On ESA Flight Software Testing and Its Independent Verification

Marek Prochazka & Maria Hernek, European Space Agency

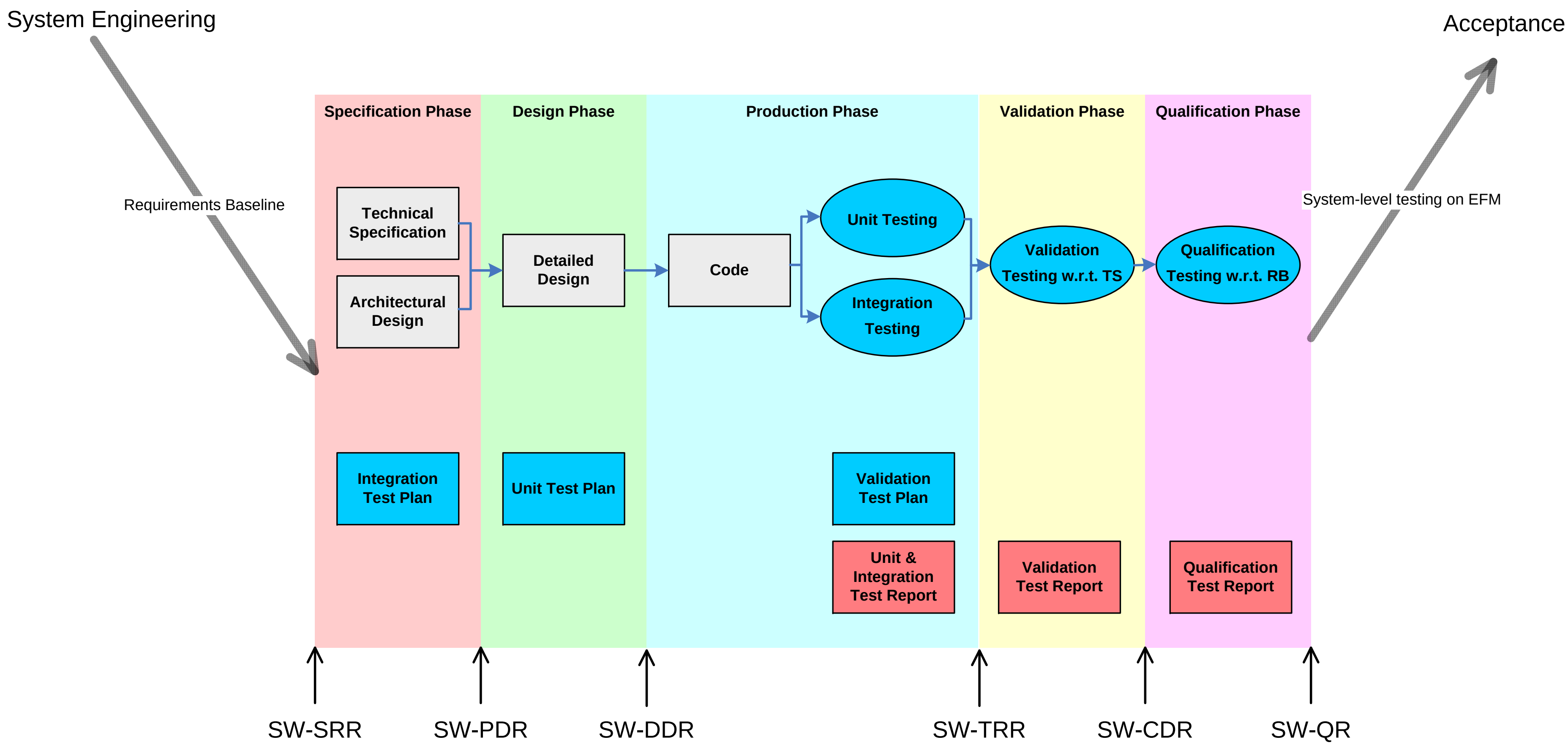
NASA IV&V Workshop

13-15 September 2011

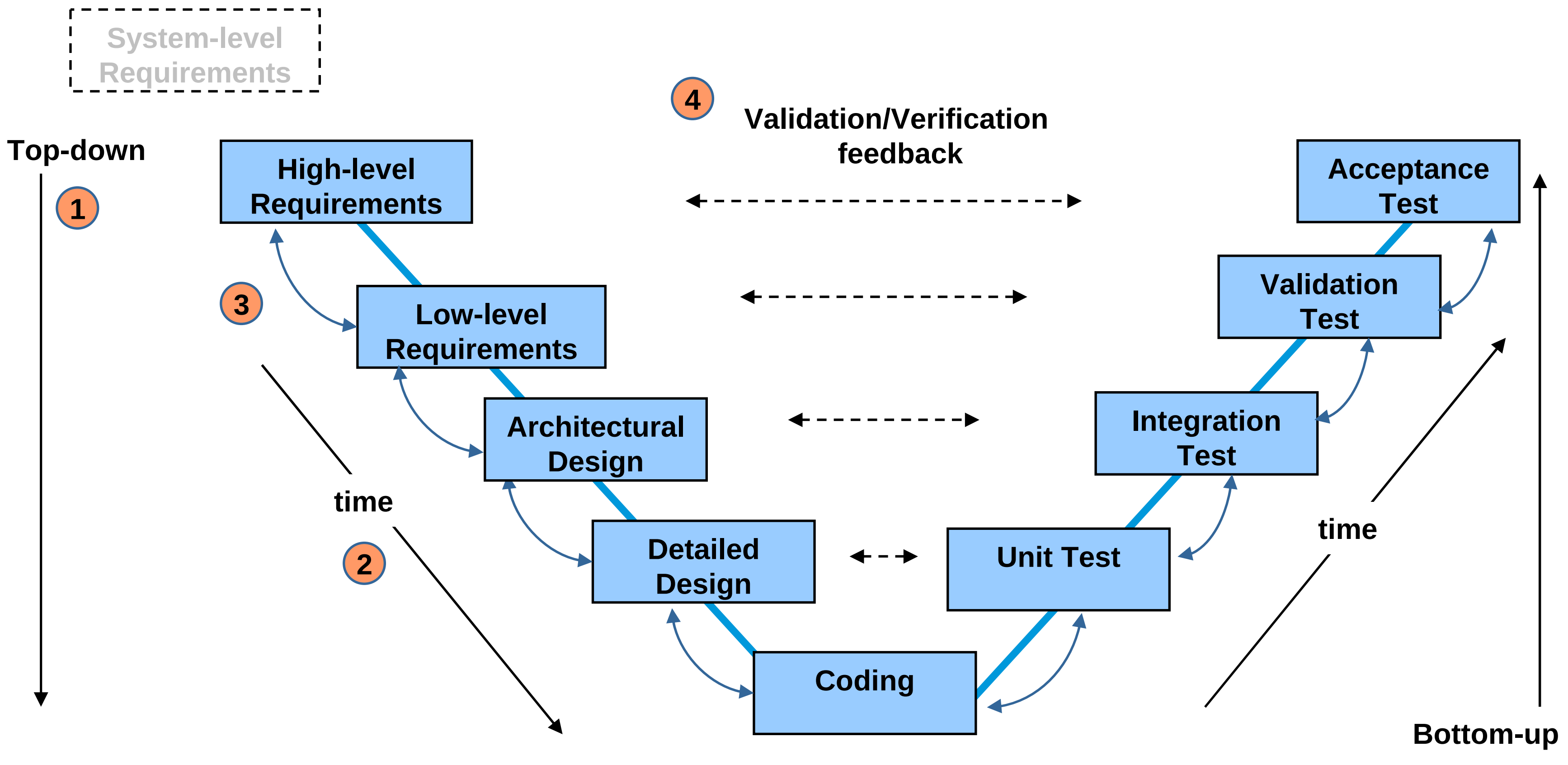
OUTLINE

- ESA flight software development lifecycle and the role of testing
  - Unit, integration and validation tests
  - Versioning approach to FSW development
- Testing in the scope of ISVV
  - Tests verification
  - Independent validation
- ESA Avionics Test Bench
- Advanced techniques to reduce testing effort
  - Automatic test generation
- Conclusions

PHASES OF FSW TESTING



PHASES OF FSW TESTING: THE V MODEL



UNIT TESTING

Testing of individual software units

- **Objectives: White box testing during code production phase**
  1) Exercise code using boundaries at n-1, n, n+1 including looping instructions, while, for and tests that use comparisons
  2) All the messages and error cases defined in the design document
