A quick introduction to TTCN-3

Chapter 1: basics

- Common TTCN Concepts
- TTCN language(s)
- TTCN-3 new capabilities

Context: Origin of the TTCN languages

- Strong need to test telecom systems (protocol implementations) for **conformance** and **interoperability**
- All telecom systems should be tested the same way
- Development cycles become shorter and shorter
  – Testing process needs to be efficient, e.g. **automated**
- Similar products are needed for different customers
  – Call for tests that can be adapted easily to different product versions
- Broad spectrum of different test hardware with similar functionality
  – Need for abstract tests, independent from test hardware

Context: Origin of the TTCN languages

Issues with manual testing:

- Manual testing is labor-intensive, unattended testing is impossible
  - Testers need to work hard ☺
- Frequent testing is time-consuming
  - Tests are executed less often than would be required
- Different testers execute tests differently
  - Results of tests are not always reproducible
- Controlling complex tests is difficult
  - Deciphering test results is complex
  - Understanding test results is imprecise

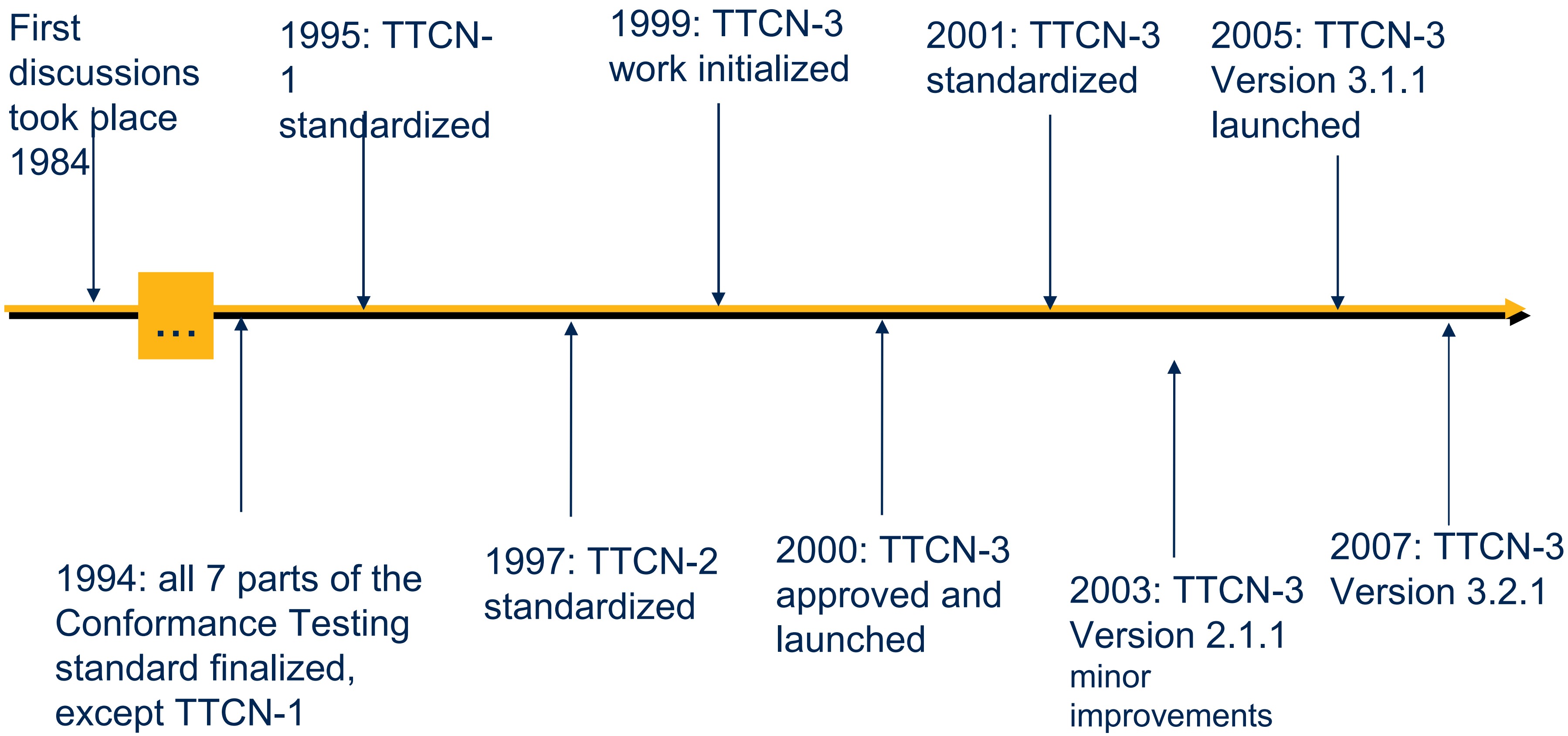
Context: Origin of the TTCN languages

- In the context of the OSI model, ISO developped a method for conformance testing: ISO 9646
- ISO 9646-3 endorsed a test language for conformance testing of protocol implementations, initialy developped in Sweden, called TTCN: Tree and Tabular Combined Notation.
- TTCN had severe limitations, which quickly led to the development of a revision called TTCN 2
- TTCN is still massively used today to test GSM, GPRS, EDGE and UMTS telecom systems (mobile and infrastructure)
- In the 90s, ISO lost interest in testing, and the focus moved to ETSI/MTS

Context: Origin of the TTCN languages

- ETSI/MTS developped a brand new test language called TTCN-3 (Test and Test Control Notation), based on the same principles as early TTCN 1 and 2, with a totally different syntax… and
- Not specific to the telecom world, so that it could be adopted by other industries (in particular the ones using datalinks, buses…)
- ETSI/MTS now provides:
  – The TTCN-3 language,
  – Protocol standards (LTE, IN),
  – Conformance test suites (LTE, IN, Wimax, IP v6…)

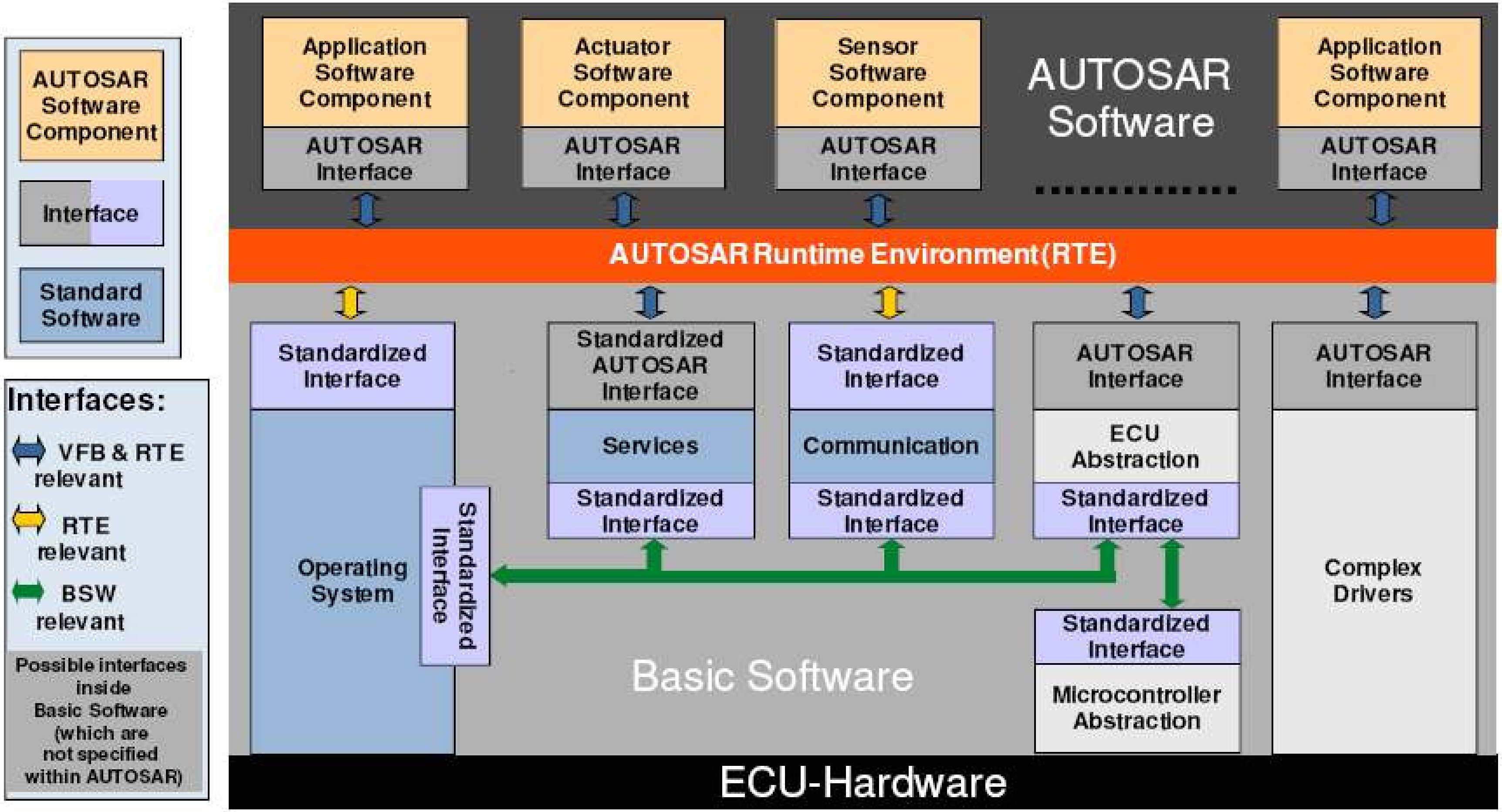
TTCN history



Context: telecom + automotive

- The automotive industry had shown interest for the TTCN ecosystem in the 90s.
- So: the lead author of TTCN made sure, when designing TTCN-3, that it would also meet the requirements of the automotive industry
- The AUTOSAR consortium (http://www.autosar.org) has reproduced in the automotive industry the ETSI+TTCN ecosystem: they provide standards as well as conformance test suites

AUTOSAR Software Architecture
Components and Interfaces



Chapter 1 : basics

- Context
- TTCN language(s)
- TTCN-3 new capabilities

The Nature of TTCN

- The one(s) and only standardized test languages (see next page)
- Ideal for black box testing
- Formal: executable
- target-independent : test any hardware or software by sending and receiving messages
- High abstraction level which enables reusability
- modularity
- data & behaviour separated
- languages with explicit and unique functionality for test (see next page)
- Enables parallel test components
- Includes ASN.1 for powerful test data definition
- Makes test script well documented and easy to read

TTCN: the standards

Name of standard document	ISO reference	ITU ref.
The Tree and Tabular Combined Notation (TTCN)	ISO 9646-3	X.292
Name of standard document The Testing and Test Control Notation version 3; …	**ETSI reference**	**ITU ref.**
Part 1: TTCN-3 Core Language	ES 201 873-1	Z.140
Part 2: Tabular Presentation Format for TTCN-3 (TFT)	ES 201 873-2	Z.141
Part 3: Graphical Presentation Format for TTCN-3 (GFT)	ES 201 873-3	Z.142
Part 4: TTCN-3 Operational Semantics	ES 201 873-4	-
Part 5: TTCN-3 Runtime Interface	ES 201 873-5	-
Part 6: TTCN-3 Control Interface	ES 201 873-6	-
Part 7: Using ASN.1 with TTCN-3	ES 201 873-7	-
TTCN-2 to TTCN-3 Mapping	ES 201 874	

TTCN vs. Programming Languages

- Tests focus only on implementation to be tested
- Rich type system including native list types and support for subtyping
- Embodies powerful built-in matching mechanism
- Snapshot semantics, i.e., well defined handling of port and timeout queues during their access
- Concept of verdicts and a verdict resolution mechanism
- Support for specification of concurrent test behaviour
- Support for timers
- Allows test configuration at run-time (TTCN-3 only)
- Offers potential for reducing training and test maintenance costs significantly
- Proven to work in very large and complex industrial tests, e.g., of 3G network elements

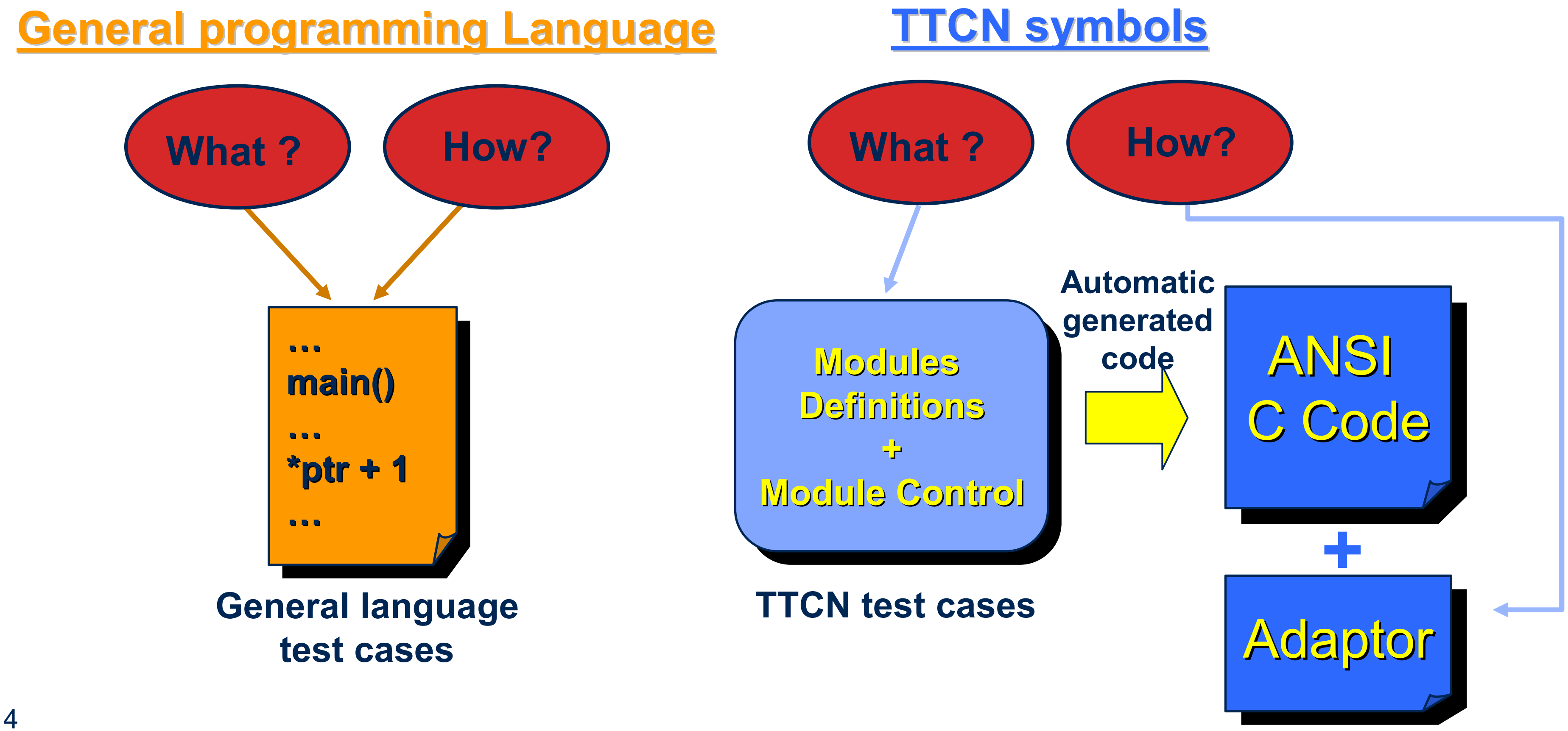
Portability: Focus on writing of the test cases

What ?

- Write the behavior of the test cases **(test objective)**

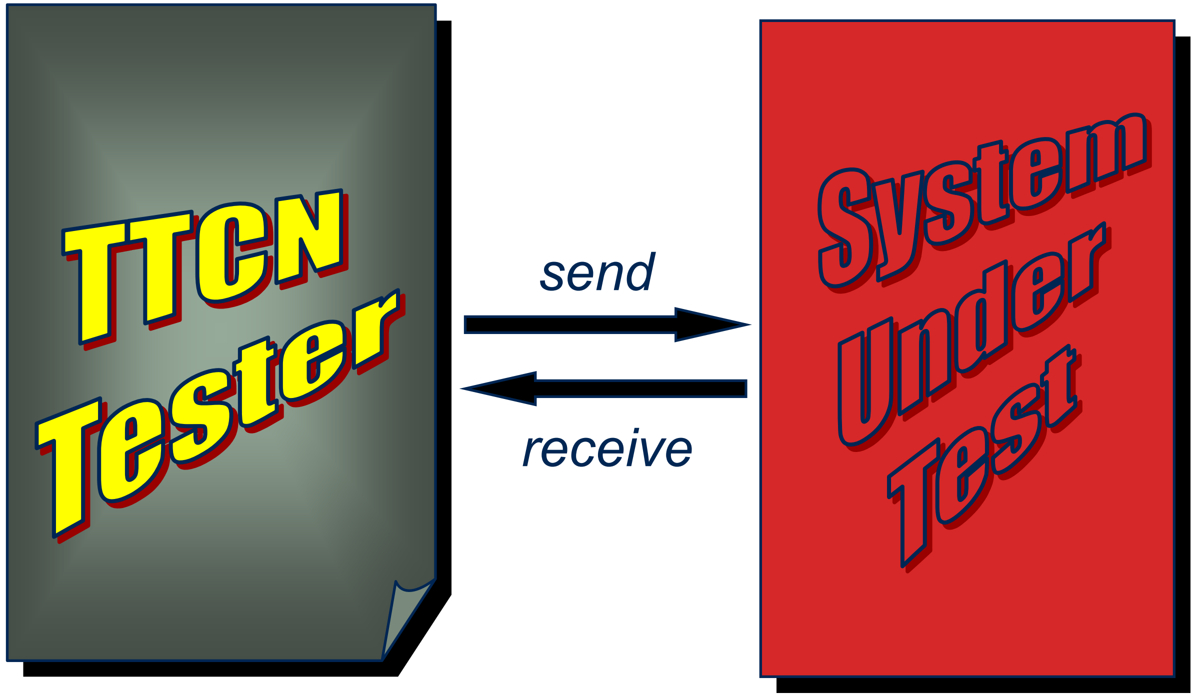
How ?

- Write the way to execute the test cases
(communication, timers, encoding/decoding of messages)



TTCN is for black box testing

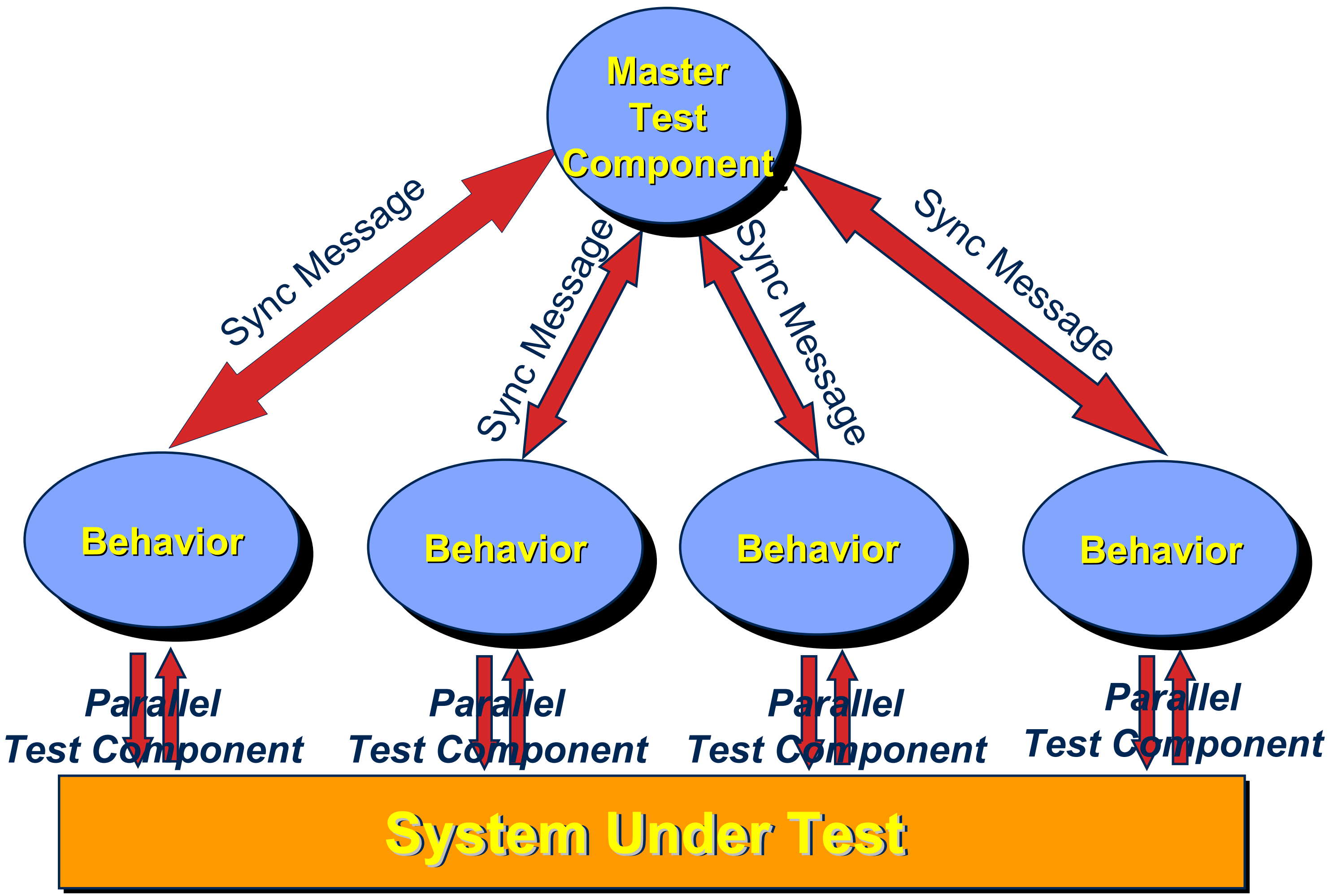
Test the behaviour of an implementation by sending & receiving messages



Black Box Testing

- The Tester has no Information about the internals of the System Under Test (SUT)
- Tests are designed along the Specification / the Requirements
- Tests are executed by stimulation of the Interfaces of the SUT and observing / checking the response
- „Test-Coverage“ means the Parts of the Specification, which is checked by testcases

