generate test cases
Manual Mapping, Symbolic Execution, A-posteriori Satisfaction
Test Cases
Instantiate tests
Scaffolding

CTIP Examples



CTIP Examples


Software V&V CTIP환경 구축

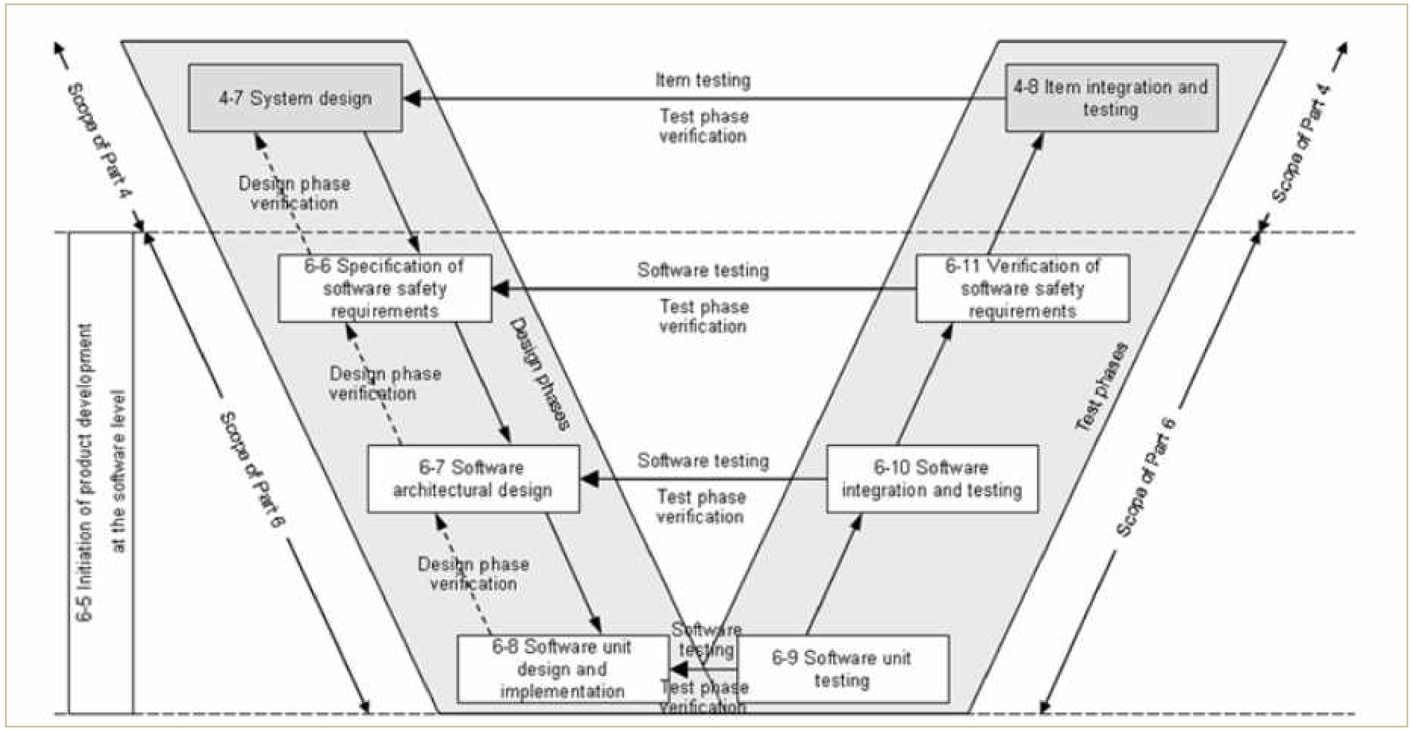
Overview

개발 일정에 맞춰 Sprint 단위로 작업(이슈) 생성 및 관리

빌드 실패(에러 발생)시 bug issue 생성 및 관리

애자일 보드를 확인하면서 프로젝트가 기한 안에 끝날 수 있도록 일정 관리

V-Model in IEC 61508



4-7 System design
Item testing
Test phase verification
4-8 Item integration and testing
Scope of Part 4
Scope of Part 4
Design phase verification
6-6 Specification of software safety requirements
Software testing
Test phase verification
6-11 Verification of software safety requirements
Design phase verification
Design phases
Test phases
6-5 Initiation of product development at the software level
Scope of Part 6
Scope of Part 6
6-7 Software architectural design
Software testing
Test phase verification
6-10 Software integration and testing
Design phase verification
6-8 Software unit design and implementation
Software testing
Test phase verification
6-9 Software unit testing

