ERICSSON

II. INTRODUCTION TO TTCN-3

HISTORY OF TTCN

TTCN-2 TO TTCN-3 MIGRATION

TTTCN-3 CAPABILITIES, APPLICATION AREAS

PRESENTATION FORMATS

STANDARD DOCUMENTS

CONTENTS

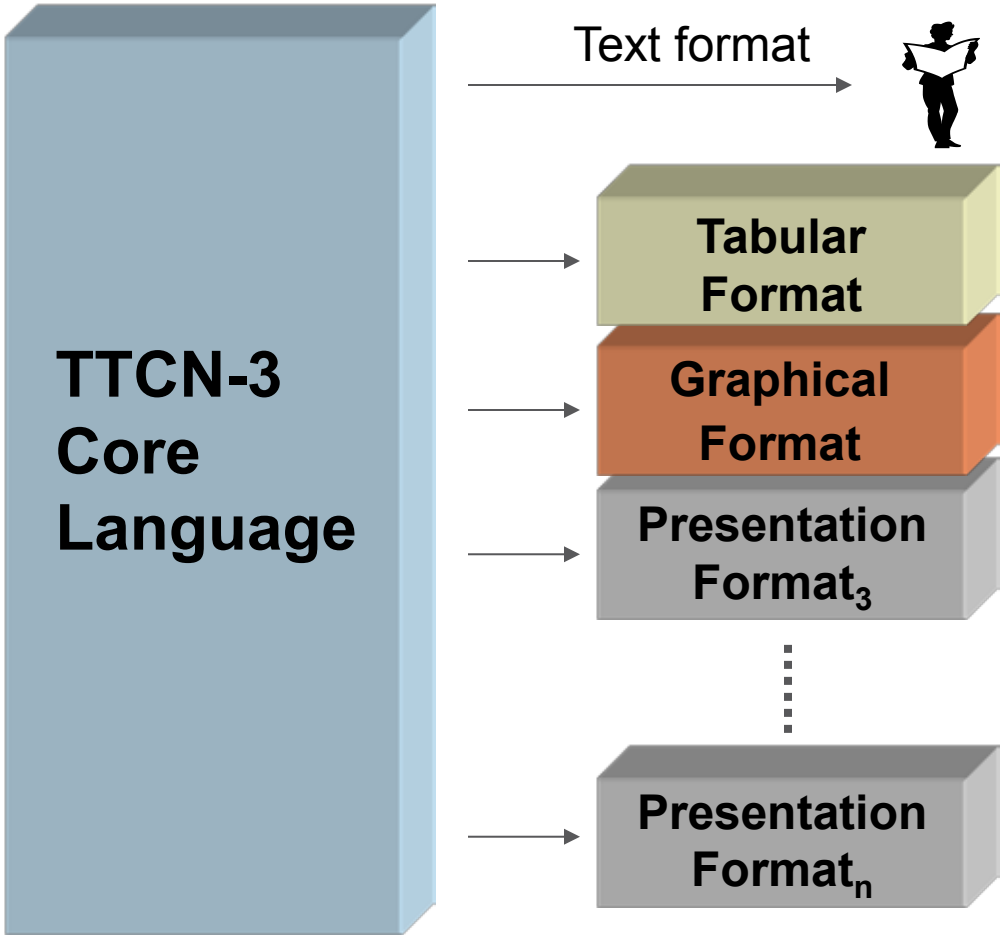
HISTORY OF TTCN

- 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
- Supports automatic test execution (e.g. SCS)
- Requires expensive tools (e.g. ITEX for editing)

TTCN-3 STANDARD DOCUMENTS

- Multi-part ETSI Standard
  - ES 201 873-1: TTCN-3 Core Language
  - 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 201 873-8: TTCN-3: The IDL to TTCN-3 Mapping
  - ES 201 873-9: Using XML schema with TTCN-3
  - ES 201 873-10: Documentation Comment Specification
- Available for download at: **http://www.ttcn-3.org/**