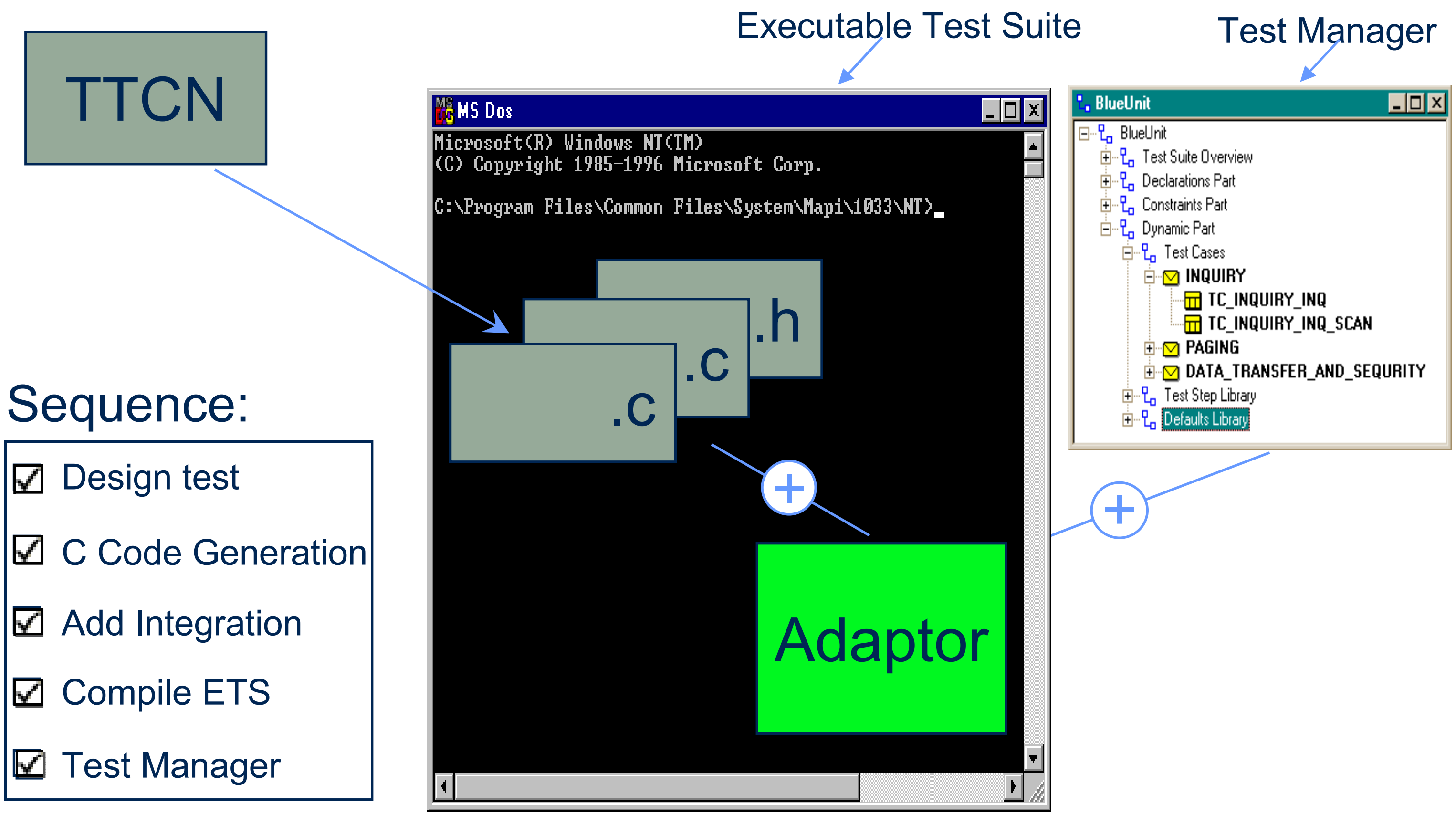
Concurrent TTCN: Distributed execution



TTCN: adaptation layer

- Both TTCN 2 and TTCN-3 are abstract languages: they make no assumption about the test platform, and the way the tester is interfaced to the SUT
- There must be an adaptation layer between the abstract test suite and the SUT
- The user provides the code for this adaptation layer
- The TTCN compiler generates C code form the TTCN ATS. This C code integrated with the user-provided adaptation layer make the ETS.

The Integration Process



Sequence:

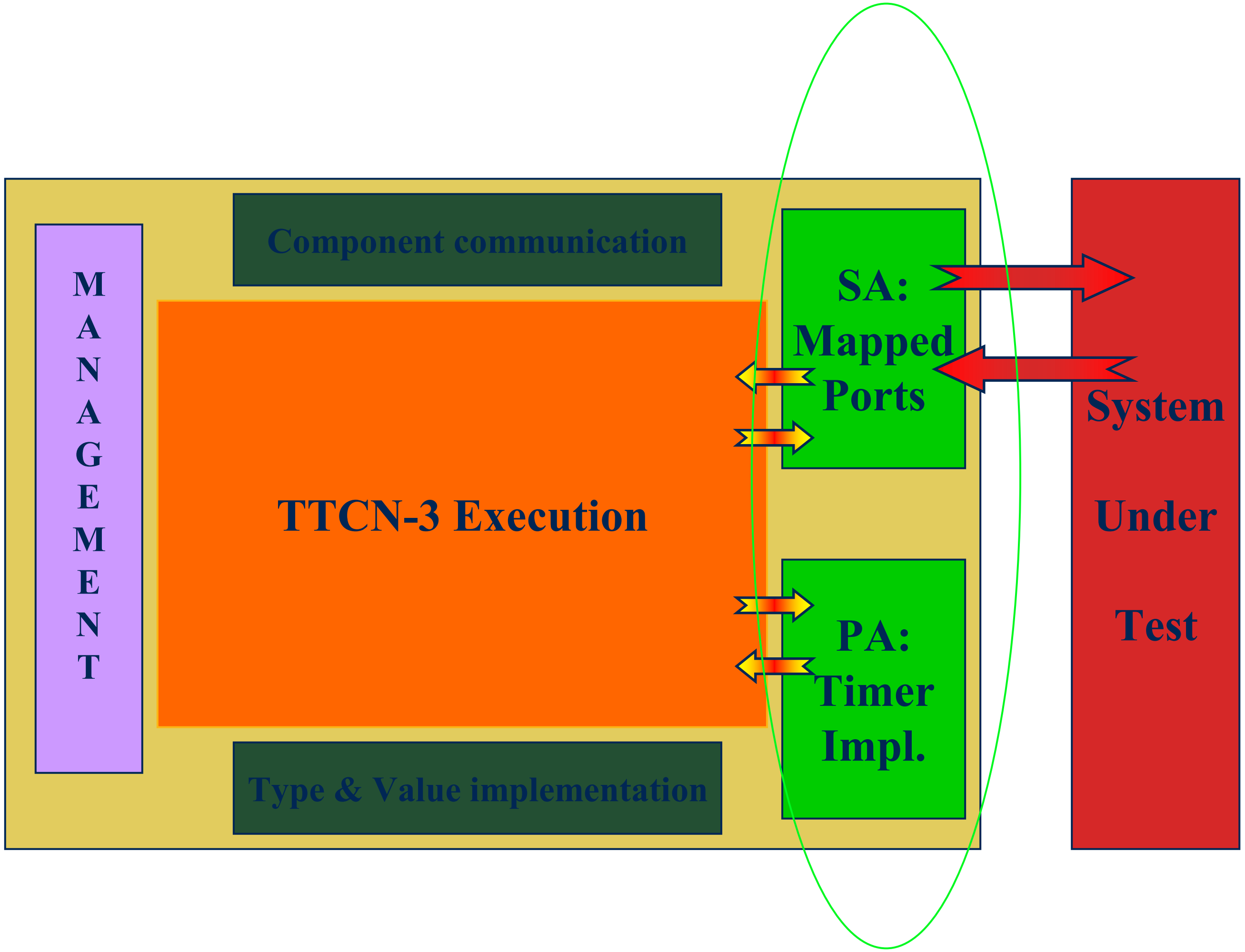
- ☑ Design test
- ☑ C Code Generation
- ☑ Add Integration
- ☑ Compile ETS
- ☑ Test Manager

The TTCN-3 adaptation layer: standardized interface

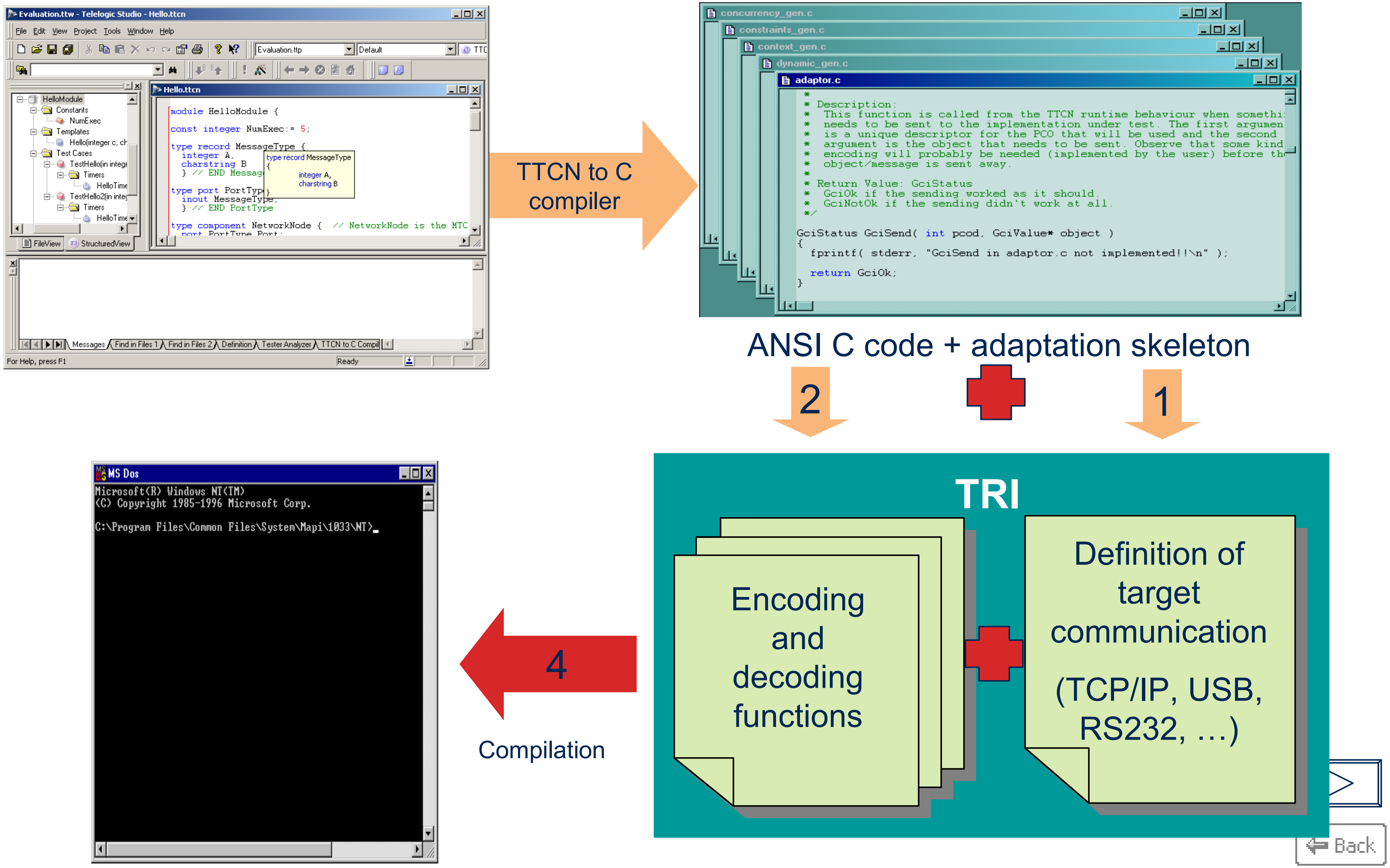
- A standardized adaptation helps the test implementors to develop communication and timing with the target/SUT faster
- Code produced by any compiler using the TRI interface can be run on all environments/test devices using the TRI interface
- The TRI interface is independant of the target
  - Platform, implementation language and environment

Back

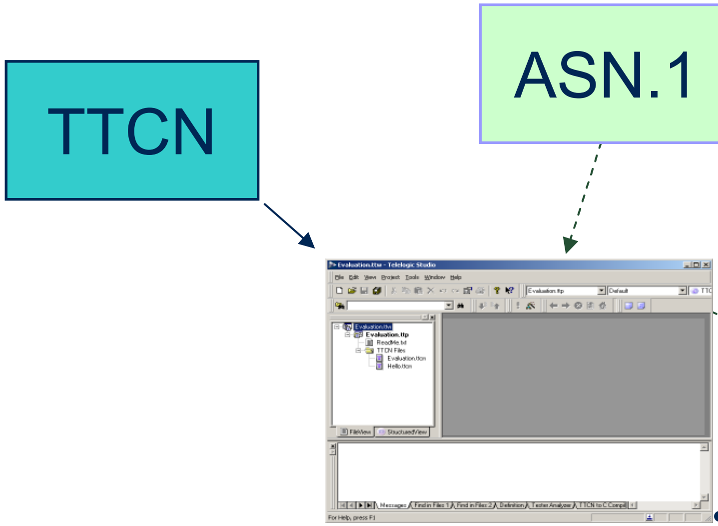
Building stones of TRI

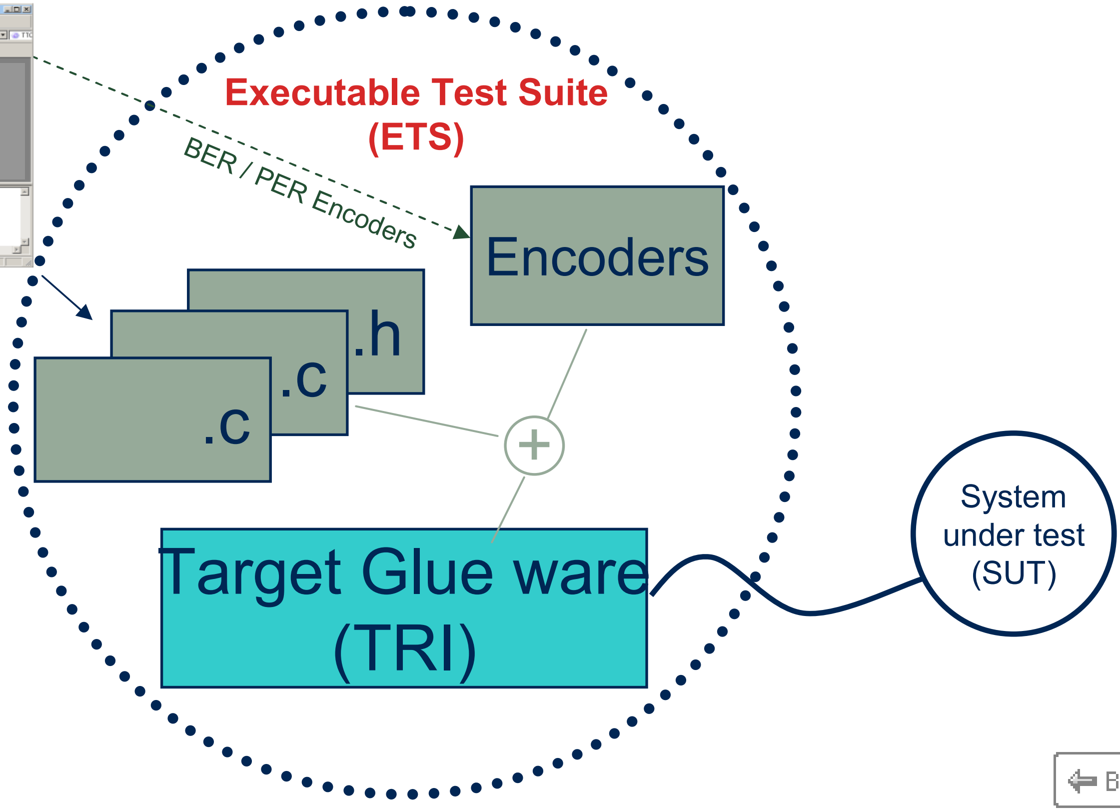


Executable Test Suite Generation



How do I get my test to work against my system?





Sequence:

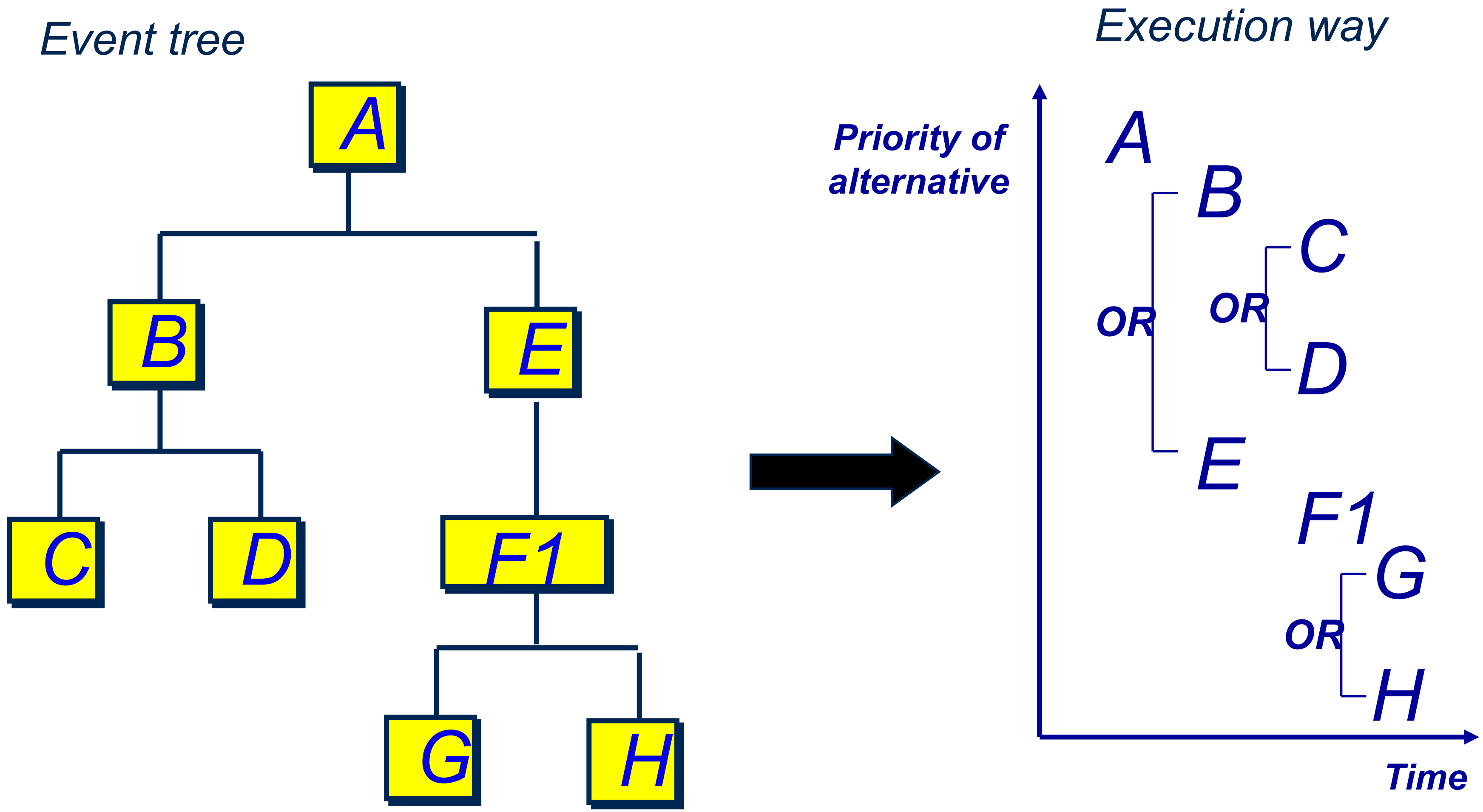
- ☑ Design test
- ☑ C Code Generation
- ☑ Add Glue ware + Encoders
- ☑ Compile ETS
- ☑ Execute against SUT

Back

Chapter 1 : basics

- Context
- Common TTCN Concepts
- TTCN-3 new capabilities

Tree and Tabular Combined Notation



TTCN: how they look

The way you will express it:

In TTCN 2:

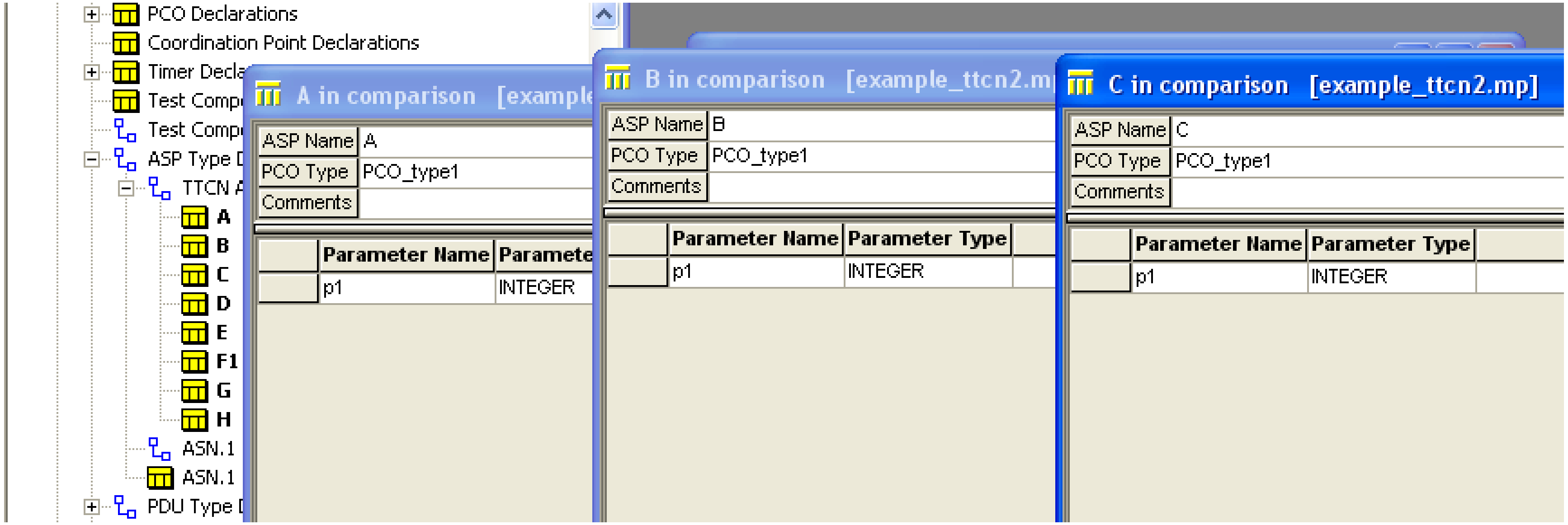
Nr	Label	Behaviour Description	Constraints	Verdict
1		my_PCO!A START RetransTimer	const_A1	
2		my_PCO?B START RetransTimer	const_B1	
3		my_PCO?C CANCEL RetransTimer	const_C1	INCONC
4		my_PCO?D CANCEL RetransTimer	const_D1	PASS
5		my_PCO?E CANCEL RetransTimer	const_E1	
6		my_PCO!F1 START RetransTimer	const_F1	
7		my_PCO?G CANCEL RetransTimer	const_G1	PASS
8		my_PCO?H CANCEL RetransTimer	const_H1	INCONC

