TESTING

Gusztáv Adamis
adamis@tmit.bme.hu

WHITE AND BLACK BOX TESTING

- White box testing – typically during development
  - Access to code
  - Access to development environment
- Black box testing
  - Internal structure of the code is not known/interested
  - Checks the communication between the tested entity and its environment
  - IUT/SUT – Implementation/System Under Test
  - Tester – may be decomposed
  - PCO – Point of Control and Observation

BLACK BOX TESTING

- Black box testing
  - Implementation/System Under Test
  - Point of Control and Observation

- Not possible to test all the situations
  - Test Purposes

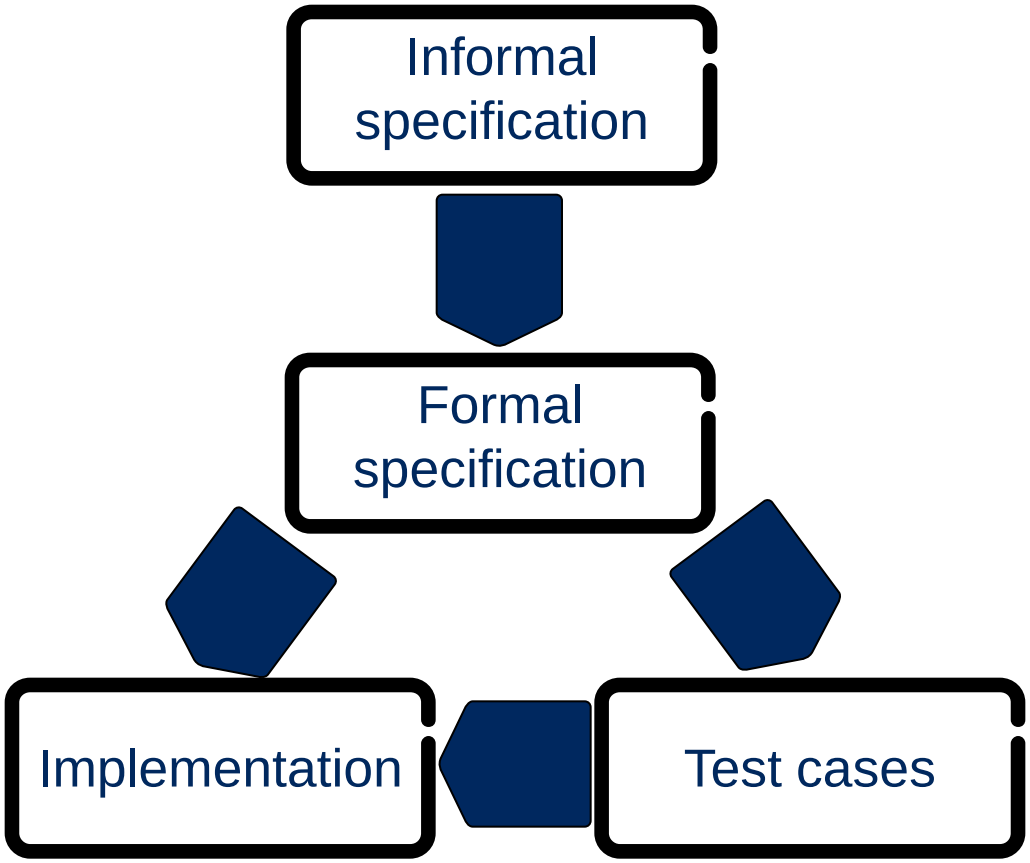
Verdict:

pass,

fail,

inconclusive

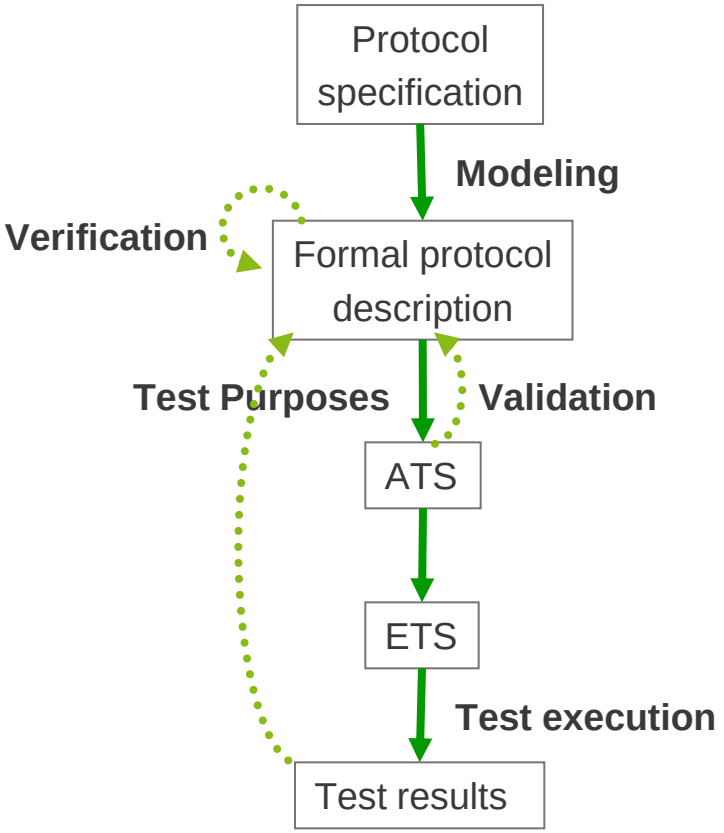
CONFORMANCE TESTING



- Checks if IUT conforms to its specification
- Experiments programmed into Test Cases

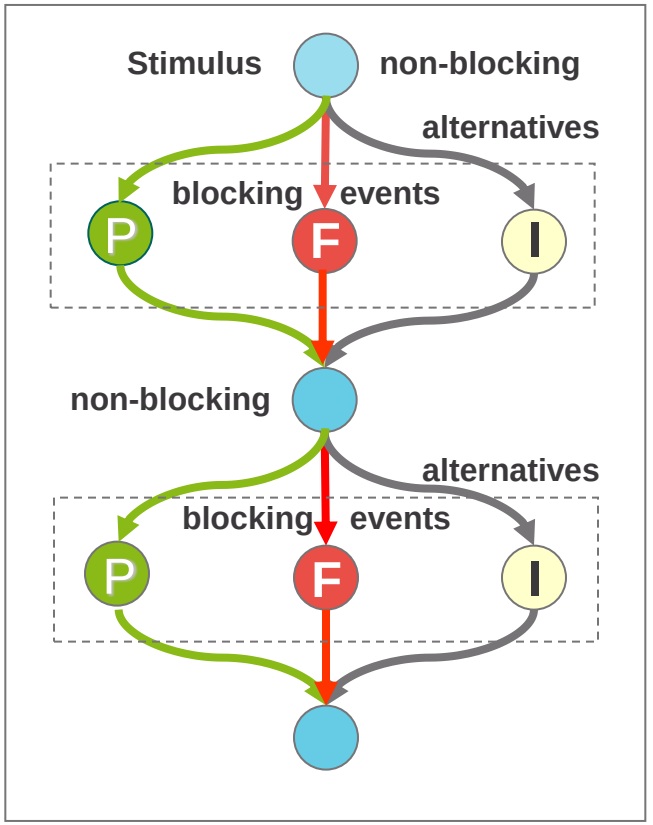
TEST SUITES

- Verification:
  - Check the correctness of formal model
- ATS – Abstract Test Suite
  - High-level communication
  - Test for every feature
  - Parameters
- ETS – Executable Test Suite
  - Coding/Decoding of messages
  - Tests only for implemented features
  - Parameters substituted by concrete values
- Validation
  - Checks the correctness of ATS



