megfelel-e a ~~[illegible]~~ szabványnak

KONFORMANCIA TESZTELÉS ÉS A TTCN-3 TESZTLEÍRÓ NYELV

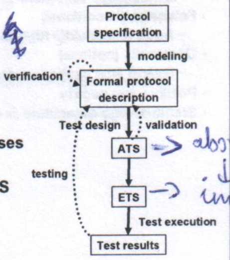
CSÖNDES TIBOR

ERICSSON KFT., TEST COMPETENCE CENTER,
TIBOR.CSONDES@ERICSSON.COM,

BME-TMIT
CSONDES@TMIT.BME.HU

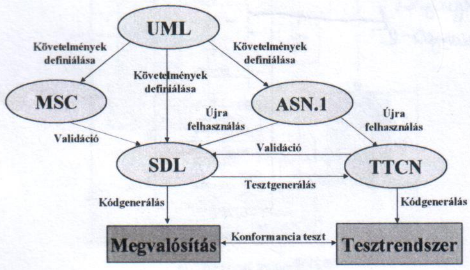
FORMAL TECHNIQUES IN CONFORMANCE ASSESSMENT

- Verification:
  - Check correctness of formal model
- Testing (blackbox):
  - See if Implementation Under Test (IUT) conforms to its specification
  - Experiments programmed into Test Cases
- Validation:
  - Ensure correctness of test cases in ATS



→ abstrakt teszt sorozat
↓
→ implementált

KAPCSOLAT A KÜLÖNBÖZŐ LEÍRÓNYELVEK KÖZÖTT



CONFORMANCE TESTING METHODOLOGY AND FRAMEWORK (CTMF)

A szabványokat eredetileg OSI protokollok tesztelésére fejlesztették ki

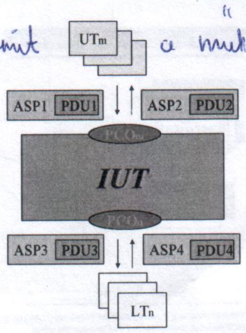
ISO/IEC	ITU-T	Title
9646-1	X.290	General Concepts
9646-2	X.291	Abstract Test Suite Specification
		Multi-protocol Testing
		Multi-party Testing
9646-3	X.292	TTCN Notation
		Concurrent TTCN
		Encoding and Modular TTCN
9646-4	X.293	Test Realization
9646-5	X.294	Requirements on Test Laboratories and Clients for the Conformance Assessment Process
9646-6	X.295	Protocol Profile Test Specification
9646-7	X.296	Implementation Conformance Statements

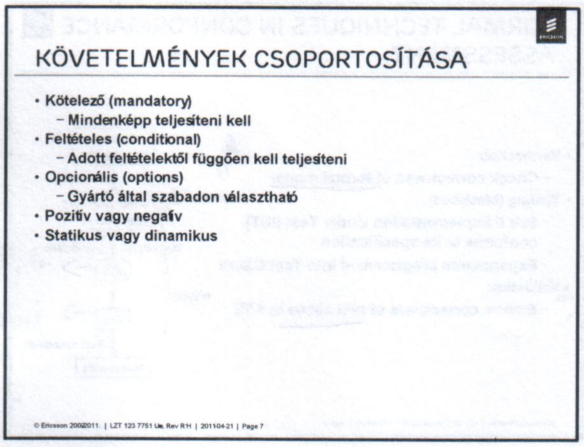
TERMINOLÓGIA

ASP	Abstract Service Primitive
ATM	Abstract Test Method
CP	Coordination Point
IUT	Implementation Under Test
LT	Lower Tester
PCO	Point of Control & Observation
PCTR	Protocol Conformance Test Report
PICS	Protocol Implementation Conformance Statement
PIXIT	Protocol Implementation Extra Information for Testing
PDU	Protocol Data Unit
SCS	System Confomance Statement
SCTR	System Conformance Test Report
SUT	System Under Test
TCP	Test Coordination Procedure
UT	Upper Tester

CONFORMANCE TESTING METHODOLOGY AND FRAMEWORK - CTMF

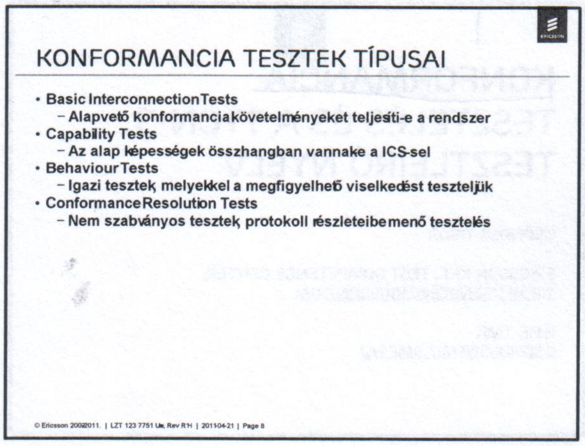
- Az IUT egy fekete doboz, nem tudunk semmit a „működéséről"
- Látható viselkedés PCO-kon keresztül vezérelhető és megfigyelhető
- A PDU-k ASP-be beágyazva kerülnek küldésre és fogadásra a felső (UT) vagy az alsó (LT) teszter által
- A teszt koordináció többrésztvevős tesztelés esetén koordinációs pontokon (CP) keresztül történik





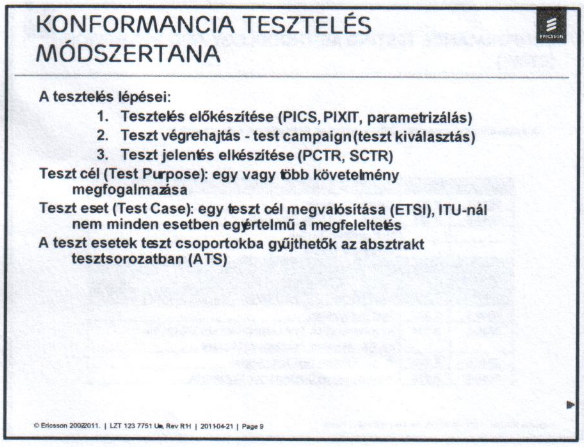
KÖVETELMÉNYEK CSOPORTOSÍTÁSA

- Kötelező (mandatory)
  - Mindenképp teljesíteni kell
- Feltételes (conditional)
  - Adott feltételektől függően kell teljesíteni
- Opcionális (options)
  - Gyártó által szabadon választható
- Pozitív vagy negatív
- Statikus vagy dinamikus



KONFORMANCIA TESZTEK TÍPUSAI

- Basic Interconnection Tests
  - Alapvető konformancia követelményeket teljesíti-e a rendszer
- Capability Tests
  - Az alap képességek összhangban vannak-e a ICS-sel
- Behaviour Tests
  - Igazi tesztek, melyekkel a megfigyelhető viselkedést teszteljük
- Conformance Resolution Tests
  - Nem szabványos tesztek, protokoll részleteibe menő tesztelés



KONFORMANCIA TESZTELÉS MÓDSZERTANA

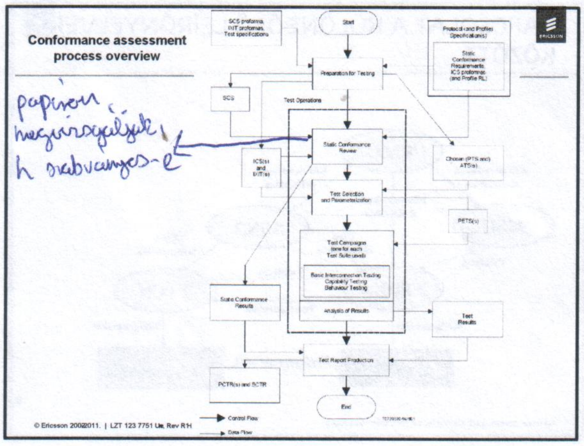
A tesztelés lépései:

1. Tesztelés előkészítése (PICS, PIXIT, parametrizálás)
2. Teszt végrehajtás - test campaign (teszt kiválasztás)
3. Teszt jelentés elkészítése (PCTR, SCTR)

Teszt cél (Test Purpose): egy vagy több követelmény megfogalmazása

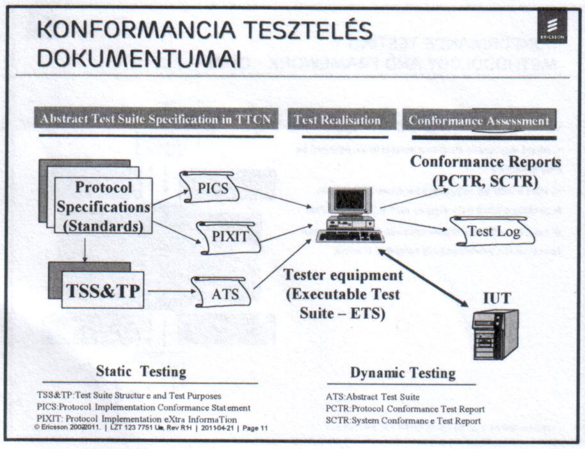
Teszt eset (Test Case): egy teszt cél megvalósítása (ETSI), ITU-nál nem minden esetben egyértelmű a megfeleltetés

A teszt esetek teszt csoportokba gyűjthetők az absztrakt tesztsorozatban (ATS)



Conformance assessment process overview

papíron megvizsgáljuk, h szabványos-e



KONFORMANCIA TESZTELÉS DOKUMENTUMAI

TSS&TP: Test Suite Structure and Test Purposes
PICS: Protocol Implementation Conformance Statement
PIXIT: Protocol Implementation eXtra InformaTion

ATS: Abstract Test Suite
PCTR: Protocol Conformance Test Report
SCTR: System Conformance Test Report

tesztkörnyezet emulálása

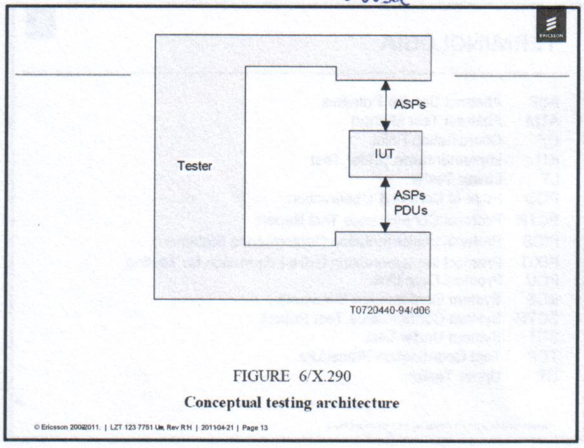
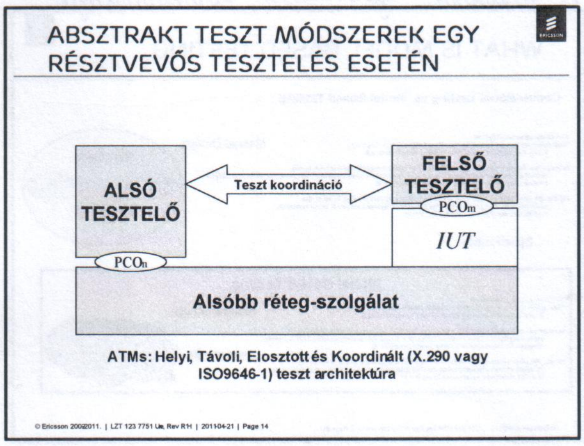


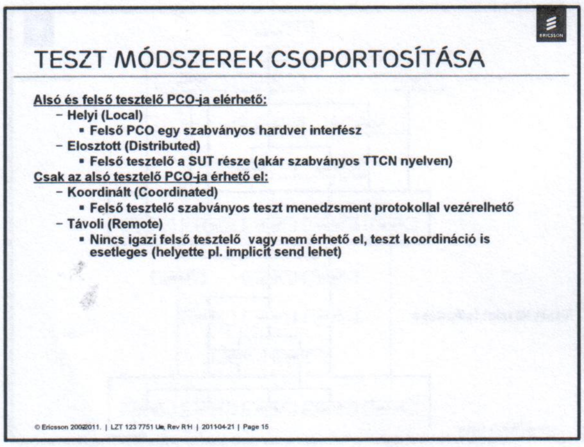
FIGURE 6/X.290

Conceptual testing architecture



ABSZTRAKT TESZT MÓDSZEREK EGY RÉSZTVEVŐS TESZTELÉS ESETÉN

ATMs: Helyi, Távoli, Elosztott és Koordinált (X.290 vagy ISO9646-1) teszt architektúra

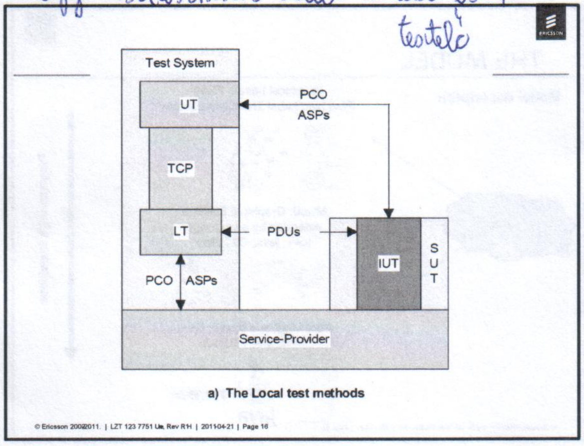


TESZT MÓDSZEREK CSOPORTOSÍTÁSA

Alsó és felső tesztelő PCO-ja elérhető:
- Helyi (Local)
  - Felső PCO egy szabványos hardver interfész
- Elosztott (Distributed)
  - Felső tesztelő a SUT része (akár szabványos TTCN nyelven)

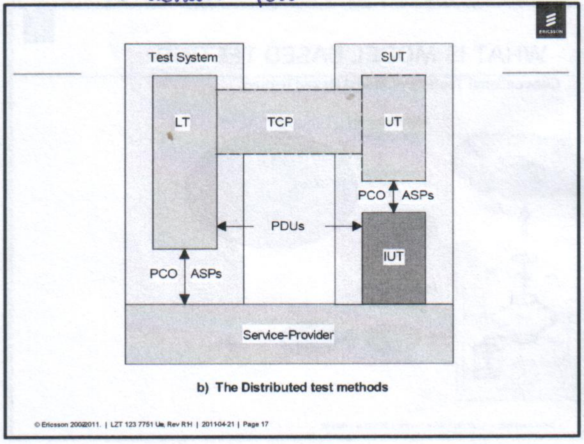
Csak az alsó tesztelő PCO-ja érhető el:
- Koordinált (Coordinated)
  - Felső tesztelő szabványos teszt menedzsment protokollal vezérelhető
- Távoli (Remote)
  - Nincs igazi felső tesztelő vagy nem érhető el, teszt koordináció is esetleges (helyette pl. implicit send lehet)

egy tesztrendszeren belül az alsó és felső tesztelő



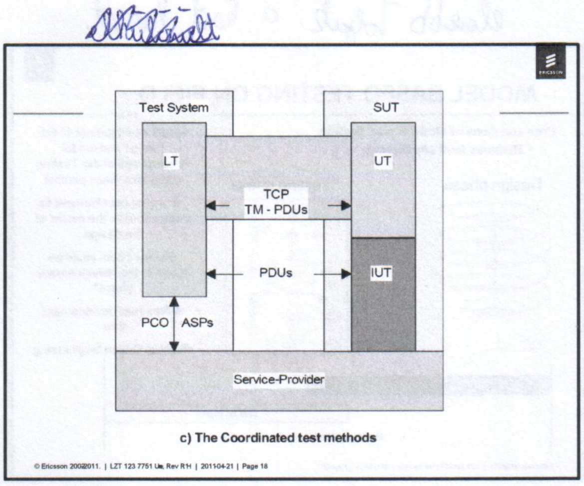
a) The Local test methods

elkülönített felső és alsó

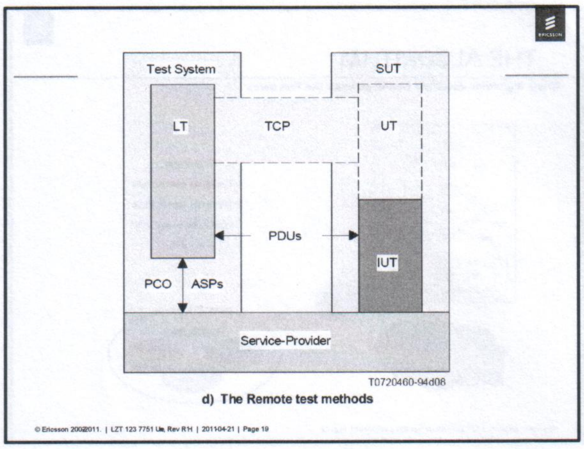


b) The Distributed test methods

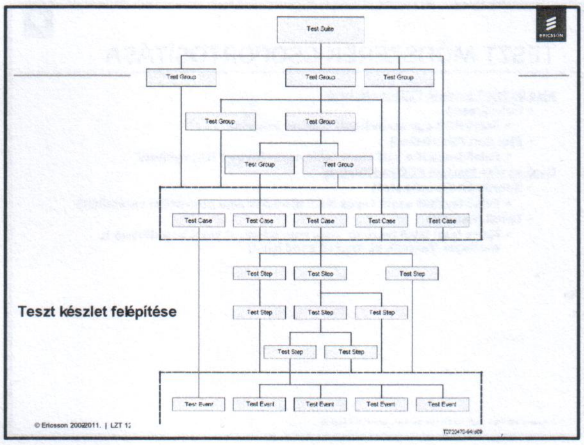
elkülönített



c) The Coordinated test methods

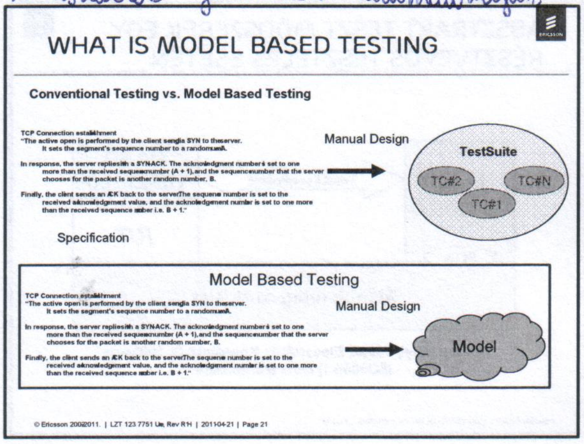


d) The Remote test methods



Teszt készlet felépítése

A tesztesetek generálását automatizáljuk



WHAT IS MODEL BASED TESTING

Conventional Testing vs. Model Based Testing

TCP Connection establishment
"The active open is performed by the client sending a SYN to the server. It sets the segment's sequence number to a random value A.

In response, the server replies with a SYN-ACK. The acknowledgment number is set to one more than the received sequence number (A + 1), and the sequence number that the server chooses for the packet is another random number, B.

Finally, the client sends an ACK back to the server. The sequence number is set to the received acknowledgement value, and the acknowledgement number is set to one more than the received sequence number i.e. B + 1."

Specification

Manual Design

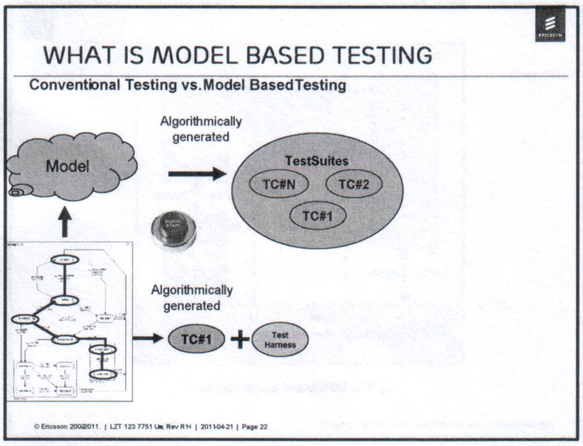
Model Based Testing

TCP Connection establishment
"The active open is performed by the client sending a SYN to the server. It sets the segment's sequence number to a random value A.

In response, the server replies with a SYN-ACK. The acknowledgment number is set to one more than the received sequence number (A + 1), and the sequence number that the server chooses for the packet is another random number, B.

Finally, the client sends an ACK back to the server. The sequence number is set to the received acknowledgement value, and the acknowledgement number is set to one more than the received sequence number i.e. B + 1."

Manual Design

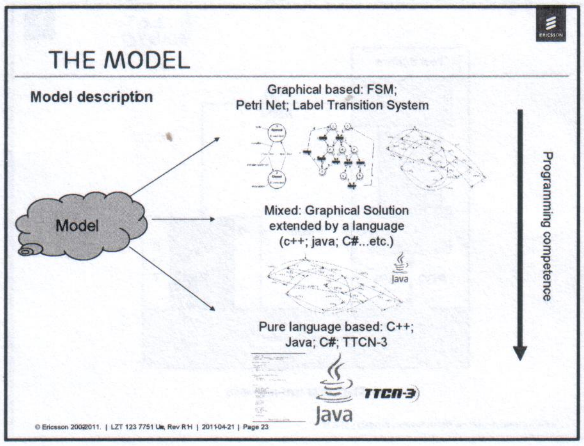


WHAT IS MODEL BASED TESTING

Conventional Testing vs. Model Based Testing

Algorithmically generated

Algorithmically generated



THE MODEL

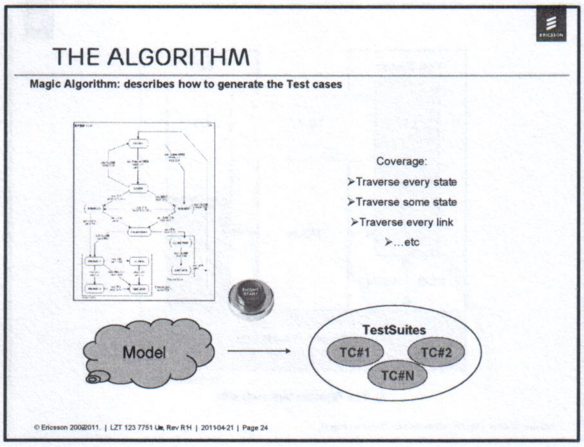
Model description

Graphical based: FSM; Petri Net; Label Transition System

Mixed: Graphical Solution extended by a language (c++; java; C#...etc.)

Pure language based: C++; Java; C#; TTCN-3

Programming competence



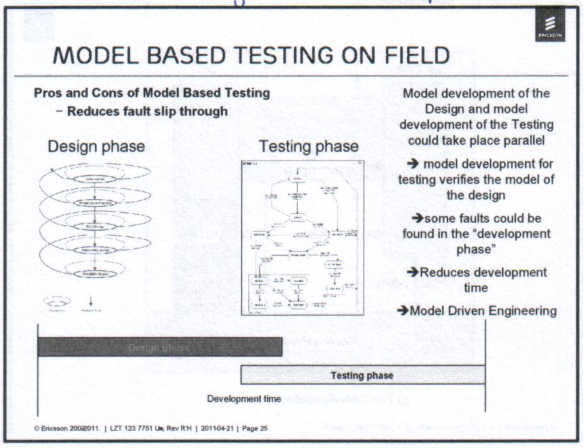
THE ALGORITHM

Magic Algorithm: describes how to generate the Test cases

Coverage:

- Traverse every state
- Traverse some state
- Traverse every link
- ...etc

előbb tolják a test fázist



MODEL BASED TESTING ON FIELD

Pros and Cons of Model Based Testing
- **Reduces fault slip through**

Design phase

Testing phase

Model development of the Design and model development of the Testing could take place parallel

→ model development for testing verifies the model of the design

→some faults could be found in the "development phase"

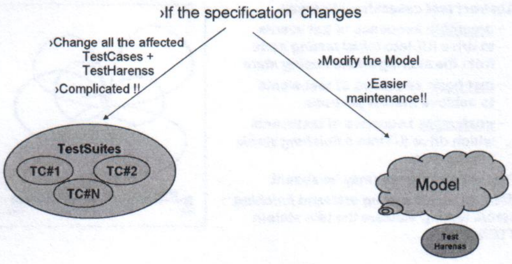
→Reduces development time

→Model Driven Engineering

változás esetén könnyebb a modell

MODEL BASED TESTING WHY?

Pros and Cons of Model Based Testing
- **Reduces fault slip through**
- **Maintenance**


›If the specification changes

›Change all the affected TestCases + TestHarenss

›Complicated !!