In TTCN-3:

```
testcase TC_1() runs on NodeType {
    var default v_DefaultHandler := activate(def_ErrorHandling());
    timer RetransTimer := RETRAN_TIMER;
    my_pco.send(temp_A1); //send message to the SUT
    RetransTimer.start; //setting the retransmission timer
    alt {
        [] my_pco.receive(temp_B1) {
            RetransTimer.start;
            alt {
                [] my_pco.receive(temp_C1) {RetransTimer.stop; setverdict(inconc);}
                [] my_pco.receive(temp_D1) {RetransTimer.stop; setverdict(pass);}
            }
        }
        [] my_pco.receive(temp_E1) {
            my_pco.send(temp_F1);
            RetransTimer.start
            alt {
                [] my_pco.receive(temp_G1) {RetransTimer.stop; setverdict(pass);}
                [] my_pco.receive(temp_H1) {RetransTimer.stop; setverdict(inconc);}
            }
        }

    }
}
```

TTCN: messages

TTCN 2.GR : definition of ASPs

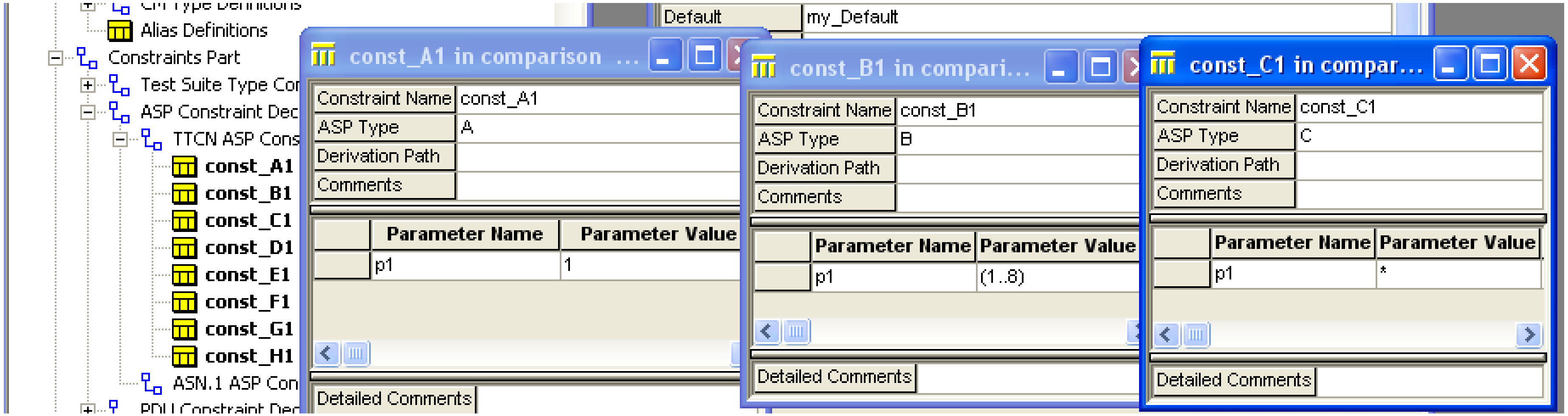


TTCN-3 (types) :

```
type integer A;
type integer B;
type integer C;
type integer D;
type integer E;
type integer F1;
type integer G;
type integer H;
```

TTCN: valuation of messages

TTCN 2.GR : definition of ***constraints***



TTCN-3: definition of ***templates***

```
template integer temp_A1 := 1;
template integer temp_B1 := (1..8);
template integer temp_C1 := *;
template integer temp_D1 := (2..10);
template integer temp_E1 := ?;
template integer temp_F1 := 2;
template integer temp_G1 := (3..7);
template integer temp_H1 := 5;
```

Chapter 1 : basics

- Context
- Common TTCN Concepts
- TTCN language(s)

New features have been introduced in TTCN-3:

- TTCN-3 adds new features which broaden the scope of applications that may be tested:
  - New communication paradigm: procedure-based
  - Allows dynamic test configurations
  - Controlling test case execution possible
  - Standardized Target Adaptation Interfaces
- TTCN-2 knowledge is preserved
  - Testing Concepts of TTCN-2 are also used in TTCN-3

Test Control Notation

It was not possible in TTCN 2 to chain the execution of several test cases. The controlled execution of several test cases could only be described in a tool-dependent maner.

TTCN-3 features a « Test Control Notation » so as to describe the successive execution of several test cases:

```
if (v_TC_Result1 == pass) {
    v_TC_Result2 := execute(tc_ConnReconf_2());
    v_TC_Result3 := execute(tc_ConnReconf_3());
}
else { /* Code inserted here... */ }
```

The Test Control Notation is paramount to make **test automation** possible!

TTCN-3 in different test variants

Applicability of TTCN-3

	Excellent	Good	Could be better...
Conformance testing	☺		
Interoperability testing	☺		
Service, function and feature testing	☺		
Performance testing		😐	
Stress, robustness and load testing		😐	
Real-time testing			☹
Regression testing	☺		

Chapter 2: cornerstones

- Methodology basics
- Structure of TTCN-3 code

Test purpose

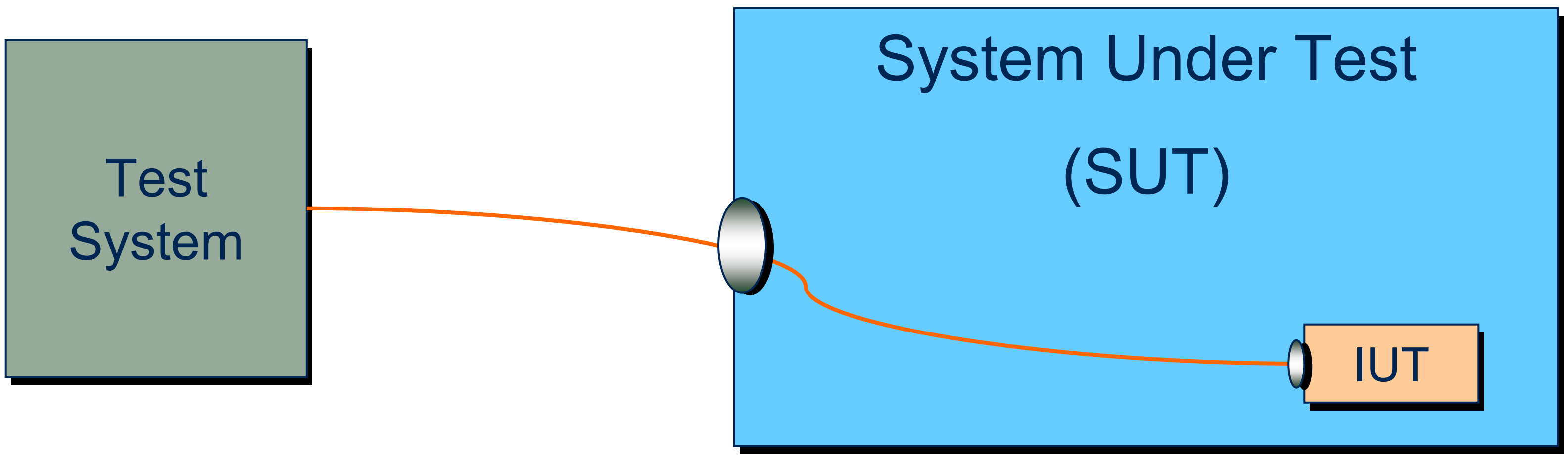
- The test case defines the implementation of the test purpose in TTCN-3
- Typically there is a one-to-one mapping between test purposes and test cases

Test purpose definition according to ISO/IEC 9646-2:

“A prose description of a well defined objective of testing, focusing on single conformance requirement or a set of related conformance requirements as specified in the appropriate OSI specification”

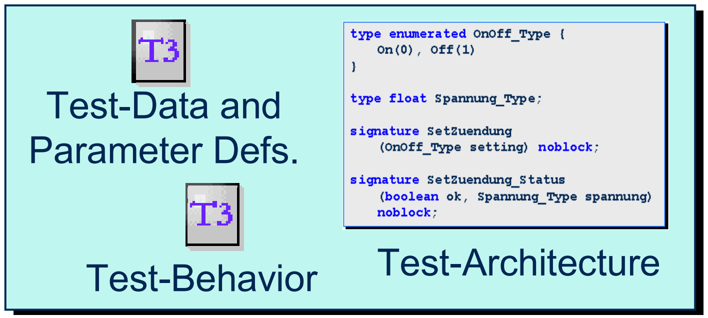
IUT and SUT

- What is the difference between an **I**mplementation **U**nder **T**est (IUT) and a **S**ystem **U**nder **T**est (SUT) ?
  - The IUT usually exists inside the SUT
  - The IUT may only be reachable through the SUT meaning that the Test System will communicate with the IUT through the SUT

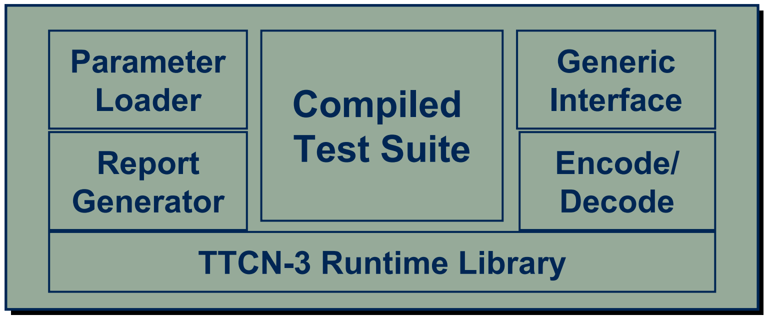


ATS and ETS

- The **A**bstract **T**est **S**uite (ATS) is a test suite written in TTCN-3
- It is abstract: it does not implement communication, data encoding, test report generation, etc.
- The ATS is compiled with a TTCN-3 compiler
- Typically, target adaptation code is added, to implement at least communication and coding
- The result is an **E**xecutable **T**est **S**uite (ETS)



Abstract Test Suite

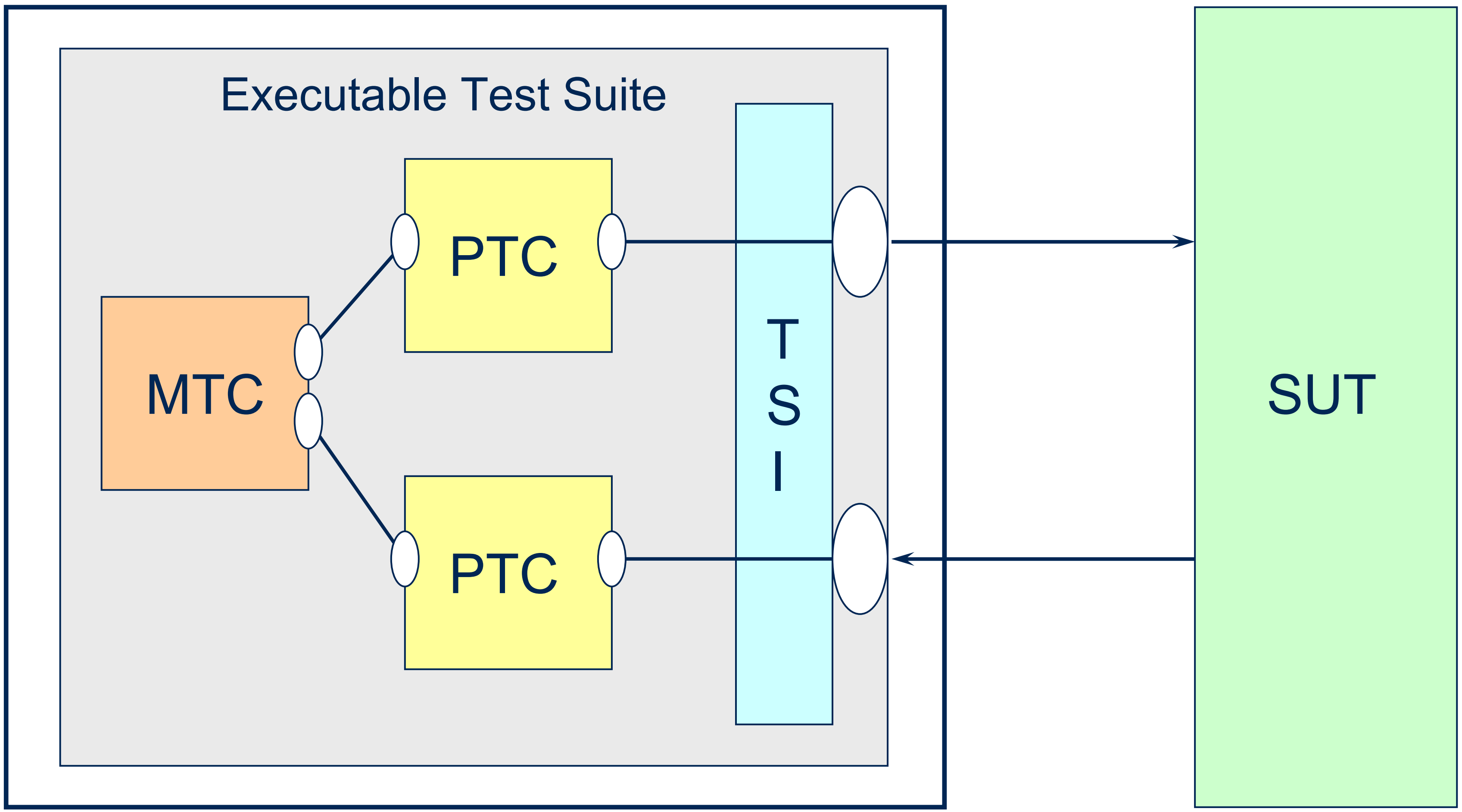


Executable Test Suite

Test component

- The test components are entities on which behavior can be executed
  - For each defined behavior which contains communication statements the type of the test component on which the behavior is executed must be specified
- Test components run in parallel
- There are three kinds of test components:
  - MTC – **M**ain **T**est **C**omponent defines the main controlling process in the tester, only one MTC may be active in a test case
  - PTC – **P**arallel **T**est **C**omponent, processes that can be created by the MTC or by other PTCs
  - TSI – **T**est **S**ystem **I**nterface, a component which provides an abstract interface to the SUT

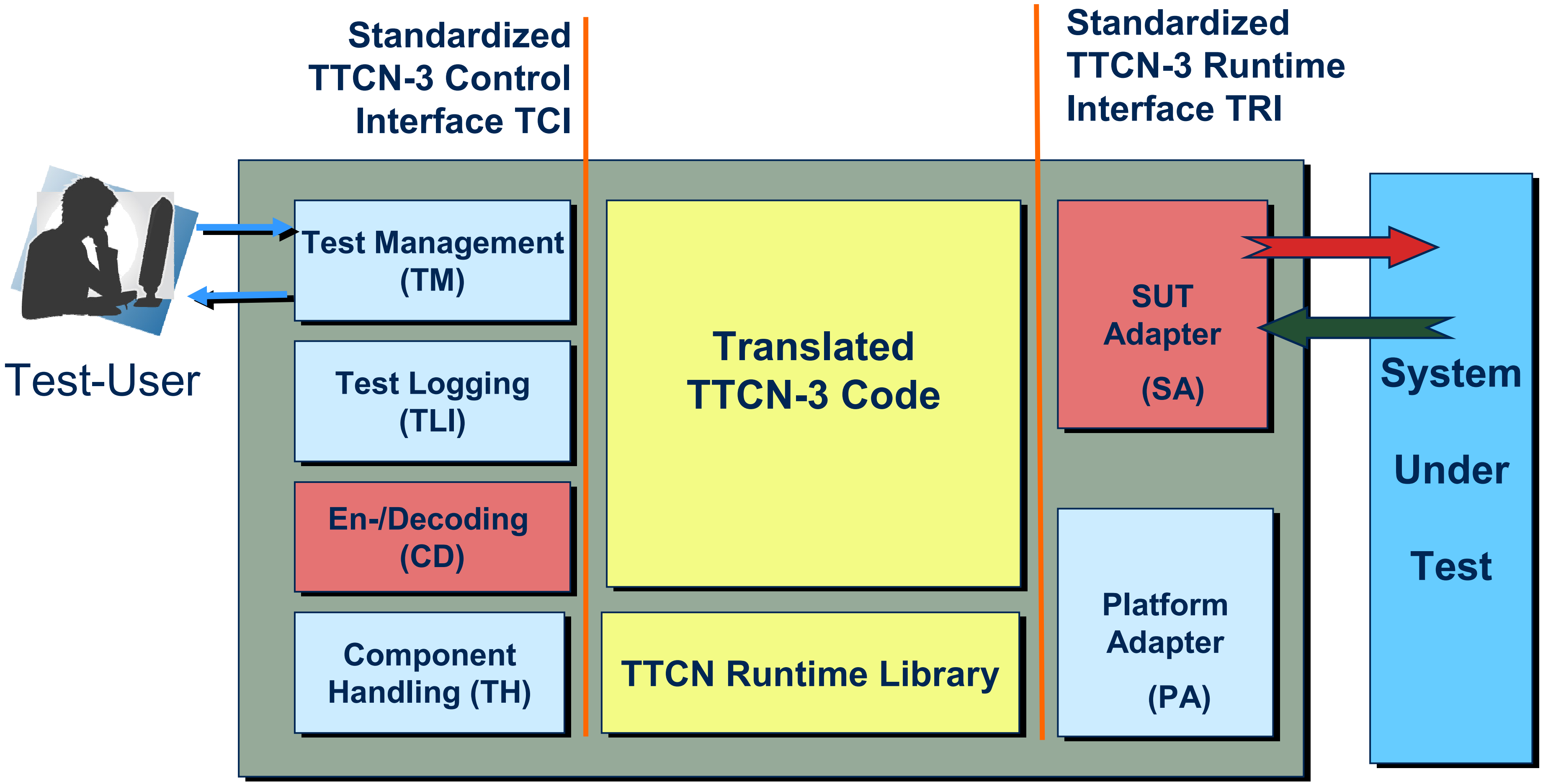
Test components in the test system



The necessary interfaces

- When executing test cases the ETS must be connected both to the user and to the SUT
- The **T**TCN-3 **C**ontrol **I**nterface (TCI) connects the TE to the test user
  - TM (test management): library offers functions for parameterization & test execution. Can be called from a “main”-function, that is connected to the user interface
  - CH (component handling): callback routines, for creating, and terminating components.
  - Tli (test logging interface): routines that can be implemented. They are called when specific events occur
  - CD (coding and decoding): routines that must be implemented. Coding and decoding values into bitstrings, to be used in the system adapter.
- The **T**TCN-3 **R**untime **I**nterface (TRI) connects the TE to the adapters
  - SA (SUT adapter): communication with the SUT and connections handling
  - PA (platform adapter): timer implementation and external functions

TCI and TRI interfaces



Chapter 2 : cornerstones

- Basic terminology
- Methodology basics
- Structure of TTCN-3 code

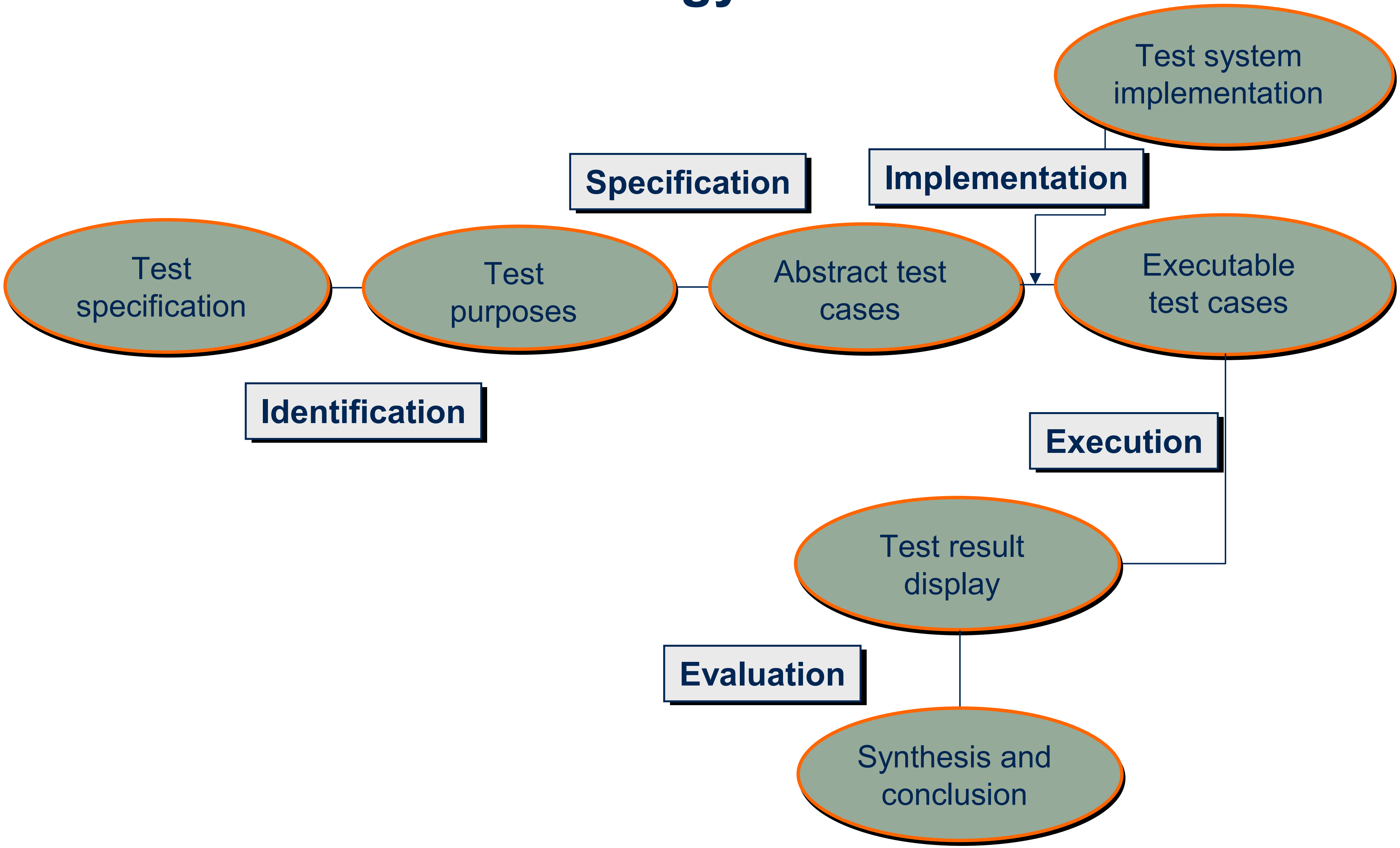
Methodology basics

- How to start a test process from scratch
  - Consider test system architecture
    - For protocol conformance testing current TTCN-2 methodology can be used
  - Start from the requirements of the system we want to test
  - Define test specification and test purposes - TPs
  - Develop test cases that match the requirements and the corresponding TPs
  - Implement and execute the test cases in the real environment
  - Log the achieved the results and draw conclusions based on the information obtained