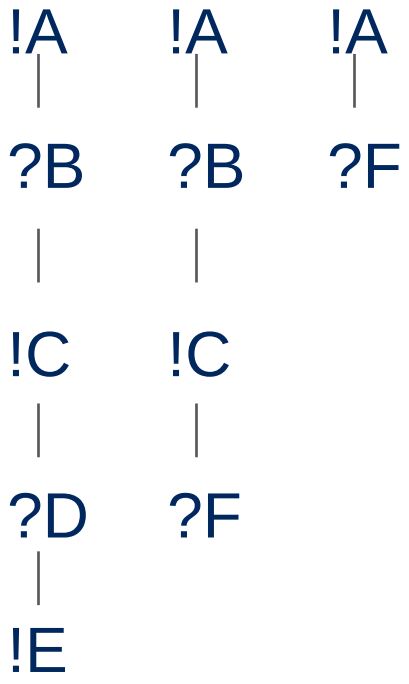
TEST CASES IN BLACK-BOX TEST

- Implementation of a Test Purpose
  - TP defines an experiment
- Focuses on a single requirement
- Returns verdict (pass, fail, inconclusive)
- Typically a sequence of action-observation-verdict update:
  - Action (stimulus): non-blocking (e.g. transmit PDU, start timer)
  - Observation (event): takes care of multiple alternative events (e.g. expected PDU, unexpected PDU, timeout)