TTCN-3 PRESENTATION FORMATS

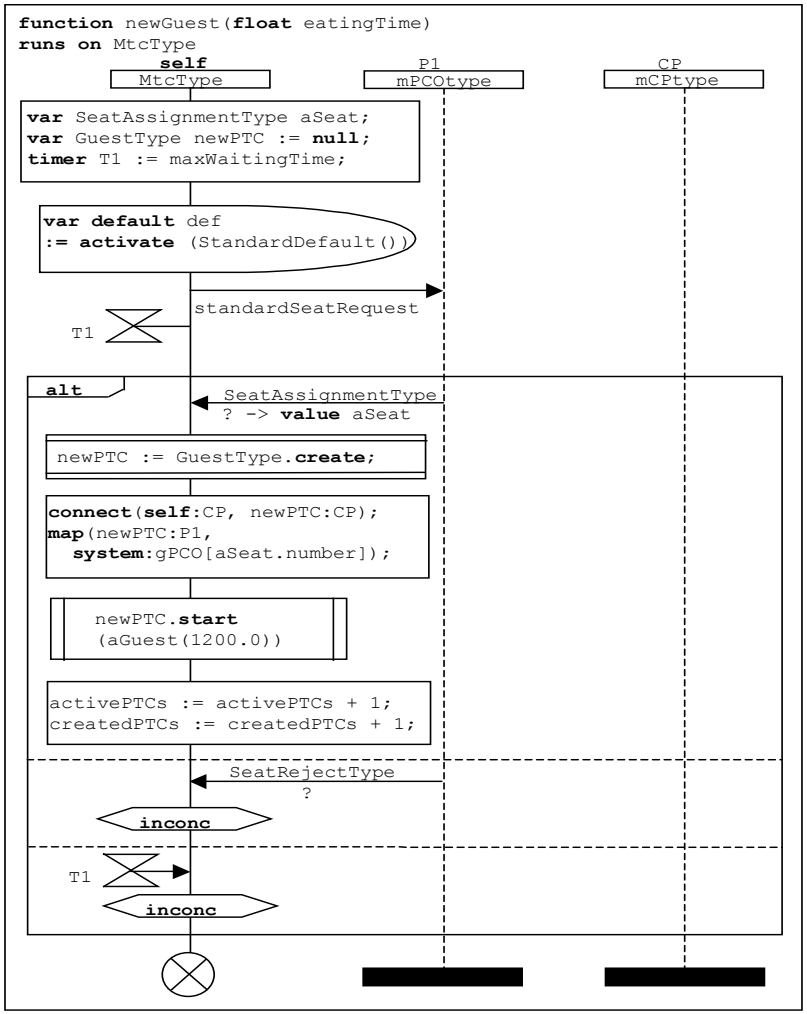


- Core Language
  - is the textual common interchange format between applications
  - can be edited as text or accessed via GUIs offered by various presentation formats
- Tabular Presentation Format (TFT)
  - Table proformas for language elements
  - conformance testing
- Graphical Presentation Format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