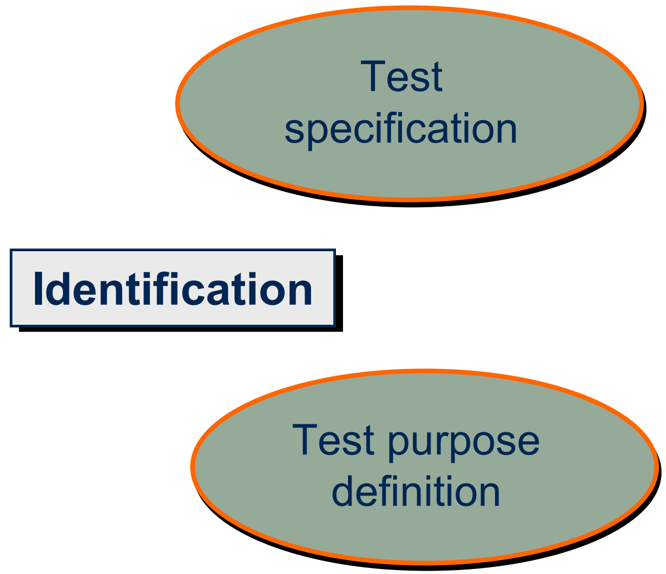
Requirements analysis

- Analyze and check the requirements valid for the implementation we are going to test
- Requirement Management (RM) tools can be used to keep the development process consistent from project initialization to project termination
  - Using proper tools it is easier to verify that all requirements have been implemented and tested

Methodology overview



Test specification and derivation of test purposes



- Derivation of the test specification
  - From a functional aspect of the implementation
  - From a standard point of view
- Defining the Test Suite Structure
- Test purpose derivation
  - The test purpose defines what we are going to test
  - Test purposes should exist prior to developing the test cases
  - Test purposes could be free text, UML SDs, MSC diagrams or other proprietary views

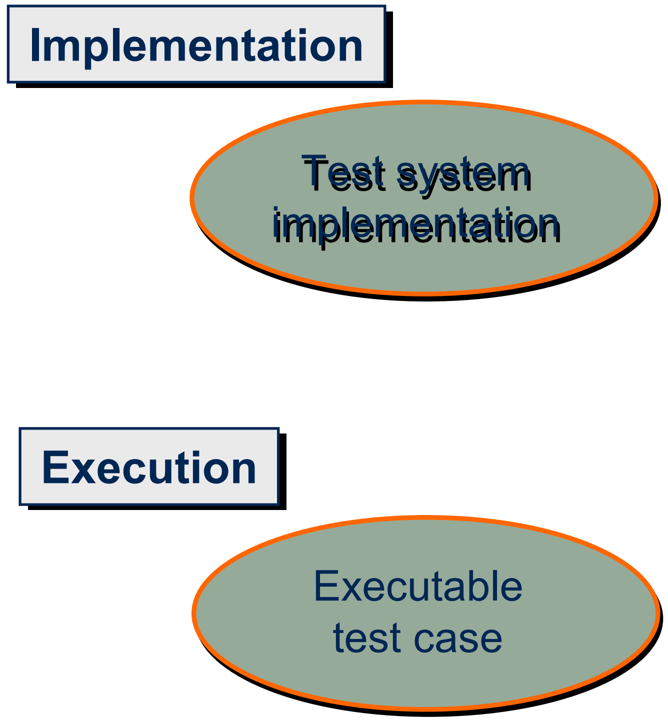
Writing the test cases

Abstract test case definition

- A test case describes the implementation of a test purpose and includes
  - PTC – parallel processes controlled by the MTC
  - Ports – abstract communication with the environment
  - MTC – the main controlling process
  - TSI – interaction on the abstract level with the SUT
- Developed in TTCN-3
  - Use Configuration Management tools (CM) for keeping the development process (e.g. files, projects…) consistent between releases of software and in a distributed development team
  - Link test cases to the requirements for control of the development cycle

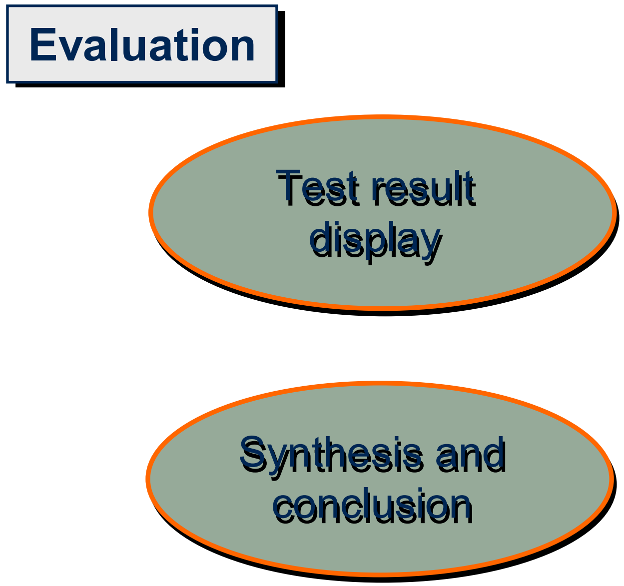
Implementation and execution

- Adapt the abstract code derived in the previous step to the target
  - Involves automatic generation of executable code
  - Adding system specific information to the generated code
    - Timer information, test port information etc.
  - Encoding and decoding of messages
- The generated code (based on the TTCN-3 ATS) makes use of the TRI – TTCN-3 Runtime Interface
  - Standardized interface for making of ETS
- Run the executable test suite on the target in the environment specified



Logging and synthesis of results

- Logging of results from the execution can be done in several ways:
  - Textual logging and graphical logging
  - Post-logging and storage in files/databases and runtime-logging
- After the execution the synthesis of the results is done and conclusions are drawn
- Reviewing of final results/verdicts
- Test report production

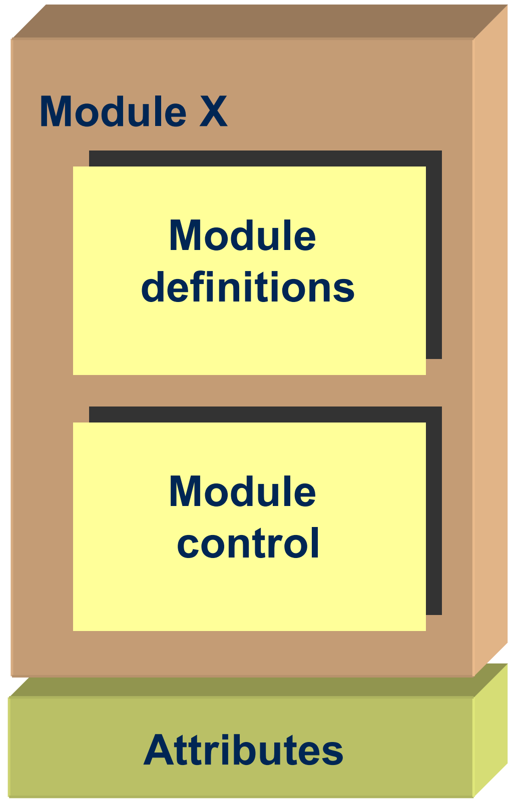


Chapter 2 : cornerstones

- Basic terminology
- Methodology basics
- Structure of TTCN-3 code

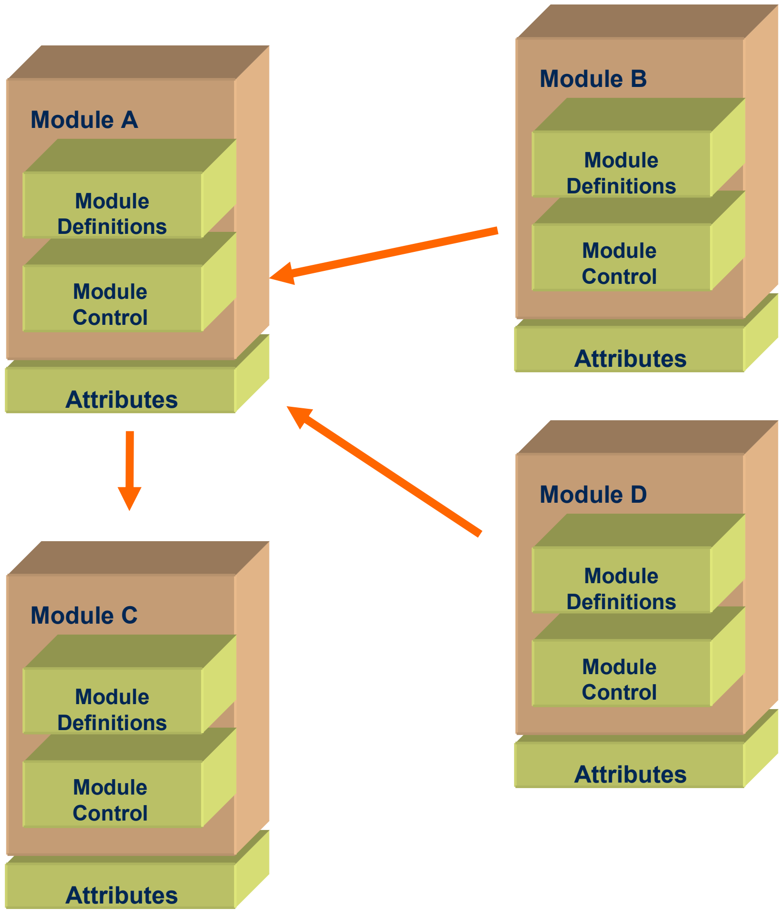
Structure of TTCN-3 code

- TTCN-3 code is placed inside modules
- Modules consist of two optional parts:
  - Module definitions
    - Types, test cases etc.
  - Module control
    - Controlling test case execution
    - One module with a control part per tester
- A TTCN-3 module can also have attributes
  - Display attributes, encoding attributes...



TTCN-3 modules

- Modularization strategies:
  - Provide **common** type **declarations**, constant definitions, functions etc. in a shared module
  - **Hide** the **implementation** of functions and test cases from other modules
  - Design Modularization carefully: **Cyclic include** is **not allowed**



Module parameterization

- Parameters can be used in a variety of definitions in TTCN-3
  - Modules, functions, altsteps, testcases, templates...
- On module level we can define external parameters that can be used to keep the module abstract, i.e. not having to hard-code implementation specific values into the test
  - This was in TTCN-2 related to PICS/PIXIT

```
module MyModuleWithParameters {
   modulepar { integer TS_MaxValue := 42; bitstring TS_TargetAddress }
   // Code inserted here...
   var integer MaxNumberOfAttempts := TS_MaxValue;
}
```

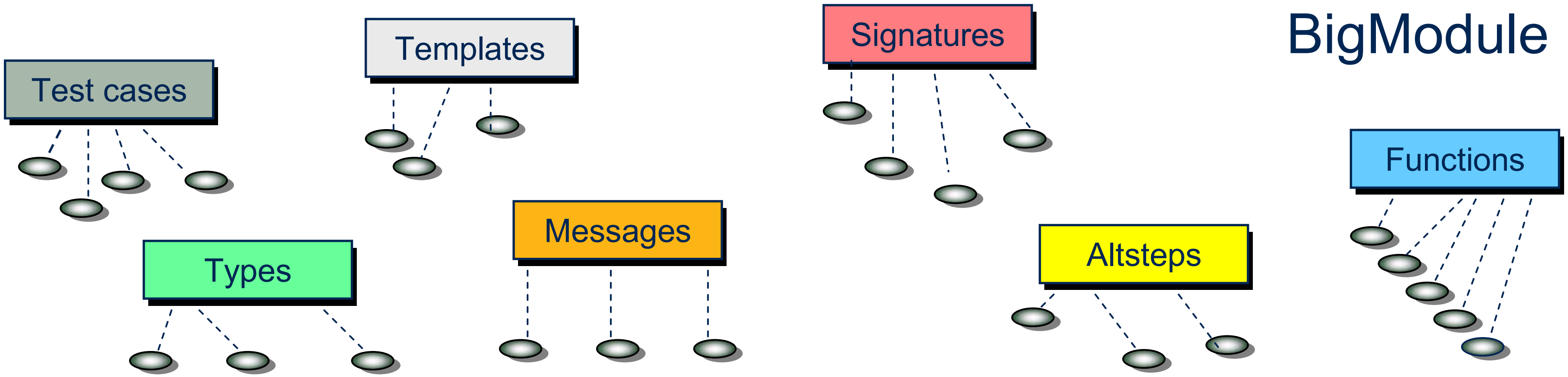
Downward language compatibility

In case of language updates, downward compatibility can be achieved by specifying the TTCN-3 language clause for the modules

```
module Edition1 language "TTCN-3:2001" {
// This module will be translated according to
// Edition 1.1.0 of TTCN-3
...
}

module Edition2 language "TTCN-3:2003" {
// This module will be translated according to
// Edition 2.2.1 of TTCN-3
...
}

module Edition3 language "TTCN-3:2005" {
// This module will be translated according to
// Edition 3.2.1 of TTCN-3
...
}
```

Structuring the code using groups

- Test suites can be hard to read and understand from a structural perspective
  - Modules can be **logically structured** by groups
  - **Hierarchies** of groups of definitions are possible
  - Groups are **not scope units**, but modules are. Use modules as building blocks to define either test suites or libraries
  - All Definitions of a **group may be imported** into other modules



Grouping and comments

- Grouping can be used to structure modules efficiently
- Both block and line comments available

```
module Address_Autoconfiguration {
   group FailureCases {
      group TimeoutCases {
          /* Block comments continue
             over line boundaries. */
      }
      group ErroneusReceptions {
         // This is a line comment. It is
         // terminated by the end of line.
      }
   }
}
```

Chapter 3: data

- Types and Values
- Importing

Scope rules of TTCN-3

- In TTCN-3 there are units of scope which consist of optional declarations and, in some cases, optional blocks of statements
  - Module definition part, control part of a module, component types, functions, altsteps, test cases and statement blocks
  - Scope units are hierarchical
    - Exact hierarchy is defined in the TTCN-3 standard
- Some examples of scope rules:
  - Declarations made in the module definition part outside other scope units are visible throughout the module – e.g. constants
  - Declarations made in the component type are visible to every function, test case and altstep which is executed on that component type

Identifiers and keywords

- Identifiers visible in the same scope shall be unique – overloading of identifiers is prohibited
- Identifiers in TTCN-3 are case sensitive
  - Permitted alphabet: (“a-z”,”A-Z”, “0-9” and “_”)
  - An identifier must begin with a letter
- TTCN-3 keywords are always lowercase

Definition of types, values and templates

- We need to define the messages sent to and received from the SUT, or the signatures called in and received from the SUT
- The data model consists of a types and values
- Messages / Signatures can also be described by templates of specific types
  - A template can be a single value or a constraint (value list, wildcard, character pattern, …) to be used in a receive statement
  - With templates, the test data can be abstracted independently from the message / signature type definitions – important testing concept!
  - No specific "message type" such as a PDU or an ASP type exists

Chapter 3 : data

- Scope rules, names, keywords
- Importing

TTCN-3 types

- The different kinds of types in TTCN-3:
  - Basic types
    - **integer**, **boolean**, **float**, **objid** and **verdicttype**
  - Basic string types
    - **bitstring**, **hexstring**, **octetstring**, **charstring** and **universal charstring**
  - User-defined structured types
    - **record**, **record of**, **set**, **set of**, **enumerated** and **union**
  - Special types for configuration, data and default handling
    - **address**, **port** and **component**
    - **anytype**
    - **default**

Defining types and subtypes - examples

```
type integer ICMPTypeIdType(0..255);
// Allowed value range specified
type integer EvenNumber (2, 4, 6, 8, 10);
// Allowed values specified
type boolean RFlagType;
type float ValidTimerValueType(0.5..0.6);
type record NeighborSolType {
    ReservedNS reserved_ns,
    TargetAddress target_address,
    SourceLinkLayerAddress source_link_layer_address optional
}
type bitstring ReservedNS length(8);
// Length exactly eight bits
type charstring TargetAddress length(16..32);
type charstring IDType (pattern "XY-???-*");
type charstring AthrouthZ ("a" .. "z" );
// also with value lists
```

Defining types – more examples

```
type union NDMessageType {
    RouterAdvType router_advertisement,
    NeighborSolType neighbor_solicitation
}
type enumerated IPVersionType { IPV4, IPV6 }
// Possible values listed
type enumerated IPVersionType { IPV4(5), IPV6(7) }
// Suggested encoding constants for values, not mandatory
type record of integer IntegerListType;
// Dynamic array, of any size
type record length(1..8) of hexstring IPAddressType length(4);
// A list of at least one, maximum 8 hexstrings, each of which
// contains exactly 4 elements
type record ExampleType {
    integer Elem1,
    boolean Elem2
}
```

Value notation examples

```
const float c_PiValue := 3.141592654;

const float c_TimerValue := 500E-3;  // always uppercase E

const ReservedNS c_AllZero := '00000000'B;
// Octet string values in single quotes followed by O,
// hex string values in single quotes followed by H

const charstring c_Hello := "Hello there!";

const IPVersionType ipversion := IPV6;
// Names of enumeration types have global scope!

const IntegerListType intlist := { 2, 3, 4 };

const IPAddressType addr := { 'ABCD'H, '0123'H };
```

Value notation examples - records

```
// Alternative 1
const ExampleType c_ExampleValue1 := {
    Elem1 := 5,
    Elem2 := false
}

// Alternative 2
const ExampleType c_ExampleValue2 :=
  { 62, false }

// Alternative 3
var ExampleType v_ExampleValue3;
v_ExampleValue3.Elem1 := 67;
v_ExampleValue3.Elem2 := true;
```

Value notation examples - unions

```
// Type Definition
type union IntOrFloat
{
    integer i,
    float f
}
```

```
// Alternative 1
const IntOrFloat c_five := {
    i := 5
}

var IntOrFloat v_fivedotone;
v_fivedotone := { f := 5.1 };

// Alternative 2
var IntOrFloat v_sixdottwo;
v_sixdottwo.f := 6.2;

// Anytype
var anytype a, b, c;
a := { integer := 42 };   //Alt. 1
b.charstring := "abcd";   //Alt. 2
c.IntOrFloat := { f := 5.1 };
//Combined
```

Thought question

Would the assignment in the last line be legal, or illegal?

```
// Type Definition
type union IntOrFloat
{
    integer i,
    float f
}

// Anytype
var anytype c;

c.IntOrFloat.f := 5.1
```

Accessing string or ’record of’ elements

```
var bitstring MyString1, MyString2 := '11101'B;
MyString1 := MyString2[3]; // MyString1 has value ‘0’B
MyString2[3] := '1'B;      // MyString2 has value ‘11111’B
// Same notation applies for other string types and the
// "record of" type
// Index 0 of a bitstring is its leftmost bit.

type record of integer RecordOfInt;
var integer MyIntVar;
var RecordOfInt MyRecordOf := { 55, 77, 99 };
MyIntVar := MyRecordOf[1];  // MyIntVar has value 77
MyRecordOf[0] := MyIntVar;  // MyRecordOf is { 77, 77, 99 }
```

Accessing record elements

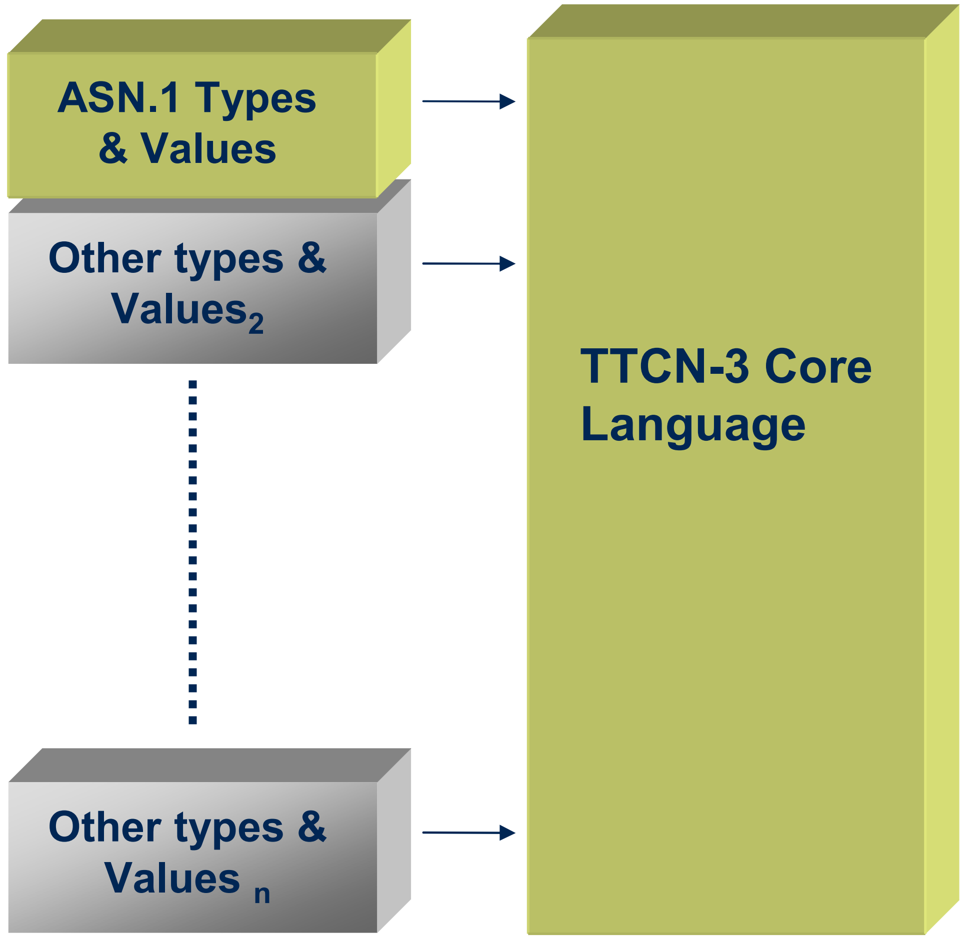
```
var ExampleType MyRecord;
var integer MyIntVar;
MyRecord := {
      Elem1 := 99,
      Elem2 := true
}
MyIntVar := MyRecord.Elem1;  // MyIntVar has value 99
MyRecord.Elem2 := false;  // MyRecord is now { 99, false }
```

Chapter 3 : data

- Scope rules, names, keywords
- Types and values
- Importing

Importing to TTCN-3

- TTCN-3 is harmonized with ASN.1
- Capabilities in the future include:
  - UML, C++
  - IDL, XML
  - Proprietary data types
  - etc.