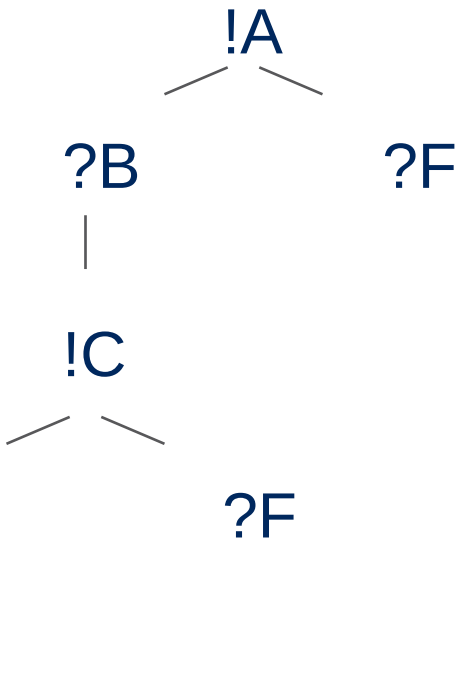


TEST TREE

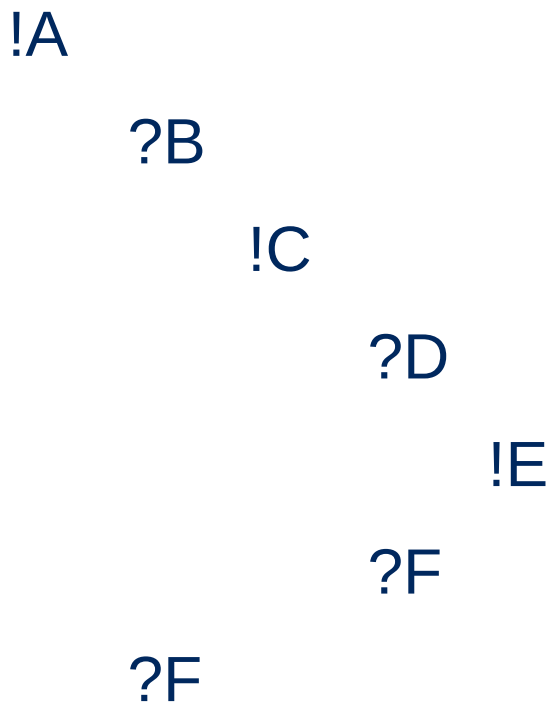
Possible event sequences



Behaviour tree



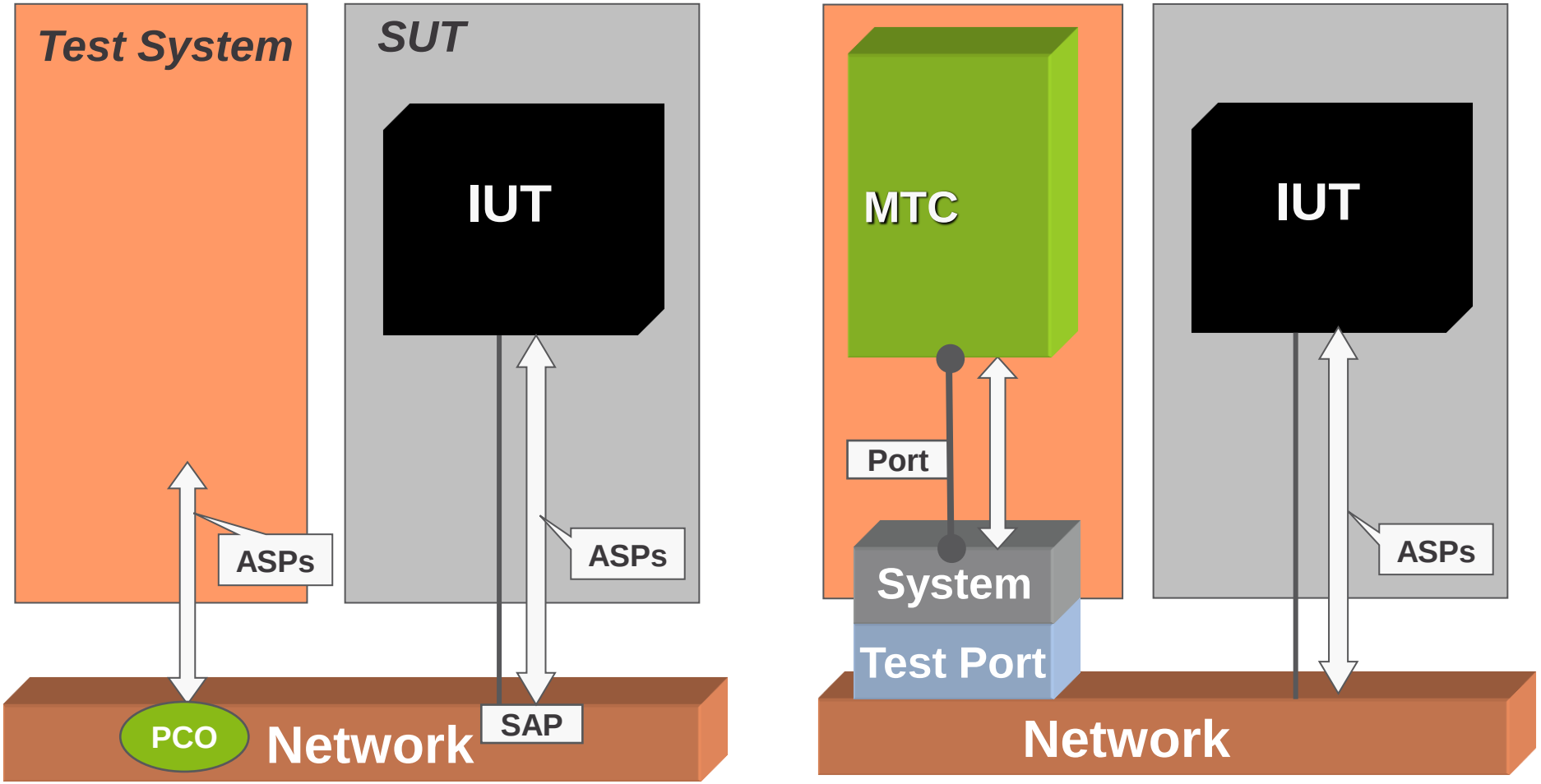
Alternatives



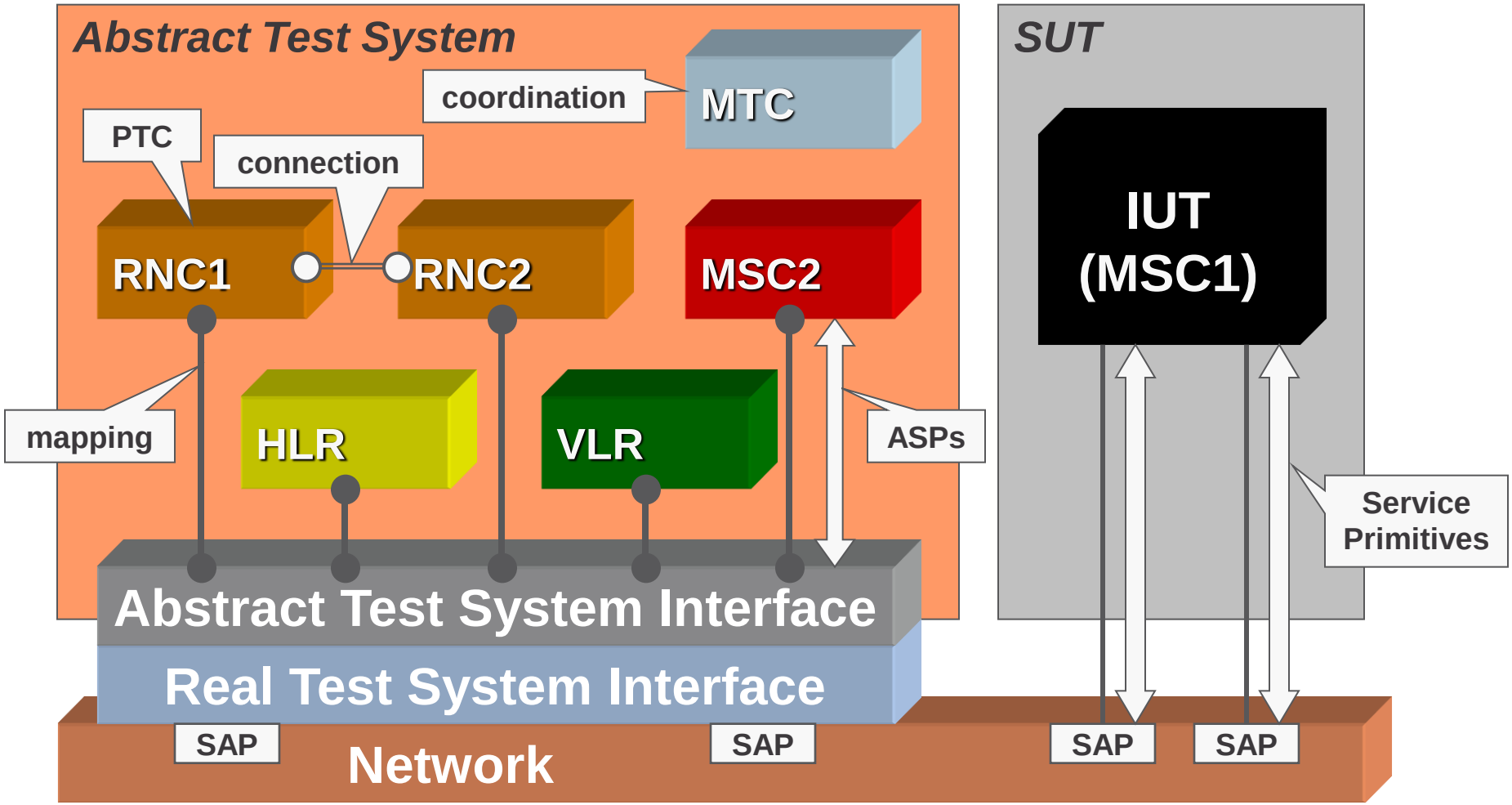
TEST EXECUTION

- Manual test execution
- Automated test execution
  - Test scripts
  - Log files

TEST ARRANGEMENT AND ITS TTCN-3 MODEL – TESTER IS A PEER ENTITY OF IUT

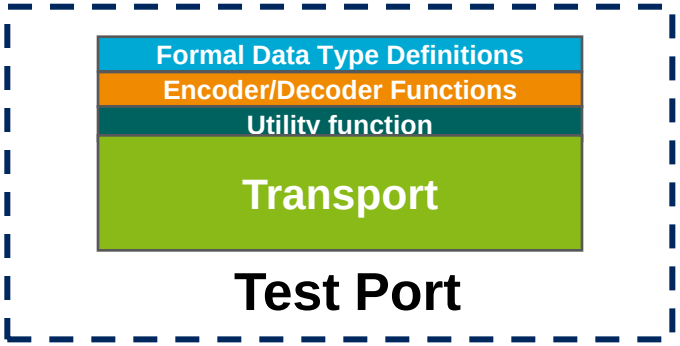


TTCN-3 VIEW OF TESTING – DISTRIBUTED TESTER

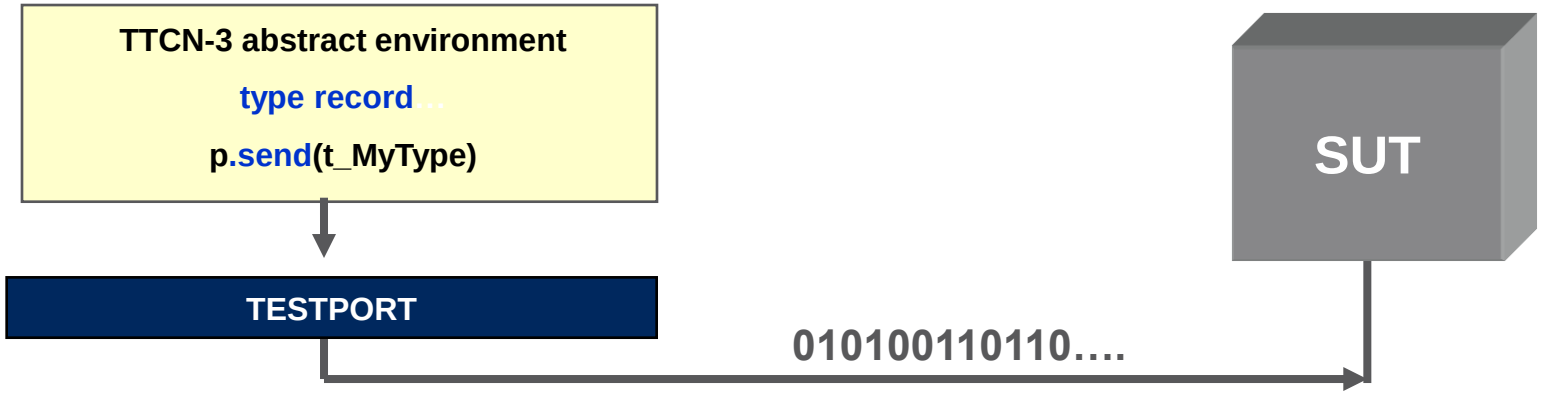


WHAT IS A TEST PORT (TP)?

- Proprietary communication interface between TITAN and SUT (i.e. abstract TTCN-3 test configuration and real world)
- Translates abstract TTCN-3 messages into real-world messages
- It contains:
  - Formal Data Type Definitions
  - Encoder/Decoder Functions
  - (Utility Functions)
  - Transport