EXAMPLE IN CORE LANGUAGE

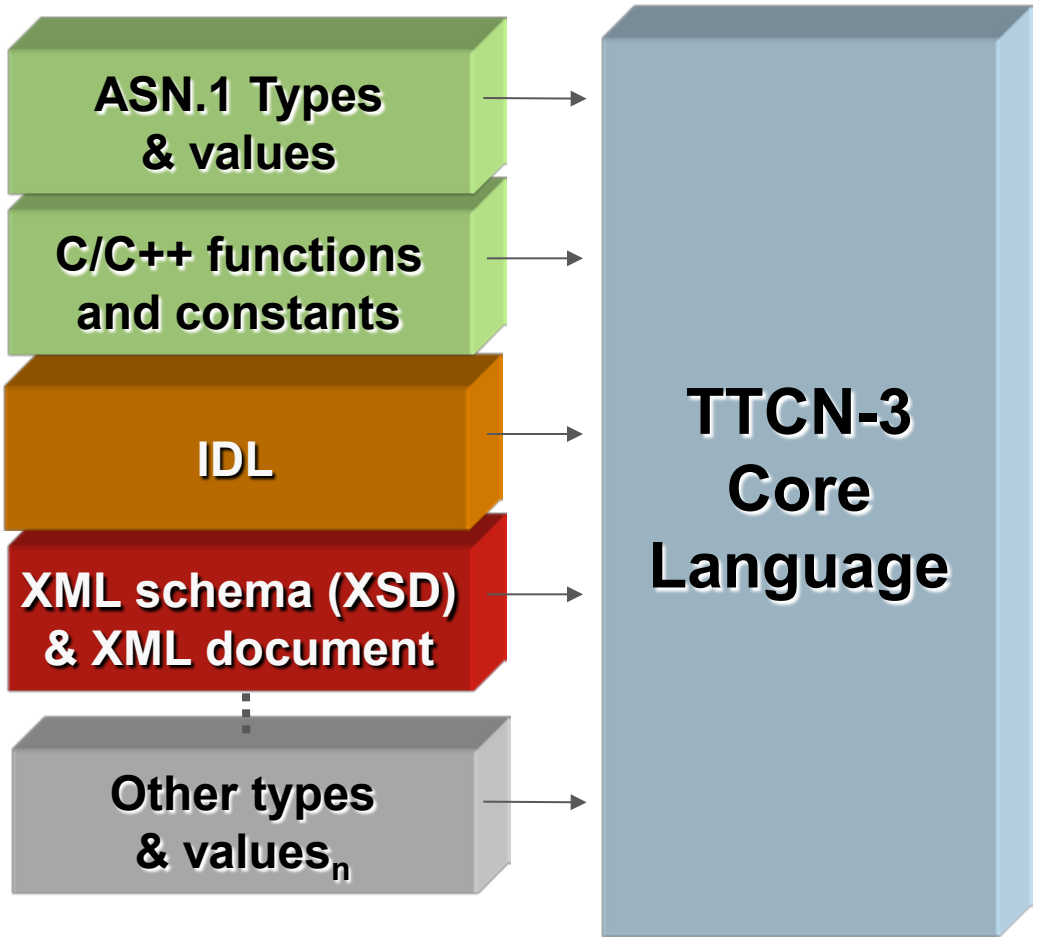
```
function PO49901(integer FL) runs on MyMTC
{
        L0.send(A_RL3(FL, CREF1, 16));
        TAC.start;
        alt {
        [] L0.receive(A_RC1((FL+1) mod 2)) {
                TAC.stop;
                setverdict(pass);
           }
        [] TAC.timeout {
                setverdict(inconc);
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

EXAMPLE IN 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



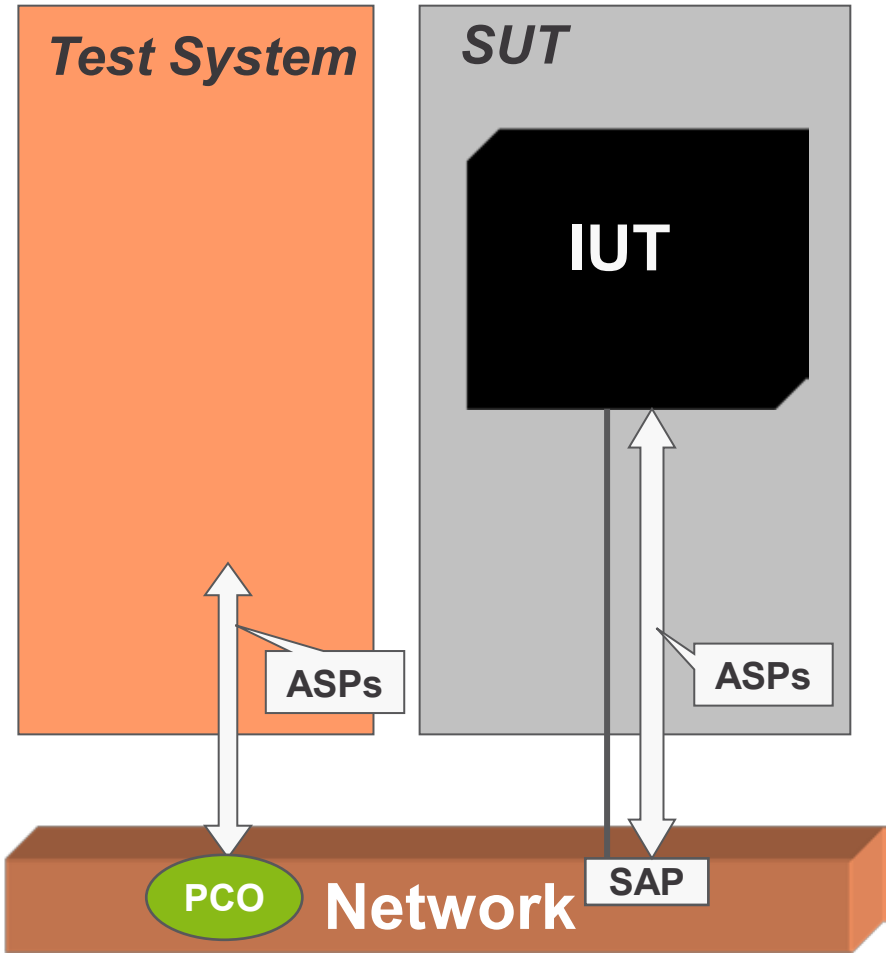
- TTCN can be integrated with other 'type and value' systems