›Modify the Model

›Easier maintenance!!

alapjába bevinni a változást

TTCN-3

PRESENTATION MATERIAL

TEST COMPETENCE CENTER
ERICSSON HUNGARY

http://ttcn.ericsson.se/

CONTENTS

Protocols and Testing 3
Introduction to TTCN-3 15
TTCN-3 module structure 25
Type system 34
Constants, variables, module parameters 66
Program statements and operators 76
Timers 83
Test configuration 89
Functions and testcases 100
Verdicts 113
Configuration operations 118
Data templates 147
Abstract communication operations 189
Behavioral statements 198
Sample testcase implementation 220

I. PROTOCOLS AND TESTING

WHAT IS "PROTOCOL"?
DEFINITIONS
PROTOCOL VERIFICATION, TESTING AND VALIDATION

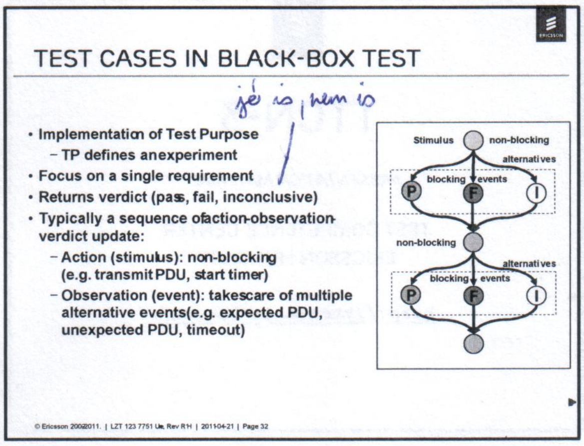
CONTENTS

COMMUNICATIONS PROTOCOL SPECIFICATION

- **Protocol is a set of rules that govern the communication**
  - **syntactical rules:**
    - **define *format (layout)* of messages**
    - **are specified using a Formal Description Technique (e.g. ASN.1, TTCN-3) or proprietary notation (e.g. tabular)**
  - **semantical rules:**
    - **describe *behavior* (how messages are exchanged) and *meaning* of messages**
    - **can be defined formally (e.g. UML, SDL, MSC, LOTOS, Estelle)**
- **Protocol specifications are formal or informal**

TROUBLES IN PROTOCOL TECHNOLOGY

- **Ambiguous protocol specifications**
  - **Formal definition often missing**
  - **Natural language protocol descriptions are ambiguous**
- **"Correct" realization is unlikely**
  - **How to implement?**
  - **How to test conformance to specification?**
- **Formal techniques are employed to detect problems**
  - **Verification, testing and validation**

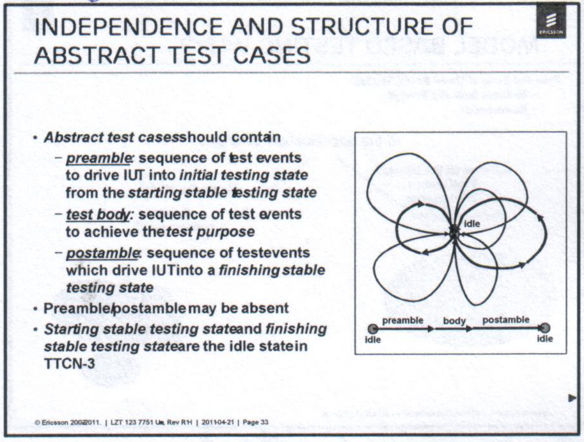


TEST CASES IN BLACK-BOX TEST

jó is, nem is

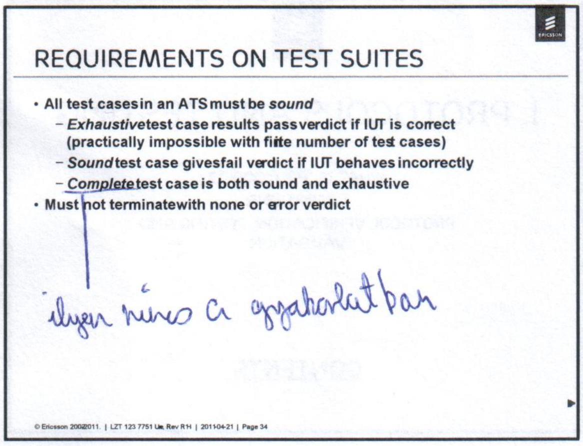
- Implementation of Test Purpose
  - TP defines an experiment
- Focus on a single requirement
- Returns verdict (pass, fail, inconclusive)
- Typically a sequence of action-observation-verdict update:
  - Action (stimulus): non-blocking (e.g. transmit PDU, start timer)
  - Observation (event): takes care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

mindig vissza az alapállapotba



INDEPENDENCE AND STRUCTURE OF ABSTRACT TEST CASES

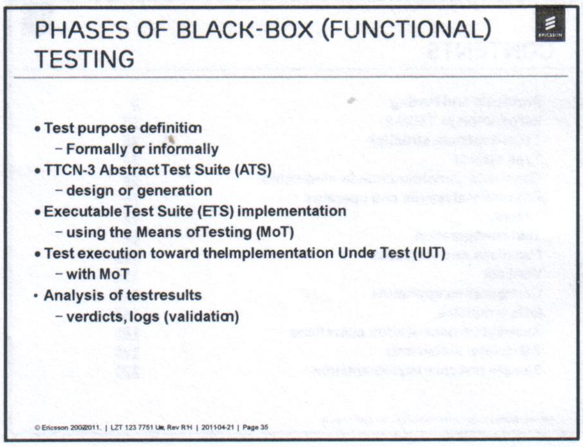
- *Abstract test cases* should contain
  - ***preamble***: sequence of test events to drive IUT into *initial testing state* from the *starting stable testing state*
  - ***test body***: sequence of test events to achieve the *test purpose*
  - ***postamble***: sequence of test events which drive IUT into a *finishing stable testing state*
- Preamble/postamble may be absent
- *Starting stable testing state* and *finishing stable testing state* are the idle state in TTCN-3



REQUIREMENTS ON TEST SUITES

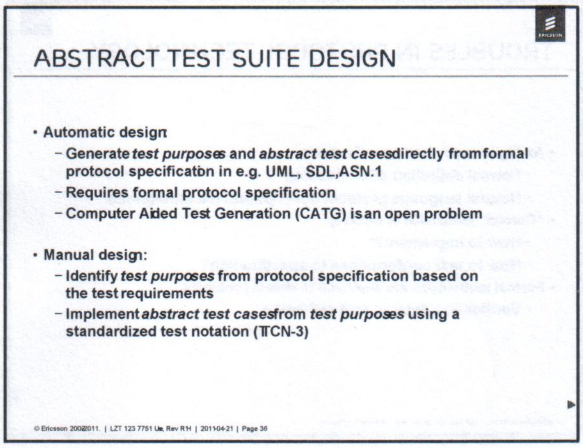
- All test cases in an ATS must be *sound*
  - ***Exhaustive*** test case results pass verdict if IUT is correct (practically impossible with finite number of test cases)
  - ***Sound*** test case gives fail verdict if IUT behaves incorrectly
  - ***Complete*** test case is both sound and exhaustive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

ilyen nincs a gyakorlatban



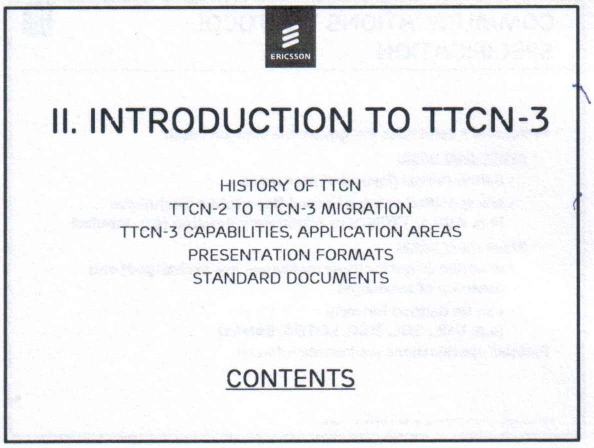
PHASES OF BLACK-BOX (FUNCTIONAL) TESTING

- Test purpose definition
  - Formally or informally
- TTCN-3 Abstract Test Suite (ATS)
  - design or generation
- Executable Test Suite (ETS) implementation
  - using the Means of Testing (MoT)
- Test execution toward the Implementation Under Test (IUT)
  - with MoT
- Analysis of test results
  - verdicts, logs (validation)



ABSTRACT TEST SUITE DESIGN

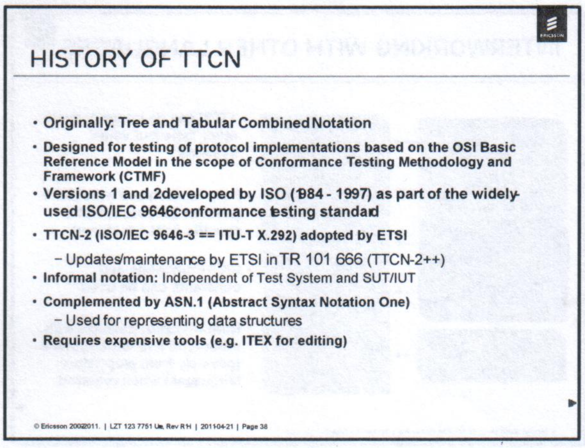
- Automatic design
  - Generate *test purposes* and *abstract test cases* directly from formal protocol specification in e.g. UML, SDL, ASN.1
  - Requires formal protocol specification
  - Computer Aided Test Generation (CATG) is an open problem
- Manual design:
  - Identify *test purposes* from protocol specification based on the test requirements
  - Implement *abstract test cases* from *test purposes* using a standardized test notation (TTCN-3)



II. INTRODUCTION TO TTCN-3

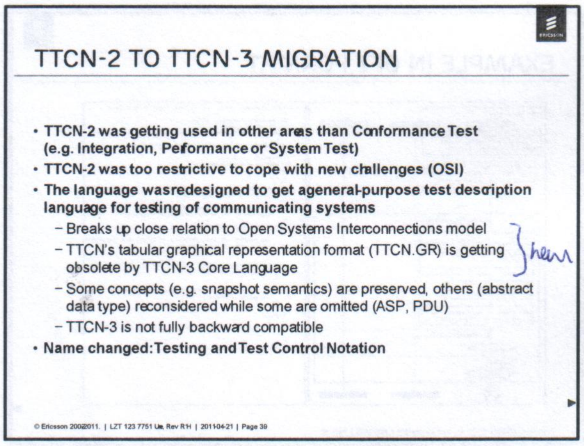
HISTORY OF TTCN
TTCN-2 TO TTCN-3 MIGRATION
TTCN-3 CAPABILITIES, APPLICATION AREAS
PRESENTATION FORMATS
STANDARD DOCUMENTS

CONTENTS



HISTORY OF TTCN

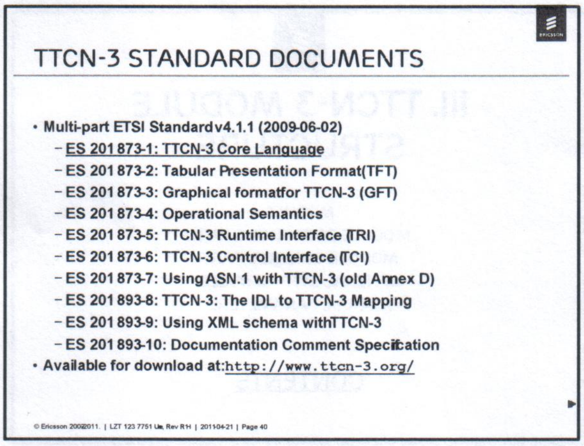
- **Originally: Tree and Tabular Combined Notation**
- **Designed for testing of protocol implementations based on the OSI Basic Reference Model in the scope of Conformance Testing Methodology and Framework (CTMF)**
- **Versions 1 and 2 developed by ISO (1984 - 1997) as part of the widely-used ISO/IEC 9646 conformance testing standard**
- **TTCN-2 (ISO/IEC 9646-3 == ITU-T X.292) adopted by ETSI**
  - Updates/maintenance by ETSI in TR 101 666 (TTCN-2++)
- **Informal notation:** Independent of Test System and SUT/IUT
- **Complemented by ASN.1 (Abstract Syntax Notation One)**
  - Used for representing data structures
- **Requires expensive tools (e.g. ITEX for editing)**



TTCN-2 TO TTCN-3 MIGRATION

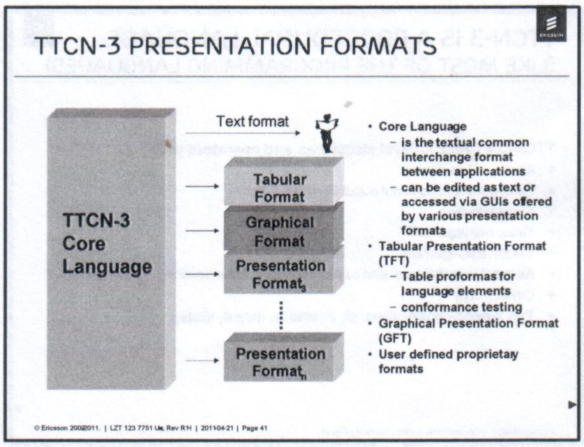
- **TTCN-2 was getting used in other areas than Conformance Test (e.g. Integration, Performance or System Test)**
- **TTCN-2 was too restrictive to cope with new challenges (OSI)**
- **The language was redesigned to get a general-purpose test description language for testing of communicating systems**
  - Breaks up close relation to Open Systems Interconnections model
  - TTCN's tabular graphical representation format (TTCN.GR) is getting obsolete by TTCN-3 Core Language
  - Some concepts (e.g. snapshot semantics) are preserved, others (abstract data type) reconsidered while some are omitted (ASP, PDU)
  - TTCN-3 is not fully backward compatible
- **Name changed: Testing and Test Control Notation**

nem használják



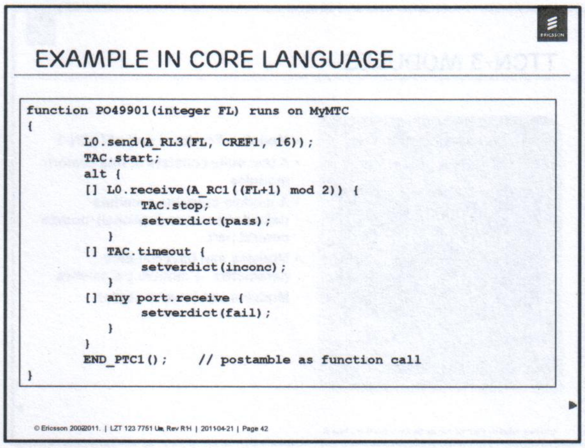
TTCN-3 STANDARD DOCUMENTS

- **Multi-part ETSI Standard v4.1.1 (2009-06-02)**
  - **<u>ES 201 873-1: TTCN-3 Core Language</u>**
  - **ES 201 873-2: Tabular Presentation Format (TFT)**
  - **ES 201 873-3: Graphical format for TTCN-3 (GFT)**
  - **ES 201 873-4: Operational Semantics**
  - **ES 201 873-5: TTCN-3 Runtime Interface (TRI)**
  - **ES 201 873-6: TTCN-3 Control Interface (TCI)**
  - **ES 201 873-7: Using ASN.1 with TTCN-3 (old Annex D)**
  - **ES 201 893-8: TTCN-3: The IDL to TTCN-3 Mapping**
  - **ES 201 893-9: Using XML schema with TTCN-3**
  - **ES 201 893-10: Documentation Comment Specification**
- **Available for download at:** `http://www.ttcn-3.org/`



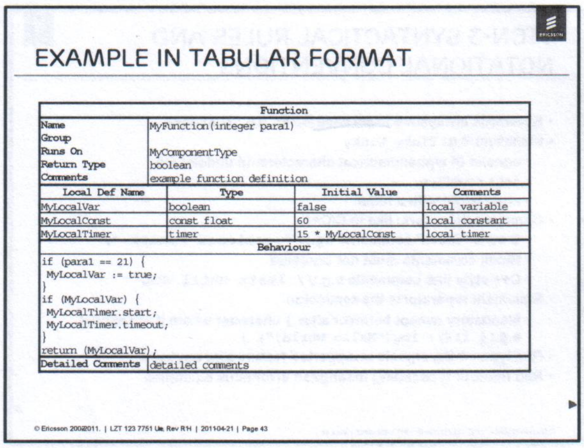
TTCN-3 PRESENTATION FORMATS

- **Core Language**
  - **is the textual common interchange format between applications**
  - **can be edited as text or accessed via GUIs offered by various presentation formats**
- **Tabular Presentation Format (TFT)**
  - **Table proformas for language elements**
  - **conformance testing**
- **Graphical Presentation Format (GFT)**
- **User defined proprietary formats**



EXAMPLE IN CORE LANGUAGE

```
function PO49901(integer FL) runs on MyMTC
{
        L0.send(A_RL3(FL, CREF1, 16));
        TAC.start;
        alt {
        [] L0.receive(A_RC1((FL+1) mod 2)) {
                TAC.stop;
                setverdict(pass);
           }
        [] TAC.timeout {
                setverdict(inconc);
           }
        [] any port.receive {
                setverdict(fail);
           }
        }
        END_PTC1();    // postamble as function call
}
```



EXAMPLE IN TABULAR FORMAT

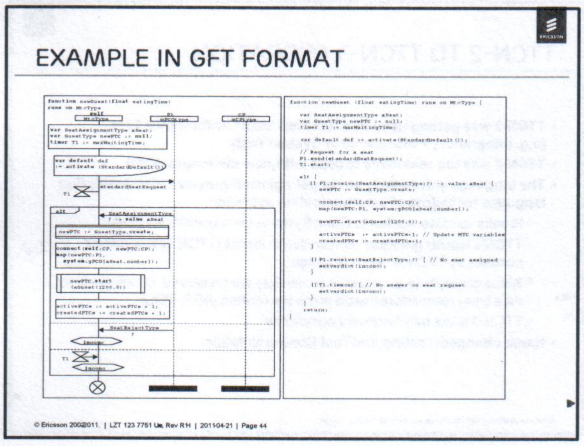
Function	
Name	MyFunction(integer para1)
Group	
Runs On	MyComponentType
Return Type	boolean
Comments	example function definition

Local Def Name	Type	Initial Value	Comments
MyLocalVar	boolean	false	local variable
MyLocalConst	const float	60	local constant
MyLocalTimer	timer	15 * MyLocalConst	local timer

Behaviour

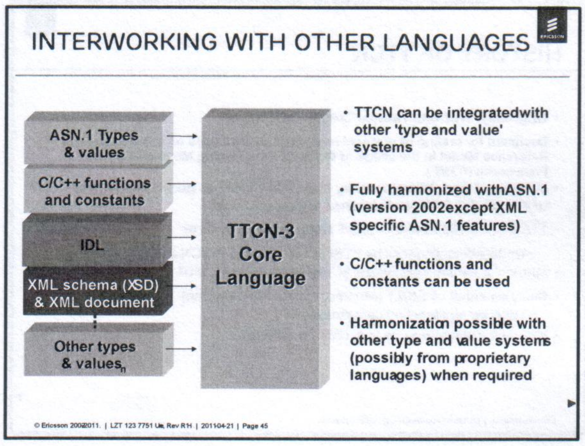
```
if (para1 == 21) {
 MyLocalVar := true;
}
if (MyLocalVar) {
 MyLocalTimer.start;
 MyLocalTimer.timeout;
}
return (MyLocalVar);
```

Detailed Comments	detailed comments



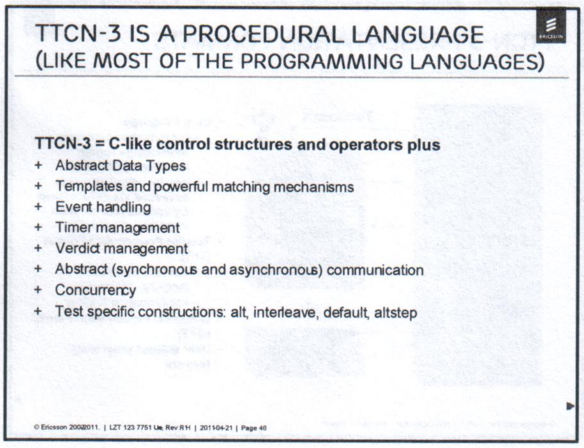
EXAMPLE IN GFT FORMAT

```
function newGuest (float eatingTime) runs on MtcType {

   var SeatAssignmentType aSeat;
   var GuestType newPTC := null;
   timer T1 := maxWaitingTime;

   var default def := activate(StandardDefault());

   // Request for a seat
   P1.send(standardSeatRequest);
   T1.start;

   alt {
     [] P1.receive(SeatAssignmentType:?) -> value aSeat {
         newPTC := GuestType.create;

         connect(self:CP, newPTC:CP);
         map(newPTC:P1, system:gPCO[aSeat.number]);

         newPTC.start(aGuest(1200,0));

         activePTCs  := activePTCs+1; // Update MTC variables
         createdPTCs := createdPTCs+1;
     }

     [] P1.receive(SeatRejectType:?) { // No seat assigned
         setverdict(inconc);
     }

     [] T1.timeout { // No answer on seat request
         setverdict(inconc);
     }
    }
    return;
}
```



INTERWORKING WITH OTHER LANGUAGES

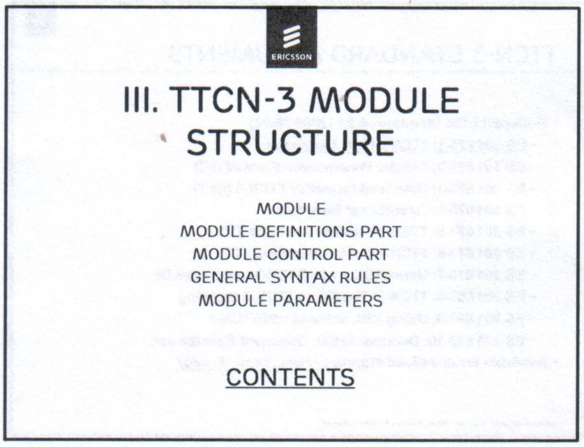
- TTCN can be integrated with other 'type and value' systems
- Fully harmonized with ASN.1 (version 2002 except XML specific ASN.1 features)
- C/C++ functions and constants can be used
- Harmonization possible with other type and value systems (possibly from proprietary languages) when required



TTCN-3 IS A PROCEDURAL LANGUAGE (LIKE MOST OF THE PROGRAMMING LANGUAGES)

TTCN-3 = C-like control structures and operators plus

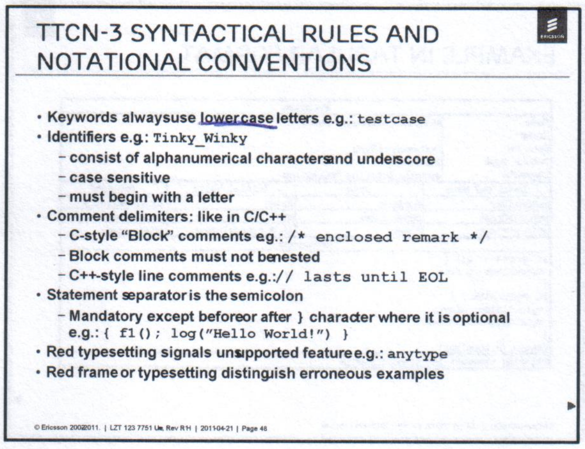
- \+ Abstract Data Types
- \+ Templates and powerful matching mechanisms
- \+ Event handling
- \+ Timer management
- \+ Verdict management
- \+ Abstract (synchronous and asynchronous) communication
- \+ Concurrency
- \+ Test specific constructions: alt, interleave, default, altstep



III. TTCN-3 MODULE STRUCTURE

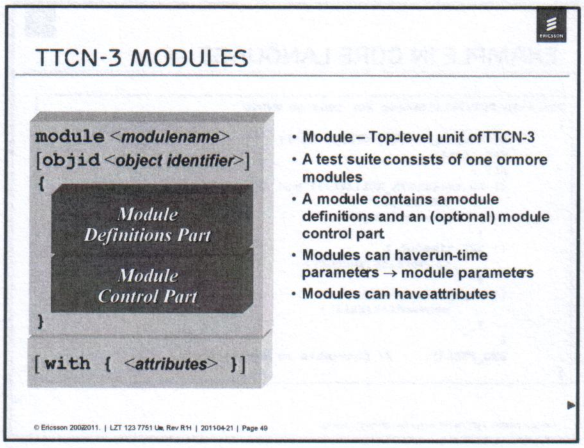
MODULE
MODULE DEFINITIONS PART
MODULE CONTROL PART
GENERAL SYNTAX RULES
MODULE PARAMETERS