  3) Access to all global variables as specified in the design document
  4) Out of range values for input data, including values that can cause erroneous results in mathematical functions
  5) Software at the limits of its requirements (stress testing)
- **Metrics:**

Code coverage versus criticality	A	B	C	D
Source code statement coverage	100%	100%	0-100%	0-100%
Source code decision coverage	100%	100%	0-100%	0-100%
Source code modified condition and decision coverage	100%	0-100%	0-100%	0-100%
Object code coverage	100%	N/A	N/A	N/A

INTEGRATION TESTING

Testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them

- **Objectives:**
  1) Problems in integrated FSW found early in the lifecycle
  2) Functional interfaces between all integrated components are tested

- **Metrics:**
  - 100% interface coverage for Cat A, B

Testing w.r.t. Technical Specification and Requirements Baseline requirements

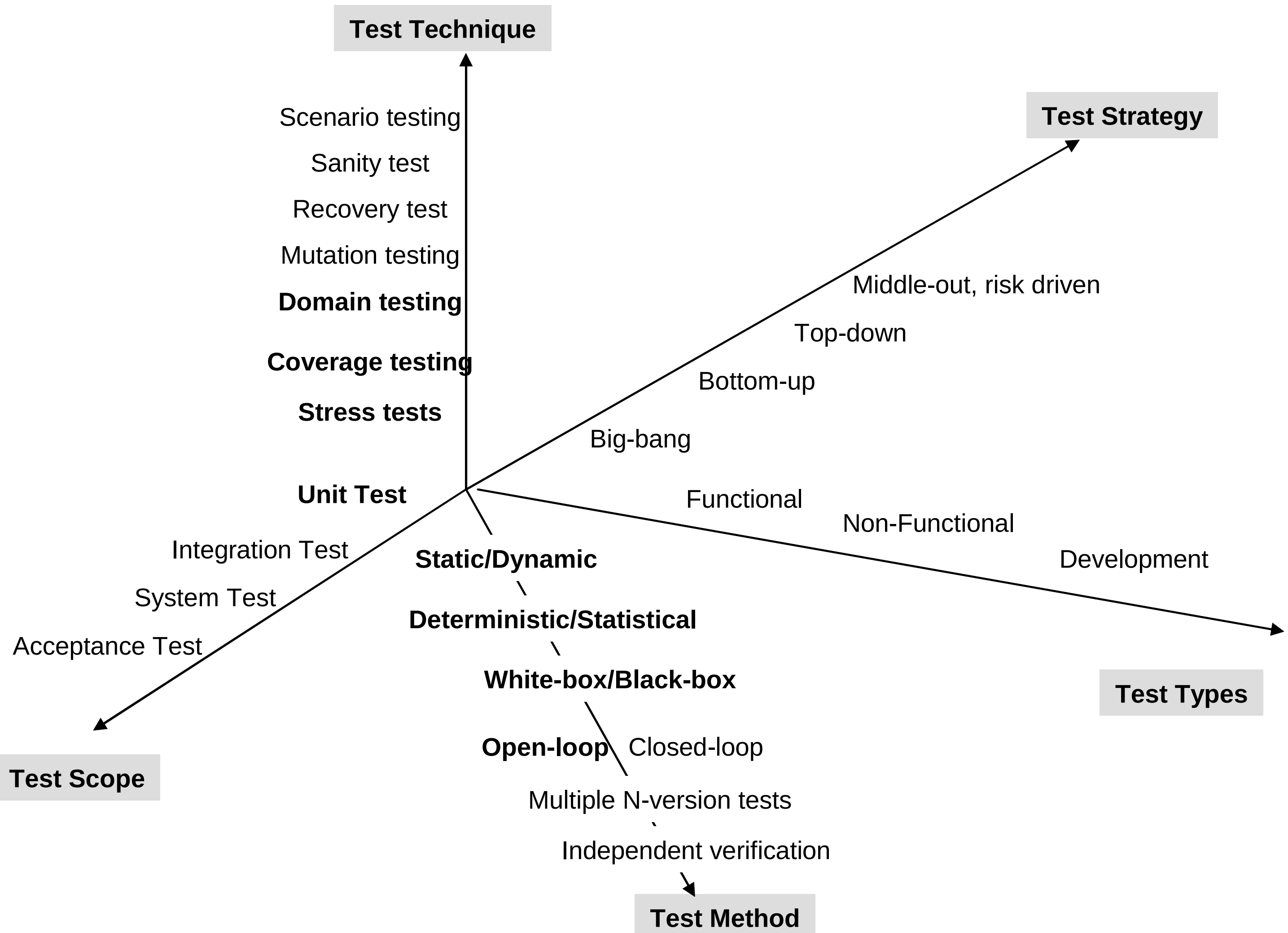
- **Objective:** Verify that all requirements are met

