ERICSSON

# TTCN-3 COURSE

*PRESENTATION MATERIAL*

ERICSSON
TEST SOLUTIONS AND COMPETENCE CENTER

# Copyright © Ericsson AB 2002-2013 - All rights reserved.

The content of this document is subject to revision without notice. Changes may periodically be made to the information and will be incorporated in new editions of this document. Telefonaktiebolaget LM Ericsson may make improvements or changes in products and programs described in this publication at anytime without notice.

Information on this document is provided "as is" without any warranty of any kind, either express or implied, including but not limited to, the implied warranties of merchantability, fitness for a particular purpose, or non-infringement. The information herein may include technical inaccuracies or typographical errors. Ericsson shall have no liability for any error or damage of any kind resulting from the use of this document.

# CONTENTS

ERICSSON

# I. PROTOCOLS AND TESTING

WHAT IS A “PROTOCOL”?
DEFINITIONS
PROTOCOL VERIFICATION, TESTING AND VALIDATION

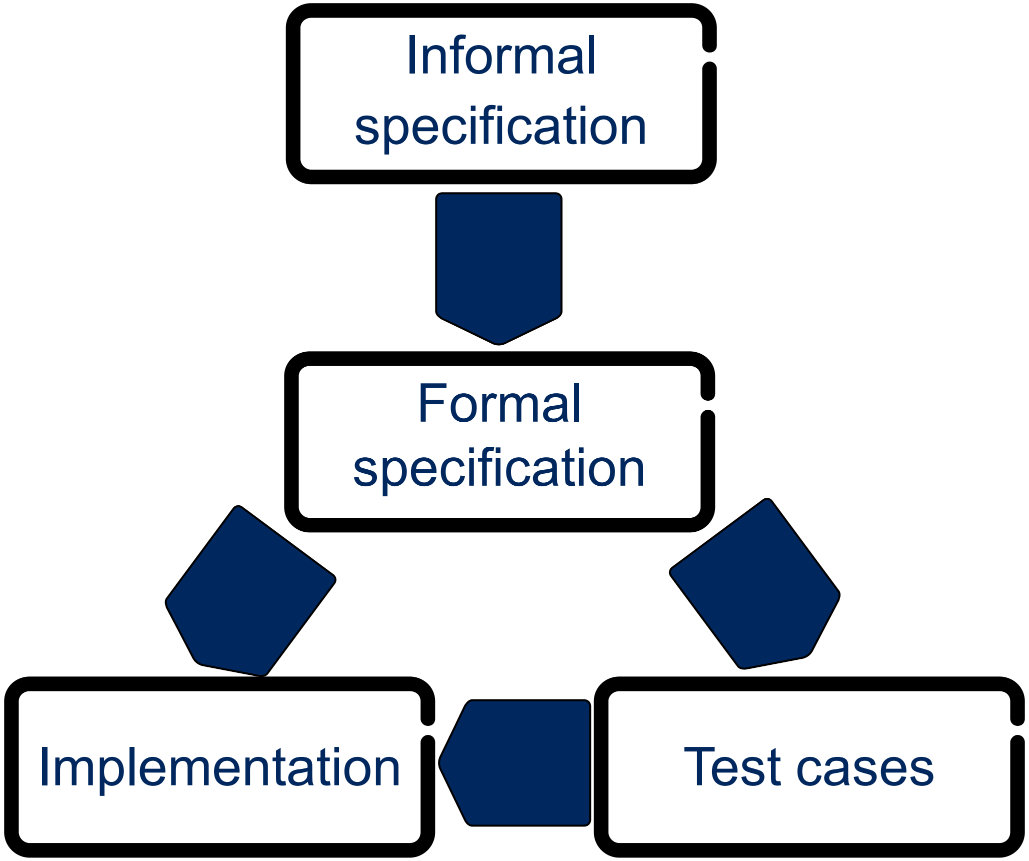
CONTENTS

# PROTOCOL

# COMMUNICATIONS PROTOCOLS

- **Protocol is a set of rules that controls the communication**
  - **syntactical rules (static part):**
    - **define *format (layout)* of messages**
  - **semantical rules (dynamic part):**
    - **describe *behavior* (how messages are exchanged) and *meaning* of messages**

# PROTOCOL TECHNOLOGY

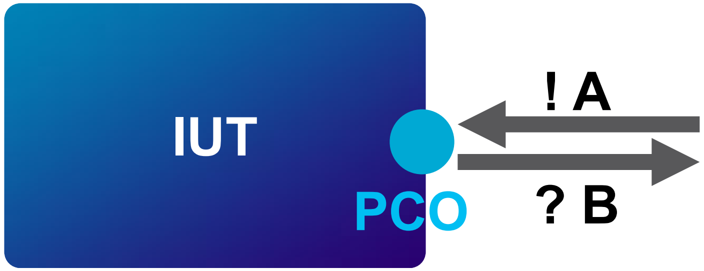


- Ambiguous
- Not complete

- ASN.1, TTCN-3, ...
- UML, SDL, MSC, ...
- Verification, validation

- Conformance tests

# TESTING



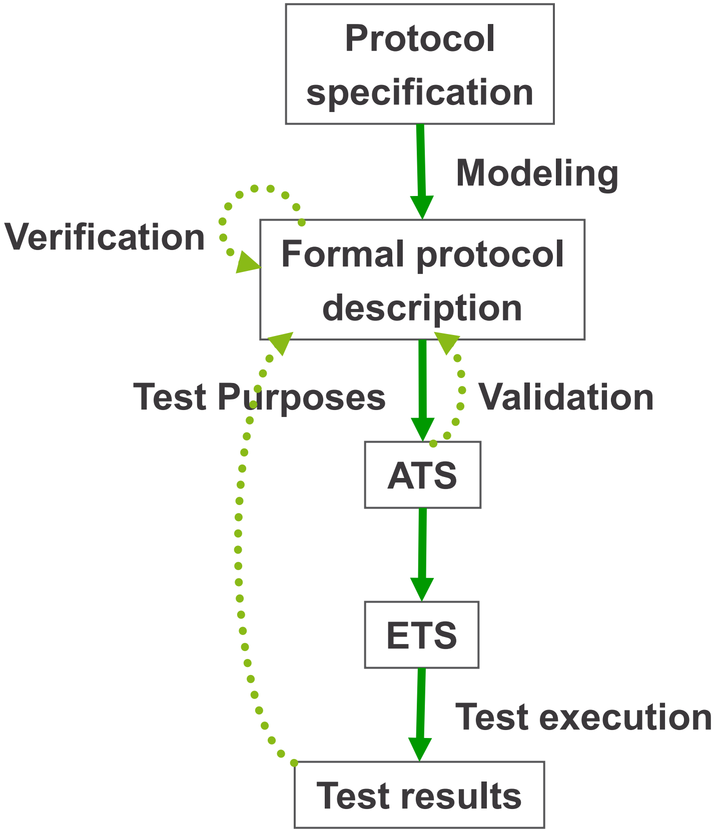
**Verdict:**

**pass,**
**fail,**
**inconclusive**

- Black box testing
  - Implementation/System Under Test
  - Point of Control and Observation
- Not possible to test all the situations
  - Test Purposes

# FORMAL TECHNIQUES IN CONFORMANCE ASSESSMENT

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



# TEST TYPES

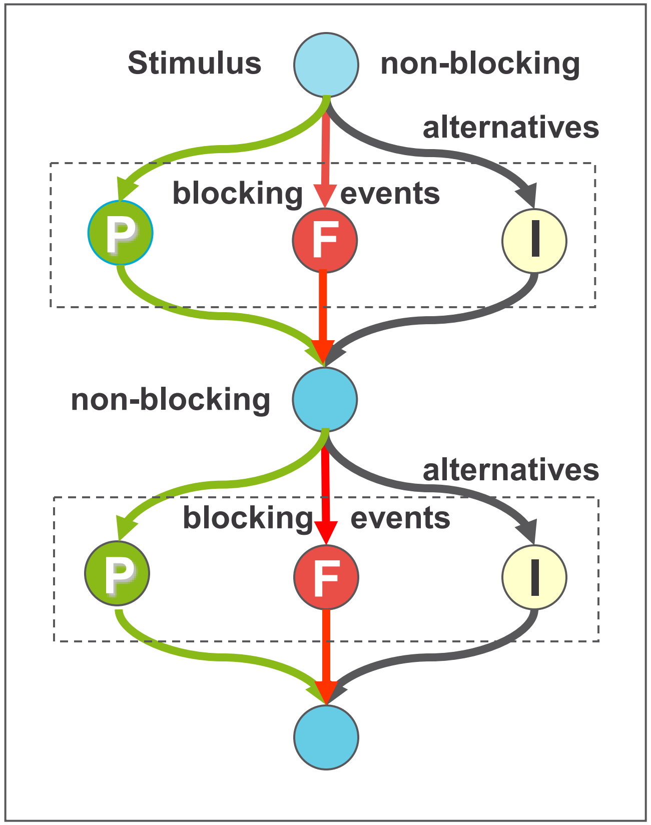
- Conformance testing
  - Function tests
  - System tests
  - Regression tests

- Interoperability testing

- Performance (Load) testing

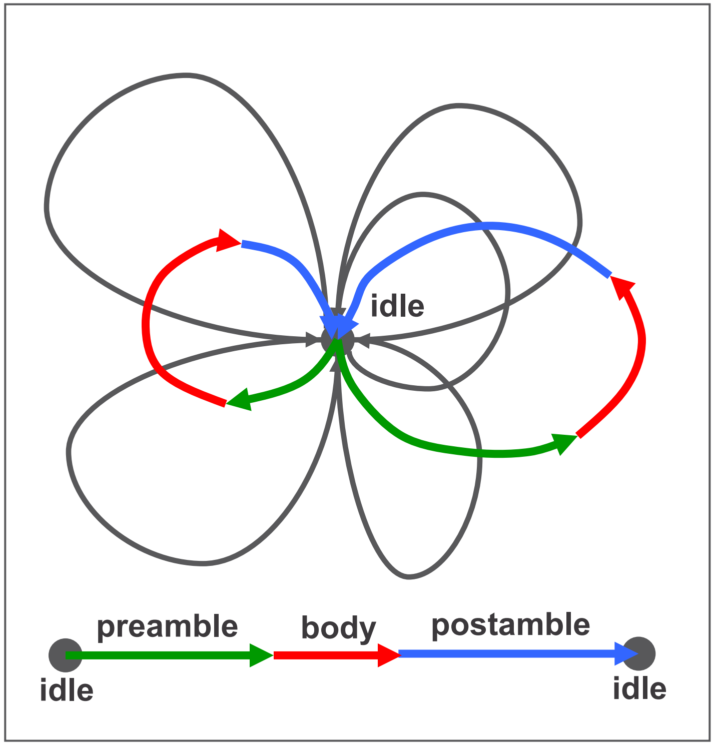
# TEST CASES IN BLACK-BOX TEST

- Implementation of Test Purpose (TP)
  - 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)



# INDEPENDENCE AND STRUCTURE OF ABSTRACT TEST CASES

- *Abstract test cases* shall 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 drives 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

# 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 against the Implementation Under Test (IUT)
  - with MoT
- Analysis of test results
  - verdicts, logs (validation)

# ABSTRACT TEST SUITE DESIGN

- 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)

- 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
  - Model Based Testing (MBT)

# TEST EXECUTION

- Realize *Executable Test Suite* (ETS) from Abstract Test Suite (ATS) using the chosen *Means of Testing (MoT)*
  - MoT = TITAN
  - ATS->ETS = build project
- Execute the *ETS* on the *test system* against the IUT
  - execute in TITAN
- Observe the verdict of executed test cases
  - pass, fail, inconclusive (none, error)

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)

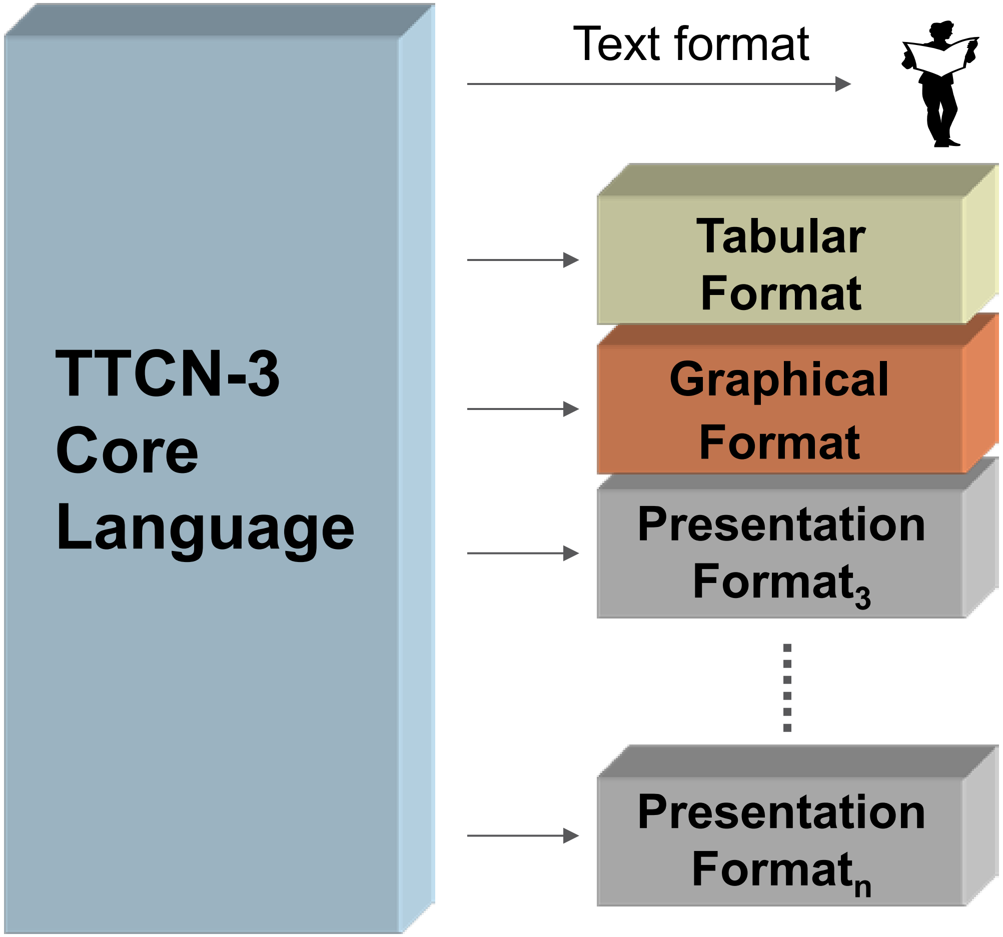
# TTCN-2 TO TTCN-3 MIGRATION

- 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 backward compatible
- Name changed: Testing and Test Control Notation

# 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 TABULAR FORMAT

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

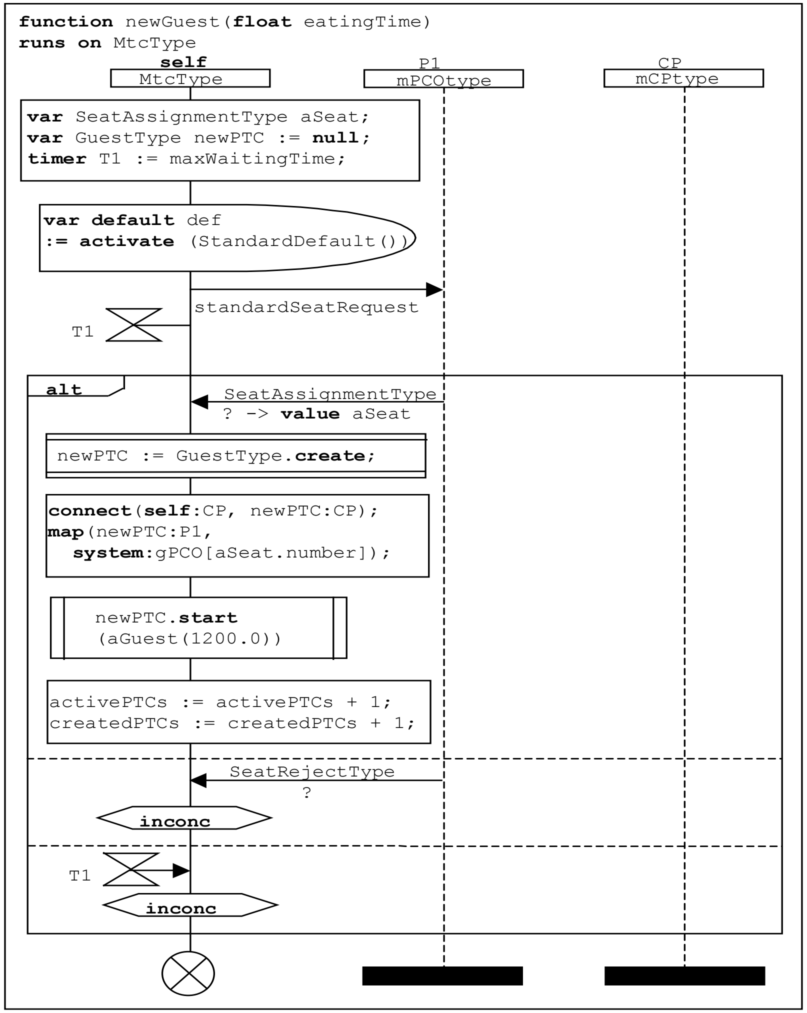
| Local Def Name | Type | Initial Value | Comments |
|---|---|---|---|
| MyLocalVar | boolean | false | local variable |
| MyLocalConst | const float | 60 | local constant |
| MyLocalTimer | timer | 15 * MyLocalConst | local timer |