Importing

- It is possible to reuse definitions from other TTCN-3 or ASN.1 modules by importing them
- Single definitions, groups, all definitions of a certain kind or the entire contents of a module can be imported
- If an imported item uses other definitions, their type names are not imported by default
  - In e.g. a record type the **identifiers** of the elements can be used to set the values of the elements, but the **types** of the elements are not imported by default
- If an item is imported **recursively**, all the type names it contains are imported as well
- Only definitions from the specified module are imported

Importing examples

```
module Address_Autoconfiguration {
// Example of a module with imported definitions

    import from IPv6Protocol all;
    // Imports everything from the IPv6Protocol module

    import from TypeLib {type MyAddressType};
    // Imports of a single definition (nonrecursive)

    import from MessageLib {template all};
    // Imports all templates from module MessageLib

    import from TypeLib recursive {type MyAddressType};
    // All types contained by MyAddressType are imported

    import from ASN1Module language "ASN.1:1997" all;
    // Import all definitions from an ASN.1 -97 module
}
```

Chapter 4: core notation

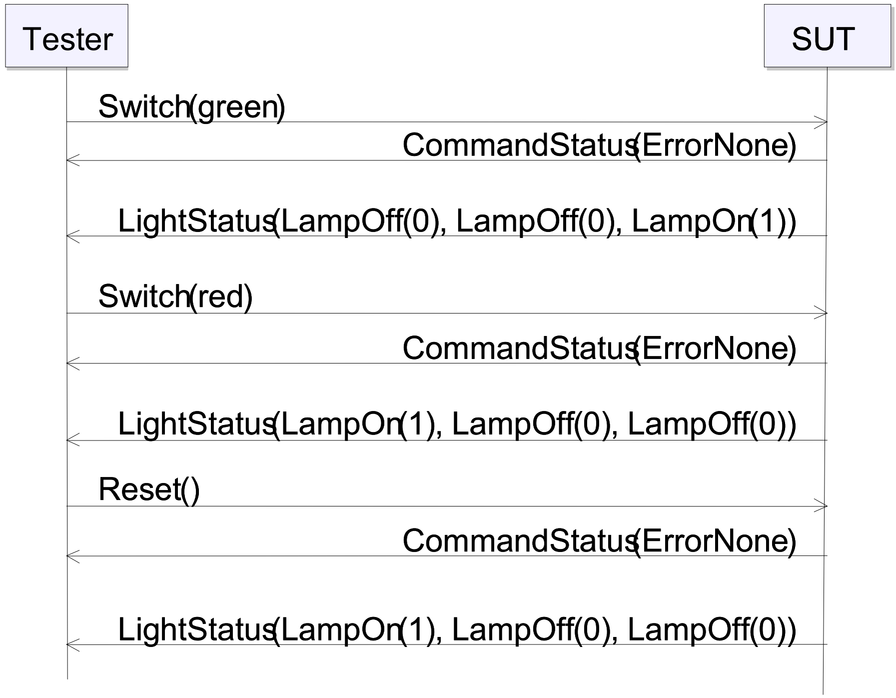
Defining messages

- Templates
- Ports and components
- Timers
- Sending and receiving
- Test cases and verdict

Defining messages for protocols

- Messages are **sequences of bytes**, that are sent from the tester to the SUT, and received in the tester from the SUT

Name	Prefix	Params.
`Switch(green)`	`0x02`	`0x01`
`Switch(red)`	`0x02`	`0x00`
`Reset`	`0x01`	
`Off`	`0x03`	



- Possible messages are defined by TTCN-3 types.

Step 1: parameters

- Read the protocol definition, and detect the various parameter types, and their possible values

```
// Parameter Type of the Switch Command
type enumerated SwitchDirection {
 RedToGreen (0),
 GreenToRed (1)
}
```

Step 2: message definition

```
type union Command {
/*1*/ ResetCommand reset,
/*2*/ SwitchCommand switch,
/*3*/ OffCommand off
}

type record SwitchCommand {
      SwitchDirection d
}

type record OffCommand {
      // no parameters
}

type record ResetCommand {
      // no parameters
}
```

☺ **Codec can be implemented**

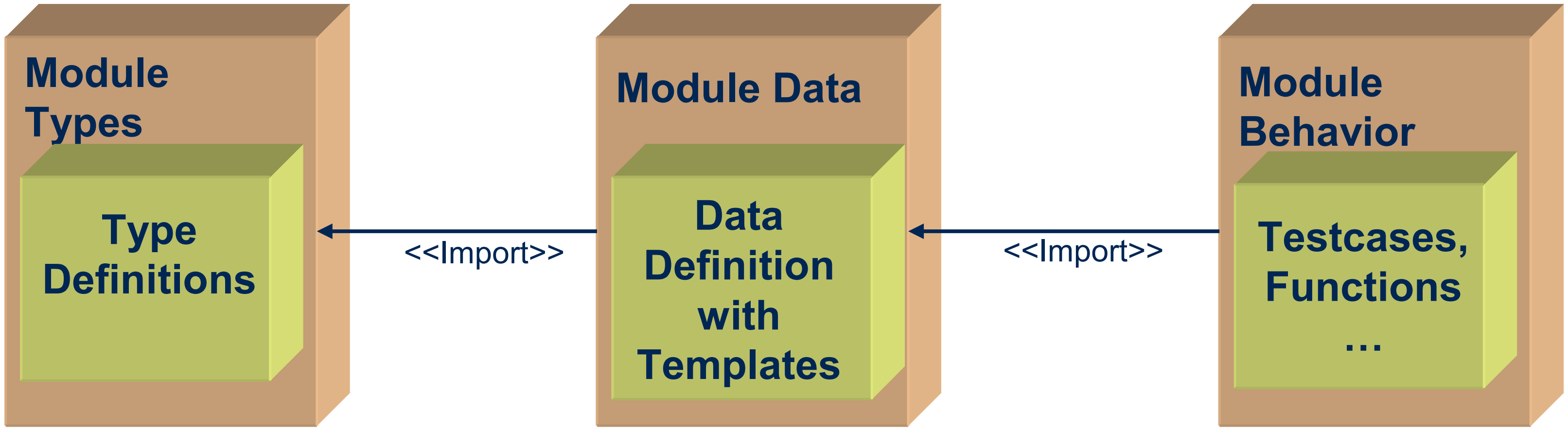
☺ **Codec is fairly easy**

- Encoding and decoding follows a fixed scheme
- Only the command union type may need special handling to map the union names to the prefix codes, and back again.
- Remember: union in ttcn-3 is a type-safe discriminated union, and **not as in c**.

Chapter 4: core notation

- Defining messages
- Templates
- Ports and components
- Timers
- Sending and receiving
- Test cases and verdict

Testing concept: templates



- **Templates**: abstraction for **sent data** and for the **conditions**, received data must fulfill
- Define the data and the conditions **separately** from the types, and the behavior
- Templates can be **developed independently** from the behavior
- Templates can be reused in various test cases

Template definitions

- Templates are used for two purposes:
  - Transmitting a distinct value (must be defined unambiguously)
  - Testing whether a received value matches certain criteria, which are specified in the template definition

- Template definition:

  template <*basic type*> <*temp name*> [(<*params*>)] **:=**
  <*value or matching expression for basic type*>

Templates and values

- Templates can be defined using other templates, constants, specific values
- All elements in e.g. a record type are by default mandatory in TTCN-3
  - The keyword optional can be used to denote possible absence of an element
- If an element is omitted using the omit keyword, no value for this element shall be sent, and must not be received

Template definitions

```
template integer temp_IntegerTemplate := 42;
template float temp_e := 2.7182;

type record NeighborSolType {
    ReservedNS reserved_ns,
    TargetAddress target_address,
    SourceLinkLayerAddress source_link_layer_address optional
}

const ReservedNS c_AllOnes := '11111111'B;

template NeighborSolType temp_NeighborSolicitation := {
    reserved_ns := c_AllOnes,  // Constant used to set value
    target_address := temp_TargetAddress,
                               // Template used to set value
    source_link_layer_address := omit
}
```

Nested template definitions

```
type record rec_a { charstring s, integer i }
type record rec_b { rec a, float f optional }

template rec_b temp_rec_b1 := {
   a := {
      s := "abc";
      i := ?},
   f := 1.5 }

template rec_a temp_rec_a := {
   s := "abc";
   i := ? }

template rec_b temp_rec_b2 := {
   a := temp_rec_a,
   f := 1.5 }
```

- Templates temp_rec_b1 and temp_rec_b2 define the same value

Defining value sets with templates

- In a *message receive event* the contents of the message are compared to a template
- Matching can be done in different ways:
  - Matching explicit values and expressions
  - Matching value sets
    - Templates containing lists of values or complements of them can be matched
  - Matching using wildcards instead of or inside values
    - AnyValue <?>, AnyValueOrNone <*>
  - Matching string patterns
  - Matching using attributes of values, such as length of a string

Matching mechanisms

```
template integer temp_Fifty := 25 * 2;
// The template matches the value of the expression, i.e. 50
template integer temp_SmallPrimes := (1,3,5,7,11);
// The template above matches any value in the set
template integer temp_NoEvenUnderTen := complement (2,4,6,8);
// The template above matches any value NOT in the list
template integer temp_AnyInt := ?;
// The template above matches any integer value
template charstring temp_StartWithFoo := pattern "Foo*";
// The template matches any charstring which begins with Foo,
// e.g. "Foo", "FooBar"... The asterisk can be replaced by any
// number of elements (chars) or by no elements at all.
template charstring temp_StartWithBar_6Chars :=
    pattern "Bar???";
// Question mark inside strings and lists matches exactly
// one element There are more character pattern options
// (not covered in the Paris VII course)
```

Matching mechanisms - continued

```
type record NeighborSolType {
    ReservedNS reserved_ns,
    TargetAddress target_address,
    SourceLinkLayerAddress source_link_layer_address optional }

template NeighborSolType temp_NeighborSolicitation_rec1 := {
    reserved_ns := '11001100'B,
    target_address := ?,                  // Any value accepted
    source_link_layer_address := *        // Any value accepted,
}                                                 // absence OK

template NeighborSolType temp_NeighborSolicitation_rec2 := {
    reserved_ns := ('00000000'B, '11111111'B),
                                              // Either value OK
    target_address := temp_TargetAddress,
                              // Other Template used to set value
    source_link_layer_address := ?          // Must be present,
}                                               // any value OK
```

Template parameterization

- Templates can be made reusable by parameterizing them
- The formal parameters are defined in the template definition
- When the template is used, actual parameters must be included in the reference to the template
- When templates are used as parameters, the keyword **template** must be used in the formal parameter list
- For templates only **in**-type parameters are allowed – no specific keyword is necessary
  - More examples of parameterization are presented later

Template parameterization - example

```
template NeighborSolType temp_NeighborSolicitation_rec1
(TargetAddress param_target) := {
    reserved_ns := '11001100'B,
    target_address := param_target, // Actual parameter used here
    source_link_layer_address := * // Any value accepted, absence OK
}
// Usage of the template in e.g. a test case:
IP.receive(temp_NeighborSolicitation_rec1(c_TargetAddress));
// Usage of the template in other template:
template NeighborSolType temp_NeighborSolicitation_target :=
      temp_NeighborSolicitation_rec1(c_TargetAddress)
```

Formal parameter list

Actual parameter list

- The number of elements in an actual parameter list is always the same as the corresponding formal parameter list
  - The order of the elements must also be the same

Parameterized template definitions

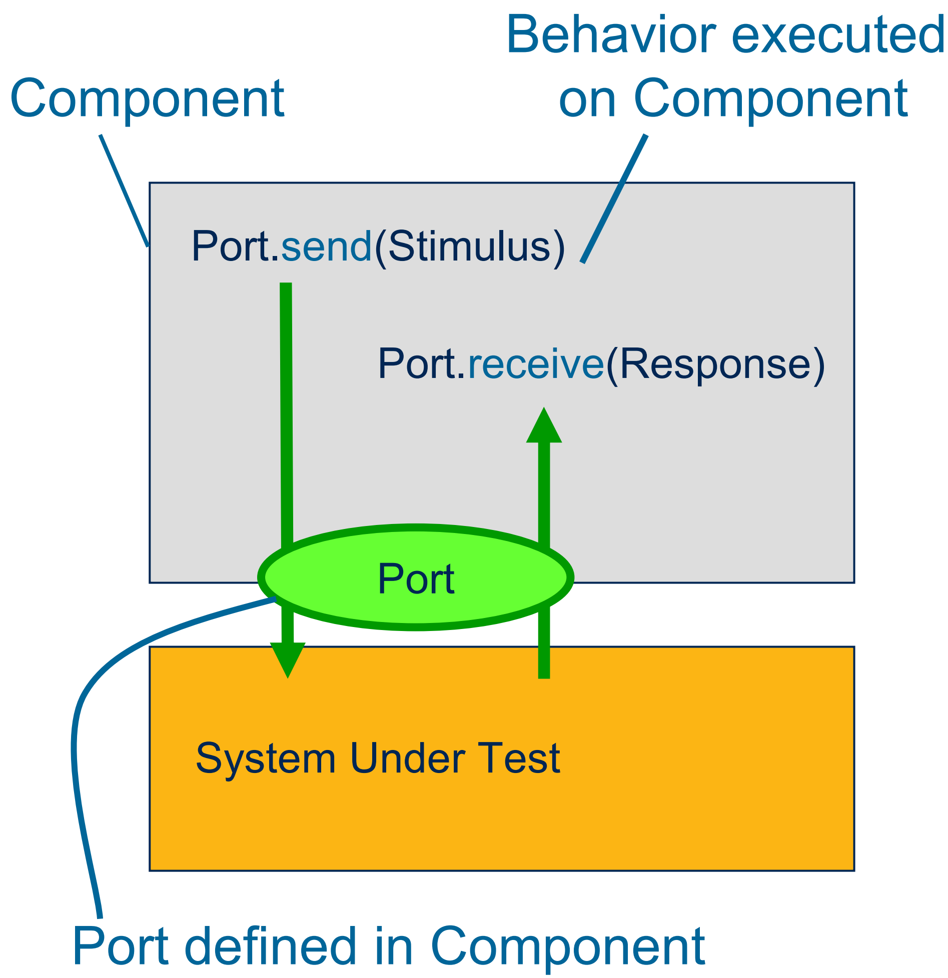
```
type enumerated SwitchDirection { RedToGreen (0),  GreenToRed (1) }
type record SwitchCommand {
      SwitchDirection d
}
type union Command {
      ResetCommand reset,
      SwitchCommand switch
}
template SwitchDirection switchdir := RedToGreen;

template SwitchCommand sw_green := { RedToGreen };
template SwitchCommand sw_green_2 := { d := switchdir };
template SwitchCommand switch_com (SwitchDirection dir) := ?


template Command com_switch := { switch := sw_green };
template Command com_switch_2 := { switch := { d := RedToGreen } }
template Command com_switch_3 (SwitchCommand com) := ?

template Command com_switch_4 (SwitchDirection dir) := ?
```

Chapter 4: core notation

- Defining messages
- Templates
- Ports and components
- Timers
- Sending and receiving
- Test cases and verdict

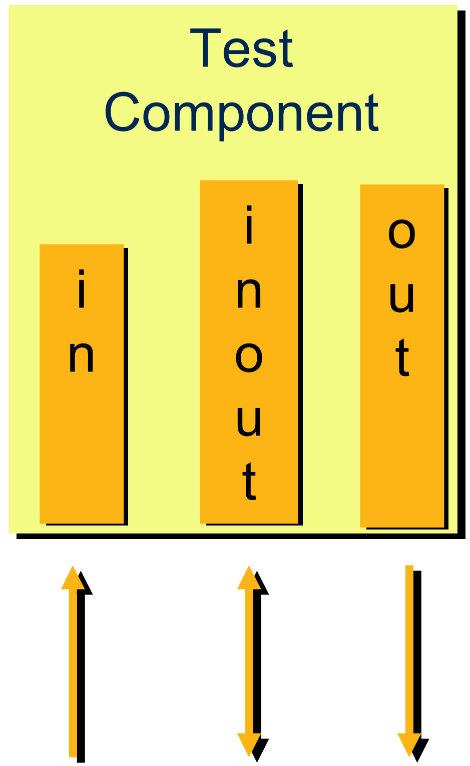
Port and component types

- Recall: test components are the building blocks with which the abstract test suite can be constructed
- There are three kinds of test components, the test system interface (TSI), the master test component (MTC), and the parallel test component (PTC)
- Components define the interface of a test component by declaring ports
- Components may also define data, that is stored locally in the component



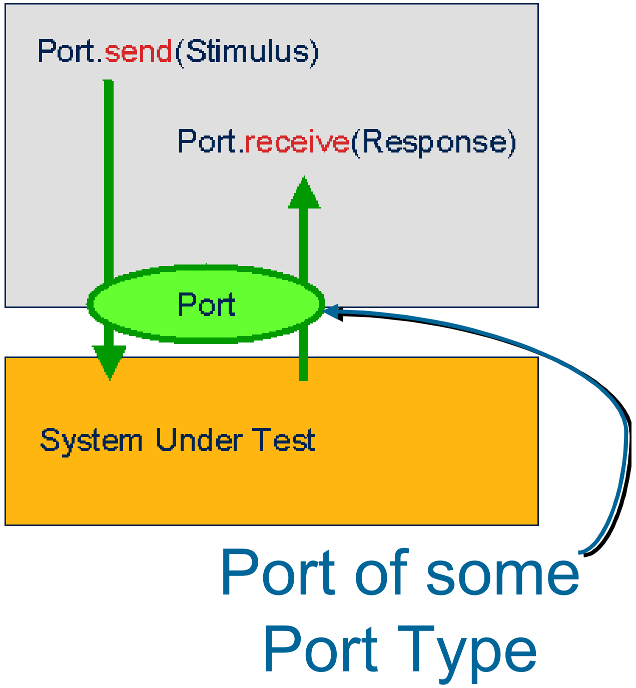
Communication model

- Ports are an abstraction for **any type of interface**
- Communication between different entities in TTCN-3 and the SUT is handled using **ports**
- Ports are defined in components by referring to a **port type**
  - A port type definition specifies the messages and signatures that can be transferred through the port
  - In the port type definition, the direction of the message or signature has to be defined:
    - **In**: messages/signatures, that can only be received through the port
    - **Inout**: messages/signatures, that can be sent and received through the port
    - **Out**: messages/signatures, that can only be sent through the port
  - The incoming messages and signatures in each port are stored in a FIFO (first in first out) queue by the run time system



Defining port types

```
type record NDMessageType ...
type port IPPortType message {
     inout NDMessageType
}
type port IPHostPortType message {
     in RouterAdvType;
     out NeighborSolType, RouterSolType
}
```

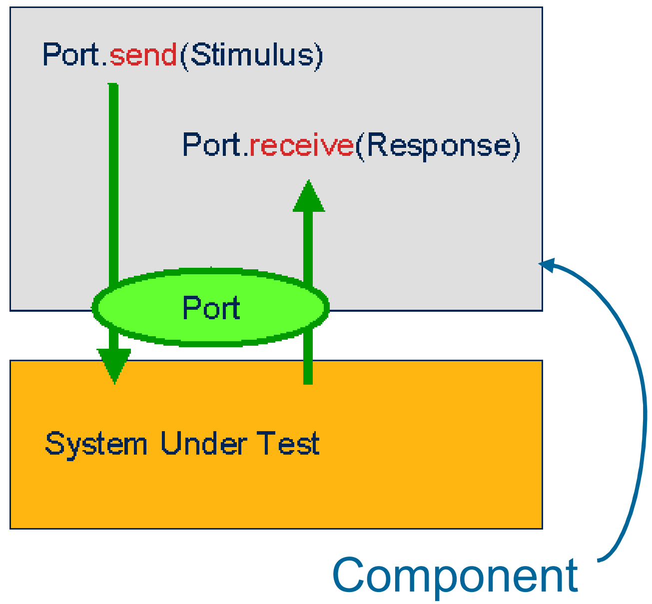


- To define ports in components, port type definitions are needed
- Port type definitions specify the communication model:messages or signatures
- Port type definitions specify the message or signature types, that can be sent and received in each direction

Component type definitions

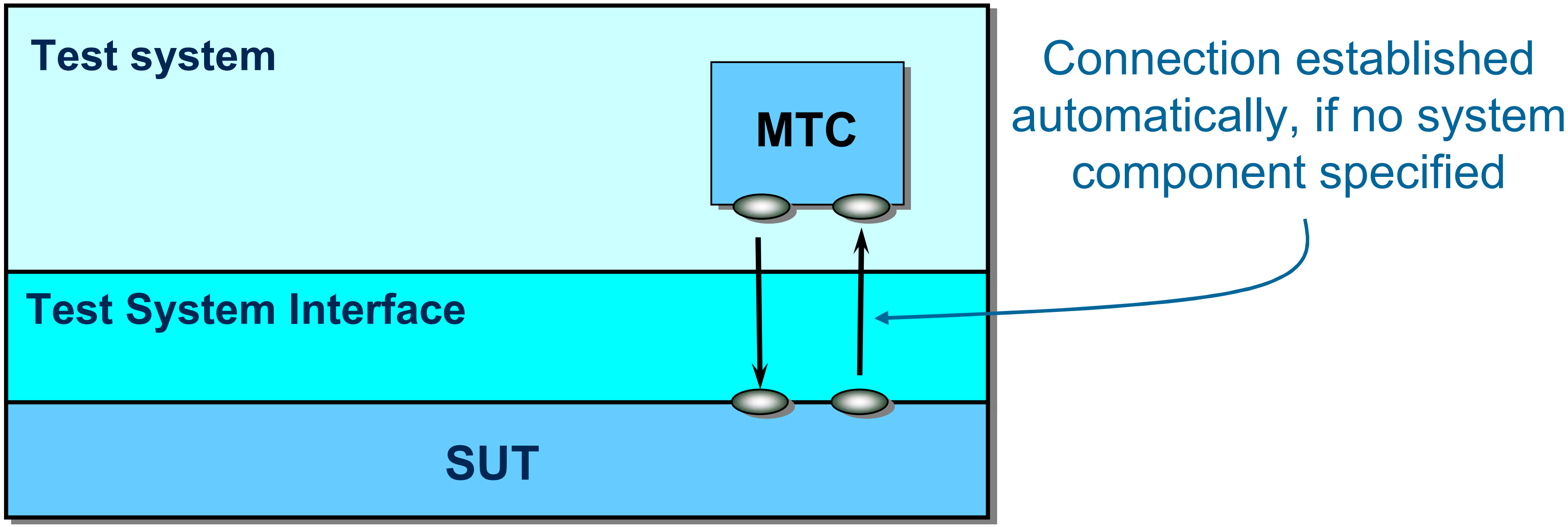
- Components specify the interface to other components by defining ports
- Multiple ports of one port type may be defined
- Behavior is executed in component instances
- A component can have local declarations, which are visible to all functions, test cases etc. running on the component

```
type component HostType {
    var boolean v_Terminate;
    port IPHostPortType IP
}
```



The sequential test configuration model

- In sequential tests, the test system interface is mapped to the ports of a single component, the master test component (MTC)
- The test system interface and the master test components are defined with the same TTCN-3 component definition (in this case)
- No connections needs to be defined between the ports of the test system interface, and the ports of the MTC (in this case)

