Software Engineering for Software-Intensive Systems: VII Verification & Validation

Stefan Henkler & Martin Hirsch
Software Engineering Group
Raum E 3.346
Tele. 60-3309 / 60-3308
[shenkler|mahirsch]@uni-paderborn.de
Outline

I Introduction
II Foundations
III The Development Life Cycle
IV Requirements
V Analysis & Design
VI Implementation
VII Verification & Validation
VIII Summary and Outlook
VII Verification & Validation

VII.1 Foundations
VII.2 Static Analysis Techniques
VII.3 Dynamic Analysis Techniques
VII.4 Summary & Discussion
VII.5 References
VII.1 Foundations

- **Verification** – The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase

  [IEEE Std 610.12-1990]

- **Validation** - The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements

  [IEEE Std 610.12-1990]
Verification & Validation

Verification: refers to the set of activities that ensure that software correctly implements a specific function.

Validation: refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements.

Boehm [Boehm81]:
- **Verification**: “Are we building the product right?”
- **Validation**: “Are we building the right product?”

The definition of V&V encompasses many of SQA activities, including
- formal technical reviews, quality and configuration audits
- performance monitoring, different types of software testing
- feasibility study and simulation
Terminology: Static/Dynamic Analysis

Static analysis:
- The process of evaluating a system or component based on its form, structure, content, or documentation.
  Contrast with: dynamic analysis.
  See also: inspection; walk-through.

Dynamic analysis:
- The process of evaluating a system or component based on its behavior during execution.
  Contrast with: static analysis.
  See also: demonstration; testing.

[IEEE-610.12-1990]
Dynamic vs. Static Analysis

Static analysis:
- investigation without operation;
- pencil and paper reviews etc.
- Modelling (mathematical representation)

Dynamic analysis (testing):
- execution of system components;
- running the software
# Quality Control Activities & Process

<table>
<thead>
<tr>
<th>Life cycle phase</th>
<th>Dynamic analysis</th>
<th>Static analysis</th>
<th>Modelling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Top-level design</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Detailed design</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Implementation</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Integration testing</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>System validation</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

[Storey1996]
VII.2 Static Analysis Techniques

Overview

- Reviews and Inspections
  - Walkthroughs, inspections, personal reviews
  - Formal technical reviews
  - Summary

- Other Techniques
  - Static Code Analysis
  - Formal Methods
  - Model Checking
Reviews and Inspections

- A family of techniques
  1. Personal reviews
  2. Inspections
  3. Walkthroughs
  4. Formal technical reviews

- Review / inspect
  - To examine closely
  - With an eye toward correction or appraisal

- People (peers) are the examiners
Purpose/Objectives

- Verify that
  - software meets its requirements
  - software follows predefined standards
  - software is developed in uniform manner

- Catching errors
  - Sooner
  - More and different
  - Breaking frame of reference

- Make projects more manageable
  - To identify new risks likely to affect the project

- Improving communication
  - Crossing organization boundaries

- Providing Education

- Making software visible
Formality of Technical Reviews

- Personal Review: individual initiative
- Walkthroughs: team-oriented
- Inspections: "Detection of flaws"
- Formal Review: "Proof of correctness"

"Improved design"
Other Techniques

- Static Code Analysis
  1. Control Flow Analysis
  2. Data Flow Analysis
  3. Information Flow Analysis
  4. Symbolic Flow Analysis
  5. Formal Code Verification
  6. Range Checking
  7. Stack Usage Analysis
  8. Timing Analysis
  9. Other Memory Usage Analysis
  10. Object Code Analysis

- Formal Methods
  - Discrete mathematics, logics, temporal logics, process algebra, ...