CONTENTS



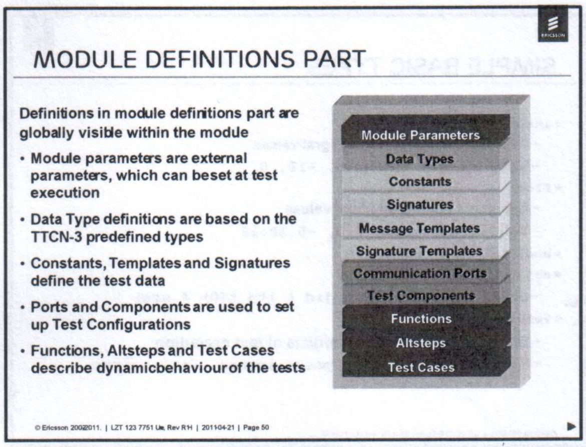
TTCN-3 SYNTACTICAL RULES AND NOTATIONAL CONVENTIONS

- Keywords always use lowercase letters e.g.: `testcase`
- Identifiers e.g.: `Tinky_Winky`
  - consist of alphanumerical characters and underscore
  - case sensitive
  - must begin with a letter
- Comment delimiters: like in C/C++
  - C-style "Block" comments e.g.: `/* enclosed remark */`
  - Block comments must not be nested
  - C++-style line comments e.g.: `// lasts until EOL`
- Statement separator is the semicolon
  - Mandatory except before or after } character where it is optional e.g.: `{ f1(); log("Hello World!") }`
- Red typesetting signals unsupported feature e.g.: `anytype`
- Red frame or typesetting distinguish erroneous examples



TTCN-3 MODULES

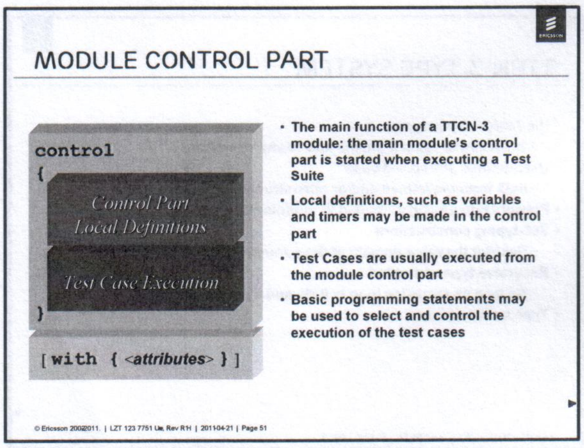
- Module – Top-level unit of TTCN-3
- A test suite consists of one or more modules
- A module contains a module definitions and an (optional) module control part
- Modules can have run-time parameters → module parameters
- Modules can have attributes



MODULE DEFINITIONS PART

Definitions in module definitions part are globally visible within the module

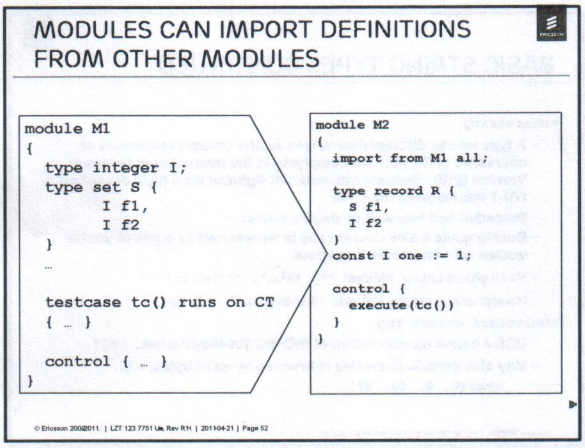
- **Module parameters are external parameters, which can beset at test execution**
- **Data Type definitions are based on the TTCN-3 predefined types**
- **Constants, Templates and Signatures define the test data**
- **Ports and Components are used to set up Test Configurations**
- **Functions, Altsteps and Test Cases describe dynamicbehaviour of the tests**



MODULE CONTROL PART

```
control
{
  Control Part
  Local Definitions

  Test Case Execution
}
[ with { <attributes> } ]
```

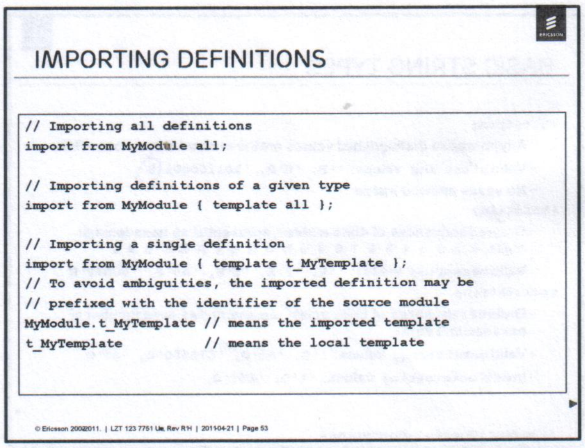
- **The main function of a TTCN-3 module: the main module's control part is started when executing a Test Suite**
- **Local definitions, such as variables and timers may be made in the control part**
- **Test Cases are usually executed from the module control part**
- **Basic programming statements may be used to select and control the execution of the test cases**



MODULES CAN IMPORT DEFINITIONS FROM OTHER MODULES

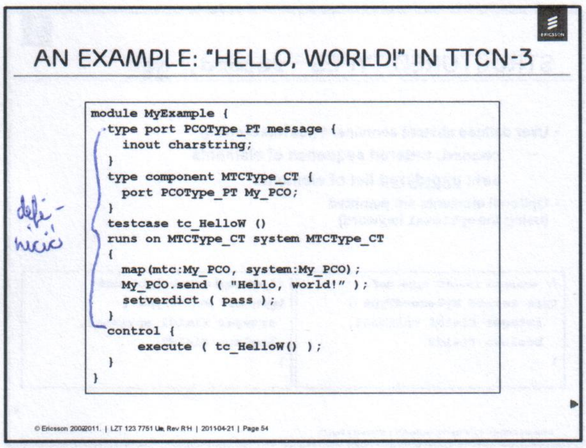
```
module M1
{
  type integer I;
  type set S {
       I f1,
       I f2
  }
  …

  testcase tc() runs on CT
  { … }

  control { … }
}
```

```
module M2
{
  import from M1 all;

  type record R {
    S f1,
    I f2
  }
  const I one := 1;

  control {
    execute(tc())
  }
}
```



IMPORTING DEFINITIONS

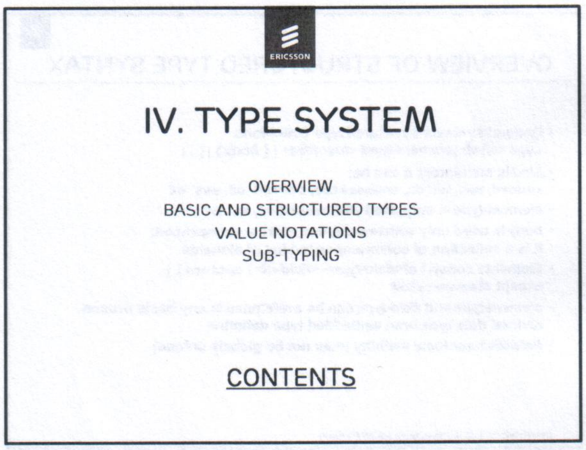
```
// Importing all definitions
import from MyModule all;

// Importing definitions of a given type
import from MyModule { template all };

// Importing a single definition
import from MyModule { template t_MyTemplate };
// To avoid ambiguities, the imported definition may be
// prefixed with the identifier of the source module
MyModule.t_MyTemplate // means the imported template
t_MyTemplate          // means the local template
```



AN EXAMPLE: "HELLO, WORLD!" IN TTCN-3

definició

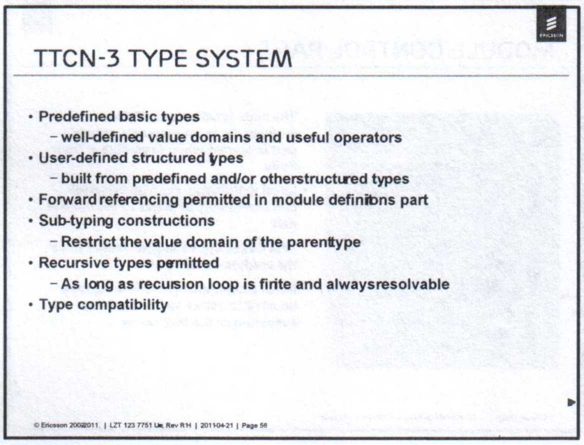
```
module MyExample {
  type port PCOType_PT message {
    inout charstring;
  }
  type component MTCType_CT {
    port PCOType_PT My_PCO;
  }
  testcase tc_HelloW ()
  runs on MTCType_CT system MTCType_CT
  {
    map(mtc:My_PCO, system:My_PCO);
    My_PCO.send ( "Hello, world!" );
    setverdict ( pass );
  }
  control {
      execute ( tc_HelloW() );
  }
}
```



IV. TYPE SYSTEM

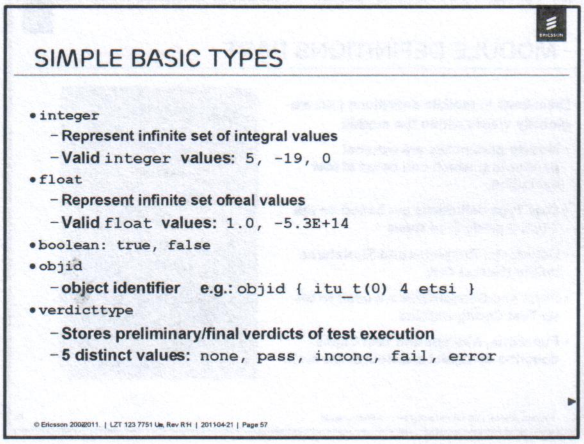
OVERVIEW
BASIC AND STRUCTURED TYPES
VALUE NOTATIONS
SUB-TYPING

CONTENTS



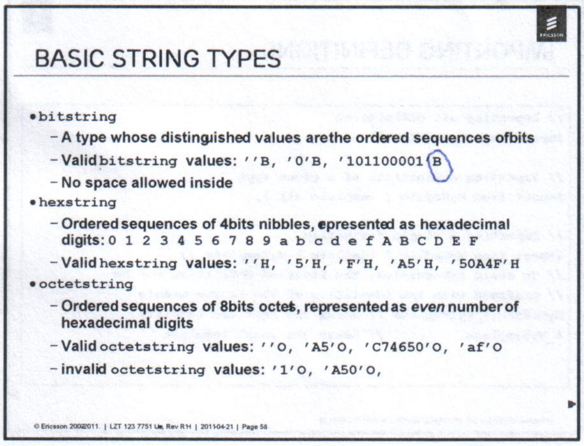
TTCN-3 TYPE SYSTEM

- Predefined basic types
  - well-defined value domains and useful operators
- User-defined structured types
  - built from predefined and/or other structured types
- Forward referencing permitted in module definitions part
- Sub-typing constructions
  - Restrict the value domain of the parent type
- Recursive types permitted
  - As long as recursion loop is finite and always resolvable
- Type compatibility



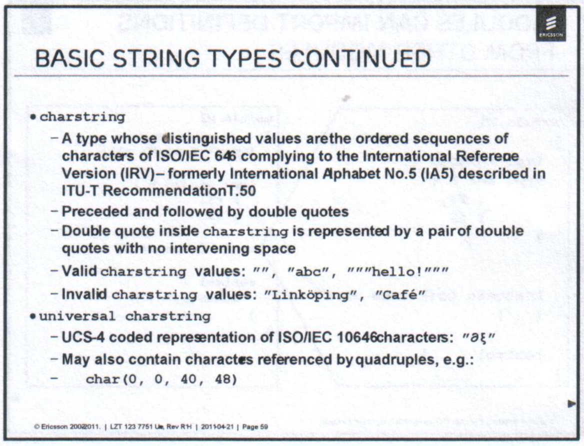
SIMPLE BASIC TYPES

- `integer`
  - Represent infinite set of integral values
  - Valid `integer` values: `5, -19, 0`
- `float`
  - Represent infinite set of real values
  - Valid `float` values: `1.0, -5.3E+14`
- `boolean: true, false`
- `objid`
  - object identifier e.g.: `objid { itu_t(0) 4 etsi }`
- `verdicttype`
  - Stores preliminary/final verdicts of test execution
  - 5 distinct values: `none, pass, inconc, fail, error`



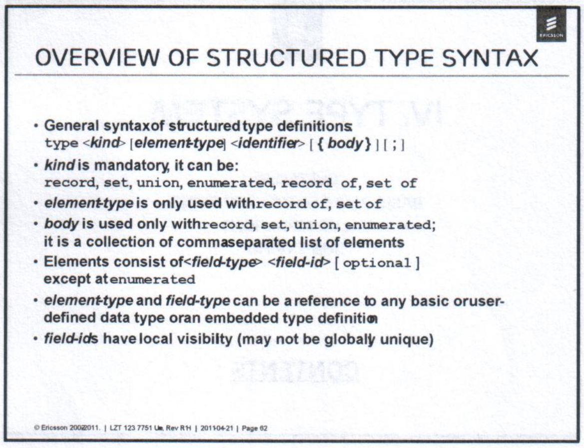
BASIC STRING TYPES

- `bitstring`
  - A type whose distinguished values are the ordered sequences of bits
  - Valid `bitstring` values: `''B, '0'B, '101100001'B`
  - No space allowed inside
- `hexstring`
  - Ordered sequences of 4bits nibbles, represented as hexadecimal digits: `0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F`
  - Valid `hexstring` values: `''H, '5'H, 'F'H, 'A5'H, '50A4F'H`
- `octetstring`
  - Ordered sequences of 8bits octets, represented as *even* number of hexadecimal digits
  - Valid `octetstring` values: `''O, 'A5'O, 'C74650'O, 'af'O`
  - invalid `octetstring` values: `'1'O, 'A50'O,`



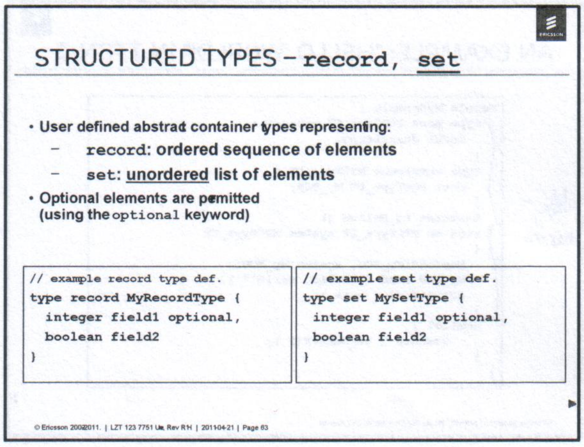
BASIC STRING TYPES CONTINUED

- `charstring`
  - A type whose distinguished values are the ordered sequences of characters of ISO/IEC 646 complying to the International Reference Version (IRV) – formerly International Alphabet No.5 (IA5) described in ITU-T Recommendation T.50
  - Preceded and followed by double quotes
  - Double quote inside `charstring` is represented by a pair of double quotes with no intervening space
  - Valid `charstring` values: `"", "abc", """hello!"""`
  - Invalid `charstring` values: `"Linköping", "Café"`
- `universal charstring`
  - UCS-4 coded representation of ISO/IEC 10646 characters: `"əξ"`
  - May also contain characters referenced by quadruples, e.g.:
  - `char(0, 0, 40, 48)`



OVERVIEW OF STRUCTURED TYPE SYNTAX

- General syntax of structured type definitions:
  `type` <*kind*> [*element-type*] <*identifier*> [{ *body* }] [;]
- *kind* is mandatory, it can be:
  `record`, `set`, `union`, `enumerated`, `record of`, `set of`
- *element-type* is only used with `record of`, `set of`
- *body* is used only with `record`, `set`, `union`, `enumerated`;
  it is a collection of comma separated list of elements
- Elements consist of <*field-type*> <*field-id*> [`optional`]
  except at `enumerated`
- *element-type* and *field-type* can be a reference to any basic or user-defined data type or an embedded type definition
- *field-id*s have local visibilty (may not be globally unique)

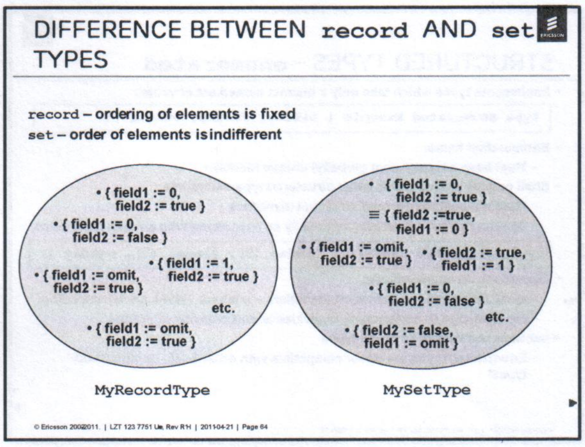


STRUCTURED TYPES – `record`, `set`

- User defined abstract container types representing:
  - `record`: ordered sequence of elements
  - `set`: unordered list of elements
- Optional elements are permitted
  (using the `optional` keyword)

```
// example record type def.
type record MyRecordType {
  integer field1 optional,
  boolean field2
}
```

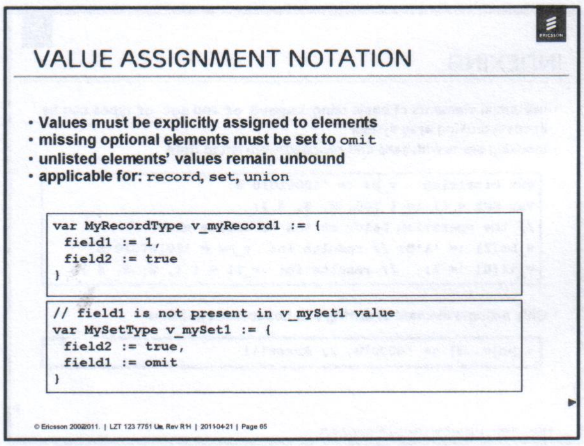
```
// example set type def.
type set MySetType {
 integer field1 optional,
 boolean field2
}
```



DIFFERENCE BETWEEN `record` AND `set` TYPES

`record` – ordering of elements is fixed

`set` – order of elements is indifferent

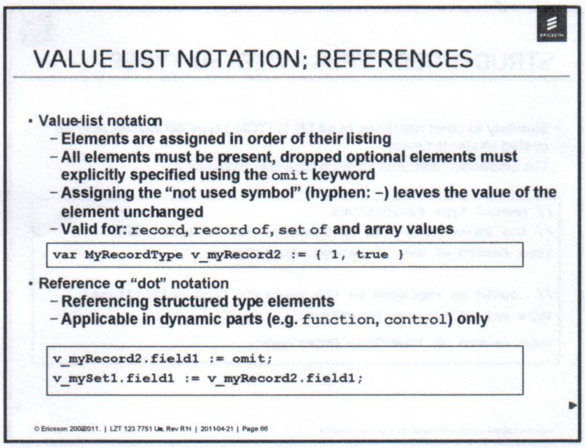


VALUE ASSIGNMENT NOTATION

- Values must be explicitly assigned to elements
- missing optional elements must be set to `omit`
- unlisted elements' values remain unbound
- applicable for: `record`, `set`, `union`

```
var MyRecordType v_myRecord1 := {
 field1 := 1,
 field2 := true
}
```

```
// field1 is not present in v_mySet1 value
var MySetType v_mySet1 := {
 field2 := true,
 field1 := omit
}
```



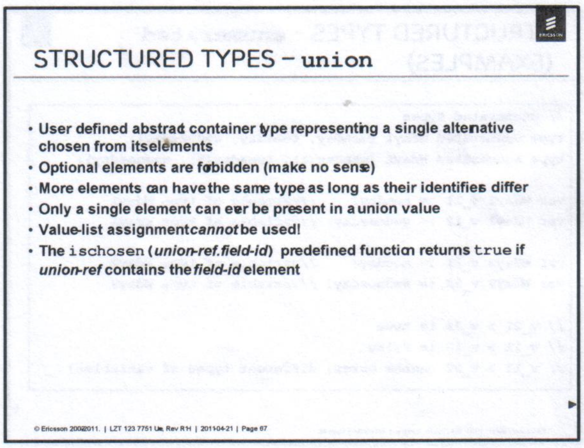
VALUE LIST NOTATION; REFERENCES

- Value-list notation
  - Elements are assigned in order of their listing
  - All elements must be present, dropped optional elements must explicitly specified using the `omit` keyword
  - Assigning the "not used symbol" (hyphen: –) leaves the value of the element unchanged
  - Valid for: `record`, `record of`, `set of` and array values

```
var MyRecordType v_myRecord2 := { 1, true }
```

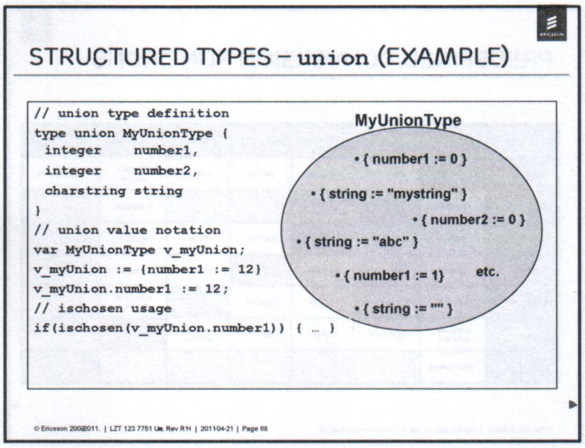
- Reference or "dot" notation
  - Referencing structured type elements
  - Applicable in dynamic parts (e.g. `function`, `control`) only

```
v_myRecord2.field1 := omit;
v_mySet1.field1 := v_myRecord2.field1;
```



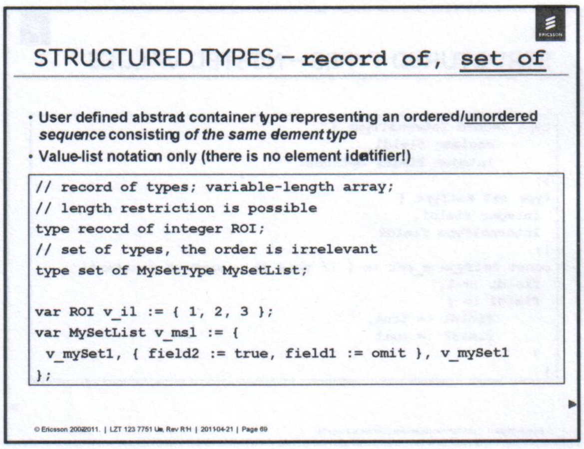
STRUCTURED TYPES – `union`

- User defined abstract container type representing a single alternative chosen from its elements
- Optional elements are forbidden (make no sense)
- More elements can have the same type as long as their identifiers differ
- Only a single element can ever be present in a union value
- Value-list assignment *cannot* be used!
- The `ischosen` (*union-ref.field-id*) predefined function returns `true` if *union-ref* contains the *field-id* element



STRUCTURED TYPES – `union` (EXAMPLE)

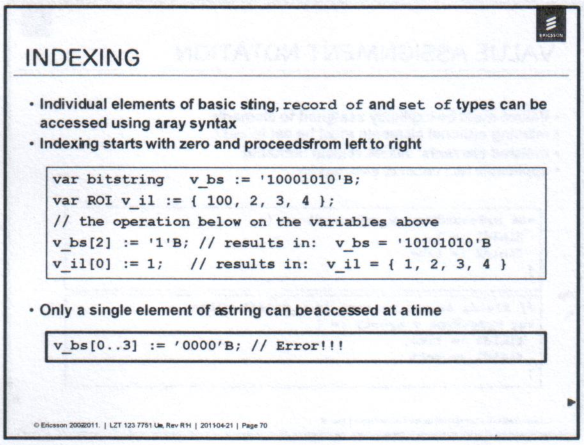
```
// union type definition
type union MyUnionType {
 integer    number1,
 integer    number2,
 charstring string
}
// union value notation
var MyUnionType v_myUnion;
v_myUnion := {number1 := 12}
v_myUnion.number1 := 12;
// ischosen usage
if(ischosen(v_myUnion.number1)) { … }
```



STRUCTURED TYPES – `record of`, `set of`

- User defined abstract container type representing an ordered/<u>unordered</u> *sequence* consisting *of the same element type*
- Value-list notation only (there is no element identifier!)

```
// record of types; variable-length array;
// length restriction is possible
type record of integer ROI;
// set of types, the order is irrelevant
type set of MySetType MySetList;

var ROI v_il := { 1, 2, 3 };
var MySetList v_msl := {
 v_mySet1, { field2 := true, field1 := omit }, v_mySet1
};
```