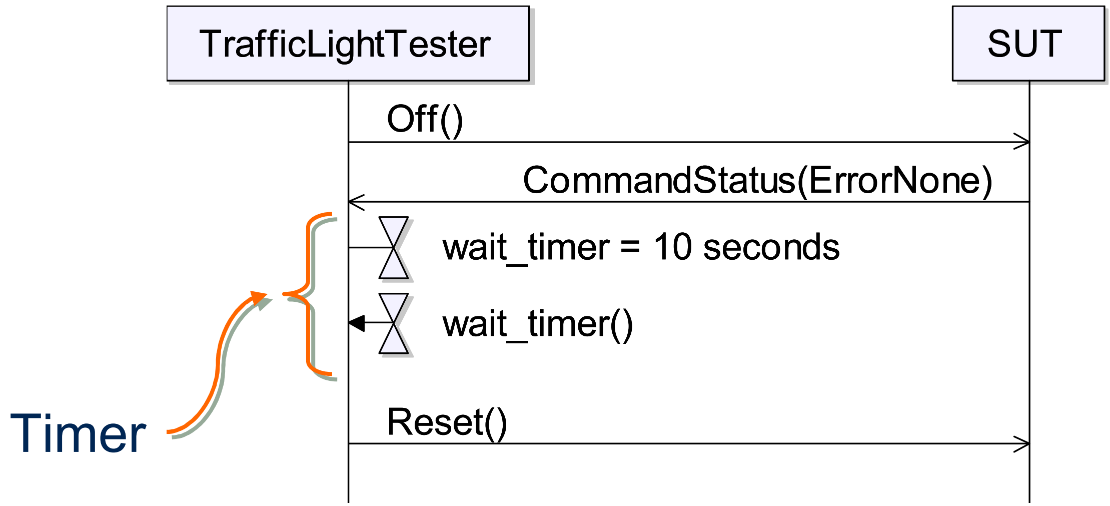


Chapter 4: core notation

- Defining messages
- Templates
- Ports and components
- Timers
- Sending and receiving
- Test cases and verdict

Timers

- Timers can be used e.g. to make sure a message is sent only after a certain amount of time has passed from another event
  - Timers can be declared in test components
  - Other uses for timers will be covered later
- Timer value are non-negative float value
- The base unit is in seconds



Timer operations

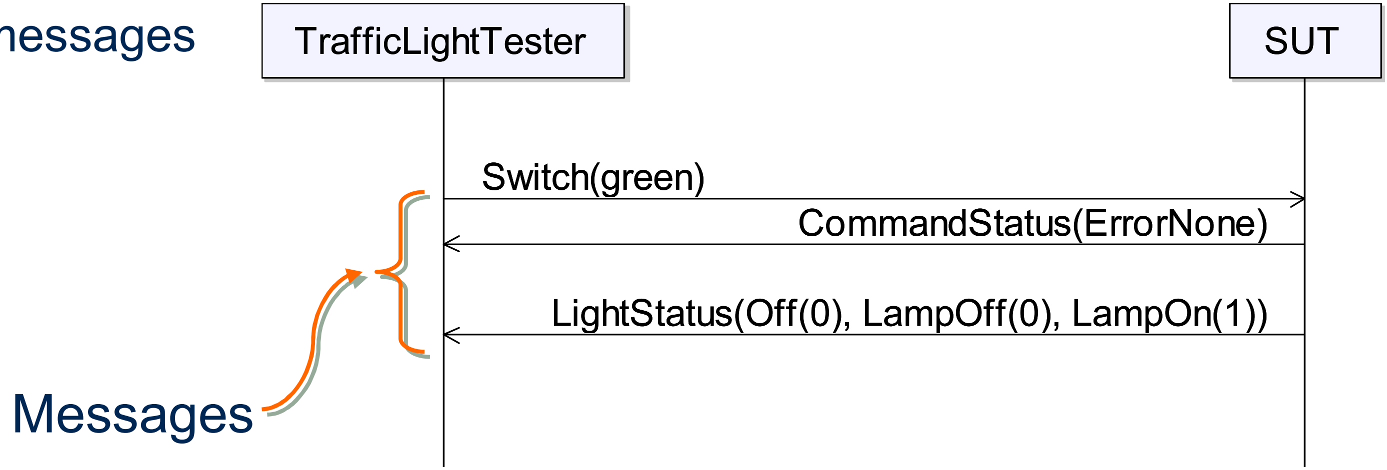
- In each active component a list of running timers and expired timers is maintained automatically
- Starting a timer: <*timer name*>**.start**[(*duration*)]
  - If the timer has no default duration, a duration has to be given in the **start** command
- Stopping a timer: <*timer name*>**.stop**
  - Timer is stopped and its entry is removed from the running timers list
  - Stopping all timers: **all timer.stop**
- Waiting until a timer has expired: <*timer name*>**.timeout**
  - Can only be executed when the timer has indeed expired
  - Waiting until any timer has timed out: **any timer.timeout**
- Checking, if a timer has expired: <timer name>**.running**
  - Returns boolean value indicating if the specified timer is running
  - Checking, if any timer is running: **any timer.running**
- Query elapsed time of a timer: <timer name>**.read**

Chapter 4: core notation

- Defining messages
- Templates
- Ports and components
- Timers
- Sending and receiving
- Test cases and verdict

Message-based communication

- What is message-based communication?
  - Messages are sequences of bits, that are sent to, or received from some other component through an interface
  - The interface is abstracted by a port
  - Messages are abstracted by values that have some type
- How to use it?
  - Sending messages
  - Receiving messages



What is message-based communication?

- Message-based communication has a sending side and a receiving side, the communication itself is asynchronous
  - The sender continues executing its behavior after the send event, but the receiver blocks on the receive event until it can be executed
  - The responding side normally acts as a black-box where messages are handled in the order they appear
- Typical application areas are communicating systems: telecom systems, datalinks, etc

Sending and receiving

- Sending: <*name of the port*>.**send**(<*value to be sent*>)
  - The value in the send statement must be defined unambiguously!

```
IP.send(temp_NeighborSolicitation);
```

- Receiving: <*name of the port*>.**receive**(<*value(s) expected*>)
  - A single value or a template matching a group of values can be specified

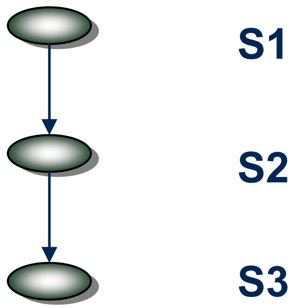
```
IP.receive(temp_RouterAdvertisement);
```

In **send** and **receive**, values of variables, constants, and templates can be used

Chapter 4: core notation

- Defining messages
- Templates
- Ports and components
- Timers
- Sending and receiving
- Test cases and verdict

Sequential behavior

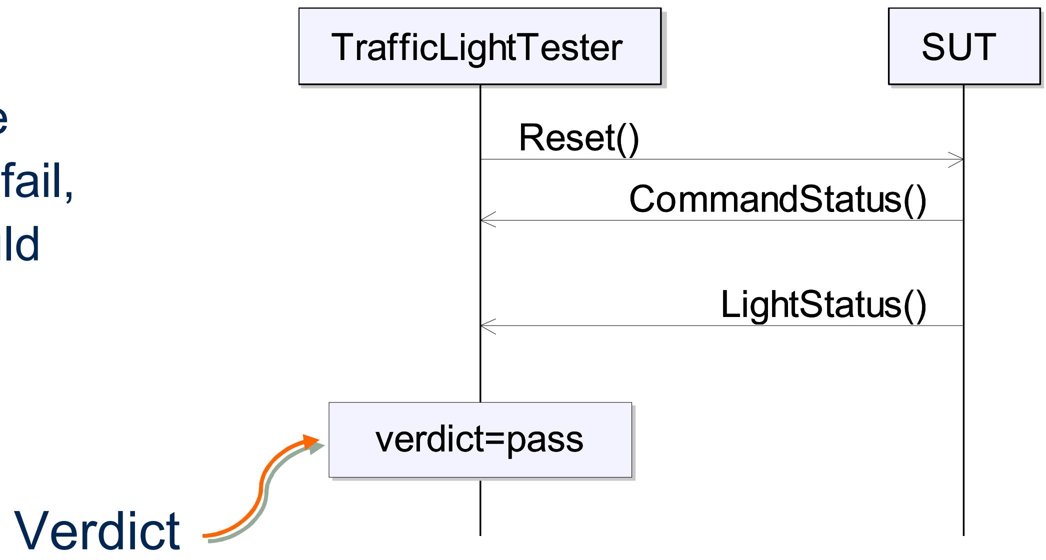



```
S1; S2; S3;
```

- Operations taking place one after another are separated by semicolons
- White space has no semantical meaning

Verdicts

- A verdict is used for displaying the result of an executed test case
- The verdict should be relative to the test purpose, not the actual success of the operation we are testing
  - Example: If the test purpose states the operation should fail, then a failed operation should lead to a pass verdict...



Verdict handling – continued

- The available verdicts are:
  none, **pass**, **inconc**, **fail**, **error**
- In each component a local verdict is maintained and it can be set and read
- A global verdict, which is the value returned by the test case after execution, is automatically maintained
  - The global verdict cannot be read or set
  - The global verdict is updated whenever a component terminates
- A verdict can never be improved
  - When a verdict is updated, the new value is the minimum of the old value and the newly assigned value

Using verdicts

- The verdicts have to be explicitly set using the setverdict operation
  - Syntax: **setverdict**(<*value*>)
  - The error verdict cannot be set using setverdict
- There is also a possibility to get a local verdict of a component using the getverdict operation
  - Syntax: <*variable of verdicttype*> := **getverdict**

```
var verdicttype MyResult;
setverdict(inconc);
MyResult := getverdict;
setverdict(pass);
```

Test cases

- Test cases are special functions which return a verdict
- The runs on keyword is used to define the component type on which the test case can be executed – the ports and variables in the component type become visible to the test case
- The parameter list after the test case name has to be present even if it is empty

```
testcase TC_AA_01() runs on MyComponentType {
    MyPort.send(temp_HelloMessage);
    MyPort.receive(temp_AnswerMessage);
    setverdict(pass)
}
```

Example of a test case

- The **stop** operation stops the component instance on which the test case is running

```
testcase BasicRedGreenRed_Test () runs on
 TrafficLight_MessageInterface {
   ...
   msgport.send(off);                 // Send a message
   msgport.receive(cmd_status(ErrorNone));
                                      // Receive a message
   timer wait_timer := 10.0;          // Declare a timer
   wait_timer.start;                  // Start a timer
   wait_timer.timeout;                // Wait for timeout
   msgport.send(reset);               // Send another message
   msgport.receive(cmd_status(ErrorNone);
                                      // Receive another message
   msgport.receive(lamp_status(red_light));
                                      // Receive a third message
   setverdict(pass);                  // Set a verdict
   stop;                              // Stop the Test Component
}
```

Test case execution

- Test case execution is handled in the control part of a TTCN-3 module using the execute-keyword
- Variables of verdicttype can be used in the control part to store verdicts

```
module MainModule {
  // Import statements not shown here...
  control {
     var verdicttype Result;
     execute(TC_AA_01()); // execute test case
     Result := execute(TC_AA_02());
     // execute and store the verdict
     Result := execute(TC_AA_03(), 5.0);
     // execute, but abort and return an 'error' verdict
     // if not completed within 5 secs
  }
}
```

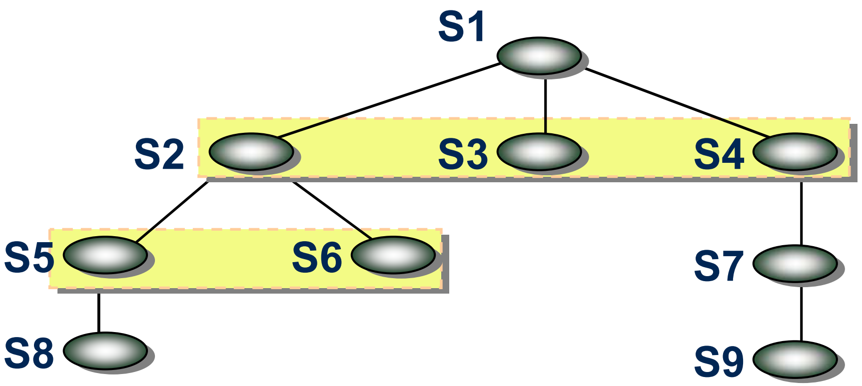
Chapter 5: Advanced Constructs

- Advanced templates
- Advanced communication
- Standard programming constructs
- Functions
- altsteps

Expanding the problem domain

- In the traffic lights example we have covered so far only a sequential flow of action
  - What if messages don't arrive within a certain time limit?
  - How to handle invalid or erroneous messages?
- The following parts expand the problem domain and address more details of the TTCN-3 language

Testing Concept: Alternative behavior




```
S1;
alt {
    [] S2 {
            alt {
                [] S5 { S8 }
                [] S6 {}
            }
        }
    [] S3 {}
    [] S4 { S7; S9 }
}
```

- Use Alternatives to branch automatically between multiple blocking statements (receive, timeout, ...)
- Alternatives are listed in an alt statement
- The square brackets can be used as qualifiers for the alternatives – to be presented later

Alt statement properties

- An alt statement consists of branches, each of which is preceded by a possibly empty guard in square brackets
- Following the guard we have two possibilities:
  - A reference to an altstep – will be explained later
  - A certain kind of operation followed by a statement block
- Only blocking operations are allowed after the guard!
  - For instance message receive or timer expiry
- For a branch to be chosen the guard must be empty or evaluate to true **and** the operation following the guard must be executable

Using alternatives

```
testcase TC_AA_03() runs on
  TrafficLight_MessageInterface {
   msgport.send(switch(GreenToRed));
   msgport.receive(cmd_status(ErrorNone));
   t_GuardTimer.start; // Defined elsewhere
   alt {
         [] msgport.receive(lamp_status(red_light)) {
               setverdict(pass); }
         [] msgport.receive(lamp_status(any_light)) {
               setverdict(fail); }
         [] msgport.receive { // any message
               setverdict(fail); }
         [] t_GuardTimer.timeout {
               setverdict(fail); }
       }
}
```

Testing concept: "snapshot"-semantic

```
alt {
  [] Port.receive(Template1) {
       setverdict(pass);
  }
  [] Port.receive(Template2) {
       setverdict(fail);
  }
}
template Template1 charstring
  :=  "data1";
template Template2 charstring
  :=  ?;
```

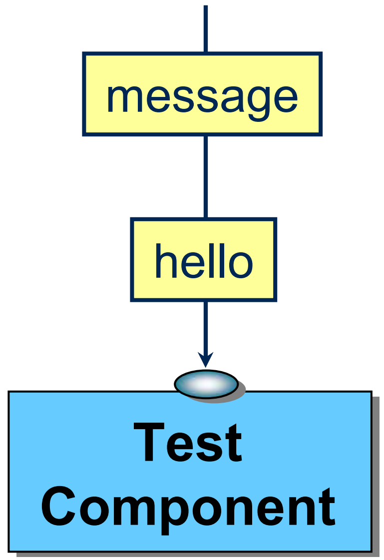
What happens, if a message matching template template1 arrives just before the second alternative is evaluated?

- Snapshot-semantic prevents against strange effects in the execution of alternatives
- Write specific alternatives (good cases) before the more general alternatives (error handling)

Repeat statement

- The repeat statement jumps back to the beginning of the alt statement block
- Useful to get rid of „hello, I am still there“-messages
- Can also be used inside altstep (later)

```
alt {
  [] p.receive(expected_message) {
      setverdict(pass);
  }
  [] p.receive(hello) {
      repeat;
  }
}
```



Guarding alternatives

- An alternative may be preceded by a guard, boolean expression
  - Only alternatives with guards evaluating to TRUE are considered when going through the alternatives
- The alternative guarded by **else** will always be executed
  - If no other alternative matched before, the alternative guarded by **else** will be executed
  - An else branch, if used, must be the last alternative in an alt statement!
  - No operation between an else guard and the statement block!

```
tim.start;
alt {
 [] IP.receive(temp_RA) {
    setverdict(pass)
 }
 [T<1.0] IP.receive(temp_NSol) {
 // T < 1.0 is a guard expr.
    setverdict(inconc)
 }
 [else] {
  // This branch will be chosen
 // when reached
    T := tim.read;
    if (tim.running) {
       repeat;
    }
    setverdict(fail) }
 }
}
```

Interleaving

- With the **interleave** statement we can specify a set of statements which may happen in any order, as long as they all happen
- Valid for **done**, **timeout**, **receive** and **getcall**
  - And for some other statements as well, but these haven't been covered in this course

```
interleave {
    [] MyComp1.done {}
    [] MyComp2.done {}
    [] MyPort2.receive(temp_AllCompleteMessage) {}
}
// All three events must occur in some order before the
// interleave statement is completely executed
```

Interleave replaced by alt statement tree

```
alt {
    [] MyComp1.done { alt {
        [] MyComp2.done {
            MyPort2.receive(temp_AllCompleteMessage);
        }
        [] MyPort2.receive(temp_AllCompleteMessage) {
            MyComp2.done;
        }
    }}
    [] MyComp2.done { alt {
        [] MyComp1.done {
            MyPort2.receive(temp_AllCompleteMessage);
        }
        [] MyPort2.receive(temp_AllCompleteMessage) {
            MyComp1.done;
        }
    }}
    [] MyPort2.receive(temp_AllCompleteMessage) {
        ...
    }
}
```

Chapter 5: Advanced Constructs

- Alternative behavior
- Advanced templates
- Advanced communication
- Standard programming constructs
- Functions
- altsteps

Advanced template constructs

```
template float yellow_to_red_time_ok := (4.25 .. 4.5);
if (match(tim_wait_for_red.read, yellow_to_red_time_ok))
{
    ...
}
```

- Templates can be defined locally, inside a function
- The name of such templates is visible in the function only. The template itself may thus be used as out parameter, or in the return statement of a function
- Template variables can be defined, they can be assigned other templates

Inline templates

- It is cumbersome to define templates for each receive operation in the project. Alternative: use **inline templates** instead

Instead of:

```
template integer temp_any_integer := ?  // Named template
  <port name>.receive(temp_any_integer)
```

You could use an **Inline Template** in the receive statement:

```
    <port name>.receive(integer : ?)      // Inline template
```

Template inheritance

```
type record rec {
   integer a,
   integer b,
   integer c
};

template rec rec_t := {
  a := 10,
  b := ?,
  c := ?
}

template rec rec_any := ?
template rec rec_t modifies rec_any := {
  a := 10
}
```

What if record 'rec' has 20 fields?

What if definition changes ?

Advanced matching mechanisms (1)

Ranges:

(*<from value expression>* **..** *<to value expression>***)**

```
template integer less100 := (0 .. 100); // don't forget ( )
// Specify ranges of integer and floats with dots

const float eps := 0.001;
template float nearly(float val) :=
     (val - eps .. val + eps);
// Also works with expressions

template integer positive := (0 .. infinity);

template integer negative := (-infinity .. -1);
```

Advanced matching mechanisms (2)

```
type record of integer intlist;

template intlist primesunderten := {1, 2, 3, 5, 7};
// This is a list of integers, not an allowed set
// for one integer!

template intlist whatsit := {1, 2, (1, 2, 3), 5, 7 };
// Each list element can be a matching expression
// for the element type. So, List and set can be mixed

template intlist endswith10 := {1, *, 10 };
// * stands for any number of elements or none

template intlist endswith11 := {1, * length (2 .. 4), 11 };
// * stands for any number of elements or none
// but the length specification restrict that further

template intlist permuted := { 0, permutation(1, 2, 3), 4};
// Order of middle 3 numbers is irrelevant
```

Advanced matching mechanisms (3)

```
type record Person {
    charstring name,
    int age optional
}

template Person chriss := { "Criss", 17 };    // must be 17

template Person allan := { "Allan", omit };
                                        // must not be there

template Person Jennifer := { "Jennifer", ? };
                                            // must be there

template Person jane := { "Jane", * };
                                  // may or may not be there

template Person joe := { "Joe", 42 ifpresent };
                                             // 42, if there
```

Matching meta-symbols for patterns (1)

```
template charstring temp_foobar := pattern "Foo*Bar???";
    // * any string (also empty), ? matches one character

template charstring temp_foobar1 := pattern "Foo\*Bar\?\?\?";
    // use backslash to protect against special meaning

template charstring digit := pattern "[0-9]";
template charstring nondigit := pattern "[^0-9]";
    // [abc] one of a, b, or c, [0-9] one of 0, 1, ... 9,
   //  ^ inverts

template charstring digit1 := pattern "\d";
    // \d equiv. to [0-9], \w equiv. to [A-Za-z0-9],
    // \t tab character, \r CR (ascii 13)
    // \n newline characters (ascii 10, 11, 12 or 13)
    // \s whitespace characters (ascii 9, 10, 11, 12, 13, or 32)
    // \b word boundary (beginning or end of a word)
    // \" or "" match double quote char

template charstring digit_or_whitespace := pattern "(\d|\s)"
    // (  ) groups expressions, | alternative expressions
```

Matching meta-symbols for patterns (2)

```
template charstring twenty_digits := pattern "[0-9]#20";
    // #n n repetitions of expression before #

template charstring number := pattern "[1-9] [0-9]#(0,)";
    // #(n,m) at least n, max. m repetitions of expression
    // any of the numbers can be left open, + means #(1,)

var charstring name := "Allan";
template charstring nametemplate := pattern "name={name}";
    // {x} refers variable, modulepar, constant or template x
    // of type charstring or univeral charstring inside the
    // value of the template.


type charstring Digit ("0".. "9");
template charstring digit := pattern "\N{Digit}";
    // \N{x} includes possible values of type x inside the
    // value of the template. x must be a charstring or
    // universal charstring type, subtyped to values
   // of length 1
```

Chapter 5: Advanced Constructs

- Alternative behavior
- Advanced templates
- Advanced communication
- Standard programming constructs
- Functions
- altsteps

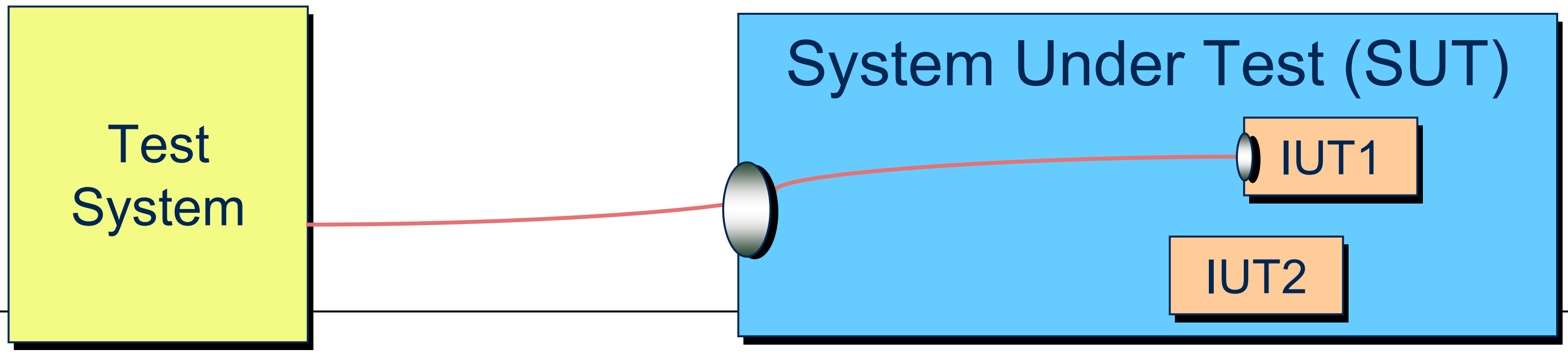
Storing value of received messages

- Values sometimes need to be stored into variables
  - E.G. To use some kind of message id in the reply
  - E.G. Context sensitive checks must be done

```
type record Request {
    integer id,
    charstring variable_name
}
type record Reply {
    integer request_id       // use id from received
request
    charstring variable_value
}

var Request r;
p.receive (Request : {?, "x") -> value r;
p.send (Reply : {r.id, "value of x" });
```