- **Metrics:**
  - 100% RB and TS requirements coverage
  - Traceability matrices between requirements and tests and vice versa

FURTHER TESTING

- **Stress testing**
  - Evaluates a system or software component at or beyond its required capabilities (e.g. TM/TC frequency)
  - Used at all levels (unit, integration, validation)
- **Robustness testing**
  - Testing beyond specification (dependability, resource sharing, division by zero, pointers, run-time errors, etc.)
  - Used at all levels (unit, integration, validation)
- **Alternative verification means**
  - When full validation on the target computer is not feasible or where performance goals are difficult to achieve
  - Only if it can be <u>justified</u> that validation by test cannot be performed, it is performed by either analysis, inspection or review of design
  - Use static analysis for errors that are difficult to detect at run-time
  - Use models & proofs
    - An example: Schedulability analysis
      - Mathematical model
      - Worst-case execution time (WCET)
      - Application of heavy stress scenario

DIMENSIONS OF TESTING

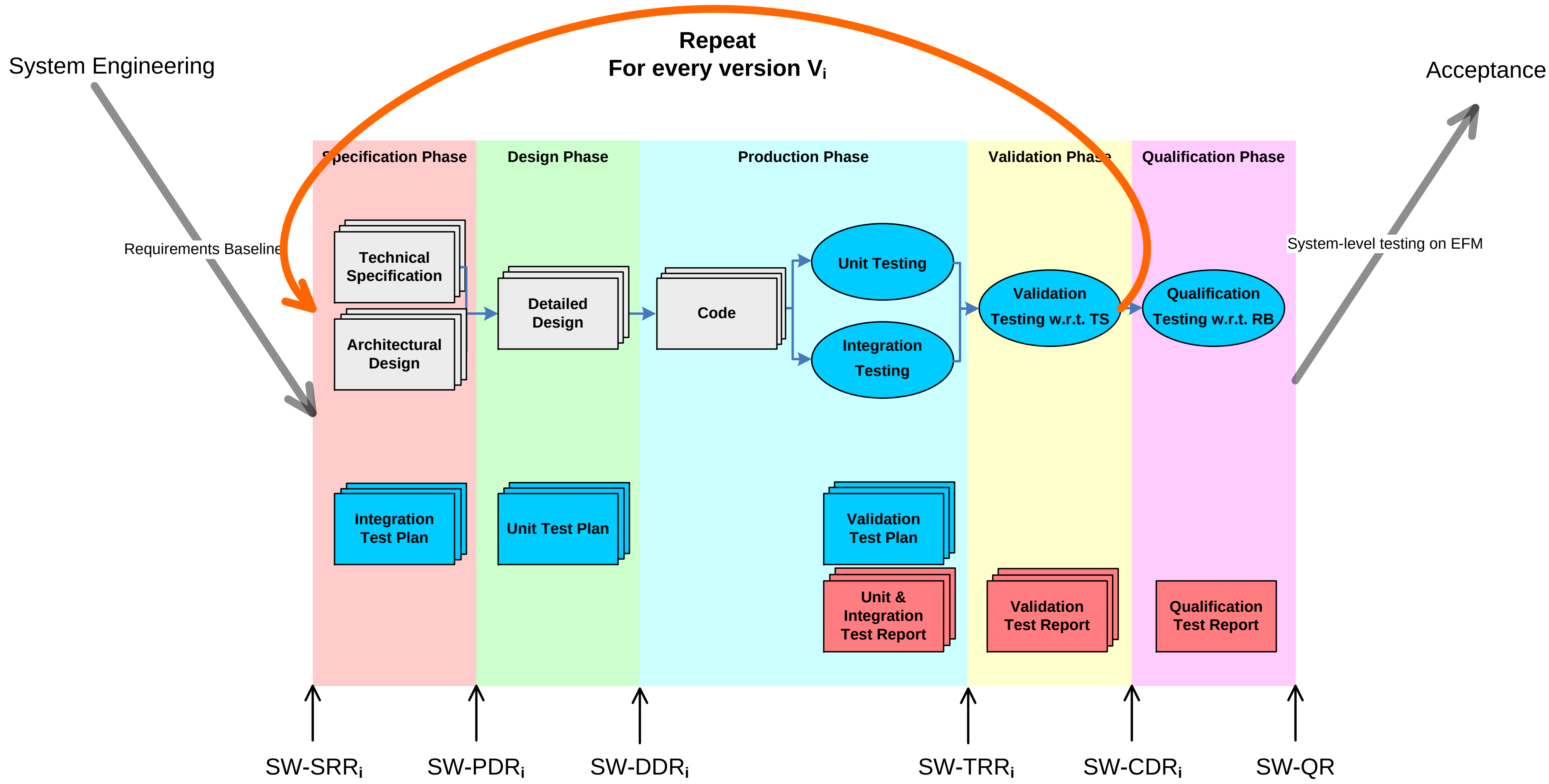


PHASES OF FSW TESTING REVISITED

Getting things more complicated: Versioning of FSW

- **Introduced to enable early releases of well-defined versions to AIT**
- **Incrementally built versions reflecting system development**
- **A typical FSW versioning:**
  - V1: System startup and initialisation, Basic Software (operating system, BSP, hardware drivers), elementary TM/TC handling, simple interaction with equipments (switch on/off), no AOCS
  - V2: Full data handling, AOCS Initial Acquisition Mode and Safe Mode allowing to perform close-loop tests, full interaction with equipment
  - V3: Full AOCS (all modes), FDIR, monitoring, reconfiguration sequences
- **Gains in the schedule of system development** due to parallelisation of AIT and further software development
- **But the software development effort increases!**
  - More versions, more test effort

PHASES OF FSW TESTING REVISITED



CAREFUL RISK ASSESSMENT

ESA is constantly re-assessing its processes, methods and standards…

- **Different approaches to testing for non-nominal software lifecycle**
  - E.g. to which extent is unit testing needed for reverse engineering?

- **Examples of <u>non-agreed</u> recent tendencies to relax the testing burden**
  1) **Achieving code coverage by a combination of validation and integration / unit testing**
     - Increases the risk that bugs are found too late
  2) **Interface coverage as part of the validation test campaign**
     - Two executions of the test suite would be necessary (code instrumentation)
     - Too late, defeats one of the two objectives of integration tests
     - Not testing interfaces & out-of value parameters which are avoided by nominal operational scenarios (reflected in validation tests)