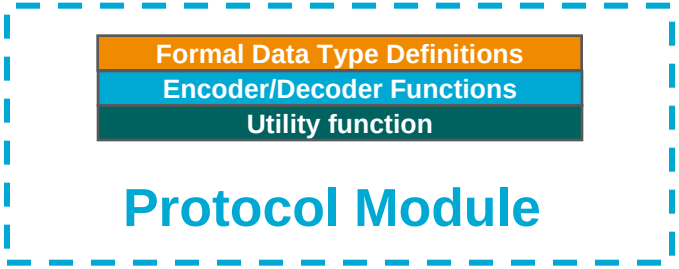


Test System Architecture

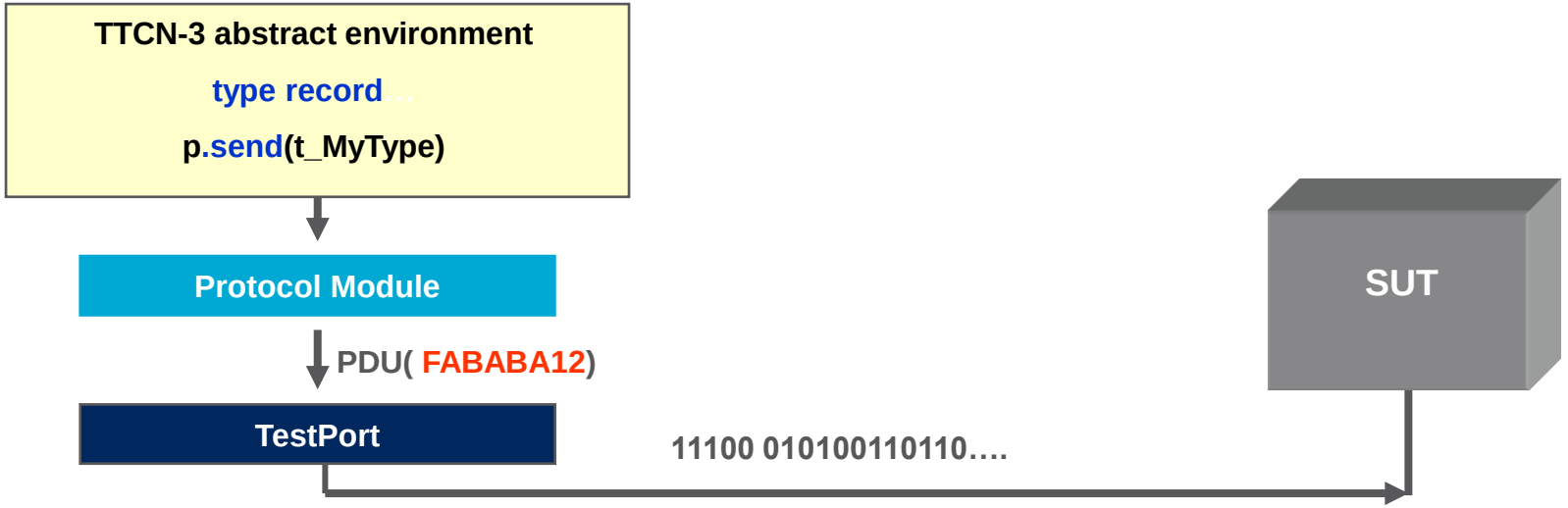


WHAT IS A PROTOCOL MODULE (PM)

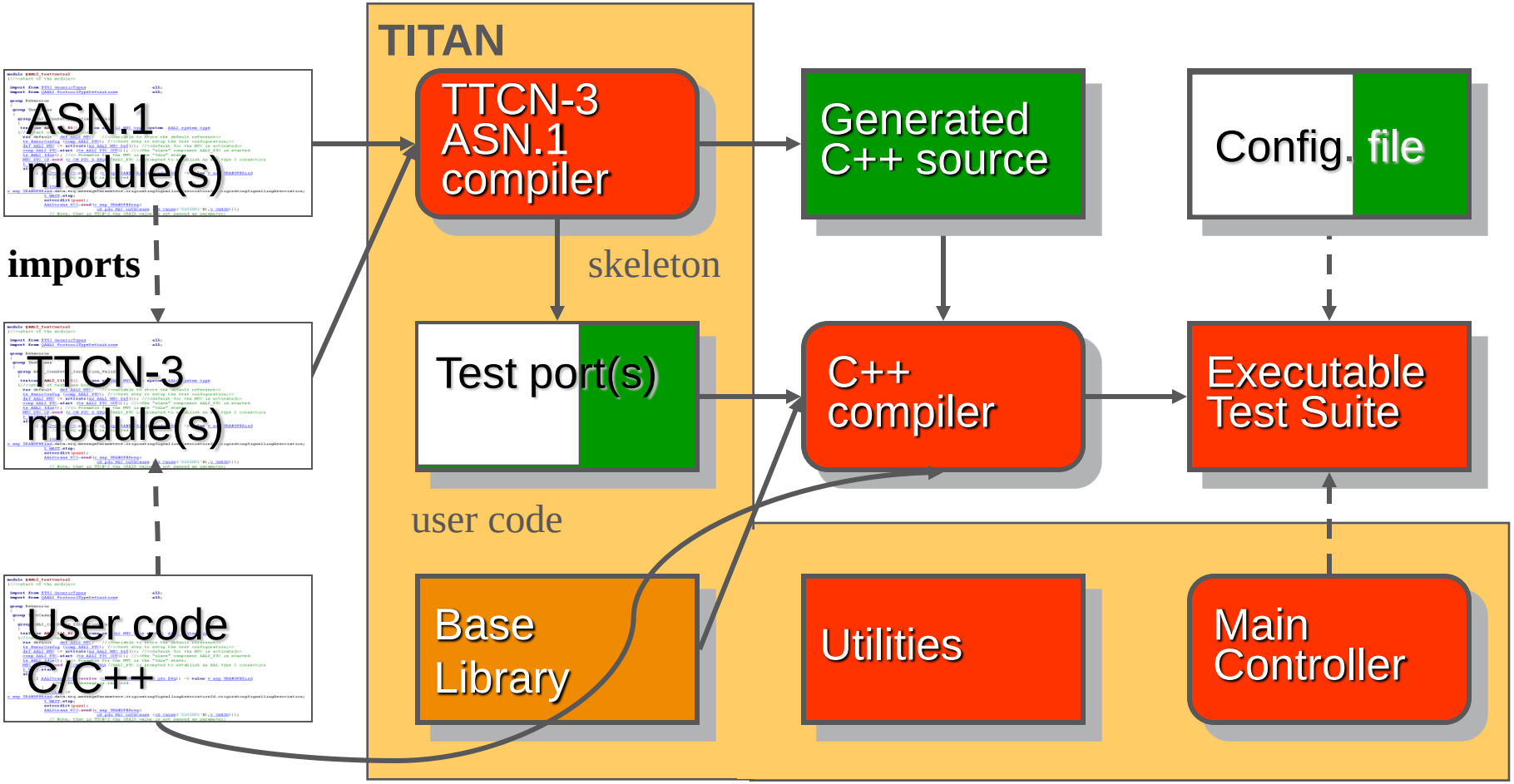
- Protocol modules implement the message structure of the related protocol in a formalized way, using the standard specification language TTCN-3.
- It contains:
  - Formal Data Type Definitions
  - [Encoder/Decoder Functions]
  - [Utility Functions]



Test System Architecture

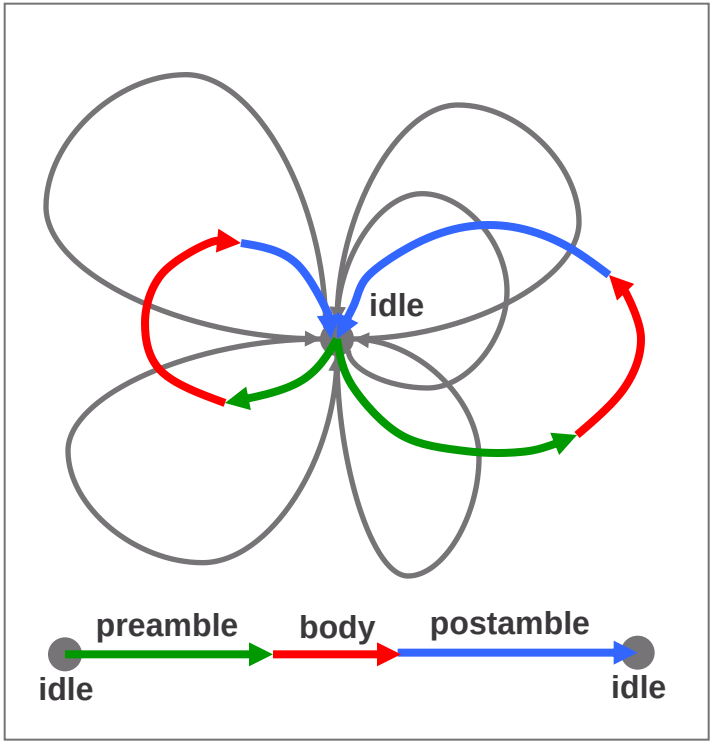


TITAN BLOCK DIAGRAM
(SIMPLIFIED)