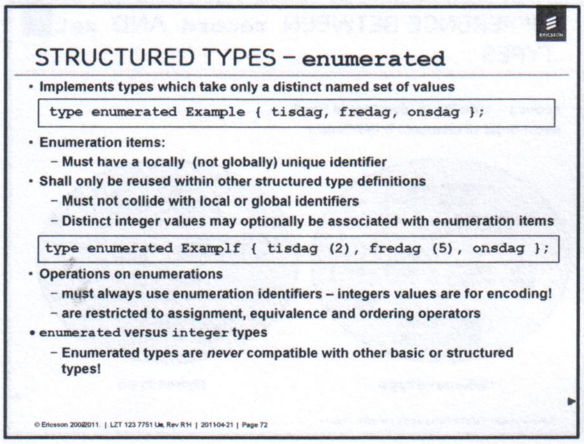
INDEXING

- Individual elements of basic string, `record of` and `set of` types can be accessed using array syntax
- Indexing starts with zero and proceeds from left to right

```
var bitstring   v_bs := '10001010'B;
var ROI v_il := { 100, 2, 3, 4 };
// the operation below on the variables above
v_bs[2] := '1'B; // results in:  v_bs = '10101010'B
v_il[0] := 1;   // results in:  v_il = { 1, 2, 3, 4 }
```

- Only a single element of a string can be accessed at a time

```
v_bs[0..3] := '0000'B; // Error!!!
```



STRUCTURED TYPES – `enumerated`

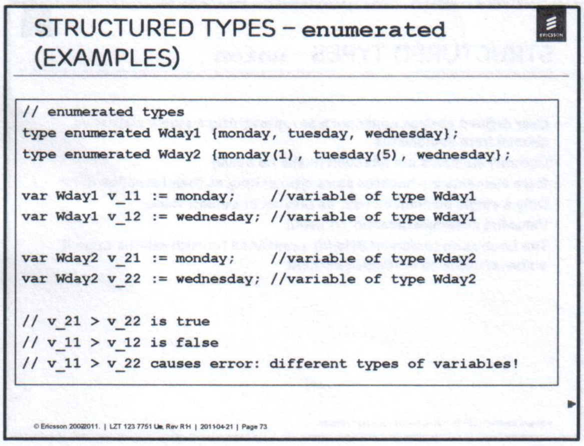
- Implements types which take only a distinct named set of values

```
type enumerated Example { tisdag, fredag, onsdag };
```

- Enumeration items:
  - Must have a locally (not globally) unique identifier
- Shall only be reused within other structured type definitions
  - Must not collide with local or global identifiers
  - Distinct integer values may optionally be associated with enumeration items

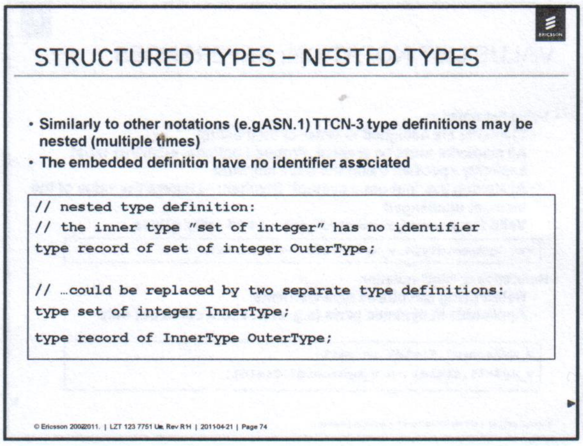
```
type enumerated Examplf { tisdag (2), fredag (5), onsdag };
```

- Operations on enumerations
  - must always use enumeration identifiers – integers values are for encoding!
  - are restricted to assignment, equivalence and ordering operators
- `enumerated` versus `integer` types
  - Enumerated types are *never* compatible with other basic or structured types!



STRUCTURED TYPES – `enumerated` (EXAMPLES)

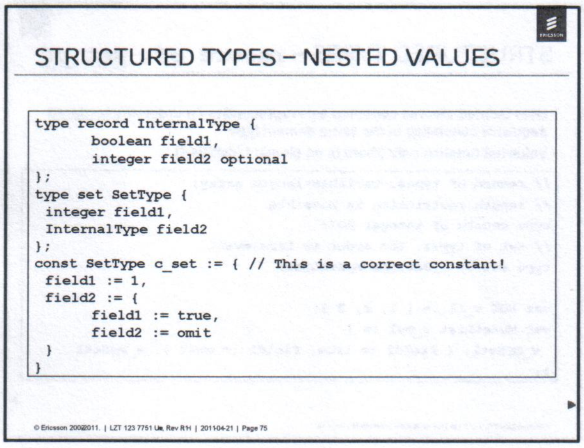
```
// enumerated types
type enumerated Wday1 {monday, tuesday, wednesday};
type enumerated Wday2 {monday(1), tuesday(5), wednesday};

var Wday1 v_11 := monday;    //variable of type Wday1
var Wday1 v_12 := wednesday; //variable of type Wday1

var Wday2 v_21 := monday;    //variable of type Wday2
var Wday2 v_22 := wednesday; //variable of type Wday2

// v_21 > v_22 is true
// v_11 > v_12 is false
// v_11 > v_22 causes error: different types of variables!
```



STRUCTURED TYPES – NESTED TYPES

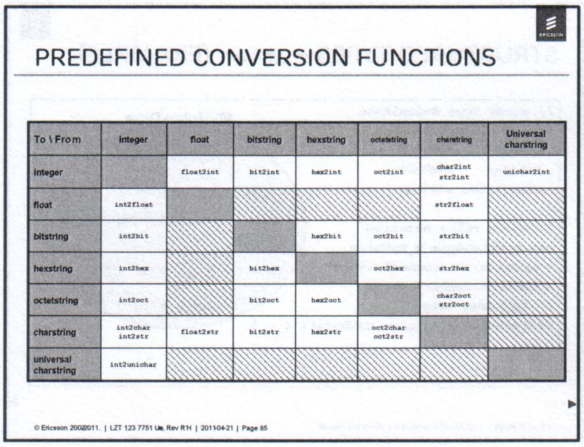
- Similarly to other notations (e.g. ASN.1) TTCN-3 type definitions may be nested (multiple times)
- The embedded definition have no identifier associated

```
// nested type definition:
// the inner type "set of integer" has no identifier
type record of set of integer OuterType;

// …could be replaced by two separate type definitions:
type set of integer InnerType;
type record of InnerType OuterType;
```



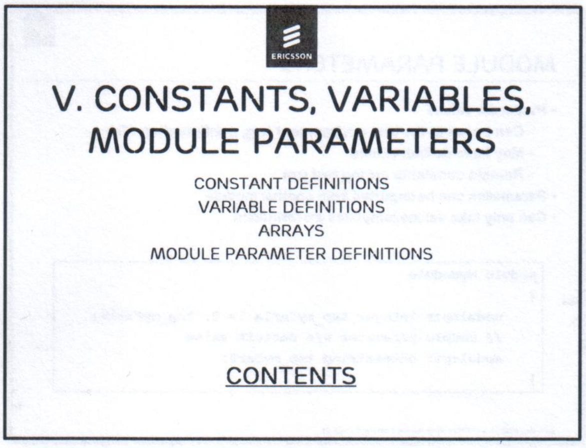
STRUCTURED TYPES – NESTED VALUES

```
type record InternalType {
       boolean field1,
       integer field2 optional
};
type set SetType {
 integer field1,
 InternalType field2
};
const SetType c_set := { // This is a correct constant!
 field1 := 1,
 field2 := {
       field1 := true,
       field2 := omit
 }
}
```



PREDEFINED CONVERSION FUNCTIONS

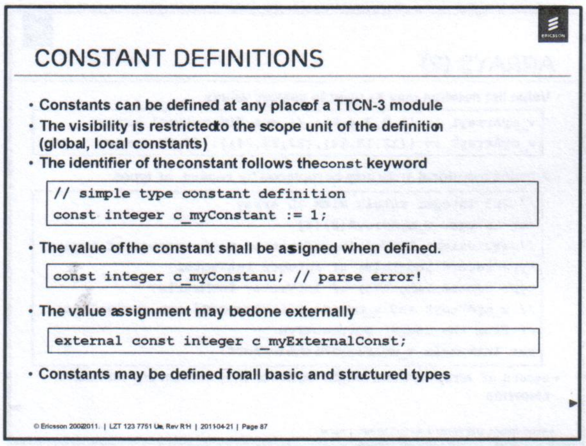
To \ From	integer	float	bitstring	hexstring	octetstring	charstring	Universal charstring
integer		float2int	bit2int	hex2int	oct2int	char2int str2int	unichar2int
float	int2float		▨	▨	▨	str2float	▨
bitstring	int2bit	▨		hex2bit	oct2bit	str2bit	▨
hexstring	int2hex	▨	bit2hex		oct2hex	str2hex	▨
octetstring	int2oct	▨	bit2oct	hex2oct		char2oct str2oct	▨
charstring	int2char int2str	float2str	bit2str	hex2str	oct2char oct2str		▨
universal charstring	int2unichar	▨	▨	▨	▨	▨	



V. CONSTANTS, VARIABLES, MODULE PARAMETERS

CONSTANT DEFINITIONS
VARIABLE DEFINITIONS
ARRAYS
MODULE PARAMETER DEFINITIONS

CONTENTS



CONSTANT DEFINITIONS

- Constants can be defined at any place of a TTCN-3 module
- The visibility is restricted to the scope unit of the definition (global, local constants)
- The identifier of the constant follows the `const` keyword

```
// simple type constant definition
const integer c_myConstant := 1;
```

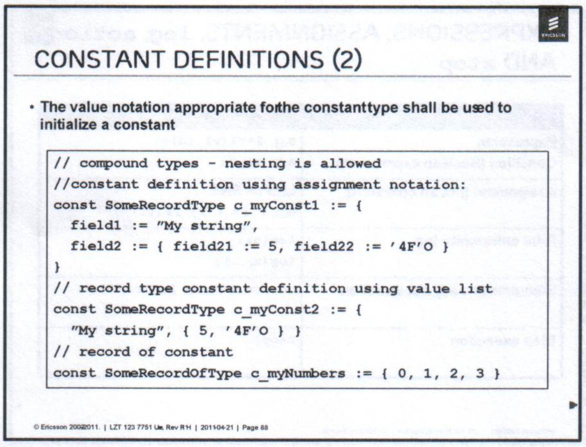
- The value of the constant shall be assigned when defined.

```
const integer c_myConstanu; // parse error!
```

- The value assignment may be done externally

```
external const integer c_myExternalConst;
```

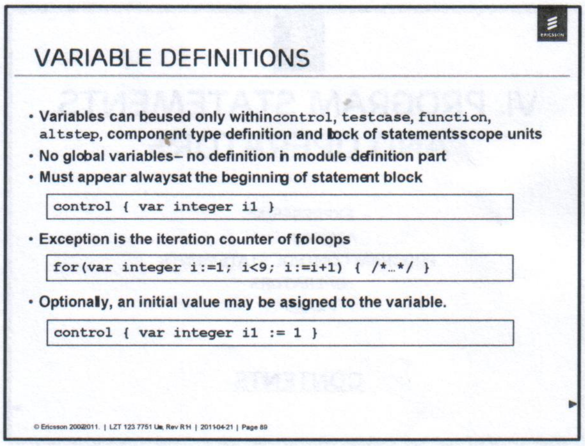
- Constants may be defined for all basic and structured types



CONSTANT DEFINITIONS (2)

- The value notation appropriate for the constant type shall be used to initialize a constant

```
// compound types - nesting is allowed
//constant definition using assignment notation:
const SomeRecordType c_myConst1 := {
  field1 := "My string",
  field2 := { field21 := 5, field22 := '4F'O }
}
// record type constant definition using value list
const SomeRecordType c_myConst2 := {
  "My string", { 5, '4F'O } }
// record of constant
const SomeRecordOfType c_myNumbers := { 0, 1, 2, 3 }
```



VARIABLE DEFINITIONS

- Variables can be used only within `control`, `testcase`, `function`, `altstep`, component type definition and block of statements scope units
- No global variables – no definition in module definition part
- Must appear always at the beginning of statement block

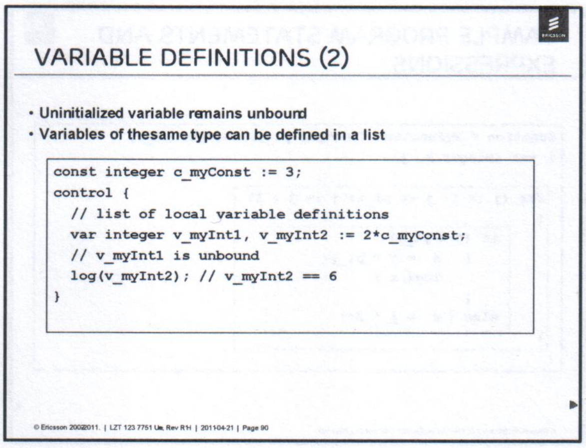
```
control { var integer i1 }
```

- Exception is the iteration counter of for loops

```
for(var integer i:=1; i<9; i:=i+1) { /*…*/ }
```

- Optionally, an initial value may be assigned to the variable.

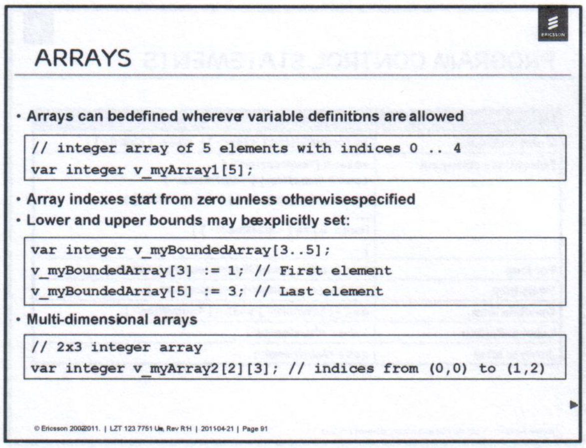
```
control { var integer i1 := 1 }
```



VARIABLE DEFINITIONS (2)

- Uninitialized variable remains unbound
- Variables of the same type can be defined in a list

```
const integer c_myConst := 3;
control {
  // list of local variable definitions
  var integer v_myInt1, v_myInt2 := 2*c_myConst;
  // v_myInt1 is unbound
  log(v_myInt2); // v_myInt2 == 6
}
```



ARRAYS

- Arrays can be defined wherever variable definitions are allowed

```
// integer array of 5 elements with indices 0 .. 4
var integer v_myArray1[5];
```

- Array indexes start from zero unless otherwise specified
- Lower and upper bounds may be explicitly set:

```
var integer v_myBoundedArray[3..5];
v_myBoundedArray[3] := 1; // First element
v_myBoundedArray[5] := 3; // Last element
```

- Multi-dimensional arrays

```
// 2x3 integer array
var integer v_myArray2[2][3]; // indices from (0,0) to (1,2)
```

ARRAYS (2)

- Value list notation may be used to setarray values

```
v_myArray1 := {1,2,3,4,5}; // one dimensional array
v_myArray2 := {{12,13,14},{22,23,24}}; // 2D array
```

- A multidimensional array may be replaced by `record of` types:

```
// 2x3 integer matrix with 2D array
var integer v_myArray2[2][3];
// equivalent IntMatrix definition using record of types
type record length(3) of integer IntVector;
type record length(2) of IntVector IntMatrix;
// v_myArray2 and v_myArray2WithRecordOf are equivalent
// from the users' perspective
var IntMatrix v_myArray2WithRecordOf;
```

- `record of` arrays without length restiction may contain any number of elements

MODULE PARAMETERS

- Parameter values
  - Can be set in the test environment (e.g. configuration file)
  - May have default values
  - Remain constants during test run
- Parameters can be imported from another module
- Can only take values,templates are forbidden

```
module MyModule
{
    modulepar integer tsp_myPar1a := 0, tsp_myPar1b;
    // module parameter w/o default value
    modulepar octetstring tsp_myPar2;
}
```

VI. PROGRAM STATEMENTS AND OPERATORS

EXPRESSIONS
ASSIGNMENTS
PROGRAM CONTROL STATEMENTS
OPERATORS
EXAMPLE

CONTENTS

EXPRESSIONS, ASSIGNMENTS, `log`, `action` AND `stop`

Statement	Keyword or symbol
Expression Condition (Boolean expression)	e.g. `2*f1(v1,c2)+1` e.g. `x+y<z`
Assignment (not an operator!)	*LHS* := *RHS* e.g. `v := { 1, f2(v1) }`
Print entries into log	`log(a);` `log(a, ...);`
Stimulate or carry out an action	`action("Press button!");`
Stop execution	`stop;`

PROGRAM CONTROL STATEMENTS

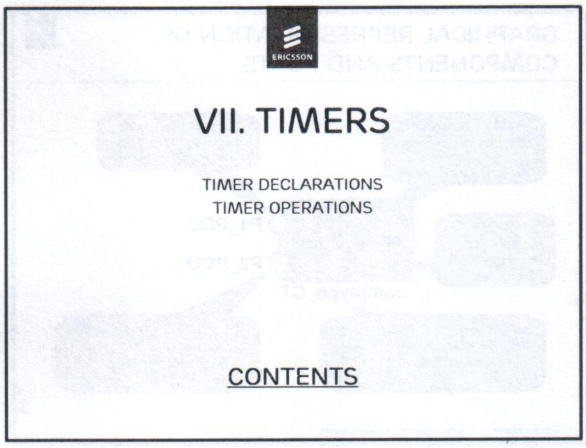
Statement	Synopsis
If-else statement	`if` (<*condition*>) { <*stmt*> } [`else` { <*stmt*> }]

Select-Case statement	`select` (<*expression*>) {
	`case` (<*template*>) { <*statement*> }
	[`case` (<*template-list*>) { <*statement*> }]
	...
	[`case else` { <*statement*> }]
	}

For loop	`for` (<*init*>; <*condition*>; <*expr*>) { <*stmt*> }
While loop	`while` (<*condition*>) { <*statement*> }
Do-while loop	`do` { <*statement*> } `while` (<*condition*>);
Label definition	`label` <*labelname*>;
Jump to label	`goto` <*labelname*>;

SAMPLE PROGRAM STATEMENTS AND EXPRESSIONS

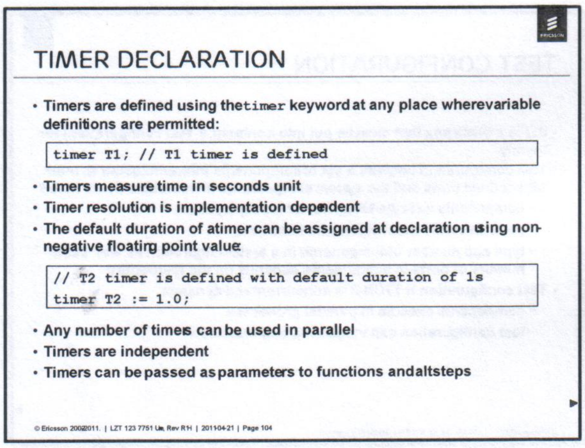
```
function f_MyFunction (integer pl_y, integer pl_i)
{ var integer x, j;

  for (j := 1; j <= pl_i; j := j + 1)
  {
      if (j < pl_y)
      {  x := j * pl_y;
         log( x )
      }
      else { x := j * 3;}
  }
}
```



VII. TIMERS

TIMER DECLARATIONS
TIMER OPERATIONS

CONTENTS



TIMER DECLARATION

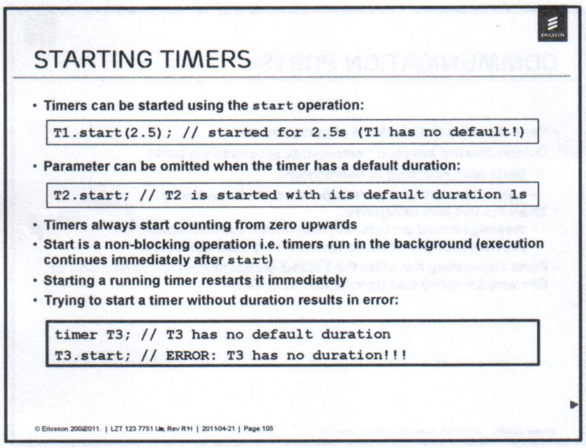
- Timers are defined using the `timer` keyword at any place where variable definitions are permitted:

```
timer T1; // T1 timer is defined
```

- Timers measure time in seconds unit
- Timer resolution is implementation dependent
- The default duration of a timer can be assigned at declaration using non-negative floating point value

```
// T2 timer is defined with default duration of 1s
timer T2 := 1.0;
```

- Any number of timers can be used in parallel
- Timers are independent
- Timers can be passed as parameters to functions and altsteps



STARTING TIMERS

- Timers can be started using the `start` operation:

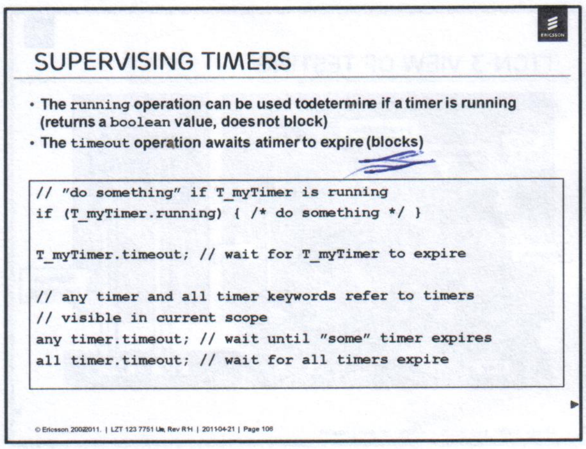
```
T1.start(2.5); // started for 2.5s (T1 has no default!)
```

- Parameter can be omitted when the timer has a default duration:

```
T2.start; // T2 is started with its default duration 1s
```

- Timers always start counting from zero upwards
- Start is a non-blocking operation i.e. timers run in the background (execution continues immediately after `start`)
- Starting a running timer restarts it immediately
- Trying to start a timer without duration results in error:

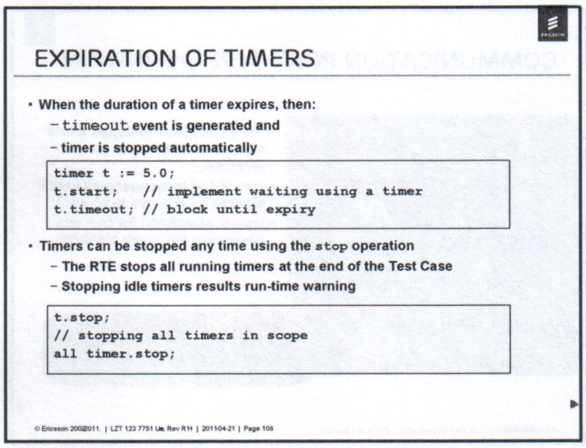
```
timer T3; // T3 has no default duration
T3.start; // ERROR: T3 has no duration!!!
```



SUPERVISING TIMERS

- The `running` operation can be used to determine if a timer is running (returns a `boolean` value, does not block)
- The `timeout` operation awaits a timer to expire (blocks)

```
// "do something" if T_myTimer is running
if (T_myTimer.running) { /* do something */ }

T_myTimer.timeout; // wait for T_myTimer to expire

// any timer and all timer keywords refer to timers
// visible in current scope
any timer.timeout; // wait until "some" timer expires
all timer.timeout; // wait for all timers expire
```



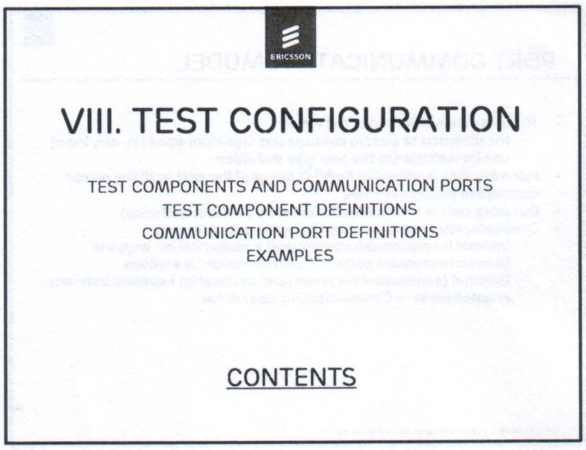
EXPIRATION OF TIMERS

- When the duration of a timer expires, then:
  - `timeout` event is generated and
  - timer is stopped automatically

```
timer t := 5.0;
t.start;   // implement waiting using a timer
t.timeout; // block until expiry
```