     - What in case that software is patched or nominal behaviour is changed?
  3) **Very limited or no integration tests**
     - Unit tests cover all interfaces anyway?

WHY INTEGRATION TESTING IS NEEDED

Unit tests also examine all interfaces & out-of-boundary values, so why repeating this effort again?

- **Consider the following examples:**

  Assumptions:

  - Unit X is unit tested, 100% code coverage
  - Unit Y is unit tested, 100% code coverage
  - Unit Y is integrated with unit X

  1) Missing functionality example
     - Y requires one more interface, not provided by X!
  2) Mismatch in communication protocol example
     - X expects init() called first, and other methods called only after this
     - However Y does not call init() before calling other methods!

- **Bottom line: Integration tests also address interface completeness, semantics and the protocols of interactions between units**

PROJECT-SPECIFIC TESTING PLATFORMS

Testing facility	CPU	Central flight computer	Equipment & Environment	Loop	Applicable Tests
Unit Test Bench	Simulated	Simulated	Simulated	N/A	Unit testing
Software Validation Facility	Simulated	Simulated	Simulated	Open Closed	SW-SW integration testing Validation testing AOCS
Software Test Bench	Target Processor	Breadboard	None Simulated	Open	SW-HW integration testing Validation testing AOCS
Functional Validation Test Bench	Simulated	None	Simulated Models	Closed	AOCS
Electrical Functional Model	Real	Real	Simulated Models	Closed	Qualification testing Acceptance testing

SYSTEM-LEVEL TESTING

- **Electrical Functional Model (EFM) is further used for System-level testing**

- **Operations validation is performed at system level after the software acceptance**
  - An option is an early System Validation Test with an earlier version of software

TESTING ACTIVITIES IN ISVV

There are several activities dedicated to testing defined in the ESA ISVV Guide

- **Independent Test Verfification**
  - Integration Test Specification and Test Data Verification
    - "Integration testing" here includes validation testing
  - Unit Test Procedures and Test Data Verification
- **Independent Validation**

INDEPENDENT TEST VERIFICATION

- **Integration Test Specification and Test Data Verification**
  - Verify consistency with Technical Spec and Architectural Design
  - Verify Test Procedures correctness and completeness (are tests testing what they are expected to test?)
  - Verify models
  - Test reports verification

- **Unit Test Procedures and Test Data Verification**
  - Verify consistency with Detailed Design
  - Verify Test Procedures correctness and completeness (are test testing what they are expected to test?)
  - Test reports verification
  - Verify models (automatic code generation)