Addressing entities inside the SUT

- Sometimes, different devices with the same functionality are contained in the SUT
- In a test suite, you can
  - choose freely how addresses look like in your test environment
  - can leave the exact address type undefined, and use values only
  - send a message to a specific Implementation Under Test (IUT)
  - receive a message from a specific IUT
  - store the address, where a received message comes from
  - send messages to multiple IUTs
  - send messages to all IUTs



Examples for addressing

```
type record address {  ... }  // use any type definition
var address a1, a2, a3;       // you have to initialize them

p.send (...) to a1;           // send to a1 only

p.send (...) to (a1, a2, a3); // send multicast message

p.send (...) to all component;
                              // send broadcast message

p.receive (...) from a1;      // receive successful if
                              // from a1

p.receive (...) from (a1, a2, a3);
                              // ... if from a1, a2, a3

p.receive (...);              // any adress allowed

p.receive (...) from ?;       // matching expressions allowed

p.receive (...) -> sender a1; // store address of sender
```

Chapter 5: Advanced Constructs

- Alternative behavior
- Advanced templates
- Advanced communication
- Standard programming constructs
- Functions
- altsteps

Operators and expressions

- Expressions can be defined using different operators
  - Arithmetic operators: **+**, **-**, *****, **/**, **mod**, **rem**
  - String operators: **&** (concatenation)
  - Relational operators: **==**, **<**, **>**, **!=**, **>=**, **<=**
  - Logical operators: **not**, **and**, **or**, **xor**
  - Bitwise operators, shift operators and rotate operators: **not4b**, **and4b**, **or4b**, **xor4b**, **<<**, **>>**, **<@**, **@>**

```
v_area := c_Pi * (v_radius * v_radius);
v_Bool := ((A or B) and (not C)) or (j >= 10);
```

Predefined functions

Conversion functions	
int2char	bit2oct
int2unichar	bit2str
int2bit	hex2int
int2hex	hex2bit
int2oct	hex2oct
int2str	hex2str
int2float	oct2int
float2int	oct2bit
char2int	oct2hex
char2oct	oct2str
unichar2int	oct2char
bit2int	str2int
bit2hex	str2float

Length/size
sizeof
lengthof

Presence / choice
ischosen
ispresent

Log functions
log

String Handling
regexp
substr
replace

Other Functions
rnd

- **sizeof** returns the number of elements
- **lengthof** returns the length of a string value
- **ischosen** determines which choice is made in a union
- **ispresent** checks for optional fields
- **log** writes its parameters to a logging device
- **regexp** returns part of string matching regular expr.
- **substr** returns part of string
- **replace** replaces or inserts into string
- **rnd** computes random number

Examples of predefined functions (1)

```
var integer MyIntVar;
var boolean MyBoolVar;

MyIntVar := char2int("a");  // result is 96

type record MyRecord { boolean f1 optional, integer f2 }
var MyRecord MyRecordVar := { omit, 22 }

MyBoolVar := ispresent(MyRecordVar.field1);  // false

MyIntVar := lengthof('11100'B);              // Returns 5
MyIntVar := lengthof("Hi there!");           // Returns 9

type union MyUnion { MyType1 p1, MyType2 p2, MyType3 p3 }
var MyUnion MyUnionVar := {p1 := c_MyType1Value }

MyBoolVar := ischosen(MyUnionVar.p2);        // Returns false
MyBoolVar := ischosen(MyUnionVar.p1);        // Returns true
```

Examples of predefined functions (2)

```
var charstring input := "abbc";
var charstring pattern := "a(b#(1,))c";

var charstring bb := regexp (input, pattern, 1);
// result is "bb"

// substr and replace count from 0
var hexstring cde := substr ('ABCDEF'H, 2, 3);
// result is 'CDE'H

var charstring joe :=
    replace ("My name is Alex", 11, 4, "Joe");

var float justanumber := rnd();
// between 0 (incl.) and 1 (excl)
```

External actions

- Not all interactions with a system under test can be done „automatically“, by sending messages (or remote procedure calls) to the SUT
  - For example, if the operator needs to reset the system under test by pressing some key, or by unplugging the device
- This kind of interaction must be done by the user, that is using the test suite
- The action operation can be called with a character string, or with any other type

```
// Testing some behaviour
...
action ("Please unplug the system under test now");
// Verify behaviour when unplugged
...
action ("Now plug in the system under test again!”);
```

Control structures

- The if-else statement
  - **if** (<*expression*>) *statementblock* {**else if** (<*expression*>) *statementblock*} [**else** *statementblock*]
- The For statement
  - **for** (<*counter initialization*>; <*loop termination condition*>; <*counter update*>) *statementblock*
- The While statement
  - **while** (<*condition*>) *statementblock*
- The Do-while statement
  - **do** *statementblock* **while** (<*condition*>)
- The Select-case statement
  - **select** (<expression>) { **case** (<expression>, ...) statementblock ... }
- *statementblock* is always enclosed in { … }

Control structures – examples (1)

```
if (v_TC_Result1 == pass) {
    v_TC_Result2 := execute(tc_ConnReconf_2());
    v_TC_Result3 := execute(tc_ConnReconf_3());
}
else { /* Code inserted here... */ }
for (var integer i := 0; i < lengthof(v_MyCharString);
     i := i + 1) {
    v_MyCharString[i] := "A"
}
while (v_MyColorVar != blue) {
    ChangeColor(v_MyColorVar);
    // Function call, details presented later
    log ("ChangeColor called!")
}
```

Control structures – examples (2)

```
select (name) {
    case ("Allan") {                // if name is Allan
        log ("Name is Allan");
    }
    case ("John", "Jane") {         // if name is John or Jane
        log ("Name is John or Jane");
    }
    case else {                 // if name is something else
        log ("Name is: " & name);
    }
}
```

Chapter 5: Advanced Constructs

- Alternative behavior
- Advanced templates
- Advanced communication
- Standard programming constructs
- Functions
- altsteps

Repeated behavior and altsteps

- There are two ways of implementing repetitive behavior in TTCN-3
  - Functions as subroutines
  - Altsteps for alternative execution
- Functions are used for preambles, test bodies and postambles when the first statement is an active event from the tester side
  - Sending, starting timers etc.
  - Functions can also be used for passive reasons but...
- Altsteps are used for grouping alternatives when the first statement is a passive event from the tester side
  - Receiving, waiting for timeouts etc.

Definition of functions

- Functions may be used to avoid repeated definition of functionality and behavior
- Functions can be predefined or user-defined
  - Example of predefined functions are conversion-functions e.G. Int2char and size/length-functions like sizeof
- Functions can **return** values or be void
- User-defined functions can be pure calculation functions, but they may also express behavior, e.G. Sending and receiving messages or signatures

Using parameters – some examples

```
function GetLinkAddress(in integer interface_id,
                        in AddType add_type )

 return AddressType {

 ... // definition of the function ...

} // END function GetLinkAddress

template ICMPHeaderType ICMPHeaderData
     (integer Ptype_id, template ICMP_CodeType Ccode,
      bitstring Pchecksum) := {

    type_id := Ptype_id,

    code := Ccode,

    checksum := Pchecksum

  }

testcase tc_ValidBehavior_001() {

    TargetAddress := GetLinkAddress(11, '111100001100'B);

}
```

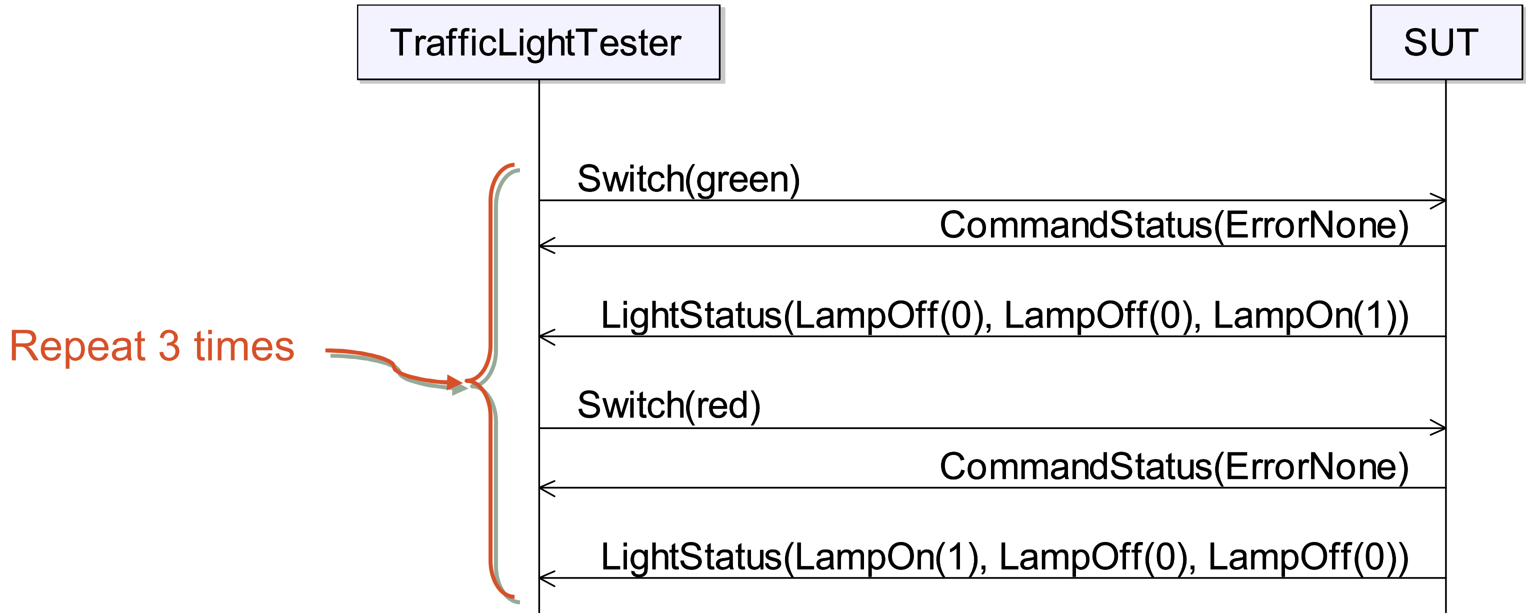
Formal parameter list

Actual parameter list

Using functions for calculation

```
function fn_CylinderVolume(in float radius, in float height)
return float
{
    var float volume;  // Functions may have local variables
    volume := c_Pi * (radius * radius) * height;
    return volume
}
function fn_BitStringCat(in bitstring preAndPostfix,
                         inout bitstring baseString)
{
    baseString := preAndPostFix & baseString & preAndPostFix
}
function fn_GiveTheBigAnswer(out integer theAnswer) {
    theAnswer := 42
}
```

Using functions for communicating - 1




```
testcase TC_AA_08()
runs on TrafficLight_MessageInterface
 {
    RepeatGreenRed(3);
    setverdict(pass);
    stop;
}
```

Using functions for communication - 2

```
function GreenRed() runs on TrafficLight_MessageInterface
{
   msgport.send(switch(RedToGreen));
   msgport.receive(cmd_status(ErrorNone));
   msgport.receive(lamp_status(green_light));
   msgport.send(switch(GreenToRed));
   msgport.receive(cmd_status(ErrorNone));
   msgport.receive(lamp_status(red_light));
} // END function GreenRed

function RepeatGreenRed (in integer nbr )
runs on TrafficLight_MessageInterface {
   for (var integer j := 0; j < nbr; j:= j+1) {
      GreenRed();
   } // END for-loop
} // END function RepeatNSBehavior
```

Using functions to compute templates

```
module my_module {
   modulepar { integer myparam := 10 }

   function compute_template () return template charstring
   {
      template charstring gt10 := "> 10";  // locally defined
      template charstring lt10 := "< 10";

      var template charstring result;
                                         // variable of template type
      if (myparam > 10) {
         result := gt10;
      } else {
         result := lt10;
      }
      return result;
   }
}
```

Chapter 5: Advanced Constructs

- Alternative behavior
- Advanced templates
- Advanced communication
- Standard programming constructs
- Functions
- altsteps

Testing concept: definitions of altstep

- Altsteps are similar to functions but more limited – altsteps consist of a single alternative (alt) statement block
- Altsteps are used:
  - To structure alternatives in an alt-statement
  - To reuse a set of alternatives
  - To express default behavior
- An altstep is denoted using the altstep-keyword

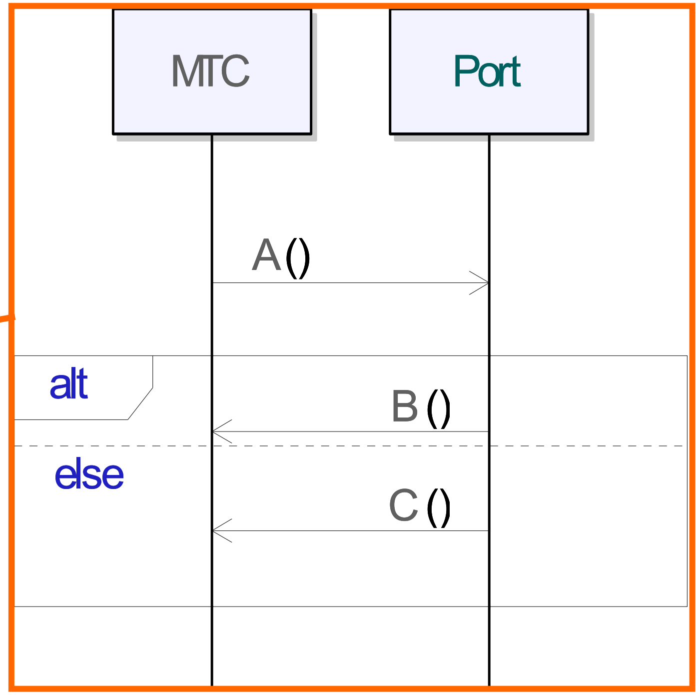
Example of an altstep

```
altstep as_MyAltstep1() runs on MyComponentType {
    [] MyPort.receive(temp_SomeMessage) { // Do something }
    [] any timer.timeout { // Do something }
}
testcase tc_MyTestCase1() runs on MyComponentType {
// Some code inserted here...
  alt {
    [] MyPort.receive(temp_MyMessage) { setverdict(pass) }
    [] as_MyAltstep1()
  }
}
```

Testing concept: default behavior

Default-Behaviour for Error Handling

```
testcase Test() runs on
MyComponentType
{
   var default TestDefault := null;
   TestDefault := activate(
                        DefaultDef());
  Timer.start;

  Port.send(A);
  alt {
    [] Port.receive(B) {
      setverdict(pass);
    }
    [] Port.receive(C) {
      setverdict(pass);
    }
  }
  Timer.stop;
  deactivate(TestDefault);
}
```



- With altstep and default, desired behavior specification is separated from error handling to a large extent
- Specification of Error handling can be re-used
- Direct Reference to the Specification (where Error Handling is usually not covered)

Error handling and defaults

- The SUT possibly responds in an unexpected way, which is why we need to be able to handle erroneous behavior and alternative behavior neatly

- To handle errors default behaviors can be activated (activate) and deactivated (deactivate)
- Variables of default type are used to declare handling of defaults
  - Must be initialized to **null**
  - These variables store a handle to an activated default. They are used in the deactivate statement.

```
// declaring a default variable
var default TestDefault :=
                              null;

// Choosing altstep def as
// default behavior
TestDefault := activate(def());

// deactivating the default
deactivate(TestDefault);

// deactivating all defaults
deactivate;
```

Default details

- A default altstep is appended at the end of all alternatives and also after all other blocking statements, such as message receive and timeout statements
- For default expansion, single blocking statements are treated as alt-blocks with one alternative
- A default is valid from the moment it is activated, until termination of the component or its deactivation, also in all called functions. This is a difference between TTCN-2 and TTCN-3.
- An altstep, that should be enabled as default, must have only “in” parameters (no timers as parameters)
- Multiple activated defaults can be used: for all components there is a list that stores the defaults and their respective activation/deactivation order
- The latest activated default is applied first

Use of default and altstep

```
altstep def_Default() runs on HostType {
    [] RetransTimer.timeout { // timeout RetransTimer
        setverdict(fail); }
    [] any timer.timeout { // checks for any timeout
        setverdict(inconc);     }
    [] IP.receive {
        // checks for any improper message
        setverdict(fail); }
} // END altstep Default
```

Use of default and altstep

```
testcase TC_AA_09() runs on HostType {
    var default defaultVar := null;
    defaultVar := activate(def_Default());
    // Some more code here...

    IP.receive(RouterAdvertisement);
    setverdict(pass); // test OK

    deactivate(defaultVar);
} // END testcase TC_AA_09()
```

Expansion of the default

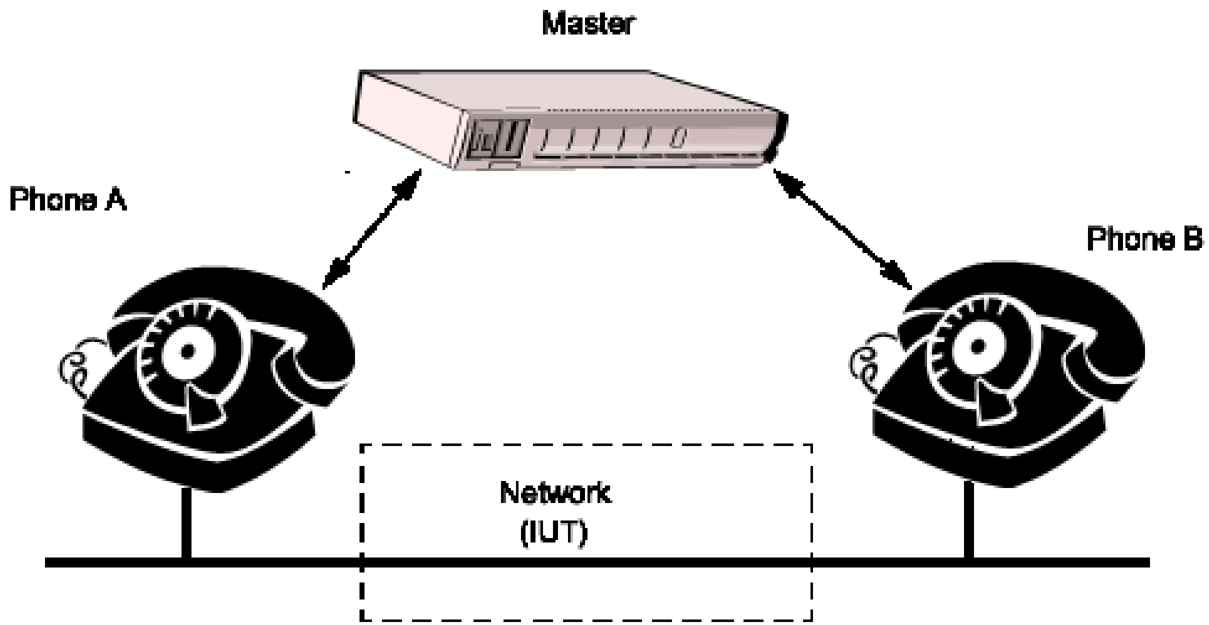
```
testcase TC_AA_09() runs on HostType {
// ...
alt {
    [] IP.receive(RouterAdvertisement) {
        setverdict(pass); } // test case OK
    [] RetransTimer.timeout { // timeout RetransTimer
        setverdict(fail); }
    [] any timer.timeout { // checks for any timeout
        setverdict(inconc);     }
    [] IP.receive {// checks for any improper message
        setverdict(fail); }
    } // END alt
} // END testcase TC_AA_09()
```

Chapter 6: concurrency

- Why testing concurrently?
- Defining the test configuration
- Starting concurrent behavior

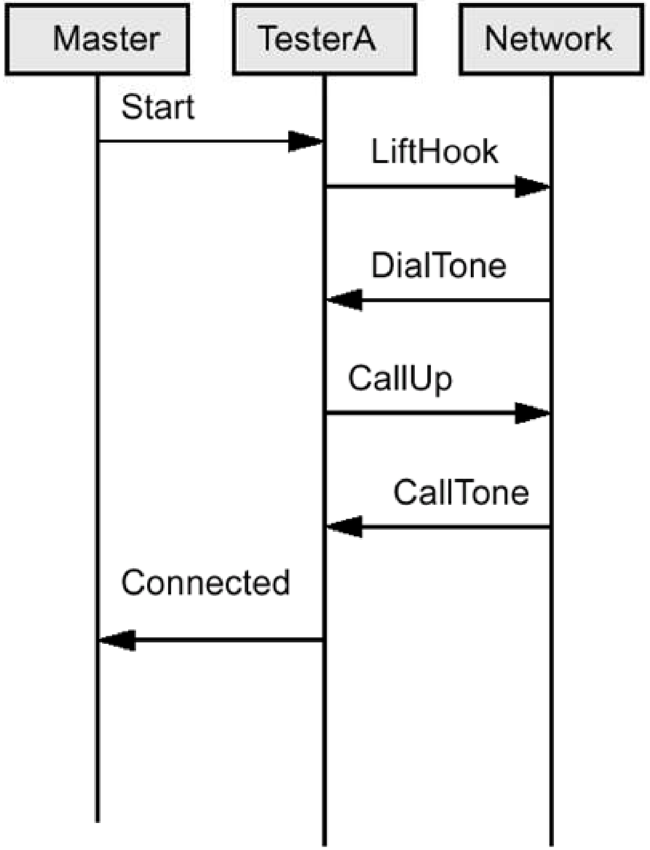
Motivation

- Concurrency is needed when we want to test the SUT using more than one tester simultaneously
- Example scenario: testing a telephone network with two phone connections