- Timers can be stopped any time using the `stop` operation
  - The RTE stops all running timers at the end of the Test Case
  - Stopping idle timers results run-time warning

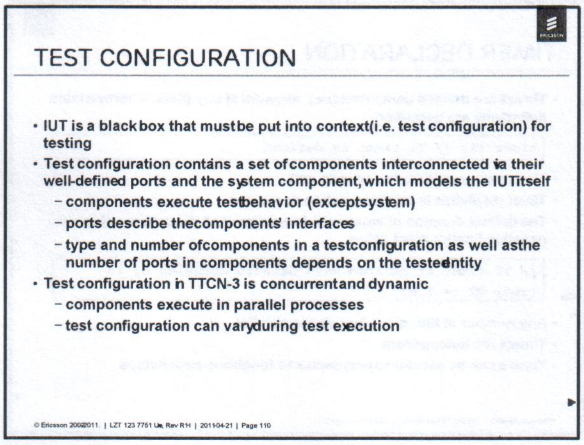
```
t.stop;
// stopping all timers in scope
all timer.stop;
```



VIII. TEST CONFIGURATION

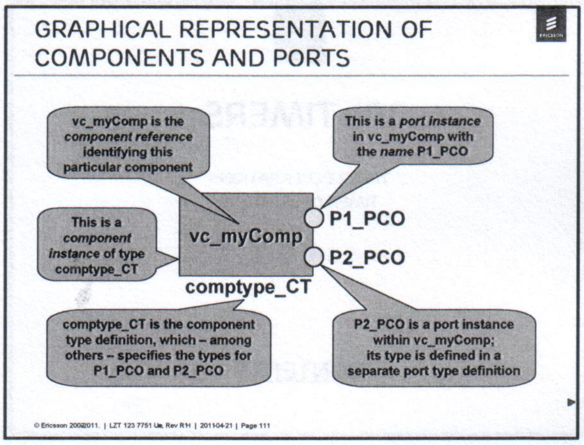
TEST COMPONENTS AND COMMUNICATION PORTS
TEST COMPONENT DEFINITIONS
COMMUNICATION PORT DEFINITIONS
EXAMPLES

CONTENTS



TEST CONFIGURATION

- IUT is a black box that must be put into context (i.e. test configuration) for testing
- Test configuration contains a set of components interconnected via their well-defined ports and the system component, which models the IUT itself
  - components execute test behavior (except system)
  - ports describe the components' interfaces
  - type and number of components in a test configuration as well as the number of ports in components depends on the tested entity
- Test configuration in TTCN-3 is concurrent and dynamic
  - components execute in parallel processes
  - test configuration can vary during test execution



GRAPHICAL REPRESENTATION OF COMPONENTS AND PORTS

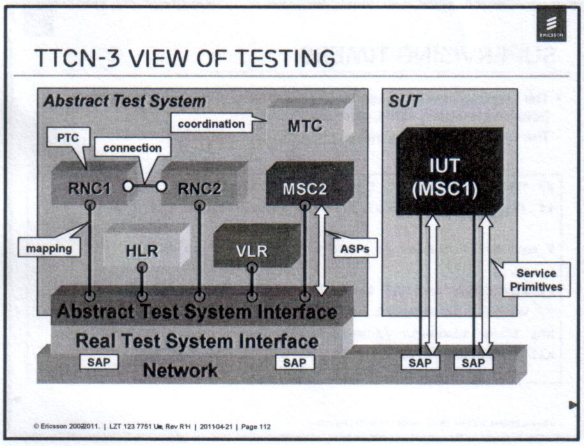
vc_myComp is the *component reference* identifying this particular component

This is a *port instance* in vc_myComp with the *name* P1_PCO

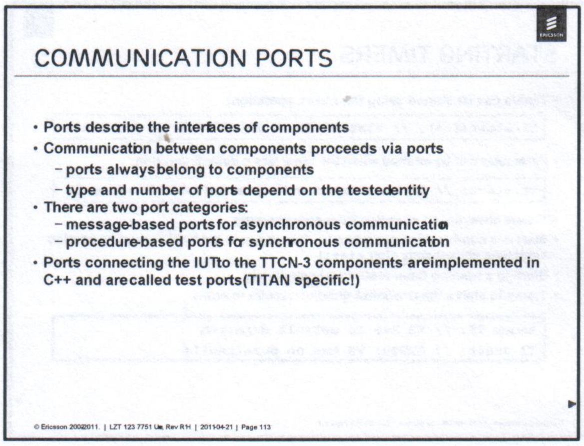
This is a *component instance* of type comptype_CT

comptype_CT is the component type definition, which – among others – specifies the types for P1_PCO and P2_PCO

P2_PCO is a port instance within vc_myComp; its type is defined in a separate port type definition

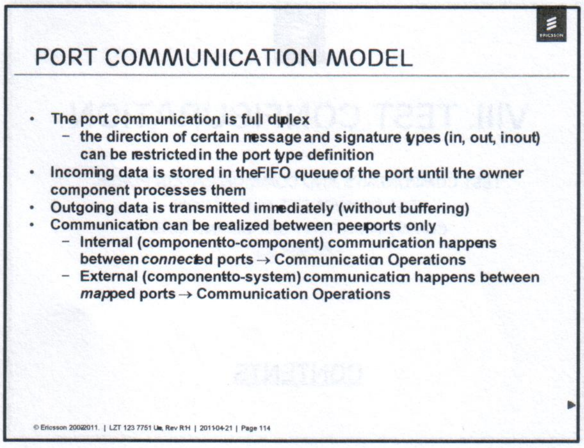


TTCN-3 VIEW OF TESTING



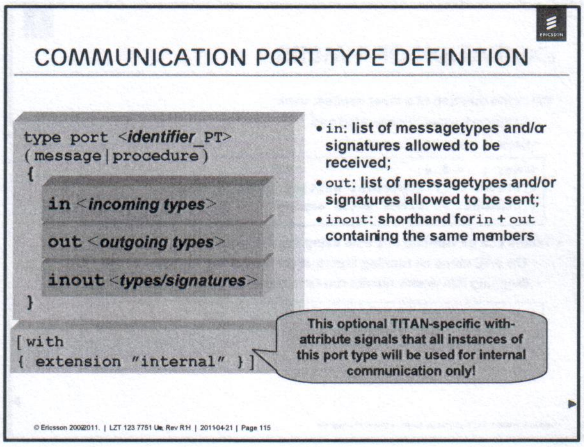
COMMUNICATION PORTS

- Ports describe the interfaces of components
- Communication between components proceeds via ports
  - ports always belong to components
  - type and number of ports depend on the tested entity
- There are two port categories:
  - message-based ports for asynchronous communication
  - procedure-based ports for synchronous communication
- Ports connecting the IUT to the TTCN-3 components are implemented in C++ and are called test ports (TITAN specific!)



PORT COMMUNICATION MODEL

- The port communication is full duplex
  - the direction of certain message and signature types (in, out, inout) can be restricted in the port type definition
- Incoming data is stored in the FIFO queue of the port until the owner component processes them
- Outgoing data is transmitted immediately (without buffering)
- Communication can be realized between peer ports only
  - Internal (component-to-component) communication happens between *connected* ports → Communication Operations
  - External (component-to-system) communication happens between *mapped* ports → Communication Operations

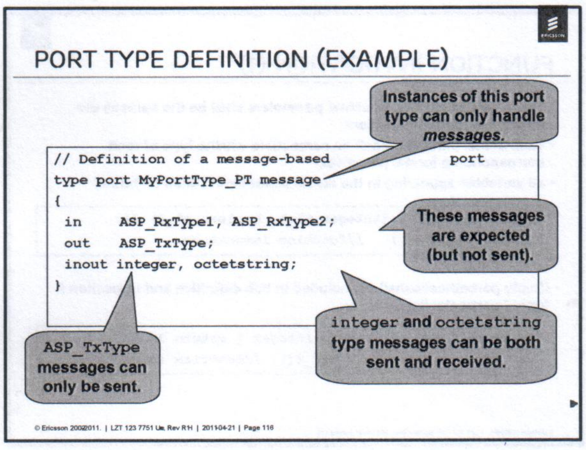


COMMUNICATION PORT TYPE DEFINITION

```
type port <identifier_PT>
(message|procedure)
{
  in <incoming types>
  out <outgoing types>
  inout <types/signatures>
}
[with
{ extension "internal" }]
```

- `in`: list of messagetypes and/or signatures allowed to be received;
- `out`: list of messagetypes and/or signatures allowed to be sent;
- `inout`: shorthand for `in` + `out` containing the same members

This optional TITAN-specific with-attribute signals that all instances of this port type will be used for internal communication only!



PORT TYPE DEFINITION (EXAMPLE)

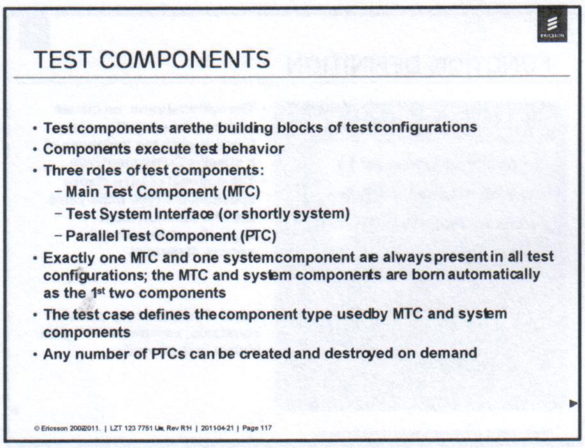
```
// Definition of a message-based                port
type port MyPortType_PT message
{
 in     ASP_RxType1, ASP_RxType2;
 out    ASP_TxType;
 inout integer, octetstring;
}
```

Instances of this port type can only handle *messages*.

These messages are expected (but not sent).

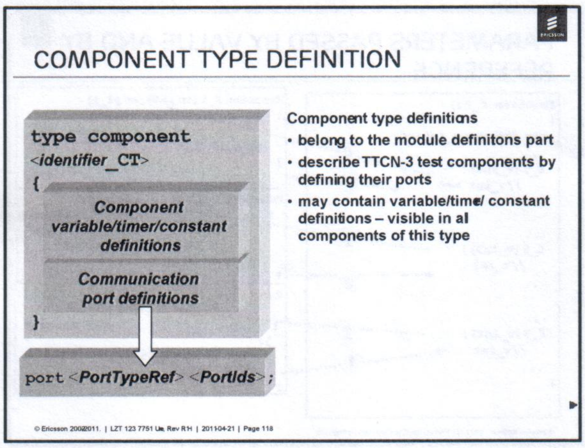
`ASP_TxType` messages can only be sent.

`integer` and `octetstring` type messages can be both sent and received.



TEST COMPONENTS

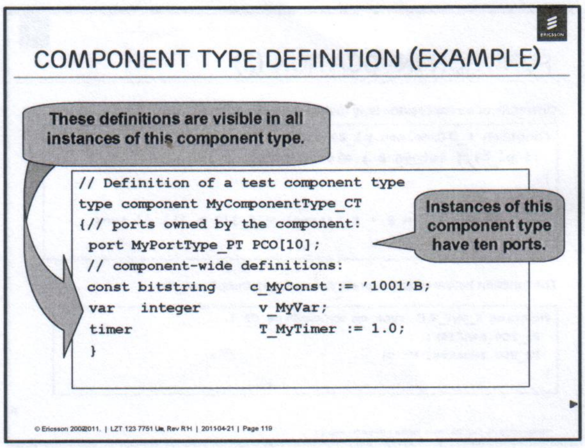
- Test components are the building blocks of test configurations
- Components execute test behavior
- Three roles of test components:
  - Main Test Component (MTC)
  - Test System Interface (or shortly system)
  - Parallel Test Component (PTC)
- Exactly one MTC and one system component are always present in all test configurations; the MTC and system components are born automatically as the 1st two components
- The test case defines the component type used by MTC and system components
- Any number of PTCs can be created and destroyed on demand



COMPONENT TYPE DEFINITION

`type component`
<identifier_CT>
`{`
Component variable/timer/constant definitions
Communication port definitions
`}`
`port` *<PortTypeRef>* *<PortIds>* `;`

Component type definitions
- belong to the module definitons part
- describe TTCN-3 test components by defining their ports
- may contain variable/timer/ constant definitions – visible in all components of this type

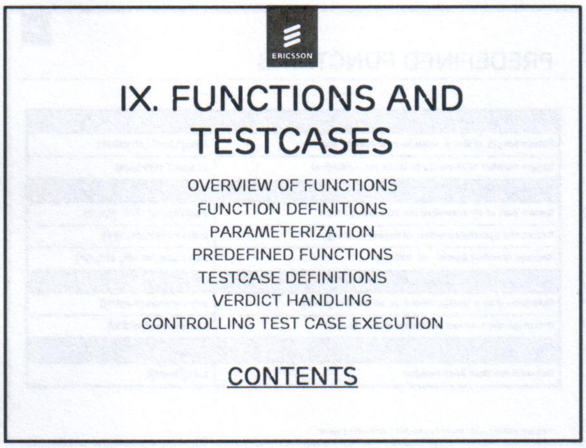


COMPONENT TYPE DEFINITION (EXAMPLE)

These definitions are visible in all instances of this component type.

```
// Definition of a test component type
type component MyComponentType_CT
{// ports owned by the component:
 port MyPortType_PT PCO[10];
 // component-wide definitions:
 const bitstring    c_MyConst := '1001'B;
 var   integer       v_MyVar;
 timer               T_MyTimer := 1.0;
 }
```

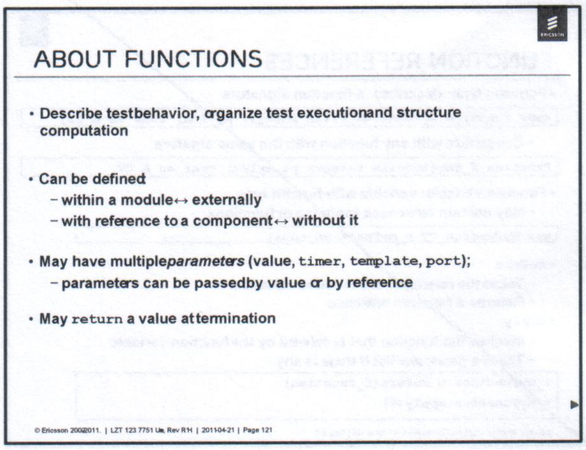
Instances of this component type have ten ports.



IX. FUNCTIONS AND TESTCASES

OVERVIEW OF FUNCTIONS
FUNCTION DEFINITIONS
PARAMETERIZATION
PREDEFINED FUNCTIONS
TESTCASE DEFINITIONS
VERDICT HANDLING
CONTROLLING TEST CASE EXECUTION

CONTENTS



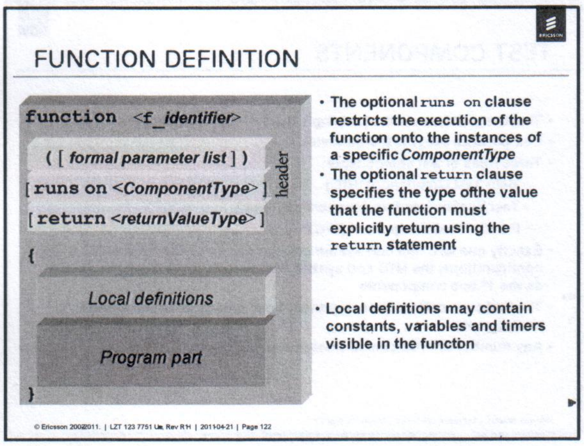
ABOUT FUNCTIONS

- Describe test behavior, organize test execution and structure computation

- Can be defined
  - within a module ↔ externally
  - with reference to a component ↔ without it

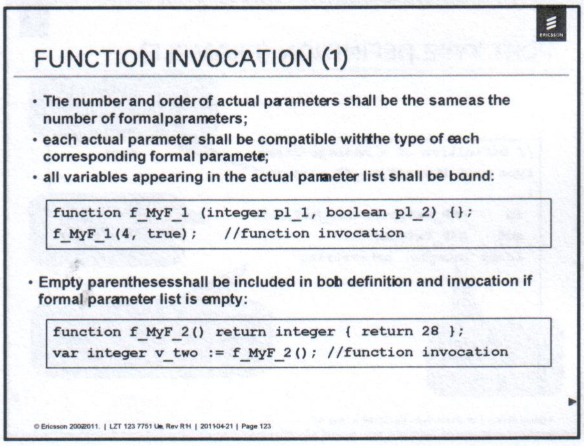
- May have multiple *parameters* (value, `timer`, `template`, `port`);
  - parameters can be passed by value or by reference

- May `return` a value at termination



FUNCTION DEFINITION

- The optional `runs on` clause restricts the execution of the function onto the instances of a specific *ComponentType*
- The optional `return` clause specifies the type of the value that the function must explicitly return using the `return` statement
- Local definitions may contain constants, variables and timers visible in the function



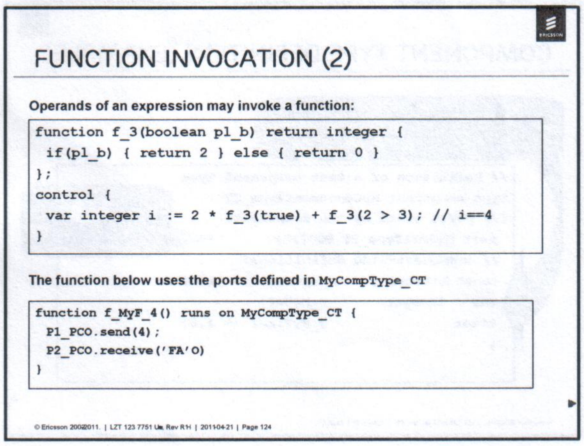
FUNCTION INVOCATION (1)

- The number and order of actual parameters shall be the same as the number of formal parameters;
- each actual parameter shall be compatible with the type of each corresponding formal parameter;
- all variables appearing in the actual parameter list shall be bound:

```
function f_MyF_1 (integer pl_1, boolean pl_2) {};
f_MyF_1(4, true);    //function invocation
```

- Empty parentheses shall be included in both definition and invocation if formal parameter list is empty:

```
function f_MyF_2() return integer { return 28 };
var integer v_two := f_MyF_2(); //function invocation
```



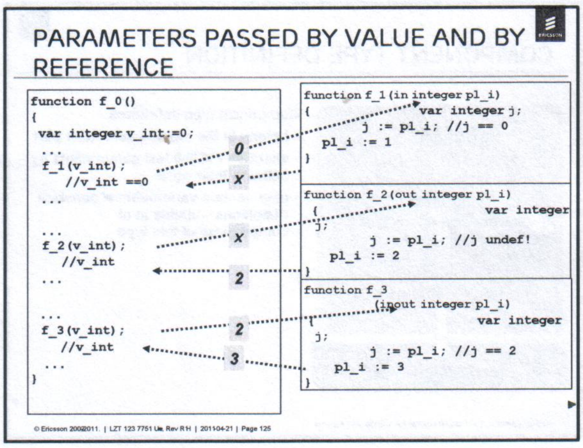
FUNCTION INVOCATION (2)

Operands of an expression may invoke a function:

```
function f_3(boolean pl_b) return integer {
  if(pl_b) { return 2 } else { return 0 }
};
control {
  var integer i := 2 * f_3(true) + f_3(2 > 3); // i==4
}
```

The function below uses the ports defined in `MyCompType_CT`

```
function f_MyF_4() runs on MyCompType_CT {
  P1_PCO.send(4);
  P2_PCO.receive('FA'O)
}
```



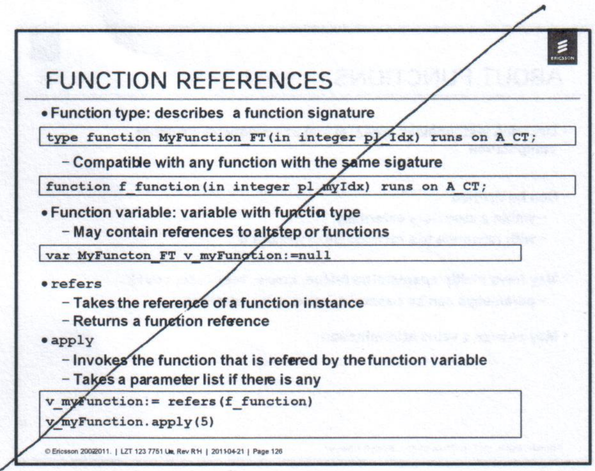
PARAMETERS PASSED BY VALUE AND BY REFERENCE

```
function f_0()
{
 var integer v_int:=0;
 ...
 f_1(v_int);
    //v_int ==0
 ...

 ...
 f_2(v_int);
    //v_int
 ...

 ...
 f_3(v_int);
    //v_int
 ...
}
```

```
function f_1(in integer pl_i)
{                        var integer j;
       j := pl_i; //j == 0
  pl_i := 1
}
```

```
function f_2(out integer pl_i)
 {                           var integer
 j;
       j := pl_i; //j undef!
   pl_i := 2
}
```

```
function f_3
         (inout integer pl_i)
{                            var integer
 j;
       j := pl_i; //j == 2
  pl_i := 3
}
```



FUNCTION REFERENCES

- Function type: describes a function signature

```
type function MyFunction_FT(in integer pl_Idx) runs on A_CT;
```

  - Compatible with any function with the same sigature

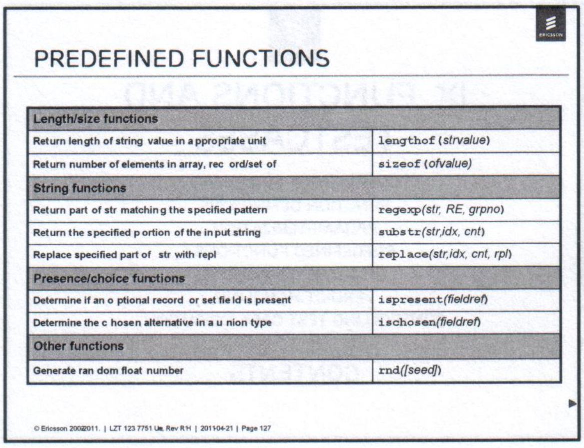
```
function f_function(in integer pl_myIdx) runs on A_CT;
```

- Function variable: variable with function type
  - May contain references to altstep or functions

```
var MyFuncton_FT v_myFunction:=null
```

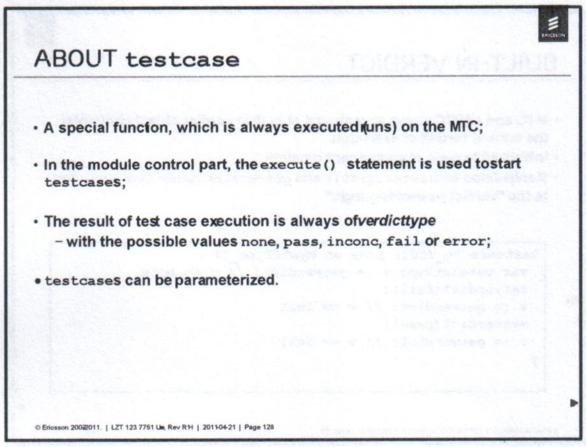
- `refers`
  - Takes the reference of a function instance
  - Returns a function reference
- `apply`
  - Invokes the function that is refered by the function variable
  - Takes a parameter list if there is any

```
v_myFunction:= refers(f_function)
v_myFunction.apply(5)
```



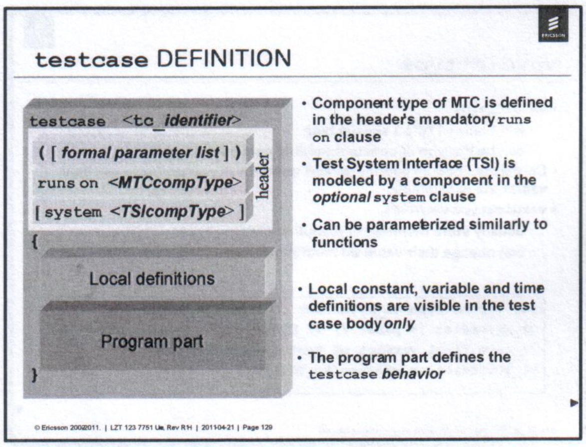
PREDEFINED FUNCTIONS

Length/size functions	
Return length of string value in appropriate unit	`lengthof` (*strvalue*)
Return number of elements in array, record/set of	`sizeof` (*ofvalue*)
String functions	
Return part of str matching the specified pattern	`regexp`(*str, RE, grpno*)
Return the specified portion of the input string	`substr`(*str,idx, cnt*)
Replace specified part of str with repl	`replace`(*str,idx, cnt, rpl*)
Presence/choice functions	
Determine if an optional record or set field is present	`ispresent`(*fieldref*)
Determine the chosen alternative in a union type	`ischosen`(*fieldref*)
Other functions	
Generate random float number	`rnd`(*[seed]*)



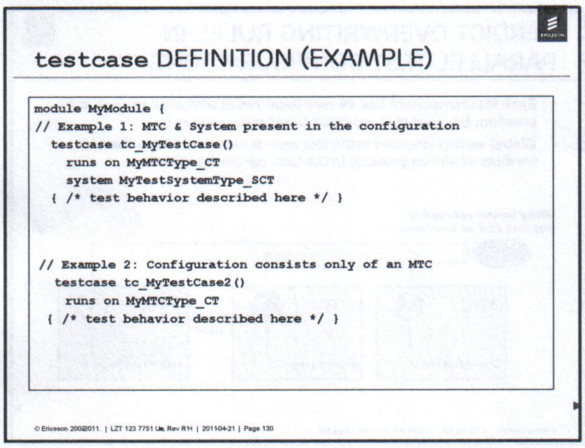
ABOUT `testcase`

- A special function, which is always executed (runs) on the MTC;
- In the module control part, the `execute()` statement is used to start `testcases`;
- The result of test case execution is always of *verdicttype*
  - with the possible values `none`, `pass`, `inconc`, `fail` or `error`;
- `testcases` can be parameterized.