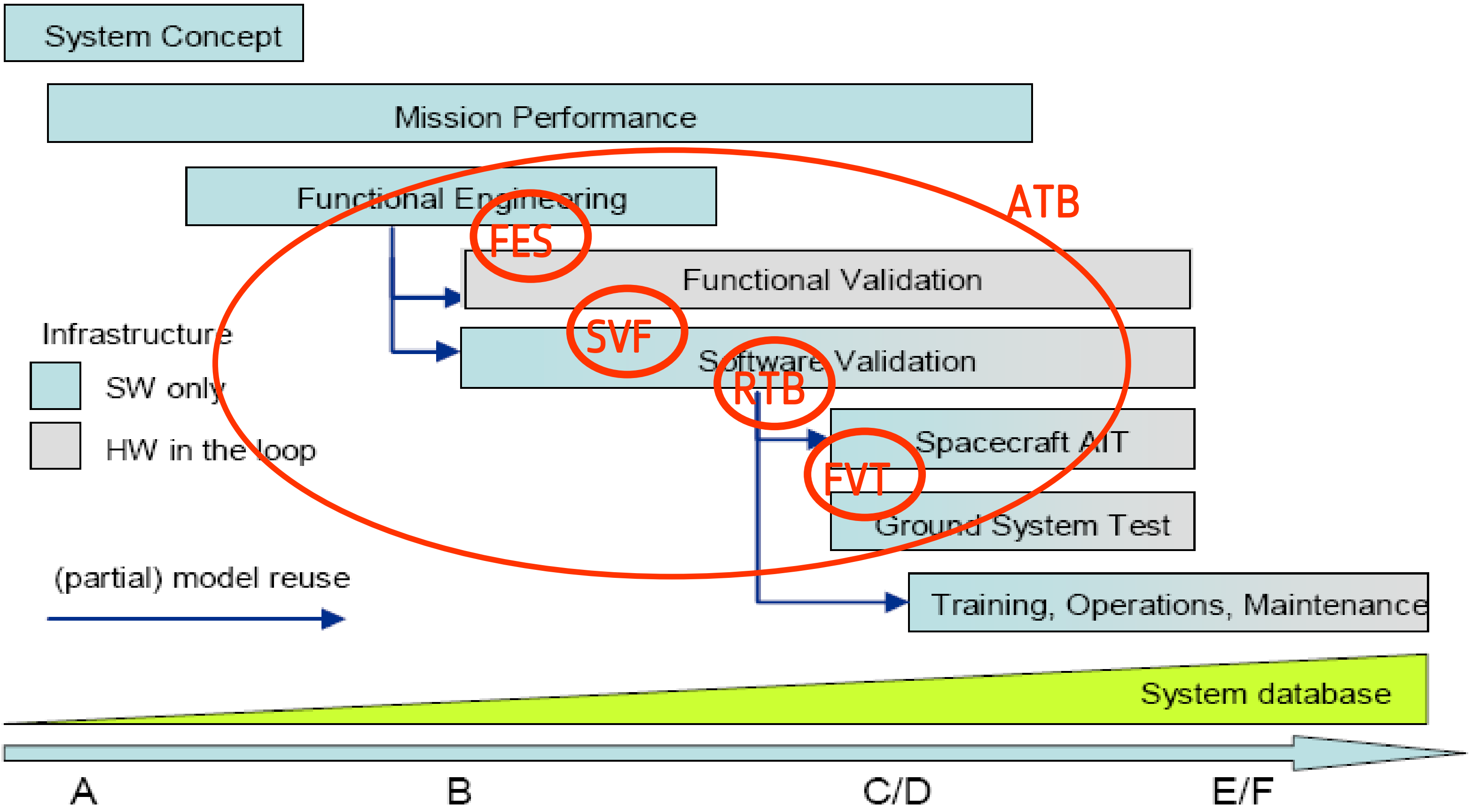
INDEPENDENT VALIDATION

- **Tasks:**
  - Identification of test cases
    - Complementary tests
    - Non-nominal test scenarios (e.g. FDIR, worst case)
  - Independent validation test plan, test procedures, test scripts
  - Independent validation test execution
  - Independent validation test reports
  - Analyse failed test cases

- **Platform used for independent validation: Software Validation Facility**
  - Nominal project-specific SVF
  - A generic SVF

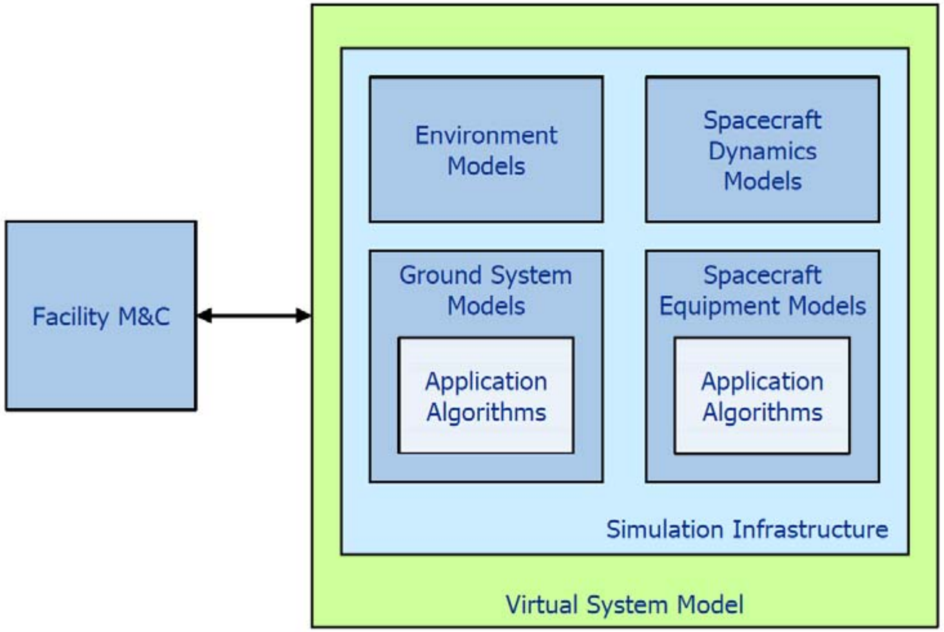
ESA AVIONICS TEST BENCH (ATB)

The ATB is an open facility that allows for technology and standards assessment, validation and demonstration in simulated environment