- Model Checking
  - Fully automatic analysis of formal models w.r.t. a formal specification

Many more, but these are the common ones
Static Code Analysis

Source Code → Compile → Compilation Environment → Link → APPLICATION SYSTEM

- Syntactic Data for Code Analysis
- Syntactic & Semantic Data for Code Analysis

Code Analysis Tool
(5) **Formal Code Verification**

- Conducted to:
  - Prove the code of a program is correct with respect to the formal specification of its requirements
  - Explore all possible program executions, which is infeasible by dynamic testing alone

- Analyses:
  - Pre-condition/Post-condition analysis
  - Demonstrate a particular safety/security property
  - Termination of all loops and any recursion
  - Proof of absence of run time errors

⇒ *Partially supported by automatic code analysis*
Timing Analysis

- Conducted to:
  - Ensure temporal properties of the input/output dependencies

- Analyses:
  - Worse Case Timing Analysis
  - Identification of infinite loops (frequently desired)

⇒ automatic code analysis possible

Certain constructs make this analysis impossible (e.g., infinite loops, manipulation of dynamic data structures)
Other Memory Usage Analysis

Conducted to:
- Ensure Memory usage does not exceed capacity

Analyses:
- Analysis of Heap Memory
- Analysis of I/O Ports
- Analysis of special purpose hardware

Automatic code analysis possible

Certain constructs make this analysis impossible (e.g., infinite loops, manipulation of dynamic data structures)
VII.2.1 Modeling

- Modeling
  - State transition diagrams (often Finite)
    - Completeness, consistency and reachability
  - Data flow diagrams
    - Errors in variable usage (written but never read;…)
  - Structure diagrams
    - Reflects relationships between program units
  - Environment modelling
    - Test system using a simulation of the operating environment
  - Software animation
    - Trace behaviour of the models (e.g., Petri Net animation tools)
  - Also formal approaches are possible
VII.2.1 Modeling (cont.)

- **Formal methods** are the use of mathematical techniques in the specification, design and analysis of computer hardware and software.
- CASE tools and graphic or diagrammatic methods to describe the requirements or specification of a system are **formalized methods**, only.

Formal methods are based on:

- Formal specifications and formal models
- Techniques to **prove** equivalence or refinement
VII.2.2 Simulation

- Simulation: “Imitation” of a operation/process
- Used in early development phase
- Types of Simulation
  - Monte Carlo Simulation
    - Using stochastic theory
  - Trace-Driven Simulation
  - Event-Driven Simulation (discrete/continuous)
    ➔ Only one execution path is simulated.
VII.2.3 Model Checking

What is Model Checking? Two answers:
... an automated proof method
... a precise system analysis tool

- For almost 50 years “program verification” has been a dream of computer science
- A rigorous proof that the program does what you say it should do
- This requires: formal semantics, specification language, and a method

“Reused” slides (partially modified/combined) from:
[Dwyer2002s] Matthew Dwyer. Kansas State University, USA Software Model Checking Tutorial, FSE’02 – Charleston, South Carolina – Nov. 19, 2002
Input-Output Patterns

Reactive programs: Carry out an ongoing interaction with their environment

Total correctness = safety + liveness + fairness …

Semantics: transition systems (Kripke structures)
Specifications: temporal logic (Pnueli, FOCS’77)
Method: manual proof (up to 1981); today model checking!

[Dwyer2002s]
Kripke Structure

Each state represents all variable values and location counters.

Each transition represents an execution step in the system.

The labels represent predicates in each state e.g. \( p \equiv (v = 5) \)

\[
K = ([p, \neg p], [x, y, z, k, h], R, \{x\}, L)
\]
Temporal Logic Model Checking

Kripke Structure

$\square (\Phi \rightarrow \Diamond \, \Omega)$

Temporal logic formula (e.g. CTL, LTL, …)

Model Checker

OK

or

Error trace

[Dwyer2002s]

Line 5: ...
Line 12: ...
Line 15: ...
Line 21: ...
Line 25: ...
Line 27: ...
...
Line 41: ...
Line 47: ...
Explicit vs. Symbolic Model Checking

Explicit State

Symbolic

- on-the-fly state generation
- works for complex transition relations

- fixed-point computation with efficient encoding of state sets
- BDDs or Boolean formulas (SAT)

[Dwyer2002s]
Why Model Check Software? [Dwyer2002s]