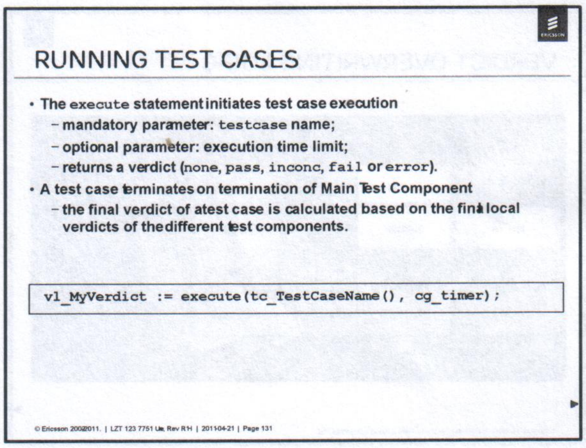
`testcase` DEFINITION

- Component type of MTC is defined in the header's mandatory `runs on` clause
- Test System Interface (TSI) is modeled by a component in the *optional* `system` clause
- Can be parameterized similarly to functions
- Local constant, variable and time definitions are visible in the test case body *only*
- The program part defines the `testcase` *behavior*



`testcase` DEFINITION (EXAMPLE)

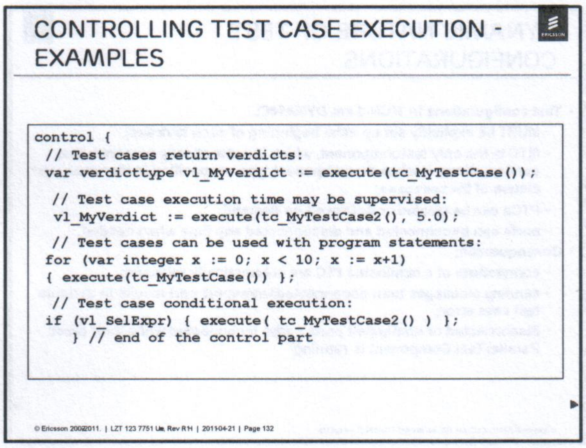
```
module MyModule {
// Example 1: MTC & System present in the configuration
  testcase tc_MyTestCase()
    runs on MyMTCType_CT
    system MyTestSystemType_SCT
  { /* test behavior described here */ }


 // Example 2: Configuration consists only of an MTC
   testcase tc_MyTestCase2()
    runs on MyMTCType_CT
  { /* test behavior described here */ }
```



RUNNING TEST CASES

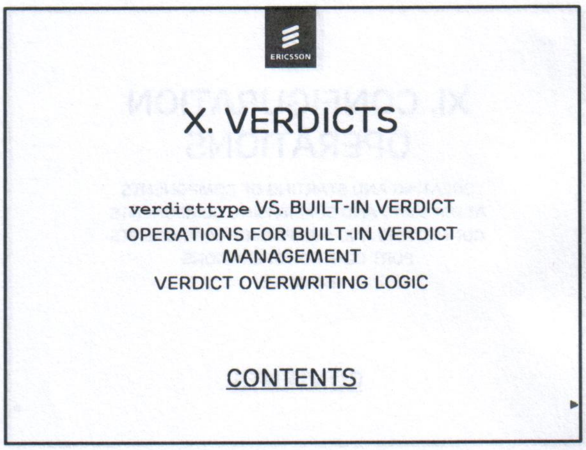
- The `execute` statement initiates test case execution
  - mandatory parameter: `testcase` name;
  - optional parameter: execution time limit;
  - returns a verdict (`none`, `pass`, `inconc`, `fail` or `error`).
- A test case terminates on termination of Main Test Component
  - the final verdict of a test case is calculated based on the final local verdicts of the different test components.

```
vl_MyVerdict := execute(tc_TestCaseName(), cg_timer);
```



CONTROLLING TEST CASE EXECUTION - EXAMPLES

```
control {
 // Test cases return verdicts:
 var verdicttype vl_MyVerdict := execute(tc_MyTestCase());

  // Test case execution time may be supervised:
  vl_MyVerdict := execute(tc_MyTestCase2(), 5.0);

  // Test cases can be used with program statements:
 for (var integer x := 0; x < 10; x := x+1)
 { execute(tc_MyTestCase()) };

  // Test case conditional execution:
 if (vl_SelExpr) { execute( tc_MyTestCase2() ) };
    } // end of the control part
```



X. VERDICTS

`verdicttype` VS. BUILT-IN VERDICT
OPERATIONS FOR BUILT-IN VERDICT MANAGEMENT
VERDICT OVERWRITING LOGIC

CONTENTS

verdicttype

- `verdicttype`
  - is a built-in TTCN-3 special type
  - can be the type of constant, module parameter or variable
- Constants, module parameters and variables of `verdicttype` get their values via assignment
- `verdicttype` variables
  - usually store the result of execution
  - can change their value w/o restriction

```
var verdicttype vl_MyVerdict := fail, vl_TCVerdict;
vl_MyVerdict := pass; // vl_MyVerdict == pass
// save final verdict of test case execution
vl_TCVerdict := execute(tc_TC());
```

BUILT-IN VERDICT

- MTC and all PTCs have an instance of built-in verdict object containing the current verdict of execution
- initialized to `none` at component creation
- Manipulated with `setverdict()` and `getverdict` operations according to the "verdict overwriting logic"

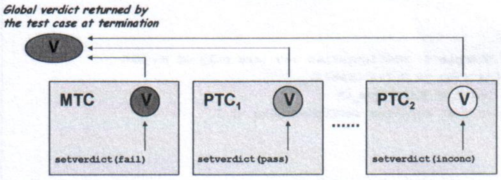
```
testcase tc_TC0() runs on MyMTCType_CT {
  var verdicttype v := getverdict; // v == none
  setverdict(fail);
  v := getverdict; // v == fail
  setverdict(pass);
  v := getverdict; // v == fail
}
```

VERDICT OVERWRITING LOGIC

Result	Partial verdict				
Former value of verdict	*none*	*pass*	*inconc*	*fail*	*error*
none	none	pass	inconc	fail	error
pass	pass	pass	inconc	fail	error
inconc	inconc	inconc	inconc	fail	error
fail	fail	fail	fail	fail	error

VERDICT OVERWRITING RULES IN PARALLEL TEST CONFIGURATIONS

- Each test component has its own local verdict initialized to `none` at its creation; the verdict is modified later by `setverdict()`
- Global verdict returned by the test case is calculated from the local verdicts of all components in the test case configuration.



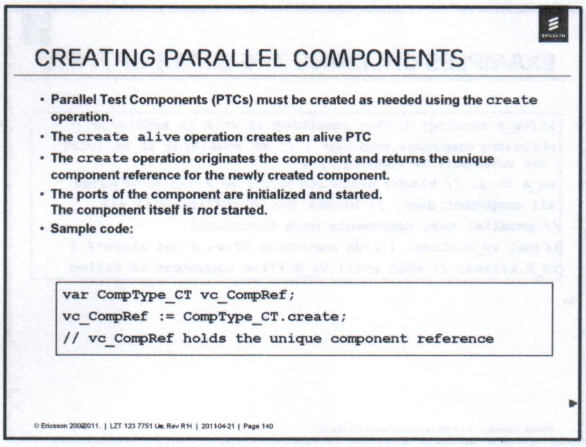
XI. CONFIGURATION OPERATIONS

CREATING AND STARTING OF COMPONENTS
ADDRESSING AND SUPERVISING COMPONENTS
CONNECTING AND MAPPING OF COMPONENTS
PORT CONTROL OPERATIONS
EXAMPLE

CONTENTS

DYNAMIC NATURE OF TEST CONFIGURATIONS

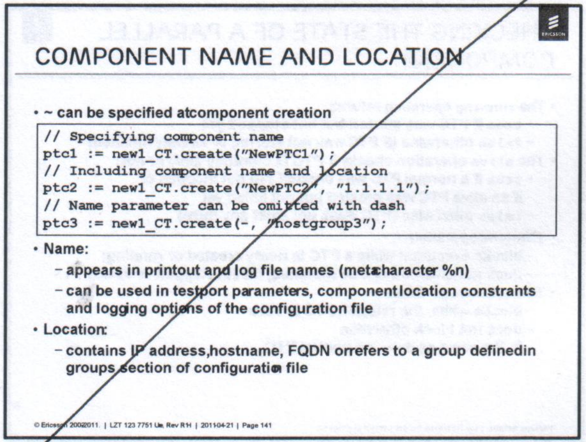
- Test configurations in TTCN-3 are *DYNAMIC*:
  - MUST be explicitly set up at the beginning of each testcase;
  - MTC is the only test component, which is automatically present in test configurations; it takes the component type as specified in the `"runs on"` clause of the `testcase`;
  - PTCs can be created or destroyed on demand;
  - ports can be connected and disconnected any time when needed.
- Consequences:
  - connections of a terminated PTC are automatically released;
  - sending messages to an unconnected/unmapped port results in dynamic test case error;
  - disconnected or unmapped ports can be reconnected while their owner Parallel Test Component is running



CREATING PARALLEL COMPONENTS

- **Parallel Test Components (PTCs) must be created as needed using the `create` operation.**
- **The `create alive` operation creates an alive PTC**
- **The `create` operation originates the component and returns the unique component reference for the newly created component.**
- **The ports of the component are initialized and started.
  The component itself is *not* started.**
- **Sample code:**

```
var CompType_CT vc_CompRef;
vc_CompRef := CompType_CT.create;
// vc_CompRef holds the unique component reference
```

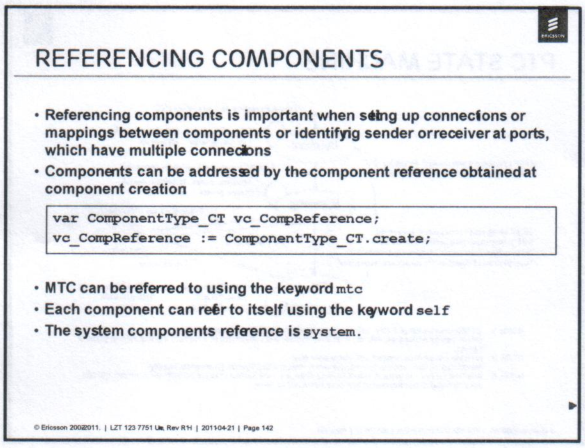


COMPONENT NAME AND LOCATION

- **~ can be specified at component creation**

```
// Specifying component name
ptc1 := new1_CT.create("NewPTC1");
// Including component name and location
ptc2 := new1_CT.create("NewPTC2", "1.1.1.1");
// Name parameter can be omitted with dash
ptc3 := new1_CT.create(-, "hostgroup3");
```

- **Name:**
  - **appears in printout and log file names (metacharacter %n)**
  - **can be used in testport parameters, component location constraints and logging options of the configuration file**
- **Location:**
  - **contains IP address, hostname, FQDN or refers to a group defined in groups section of configuration file**

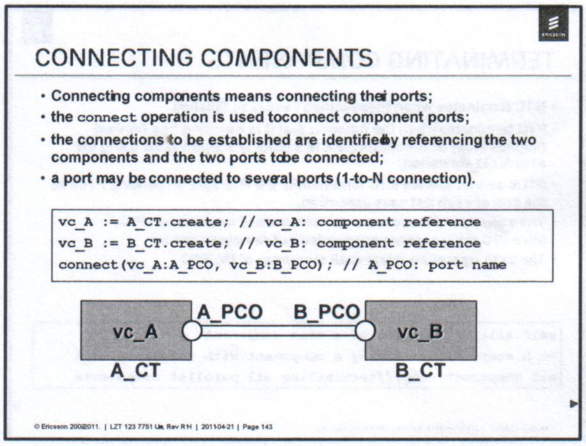


REFERENCING COMPONENTS

- **Referencing components is important when setting up connections or mappings between components or identifying sender or receiver at ports, which have multiple connections**
- **Components can be addressed by the component reference obtained at component creation**

```
var ComponentType_CT vc_CompReference;
vc_CompReference := ComponentType_CT.create;
```

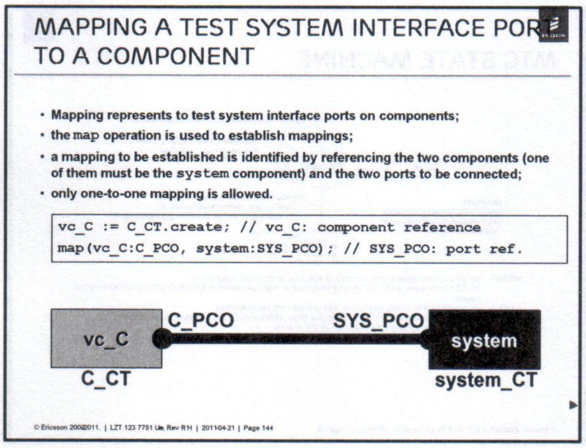
- **MTC can be referred to using the keyword `mtc`**
- **Each component can refer to itself using the keyword `self`**
- **The system components reference is `system`.**



CONNECTING COMPONENTS

- **Connecting components means connecting their ports;**
- **the `connect` operation is used to connect component ports;**
- **the connections to be established are identified by referencing the two components and the two ports to be connected;**
- **a port may be connected to several ports (1-to-N connection).**

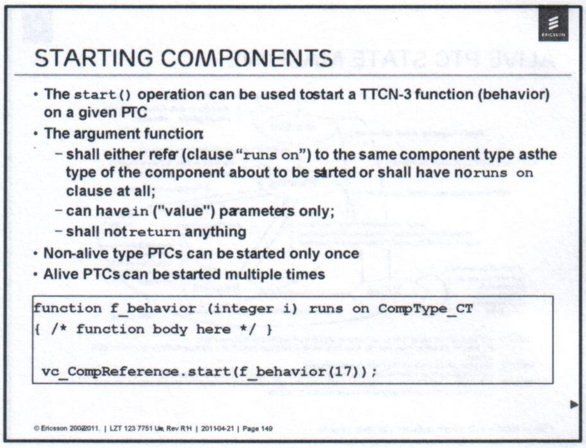
```
vc_A := A_CT.create; // vc_A: component reference
vc_B := B_CT.create; // vc_B: component reference
connect(vc_A:A_PCO, vc_B:B_PCO); // A_PCO: port name
```



MAPPING A TEST SYSTEM INTERFACE PORT TO A COMPONENT

- **Mapping represents to test system interface ports on components;**
- **the `map` operation is used to establish mappings;**
- **a mapping to be established is identified by referencing the two components (one of them must be the `system` component) and the two ports to be connected;**
- **only one-to-one mapping is allowed.**

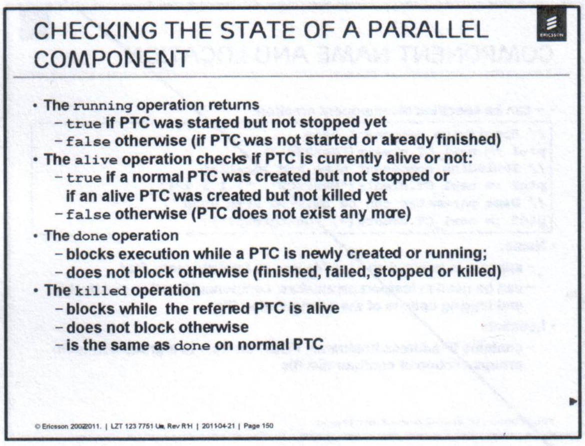
```
vc_C := C_CT.create; // vc_C: component reference
map(vc_C:C_PCO, system:SYS_PCO); // SYS_PCO: port ref.
```



STARTING COMPONENTS

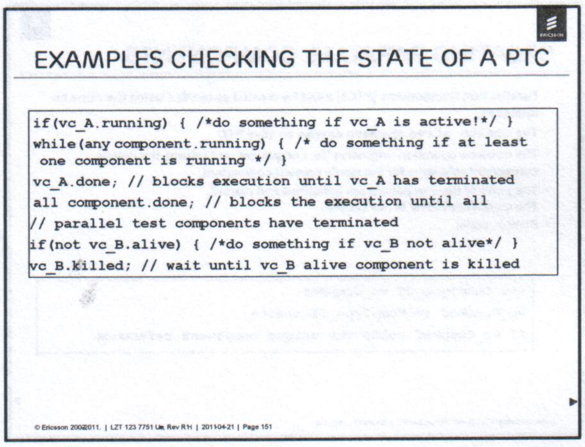
- **The `start()` operation can be used to start a TTCN-3 function (behavior) on a given PTC**
- **The argument function:**
  - **shall either refer (clause "`runs on`") to the same component type as the type of the component about to be started or shall have no `runs on` clause at all;**
  - **can have `in` ("value") parameters only;**
  - **shall not `return` anything**
- **Non-alive type PTCs can be started only once**
- **Alive PTCs can be started multiple times**

```
function f_behavior (integer i) runs on CompType_CT
{ /* function body here */ }

 vc_CompReference.start(f_behavior(17));
```



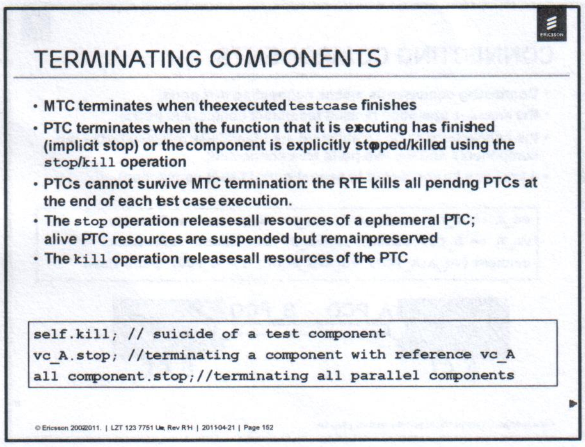
CHECKING THE STATE OF A PARALLEL COMPONENT

- The `running` operation returns
  - `true` if PTC was started but not stopped yet
  - `false` otherwise (if PTC was not started or already finished)
- The `alive` operation checks if PTC is currently alive or not:
  - `true` if a normal PTC was created but not stopped or if an alive PTC was created but not killed yet
  - `false` otherwise (PTC does not exist any more)
- The `done` operation
  - blocks execution while a PTC is newly created or running;
  - does not block otherwise (finished, failed, stopped or killed)
- The `killed` operation
  - blocks while the referred PTC is alive
  - does not block otherwise
  - is the same as `done` on normal PTC



EXAMPLES CHECKING THE STATE OF A PTC

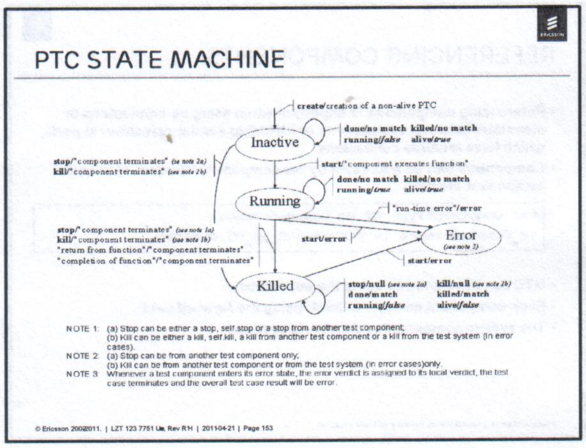
```
if(vc_A.running) { /*do something if vc_A is active!*/ }
while(any component.running) { /* do something if at least
 one component is running */ }
vc_A.done; // blocks execution until vc_A has terminated
all component.done; // blocks the execution until all
// parallel test components have terminated
if(not vc_B.alive) { /*do something if vc_B not alive*/ }
vc_B.killed; // wait until vc_B alive component is killed
```



TERMINATING COMPONENTS

- MTC terminates when the executed `testcase` finishes
- PTC terminates when the function that it is executing has finished (implicit stop) or the component is explicitly stopped/killed using the `stop`/`kill` operation
- PTCs cannot survive MTC termination: the RTE kills all pending PTCs at the end of each test case execution.
- The `stop` operation releases all resources of a ephemeral PTC; alive PTC resources are suspended but remain preserved
- The `kill` operation releases all resources of the PTC

```
self.kill; // suicide of a test component
vc_A.stop; //terminating a component with reference vc_A
all component.stop;//terminating all parallel components
```

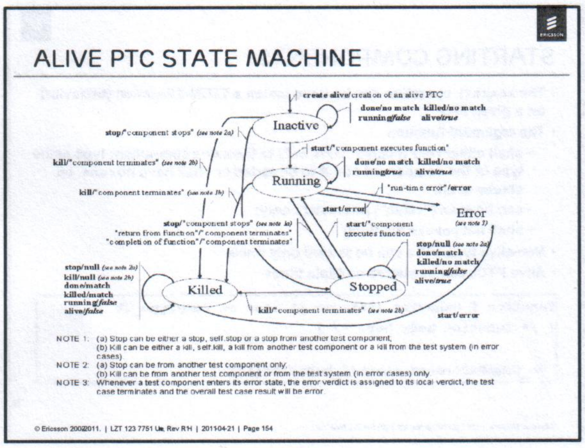


PTC STATE MACHINE

NOTE 1: (a) Stop can be either a stop, self.stop or a stop from another test component;
(b) Kill can be either a kill, self.kill, a kill from another test component or a kill from the test system (in error cases).

NOTE 2: (a) Stop can be from another test component only;
(b) Kill can be from another test component or from the test system (in error cases)only.

NOTE 3: Whenever a test component enters its error state, the error verdict is assigned to its local verdict, the test case terminates and the overall test case result will be error.

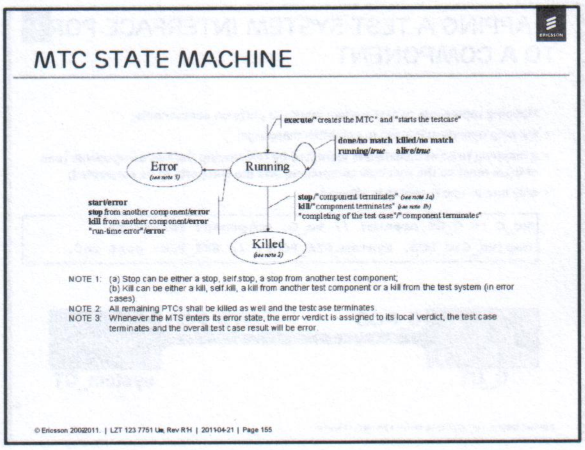


ALIVE PTC STATE MACHINE

NOTE 1: (a) Stop can be either a stop, self.stop or a stop from another test component,
(b) Kill can be either a kill, self.kill, a kill from another test component or a kill from the test system (in error cases)

NOTE 2: (a) Stop can be from another test component only.
(b) Kill can be from another test component or from the test system (in error cases) only

NOTE 3: Whenever a test component enters its error state, the error verdict is assigned to its local verdict, the test case terminates and the overall test case result will be error.