- Fully harmonized with ASN.1 (version 2002 except XML specific ASN.1 features)
- C/C++ functions and constants can be used
- Harmonization possible with other type and value systems (possibly from proprietary languages) when required

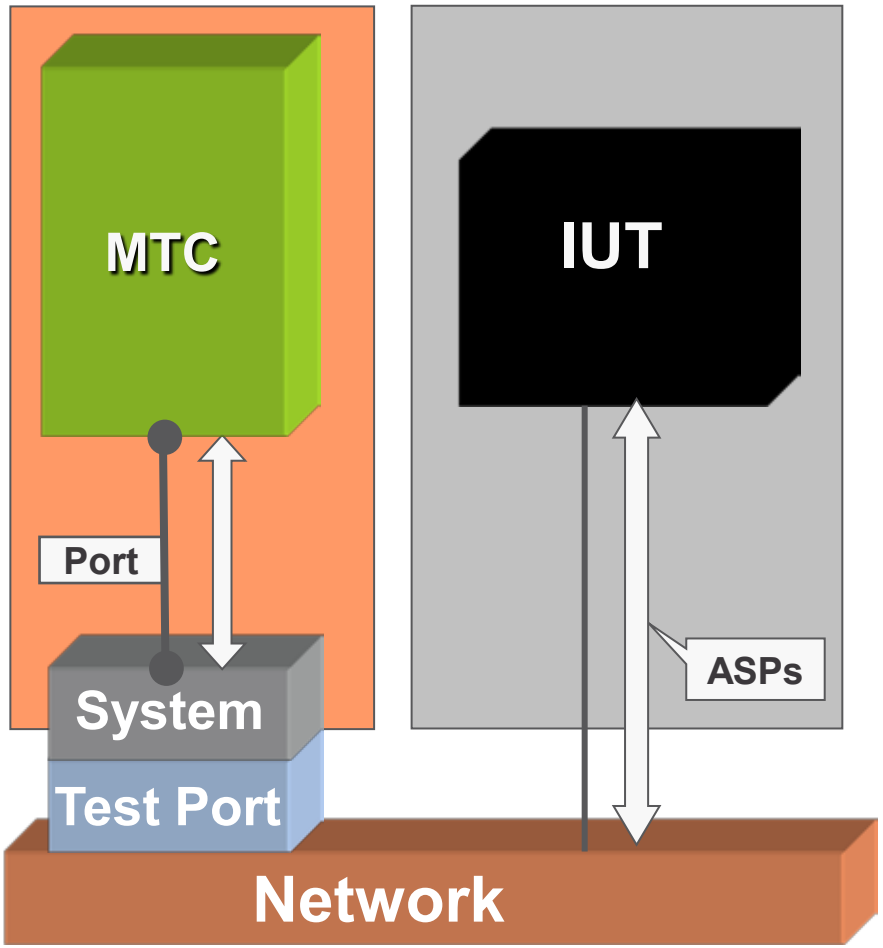
TTCN-3 IS A PROCEDURAL LANGUAGE
(LIKE MOST OF THE PROGRAMMING LANGUAGES)

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

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

TEST ARRANGEMENT AND ITS TTCN-3 MODEL





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