INDEPENDENCE AND STRUCTURE OF ABSTRACT TEST CASES

- *Abstract test cases* should contain
  - *preamble:* sequence of test events to drive IUT into *initial testing state* from the *starting stable testing state*
  - *test body:* sequence of test events to achieve the *test purpose*
  - *postamble*: sequence of test events which drive IUT into a *finishing stable testing state*
- Preamble/postamble may be absent



REQUIREMENTS ON TEST SUITES

- All test cases in an ATS must be *sound*
  - *Sound* test case results pass verdict if IUT is correct (practically impossible with finite number of test cases)
  - *Exhaustive* test case gives fail verdict if IUT behaves incorrectly
  - *Complete* test case is both sound and exhaustive
- Must not terminate with none or error verdict

TEST RESULTS

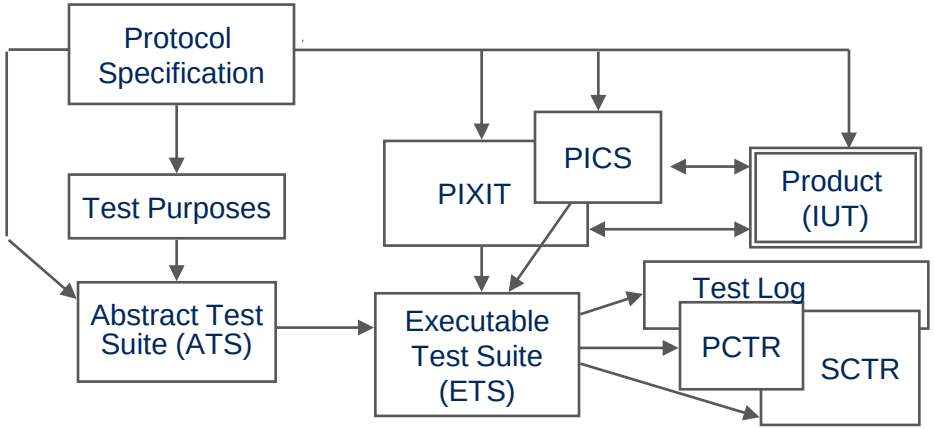
- Test outcome
  - foreseen
  - unforeseen – test case errors
- Verdict
  - pass
  - fail
  - inconclusive
- Test log
- Requirements on test outcomes
  - repeatable
  - comparable
  - auditable

CONFORMANCE TEST PHASES

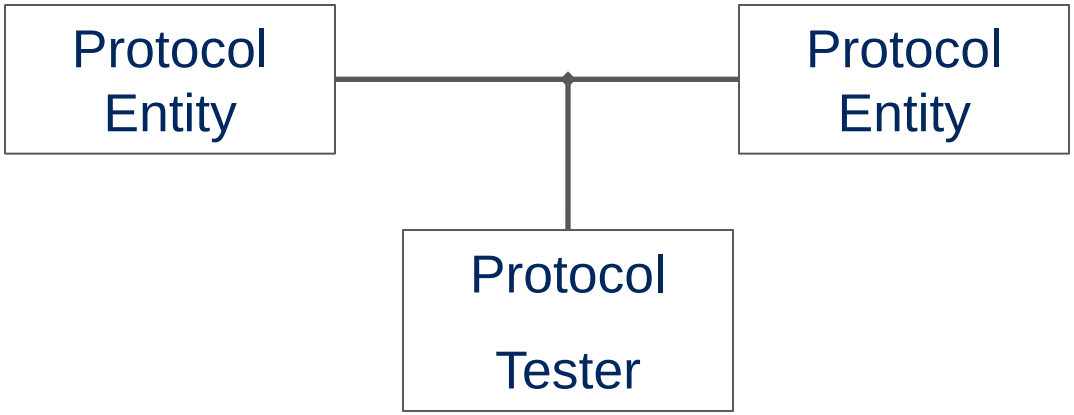
- Capability Test
  - Static analysis
    - if protocol options selected correctly
- Basic Interconnection Test
  - IUT able to communicate at all
- Behaviour Test
- Conformance Resolution Test
  - Non standardised methods
  - Multilayer tests
  - Detects reasons of non-conform situations
    - inconclusive

CONFORMANCE TEST DOCUMENTS

- PICS: Protocol Implementation Conformance Statement
- PIXIT: Protocol Implementation eXtra Information on Testing
- PCTR/SCTR: Protocol/System Conformance Test Report



PASSIVE TESTER



- Only observes
  - waits for error
    - no guarantee to happen
- Protocol Analyzer

ACTIVE TESTER

- Active
  - can send messages
- Valid testing
- Provocative testing
  - Invalid
    - Sends syntactically incorrect messages
  - Improper
    - Sends syntactically correct messages, but at wrong time/state
- Test cases are generated before testing starts