**Behaviour**

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

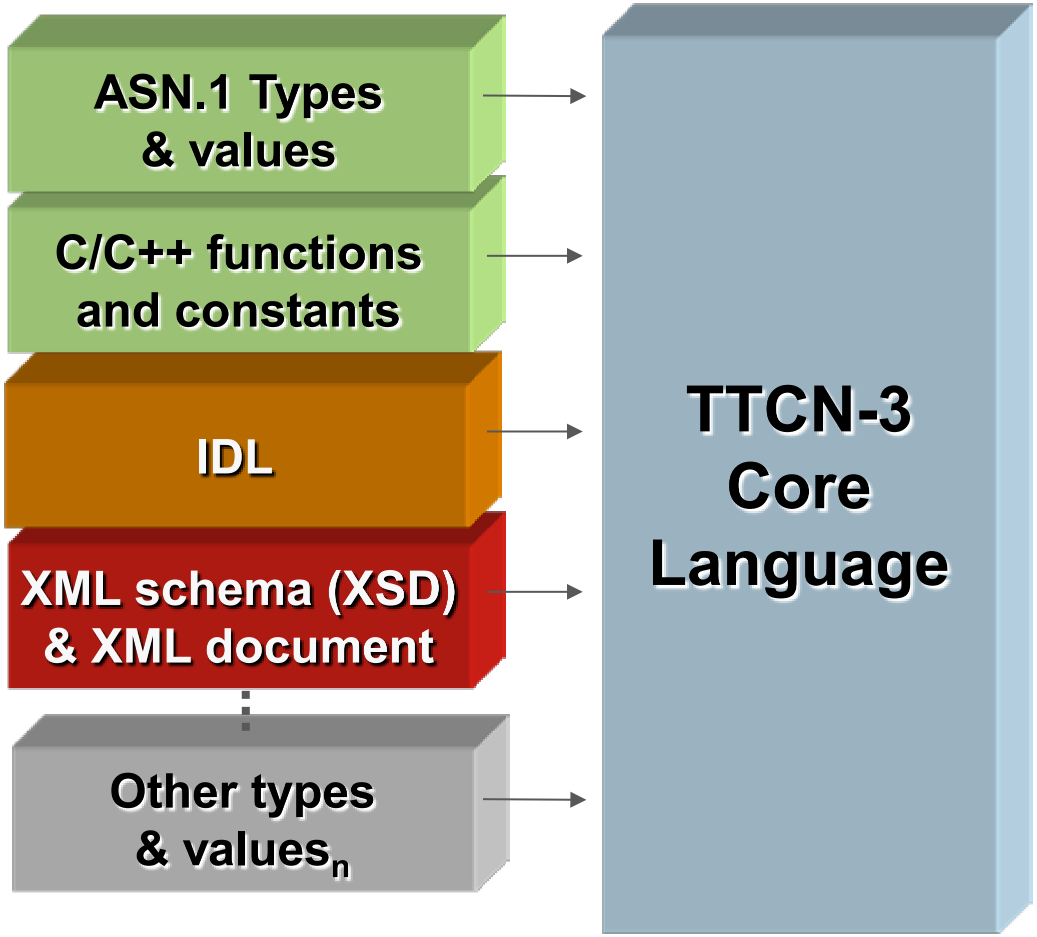
| | |
|---|---|
| **Detailed Comments** | detailed comments |

# EXAMPLE IN GFT FORMAT


function newGuest(float eatingTime)
runs on MtcType
self
MtcType
P1
mPCOtype
CP
mCPtype
var SeatAssignmentType aSeat;
var GuestType newPTC := null;
timer T1 := maxWaitingTime;
var default def
:= activate (StandardDefault())
standardSeatRequest
T1
alt
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;
createdPTCs := createdPTCs + 1;
SeatRejectType
?
inconc
T1
inconc

```
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

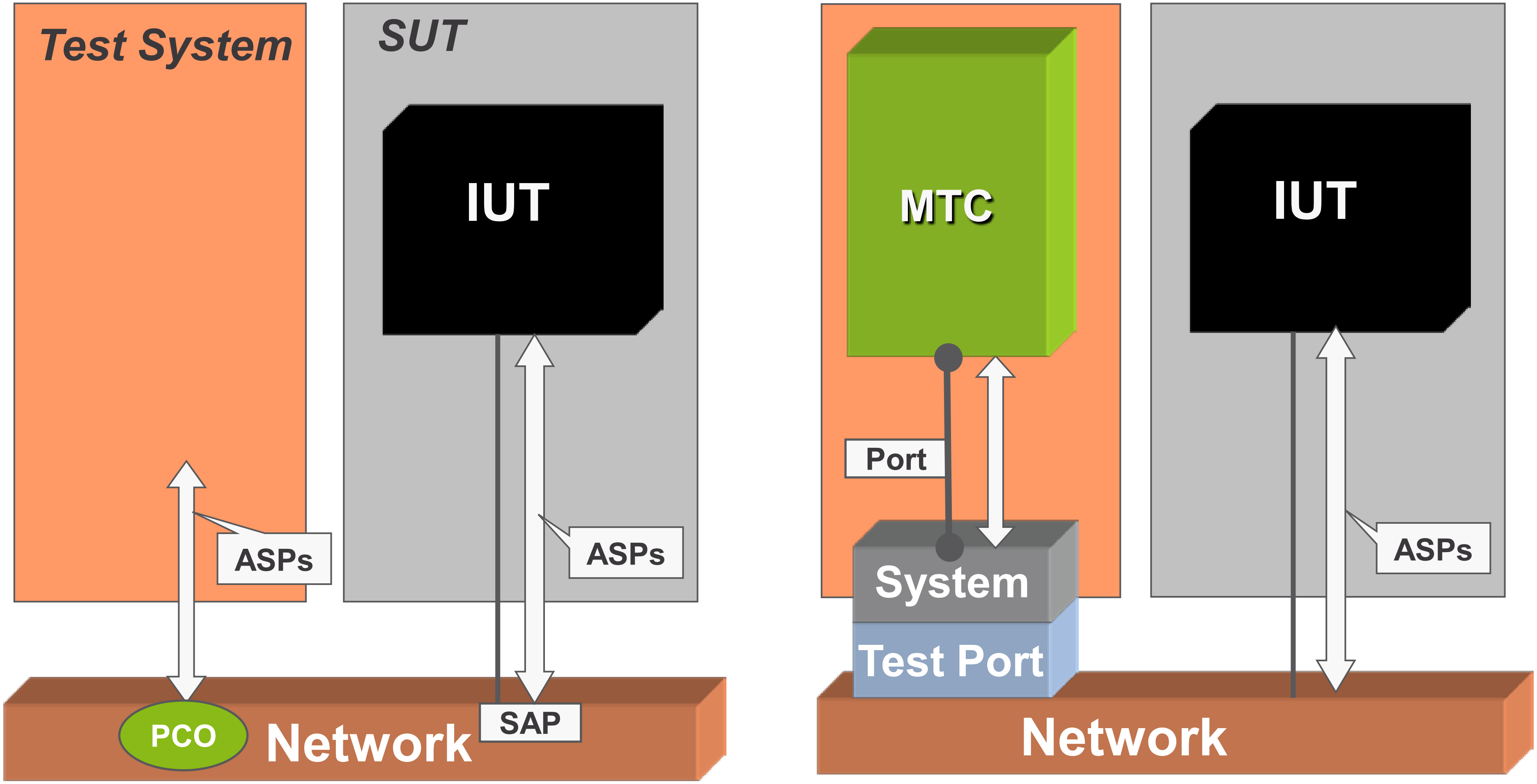
# TTCN-3 IS A PROCEDURAL LANGUAGE

## (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



# III. TTCN-3 MODULE STRUCTURE

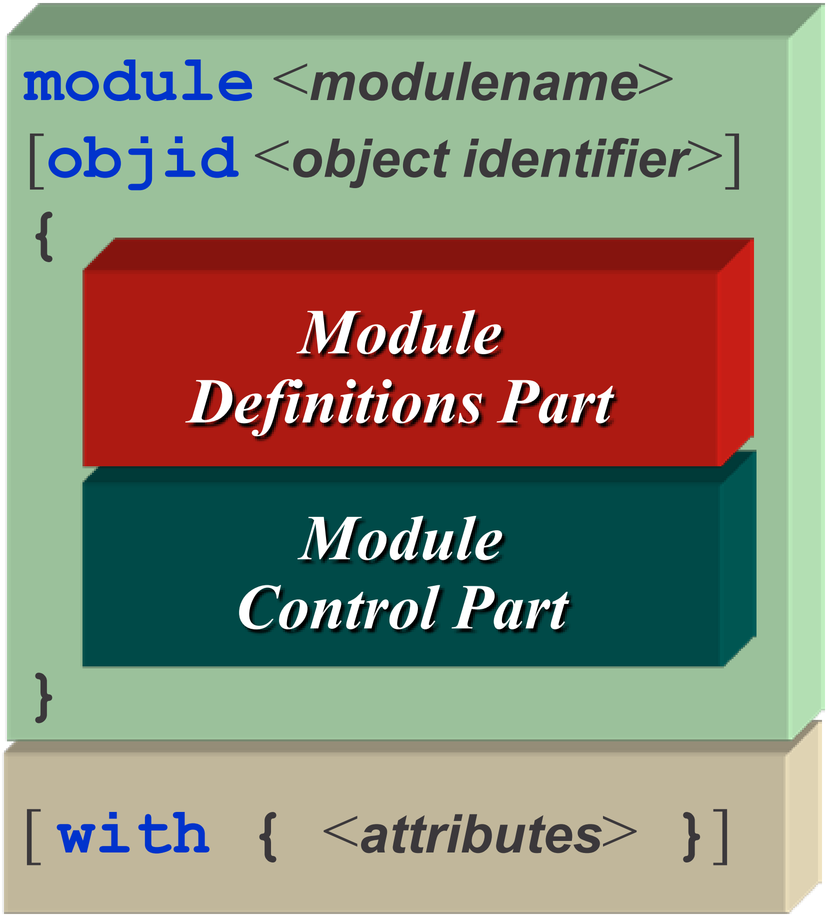
SYNTACTICAL RULES
MODULE
MODULE DEFINITIONS PART
MODULE CONTROL PART
GENERAL SYNTAX RULES
MODULE PARAMETERS

CONTENTS

# TTCN-3 SYNTACTICAL RULES AND NOTATIONAL CONVENTIONS

- Keywords always use lower case 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 comment */`
  - 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!") }`
- In this material:
  - Red letters or red frames : erroneous examples

# TTCN-3 MODULES



module <*modulename*>
[objid <*object identifier*>]
{

***Module Definitions Part***

***Module Control Part***

}

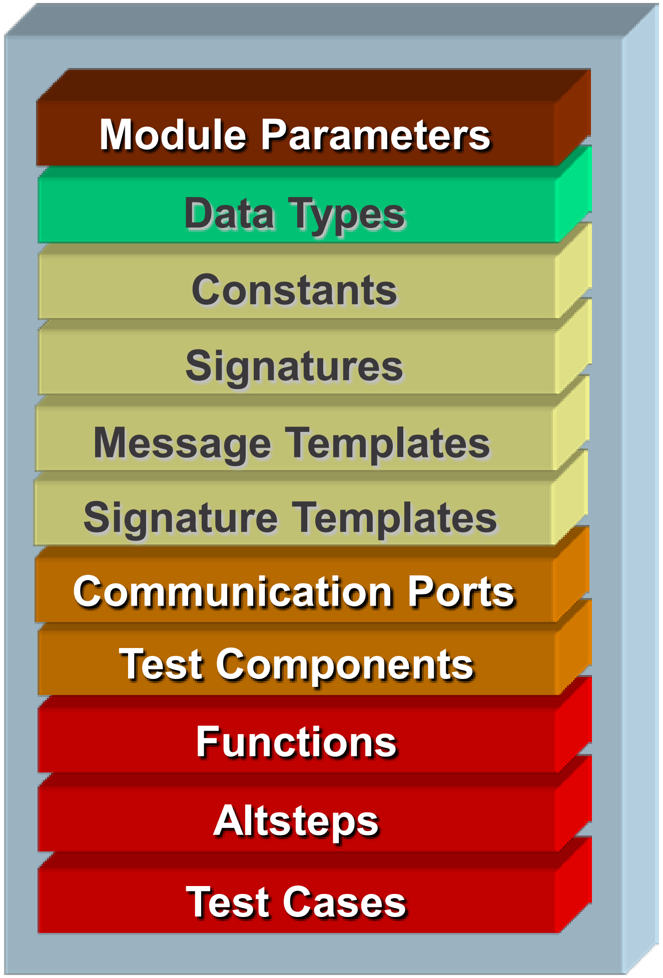
[ with { <*attributes*> } ]

- 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

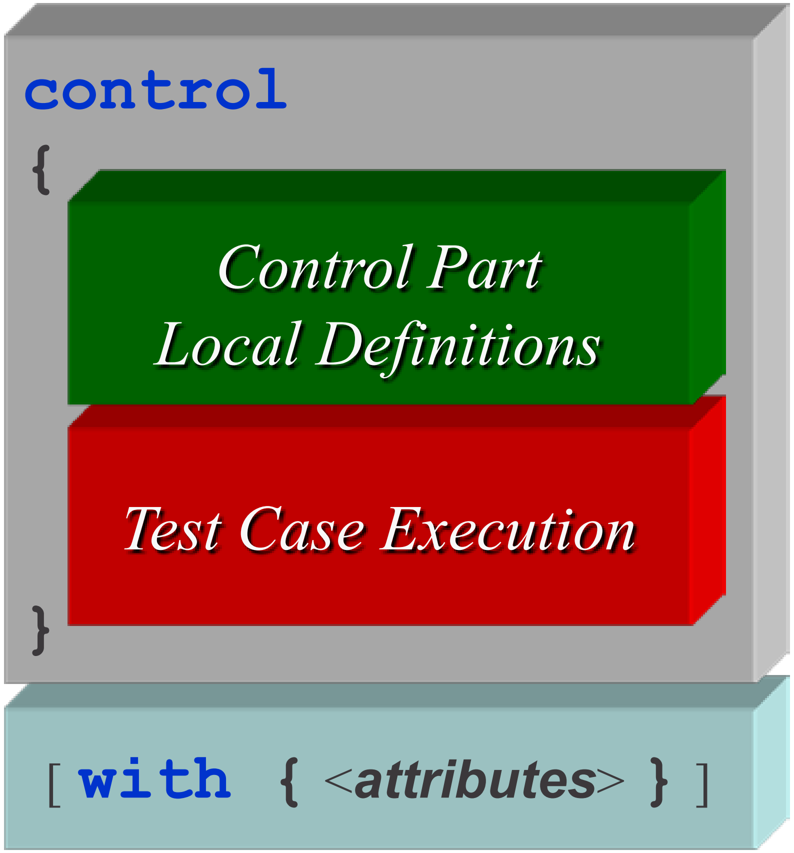
# MODULE DEFINITIONS PART

**Definitions in module definitions part are globally visible within the module**

- Module parameters are external parameters, which can be set 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 dynamic behaviour of the tests



# MODULE CONTROL PART



- 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 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
```

# VERSION INFORMATION

- Specifies if the TTCN-3 module requires a minimum version of another TTCN-3 module or a minimum version of TITAN.

```
module supplier {
 …
}
with {
extension "version R1A";
}
```

module's own version information can be specified in an extension attribute

module X has to be compiled with TITAN R8C or later.

```
module X {
…
}
with {
extension "requires TITAN R8C";
}
```

minimum version of an imported module can be specified

```
module importer {
import from supplier all;
}
with {
extension "requires supplier R2A"
}
```

# 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

# 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
- Sub-typing constructions
  - Restrict the value domain of the parent type
- Aliasing

- Type compatibility
- Forward referencing permitted in module definitions part

# 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: `"∂ξ"`
  - May also contain characters referenced by quadruples, e.g.:
  - `char(0, 0, 40, 48)`

# SPECIAL TYPES (1)

- **anytype**
  - Introduced to allow mapping of CORBA IDL to TTCN-3;
  - Defined as a shorthand for the union of all *known types* in a TTCN-3 module, where *known type* embraces all built-in types, user-defined types, imported ASN.1 and other imported external types.
  - The fieldnames of the **anytype** shall be uniquely identified by the corresponding type names using the "dot" notation.
  - Performance problems – not to use unless explicitly necessary!
    - all used types must be listed at the end of the module – with {extension “anytype ... “}

```
module my_Module {
  type record MyRec {integer i,float f}
  control { var anytype v_any;
            v_any.charstring := "three";
            v_any.MyRec := {{ 1,true} }
  } with { extension "anytype charstring, MyRec"}
```

# SPECIAL TYPES (2)

Configuration types are used to define the architecture of the test system:

- **`port`**
  - A port type defines the allowed message and signature types between test components → Test Configuration
- **`component`**
  - Component type defines which ports are associated with a component → Test Configuration
- **`address`**
  - Single user defined type for addressing components
  - Used
    - to interconnect components → Test Configuration
    - in **`send to`**/**`receive from`** operations and **`sender`** clause → Abstract Communication Operations

# SPECIAL TYPES (3)

- **default**
  - Implementation-dependent type for storing the default reference
  - A default reference is the result of an **activate** operation
  - The default reference can be used to a **deactivate** given default
    → Behavioral Statements

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

# 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 visibility (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)