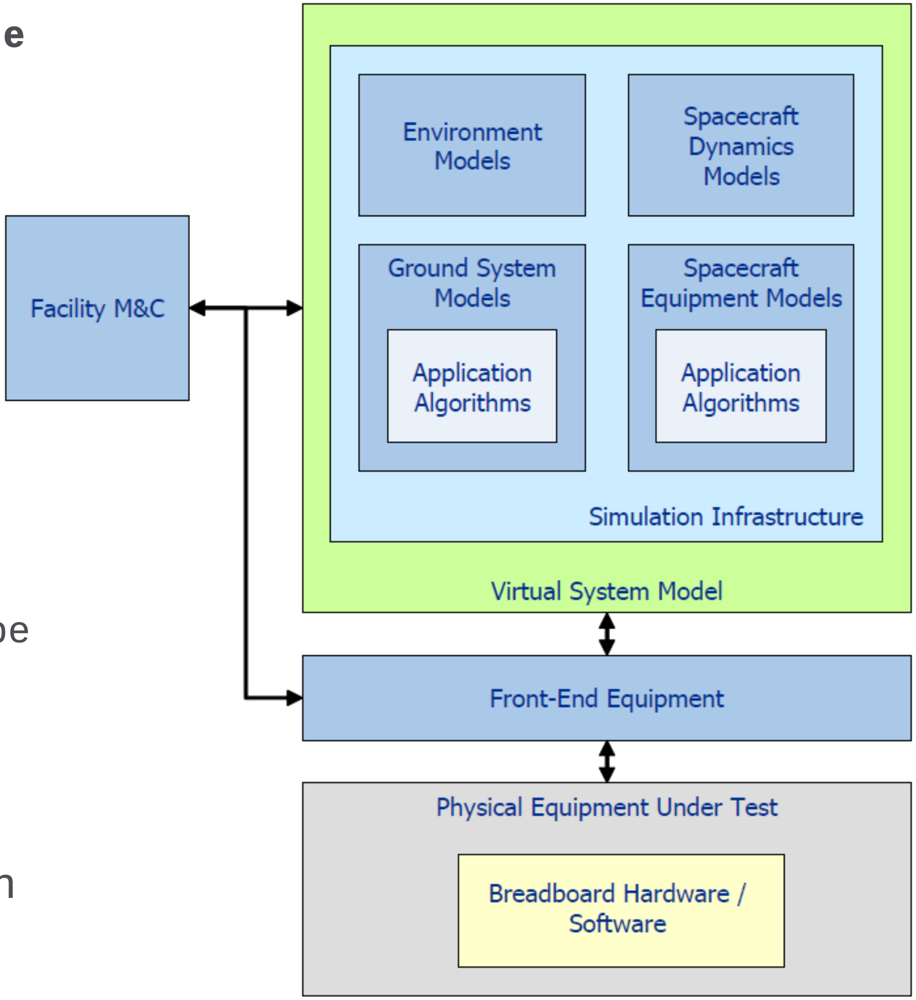
FUNCTIONAL ENGINEERING SIMULATOR

- **Verification of critical elements of baseline system design**
- **Building blocks:**
  - Models of sensors/actuators, spacecraft kinematics, dynamics and environment
  - Simulation infrastructure (e.g. Matlab/Simulink, Eurosim)
  - Monitoring and control
- **Use cases:**
  - Mission analysis verification
  - (Sub)system verification
  - Equipment modelling
  - Functional modelling
  - Demonstration of autonomous mission management



FUNCTIONAL VALIDATION BENCH

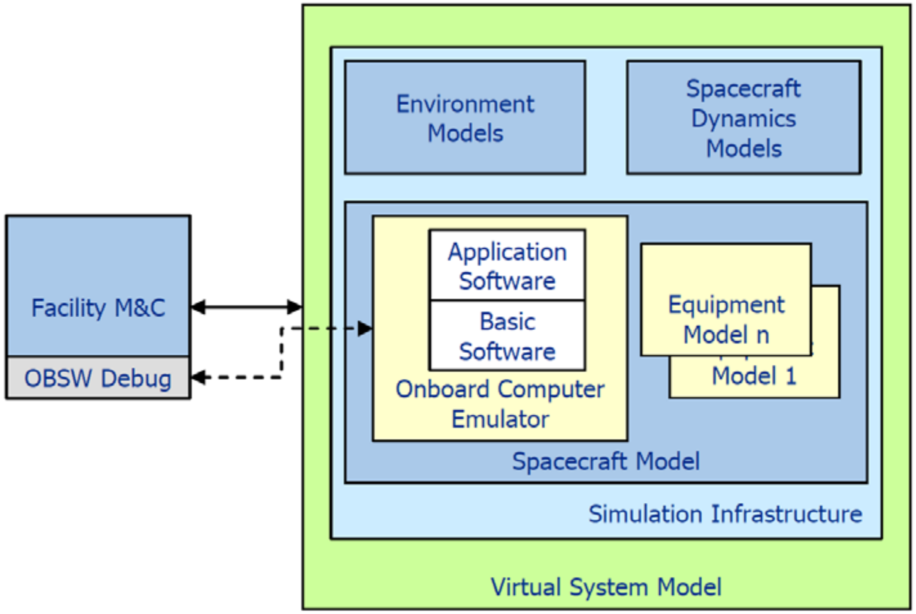
- **Test bench for hardware and software prototyping**
  - Performance analysis and validation of critical elements / subsystems (e.g. AOCS)
- **Building blocks:**
  - Simulation infrastructure
  - Monitoring and Control
  - Functional models
  - Models for protocols and electrical interfaces
  - Support equipment for prototype / breadboard under test
  - Prototype / breadboard under test
- **Use cases:**
  - Complete system simulation with prototyped / breadboarded elements