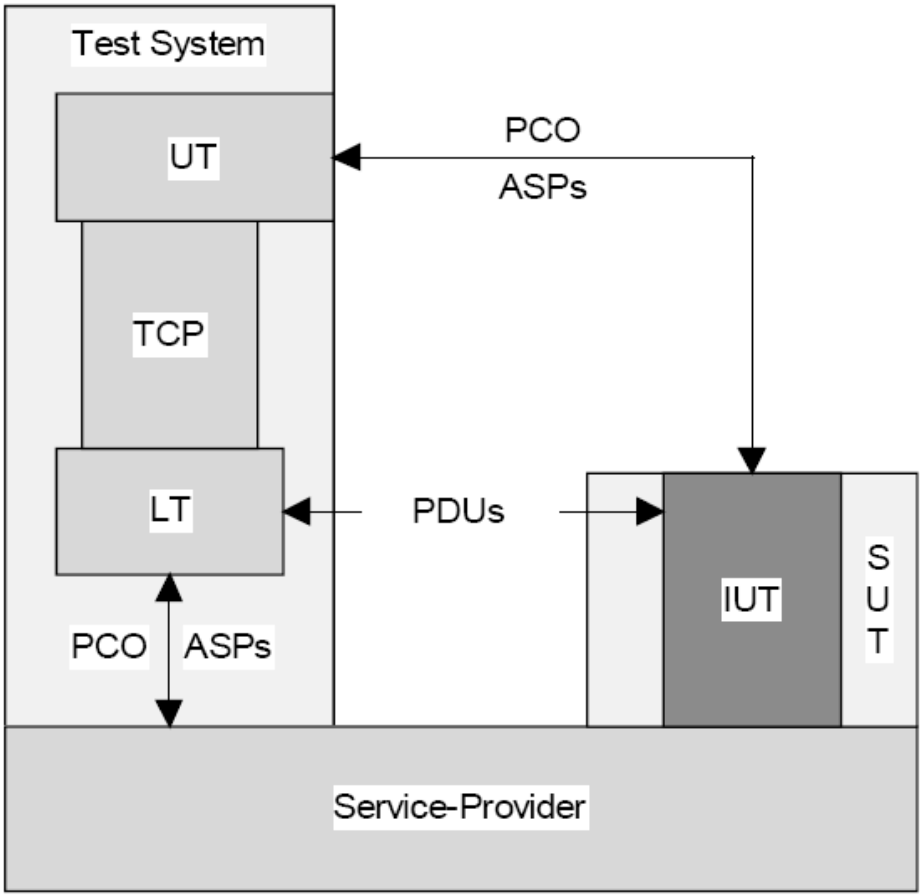
TEST ARRANGEMENTS

- ISO 9646

- Upper Tester
- Lower Tester

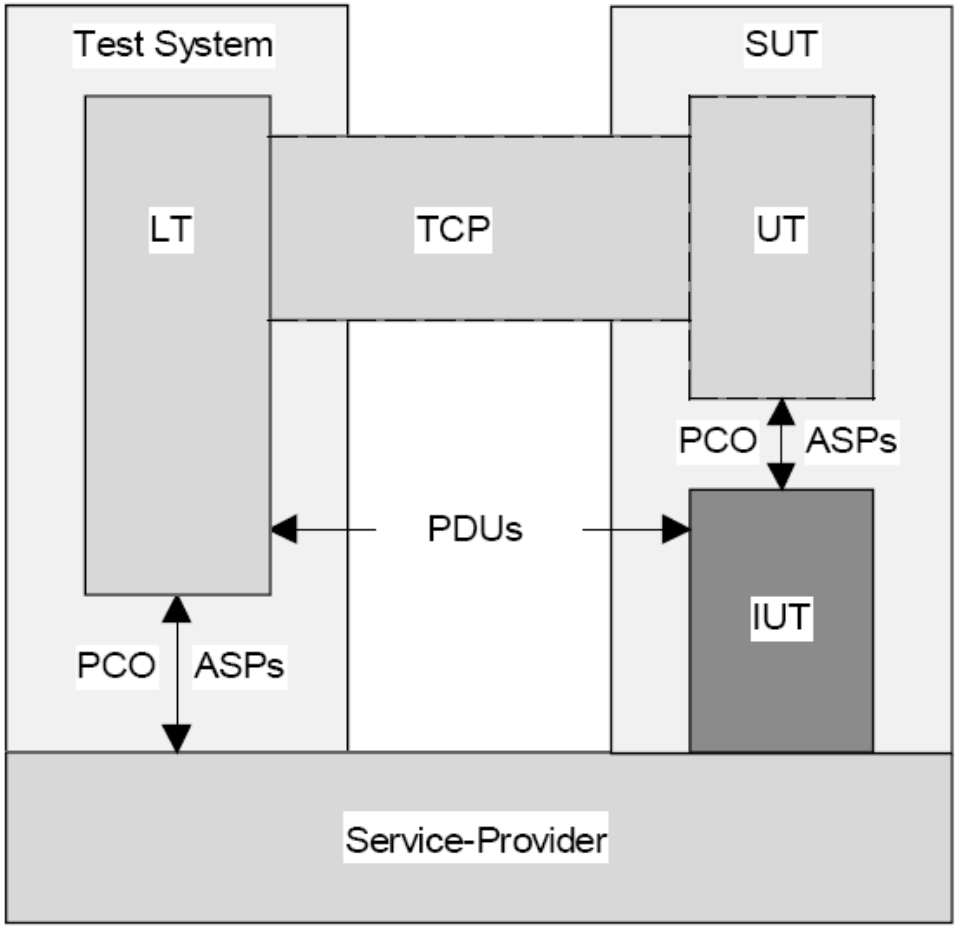
- Local Test Method
- Distributed Test Method
- Coordinated Test Method
- Remote Test Method