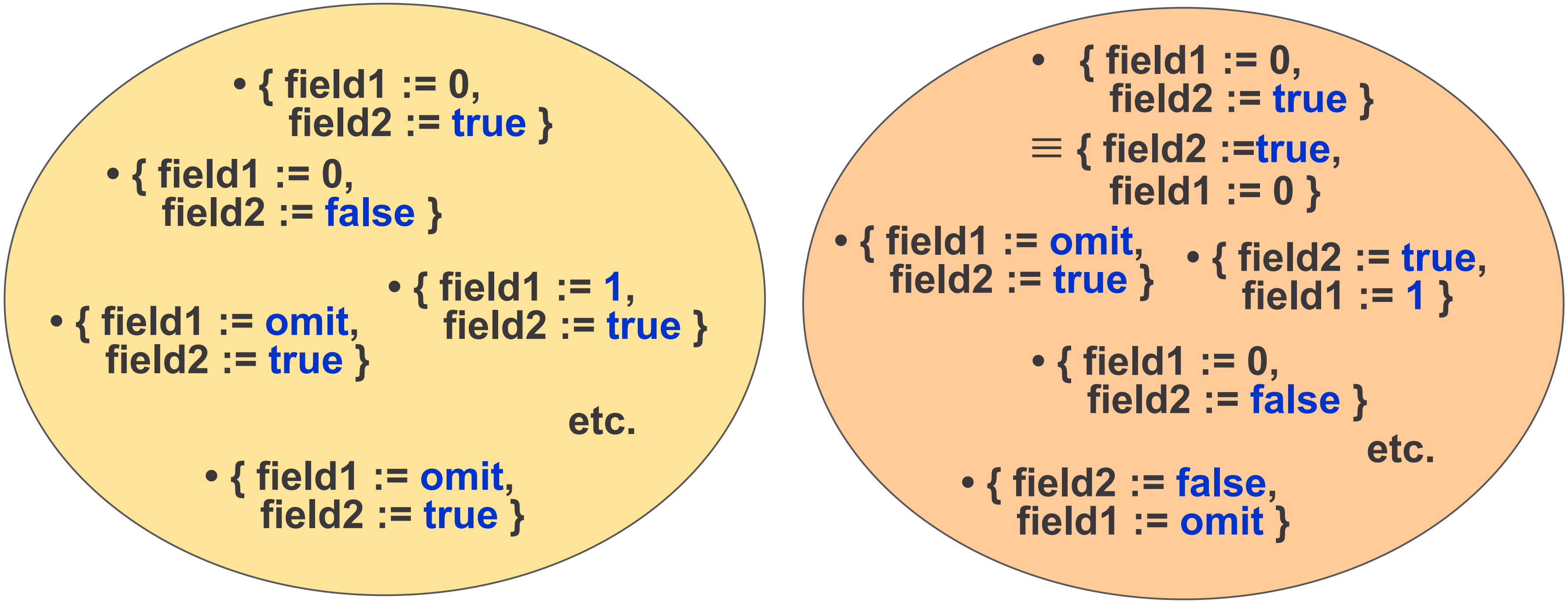
```
// example record type def.
type record MyRecordType {
  integer field1 optional,
  boolean field2
}
```

```
// 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



- { field1 := 0,
  field2 := true }
- { field1 := 0,
  field2 := false }
- { field1 := 1,
  field2 := true }
- { field1 := omit,
  field2 := true }

etc.

- { field1 := omit,
  field2 := true }

- { field1 := 0,
  field2 := true }

≡ { field2 :=true,
  field1 := 0 }
- { field1 := omit,
  field2 := true }
- { field2 := true,
  field1 := 1 }
- { field1 := 0,
  field2 := false }

etc.

- { field2 := false,
  field1 := omit }

**MyRecordType** **MySetType**

# VALUE ASSIGNMENT NOTATION

- Values may be explicitly assigned to fields
  - not present optional elements must be set to **omit**
  - values of the unlisted elements remain unbound
  - applicable for: **record**, **set**, **union**

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

```
var MyRecordType v_myRecord2 := {
  field2 := true // field1 presents, but unbound
}
```

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

# VALUE LIST NOTATION

- Value list notation
  - Elements are assigned in the order of their definition
  - All elements must 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, but not for **set**

```
var MyRecordType v_myRecord3 := { 1, true }
var MyRecordType v_myRecord4 := { omit, true }
var MyRecordType v_myRecord5 := { -, true } // <unbound>,true
                 v_myRecord5 := { 1, - }    // 1, true
```

```
var MySetType v_mySet2 := { 1, true }     // not for set

var MyRecordType v_myRecord6 := { true } // not all fields!
```

# STRUCTURED TYPES – NESTED VALUES

```
type record InternalType {
       boolean field1,
       integer field2 optional
};
type record RecType {
       integer field1,
       InternalType field2
};
const RecType c_rec := {
       field1 := 1,
       field2 := {  field1 := true,
                    field2 := omit
                    }
 };
// same as previous, but with value list
const RecType c_rec2 := { 1, { true, omit } }
```

# FIELD REFERENCES

- Reference or “dot” notation
  - – Can not be used at specification, only for previously defined variables
  - Referencing structured type fields
  - Applicable in dynamic parts (e.g. **function**, **control**) only

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

```
type record R1 {
  integer i,
  boolean b
}
type record R2 {
  R1 r1,
  integer i2
}
```

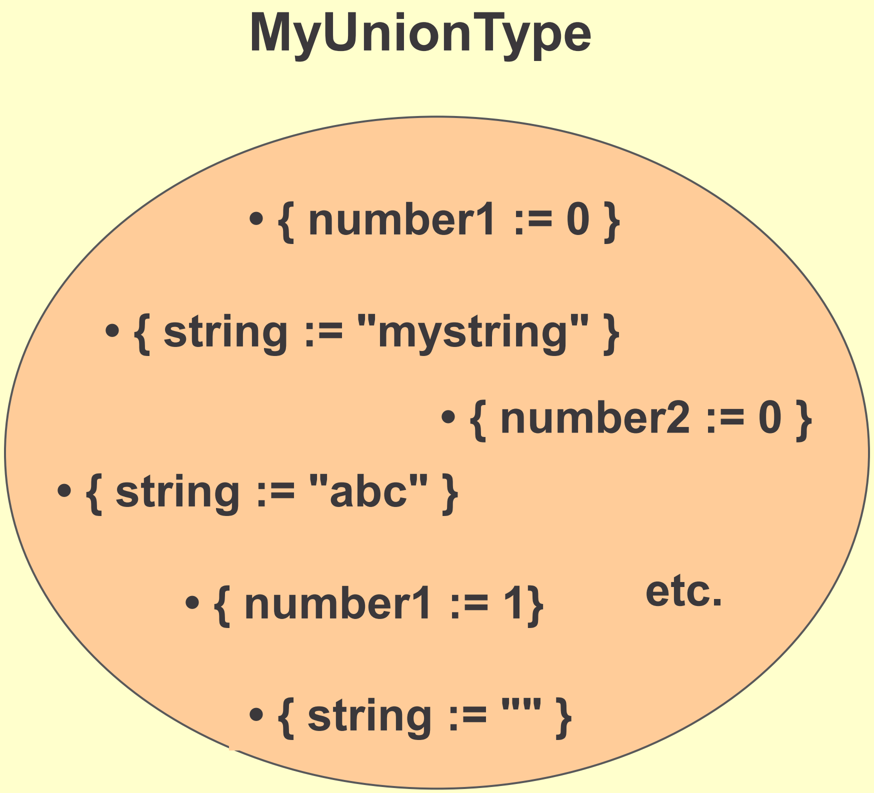
```

var R2 r2;

r2.i2 := 2;
r2.r1.i := 1;
r2.i := 11;
```

# 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 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 :=
               {number1 := 12}
var MyUnionType v_myUnion;
v_myUnion := {number1 := 12}
v_myUnion.number1 := 12;

// usage of ischosen
if(ischosen(v_myUnion.number1)) { … }
```



# STRUCTURED TYPES – RECORD OF, SET OF

- User defined abstract container type representing an ordered /unordered *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;
var ROI v_il := { 1, 2, 3 };

// set of types, the order is irrelevant
type set of MySetType MySetList;
var MySetList v_msl := {
 v_mySet1, { field2 := true, field1 := omit }, v_mySet1
};
```

**remember :**

```
var MySetType v_mySet1 := {
  field2 := true,
  field1 := omit
}
```

# 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;
```

# INDEXING

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

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

# NOT-USED, OMIT AND UNBOUND

- omit – structured type’s optional field not present
- unbound – uninitialized value
- not-used (“-”) – preserves the original value, only in value list notation

```
var ROI u, v := { -, 2, - }; // v == {<unbound>, 2, <unbound>}
log(sizeof(v)); // 3
v[0] := 1; // v == { 1, 2, <unbound> }
u := v;
v := { -, -, 3 }; // v == { 1, 2, 3 }
```

```
var MyRecordType r1, r2, r3, r4;
r1 := { field2 := true } // r1 == { <unbound>, true }
r2 := { -, true }; // r2 == { <unbound>, true } == r1
r3 := { omit, true }; // r3 == { omit, true } != r1
r4 := { 1 }; // PARSE ERROR!
```

```
type record MyRecordType {
  integer field1 optional,
  boolean field2
}
```

# STRUCTURED TYPES – ENUMERATED

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

```
type enumerated Ex1 {tuesday, friday, wednesday, monday};
```

- Enumeration items (literals):
  - 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 Ex2 {tuesday(1),friday(5), wednesday, monday};
```

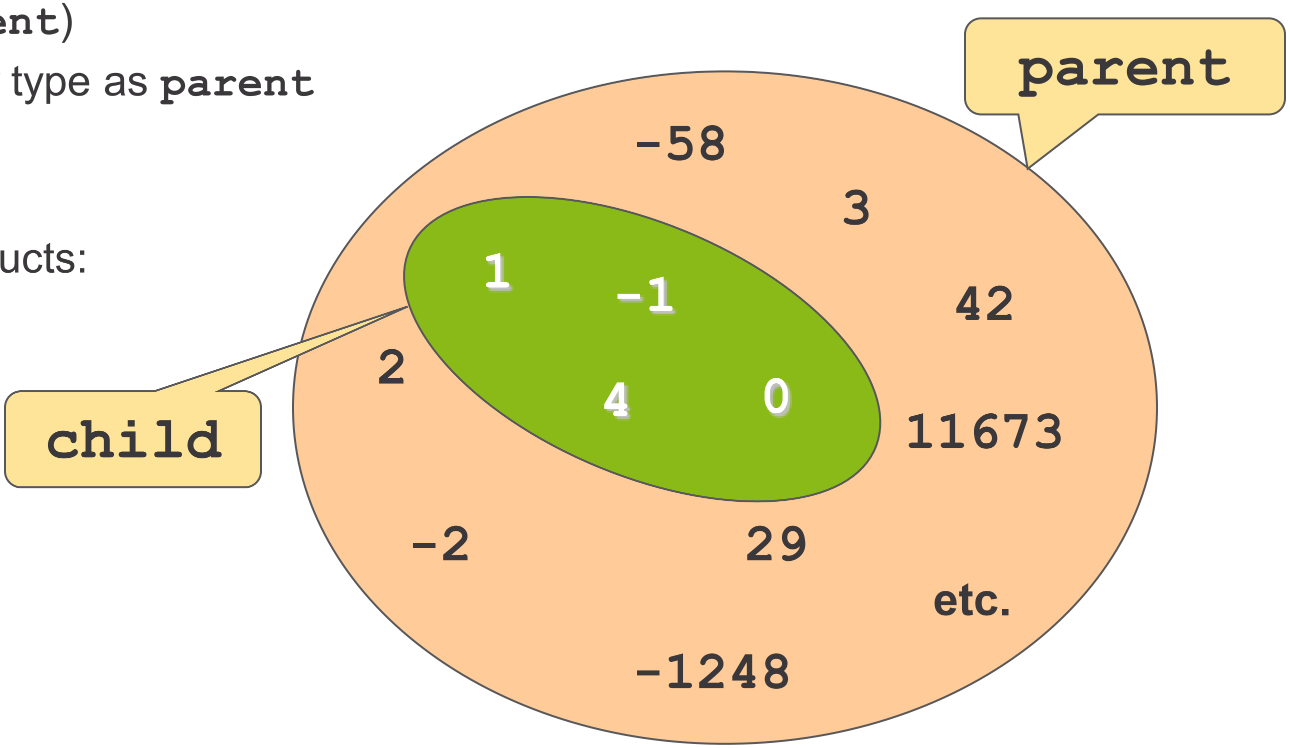
- Operations on enumerations
  - must always use literals – integer values are only for encoding!
  - are restricted to assignment, equivalence and comparing (<,>) operators
- `enumerated` versus `integer` types
  - Enumerated types are *never* compatible with other basic or structured types!

# 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
// v_11 > v_12 is false

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_22 causes error: different types of variables!
// v_11 > 2 causes error: enumerated is not integer
```

# SUB-TYPING

- Deriving a new type **`child`** from an existing **`parent`** type by restricting the new type's domain to a subset of the parent types value domain:
  - D(**`child`**) ⊆ D(**`parent`**)
- **`child`** has the same *root* type as **`parent`**
- Applicable to elements of structured types also
- Various sub-typing constructs:
  - value range,
  - value list,
  - length restriction,
  - patterns,
  - type alias.



# 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: LENGTH RESTRICTIONS (1)

- Length restrictions are applicable for basic string types.
- The unit of length depends on the constrained type:
  - **bitstring** – bit,
  - **hexstring** – hexa digit,
  - **octetstring** – octet,
  - **charstring**/**universal charstring** – character

```
// length exactly 8 bits
       type bitstring MyByte length(8);
// length exactly 8 hexadecimal digits
       type hexstring MyHex length(8);
// minimum length 4, maximum length 8 octets
       type octetstring MyOct length(4 .. 8);
```

# SUB-TYPING: LENGTH RESTRICTIONS (2)

- `length` keyword is used to restrict the number of elements in `record of` and `set of`.
- It is permitted to use a range inside the length restriction

```
// a record of exactly 10 integers
      type record length(10) of integer RecOfExample;

// a record of a maximum of 10 integers
      type record length(0..10) of integer RecOfExamplf;

// a set of at least 10 integers
      type set length(10..infinity) of integer RecOfExampg;
```

# 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'*/
```

# SUB-TYPING: TYPE ALIAS

- An alternative name to an existing type;
- similar to a subtype definition, but the subtype restriction tag (value list, value or length restriction) is missing.

```
type MyType MyAlternativeName;
```

# OVERVIEW OF SUBTYPE CONSTRUCTS FOR TTCN-3 TYPES

| Class of type | Type name (keyword) | Sub-Type |
|---|---|---|
| Simple basic types | `integer, float` | range, list |
| | `boolean, objid, verdicttype` | list |
| Basic string types | `bitstring, hexstring, octetstring` | list, length |
| | `charstring, universal charstring` | range, list, length, pattern |
| Structured types | `record, set, union, enumerated` | list |
| | `record of, set of` | list, length |
| Special data types | `anytype` | list |

# TYPE COMPATIBILITY IN TITAN

- Deviations from TTCN-3:
  - Aliased types and sub-types are treated to be equivalent to their unrestricted root types
  - Different structured types are incompatible to each other
  - Two array types are compatible if both have the same size and index offset and the types of the elements are compatible according to the rules above
- Built-in functions available for converting between incompatible types:

```
int2char(65) == "A" // ASCII(65): letter A
int2str(65) == "65"
hex2str('FABABA'H) == "FABABA"
```

# PREDEFINED CONVERSION FUNCTIONS

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

***log2str; enum2int***

# 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)
- `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 variable definition in module definition part

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

- Iteration counter of for loops

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

- Optionally, an initial value may be assigned to a 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 indexes 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]; // array of 3 integers
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 set array 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`

```
// 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 restriction 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;
}
```

# SCOPES

- TTCN-3 provides seven basic units of scope:
  - module definition part ( `module` ) – global
  - control part of a module ( `control` )
  - block of statements ( `{...}` )
  - functions ( `function` )
  - altsteps ( `altstep` )
  - test cases ( `testcase` )
  - component types ( `component` ) – ‘runs on’ clause
- Identifiers must be unique within the entire scope hierarchy

# VISIBILITY MODIFIERS

- On module level
  - **public** definition is visible in every module importing the module. (default)
  - **private** the definition is only visible within the same module.
  - **friend** the definition is only visible within the friend declared module.