```
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

SIMPLE BASIC TYPES

- **integer**
  - Represents infinite set of integer values
  - Valid **integer** values: `5, -19, 0`
- **float**
  - Represents 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 8bit-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**
  - 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
  - In between double quotes
    - Double quote inside a **charstring** is represented by a pair of double quotes
  - Valid **charstring** values: `""`, `"abc"`, `""hello!""`
  - Invalid **charstring** values: `"Linköping"`, `"Café"`
- **universal charstring**
  - UCS-4 coded representation of ISO/IEC 10646 characters: `"ǝξ"`
  - May also contain characters referenced by quadruples, e.g.:
  - `char(0, 0, 40, 48)`

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
}
```

SUB-TYPING: VALUE RANGE RESTRICTIONS

- Value-range subtype definition is applicable only for **integer**, **charstring**, **universal charstring** and **float** types
  - for charstrings: restricts the permitted characters!

```
type integer    MyIntegerRange    (1 .. 100);
type integer    MyIntegerRange8   (0 .. infinity);
type charstring MyCharacterRange ("k" .. "w");
```

- **-infinity**/**infinity** keywords can be used instead of a value indicating that there is no lower/upper boundary

- Note that **-infinity**/**infinity** are *NOT values* and cannot be used in expressions, thus *the following example is invalid*:

```
var integer v_invalid := infinity; // error!!!
```

SUB-TYPING: VALUE LIST RESTRICTIONS

- Value list restriction subtype is applicable for all basic type as well as in fields of structured types:

```
type charstring SideType ("left", "right");
type integer MyIntegerList (1, 2, 3, 4);
type record MyRecordList {
  charstring userid ("ethxyz", "eraxyz"),
  charstring passwd ("xxxxxx", "yyyyyy")
};
```

- For **`integer`** and **`float`** types it is permitted to mix value list and value range subtypes:

```
type integer MyIntegerListAndRange (1..5, 7, 9);
```

SUB-TYPING: PATTERNS

- **charstring** and **universal charstring** types can be restricted with patterns (→ charstring value patterns)
- All values denoted by the pattern shall be a true subset of the type being sub-typed

```
// all permitted values have prefix abc and postfix xyz
   type charstring MyString (pattern "abc*xyz");
// a character preceded by abc and followed by xyz
   type charstring MyString2 (pattern "abc?xyz");
//all permitted values are terminated by CR/LF
   type charstring MyString3 (pattern "*\r\n")

type MyString MyString3 (pattern "d*xyz");
/* causes an error because MyString does not contain a
   value starting with character 'd'*/
```

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

XII. DATA TEMPLATES

INTRODUCTION TO TEMPLATES
TEMPLATE MATCHING MECHANISMS
INLINE TEMPLATES
MODIFIED TEMPLATES
PARAMETERIZED TEMPLATES
PARAMETERIZED MODIFIED TEMPLATES
TEMPLATE HIERARCHY

CONTENTS

TEMPLATE CONCEPT

Message to send

TYPE: REQUEST
ID: 23
FROM: 231.23.45.4
TO: 232.22.22.22
FIELD1: 1234
FIELD2: ”Hello”

Acceptable answer

TYPE: RESPONSE
ID: SAME as in REQ.
FROM: 230.x – 235.x
TO: 231.23.45.4
FIELD1: 800-900
FIELD2: Do not care

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
 }
```

- Syntax is similar to variable definition
  - but not only concrete values, but also matching mechanisms may stand at the right side of the assignment

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
  - Determines the permitted characters

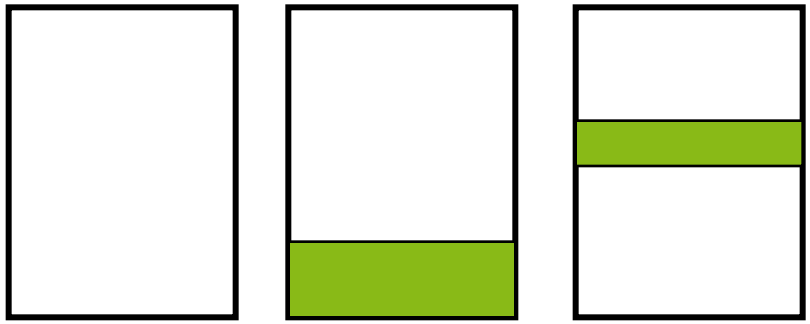
```
// Match strings consisting of any number of A, B and C
template charstring tr_PermittedAlphabet := ("A" .. "C");
```