Figure 2 — Reference phase model for the software development

STA Techniques in IEC 61508 Standard

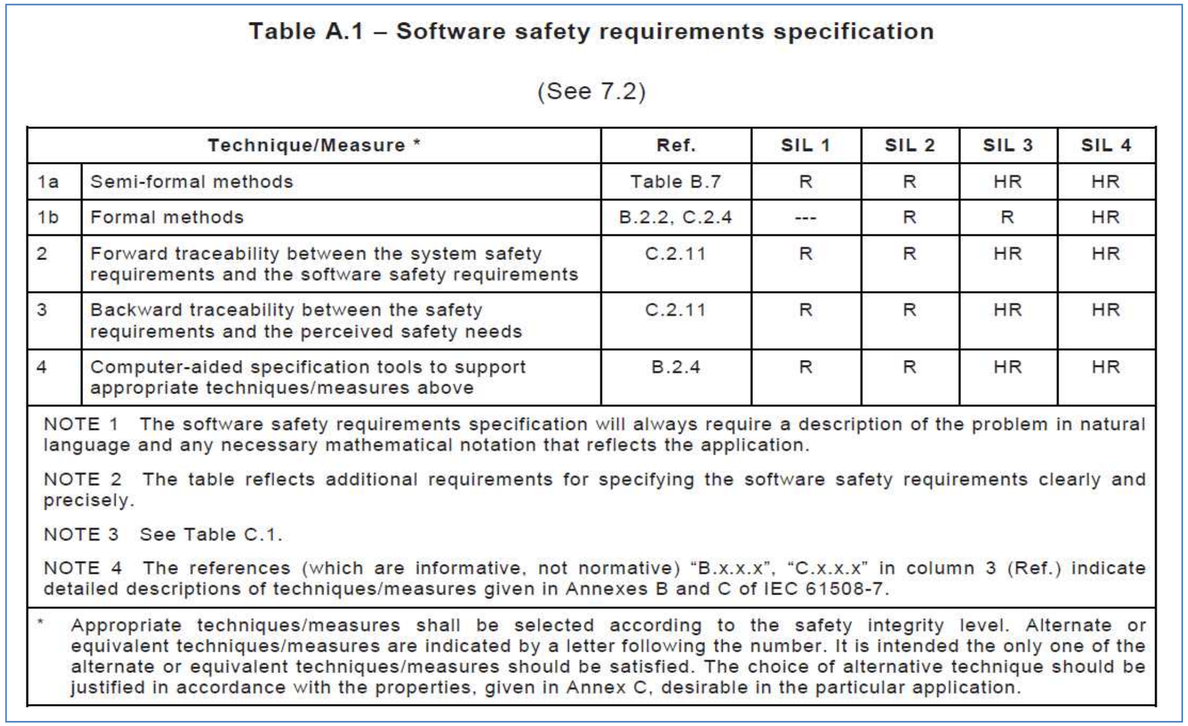


Table A.1 – Software safety requirements specification

(See 7.2)

Technique/Measure *		Ref.	SIL 1	SIL 2	SIL 3	SIL 4
1a	Semi-formal methods	Table B.7	R	R	HR	HR
1b	Formal methods	B.2.2, C.2.4	---	R	R	HR
2	Forward traceability between the system safety requirements and the software safety requirements	C.2.11	R	R	HR	HR
3	Backward traceability between the safety requirements and the perceived safety needs	C.2.11	R	R	HR	HR
4	Computer-aided specification tools to support appropriate techniques/measures above	B.2.4	R	R	HR	HR

NOTE 1 The software safety requirements specification will always require a description of the problem in natural language and any necessary mathematical notation that reflects the application.

NOTE 2 The table reflects additional requirements for specifying the software safety requirements clearly and precisely.

NOTE 3 See Table C.1.

NOTE 4 The references (which are informative, not normative) "B.x.x.x", "C.x.x.x" in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7.

* Appropriate techniques/measures shall be selected according to the safety integrity level. Alternate or equivalent techniques/measures are indicated by a letter following the number. It is intended the only one of the alternate or equivalent techniques/measures should be satisfied. The choice of alternative technique should be justified in accordance with the properties, given in Annex C, desirable in the particular application.

STA Techniques in IEC 61508 Standard

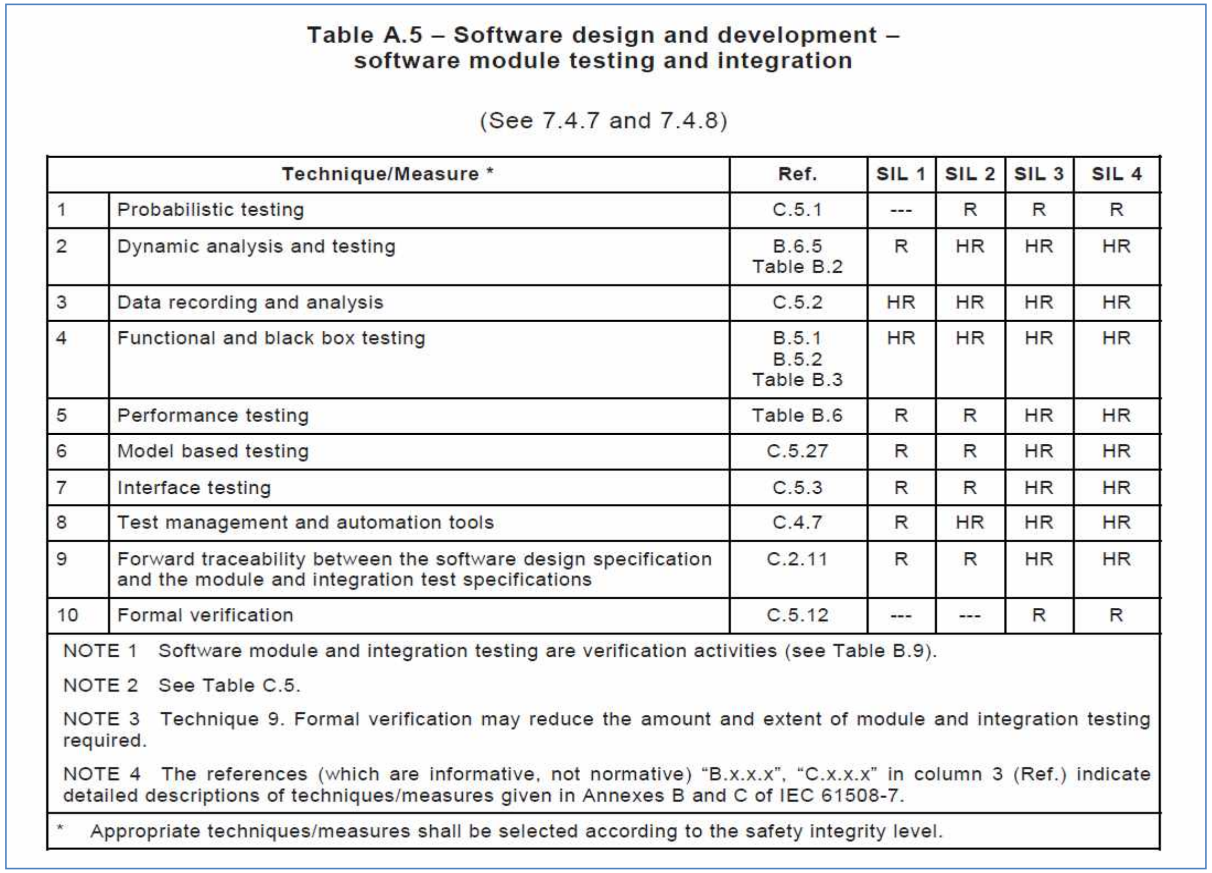


Table A.5 – Software design and development – software module testing and integration

(See 7.4.7 and 7.4.8)

Technique/Measure *		Ref.	SIL 1	SIL 2	SIL 3	SIL 4
1	Probabilistic testing	C.5.1	---	R	R	R
2	Dynamic analysis and testing	B.6.5 Table B.2	R	HR	HR	HR
3	Data recording and analysis	C.5.2	HR	HR	HR	HR
4	Functional and black box testing	B.5.1 B.5.2 Table B.3	HR	HR	HR	HR
5	Performance testing	Table B.6	R	R	HR	HR
6	Model based testing	C.5.27	R	R	HR	HR
7	Interface testing	C.5.3	R	R	HR	HR
8	Test management and automation tools	C.4.7	R	HR	HR	HR
9	Forward traceability between the software design specification and the module and integration test specifications	C.2.11	R	R	HR	HR
10	Formal verification	C.5.12	---	---	R	R

NOTE 1 Software module and integration testing are verification activities (see Table B.9).