LOCAL TEST METHOD



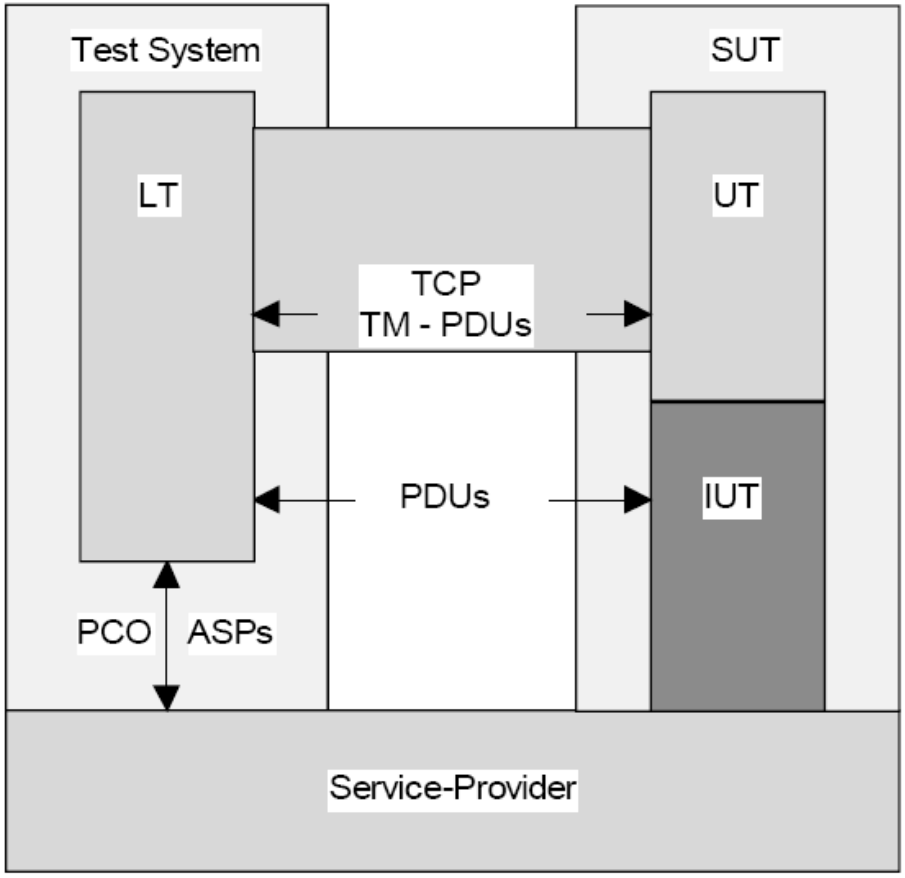
a) The Local test methods

DISTRIBUTED TEST METHOD



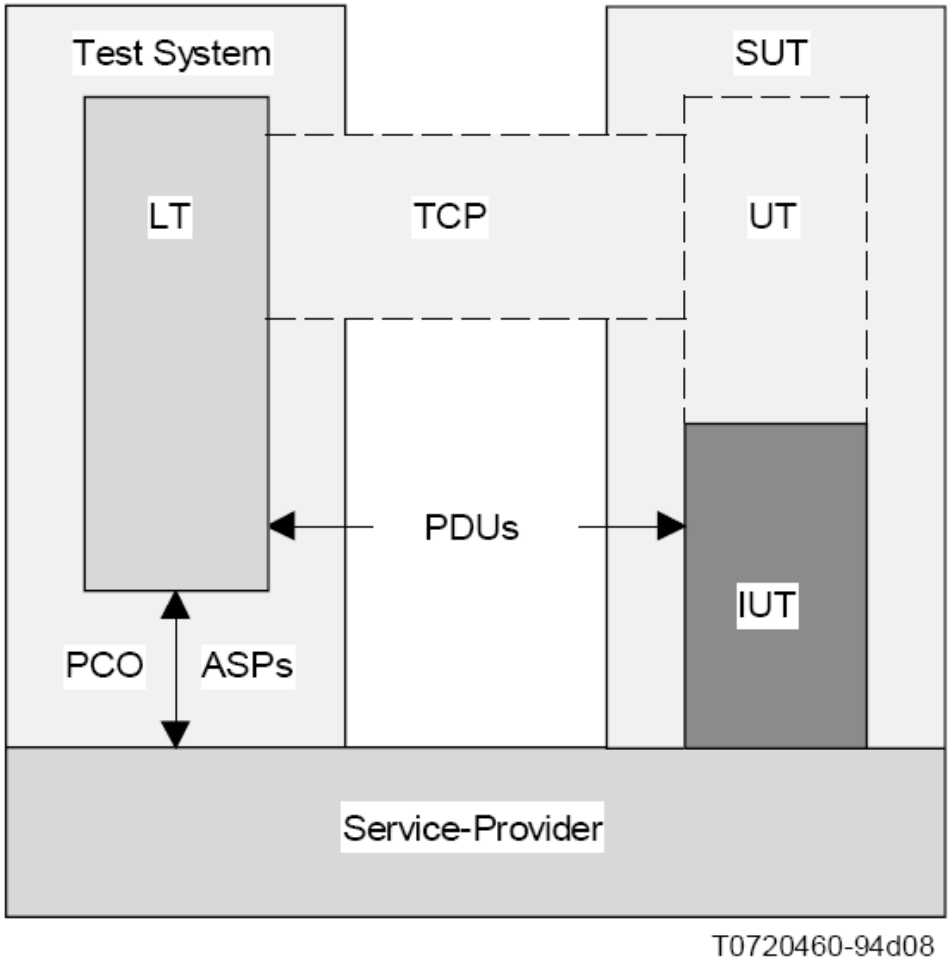
b) The Distributed test methods

COORDINATED TEST METHOD

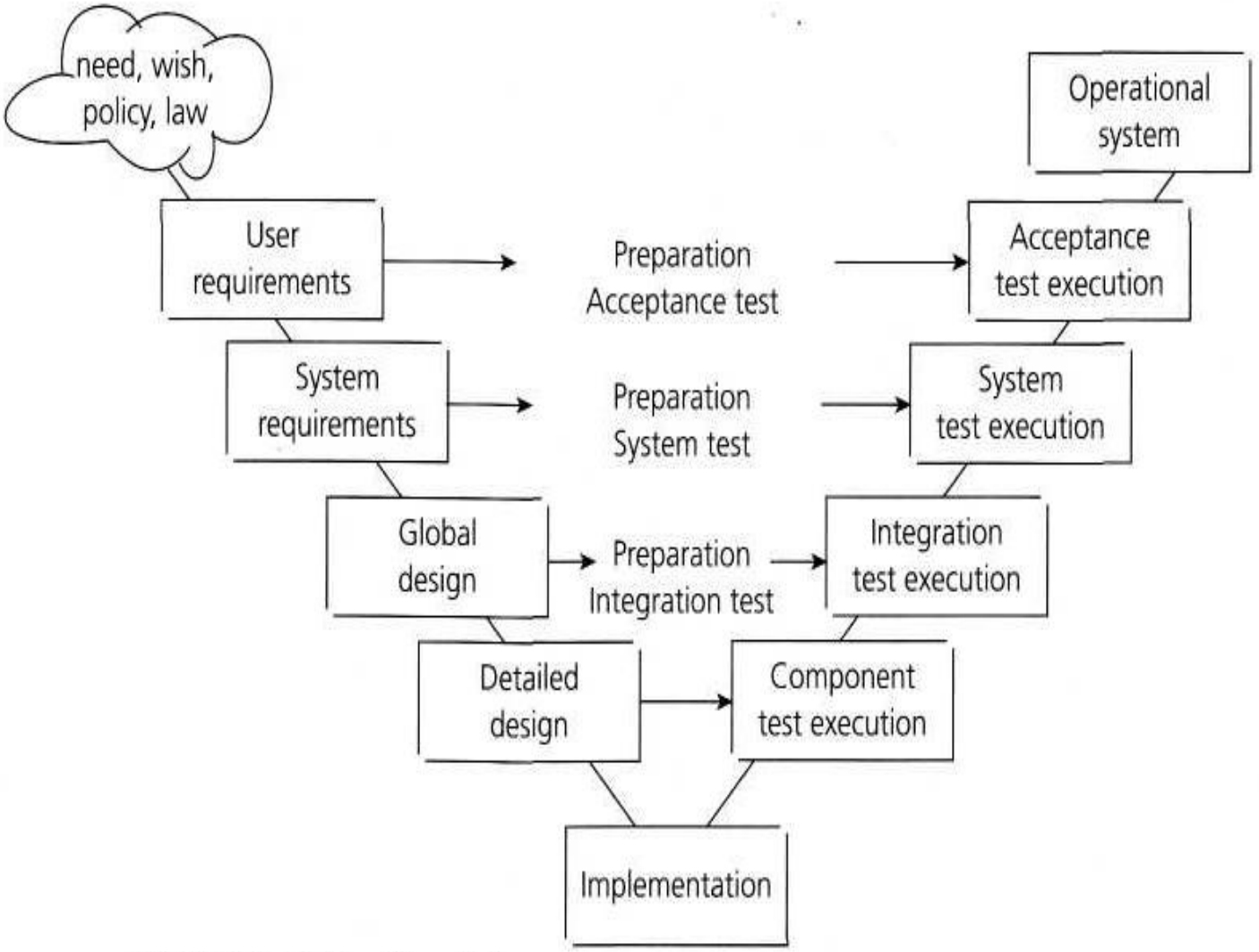


c) The Coordinated test methods

REMOTE TEST METHOD



d) The Remote test methods


need, wish,
policy, law
User
requirements
System
requirements
Global
design
Detailed
design
Implementation
Component
test execution
Integration
test execution
System
test execution
Acceptance
test execution
Operational
system
Preparation
Integration test
Preparation
System test
Preparation
Acceptance test

COMPONENT / UNIT TEST

- Component testing
  - also known as unit, module and program testing,
- Searches for defects in, and verifies the functioning of software
  - e.g. modules, programs, objects, classes, etc.

    that are separately testable
- Focuses on one class or method
- Small, fast
  - Unit tests run fast. If they don't run fast, they aren't unit tests.
  - All the unit tests shall run in less than ~10 seconds

COMPONENT / UNIT TEST

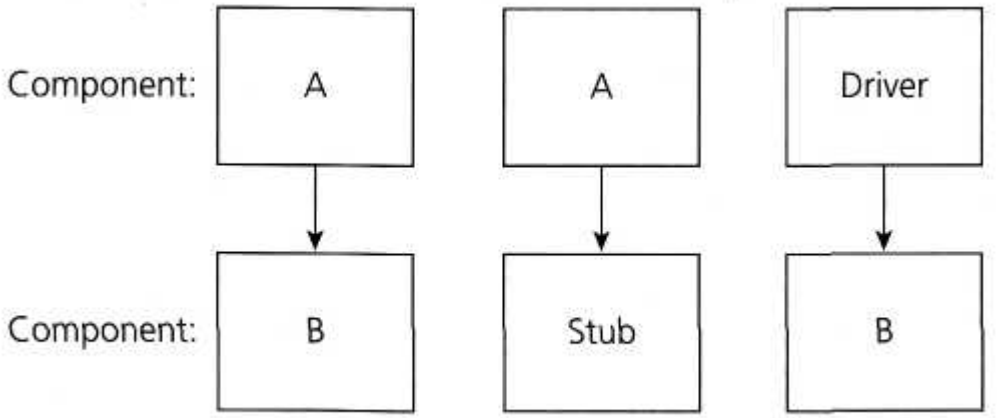
- White-box testing type
  - Access to code
  - Access to development environment
  - Writes the programmer/developer
    - Sometimes a different one
  - Defects fixed when found
- They test *how* the code is implemented rather the concept

COMPONENT / UNIT TEST

- All code must have unit tests
- All code must pass all unit tests before it can be released
- When a bug is found, tests shall be created

UNIT TESTS

- Mocking:
  - substitutes its own object (the “mock object”) for an object that talks to the outside world
  - checks that it is called correctly and provides a pre-scripted response
- Stubs and Drivers



INTEGRATION TESTS

- **Integration testing** tests interfaces between components, interactions to different parts of a system such as an operating system, file system and hardware or interfaces between systems
- Checks how code communicates with the rest of world
  - talks to a database
  - communicates across a network
  - touches the file system
  - special things to your environment (such as editing configuration files) to be done to run it
- Focused integration test
  - Tests just one interaction

LEVELS OF INTEGRATION TESTING

- Component integration testing
  - tests the interactions between software components and is done after component testing;
- System integration testing
  - tests the interactions between different systems and may be done after system testing.