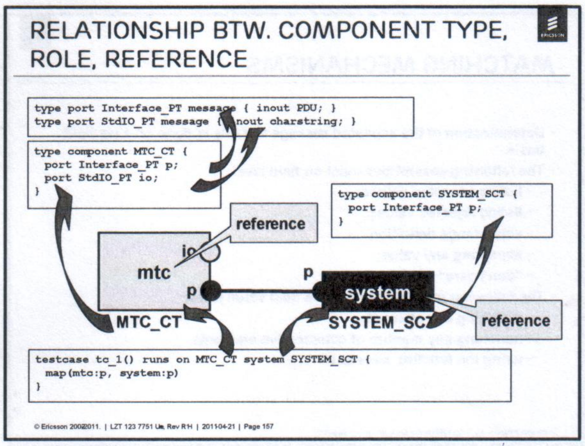
MTC STATE MACHINE

NOTE 1: (a) Stop can be either a stop, self.stop, a stop from another test component;
(b) Kill can be either a kill, self.kill, a kill from another test component or a kill from the test system (in error cases)

NOTE 2: All remaining PTCs shall be killed as well and the testcase terminates.

NOTE 3: Whenever the MTS enters its error state, the error verdict is assigned to its local verdict, the test case terminates and the overall test case result will be error.

create -> connect -> start -> done



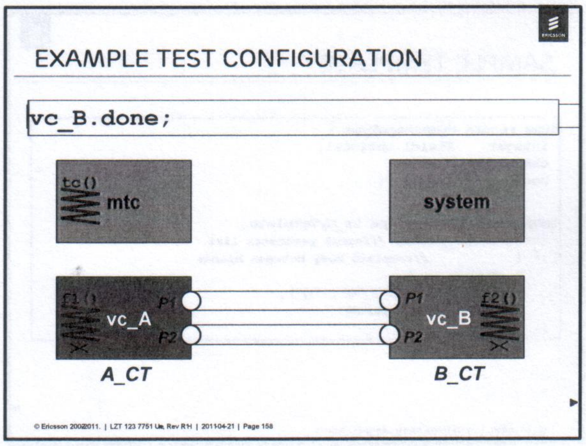
RELATIONSHIP BTW. COMPONENT TYPE, ROLE, REFERENCE

```
type port Interface_PT message { inout PDU; }
type port StdIO_PT message { inout charstring; }
```

```
type component MTC_CT {
 port Interface_PT p;
 port StdIO_PT io;
}
```

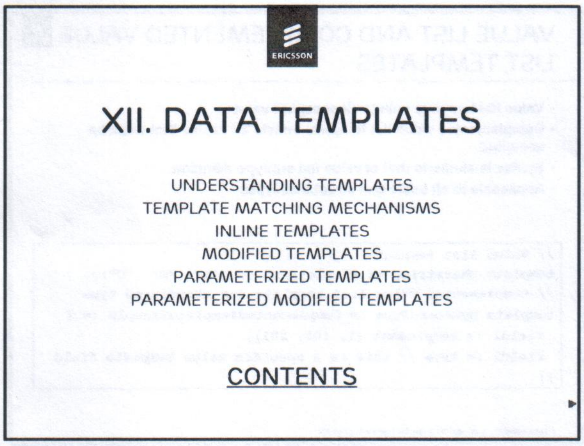
```
type component SYSTEM_SCT {
 port Interface_PT p;
}
```

```
testcase tc_1() runs on MTC_CT system SYSTEM_SCT {
 map(mtc:p, system:p)
}
```



EXAMPLE TEST CONFIGURATION

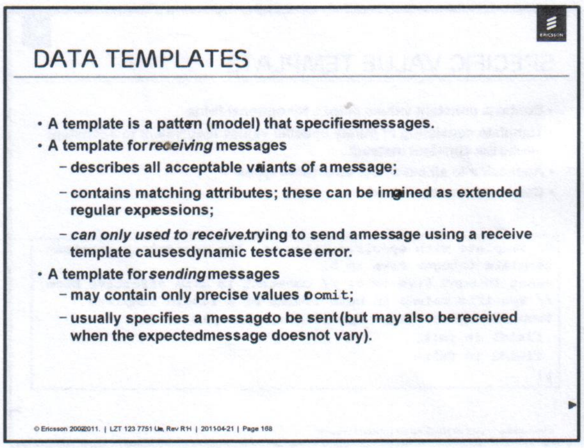
```
vc_B.done;
```



XII. DATA TEMPLATES

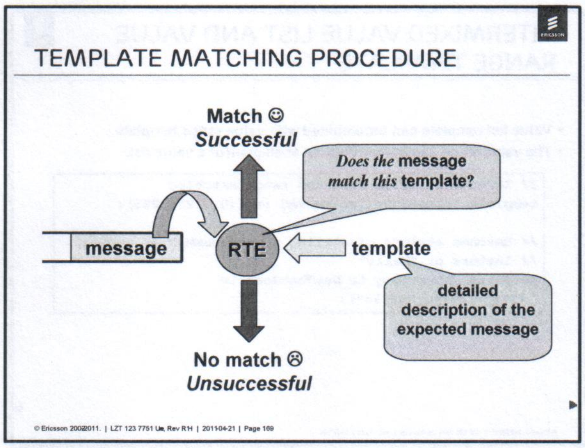
UNDERSTANDING TEMPLATES
TEMPLATE MATCHING MECHANISMS
INLINE TEMPLATES
MODIFIED TEMPLATES
PARAMETERIZED TEMPLATES
PARAMETERIZED MODIFIED TEMPLATES

CONTENTS

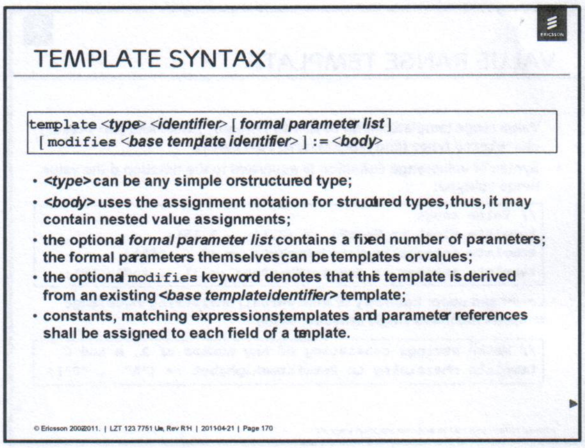


DATA TEMPLATES

- A template is a pattern (model) that specifies messages.
- A template for *receiving* messages
  - describes all acceptable variants of a message;
  - contains matching attributes; these can be imagined as extended regular expressions;
  - *can only used to receive*. trying to send a message using a receive template causes dynamic testcase error.
- A template for *sending* messages
  - may contain only precise values or `omit`;
  - usually specifies a message to be sent (but may also be received when the expected message does not vary).



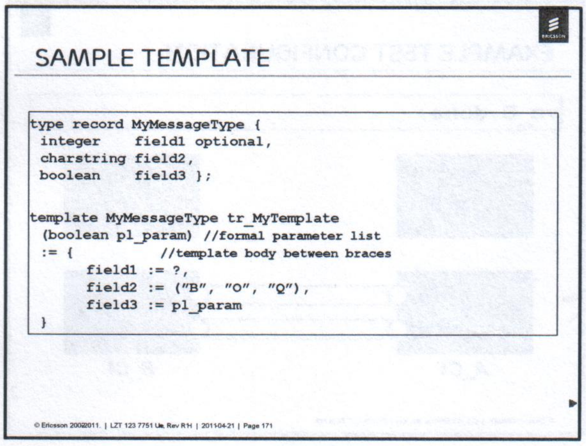
TEMPLATE MATCHING PROCEDURE



TEMPLATE SYNTAX

`template` <*type*> <*identifier*> [*formal parameter list*]
 [`modifies` <*base template identifier*>] := <*body*>

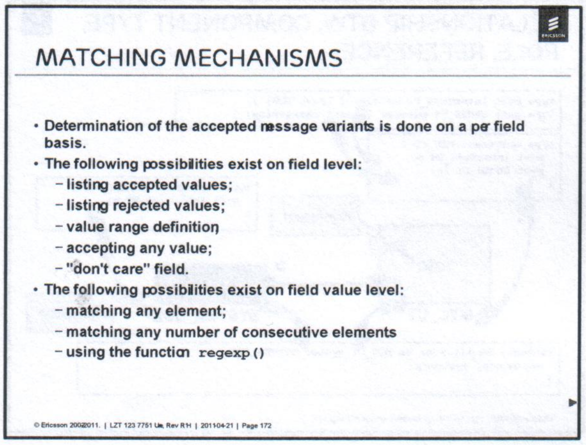
- <*type*> can be any simple or structured type;
- <*body*> uses the assignment notation for structured types, thus, it may contain nested value assignments;
- the optional *formal parameter list* contains a fixed number of parameters; the formal parameters themselves can be templates or values;
- the optional `modifies` keyword denotes that this template is derived from an existing <*base template identifier*> template;
- constants, matching expressions, templates and parameter references shall be assigned to each field of a template.



SAMPLE TEMPLATE

```
type record MyMessageType {
 integer     field1 optional,
 charstring field2,
 boolean     field3 };

template MyMessageType tr_MyTemplate
 (boolean pl_param) //formal parameter list
 := {           //template body between braces
      field1 := ?,
      field2 := ("B", "O", "Q"),
      field3 := pl_param
}
```

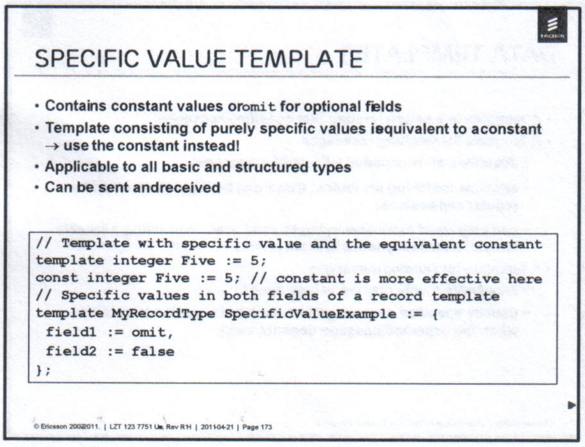
© Ericsson 2002-2011. | LZT 123 7751 Uen, Rev R1H | 2011-04-21 | Page 171



MATCHING MECHANISMS

- Determination of the accepted message variants is done on a per field basis.
- The following possibilities exist on field level:
  - listing accepted values;
  - listing rejected values;
  - value range definition;
  - accepting any value;
  - "don't care" field.
- The following possibilities exist on field value level:
  - matching any element;
  - matching any number of consecutive elements
  - using the function `regexp()`

© Ericsson 2002-2011. | LZT 123 7751 Uen, Rev R1H | 2011-04-21 | Page 172

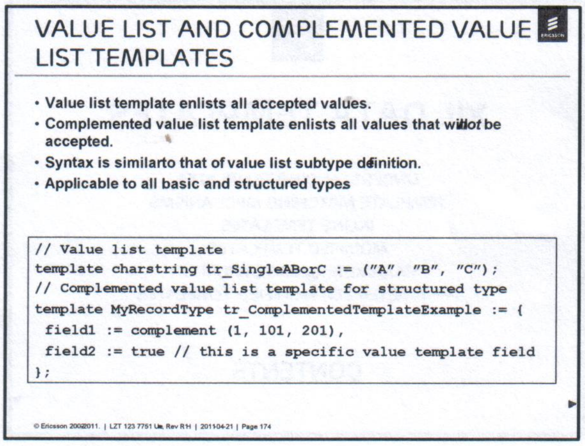


SPECIFIC VALUE TEMPLATE

- Contains constant values or `omit` for optional fields
- Template consisting of purely specific values is equivalent to a constant → use the constant instead!
- Applicable to all basic and structured types
- Can be sent and received

```
// Template with specific value and the equivalent constant
template integer Five := 5;
const integer Five := 5; // constant is more effective here
// Specific values in both fields of a record template
template MyRecordType SpecificValueExample := {
 field1 := omit,
 field2 := false
};
```

© Ericsson 2002-2011. | LZT 123 7751 Uen, Rev R1H | 2011-04-21 | Page 173

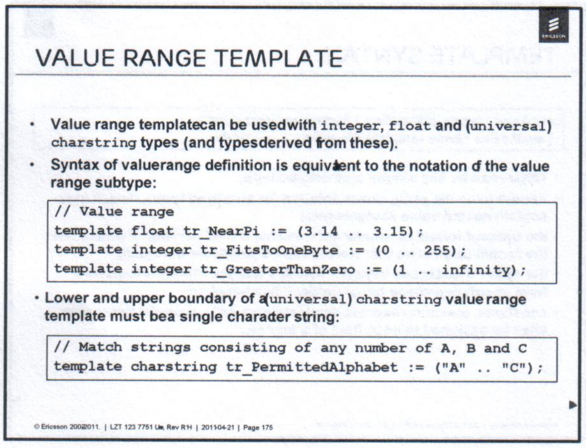


VALUE LIST AND COMPLEMENTED VALUE LIST TEMPLATES

- Value list template enlists all accepted values.
- Complemented value list template enlists all values that will *not* be accepted.
- Syntax is similar to that of value list subtype definition.
- Applicable to all basic and structured types

```
// Value list template
template charstring tr_SingleABorC := ("A", "B", "C");
// Complemented value list template for structured type
template MyRecordType tr_ComplementedTemplateExample := {
 field1 := complement (1, 101, 201),
 field2 := true // this is a specific value template field
};
```

© Ericsson 2002-2011. | LZT 123 7751 Uen, Rev R1H | 2011-04-21 | Page 174



VALUE RANGE TEMPLATE

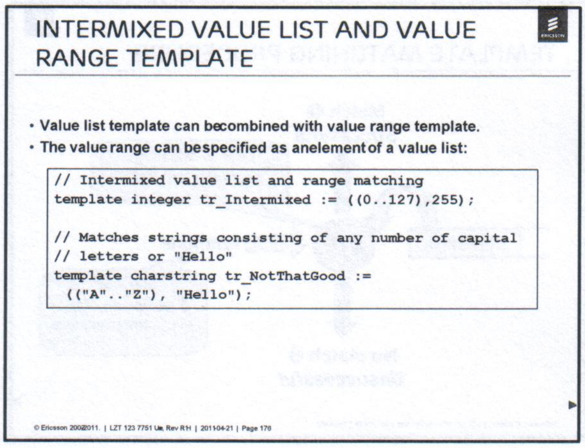
- Value range template can be used with `integer`, `float` and (`universal`) `charstring` types (and types derived from these).
- Syntax of value range definition is equivalent to the notation of the value range subtype:

```
// Value range
template float tr_NearPi := (3.14 .. 3.15);
template integer tr_FitsToOneByte := (0 .. 255);
template integer tr_GreaterThanZero := (1 .. infinity);
```

- Lower and upper boundary of a (`universal`) `charstring` value range template must be a single character string:

```
// Match strings consisting of any number of A, B and C
template charstring tr_PermittedAlphabet := ("A" .. "C");
```

© Ericsson 2002-2011. | LZT 123 7751 Uen, Rev R1H | 2011-04-21 | Page 175

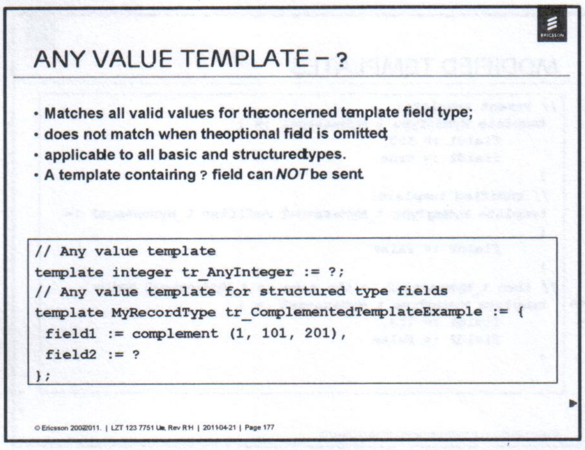


INTERMIXED VALUE LIST AND VALUE RANGE TEMPLATE

- Value list template can be combined with value range template.
- The value range can be specified as an element of a value list:

```
// Intermixed value list and range matching
template integer tr_Intermixed := ((0..127),255);

// Matches strings consisting of any number of capital
// letters or "Hello"
template charstring tr_NotThatGood :=
  (("A".."Z"), "Hello");
```

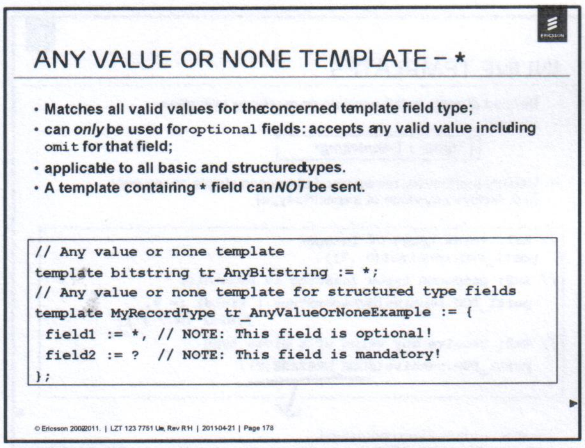
© Ericsson 2002-2011. | LZT 123 7751 Uen, Rev R1H | 2011-04-21 | Page 176



ANY VALUE TEMPLATE – ?

- Matches all valid values for the concerned template field type;
- does not match when the optional field is omitted;
- applicable to all basic and structured types.
- A template containing ? field can *NOT* be sent.

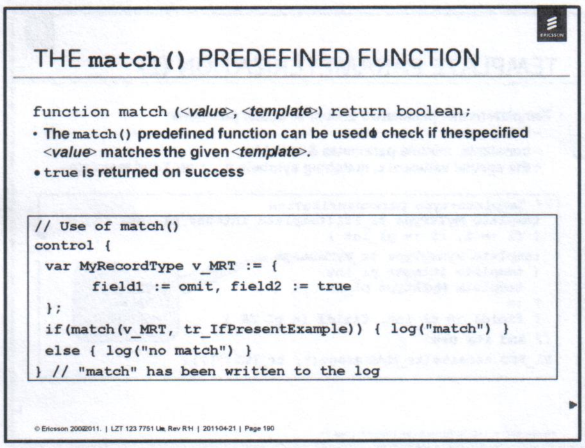
```
// Any value template
template integer tr_AnyInteger := ?;
// Any value template for structured type fields
template MyRecordType tr_ComplementedTemplateExample := {
 field1 := complement (1, 101, 201),
 field2 := ?
};
```



ANY VALUE OR NONE TEMPLATE – *

- Matches all valid values for the concerned template field type;
- can *only* be used for `optional` fields: accepts any valid value including `omit` for that field;
- applicable to all basic and structured types.
- A template containing * field can *NOT* be sent.

```
// Any value or none template
template bitstring tr_AnyBitstring := *;
// Any value or none template for structured type fields
template MyRecordType tr_AnyValueOrNoneExample := {
 field1 := *, // NOTE: This field is optional!
 field2 := ?  // NOTE: This field is mandatory!
};
```

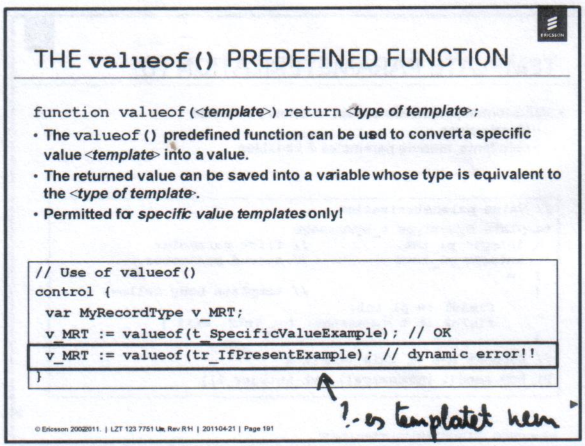


THE `match()` PREDEFINED FUNCTION

`function match (<value>, <template>) return boolean;`

- The `match()` predefined function can be used to check if the specified <*value*> matches the given <*template*>.
- `true` is returned on success

```
// Use of match()
control {
 var MyRecordType v_MRT := {
      field1 := omit, field2 := true
 };
 if(match(v_MRT, tr_IfPresentExample)) { log("match") }
 else { log("no match") }
} // "match" has been written to the log
```



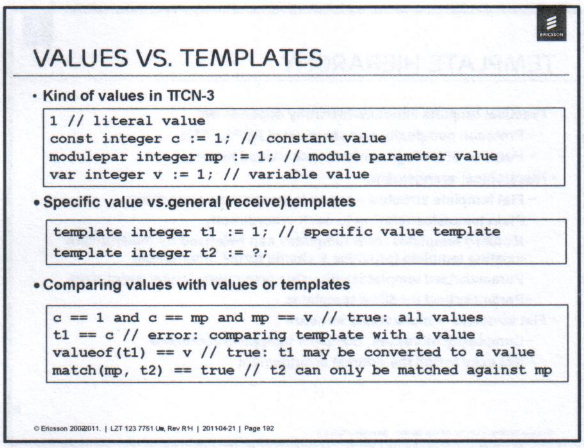
THE `valueof()` PREDEFINED FUNCTION

`function valueof(<template>) return <type of template>;`

- The `valueof()` predefined function can be used to convert a specific value <*template*> into a value.
- The returned value can be saved into a variable whose type is equivalent to the <*type of template*>.
- Permitted for *specific value templates* only!

```
// Use of valueof()
control {
 var MyRecordType v_MRT;
 v_MRT := valueof(t_SpecificValueExample); // OK
 v_MRT := valueof(tr_IfPresentExample); // dynamic error!!
}
```

?-es templatet nem tud konvertálni



VALUES VS. TEMPLATES

- Kind of values in TTCN-3

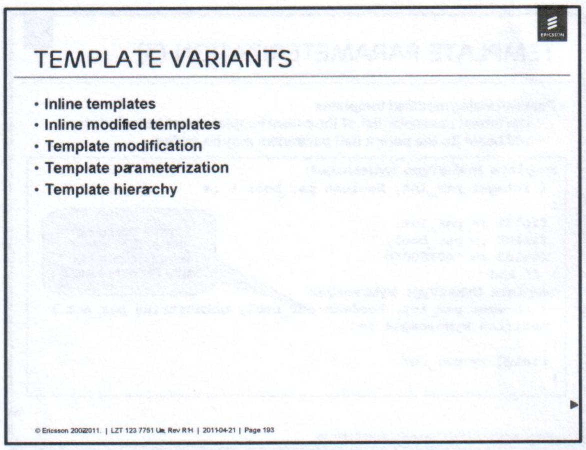
```
1 // literal value
const integer c := 1; // constant value
modulepar integer mp := 1; // module parameter value
var integer v := 1; // variable value
```

- Specific value vs. general (receive) templates

```
template integer t1 := 1; // specific value template
template integer t2 := ?;
```

- Comparing values with values or templates

```
c == 1 and c == mp and mp == v // true: all values
t1 == c // error: comparing template with a value
valueof(t1) == v // true: t1 may be converted to a value
match(mp, t2) == true // t2 can only be matched against mp
```



TEMPLATE VARIANTS

- Inline templates
- Inline modified templates
- Template modification
- Template parameterization
- Template hierarchy

INLINE TEMPLATES

- Defined directly in the sending or receiving operation
- Syntax:

  [<*type*> :] <*matching*>

- Usually ineffective, recommended to use in simple cases only (e.g. receive any value of a specific type)

```
// Ex1: value range of integer
  port1_PCO.receive((0..7));
// Ex2: compound types (nesting is possible)
  port1_PCO.receive(MyRecordType:{ field1 := *,
                                   field2 := ? } );
// Ex3: receive any value of a given type
  port1_PCO.receive(BCCH_MESSAGE:?);
```

típus konkrét

MODIFIED TEMPLATES

```
// Parent template:
 template MyMsgType t_MyMessage1 := {
       field1 := 123,
       field2 := true
 }
 // modified template:
 template MyMsgType t_MyMessage2 modifies t_MyMessage1 :=
 {
       field2 := false
 }
// then t_MyMessage2 is the same as t_MyMessage3 below
 template MyMsgType t_MyMessage3 := {
       field1 := 123,
       field2 := false
 }
```

TEMPLATE PARAMETERIZATION (1)

- *Value* formal parameters accept as actual parameter:
  - literal values
  - constants, module parameters & variables

```
// Value parameterization
template MyMsgType t_MyMessage
  ( integer pl_int,             // first parameter
    integer pl_int2             // second parameter
  ) :=
  {                             // template body follows
        field1 := pl_int,
        field2 := t_MyMessage1 (pl_int2, omit )
  }
// Example use of this template
P1_PCO.send(t_MyMessage(1, vl_integer_2))
```

TEMPLATE PARAMETERIZATION (2)

- *Template* formal parameters accept as actual parameter:
  - literal values
  - constants, module parameters & variables
  - \+ the special value `omit`, matching symbols (`?`, `*` etc.) and templates

```
// Template-type parameterization
 template MyIEType tr_IE1(template integer pl_int) :=
 { f1 := 1, f2 := pl_int }
 template MyMsgType tr_MyMessage
 ( template integer pl_int,
   template MyIEType pl_IE
 ) :=
 { field1 := pl_int, field2 := pl_IE }
// And its use:
P1_PCO.receive(tr_MyMessage(?, tr_IE1(*)));
```

Note the template keyword!