NOTE 2 See Table C.5.

NOTE 3 Technique 9. Formal verification may reduce the amount and extent of module and integration testing required.

NOTE 4 The references (which are informative, not normative) "B.x.x.x", "C.x.x.x" in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7.

* Appropriate techniques/measures shall be selected according to the safety integrity level.

STA Techniques in IEC 61508 Standard

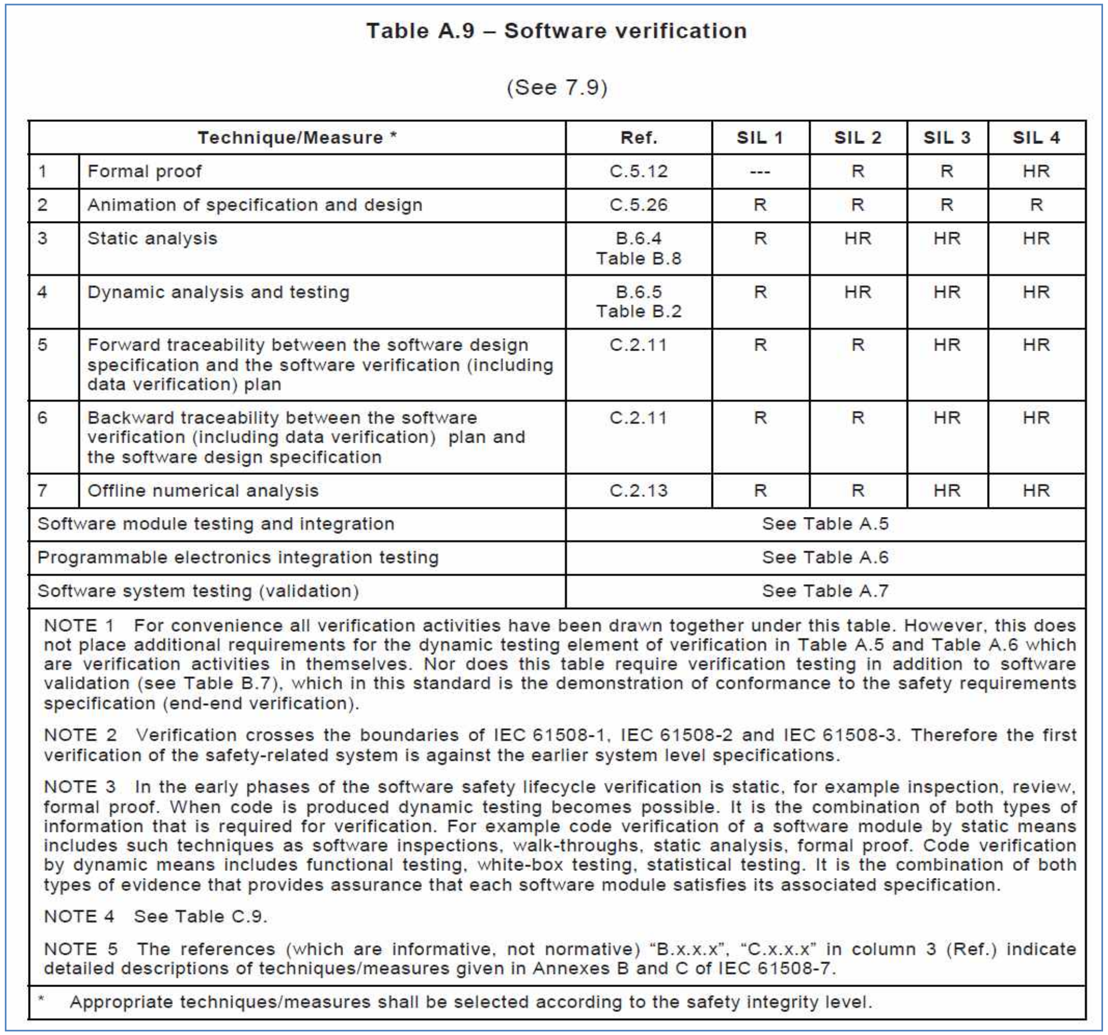


Table A.9 – Software verification

(See 7.9)

Technique/Measure *		Ref.	SIL 1	SIL 2	SIL 3	SIL 4
1	Formal proof	C.5.12	---	R	R	HR
2	Animation of specification and design	C.5.26	R	R	R	R
3	Static analysis	B.6.4 Table B.8	R	HR	HR	HR
4	Dynamic analysis and testing	B.6.5 Table B.2	R	HR	HR	HR
5	Forward traceability between the software design specification and the software verification (including data verification) plan	C.2.11	R	R	HR	HR
6	Backward traceability between the software verification (including data verification) plan and the software design specification	C.2.11	R	R	HR	HR
7	Offline numerical analysis	C.2.13	R	R	HR	HR
Software module testing and integration		See Table A.5				
Programmable electronics integration testing		See Table A.6				
Software system testing (validation)		See Table A.7				

NOTE 1 For convenience all verification activities have been drawn together under this table. However, this does not place additional requirements for the dynamic testing element of verification in Table A.5 and Table A.6 which are verification activities in themselves. Nor does this table require verification testing in addition to software validation (see Table B.7), which in this standard is the demonstration of conformance to the safety requirements specification (end-end verification).

NOTE 2 Verification crosses the boundaries of IEC 61508-1, IEC 61508-2 and IEC 61508-3. Therefore the first verification of the safety-related system is against the earlier system level specifications.

NOTE 3 In the early phases of the software safety lifecycle verification is static, for example inspection, review, formal proof. When code is produced dynamic testing becomes possible. It is the combination of both types of information that is required for verification. For example code verification of a software module by static means includes such techniques as software inspections, walk-throughs, static analysis, formal proof. Code verification by dynamic means includes functional testing, white-box testing, statistical testing. It is the combination of both types of evidence that provides assurance that each software module satisfies its associated specification.

NOTE 4 See Table C.9.

NOTE 5 The references (which are informative, not normative) "B.x.x.x", "C.x.x.x" in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7.

* Appropriate techniques/measures shall be selected according to the safety integrity level.