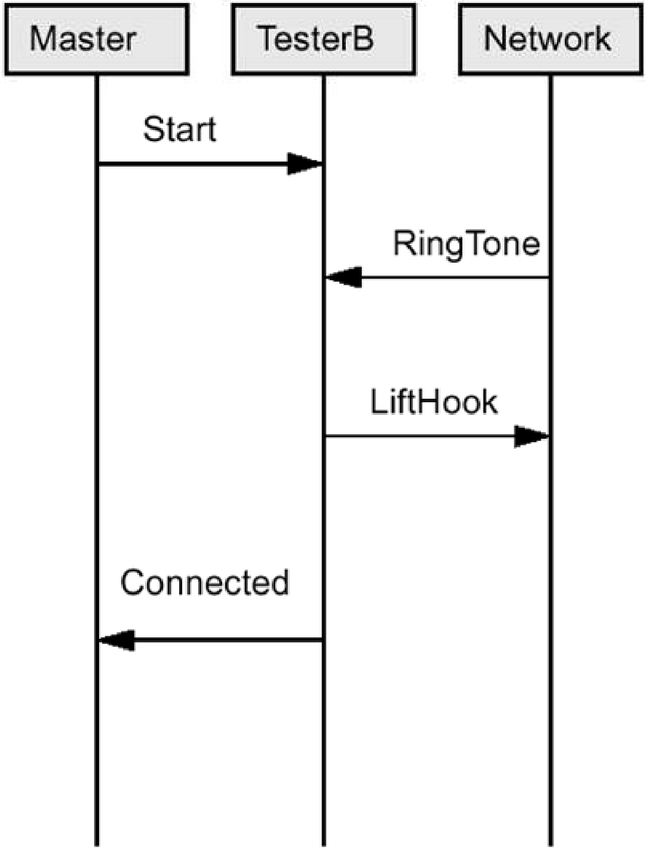


Motivation – test sequences

Test sequence from TesterA

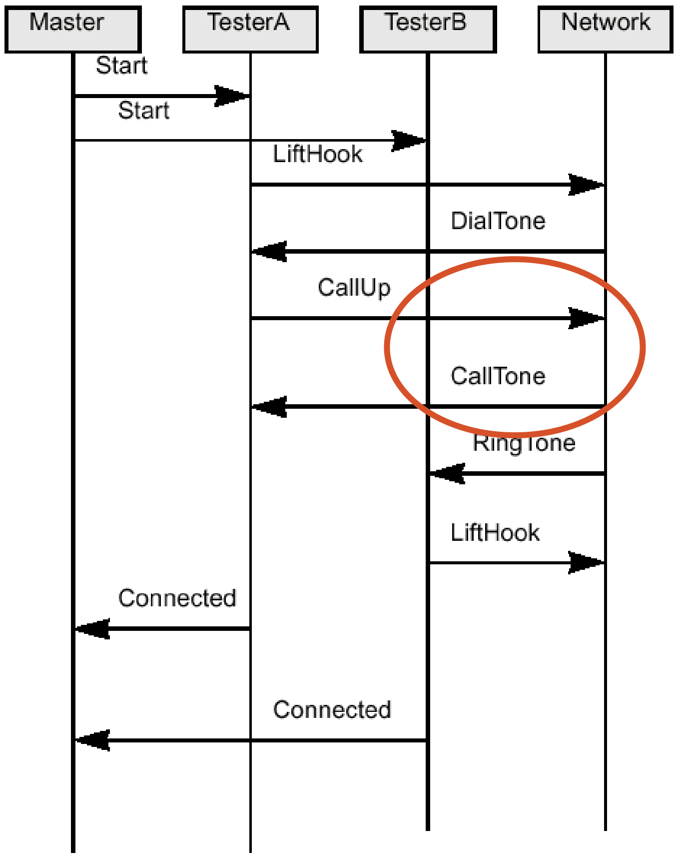


Test sequence from TesterB

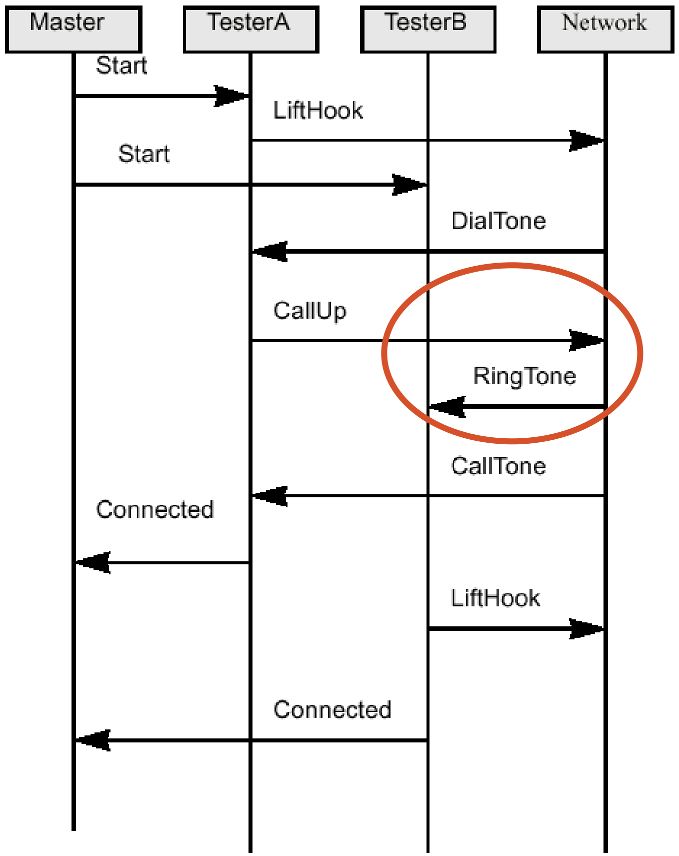


Motivation – different test sequences

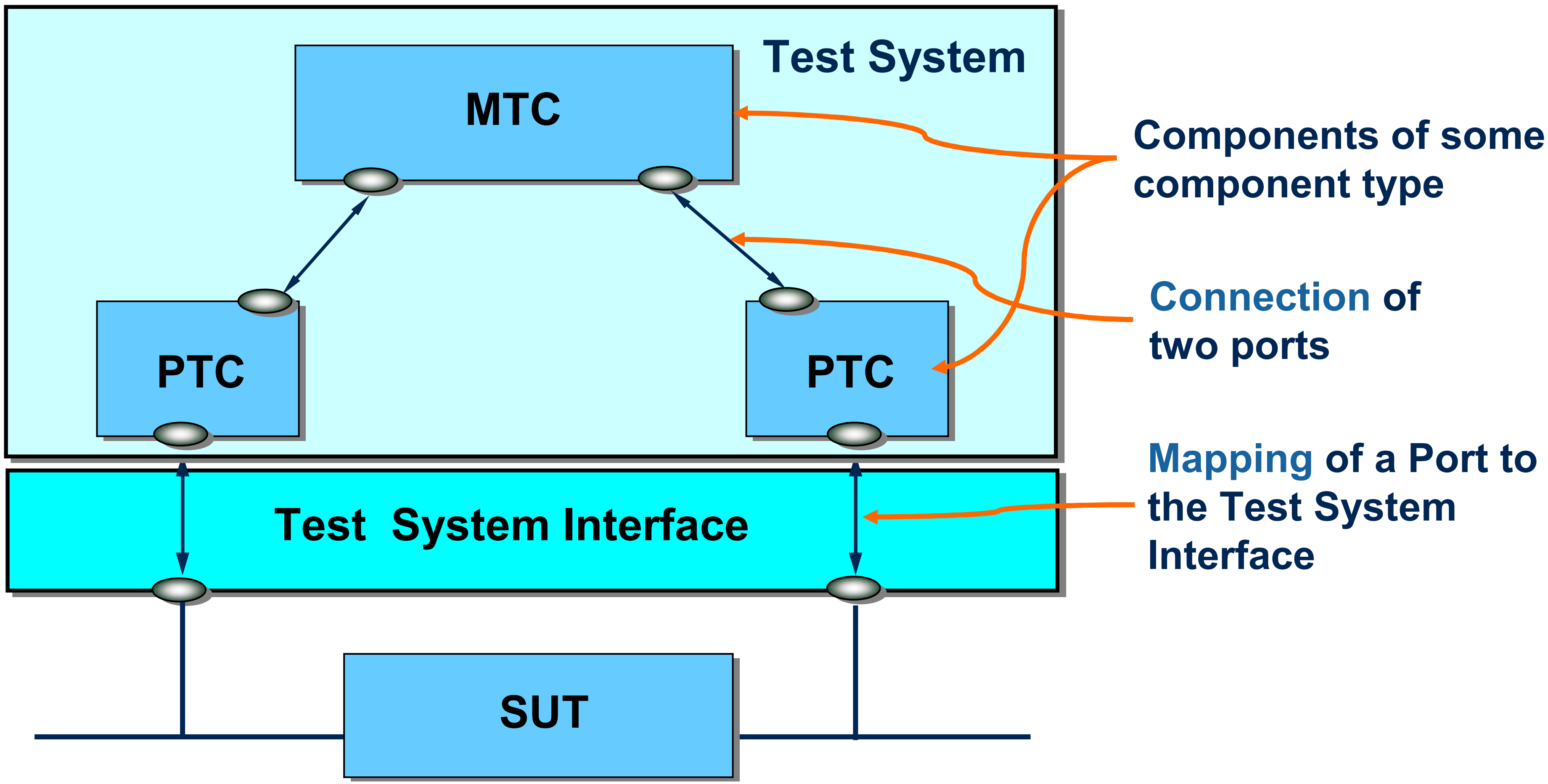
One possible test sequence



Another possible test sequence



Parallel test components



Chapter 6: concurrency

- Why testing concurrently?
- Defining the test configuration
- Starting concurrent behavior

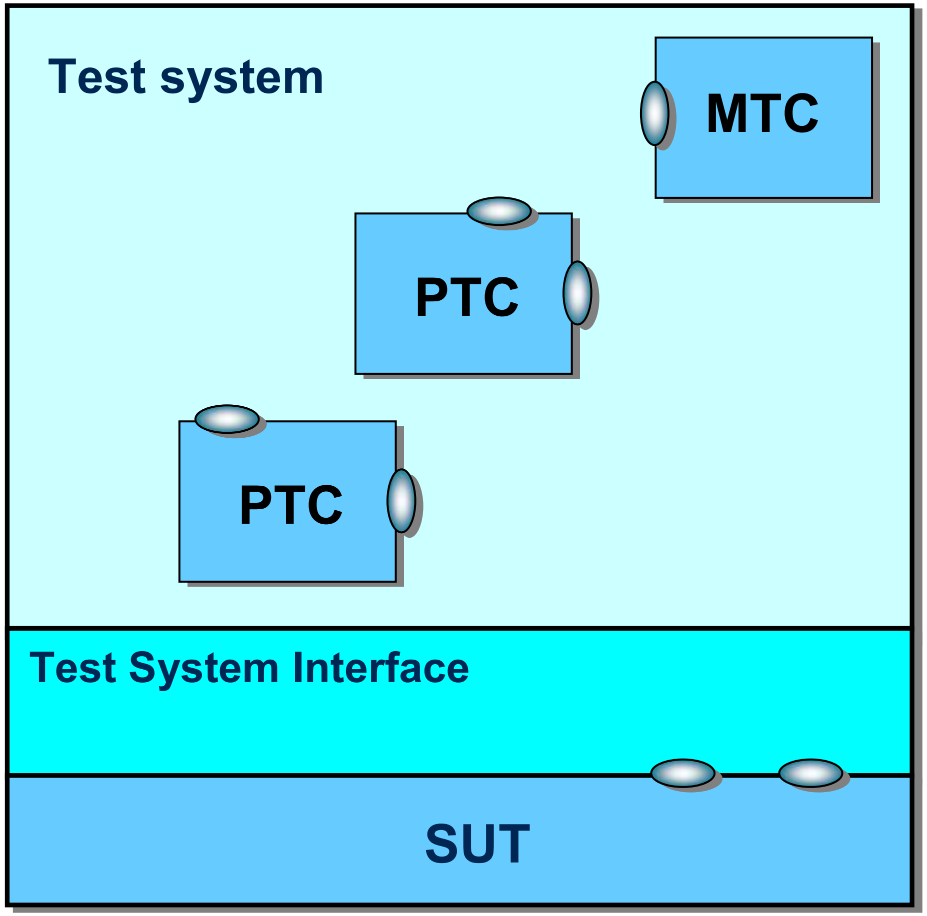
Creation of PTCs and configurations

- The MTC is created automatically when a test case is started
  - The type of the MTC is specified in the test case definition (**runs on**)
  - If no test system interface (TSI) component type is specified, the TSI is the same component type as the MTC. All ports are automatically mapped
  - If a test system interface (TSI) is specified in the test case (**system**), the test system interface differs from the MTC, and ports are not automatically mapped
- Other components must be created explicitly using the **create** command, any component can create new components

```
type component TSIType {
   port IPHostPortType IP_pco1,
   port IPHostPortType IP_pco2
}

type component MTCType {
   port SCP_MTCType SCP
}

testcase TC_AA_06()
   runs on MTCType
   system TSIType {

    // Parallel Test Behaviour
    // goes here
}
```

Creating components

```
type component PTCType {
 port SCP_PTCType SCP;
 port IPHostPortType IP
}
testcase TC_AA_06()
   runs on MTCType
   system TSIType
{
// creation of components
var PTCType Host1, Host2;
Host1 := PTCType.create;
Host2 :=
  PTCType.create ("TesterA");
... // more code...
}
```



Two operation modes for components

- Normal components: components are automatically stopped at the end of the executed behavior function

```
var PTCType ptcname;
ptcname := PTCType.create ("Instancename");
```

- Alive components: components can execute multiple behavior functions (one after the other)

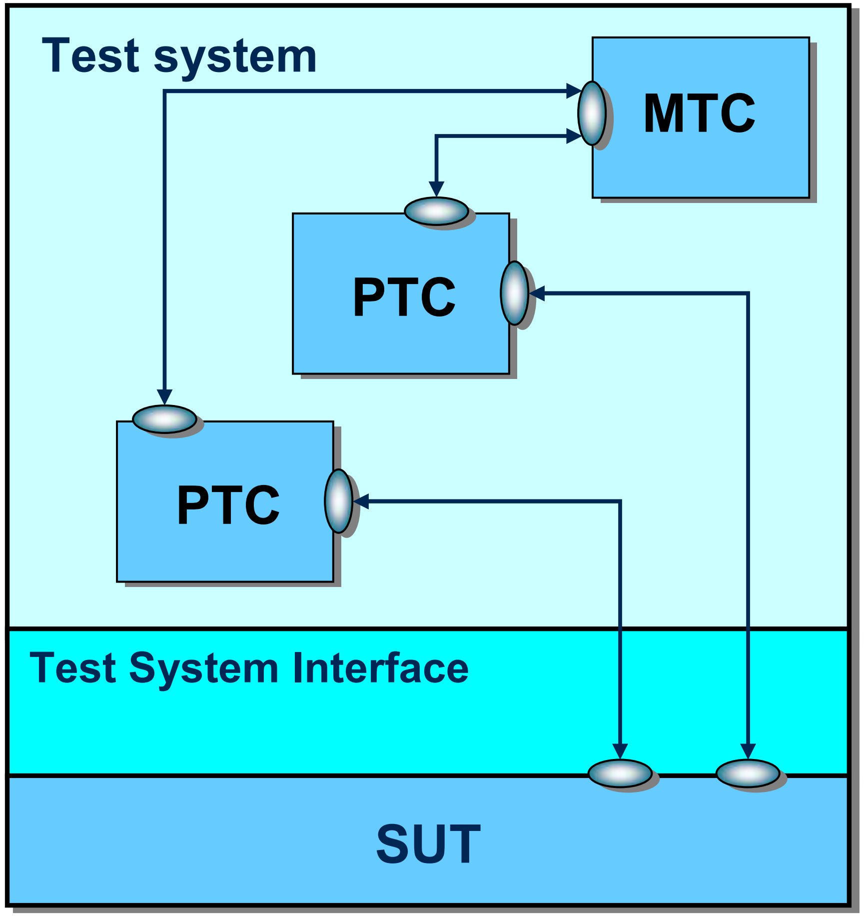
```
var PTCType ptcname;
ptcname := PTCType.create ("Instancename") alive;
```

Connecting and mapping

- After creation of the components we need to **connect** ports between MTC/PTC components and **map** ports between an MTC/PTC component and the test system interface – TSI
  - The **mtc**-keyword identifies the MTC, **system** identifies the TSI instance and the **self**-keyword identifies the currently executing MTC/PTC
- Without connecting/mapping a component cannot communicate with the outside world
- When **connecting** port A and port B, the **in** list of port A must match the **out** list of port B and vice versa
- When **mapping** port A and port B, the **in** list of port A must match the **in** list of port B, and the **out** list of port A must match the **out** list of port B

Connecting and mapping

```
testcase TC_AA_06()
runs on MTCType
system TSIType {
// Creation of the PTCs...
// connecting and mapping
connect(Host1:SCP, mtc:SCP);
connect(Host2:SCP, mtc:SCP);
map(Host1:IP,system:IP_pco1);
map(Host2:IP, system:IP_pco2)
// more code...
}
```



Unconnect and unmap

- Connections and mappings can be undone, to change configuration during the runtime of the test
- Syntax is the same as for connect and map, but shortcuts are available

```
unconnect(Host1:SCP, mtc:SCP);
                    // Unconnects specific connection
unconnect(SCP);     // A PTC unconnects its own port
unconnect;          // A PTC unconnects all its ports
unconnect(Host1:all port);
                    // Unconnects all ports of a component
unconnect(all component:all port);  // Unconnects everything
unmap(Host1:IP, system:IP_pco1);
                    // Unmaps a specific mapping
unmap(IP);          // A PTC unmaps its own port
unmap;              // A PTC unmaps all its ports
unmap(Host1:all port);  // unmaps all ports of a component
unmap(all component:all port);  // unmaps everything
```

Chapter 6: concurrency

- Why testing concurrently?
- Defining the test configuration
- Starting concurrent behavior

Starting test components

- Once components are created and connected/mapped, they can be started
- The behavior to be executed by the component is given in the **start** command
  - The behavior is defined as a function
- Components can also be stopped using the **stop** command
  - Only the execution of test behavior is stopped. Alive components will be ready to execute another behavior function after the operation
  - Non-alive components will be destroyed after the operation
  - Components can stop themselves, or other components
- Components can be destroyed using the **kill** command
  - Does the same as **stop**. Additionally, for alive components, this also destroys the component
  - Components can kill themselves, or other components

Querying test components

- The **running** operation returns a boolean value based on whether the component is running or not
- The **alive** operation returns a boolean value based on weather the component is already executing or ready to execute behavior, or not
- The **done** operation can only be executed when the component has completed its behavior – similar semantics to **timeout**
- The **killed** operation can only be executed when the component has been destroyed – similar semantics to **timeout**

Running concurrent test components

```
testcase TC_AA_06() runs on MTCType system TSIType {
  // Creating the Components ...
  // Mapping and Connecting the Ports ...
  Host1.start(TS_AA_Resend());
  Host2.start(TS_AA_Resend());
  // starting the PTCs with behavior defined in function
 TS_AA_Resend()
  Host1.done; // blocking/waiting until host1 is done
  if (Host2.running) {
    Host2.stop
  }
  stop; // Stops the MTC
} // more TTCN-3 code...
```

Component references

- One local port may be connected to several remote ports (one-to-many mapping)
- Component references can be used to specify to which component a message is sent or from which component we are expecting a message
- Broadcast / multicast with the same syntax as addressing

```
var MyPTCType MyPTC_1;
MyPTC_1 := MyPTCType.create;
// ... Connecting and starting MyPTC_1 not shown ...
MyPort.receive(temp_MyHelloMessage) from MyPTC_1;
// The receive event is executable only if the message
// matches the template and comes from MyPTC_1
MyPort.send(temp_MyAnsweringMessage) to MyPTC_1;
```

Module summary

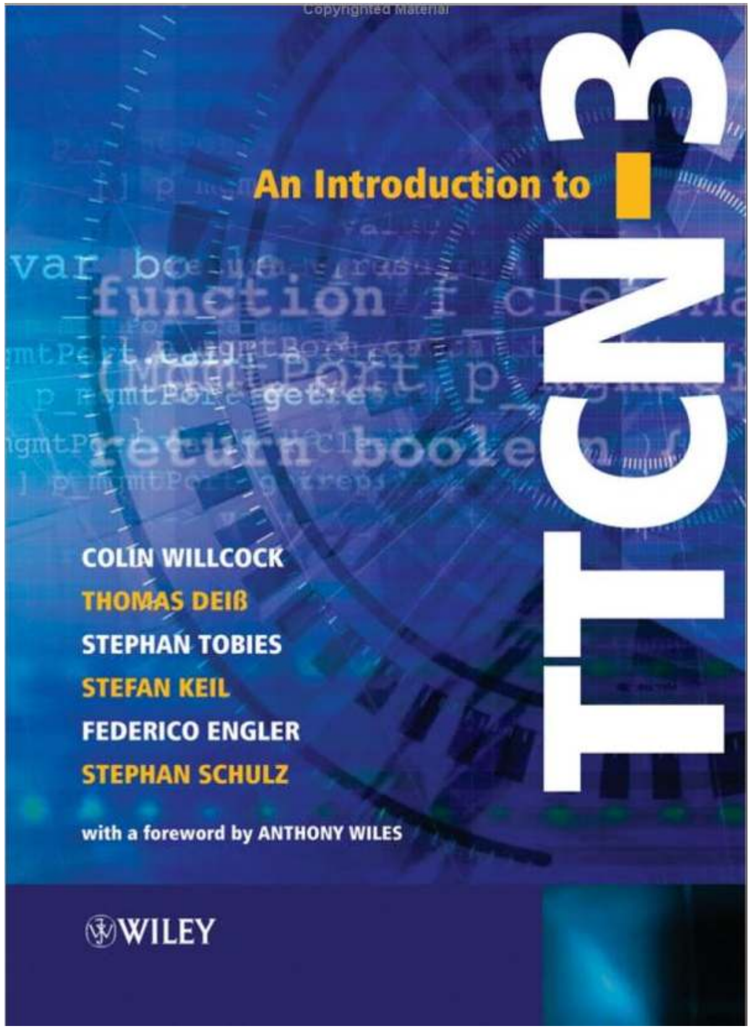
- We can have several components executing simultaneously
- Only the MTC is created automatically
- Other components can be created at any time during the test case execution
- Dealing with components:
  - Creating (**create**)
  - Connecting (**connect**) and mapping (**map**) the ports of the components
  - Starting (**start**) behavior on components
  - Stopping (**stop**) and testing (**running**)

Publicly Available Test Suites

- DMR (Digital Mobile Radio)
  - Standardized Conformance Test Suite, written at ETSI
- Dynamic Host Configuration Protocol (DHCPv6)
  - Conformance Testsuite, written at Fraunhofer FOKUS
  - Comes with Codec, SUT, and Platform Adapter in Java
- IPv6
  - Conformance Testsuite, for the Core part, validated in ETSI IPv6 testbed
  - Other parts (Mobility, Security) will follow
- SIP (Voice over IP with the Session Initiation Protocol)
  - Standardized Conformance test suite written at ETSI
- WiMax (802.16)
  - Ongoing development

Nice to have…

- There exists one off-the-shelf textbook about TTCN-3:



- The TTCN-3 community is very lively, look at http://www.ttcn-3.org/
- There is a regular TTCN-3 user conference, where users meet the language authors and steer the evolutions of the language
- The standard documents can be downloaded from:
- http://www.ttcn-3.org/StandardSuite.htm