- Works for hardware: Intel, IBM, Motorola ... now employ hundreds of model checking experts
- Existing assurance methods aren’t good enough
  - Testing for concurrency-related defects is not effective
  - Inspections are difficult to scale
  - … need something that is automatic and thorough
- First some terminology:
  - Model checking of software (directly)
  - Model checking of models derived from software systems (usually using abstraction)
Specifications are partial
- They do not define complete functional correctness
- Focus on crucial properties

Cost of checking is enormous
- Must approximate
- What kinds of approximation are useful?
Model Checking vs. Static Analysis

When I use a model checker, it runs and runs for ever and never comes back… when I use a static analysis tool, it comes back immediately and says “I don’t know”

- Patrick Cousot

- Reasoning about sequencing properties requires (at least) flow sensitive approximations

- There are qualitative differences between approximations

[Dwyer2002s]
Abstract Software Model Checking

[\Box (\Phi \rightarrow \Diamond \Omega)]

Software model

Correctness property

Checker

Approximation

No false positives

OK

No false negatives

Error trace

[Dwyer2002s]
Why is model-checking software difficult?

[Dwyer2002s]

Problems using existing checkers:

- Model construction
- Property specification
- State explosion
- Output interpretation
Model Construction Problem

[Dwyer2002s]

void add(Object o) {
    buffer[head] = o;
    head = (head+1)%size;
}

Object take() {
    tail=(tail+1)%size;
    return buffer[tail];
}

Program

Model Description

Model Checker

Semantic gap:

- Programming Languages with *methods*, *inheritance*, *dynamic creation*, *exceptions*, etc.
- Model Description Languages are only *automata*
Property Specification Problem

- Difficult to formalize a requirement in temporal logic

"Between the window open and the window close, button X can be pushed at most twice."

...is rendered in LTL as...

```plaintext
[]((open ∧ <>close) →
  (!pushX ∧ !close) ∨
  (close ∨ (!pushX ∧ !close) ∨
   (close ∨ (!pushX ∧ !close) ∨
    (close ∨ (!pushX ∧ !close) ∨
     (close ∨ (!pushX ∨ close))))))
```
State Explosion Problem

- Cost is exponential in the number of components

  Bit $x_1, \ldots, x_N \rightarrow 2^N$ states

- Moore’s law and algorithm advances can help
  - Holzmann: 7 days (1980) $\Rightarrow$ 7 seconds (2000)

- Explosive state growth in software
  - 10^6 LOC embedded systems, 500 KLOC device driver

- Are we even gaining on the problem?
Output Interpretation Problem

Raw error trace may be 1000’s of steps long

- Must map line listing onto model description
- Mapping to source is made difficult by
  - Semantic gap & clever encodings of complex features
  - multiple optimizations and transformations

void add(Object o) {
    buffer[head] = o;
    head = (head+1)%size;
}

object take() {
    tail=(tail+1)%size;
    return buffer[tail];
}
Input Languages for MC

- Synchronous Languages
- Finite Automata/State Machines
- Timed Automata
- Hybrid Automata
Model Checking: Summary

- Instead of a „complete“ specification use only one that consists of relevant properties (e.g., for safety)
- Usually only restricted notions for formal models
  - Finite automata (or similar restricted models)
- Often restricted notions for formal properties
  - Prepositional logic
  - Temporal logic (Model-Checking)

Benefits:
- Counterexample when a property is not fulfilled

Limitations:
- Not feasible for too large models (state explosion)
- Not feasible for too complex formal properties
VII.3 Dynamic Analysis

- VII.3.1 Testing
  - Objectives
  - Fundamentals
  - Techniques
  - Process

- VII.3.2 Testing Software Intensive Systems
  - Embedded Systems
VII.3.1 Testing

Common definition [Myers1979]:

- Testing is a process of executing a program with the intent of finding an error.
- A good test case is one that has high probability of finding an undiscovered error.
- A successful test is one that uncovers an as-yet undiscovered error.