STA Techniques in IEC 61508 Standard

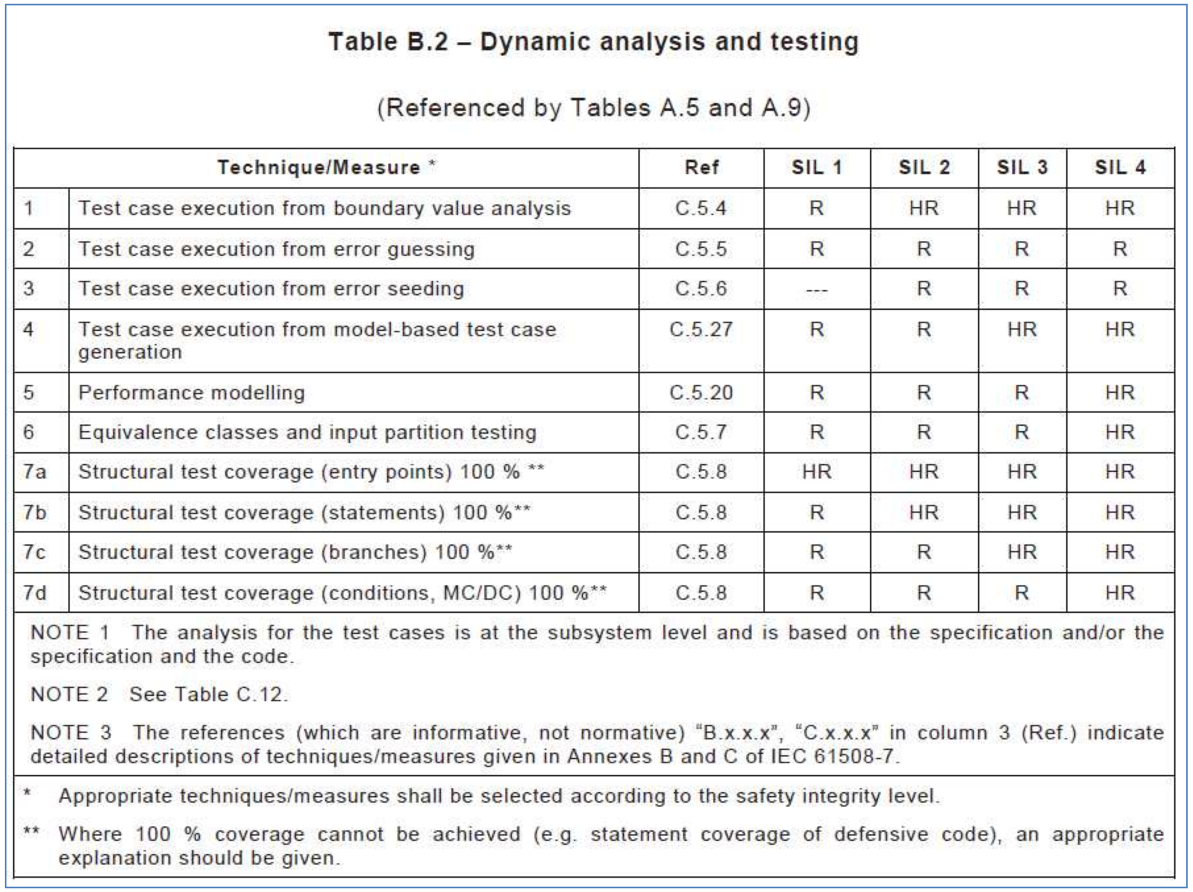


Table B.2 – Dynamic analysis and testing

(Referenced by Tables A.5 and A.9)

	Technique/Measure *	Ref	SIL 1	SIL 2	SIL 3	SIL 4
1	Test case execution from boundary value analysis	C.5.4	R	HR	HR	HR
2	Test case execution from error guessing	C.5.5	R	R	R	R
3	Test case execution from error seeding	C.5.6	---	R	R	R
4	Test case execution from model-based test case generation	C.5.27	R	R	HR	HR
5	Performance modelling	C.5.20	R	R	R	HR
6	Equivalence classes and input partition testing	C.5.7	R	R	R	HR
7a	Structural test coverage (entry points) 100 % **	C.5.8	HR	HR	HR	HR
7b	Structural test coverage (statements) 100 %**	C.5.8	R	HR	HR	HR
7c	Structural test coverage (branches) 100 %**	C.5.8	R	R	HR	HR
7d	Structural test coverage (conditions, MC/DC) 100 %**	C.5.8	R	R	R	HR

NOTE 1 The analysis for the test cases is at the subsystem level and is based on the specification and/or the specification and the code.

NOTE 2 See Table C.12.

NOTE 3 The references (which are informative, not normative) "B.x.x.x", "C.x.x.x" in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7.

* Appropriate techniques/measures shall be selected according to the safety integrity level.

** Where 100 % coverage cannot be achieved (e.g. statement coverage of defensive code), an appropriate explanation should be given.