```
module module1
{
friend module module2;
type integer module2Type;
public type integer module2TypePublic;
friend type integer module2TypeFriend;
private type integer module2TypePrivate;
} // end of module
```

```
module module2
{
import from module1 all;
const module2Type c_m2t:= 1;
//OK, type is implicitly public
const module2TypePublic c_m2tp := 2;
//OK, type is explicitly public
const module2TypeFriend c_m2tf := 3;
//OK, module1 is friend of module2
const module2TypePrivate c_m2tpr := 4;
//NOK, module2TypePrivate is private
  to module2
```

# VI. PROGRAM STATEMENTS AND OPERATORS

EXPRESSIONS
ASSIGNMENTS
PROGRAM CONTROL STATEMENTS
OPERATORS
EXAMPLE

CONTENTS

# EXPRESSIONS, ASSIGNMENTS, LOG, ACTION AND STOP

| Statement | Keyword or symbol |
|---|---|
| Expression<br>Condition (Boolean expression) | e.g. `2*f1(v1,c2)+1`<br>e.g. `x+y<z` |
| Assignment (not an operator!) | *LHS* `:=` *RHS*<br>e.g. `v := { 1, f2(v1) }` |
| Print entries into log | `log(a);`<br>`log(a, …);`<br>`log(″a = ″, 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*>) {<br>case (<*template*>) { <*statement*> }<br>[case (<*template-list*>) { <*statement*> } ]<br>…<br>[case else { <*statement*> } ]<br>} |
| 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*>; |

# BREAK AND CONTINUE

- `break`
  - Leaves innermost loop
  - *or* alternative within `alt` or `interleave` statement
- `continue`
  - Forces next iteration of innermost loop

# OPERATORS (1)

| Category | Operation | Format | Type of operands and result |
|---|---|---|---|
| Arithmetical | Addition | $+op$ or $op_1 + op_2$ | $op$ , $op_1$, $op_2$, *result*:<br>integer, float |
| | Subtraction | $-op$ or $op_1 - op_2$ | |
| | Multiplication | $op_1 * op_2$ | |
| | division | $op_1 / op_2$ | |
| | Modulo | $op_1$ mod $op_2$ | $op_1$, $op_2$, *result*: integer |
| | Remainder | $op_1$ rem $op_2$ | |
| String | Concatenation | $op_1$ & $op_2$ | $op_1$, $op_2$, *result*: *string |
| Relational | Equal | $op_1 == op_2$ | $op_1$, $op_2$: all;<br>*result*: boolean |
| | Not equal | $op_1$ != $op_2$ | |
| | Less than | $op_1 < op_2$ | $op_1$, $op_2$: integer, float, enumerated;<br>*result*: boolean |
| | Greater than | $op_1 > op_2$ | |
| | Less than or equal | $op_1 <= op_2$ | |
| | Greater than or equal | $op_1 >= op_2$ | |

# OPERATORS (2)

| Category | Operator | Format | Type of operands and result |
|---|---|---|---|
| Logical | NOT | `not` *op* | *op*, $op_1$, $op_2$, *result*: `boolean` |
| | AND | $op_1$ `and` $op_2$ | |
| | OR | $op_1$ `or` $op_2$ | |
| | exclusive OR | $op_1$ `xor` $op_2$ | |
| Bitwise | NOT | `not4b` *op* | op, $op_1$, $op_2$, *result*: `bitstring, hexstring, octetstring` |
| | AND | $op_1$ `and4b` $op_2$ | |
| | OR | $op_1$ `or4b` $op_2$ | |
| | exclusive OR | $op_1$ `xor4b` $op_2$ | |
| Shift | left | $op_1$ << $op_2$ | $op_1$, *result*: `bitstring, hexstring, octetstring`; $op_2$: `integer` |
| | right | $op_1$ >> $op_2$ | |
| Rotate | left | $op_1$ `<@` $op_2$ | $op_1$, *result*: `bitstring, hexstring, octetstring, (universal) charstring;` $op_2$: `integer` |
| | right | $op_1$ `@>` $op_2$ | |

# OPERATOR PRECEDENCE

| Precedence | Operator type | Operator |
|---|---|---|
| **Highest** | *parentheses* | `()` |
| | Unary | `+, -` |
| | Binary | `*, /, mod, rem` |
| | Binary | `+, -, &` |
| | Unary | `not4b` |
| | Binary | `and4b` |
| | Binary | `xor4b` |
| | Binary | `or4b` |
| | Binary | `<<, >>, <@, @>` |
| | Binary | `<, >, <=, >=` |
| | Binary | `==, !=` |
| | Unary | `not` |
| | Binary | `and` |
| | Binary | `xor` |
| | Binary | `or` |
| **Lowest** | | |

# 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 **float** 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
T2.start(2.5); // started for 2.5s (overrides default)
```

- 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 **timeout** operation waits a timer to expire (blocking operation)

```
T_myTimer.timeout; // waits 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 T := 5.0;
T.start; or T.start(2.5);
T.timeout; // block until timer 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;
```

# OTHER TIMER OPERATIONS: RUNNING, READ

- The **running** operation can be used to determine if a timer is running (returns a **boolean** value, does not block)

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

- Timers count from zero upwards
- The running timer’s elapsed value can be retrieved and optionally saved into a **float** variable using the **read** operation:

```
// Reading the elapsed time of the timer
var float v_myVar := T_myTimer.read;
```

- **read** returns zero for an inactive timer:

```
timer T_myTimer2;
var float v_myVar2 := T_myTimer2.read; // v_myVar2 == 0.0
```

# VIII. TEST CONFIGURATION

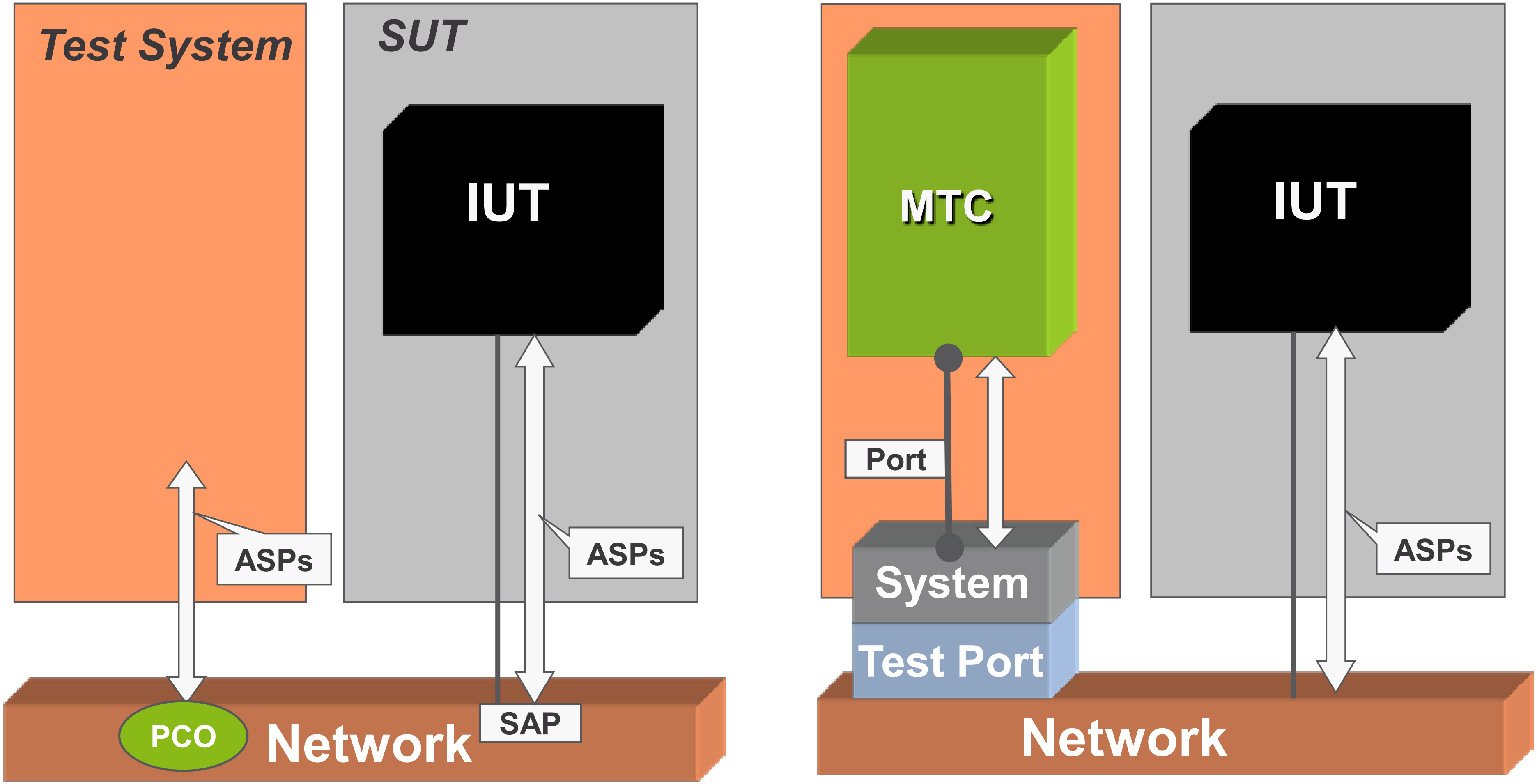
TEST COMPONENTS AND COMMUNICATION PORTS
TEST COMPONENT DEFINITIONS
COMMUNICATION PORT DEFINITIONS
EXAMPLES

CONTENTS

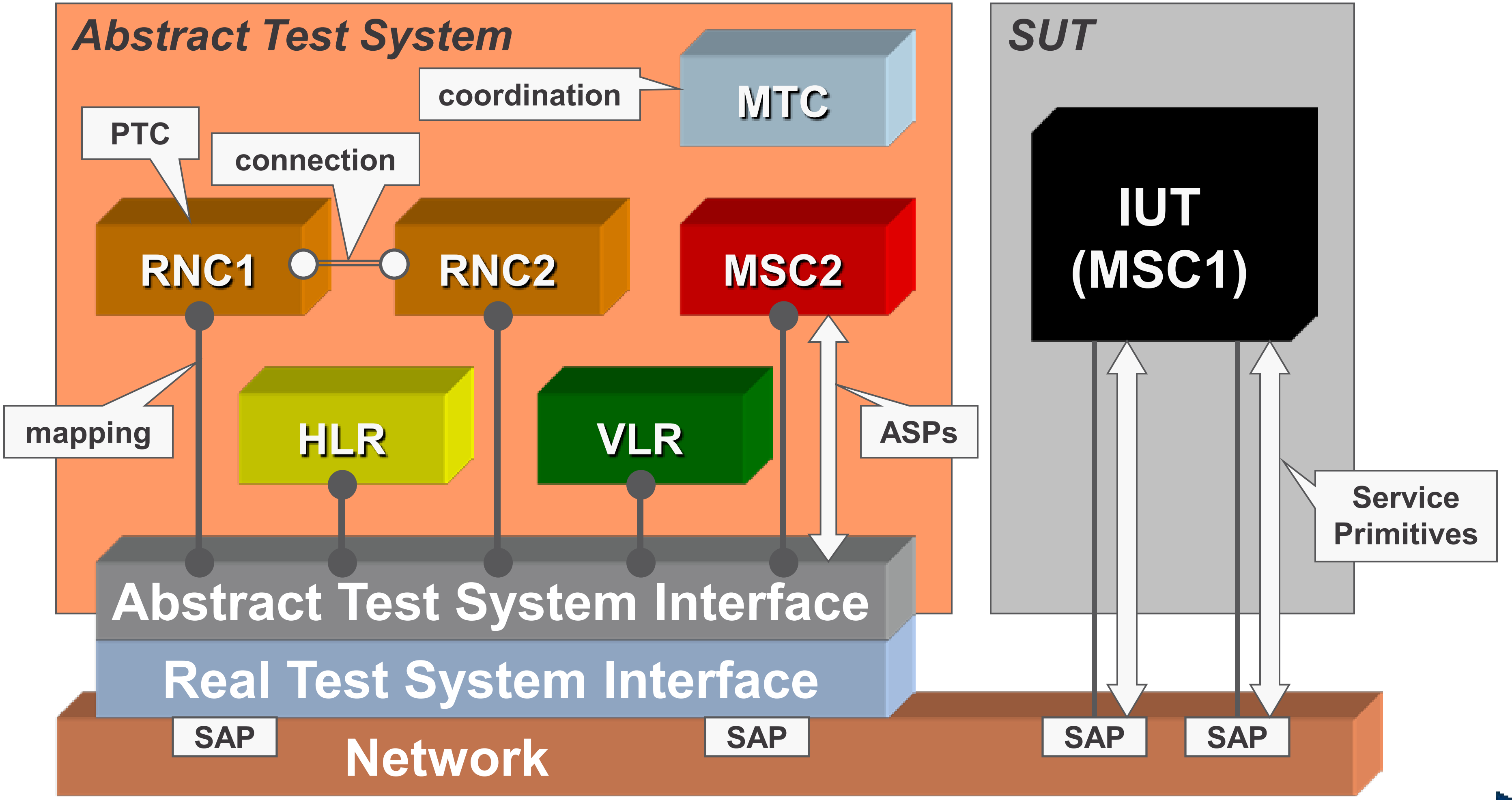
# TEST CONFIGURATION

- 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 parallel processes
  - at the beginning of the `testcase` the test configuration must be established → Configuration Operations
  - test configuration can be changed during test execution

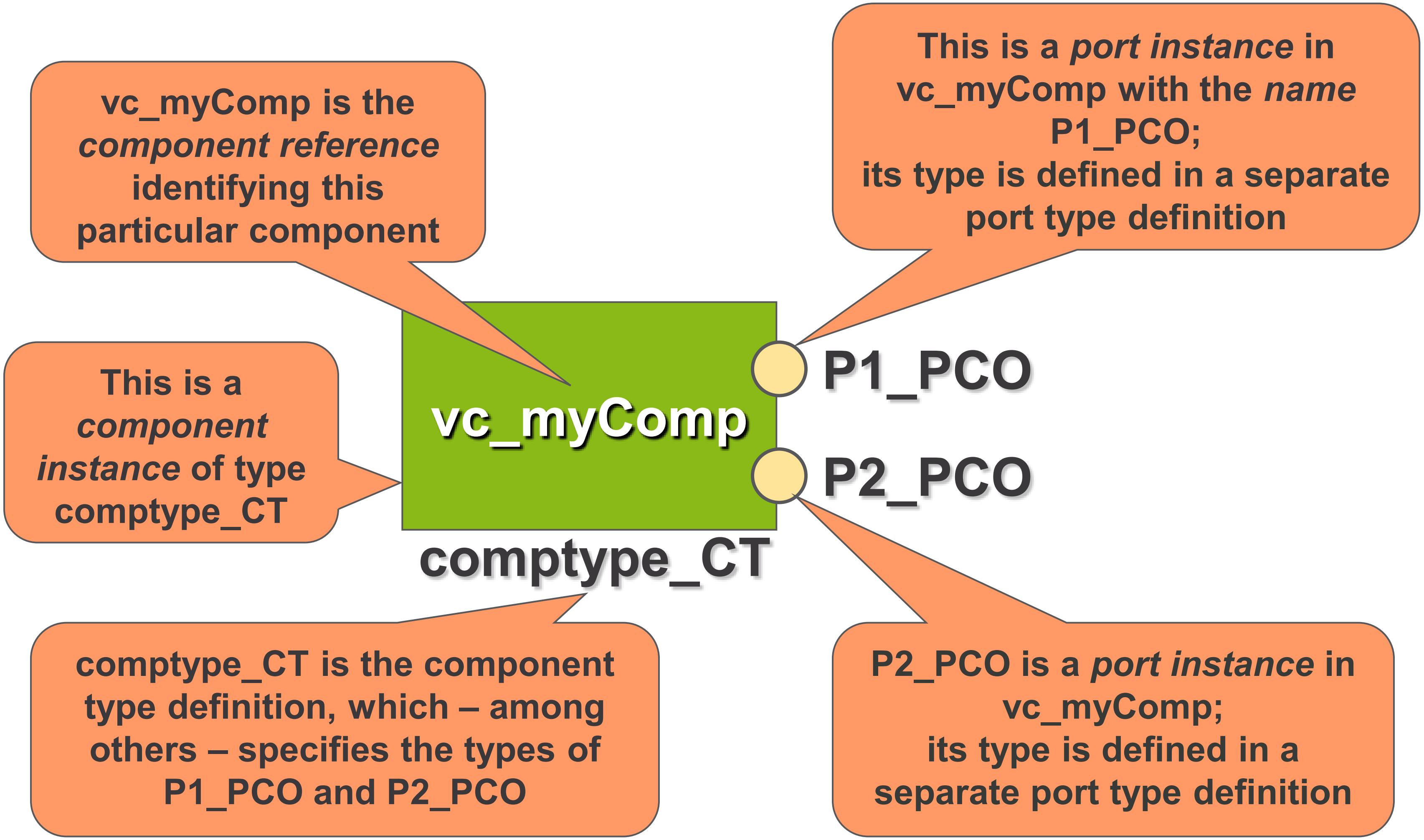
# TEST ARRANGEMENT AND ITS TTCN-3 MODEL – TESTER IS A PEER ENTITY OF IUT



# TTCN-3 VIEW OF TESTING – DISTRIBUTED TESTER



# 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;
its type is defined in a separate port type definition

This is a *component instance* of type comptype_CT

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

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

# 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
- Interfaces connecting the TTCN-3 components with the real IUT are implemented in C++ and are called *test ports* (TITAN specific!)