Also: testing to assess reliability
(not considered here)

Test:

1. An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.
2. To conduct an activity as in (1).
3. (IEEE Std 829-1983 [5]) A set of one or more test cases.
4. (IEEE Std 829-1983 [5]) A set of one or more test procedures.
5. (IEEE Std 829-1983 [5]) A set of one or more test cases and procedures. [IEEE-610.12-1990]
Objectives

- Design tests that **systematically uncover defects**
  - Maximize bug count: uncover as many defects (or bugs) as possible
  - Using the minimum cost and efforts, within the limits budgetary/scheduling
  - Guide correction such that an acceptable level of quality is reached
- Help managers **make ship / no-ship decisions**
  - Assess quality (depends on the nature of the product)
  - Block premature product releases; Minimize technical support costs
  - It is impossible to verify correctness of the product by testing, but you can prove that the product is not correct or you can demonstrate that you didn’t find any errors in a given time period.
- **Minimize risks** (especially safety-related lawsuit risk):
  - Assess conformance to specification
  - Conform to regulations
- Find safe **scenarios for use** (find ways to get it to work, in spite of the bugs)

**Indirect objectives**

- To compile a **record of software defects** for use in error prevention (by corrective and preventive actions)
Testing Fundamentals

- Software Testing is a critical element of Software Quality Assurance
  - It represents the ultimate review of the requirements specification, the design, and the code.
  - It is the most widely used method to insure software quality.
  - Many organizations spend 40-50% of development time in testing.

- Testing is the one step in software engineering process that could be viewed as destructive rather than constructive.
  - “A successful test is one that breaks the software.” [McConnell 1993]
  - A successful test is one that uncovers an as yet undiscovered defect.
  - Testing can not show the absence of defects, it can only show that software defects are present.
  - For most software exhaustive testing is not possible.
Model of Testing

- Input test data
- I
- Inputs causing anomalous behaviour
- System
- O_e
- Outputs which reveal the presence of defects
Testing Approaches

Black box testing

1. Testing that ignores the internal mechanism of the system or component and focuses solely on the outputs in response to selected inputs and execution conditions

2. Testing conducted to evaluate the compliance of a system or component with specified functional requirements (sometimes named functional testing)

White box testing

Testing that takes into account the internal mechanism of a system or component (sometimes named structural testing)
III.3.1.1 Black Box Testing Techniques

- An approach to testing where the program is considered as a “black-box”

**Black box testing:**

1. Function testing
2. Domain testing
3. Specification-based testing
4. Risk-based testing
5. Stress testing
6. Regression testing
7. User testing
8. Scenario testing
9. State-model based testing
10. High volume automated testing
11. Exploratory testing
(1) Function Testing

- Test each function / feature / variable in isolation.
- Usually start with fairly simple function testing
  - focuses on a single function and tests it with middle-of-the-road values
  - don’t expect the program to fail
  - but if it fails the algorithm is fundamentally wrong, the build is broken, or …
- Later switch to a different style which often involves the interaction of several functions
- These tests are highly credible and easy to evaluate but not particularly powerful
(6) Regression Testing

- **Process**
  1. design tests with the intent of regularly reusing them
  2. repeat the tests after making changes to the program.

- **A good regression test is designed for reuse.**
  - adequately documented and maintainable
  - designed to be likely to fail if changes induce errors in the addressed part of the program

- **Discussion:**
  - First run of the tests may have been powerful, but after passed many times detecting defects is not very likely (unless there have been major changes or changes in part of the code directly involved with this test.) ⇒ Usually carries little information value
  - Automation can make regression testing very “cheap”
Regression Testing Process

Requirements changes
- Affect design, coding, and testing document update

Design changes
- Affect coding, tests, associated components, system architecture, related component interactions

Implementation changes
- Affect test cases, test data, test scripts, test specification (see also code change impact)

Test changes
- Affect other tests and test documentation

Document changes
- Affect other document
Regression Testing Strategies

- **Random**: The test cases are randomly selected from the existing test suite.
- **Retest-all**: Run all the tests in the existing suite.
- **Safe**: The test selection algorithm excludes no test from the original test suite that if executed would reveal faults in the modified program.
  - **Based on modifications**: Place an emphasis on selecting existing test cases to cover modified program components and those that may be affected by the modifications.
  - **Dataflow/coverage based**: Select tests that exercise data interactions that have been affected by modifications.