STA Techniques in IEC 61508 Standard

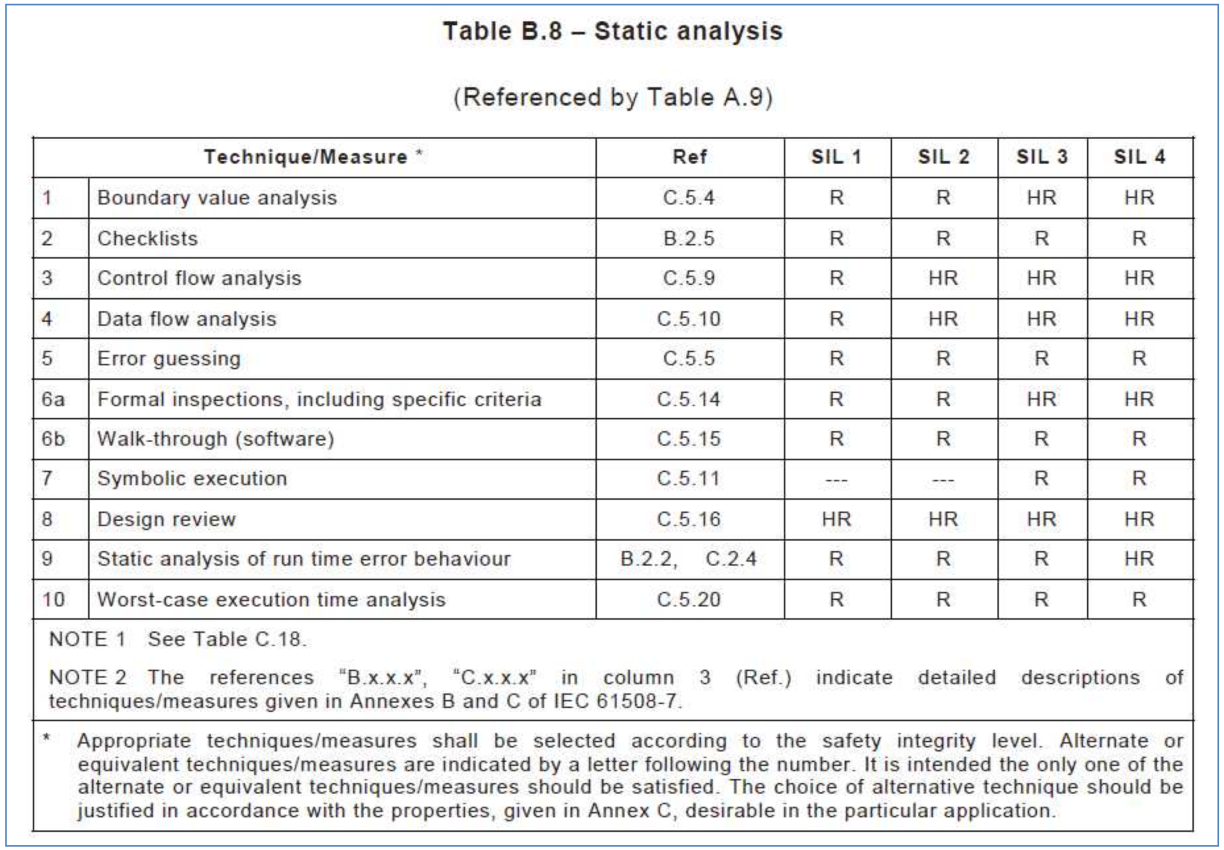


Table B.8 – Static analysis

(Referenced by Table A.9)

Technique/Measure *		Ref	SIL 1	SIL 2	SIL 3	SIL 4
1	Boundary value analysis	C.5.4	R	R	HR	HR
2	Checklists	B.2.5	R	R	R	R
3	Control flow analysis	C.5.9	R	HR	HR	HR
4	Data flow analysis	C.5.10	R	HR	HR	HR
5	Error guessing	C.5.5	R	R	R	R
6a	Formal inspections, including specific criteria	C.5.14	R	R	HR	HR
6b	Walk-through (software)	C.5.15	R	R	R	R
7	Symbolic execution	C.5.11	---	---	R	R
8	Design review	C.5.16	HR	HR	HR	HR
9	Static analysis of run time error behaviour	B.2.2, C.2.4	R	R	R	HR
10	Worst-case execution time analysis	C.5.20	R	R	R	R

NOTE 1 See Table C.18.

NOTE 2 The references "B.x.x.x", "C.x.x.x" in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7.

* Appropriate techniques/measures shall be selected according to the safety integrity level. Alternate or equivalent techniques/measures are indicated by a letter following the number. It is intended the only one of the alternate or equivalent techniques/measures should be satisfied. The choice of alternative technique should be justified in accordance with the properties, given in Annex C, desirable in the particular application.

STA Techniques in IEC 61508 Standard

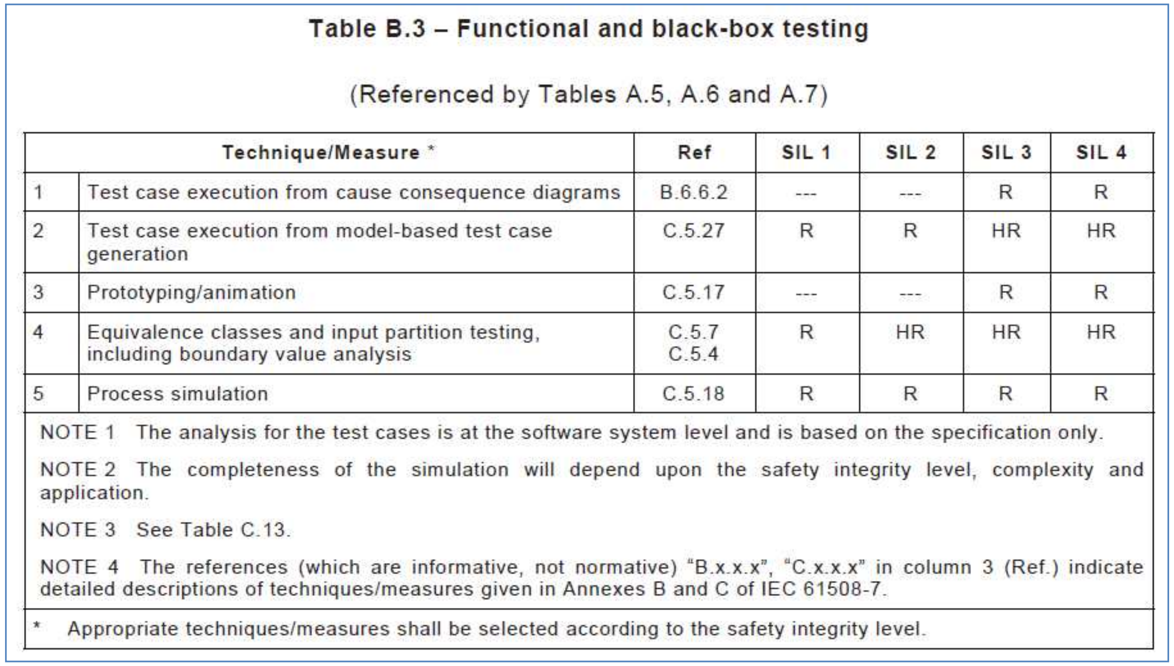


Table B.3 – Functional and black-box testing

(Referenced by Tables A.5, A.6 and A.7)

Technique/Measure *		Ref	SIL 1	SIL 2	SIL 3	SIL 4
1	Test case execution from cause consequence diagrams	B.6.6.2	---	---	R	R
2	Test case execution from model-based test case generation	C.5.27	R	R	HR	HR
3	Prototyping/animation	C.5.17	---	---	R	R
4	Equivalence classes and input partition testing, including boundary value analysis	C.5.7 C.5.4	R	HR	HR	HR
5	Process simulation	C.5.18	R	R	R	R

NOTE 1 The analysis for the test cases is at the software system level and is based on the specification only.