# PORT COMMUNICATION MODEL

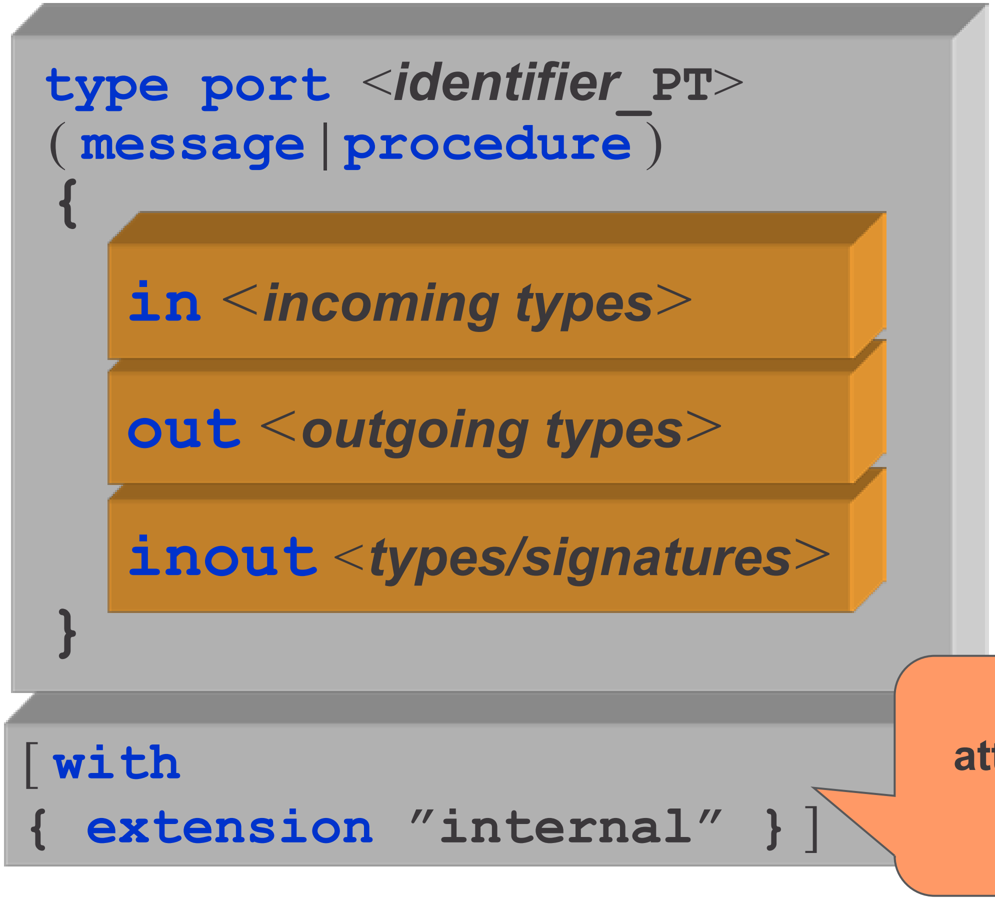
- **The port communication is full duplex**
  - **the direction of certain message and signature types (`in`, `out`, `inout`) can be restricted in the port type definition**

- **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 only between peer ports**
  - **Internal (component-to-component) communication**
    - **between *connect*ed ports → Communication Operations**
  - **External (component-to-system) communication**
    - **between *map*ped ports → Communication Operations**
    - **test ports to be added**

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

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

# 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).

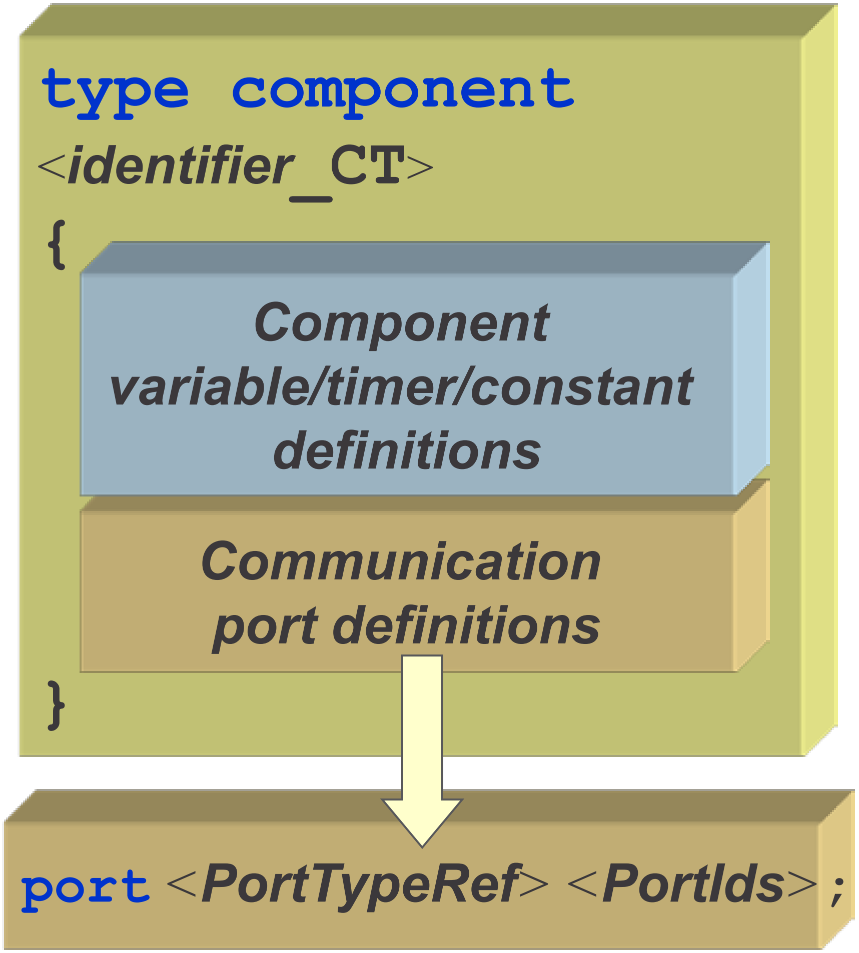
`integer` and `octetstring` type messages can be both sent and received.

`ASP_TxType` messages can only be sent.

# TEST COMPONENTS

- Test components are the building blocks of test configurations
- Components execute test behavior
- Three types 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 generated automatically in all test configurations (as the first two components)
- The (`runs on` clause of) 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

- in module definitions 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 each instance of this component type (local copies in each component instance)

```
// Definition of a test component type
type component MyComponentType_CT
{// ports owned by the component:
 port MyPortType_PT PCO;
 port MyPortType_PT PCO_Array[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

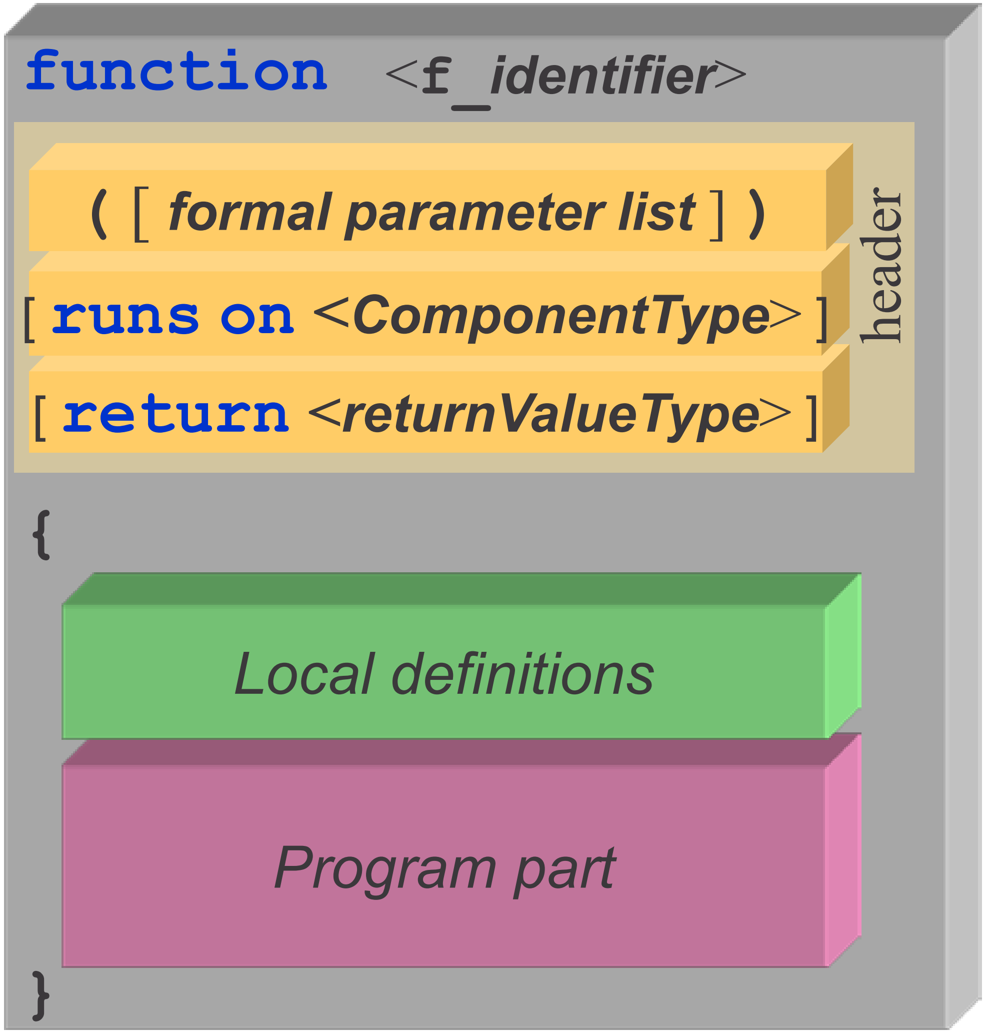
# ABOUT FUNCTIONS

- 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


function <f_identifier>
( [ formal parameter list ] )
[ runs on <ComponentType> ]
[ return <returnValueType> ]
header
{
Local definitions
Program part
}

- The optional `runs on` clause restricts the execution of the function onto the instances of a specific *ComponentType*
  - BUT: local definitions of *ComponentType* (***ports!!*** etc.) can be used
- 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 type, number and order of actual parameters shall be the same as of the formal parameters;
- All variables in the actual parameter list must be bound:

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

- Empty parentheses indicate 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)

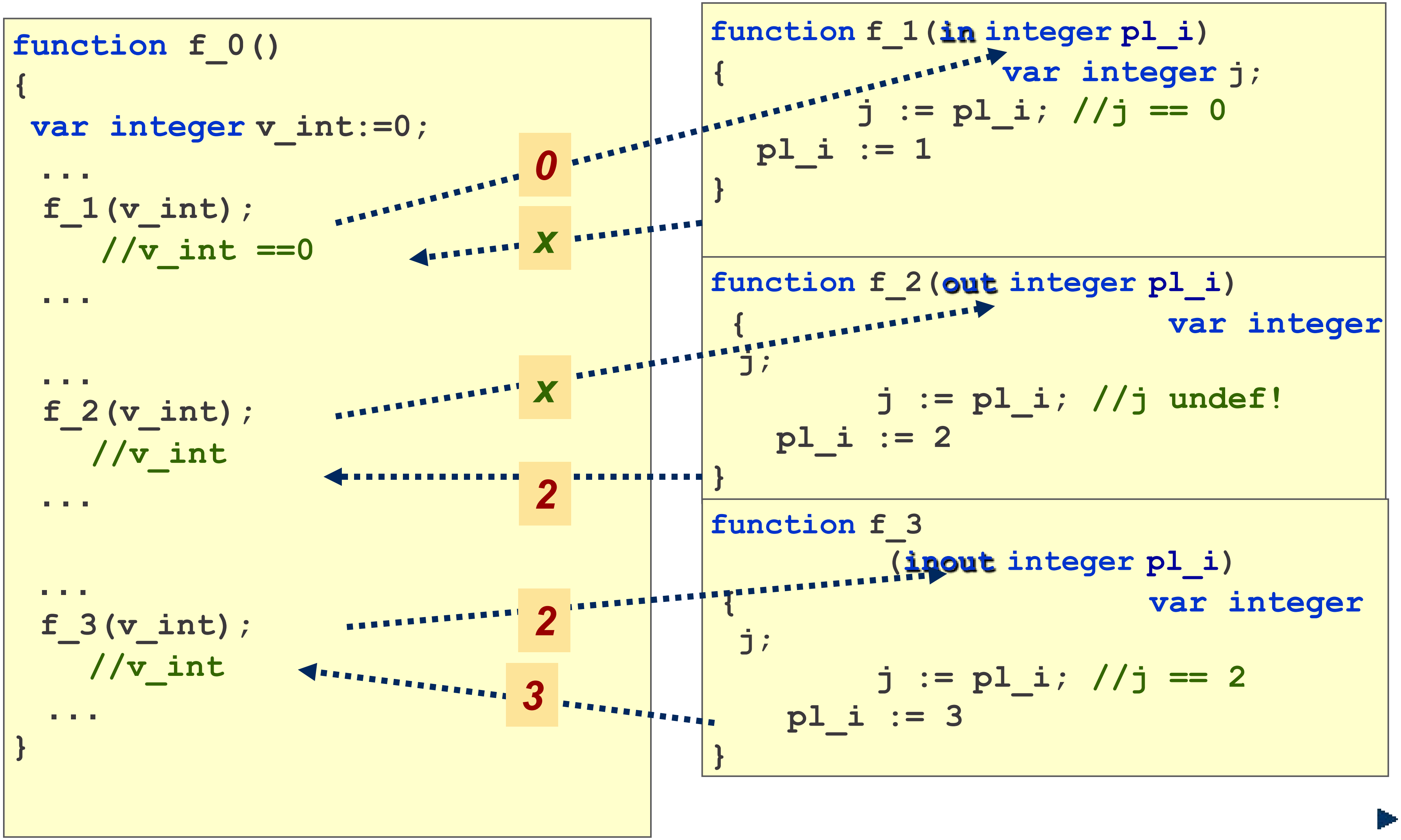
Operands of an expression may invoke a function:

```
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
}
```

# DEFAULT VALUES

- `in` parameters may have default values
- at invocation
  - “`-`” (hyphen) skips the parameter with default value
  - simply leaving out (if it is the last, or all the following have default values)
  - default value may be overwritten

```
function f_MyFDef (integer i, integer j:=2, integer k){}
function f_MyFDef2 (integer i, integer j:=2, integer k:=3){}

// invocation
f_MyFDef(1,-,3); // f_MyFDef(1,2,3);
f_MyFDef(1,5,3); // f_MyFDef(1,5,3);
f_MyFDef2(1,5,7);// f_MyFDef2(1,5,7);
f_MyFDef2(1,5);  // f_MyFDef2(1,5,3);
f_MyFDef2(1);    // f_MyFDef2(1,2,3);
```

# 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]*) |
| Returns the name of the currently executing test case | `testcasename` () |

# PREDEFINED CONVERSION FUNCTIONS

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

***log2str; enum2int***

# NEW PREDEFINED FUNCTIONS

- **log2str** (*log-arguments*) **return charstring**
  Returns formatted output of arguments instead of placing them to log file (TITAN)

```
// Save output of log statement instead of
var charstring str
str := log2str(”Value of v is:”, v);
```

- **enum2int** (*enumeration-reference*) **return integer**
  Gives **integer** value associated with enumeration item

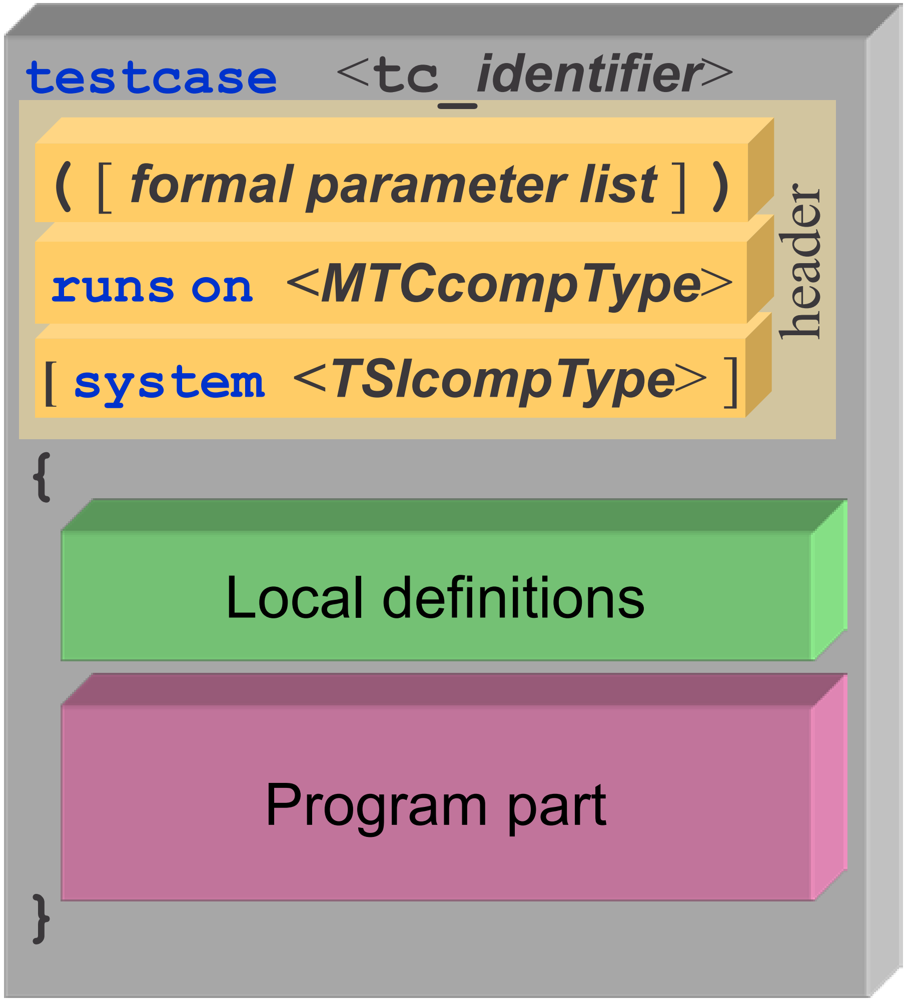
```
type enumerated E { zero, one, two, three };
var E e := one;
integer i := enum2int(one); // i == 1
```

- **isvalue** (*inline-template*) **return boolean**
  Returns **true** if argument template contains specific value or **omit**

# A TESTCASE

- A special function, which is always executed (runs) on the MTC;
- In the module control part, the **`execute()`** statement is used to start **`testcase`**s;
- The result of test case execution is always of *verdicttype*
  - with the possible values: **`none`**, **`pass`**, **`inconc`**, **`fail`** or **`error`**;
- **`testcase`**s 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 timer 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(), 5.0);
```