**Remark ([Juriso+2004])**:
- For small programs and set of test cases is small, then retest-all.
- Use safe techniques for large programs and programs with a large number of test cases.
VII.3.1.2 White Box Testing Techniques

- Black box testing:
  - Requirements fulfilled
  - Interfaces available and working
- But can we also exploit
  - the internal structure of a component,
  - interactions between objects?
  → white box testing

White box testing:
1. Control flow testing
2. Data flow testing
3. Mutation testing
(1) Control Flow Testing

a) **Statement coverage**: The test cases are generated so that all the program statements are executed at least once.

b) **Decision coverage** (branch coverage): The test cases are generated so that the program decisions take the value true or false.

c) **Condition coverage**: The test cases are generated so that all the program conditions (predicates) that form the logical expression of the decision take the value true or false.

d) **Path coverage**: Test cases are generated to execute all/some program paths.
(2) Data Flow Testing

Basic Definitions:

\[ V \] = the set of variables.
\[ N \] = the set of nodes
\[ E \] = the set of edges

**Definition/write** on variables (nodes):

\[ \text{def}(i) = \{ x \in V | x \text{ has a global def in block } i \} \]

**Computational-use** of variables (nodes):

\[ \text{c-use}(i) = \{ x \in V | x \text{ has a global c-use in block } i \} \]

**Predicate-use** of variables (edges):

\[ \text{p-use}(i,j) = \{ x \in V | x \text{ has a p-use in edge } (i,j) \} \]
VII.3.1.3 Comparison

Black box tests:
- tester has no access to information about the system implementation

Characteristics:
- Good for independence of tester
- But not good for formative tests
- Hard to test individual modules...

White box tests:
- tester can access information about the system implementation

Characteristics:
- Simplifies diagnosis of results
- Can compromise independence
- How much do they need to know?
VII.3.1.4 Testing Process & Activities

Beforehand:
- Requirement analysis
- Design analysis

Testing Process & Activities
(1) Unit test
(2) Integration test
(3) Function test
(4) Performance test
(5) Acceptance test
(6) Installation test

System test
Testing Activities

Unit code → Unit test

Design specifications
System functional requirements
Other software requirements
Customer requirements specification
User environment

Integration test → Function test → Performance test → Acceptance test → Installation test

Integrated modules → Functioning system → Verified, validated software → Accepted system

SYSTEM IN USE!

[Pfleeger 2001]
(1) Unit Testing

- Individual components are tested independently to ensure their quality. The focus is to uncover errors in design and implementation, including:
  - data structure in a component
  - program logic and program structure in a component
  - component interface
  - functions and operations of a component
- There is some debate about what constitutes a “unit”. Some common definitions:
  - the smallest chunk that can be compiled by itself
  - a stand alone procedure of function
  - something so small that it would be developed by a single person

Unit testers:

- developers of the components.
(2) Integration Testing

- A group of dependent components are tested together to ensure their the quality of their integration unit.
- The focus is to uncover errors in:
  - Design and construction of software architecture
  - Integrated functions or operations at sub-system level
  - Interfaces and interactions between them
  - Resource integration and/or environment integration

Integration testers:
- either developers and/or test engineers.

Tests complete systems or subsystems composed of integrated components
- Integration testing should be black-box testing with tests derived from the specification
- Main difficulty is localising errors
- Incremental integration testing reduces this problem
Integration Testing Strategies

Incremental testing strategies:

- **Bottom-up testing**: Integrate individual components in levels until the complete system is created
- **Top-down testing**: Start with high-level system and integrate from the top-down replacing individual components by stubs where appropriate
- **Outside-in integration**: Do bottom-up and top-down testing at the same time such that the final integration step occurs in a middle layer

Non incremental testing strategies:

- **Big bang testing**: put all together and test it

**Remark**: In practice, most integration involves a combination of these strategies
(3) Function Testing

- The integrated software is tested based on requirements to ensure that we have a right product (validate functional requirements).