- The greater the scope of integration, the more difficult it becomes to isolate failures to a specific interface

INTEGRATION TEST APPROACHES

- ‘Big-bang' integration testing
  - All components or systems are integrated simultaneously
  - Advantage: everything is finished before integration testing starts
    - no need to simulate (as no yet unfinished) parts
  - Disadvantage: time-consuming, difficult to trace the cause of failures with this late integration
  - Good if expecting to find no problems
- Incremental testing
  - All components are integrated one by one, and a test is carried out after each step
  - Advantage: defects are found early in a smaller assembly when it is relatively easy to detect the cause.
  - Disadvantage: it can be time-consuming since stubs and drivers have to be developed and used in the test

TYPES OF INCREMENTAL INTEGRATION TESTS

- Top-down: testing takes place from top to bottom, following the control flow or architectural structure (e.g. starting from the GUI or main menu)
  - Components or systems are substituted by stubs.
- Bottom-up: testing takes place from the bottom of the control flow upwards
  - Components or systems are substituted by drivers
- Functional incremental: integration and testing takes place on the basis of the functions or functionality, as documented in the functional specification

INTEGRATION TESTS

- Start with testing high-risk interfaces
  - Prevents major defects at the end of the integration test stage
  - If integration tests are planned before components or systems are built, they can be developed in the order required for most efficient testing
- Integration tests concentrate solely on the integration itself
  - Checks the communication between the integrated components not the functionality of them
- Testing of specific non-functional characteristics (e.g. performance) may also be included
- May be carried out by developers or by testers

INTEGRATION TESTS

- Shall run in the same way
  - If e.g. a data-base value needed – write it before the test
  - Independently the execution order
- Shall run on its own
  - Set up its environment
  - Restore the previous environment at the end
    - Even if fails or exception thrown (!)
- Not needed too many
  - Each shall test just one aspect of the communication
  - Number is proportional to the external interaction types
  - If lot of needed can indicate design problem
    - Business logic is not well separated from communication

SYSTEM TESTS

- System testing is concerned with the behavior of the whole system/product
  - It may include tests based on risks and/or requirements specification, business processes, use cases
  - System testing is most often the final test on behalf of development to verify that the system to be delivered meets the specification
  - Purpose: to find as many defects as possible
  - Investigate both **functional** and **non-functional requirements**
    - Typical non-functional tests include performance and reliability
  - Requires a controlled **test environment**
    - should correspond to the final target or production environment

ACCEPTANCE TESTS

- When development organization has performed system test, system will be delivered to the user or customer for **acceptance testing**
  - Acceptance testing is the responsibility of the user or customer
  - The execution of the acceptance test requires a test environment that is representative of the production environment
  - Acceptance testing determines whether the system is fit for its purpose
  - Finding defects should not be the main focus in acceptance testing
  - Although it assesses the system's readiness for deployment and use
  - Not necessarily the final level of testing
    - large-scale system integration test may come after the acceptance of a system.

TYPES OF ACCEPTANCE TESTING

- User acceptance test
  - Focuses on the functionality: validates the fitness-for-use of the system by the business user
- Operational (or production) acceptance test
  - Validates whether the system meets the requirements for operation
  - May include testing of backup/restore, disaster recovery, maintenance tasks and periodic check of security vulnerabilities
- Contract acceptance testing
  - Contract acceptance testing is performed against a contract's acceptance criteria
  - Acceptance should be formally defined when the contract is agreed
- Compliance (regulation) acceptance testing
  - Performed against the regulations which must be adhered to, such as governmental, legal or safety regulations

ALPHA/BETA TESTS

- If the system has been developed for the mass market
  - Feedback is needed from potential or existing users before the software product is put out for sale commercially.
- Alpha testing
  - Takes place at the developer's site.
  - A cross-section of potential users and members of the developer's organization are invited
  - Developers observe the users and note problems
- Beta testing,
  - A cross-section of users invited, who install it and use it under real-world working conditions.
  - The users send records of incidents with the system to the development organization where the defects are repaired.

END-TO-END TESTS

- Typically tests use cases
  - Acceptance tests
  - Functional tests
- Touches (almost) all components of the system
  - User interface, business layer, database
- Slow
  - Labor intensive setup, configuration, teardown
  - Tend to break when the system/labor configuration changes
  - Tests a lot of branches in code
  - Run seldom – at releasing

TEST TYPES: THE TARGETS OF TESTING

- A **test type** is focused on a particular test objective
  - testing of a function to be performed by the component or system;
  - a nonfunctional quality characteristic, such as reliability or usability;
  - the structure/architecture of the component or system;
  - related to changes,
    - i.e. confirming that defects have been fixed (confirmation testing, or re-testing)
    - looking for unintended changes (regression testing).
- Depending on its objectives, testing will be organized differently
  - E.g component testing aimed at performance would be quite different to component testing aimed at achieving decision coverage.

TESTING OF FUNCTION (FUNCTIONAL TESTING)

- The function of a system (or component) is 'what it does'.
  - Typically described in a requirements specification, a functional specification, or in use cases
- Functional tests are based on these functions, described in documents
- **Functional testing** considers the specified behavior
  - Black-box testing
  - Based upon ISO 9126
  - Can focus on suitability, interoperability, security, accuracy and compliance

VERSIONS OF FUNCTION TESTING

- Requirements-based testing
  - Uses a specification of the functional requirements
  - A good way to start is to use the table of contents of the requirements specification
  - Decide what to test (or not to test)
  - Prioritize the requirements based on risk criteria
    - This ensures that the most important/critical tests are included
- Business-process-based testing
  - Uses knowledge of the business processes
  - E.g business processes of a personnel and payroll system can be:
    - someone joins the company,
    - is paid on a regular basis
    - leaves the company, etc.

NON-FUNCTIONAL TESTING

- Testing of product quality characteristics or non-functional attributes of the system
  - how well or how fast the system works
    - performance testing (different load)
    - load testing (expected load)
    - stress testing (overloading)
    - usability testing
    - maintainability testing
    - reliability testing
    - portability testing

LOAD TEST

- Test how the system behaves in real environment
  - Expected traffic
- Testing with (high) traffic
  - Different traffic models
  - Simulating a lot of users
  - Need automation
  - Time limits
    - Off-line, on-line
- Very expensive tools

QUALITY CHARACTERISTICS ISO 9126

- **Functionality** (Functional testing)
  - suitability, accuracy, security, interoperability;
- **Reliability**
  - Maturity (robustness), fault-tolerance, recoverability
- **Usability**
  - understandability, learnability, operability, attractiveness
- **Efficiency**
  - time behavior (performance), resource utilization
- **Maintainability**
  - analyzability, changeability, stability, testability
- **Portability**
  - adaptability, installability, co-existence, replaceability

STRUCTURAL TESTING

- Testing of software structure/architecture
- 'white-box' or 'glass-box‘ testing
- Coverage of a set of structural elements
  - tool support to measure code coverage
  - Statements, decisions, functions, etc
- If coverage is not 100%, additional tests need to be written and run to cover those parts that have not yet been exercised
  - Depending on the exit criteria
  - Test Driven Development

TESTING RELATED TO CHANGES - 1

- Confirmation testing (re-testing)
  - Test fails -> determine the cause -> defect is reported -> new version of the software in which defect fixed
  - Execute the failed test again to confirm that the defect has indeed been fixed
- Important to ensure that the test is executed in exactly the same way as it was the first time using the same
  - Inputs
  - Data
  - Environment

TESTING RELATED TO CHANGES - 2

- Regression testing
  - Check if the modification of software/environment do not introduce bug in the non-modified part
- Also executes test cases that have been executed before
  - for regression testing, the test cases probably passed the last time they were executed
  - but in confirmation testing - they failed the last time
- Designed to collectively exercise most functions

REGRESSION TESTS – CTD.

- All the regression tests shall be executed every time a new version of software is produced
  - After bug-fixes
  - Change existing functionality
  - Introduce new functionality
  - Environment changes
    - E.g. new Data-base, new complier
- Ideal candidates for **automation**

EVOLUTION OF REGRESSION TEST SUITE

- Maintenance of a regression test suite is necessary
  - Shall evolve in line with the software
- When new functionality is added to a system
  - new regression tests should be added
- If old functionality is changed or removed
  - regression tests be changed or removed
- If becomes too large
  - subset of the test cases has to be chosen
  - keep the new/recently failed tests
  - eliminate test cases that have not found a defect for a long time (though this approach should be used with some care!)