# 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(), 0.5);

  // 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 without 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
}
```

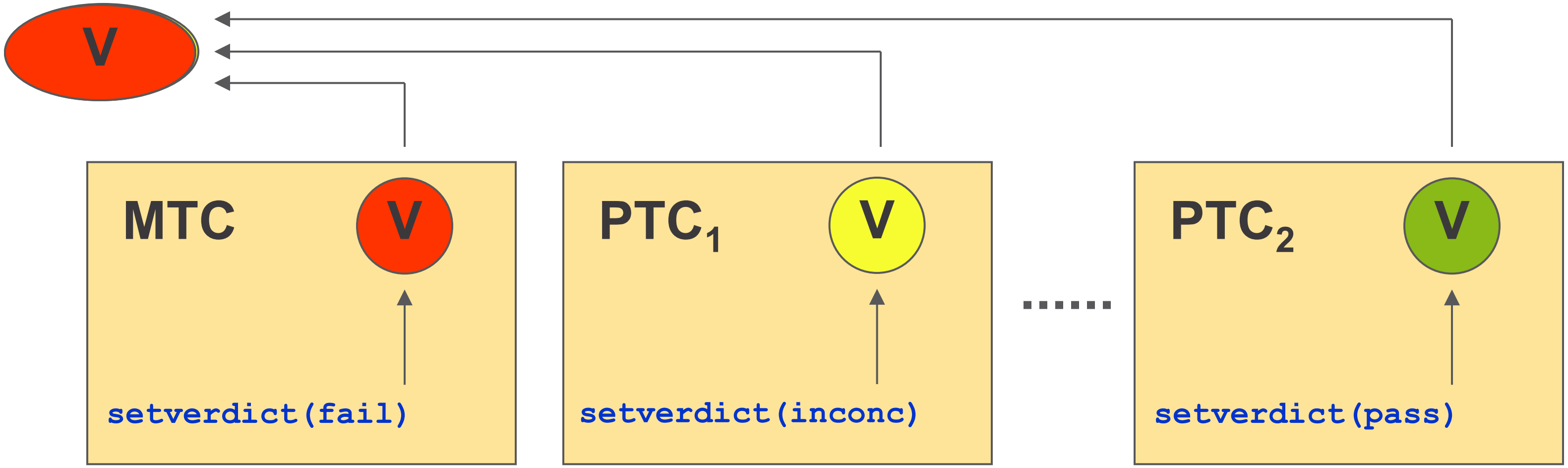
# VERDICT OVERWRITING LOGIC

| 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.

*Global verdict returned by the test case at termination*



# 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 configuration in TTCN-3 is *DYNAMIC*:
  - MUST be explicitly set up at the beginning of each test case;
  - MTC is the only test component, which is automatically generated 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 at 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 (an alive component can be restarted after it is stopped)
- The **create** operation creates the component and returns by the unique component reference of the newly created component
  - this reference is to be stored in a Component Type (address) variable
- 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”);
// Specifying 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 (meta character %n)
  - can be used in test port 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:

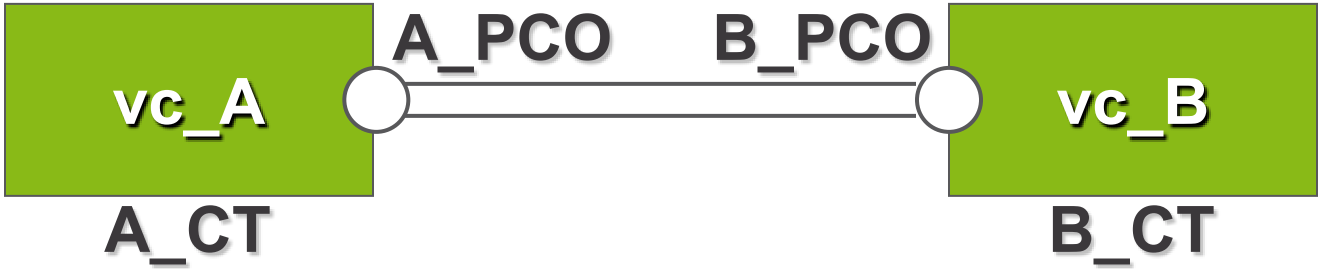
```
var ComponentType_CT vc_CompReference;
vc_CompReference := ComponentType_CT.create;
```

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

# CONNECTING COMPONENTS

- Connecting components means connecting their ports;
- The **connect** operation is used to connect component ports;
- A connection to be established is 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).

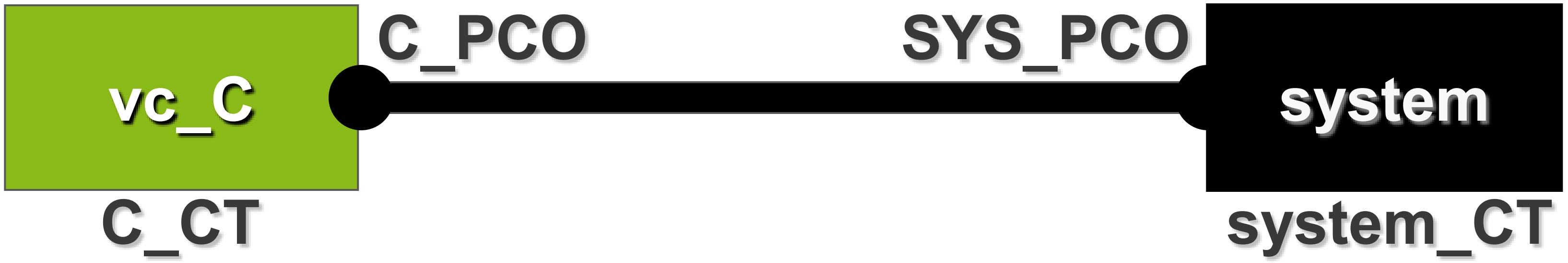
```
vc_A := A_CT.create; // vc_A: component reference
vc_B := B_CT.create; // vc_B: component reference
connect(vc_A:A_PCO, vc_B:B_PCO); // A_PCO: port name
```



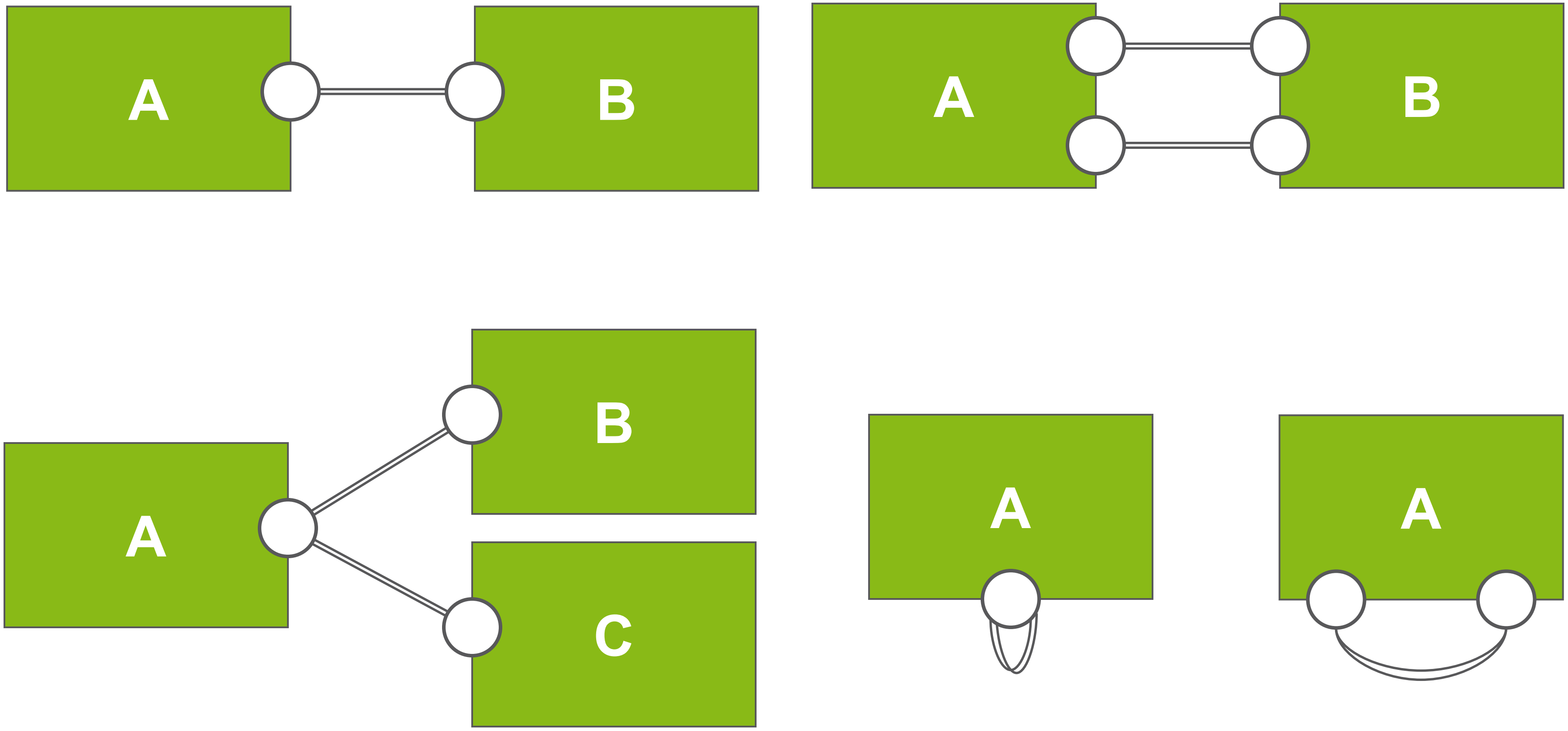
# MAPPING A TEST SYSTEM INTERFACE PORT TO A COMPONENT

- The **map** operation is used to establish a connection between a port of the system and a port of a component;
  - Test port must be added
- 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.

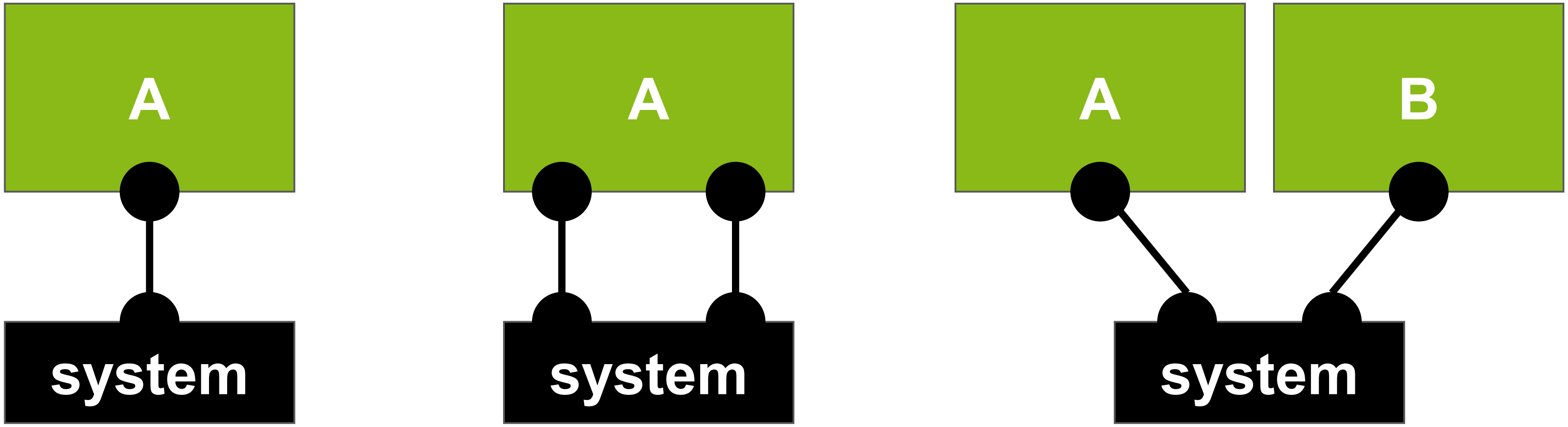
```
vc_C := C_CT.create; // vc_C: component reference
map(vc_C:C_PCO, system:SYS_PCO); // SYS_PCO: port ref.
```



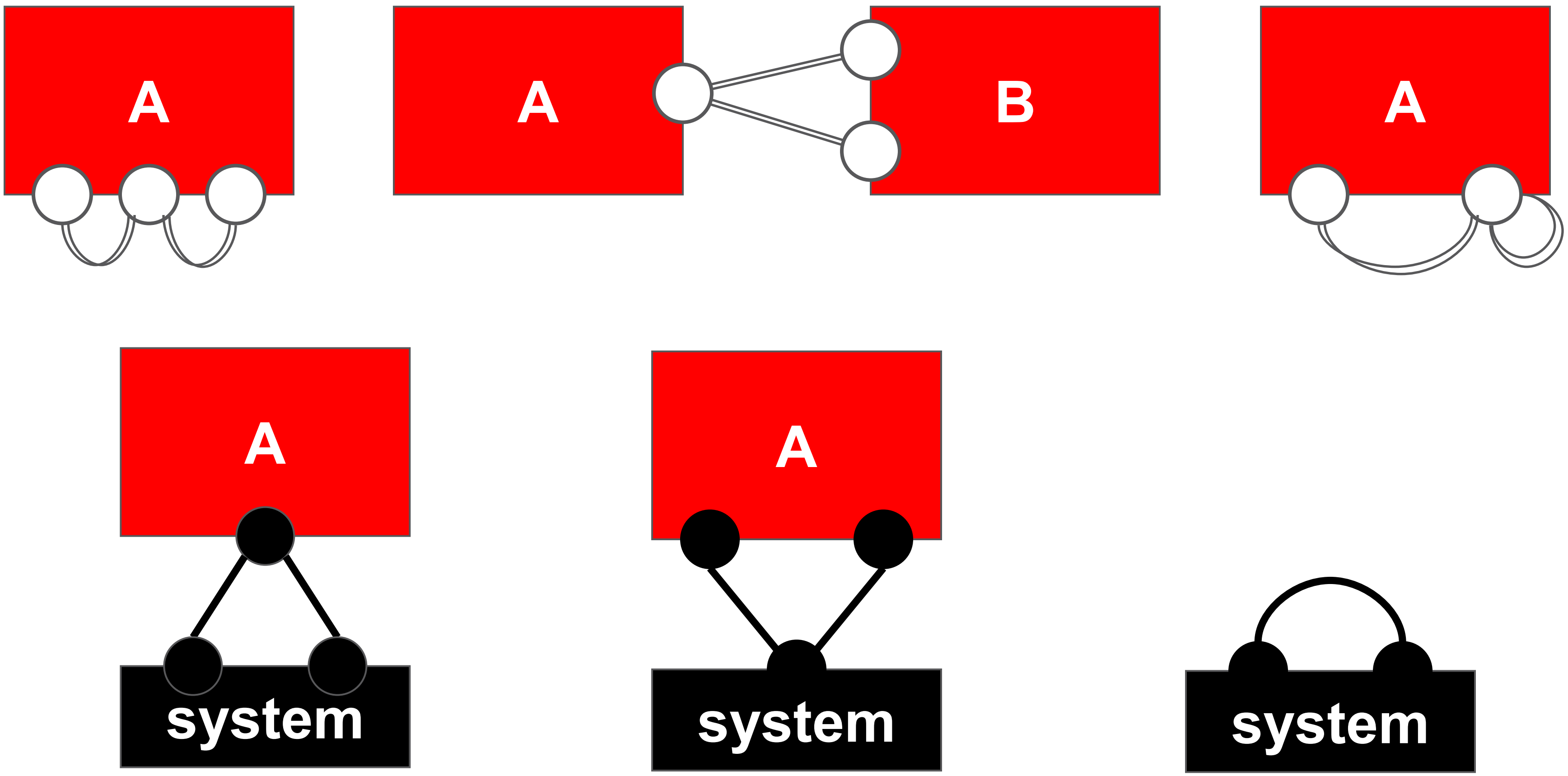
# BASIC EXAMPLES FOR VALID CONNECTIONS



# VALID MAPPINGS



# INVALID CONNECTIONS AND MAPPINGS



# DYNAMIC TEST CONFIGURATION

- Creating or destroying connection between two ports of different parallel test components

```
connect(vc_A : A1_PCO, vc_B : B1_PCO);
disconnect(vc_A : A1_PCO, vc_B : B1_PCO);
```

- Creating or destroying connection between a port of SUT and a port of a TTCN-3 test component

```
map(system:SYS_PCO, vc_B:B1_PCO);
unmap(system:SYS_PCO, vc_B:B1_PCO);
```

- Where **`vc_A`**, **`vc_B`** are component references, **`A1_PCO`** and **`B1_PCO`** are port references

# 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));
```

# 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
```

# WAITING FOR A PTC TO TERMINATE

- The **done** operation
  - blocks execution while a PTC is 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