**The focus is to uncover errors in:**
- System input/output
- System functions and information data
- System interfaces with external parts
- User interfaces
- System behavior

**Function testers:**
- test engineers or SQA people.
VII.3.2 Testing Software Intensive Systems

- Software Intensive Systems include:
  - Large scale heterogenous systems
  - Embedded systems for automotive applications
  - Telecommunications
  - Wireless ad-hoc systems
  - Business applications with an emphasis on web services

- Testing embedded systems
  - What is an embedded system?
  - What is the correct behavior?
  - Development Stages
    - Simulation
    - Prototyping
    - Pre-Production
    - Production

[Broekman&Notenboom2003]
What is an Embedded System?

- Software **embedded** into a **technical system** that interacts with the real physical world, controlling some specific hardware.
- Interaction with the technical system via **actors** and **sensors**

[Broekman&Notenboom2003]
What is the Correct Behavior?

- Continuous and discrete signals in the
  - **Time** domain
  - **Value** domain
- Exact match is not realistic ⇒ **tolerance** in value and time

[Broekman&Notenboom2003]
Different Development Stages

(1) Simulation
(2) Prototyping
(3) Pre-Production
(4) Production

MT = model test
MiL = model-in-the-loop
RP = rapid prototyping
SiL = software-in-the-loop
HiL = hardware-in-the-loop
ST = system test

[Broekman&Notenboom2003]
(1) Simulation

One-way simulation (model test)
- Test plant or embedded software model

Feedback-simulation (model-in-the-loop)
- Let the models of the plant and the embedded software interact

Rapid Prototyping
- Let the “real” plant (or a simplified version) interact with the model of the embedded software (on high performance hardware; floating-point arithmetic)

[Broekman&Notenboom2003]
(2) Prototyping (1/3)

(1) **Embedded software**
- Compiled for host
- Compiled for target platform

(2) **Processor**
- High performance host
- Target platform emulator

(3) **Rest of the embedded system**
- Simulated on the host
- Experimental hardware platform
- Prototype hardware platform

(4) **Plant**
- Statically (signal generator)
- Dynamically (simulation model)

[Broekman & Notenboom 2003]
Prototyping (2/3)

(1) Software unit and software integration test
- Same environment as MIL
- Final software instead of model
  a) Host environment
  b) Target (emulator)

(2) Hardware/software integration test
- Test hardware integration
  - Signal processors
  - Output monitoring with oscilloscopes or logic analyzers
  - In-circuit test equipment (not restricted to outputs)
  - Real-time simulation of the plant
Prototyping (3/3) [Broekman&Notenboom2003]

(3) System integration test
- Integrate all hardware parts (often prototypes)
- Similar to hardware integration (but complete!)

(4) Environmental test
- Measuring influence of the system on the environment (EMC)
- Measuring robustness of the system w.r.t. environment conditions (temperature, humidity, EMI, shock or vibration, ...)

<table>
<thead>
<tr>
<th></th>
<th>Embedded software</th>
<th>Processor</th>
<th>Rest of embedded system</th>
<th>Plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW/U, SW/I (1)</td>
<td>Experimental (host)</td>
<td>Host</td>
<td>Simulated</td>
<td>Simulated</td>
</tr>
<tr>
<td>SW/U, SW/I (2)</td>
<td>Real (target)</td>
<td>Emulator</td>
<td>Simulated</td>
<td>Simulated</td>
</tr>
<tr>
<td>HW/SW/I</td>
<td>Real (target)</td>
<td>Real (target)</td>
<td>Experimental</td>
<td>Simulated</td>
</tr>
<tr>
<td>System integration</td>
<td>Real (target)</td>
<td>Real (target)</td>
<td>Prototype</td>
<td>Simulated</td>
</tr>
<tr>
<td>Environmental</td>
<td>Real (target)</td>
<td>Real (target)</td>
<td>Real</td>
<td>Simulated</td>
</tr>
</tbody>
</table>
Pre-Production