NOTE 2 The completeness of the simulation will depend upon the safety integrity level, complexity and application.

NOTE 3 See Table C.13.

NOTE 4 The references (which are informative, not normative) "B.x.x.x", "C.x.x.x" in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7.

* Appropriate techniques/measures shall be selected according to the safety integrity level.

STA Techniques in IEC 61508 Standard

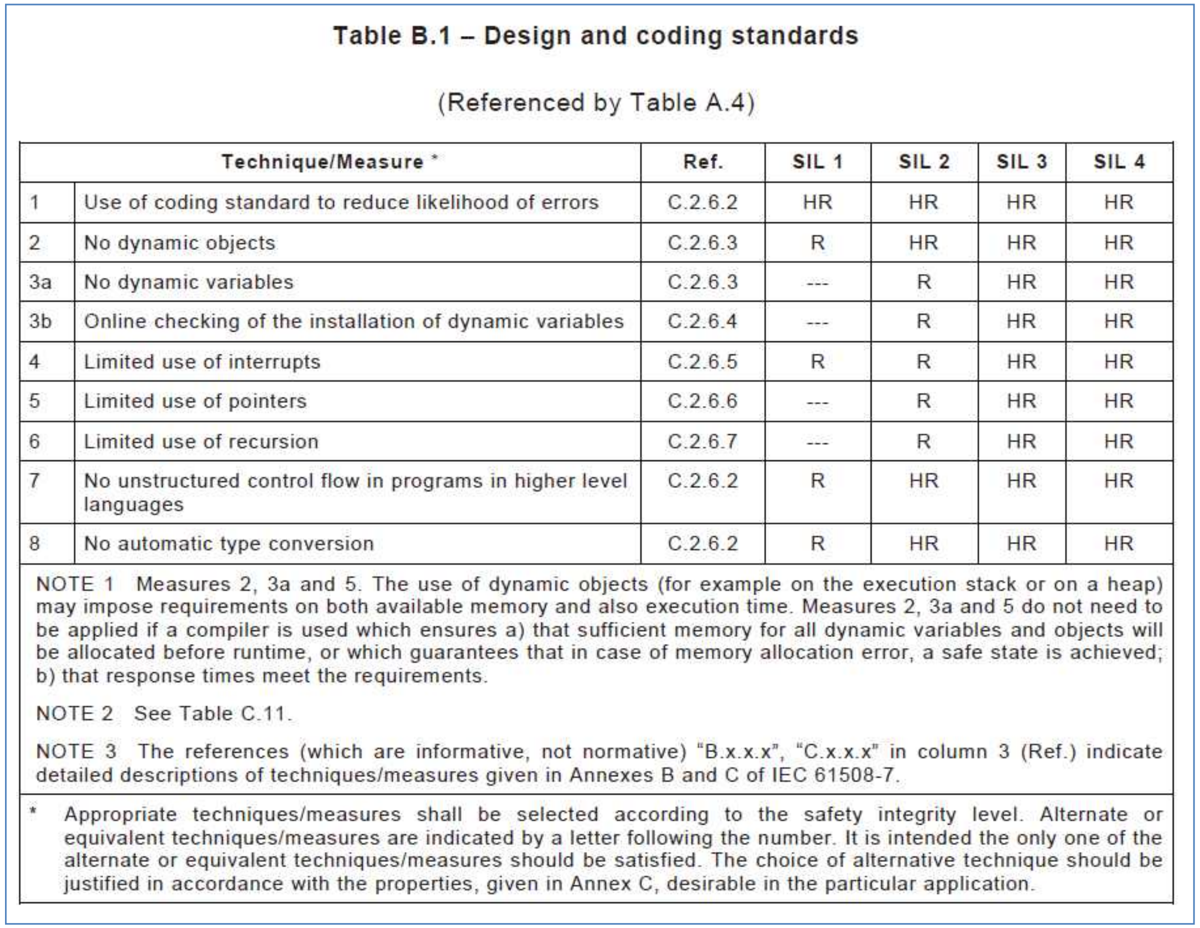


Table B.1 – Design and coding standards

(Referenced by Table A.4)

Technique/Measure *		Ref.	SIL 1	SIL 2	SIL 3	SIL 4
1	Use of coding standard to reduce likelihood of errors	C.2.6.2	HR	HR	HR	HR
2	No dynamic objects	C.2.6.3	R	HR	HR	HR
3a	No dynamic variables	C.2.6.3	---	R	HR	HR
3b	Online checking of the installation of dynamic variables	C.2.6.4	---	R	HR	HR
4	Limited use of interrupts	C.2.6.5	R	R	HR	HR
5	Limited use of pointers	C.2.6.6	---	R	HR	HR
6	Limited use of recursion	C.2.6.7	---	R	HR	HR
7	No unstructured control flow in programs in higher level languages	C.2.6.2	R	HR	HR	HR
8	No automatic type conversion	C.2.6.2	R	HR	HR	HR

NOTE 1 Measures 2, 3a and 5. The use of dynamic objects (for example on the execution stack or on a heap) may impose requirements on both available memory and also execution time. Measures 2, 3a and 5 do not need to be applied if a compiler is used which ensures a) that sufficient memory for all dynamic variables and objects will be allocated before runtime, or which guarantees that in case of memory allocation error, a safe state is achieved; b) that response times meet the requirements.

NOTE 2 See Table C.11.

NOTE 3 The references (which are informative, not normative) "B.x.x.x", "C.x.x.x" in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7.

* Appropriate techniques/measures shall be selected according to the safety integrity level. Alternate or equivalent techniques/measures are indicated by a letter following the number. It is intended the only one of the alternate or equivalent techniques/measures should be satisfied. The choice of alternative technique should be justified in accordance with the properties, given in Annex C, desirable in the particular application.