```
vc_A.done; // blocks execution until vc_A terminates

all component.done; // blocks the execution until all
                   // parallel test components terminate

vc_B.killed; // wait until vc_B alive component is killed
```

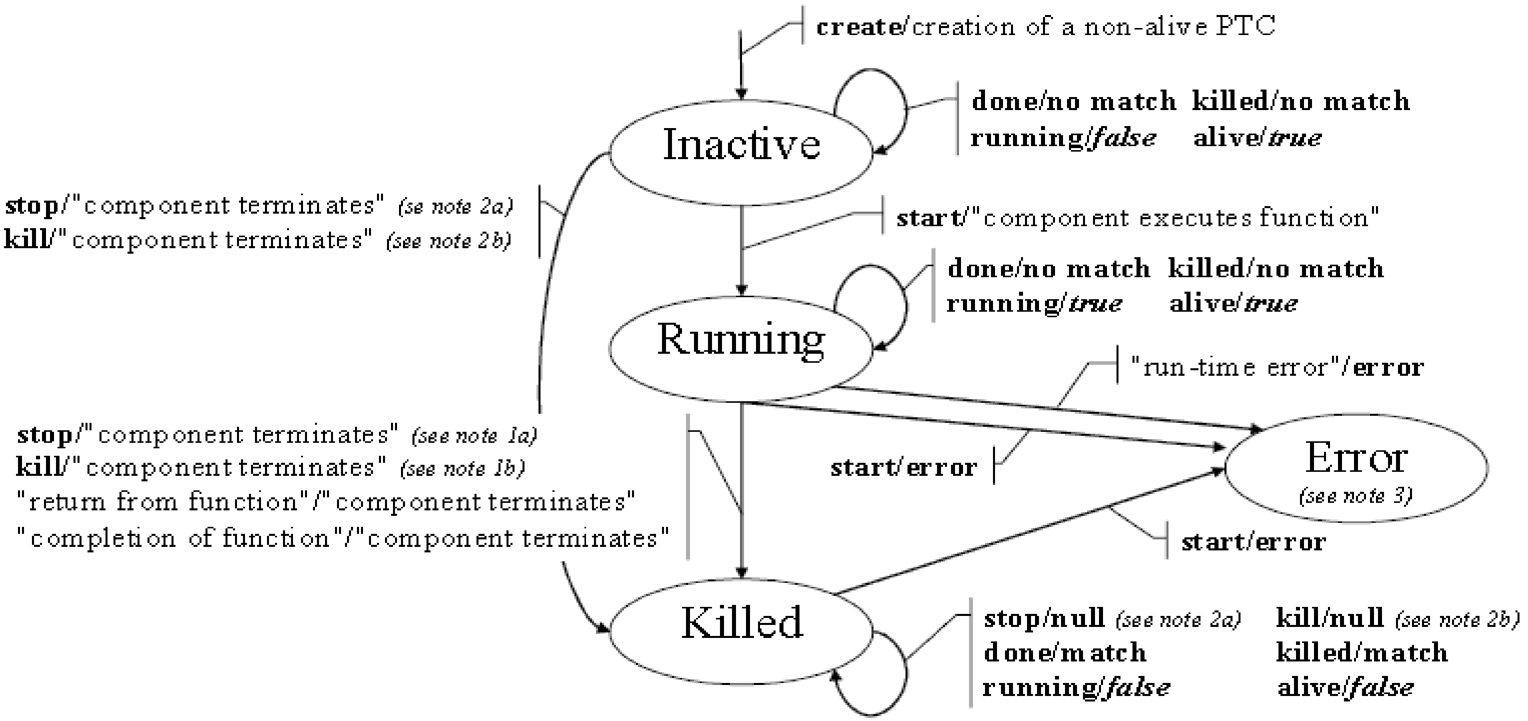
# CHECKING THE STATE OF A PARALLEL COMPONENT

- 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)

```
if(vc_A.running) { /*do something if vc_A is active!*/ }
while(any component.running) { /* do something if at least
                                 one component is running */ }


if(not vc_B.alive) { /*do something if vc_B not alive*/ }
vc_B.killed; // wait until vc_B alive component is killed
```

# PTC STATE MACHINE



**create**/creation of a non-alive PTC

**done/no match** **killed/no match**
**running/*false*** **alive/*true***

**stop**/"component terminates" *(se note 2a)*
**kill**/"component terminates" *(see note 2b)*

**start**/"component executes function"

**done/no match** **killed/no match**
**running/*true*** **alive/*true***

"run-time error"/**error**

**stop**/"component terminates" *(see note 1a)*
**kill**/"component terminates" *(see note 1b)*
"return from function"/"component terminates"
"completion of function"/"component terminates"

**start/error**

**start/error**

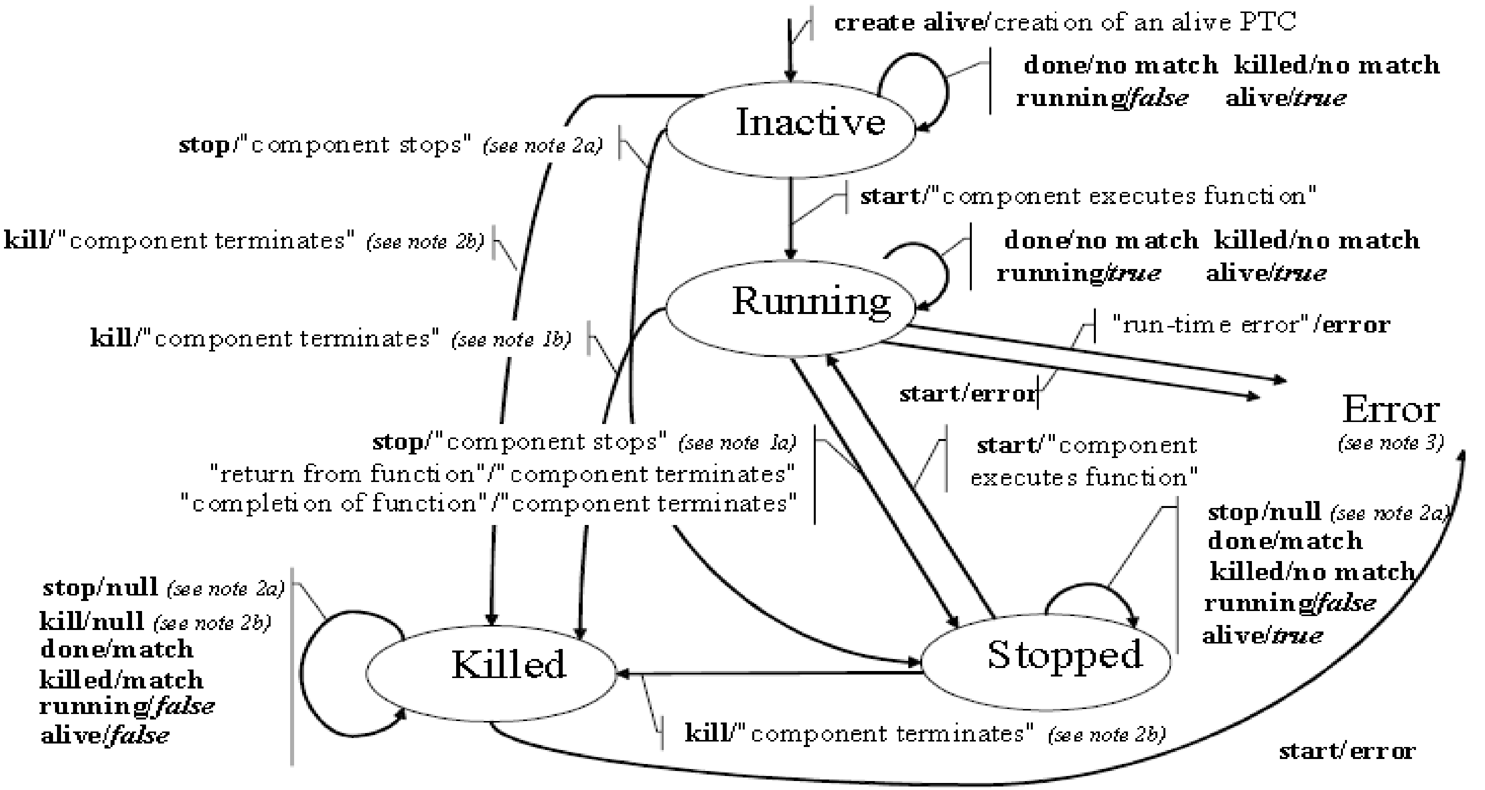
**stop/null** *(see note 2a)* **kill/null** *(see note 2b)*
**done/match** **killed/match**
**running/*false*** **alive/*false***

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

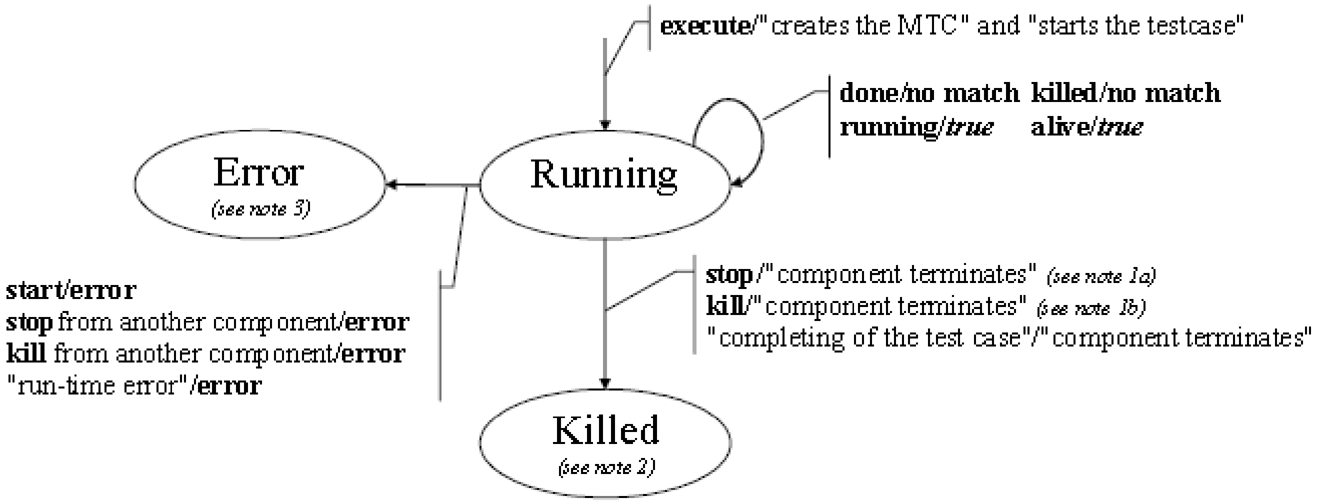

create alive/creation of an alive PTC
done/no match killed/no match
running/*false* alive/*true*
Inactive
stop/"component stops" (see note 2a)
start/"component executes function"
kill/"component terminates" (see note 2b)
done/no match killed/no match
running/*true* alive/*true*
Running
kill/"component terminates" (see note 1b)
"run-time error"/error
start/error
Error
(see note 3)
stop/"component stops" (see note 1a)
"return from function"/"component terminates"
"completion of function"/"component terminates"
start/"component executes function"
stop/null (see note 2a)
done/match
killed/no match
running/*false*
alive/*true*
stop/null (see note 2a)
kill/null (see note 2b)
done/match
killed/match
running/*false*
alive/*false*
Killed
Stopped
kill/"component terminates" (see note 2b)
start/error

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


execute/"creates the MTC" and "starts the testcase"
done/no match killed/no match
running/true alive/true
Error
(see note 3)
Running
Killed
(see note 2)
start/error
stop from another component/error
kill from another component/error
"run-time error"/error
stop/"component terminates" (see note 1a)
kill/"component terminates" (see note 1b)
"completing of the test case"/"component terminates"

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.

# SPECIAL FEATURES OF COMPONENT HANDLING

- The **running**, **alive**, **done**, **killed** and **stop** operations can be combined with the special **any component** or **all component** as well as with the **self** and **mtc** keywords

| Operation | any component | all component | self | mtc | system |
|---|---|---|---|---|---|
| running<br>alive | YES* | YES* | YES# | NO | NO |
| done<br>killed | YES* | YES* | YES# | NO | NO |
| stop<br>kill | NO | YES* | YES | YES | NO |

YES* = from MTC only!

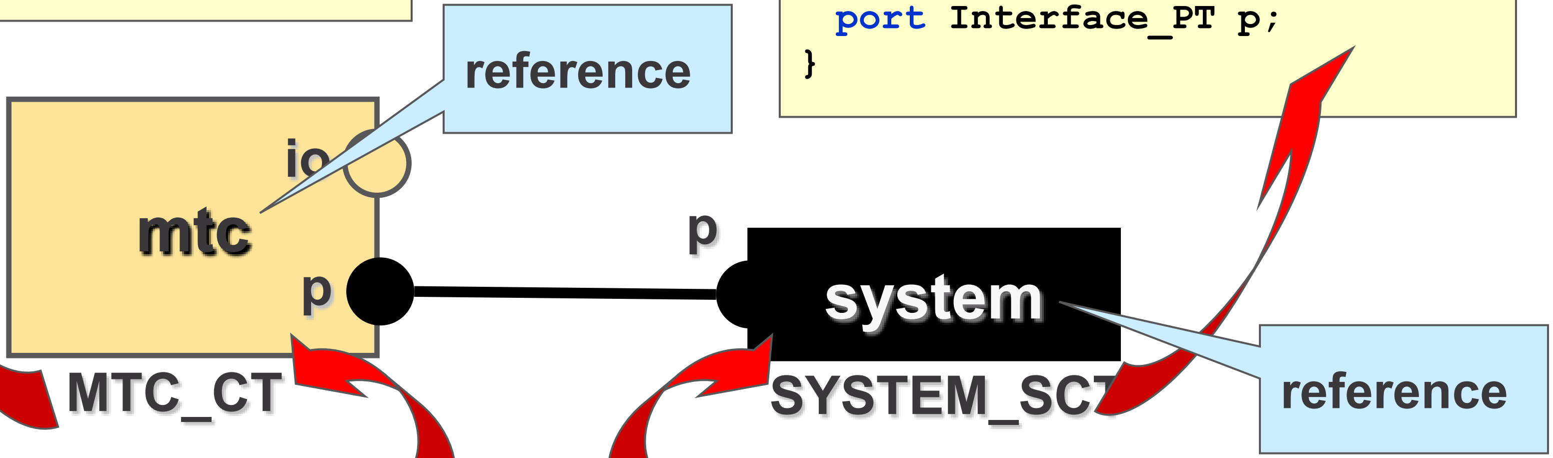
YES# = from PTCs only!

# RELATIONSHIP BETWEEN 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;
}
```



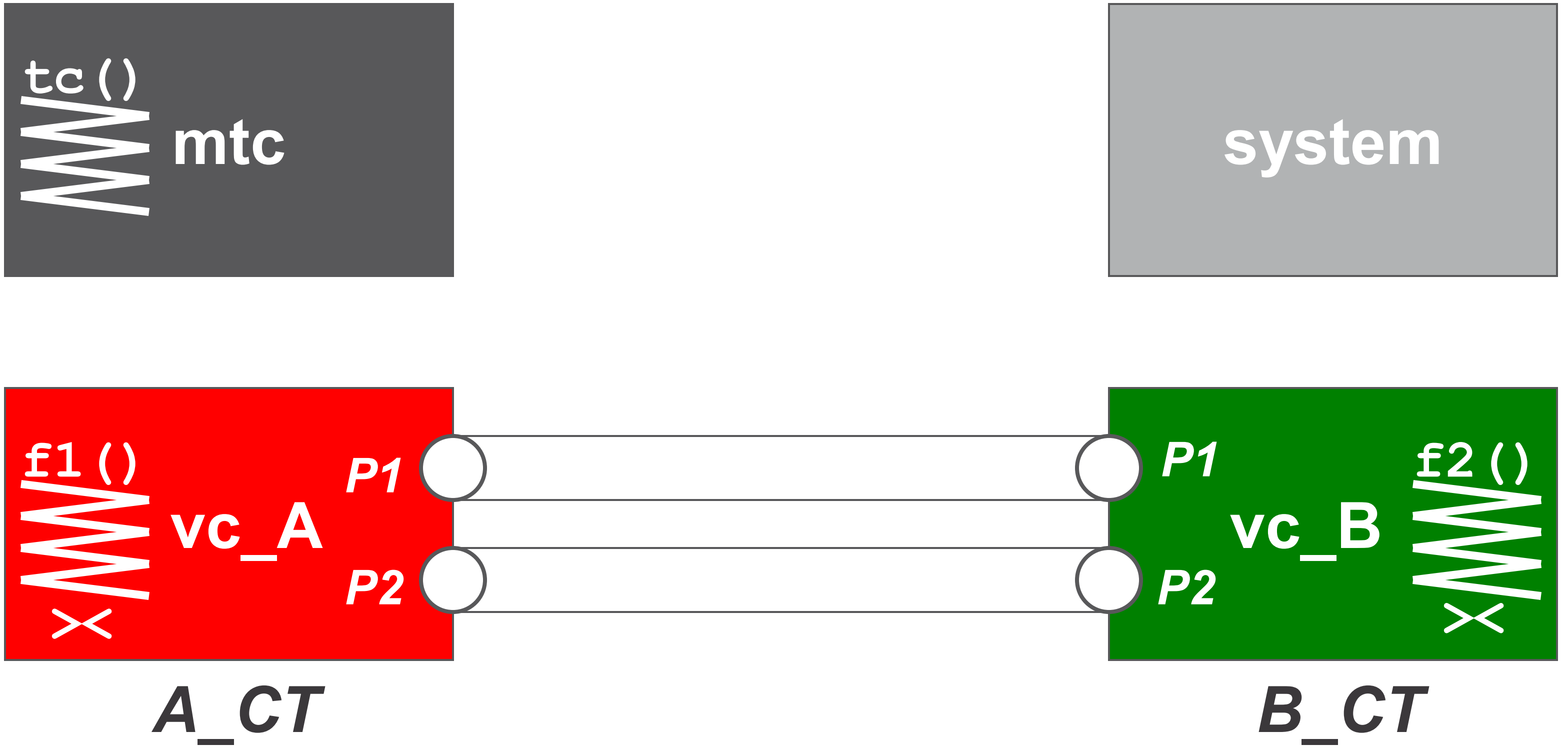
```
testcase tc_1() runs on MTC_CT system SYSTEM_SCT {
  map(mtc:p, system:p)
}
```

# ELEMENTARY STEPS OF SETTING UP THE TEST CONFIGURATION

1) Create PTCs (ports of components are created and started automatically) – `create`
2) Establish connections and mappings – `connect` or `map`
3) Start behavior on PTCs – `start`
4) Wait for PTCs to complete – `done` or `all component.done`