SOFTWARE VALIDATION FACILITY

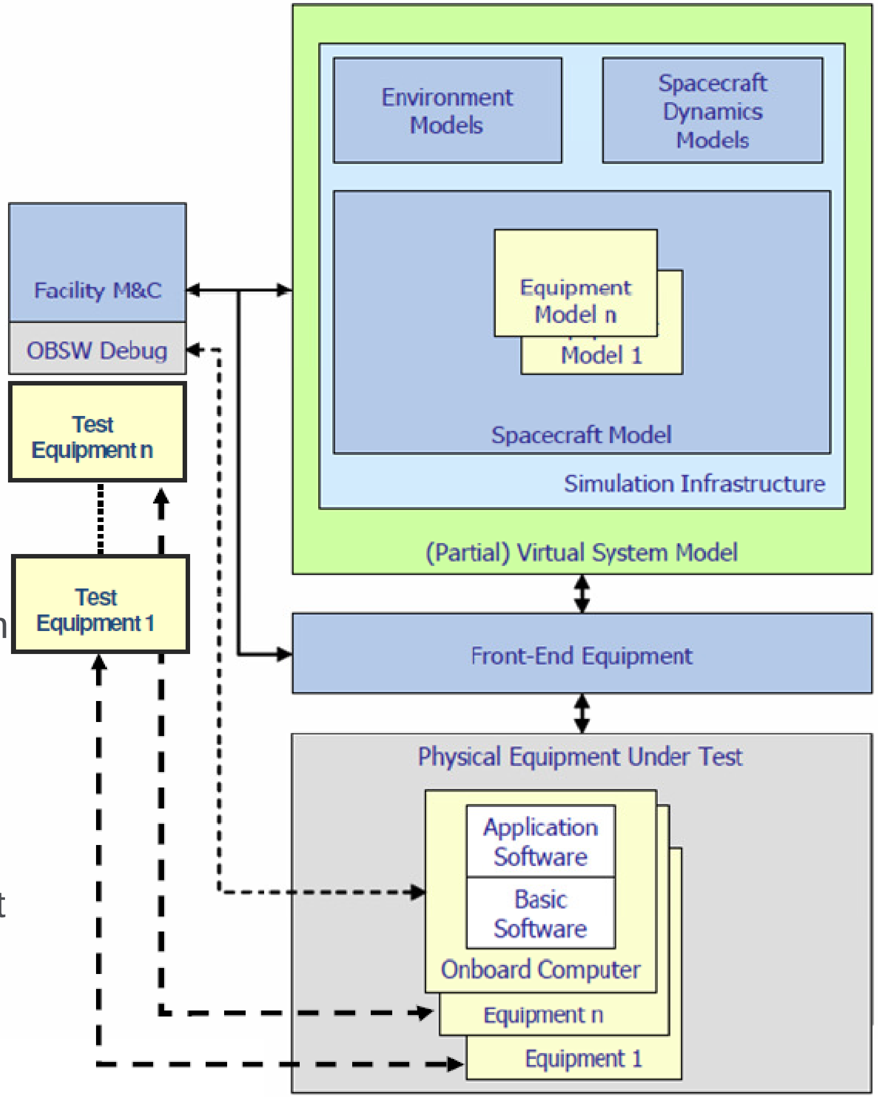
- **Validation of flight software in context**

- **Building blocks:**
  - Functional models
  - Simulated TM/TC interface
  - Fully functional simulation of HW, including performance and dynamic behaviour
- **Use cases:**
  - Verification of transitions between FSW modes
  - FDIR design verification
  - FSW validation including regression testing
  - ISVV



REAL-TIME TEST BENCH

- **HW in the loop testing**
- **Building blocks:**
  - Functional models
  - On boards computer
  - Simulated TM/TC link
  - Representative TM/TC reference facility (SCOS2000-based)
  - Simulated sensors/actuators connected by MIL-STD-1553 bus
- **Use cases:**
  - HW–SW integration and verification
  - HW function verification
  - System validation with HW in the loop
- **An extension: End-to-End ATB**
  - Database to store/retrieve different configurations



TECHNIQUES TO OPTIMISE TESTING EFFORT

- **Careful risk assessment**

- **Correctness-by-design & formal methods**
  - Especially for extra-functional properties
  - In some cases this is already in use (e.g. schedulability analysis – mathematical model rather than testing)

- **Testing logistics**
  - Configurable test benches
  - Avionics Test Bench

- **Saving time / effort by limiting the involvement of humans**
  - Automatic test case identification
  - Automatic test generation
  - Automatic test execution
  - Automatic test results analysis
  - Automatic test report generation