TEMPLATE PARAMETERIZATION (3)

- Parameterizing modified templates
  - The formal parameter list of the parent template must be included;
  - additional (to the parent list) parameters *may* be added

```
template MyMsgType MyMessage4
  ( integer par_int, boolean par_bool ) :=
{
 field1 := par_int,
 field2 := par_bool,
 field3 := '00FF00'O
} // and
template MyMsgType MyMessage2
  ( integer par_int, boolean par_bool, octetstring par_oct )
 modifies MyMessage4 :=
{
 field3 := par_oct
}
```

Formal parameter list of the parent template must be fully repeated here!

TEMPLATE HIERARCHY

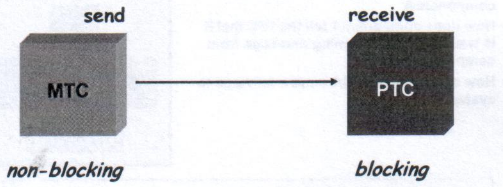
- Practical template structure/hierarchy depends on:
  - Protocol: complexity and structure of ASPs, PDUs
  - Purpose of testing: conformance vs load testing
- Hierarchical arrangement:
  - Flat template structure – separate template for everything
  - Plain templates referring to each other directly
  - Modified templates: new templates can be derived by modifying an existing template (provides a simple form of inheritance)
  - Parameterized templates with value or template formal parameters
  - Parameterized modified templates
- Flat structure → hierarchical structure
  - Complexity increases, number of templates decreases
  - Not easy to find the optimal arrangement

XIII. ABSTRACT COMMUNICATION OPERATIONS

ASYNCHRONOUS COMMUNICATION
SEND AND RECEIVE OPERATIONS
CHECK-RECEIVE AND TRIGGER OPERATIONS
PORT CONTROL OPERATIONS (START, STOP, CLEAR)
VALUE AND SENDER REDIRECTS
SEND TO AND RECEIVE FROM OPERATIONS
SYNCHRONOUS COMMUNICATION

CONTENTS

ASYNCHRONOUS COMMUNICATION



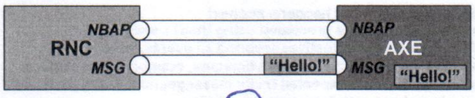
`send` AND `receive` SYNTAX

- *<PortId>*`.send(`*<ValueRef>*`)`
  where *<PortId>* is the name of a `message` port containing an `out` or `inout` definition for the type of *<ValueRef>* and *<ValueRef>* can be:
  - Literal value; constant, variable specific value template (i.e. send template) reference or expression
- *<PortId>*`.receive(`*<TemplateRef>*`)` or *<PortId>*`.receive`
  where *<PortId>* is the name of a `message` port containing an `in` or `inout` definition for the type of *<TemplateRef>* and *<TemplateRef>* can be:
  - Literal value; constant, variable, template (even with matching mechanisms) reference or expression; inline template

SEND AND RECEIVE OPERATIONS

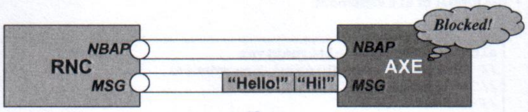
- Send and receive operations can be used on connected ports
- Sending or receiving on a port, which has neither connections nor mappings results in test case error
- The send operation is nonblocking
- The receive operation has blocking semantics
  (except it is used within an alt or interleave statement!)
- Arriving messages stay in the incoming queue of the destination port
- Messages are sent and received in order
- The receive operation examines the 1st message on the port's queue but extracts this *only if* the message matches the receive operation's template

SEND AND RECEIVE EXAMPLES



① `MSG.send("Hello!");`

② `MSG.receive("Hello!");`



③ `MSG.send("Hi!");`

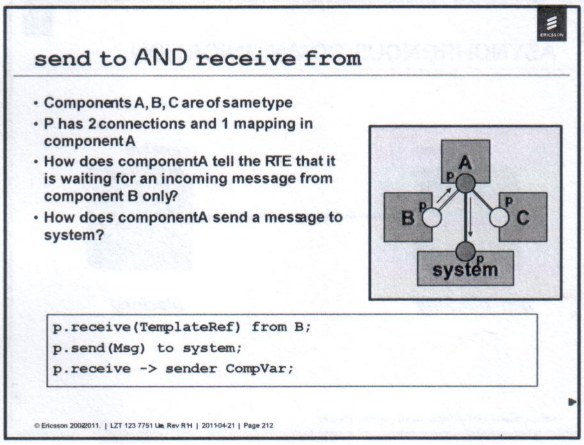
④ `MSG.send("Hello!");`

⑤ `MSG.receive("Hello!");`

VALUE AND SENDER REDIRECT

- Value redirect stores the matched message into a variable
- Sender redirect saves the component reference or address of the matched message's originator
- Works with both `receive` and `trigger`

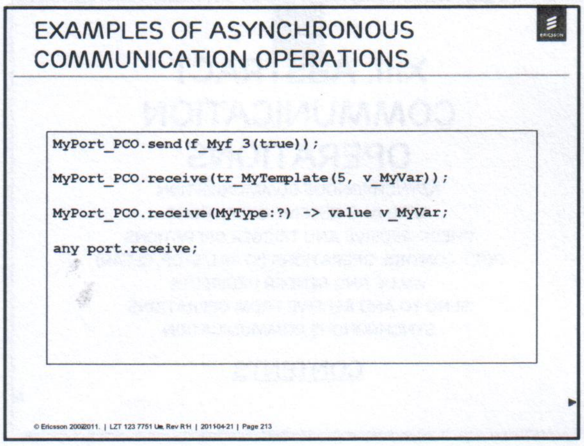
```
template MsgType MsgTemplate := { /* valid content */ }

var MsgType MsgVar;
var CompRef Peer;
// save message matched by MsgTemplate into MsgVar
PortRef.receive(MsgTemplate) -> value MsgVar;
// obtain sender of message in queue w/o removing it
PortRef.check(receive(MsgTemplate) -> sender Peer);
// extract MsgType message and save it with its sender
PortRef.trigger(MsgType:?) -> value MsgVar sender Peer;
```



send to AND receive from

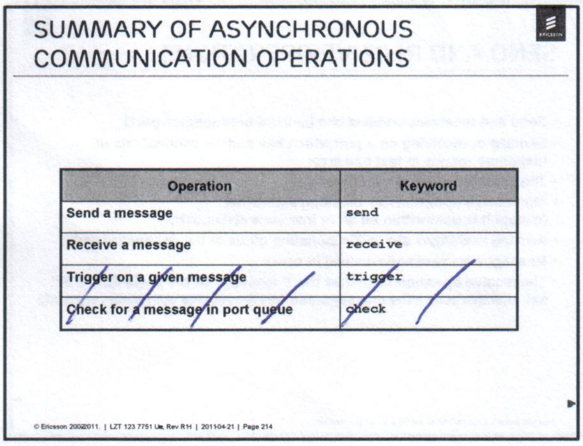
- Components A, B, C are of same type
- P has 2 connections and 1 mapping in component A
- How does component A tell the RTE that it is waiting for an incoming message from component B only?
- How does component A send a message to system?

```
p.receive(TemplateRef) from B;
p.send(Msg) to system;
p.receive -> sender CompVar;
```



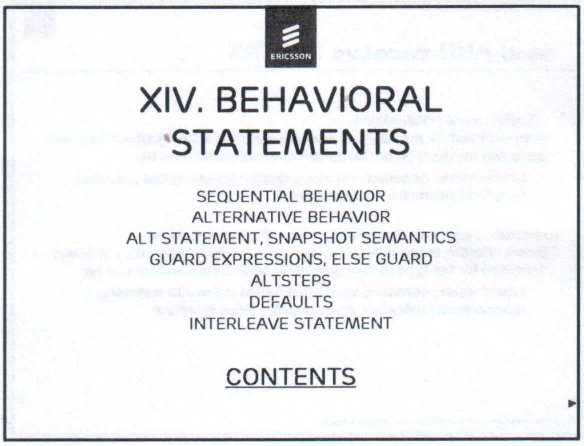
EXAMPLES OF ASYNCHRONOUS COMMUNICATION OPERATIONS

```
MyPort_PCO.send(f_Myf_3(true));

MyPort_PCO.receive(tr_MyTemplate(5, v_MyVar));

MyPort_PCO.receive(MyType:?) -> value v_MyVar;

any port.receive;
```



SUMMARY OF ASYNCHRONOUS COMMUNICATION OPERATIONS

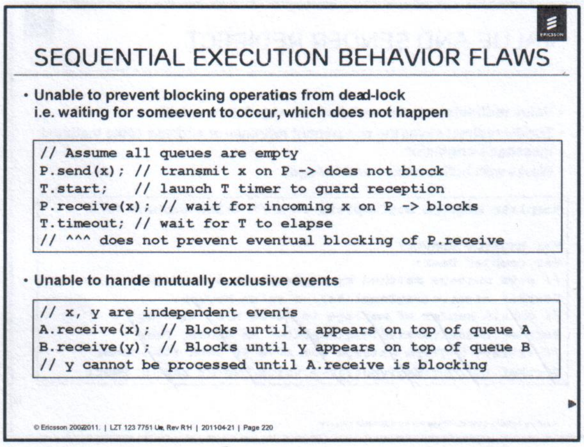
Operation	Keyword
Send a message	`send`
Receive a message	`receive`
~~Trigger on a given message~~	~~`trigger`~~
~~Check for a message in port queue~~	~~`check`~~



XIV. BEHAVIORAL STATEMENTS

SEQUENTIAL BEHAVIOR
ALTERNATIVE BEHAVIOR
ALT STATEMENT, SNAPSHOT SEMANTICS
GUARD EXPRESSIONS, ELSE GUARD
ALTSTEPS
DEFAULTS
INTERLEAVE STATEMENT

CONTENTS



SEQUENTIAL EXECUTION BEHAVIOR FLAWS

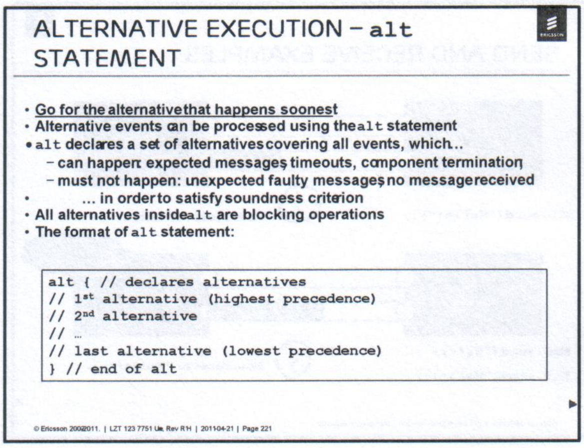
- Unable to prevent blocking operations from dead-lock i.e. waiting for some event to occur, which does not happen

```
// Assume all queues are empty
P.send(x); // transmit x on P -> does not block
T.start;   // launch T timer to guard reception
P.receive(x); // wait for incoming x on P -> blocks
T.timeout; // wait for T to elapse
// ^^^ does not prevent eventual blocking of P.receive
```

- Unable to hande mutually exclusive events

```
// x, y are independent events
A.receive(x); // Blocks until x appears on top of queue A
B.receive(y); // Blocks until y appears on top of queue B
// y cannot be processed until A.receive is blocking
```

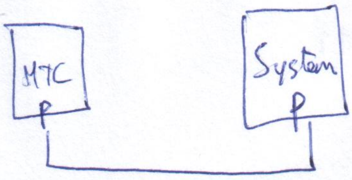
[illegible],
mit Räungen
[illegible]



ALTERNATIVE EXECUTION – alt STATEMENT

- Go for the alternative that happens soonest
- Alternative events can be processed using the `alt` statement
- `alt` declares a set of alternatives covering all events, which...
  - can happen: expected messages, timeouts, component termination
  - must not happen: unexpected faulty messages, no message received
- ... in order to satisfy soundness criterion
- All alternatives inside `alt` are blocking operations
- The format of `alt` statement:

```
alt { // declares alternatives
// 1st alternative (highest precedence)
// 2nd alternative
// …
// last alternative (lowest precedence)
} // end of alt
```



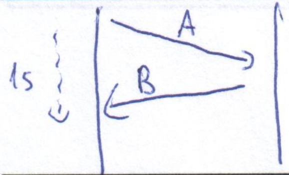
FORMAT OF ALTERNATIVES

- Guard condition enables or disables the alternative:
  - Usually empty: `[]` equivalent to `[true]`
  - Sometimes contains a condition (boolean expression): `[x > 0]`
  - Occasionally the else keyword: `[else]` → else branch
- Blocking operation (event):
  - Any of `receive`, `trigger`, `getcall`, `getreply`, `catch`, `check`, `timeout`, `done` or `killed`
  - `altstep` invocation → `altstep`
  - Empty in conjunction with `[else]` guard only
- Statement block:
  - Describes actions to be executed on event occurrence
  - Optional: can be empty (ie. `{}` or `;`)

`alt` STATEMENT EXECUTION SEMANTICS

- Alternatives are processed according to snapshot semantics
  - Alternatives are evaluated in the same context (snapshot) such that each alternative event has "equal chance"
- `alt` waits for one of the declared events to happen then executes corresponding statement block using sequential behavior!
  - i.e. only a single declared alternative is supposed to happen
- `alt` quits after completing the actions related to the event that happened first
- First alternative has highest priority, last has the least
- When no alternatives apply → programming error (not sound) → dynamic testcase error!
- Question: What's the difference between `if` and `alt`?

→ lefagyasztja a portot, hogy ne fogadjon több üzenetet a feldolgozás végéig



p.send(A);
T.start(1.0);
alt {

ALTERNATIVE EXECUTION BEHAVIOR EXAMPLES

- Take care of unexpected event:

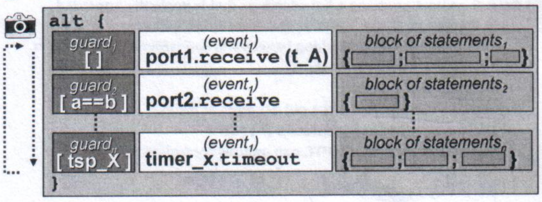
```
alt {
[] P.receive(x)        { /* actions to do */ }
[] any port.receive  { /* handle unexpected event */ }
}
```

- Handle unexpected events but enforce event reception with time constraint:

```
P.send(req)
T.start;
// …
alt {
[] P.receive(resp)   { /* actions to do and exit alt */ }
[] any port.receive  { /* handle event */; repeat }
[] T.timeout          { /* handle timer expiry and exit */ }
}
```

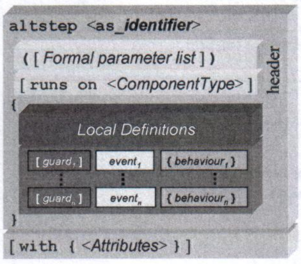
SNAPSHOT SEMANTICS

1. Take a snapshot reflecting current state of test system
2. For all alternatives starting with the 1st:
   a) Evaluate guard: false → 2
   b) Evaluate event: would block → 2
   c) Discard snapshot; execute statement block and exit alt → READY
3. → 1



[] p.receive(B); {setverdict (pass)}
[] p.receive {setverdict(fail);}
[] T.timeout {setverdict(inconc)}}
}

STRUCTURING ALTERNATIVE BEHAVIOR – `altstep`



- Collection of a set of "common" alternatives
- Run-time expansion
- Invoked in-line, inside alt statements or activated as default Run-time parameterization
- Optional runs on clause
- No return value
- Local definitions deprecated

THREE USES OF `altstep`

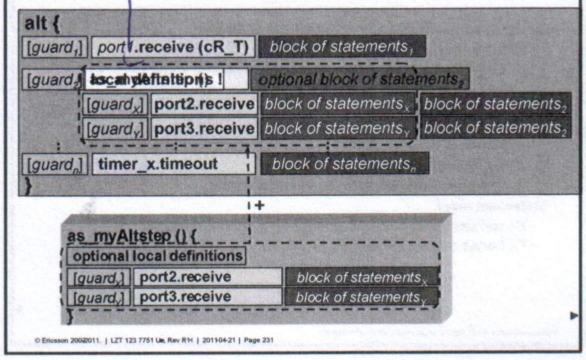
- Direct invocation
  - Expands dynamically to an `alt` statement
- Dynamic invocation from alt statement:
  - Attaches further alternatives to the place of invocation
- Default activation
  - Automatic attachment of activated `altstep` branches to the end of each `alt`/blocking operation

as_myAltstep()

SAMPLE `altstep` DIRECT INVOCATION

```
// Definition in module definitions part
altstep as_MyAltstep(integer pl_i) runs on My_CT {
[] PCO.receive(pl_i) {…}
[] PCO.receive(tr_Msg) {…}
}
// Use of the altstep
testcase tc_101() runs on My_CT {
  as_MyAltstep(4); // Direct altstep invocation…
}
// …as the same effect as
testcase tc_101() runs on My_CT {
  alt {
  [] PCO.receive(4) {…}
  [] PCO.receive(tr_Msg) {…}
  }
}
```

`altstep` USAGE – INVOCATION IN `alt`



alt {

[*guard*$_1$] *port1*.receive (cR_T) *block of statements*$_1$

[*guard*$_2$] as_myAltstep() local definitions ! *optional block of statements*$_2$

[*guard*$_X$] port2.receive *block of statements*$_X$ *block of statements*$_2$

[*guard*$_Y$] port3.receive *block of statements*$_Y$ *block of statements*$_2$

[*guard*$_n$] timer_x.timeout *block of statements*$_n$

}

\+

as_myAltstep () {

optional local definitions

[*guard*$_X$] port2.receive *block of statements*$_X$

[*guard*$_Y$] port3.receive *block of statements*$_Y$

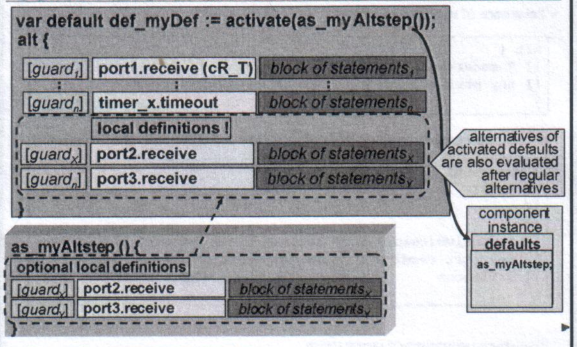
}

ACTIVATION OF `altstep` TO DEFAULTS

- Altsteps can be used as Default operations:
  - `activate`: prepends an `altstep` with given actual parameters to the current default context, returns an unique default reference
  - `deactivate`: removes the given default reference from the context
- Default context contains a list of altsteps that is implicitly appended:
  - At the end of all `alt` statements *except* those with `else` branch
  - After all stand-alone blocking `receive`/`timeout`/`done` … operations
- Defaults are used for handling:
  - Incorrect SUT behavior
  - Periodic messages that are out of scope of testing
  - There are only dynamic defaults in TTCN-3
  - The default context of a PTC can entirely controlled runtime

pl. Receive Ready LAPD-ben

USING ALTSTEPS: ACTIVATED AS DEFAULT



var default def_myDef := activate(as_my Altstep());

alt {

[*guard*$_1$] port1.receive (cR_T) *block of statements*$_1$

[*guard*$_n$] timer_x.timeout *block of statements*$_n$

local definitions !

[*guard*$_X$] port2.receive *block of statements*$_X$

[*guard*$_n$] port3.receive *block of statements*$_Y$

}

as_myAltstep () {

optional local definitions

[*guard*$_X$] port2.receive *block of statements*$_X$

[*guard*$_Y$] port3.receive *block of statements*$_Y$

}

alternatives of activated defaults are also evaluated after regular alternatives

component instance

defaults

as_myAltstep;

STANDALONE RECEIVING STATEMENTS VS `alt`

- Any standalone receiving statement (`receive`, `check`, `getcall`, `getreply`, `done`, `timeout`) behaves identically as if it was embedded into an `alt` statement!

```
MyPort_PCO.receive(tr_MyMessage);
```

- … is equivalent to:

```
alt {
   [] MyPort_PCO.receive(tr_MyMessage) {}
}
```

STANDALONE RECEIVING STATEMENTS VS `default`

- Activated default branches are appended to standalone receiving statements, too!

```
var default d := activate(myAltstep(2));
MyTimer.timeout;
```

- … is equivalent to:

```
alt {
   [] MyTimer.timeout {}
   [] MyPort.receive(MyTemplate(2))
      { MyPort.send(MyAnswer); repeat }
   [] MyPort.receive
      { setverdict(fail) }
}
```

MULTIPLE DEFAULTS

- Default branches areappended in the opposite order oftheir activation to the end of altso that recently activated default branches can "override" elder branches

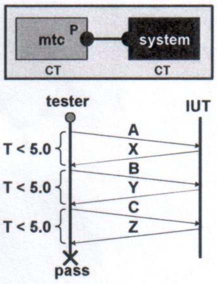
```
altstep as1() runs on CT {
[] any port.receive { setverdict(fail) }
}
altstep as2() runs on CT {
[] PCO.receive(MgmtPDU:?) {}
}
var default d1, d2, d3; // evaluation order
d1 := activate(as1());  // +d1
d2 := activate(as2());  // +d2+d1
d3 := activate(as3());  // +d3+d2+d1
deactivate(d2);         // +d3+d1
d2 := activate(as2());  // +d2+d3+d1
```

XV. SAMPLE TEST CASE IMPLEMENTATION

TEST PURPOSE IN MSC
TEST CONFIGURATION
MULTIPLE IMPLEMENTATIONS

CONTENTS

SAMPLE TEST CASE IMPLEMENTATION



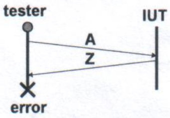
- Single component test configuration
- Test purpose defined by MSC:
  - Simple request-response protocol
  - Answer time less than 5s
  - Result is pass fordisplayed operation, otherwise the verdict shall be fail

FIRST IMPLEMENTATION W/O TIMING CONSTRAINTS

patch összekötése

```
testcase test1() runs on CT {
   map(mtc:P, system:P);
   P.send(a);
   P.receive(x);
   P.send(b);
   P.receive(y);
   P.send(c);
   P.receive(z);
   setverdict(pass);
}
type port PT message {
   out A, B, C;
   in  X, Y, Z;
}
type component CT {
   port PT P;
}
```

- Test case `test1` results error verdict on incorrect IUT behavior → test case is not sound



- Lower case identifiers refer to valid data of appropriate uppercase type!

SOUND IMPLEMENTATION

```
testcase test2() runs on CT {
   map(mtc:P, system:P);
   P.send(a); T.start;
  alt {
  [] P.receive(x) {setverdict(pass)}
  [] P.receive {setverdict(fail)}
  [] T.timeout {setverdict(fail)}
  }
   P.send(b);  T.start;
  alt {
  [] P.receive(y) {setverdict(pass)}
  [] P.receive {setverdict(fail)}
  [] T.timeout {setverdict(fail)}
  }
   P.send(c);   T.start;
  alt {
  [] P.receive(z) {setverdict(pass)}
  [] P.receive {setverdict(fail)}
  [] T.timeout {setverdict(fail)}
  }
}
```

```
type port PT message {
   out A, B, C;
   in  X, Y, Z;
}
type component CT {
   timer T := 5.0;
   port PT P;
}
```

- This test case works fine but its operation is hard to follow between copy/paste lines!

ADVANCED IMPLEMENTATION

```
testcase test3() runs on CT {
   var default d := activate(as());

   map(mtc:P, system:P);
   P.send(a); T.start;
   P.receive(x);
   P.send(b); T.start;
   P.receive(y);
   P.send(c); T.start;
   P.receive(z);

   deactivate(d);
   setverdict(pass);
}
```

```
altstep as() runs on CT {
[] P.receive {setverdict(fail)}
[] T.timeout {setverdict(fail)}
}
```

```
type port PT message {
   out A, B, C;
   in  X, Y, Z;
}

type component CT {
   timer T := 5.0;
   port PT P;
}
```

- This example demonstrates one specific use ofdefaults
- Compact solution employing defaults for handling incorrect IUT behavior