Test a pre-production unit
- Use pre-production unit
- Execute with the real plant

Goals:
- Prove that all requirements are met
- Demonstrate conformity with relevant standards/norms
- Demonstrate that production effort and schedule can be met
- Demonstrate maintainability and mean-time-to-repair requirements
- Demonstrate product to (potential) customers

Applicable tests:
- Acceptance test
- Qualification test
- Safety execution test
- Test of the production and maintenance test facilities
- Inspection and/or test by government officials

Typical test techniques:
- Real-life testing
- Random testing
- Fault injection

Remark:
- Special tools for gathering and recording the data as well as fault injection are required
- Often environment test facilities (climate chambers, vibration tables, …) are also used
(4) Production

Production (or Post-development stage):

- Development and test of the production facilities (often equally important for quality)
- Often a first article inspection of similar style as the pre-production test is required
- Production and maintenance tests on production units may be required for quality control

[Broekman&Notenboom2003]
# Comparison of the Test Levels

<table>
<thead>
<tr>
<th>Test levels</th>
<th>Embedded software</th>
<th>Processor</th>
<th>Rest of embedded system</th>
<th>Plant system</th>
</tr>
</thead>
<tbody>
<tr>
<td>One-way simulation</td>
<td>MT, Simulated</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Feed-back simulation</td>
<td>MiL, Simulated</td>
<td>–</td>
<td>–</td>
<td>Simulated</td>
</tr>
<tr>
<td>Rapid prototyping</td>
<td>RP, Experimental</td>
<td>Experimental</td>
<td>Experimental</td>
<td>Real</td>
</tr>
<tr>
<td>SW/U, SW/I (1)</td>
<td>SiL, Experimental (host)</td>
<td>Host</td>
<td>Simulated</td>
<td>Simulated</td>
</tr>
<tr>
<td>SW/U, SW/I (2)</td>
<td>SiL, Real (target)</td>
<td>Emulator</td>
<td>Simulated</td>
<td>Simulated</td>
</tr>
<tr>
<td>HW/SW/I</td>
<td>HiL, Real (target)</td>
<td>Real (target)</td>
<td>Experimental</td>
<td>Simulated</td>
</tr>
<tr>
<td>System integration</td>
<td>HiL, Real (target)</td>
<td>Real (target)</td>
<td>Prototype</td>
<td>Simulated</td>
</tr>
<tr>
<td>Environmental</td>
<td>HiL/ST, Real (target)</td>
<td>Real (target)</td>
<td>Real</td>
<td>Simulated</td>
</tr>
<tr>
<td>Pre-production</td>
<td>ST, Real (target)</td>
<td>Real (target)</td>
<td>Real</td>
<td>Real</td>
</tr>
</tbody>
</table>

[Broekman&Notenboom2003]
VII.3.3 Summary

- Two main approaches to testing (dynamic analysis) are **black box testing**, which ignores the internal mechanism of the system or component (sometimes named functional testing), or **white box testing**, which takes into account the internal mechanism of a system or component (sometimes named structural testing).

- The testing activities during development should include, **unit tests**, **integration tests**, and **system tests**. System tests may be divided into function tests, performance tests, acceptance tests, and installation tests.

- Testing embedded systems for automotive application (SIS) includes testing processes like Model-/Software-/Hardware-in-the-loop at different development stages.
Besides **dynamic analysis** (testing), which includes the execution of the software, also effective **static analysis** techniques such as reviews or inspections and formal methods are available.

The complexity and properties (e.g. safety) of software intensive systems require exhaustive verification and validation activities.

Due to the lack of one technique (e.g. Model Checking, Testing) an appropriate combination of the V&V techniques is needed.
## VII.5 Bibliography (1/1)

<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
<th>Editions</th>
</tr>
</thead>
</table>