TECHNIQUES TO OPTIMISE TESTING EFFORT

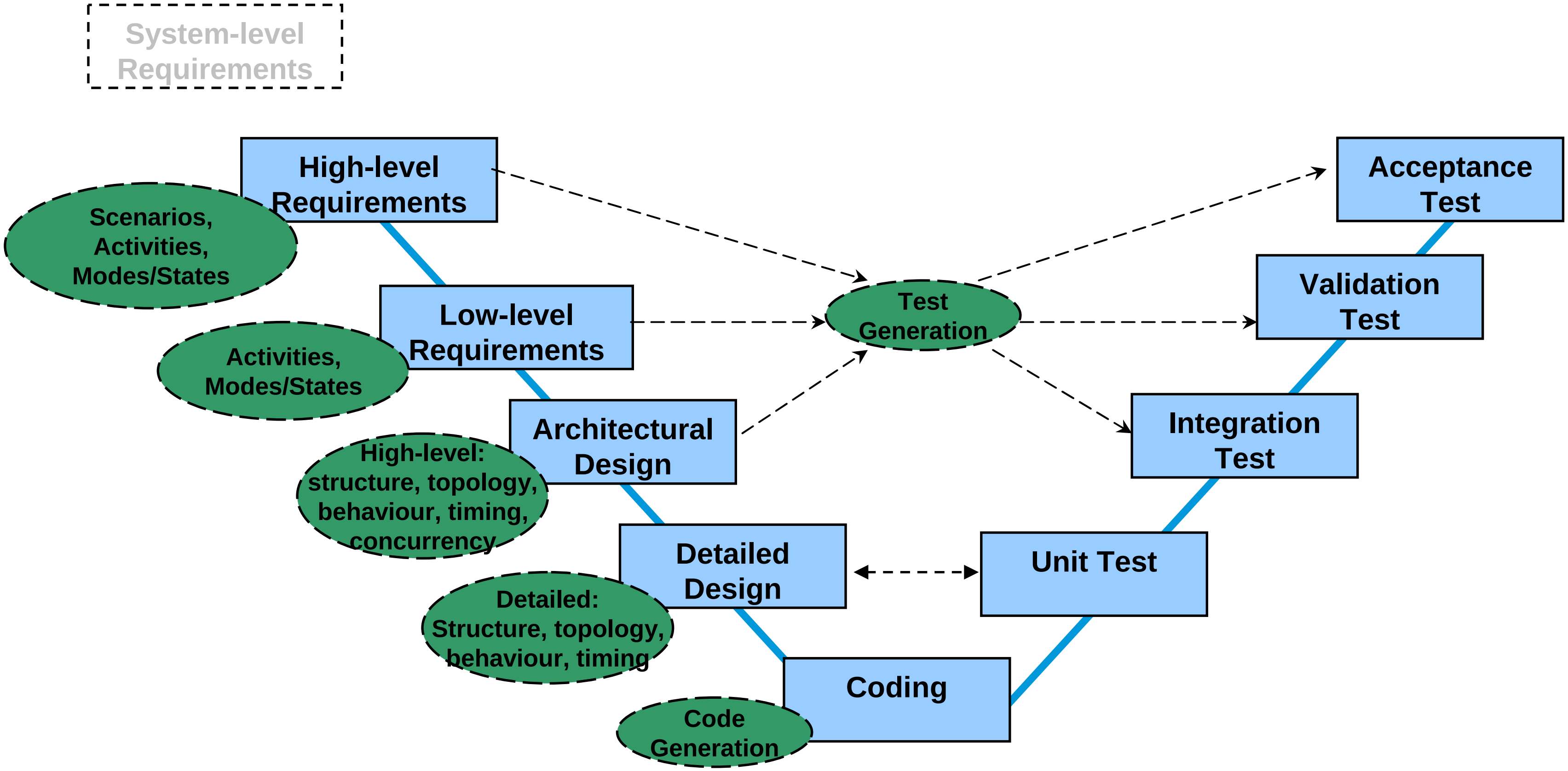
- **Careful risk assessment**

- **Correctness-by-design & formal methods**
  - Especially for extra-functional properties
  - In some cases this is already in use (e.g. schedulability analysis – mathematical model rather than testing)

- **Testing logistics**
  - Configurable test benches
  - Avionics Test Bench

- **Saving time / effort by limiting the involvement of humans**
  - Automatic test case identification
  - **Automatic test generation**
  - **Automatic test execution**
  - **Automatic test results analysis**
  - Automatic test report generation

MODEL-BASED TESTING



- Consistency between models
- Multi-formalism, model translations (horizontal)
- Multi-formalism, model refinement (vertical)
- Code w.r.t. the higher-level models

A STUDY: MODEL-BASED TEST GENERATION

Key idea: Automate test design by creating a system model and generate integration test cases out of behaviour diagrams

- **System model:** Formal modelling of requirements facilitates V&V of their completeness and correctness early on in the software development lifecycle
- **Test case identification and development:** Automatic test case generation from the test models improves efficiency and effectiveness of test case identification and development, possibly with improved test coverage
- **Testing automation:** Automatic test execution makes the testing phase more efficient, with support for test re-execution and regression testing
- **Quality assurance of the test model:** Static analysis, inspection, simulation
- **Improved traceability** from system requirements through the system model and test model to test cases and their results
- **Applicability to nominal as well as Independent V&V**
- **Weaknesses:** Not capturing of dynamic behaviour (e.g. interrupts)

MODEL-BASED TEST GENERATION WORKFLOW

1) **System modelling**: Model the System under Test (SUT) and model the environment around the SUT
   - Sequence-based specification

2) **Abstract test generation**: Use system models to generate abstract tests
   - I.e. tests which are independent of both the test programming language used and the test environment
   - Statistical operational testing
     - Markov chain usage model

3) **Concrete test generation**: Make tests executable

4) **Test execution**: Execute the tests on the SUT respecting the pass/fail criteria

5) **Test results analysis**

A STUDY: SOURCE CODE-BASED TEST GENERATION

Key idea: Generate some units tests directly from the source code (instead from the specification)

- **Not the same as static analysis!**
  - Two ways of detecting a deviation
    - Rule-based: violation of a rule
    - Symptom based: observation of a symptom (e.g. unhandled exception)
  - Static analysis cannot detect unanticipated faults
- **Expected benefits:**
  - Source code provides a good start to generate the test environment
  - Stimuli are generated from the source code (path-driven stimuli identification)
  - Comparison of the stimuli set to the specification
    - Specification maybe incomplete if the code represents what is really needed (or vice versa)
  - Easier finding of test cases to achieve full code coverage (e.g. MC/DC)

AUTOMATIC TEST GENERATION: OBJECTIVES

- **Apply to real FSW of existing ESA satellite (cat B criticality)**
  - Model-based testing: Focus on integration/validation testing
  - Source code-based testing: Focus on unit testing
- **Compare with manual testing**
  - Cost/effort
  - Regression testing
  - Measure efficiency to find bugs
    - Fault injection
    - Comparison with FSW version with known bugs
    - Finding bugs which no other method found
  - Coverage metrics
  - Sensitivity to particular fault types?
  - Nominal or complementary method?
- **Assess applicability to our domain**
  - Full spacecraft FSW
  - Only some subsystems (especially model-based testing)
  - Nominal and independent V&V
- **Assess scalability**

CONCLUSIONS

- **Testing of FSW takes a lot off time / effort / cost**
  - Testing at different levels serves its purpose and it is very sensitive to reduce the effort by re-defining the objectives and/or combing more levels
  - Software lifecycle change may cause testing effort to increase (e.g. versioning)

- **Advanced methods are being developed to**
  - Substitute testing by other means (e.g. proofs)
  - Reduce testing effort
  - Increase testing efficiency
  - Increase confidence in testing

- **ESA is exploring several ways of reducing this effort**
  - Constantly assessing existing processes
  - Finding innovative methods
    - SW development (models)
    - V&V (formal proofs)
    - Testing
  - Applicable both to nominal and independent V&V

THANK YOU

Marek Prochazka

European Space Agency

Marek.Prochazka@esa.int