STA Techniques in IEC 61508 Standard

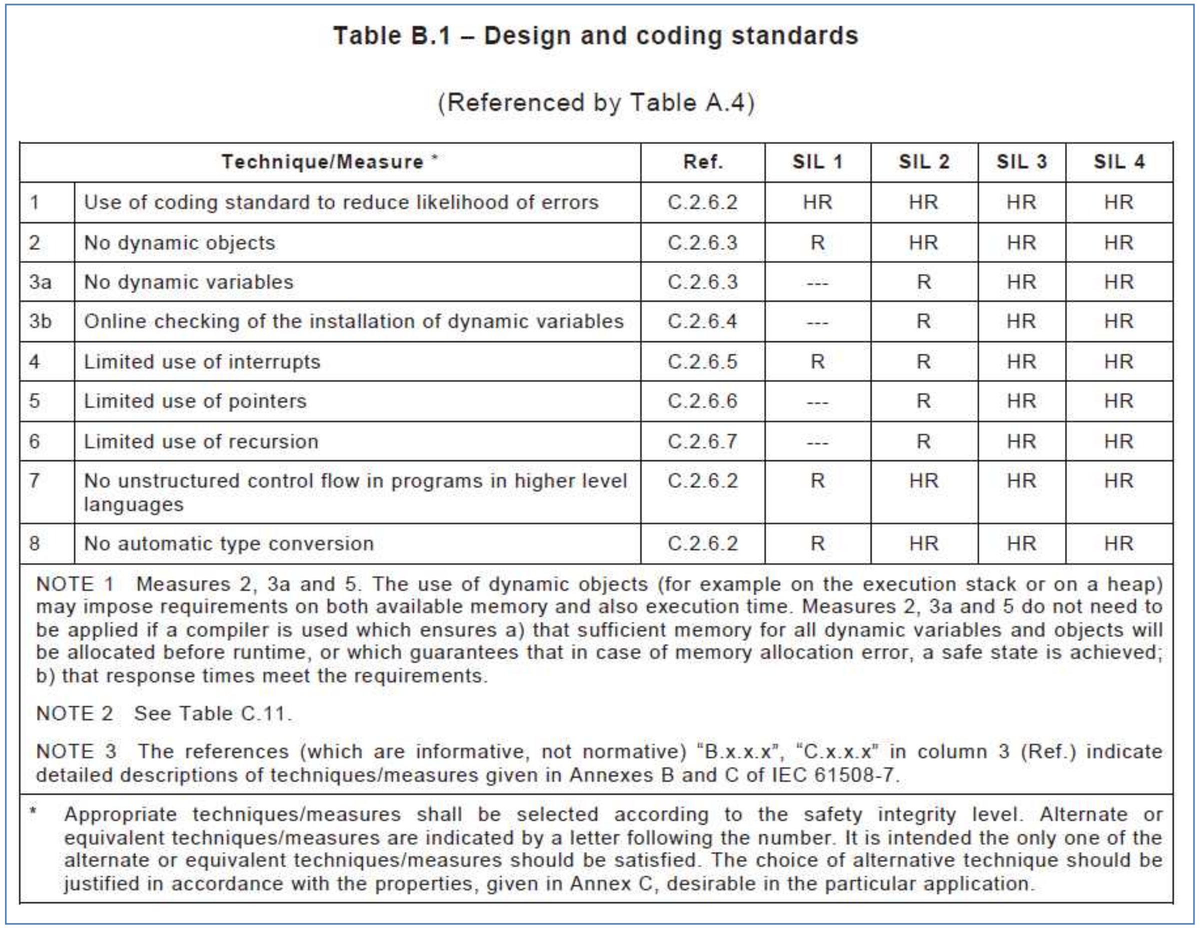


Table B.1 – Design and coding standards

(Referenced by Table A.4)

Technique/Measure *		Ref.	SIL 1	SIL 2	SIL 3	SIL 4
1	Use of coding standard to reduce likelihood of errors	C.2.6.2	HR	HR	HR	HR
2	No dynamic objects	C.2.6.3	R	HR	HR	HR
3a	No dynamic variables	C.2.6.3	---	R	HR	HR
3b	Online checking of the installation of dynamic variables	C.2.6.4	---	R	HR	HR
4	Limited use of interrupts	C.2.6.5	R	R	HR	HR
5	Limited use of pointers	C.2.6.6	---	R	HR	HR
6	Limited use of recursion	C.2.6.7	---	R	HR	HR
7	No unstructured control flow in programs in higher level languages	C.2.6.2	R	HR	HR	HR
8	No automatic type conversion	C.2.6.2	R	HR	HR	HR

NOTE 1 Measures 2, 3a and 5. The use of dynamic objects (for example on the execution stack or on a heap) may impose requirements on both available memory and also execution time. Measures 2, 3a and 5 do not need to be applied if a compiler is used which ensures a) that sufficient memory for all dynamic variables and objects will be allocated before runtime, or which guarantees that in case of memory allocation error, a safe state is achieved; b) that response times meet the requirements.

NOTE 2 See Table C.11.

NOTE 3 The references (which are informative, not normative) "B.x.x.x", "C.x.x.x" in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7.

* Appropriate techniques/measures shall be selected according to the safety integrity level. Alternate or equivalent techniques/measures are indicated by a letter following the number. It is intended the only one of the alternate or equivalent techniques/measures should be satisfied. The choice of alternative technique should be justified in accordance with the properties, given in Annex C, desirable in the particular application.