MATCHING INSIDE VALUES

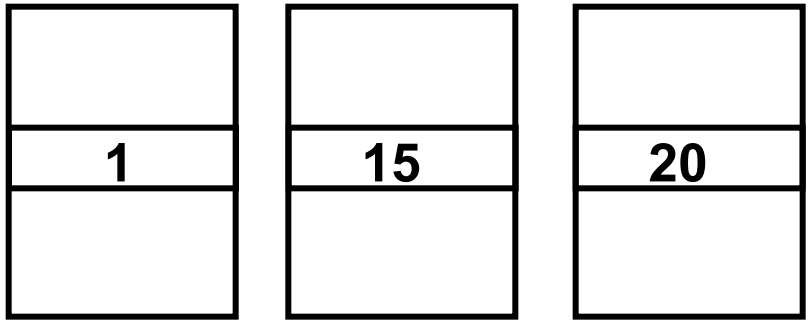
- `?` matches an arbitrary element,
  `*` matches any number of consecutive elements;
- applicable inside **`bitstring`**, **`hexstring`**, **`octetstring`**, **`record of`**, **`set of`** types and arrays;
- not allowed for **`charstring`** and **`universal charstring`**:
  - **`pattern`** shall be used instead! (see next slide)

```
// Using any element matching inside a bitstring value
// Last 2 bits can be '0' or '1'
template bitstring tr_AnyBSValue := '101101??'B;

// Any elements or none in record of
// '2' and '3' must appear somewhere inside in that order
template ROI tr_TwoThree := { *, 2, 3, * };
```

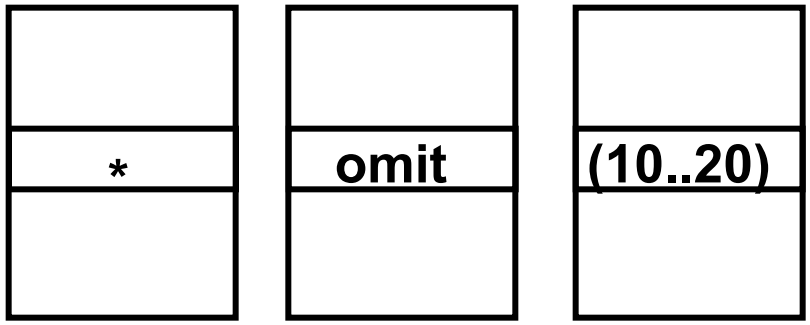
TEMPLATE HIERARCHY – TYPICAL SITUATIONS



modified template



parametrized template



template parameter

ERICSSON

XIV. BEHAVIORAL STATEMENTS

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

CONTENTS

SEQUENTIAL EXECUTION BEHAVIOR FEATURES

- Program statements are executed in order
- Blocking statements block the execution of the component
  - all receiving communication operations, **`timeout`**, **`done`**, **`killed`**
- Occurrence of unexpected event may cause infinite blocking

```
// x must be the first on queue P, y the second
P.receive(x); // Blocks until x appears on top of queue P
P.receive(y); // Blocks until y appears on top of queue P
// When y arrives first then P.receive(x) blocks -> error
```

PROBLEMS OF SEQUENTIAL EXECUTION

- 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(x)
```

- Unable to handle 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(x) is blocking
```

SOLUTION: ALTERNATIVE EXECUTION – ALT STATEMENT

- Go for the alternative that happens earliest!
- 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
```

ALTERNATIVE EXECUTION BEHAVIOR EXAMPLES

- Take care of unexpected event and timeout:

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

NESTED ALT STATEMENT

```
alt {
[] P.receive(1)
   {
     P.send(2)
     alt { // embedded alt
     [] P.receive(3) { P.send(4) }
     [] any port.receive { setverdict(fail); }
     [] any timer.timeout { setverdict(inconc) }
     } // end of embedded alt
   }
[] any port.receive { setverdict(fail); }
[] any timer.timeout { setverdict(inconc) }
}
```

THE REPEAT STATEMENT

- Takes a new snapshot and re-evaluates the **`alt`** statement
- Can appear as last statement in statement blocks of statements

- Can be used for example to filter “keep alive” messages :

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