STA Techniques in ISO 26262 Standard

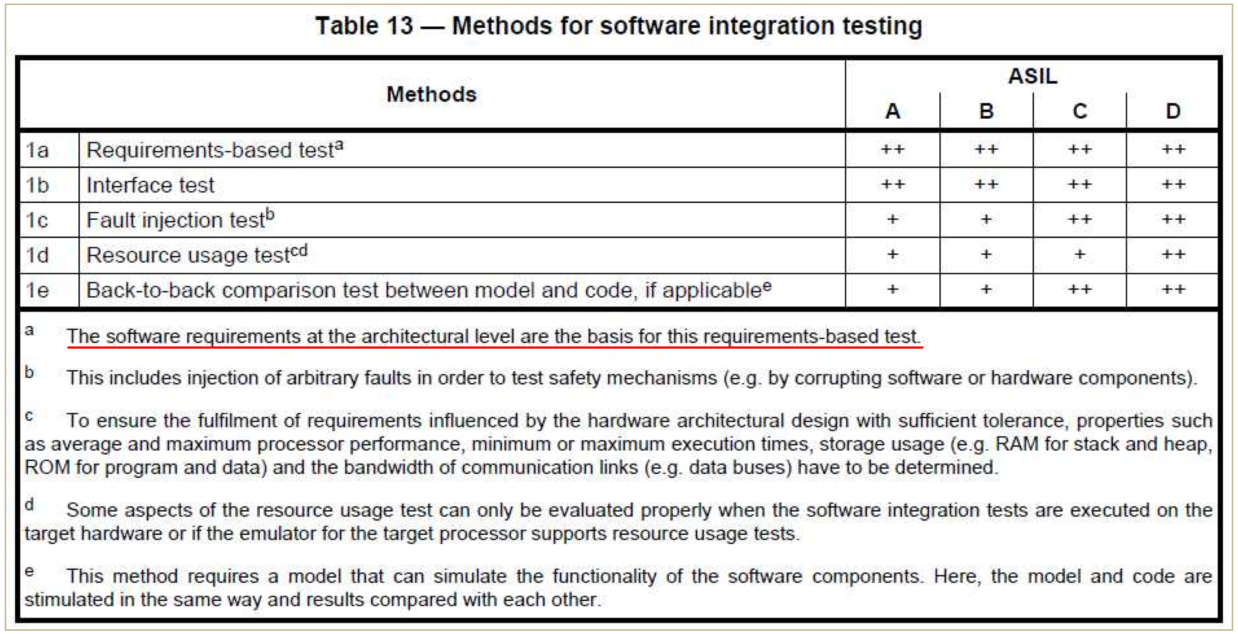


Table 13 — Methods for software integration testing

Methods		ASIL			
		A	B	C	D
1a	Requirements-based test[a]	++	++	++	++
1b	Interface test	++	++	++	++
1c	Fault injection test[b]	+	+	++	++
1d	Resource usage test[cd]	+	+	+	++
1e	Back-to-back comparison test between model and code, if applicable[e]	+	+	++	++

[a] The software requirements at the architectural level are the basis for this requirements-based test.

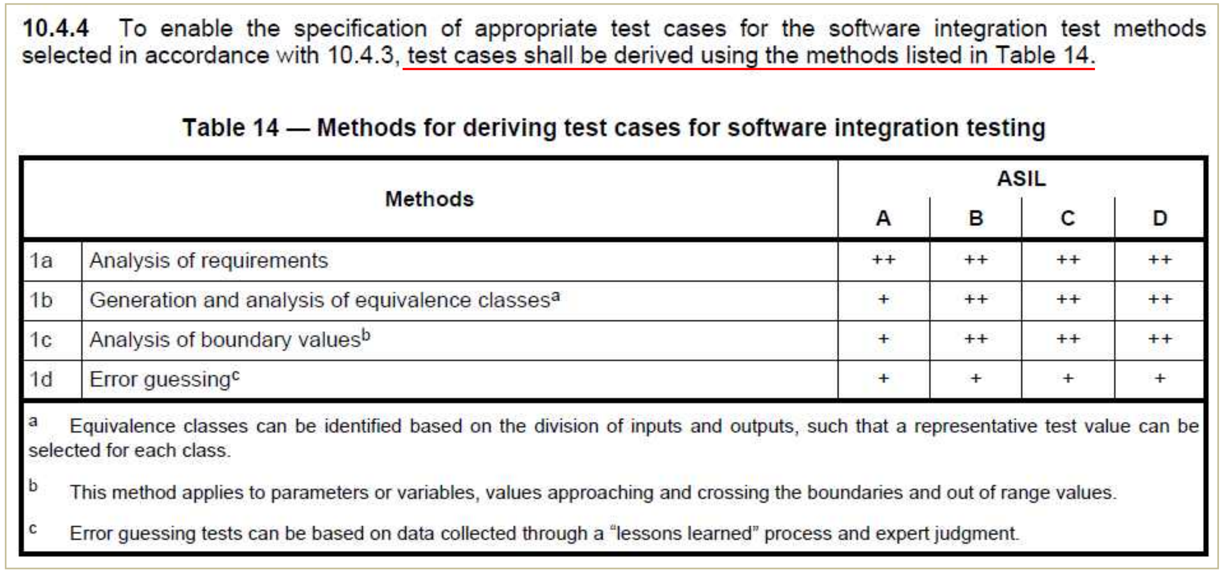
[b] This includes injection of arbitrary faults in order to test safety mechanisms (e.g. by corrupting software or hardware components).

[c] To ensure the fulfilment of requirements influenced by the hardware architectural design with sufficient tolerance, properties such as average and maximum processor performance, minimum or maximum execution times, storage usage (e.g. RAM for stack and heap, ROM for program and data) and the bandwidth of communication links (e.g. data buses) have to be determined.

[d] Some aspects of the resource usage test can only be evaluated properly when the software integration tests are executed on the target hardware or if the emulator for the target processor supports resource usage tests.

[e] This method requires a model that can simulate the functionality of the software components. Here, the model and code are stimulated in the same way and results compared with each other.

STA Techniques in ISO 26262 Standard



10.4.4 To enable the specification of appropriate test cases for the software integration test methods selected in accordance with 10.4.3, test cases shall be derived using the methods listed in Table 14.

Table 14 — Methods for deriving test cases for software integration testing

Methods		ASIL			
		A	B	C	D
1a	Analysis of requirements	++	++	++	++
1b	Generation and analysis of equivalence classes[a]	+	++	++	++
1c	Analysis of boundary values[b]	+	++	++	++
1d	Error guessing[c]	+	+	+	+

[a] Equivalence classes can be identified based on the division of inputs and outputs, such that a representative test value can be selected for each class.

[b] This method applies to parameters or variables, values approaching and crossing the boundaries and out of range values.

[c] Error guessing tests can be based on data collected through a "lessons learned" process and expert judgment.

Challenges in Software Testing

- Testing embedded SW and systems
- SW-HW Co-Development and Co-Testing
- Testing SoC (System On Chips)
- Testing Agile SW
- Testing legacy SW with no documentation
- Continuous Testing and Integration Platform (CTIP)

Code Review

Clean Code

Refactoring

코드 기반 테스트

테스트 계획서

TDD

구조 기반 테스트

소프트웨어 테스트

Boundary Value 테스트

3점 점합

TFD

블랙박스 테스트

구조 테스트

테스트 실행

CI / CD

모델 기반 테스트

테스트 결과 보고서

N-Pairwise 테스트

테스트 Stub

기능 테스트

화이트박스 테스트

Static Code Analysis

스펙 기반 테스트

CTIP(Continuous Test & Integration Platform)

테스트 케이스 자동 생성

테스트 오라클

Code Review

테스트 케이스

Category-Partitioning 테스트

테스트 데이터

요구공학

Coverage Criteria

테스트 자동화 도구

xUnit

테스트 데이터 자동 생성

테스트 Scaffolding

시스템 테스트 계획서

테스트 드라이버

Coverage 측정

테스트 명세서

gTest

QAS

HW/SW Co-테스트

비기능/품질 테스트

Cyclomatic Complexity

시뮬레이션 기반 테스트