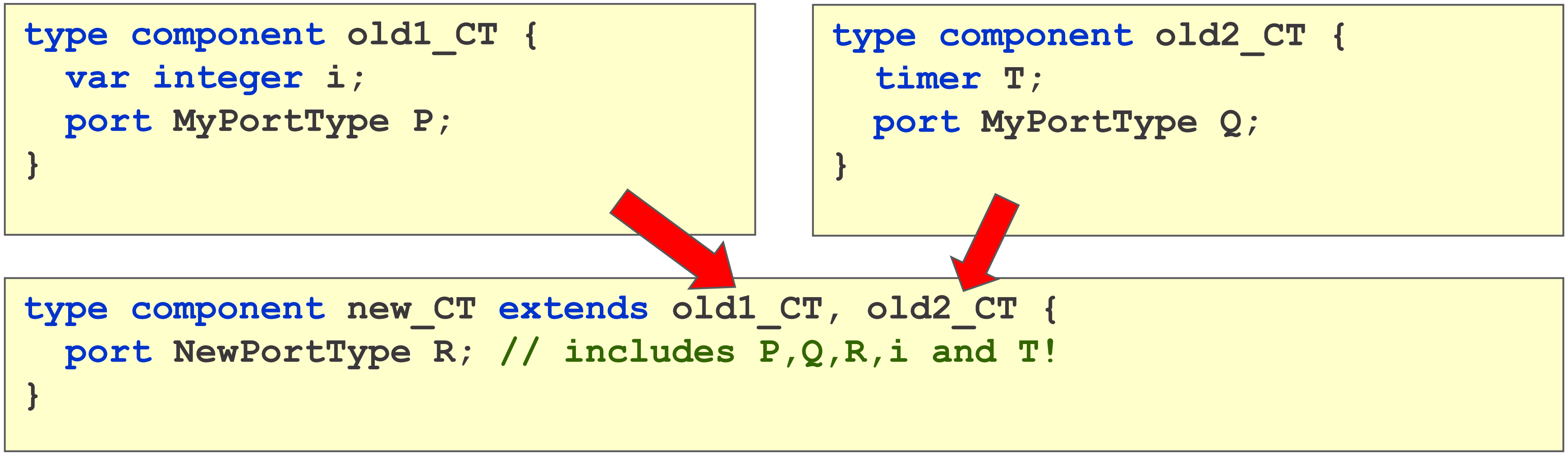
# EXAMPLE TEST CONFIGURATION

```
vc_B.done;
```



# EXTENDING COMPONENT TYPES

- Reuse of existing component type definitions:
  - “Derived” component type inherits all resources (ports, timers, variables, constants )of extended “parent” component type(s)
- Restrictions:
  - no cyclic extensions
  - avoid name clashes between different definitions



```
type component old1_CT {
  var integer i;
  port MyPortType P;
}
```

```
type component old2_CT {
  timer T;
  port MyPortType Q;
}
```

```
type component new_CT extends old1_CT, old2_CT {
  port NewPortType R; // includes P,Q,R,i and T!
}
```

# “RUNS ON-COMPATIBILITY”

- Function/altstep/testcase with “runs on” clause referring to an extended component type can also be executed on all derived component types

```
function f() runs on old1_CT {
  P.receive(integer:?) -> value i;
}
```

```
ptc := new1_CT.create;
ptc.start(f()); // OK: new1_CT is derived from old1_CT
```

# VISIBILITY MODIFIERS

- In component member definitions
  - `public` functions/testcases/altsteps running on that component can access the definition
  - `private` only the functions/testcases/altsteps  runs on the component type directly can access the definition which
  - `friend` modifier is not available within component types.

```
type component old1_CT {
  var integer i;
  public var charstrings v_char;
  private var boolean v_bool;
  port MyPortType P;
}
```

```
type component new_CT extends old1_CT
  {};

function f_set_int() runs on new_CT
  { i := 0 } //OK

function f_set_char() runs on new_CT
  { v_char := "a"} //OK

function f_set_bool() runs on new_CT
  {v_bool := true }
   //NOK, v_bool is private
```

# PORT CONTROL OPERATIONS

- Ports are automatically started at component creation and stopped when the component terminates (implicit stop)
- The **stop** operation shuts down the port (input queue contents are inaccessible) connections are *NOT* released!
- The **halt** operation blocks new incoming messages, but the messages in port queue remain intact and receivable
- The **clear** operation clears the port queue
- The **start** operation clears the queue and restarts the port

```
A_PCO.halt; //no new messages can get into port queue
A_PCO.stop; //no more activity on A_PCO
A_PCO.clear; //removes all messages from port queue
A_PCO.start; //clears port queue and restarts port
```

# SUMMARY OF CONFIGURATION OPERATORS (1)

| Operation | Keyword |
|---|---|
| Crete new parallel test component | `CT.create` |
| Create an alive component | `CT.create alive` |
| Connect two components | `connect(c1:p1,c2:p2)` |
| Disconnect two components | `disconnect(c1:p1,c2:p2)` |
| Connect (map) component to system | `map(c1:p1,c2:p2)` |
| Unmap port from system | `unmap(c1:p1,c2:p2)` |
| Get MTC address | `mtc` |
| Get test system interface address | `system` |
| Get own address | `self` |
| Start execution of test component | `ptc.start(f())` |

**Where `CT` is a component type definition; `ptc` is a PTC; `f()` is a function; `c`, `c1`, `c2` are component references and `p`, `p1`, `p2` are port identifiers**

# SUMMARY OF CONFIGURATION OPERATORS (2)

| Operation | Keyword |
|---|---|
| Check termination of a PTC | `ptc.running` |
| Check if a PTC is alive | `ptc.alive` |
| Stop execution of test component | `c.stop` |
| Kill an alive component | `c.kill` |
| Wait for termination of a test component | `ptc.done` |
| Wait for a PTC to be killed | `ptc.killed` |
| Start or restart port (queue is cleared!) | `p.start` |
| Stop port and block incoming messages | `p.stop` |
| Pause port operation | `p.halt` |
| Remove messages from the input queue | `p.clear` |

**Where `c` is a component reference; `ptc` is a PTC and `p` is a port identifier**

# 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 | Acceptable answer |
|---|---|
| TYPE: REQUEST | TYPE: RESPONSE |
| ID: 23 | ID: SAME as in REQ. |
| FROM: 231.23.45.4 | FROM: 230.x – 235.x |
| TO: 232.22.22.22 | TO: 231.23.45.4 |
| FIELD1: 1234 | FIELD1: 800-900 |
| FIELD2: ”Hello” | FIELD2: Do not care |

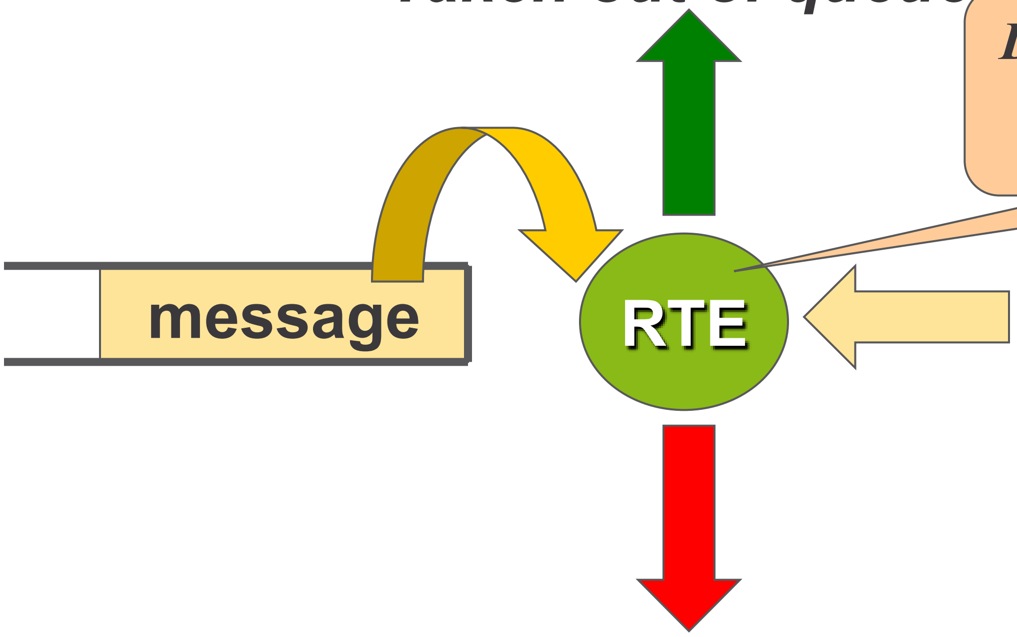
# DATA TEMPLATES

- A template is a pattern that specifies messages.

- A template for *sending* messages
  - may contain only specific values or `omit`;
  - usually specifies a message to be sent (but may also be received when the expected message does not vary).

- A template for *receiving* messages
  - describes all acceptable variants of a message;
  - contains matching attributes; these can be imagined as extended regular expressions;
  - *can be used only to receive:* trying to send a message using a receive template causes dynamic test case error.

# TEMPLATE MATCHING PROCEDURE



**Match ☺**
***Successful***
***Taken out of queue***

*Does the* **message** *match this* **template?**

**detailed description of the expected message**

**No match ☹**
***Unsuccessful***
***Remains in queue***

# 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
  }
```

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

# 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()`**

# 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
};
```

# 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
};
```

# 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");
```

# 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), 200, 255);

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

# 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!
};
```

# 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, * };
```

# CHARSTRING MATCHING – PATTERN

- Provides regular expression-based pattern matching for **`charstring`** and **`universal charstring`** values.
- Format: **`pattern`** <*charstring*>
  where <*charstring*> contains a TTCN-3 style regular expression.
- Patterns can be used in templates only.

```
// Matches charstrings with the first character "a"
// and the last one "z"
template charstring tr_0 := pattern "a*z";

// Match 3 character long strings such as AAC, ABC, …
template charstring tr_01 := pattern "A?C";
```

# PATTERN METACHARACTERS

- `?` Matches any single character
- `*` Matches any number of any character
- `#(n,m)` Repeats the preceding expression at least n but at most m times
- `#n` Repeats the preceding expression exactly n times
- `+` Repeats the preceding expression one or several times (postfix); the same as #(1,)
- `[]` Specifies character classes: matches any char. from the specified class
- `-` Hyphen denotes character range inside a class
- `^` Caret in first position of a class negates class membership

  e.g. `[^0-9]` matches any non-numerical character
- `()` Creates a group expression
- `|` Denotes alternative expressions
- `{}` Inserts and interprets the user-defined string as a regular expression
- `\` Escapes the following metacharacter, e.g. `\\` escapes `\`
- `\d` Matches any numerical digit, equivalent to `[0-9]`
- `\w` Matches any alphanumeric character, equivalent to `[0-9a-zA-Z]`
- `\t` TABULATOR, `\n` NEWLINE, `\r` CR, `\”` DOUBLE QUOTE
- `\q{<group>, <plane>, <row>, <cell>}`
- Matches the universal character specified by the quadruple

# SAMPLE PATTERNS

- Set expression

```
// Matches any charstring beginning with a capital letter
template charstring tr_1 := pattern "[A-Z]*";
```

- Reference expression

```
// Matches 3 characters long charstrings like "AxB"
var charstring cg_in := "?x?";
template charstring tr_2 := pattern "{cg_in}";
```

- Multiple match

```
// Matches a string containing at least 3 at most 5 capitals
template charstring tr_4 := pattern "[A-Z]#(3,5)";

// Matches any ASN.1 type name
template charstring tr_3 :=
                  pattern "[A-Z](-#(,1)\w#(1,))#(,)";
```

# THE FUNCTION REGEXP()

- **function regexp** (*<input-string>*, *<regexp>*, *<group-number>*)
- **return** *<type of input-string>*;
- returns a substring of <*input-string*>, which is the content of
- (*<group-number>* + *1)*th group matching the <*regexp*>
- <*input-string*> type can be any (**universal**) **charstring**
- the type of returned value equals to the type of the input string

```
control {
 var charstring v_string := "0036 (1) 737-7698";
 var charstring v_regexp :=
        "0036 #(,)\((\d#(1,))\) #(,)[\d-]#(1,)";
 var charstring v_result := regexp(v_string, v_regexp, 0);
} // v_result contains the number in parentheses, i.e. 1
```

# MATCHING MECHANISMS (2)

- Value attributes on field level:
  - `length` restriction;
  - `ifpresent` modifier.

- Special matching for `set of` types:
  - `subset` and `superset` matching.
- Special matching for `record of` types:
  - `permutation` matching.

- Predefined functions operating on templates:
  - `match()`
  - `valueof()`

# LENGTH RESTRICTION

- Matches values of specified length – length can be a range.
- The unit of length is determined by the template’s type.
- Permitted only in conjunction with other matching mechanism (e.g. `?` or `*`)
- Applicable to all basic string types and record-of/set-of types

```
// Any value template with length restriction
template charstring tr_FourLongCharstring := ? length(4);
// type record of integer ROI;
template ROI tr_One2TenIntegers := ? length(1..10);
```

```
// Standalone length modifier is not allowed!
template bitstring tr_ERROR := length(3); // Parse error!!!
```

# PRESENCE ATTRIBUTE – IFPRESENT

- Used together with an other matching mechanism for constraining, **ifpresent** can be applied only to **optional** fields.
- Operation mode:
  - Absent optional field (**omit**) → always match
  - Present optional field → other matching mechanism decides matching
- Presence attribute makes sense with all matching mechanisms except **?** and *****  (***** is equivalent to **? ifpresent**)

```
// Presence attribute with structured type fields
template MyRecordType tr_IfpresentExample := {
 field1 := complement (1, 101, 201) ifpresent,
 field2 := ?
};
```

# SUBSET AND SUPERSET TEMPLATES

- Applicable to **set of** types only.
- **subset** matches if all elements of the incoming field are defined in the subset

```
type set of integer SOI;
template SOI tr_SOIb := subset ( 1, 2, 3 );
// Matches {1,3,2} and {1,3}
// Does not match {4,3,2} and {0,1,2,3,4}
```

- **superset** matches if all elements of the defined superset can be found in the incoming field

```
template SOI tr_SOIp := superset ( 1, 2, 3 );
// Matches {1,3,1,2} and {0,1,2,3,4}
// Does not match {1,3}(2 is missing) and {4,3,2}(1 is missing)
```

# PERMUTATION

- Applicable to **record of** types only
- **permutation** matches all permutations of enlisted elements (i.e. the very same elements enlisted in any order)

```
type record of integer ROI;
template ROI tr_ROIa := { permutation ( 1, 2, 3 ) };
// Matches {1,3,2} and {2,1,3}
// Does not match {4,3,2}, {0,1,2,3} and {1,2}(3 is missing)
```

# MATCHING AND TYPES

| What kind of matching mechanisms are applicable to which types?<br>Y = permitted<br>N = not applicable | Specific value, omit | Value list, complemented | Any value, any value or none | Range | Subset, superset | Permutation | Any element, any elements or none | Length restriction | ifpresent |
|---|---|---|---|---|---|---|---|---|---|
| `boolean` | Y | Y | Y | N | N | N | N | N | Y |
| `integer, float` | Y | Y | Y | Y | N | N | N | N | Y |
| `bitstring, octetstring, hexstring` | Y | Y | Y | N | N | N | Y | Y | Y |
| `charstring,`<br>`universal charstring` | Y | Y | Y | Y | N | N | Y | Y | Y |
| `record, set, union, enumerated` | Y | Y | Y | N | N | N | N | N | Y |
| `record of` | Y | Y | Y | N | N | Y | Y | Y | Y |
| `set of` | Y | Y | Y | N | Y | N | Y | Y | Y |

# 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!!
}
```

# TEMPLATES ARE NOT VALUES

- Value types 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 templates vs. general (receive) templates

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

- 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
valueof(t2) == v // error:t2 cannot be converted to a value
match(mp,t2) == true // true: mp matches t2
```

# 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: receive any value of a given type
   port1_PCO.receive(BCCH_MESSAGE:?);


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

# MODIFIED TEMPLATES

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

# INLINE MODIFIED TEMPLATES

- Defined directly in the communication operation
- Valid only for that one operation (No identifier, no reusability)
- Can not be parameterized
- Usually ineffective, not recommended to use!

```
template MyRecordType t_1 := {
 field1 := omit,
 field2 := false
}
control {
 …
 port_PCO.receive(modifies t_1 := { field1 := * } );
 …
}
```

# 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)

- 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 PARAMETERIZATION (3)

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

```
// Template-type parameterization
template integer tr_Int := ( (3..6), 88, 555) );
template MyIEType tr_TemplPm(template integer pl_int) :=
  { f1 := 1, f2 := pl_int }

// Can be used:
P1_PCO.send(tr_ TemplPm( 5 ) );
P1_PCO.receive (tr_ TemplPm( ? ) );
P1_PCO.receive (tr_ TemplPm( tr_Int ) );
P1_PCO.receive (tr_ TemplPm( (3..55) ) );
P1_PCO.receive (tr_ TemplPm( complement (3,5,9) );
```

**Note the `template` keyword!**

# RESTRICTED TEMPLATES

Templates can be restricted to

- `(omit)` evaluate to a specific value or `omit`
- `(present)` evaluate to any template except `omit`
- `(value)` specific value (i.e. the entire template must not be `omit`)

Applicable to any kind of templates (i.e. template definitions, variable templates and template formal parameters)

| | `template (omit)` | `template (present)` | `template (value)` |
|---|---|---|---|
| `omit` | `Ok` | `error` | `error` |
| **Specific value template** | `Ok` | `Ok` | `Ok` |
| **Receive template** | `error` | `Ok` | `error` |

```
function f_omit(template (omit) integer p) {}
function f_present(template (present) integer p) {}
function f_value(template (value) integer p) {}
```

# RESTRICTED TEMPLATE EXAMPLES

```
// omit restriction
function f_omit(template (omit) integer p) {}
f_omit(omit); // Ok
f_omit(integer:?); // Error
f_omit(1); // Ok
// present restriction
function f_present(template (present) integer p) {}
f_present(omit); // Error: omit is excluded
f_present(integer:?); // Ok
f_present(1); // Ok
// value restriction
function f_value(template (value) integer p) {}
f_value(omit); // Error: entire argument must not be omit
f_value(integer:?); // Error: not value
f_value(1); // Ok
```

# TEMPLATE VARIABLES

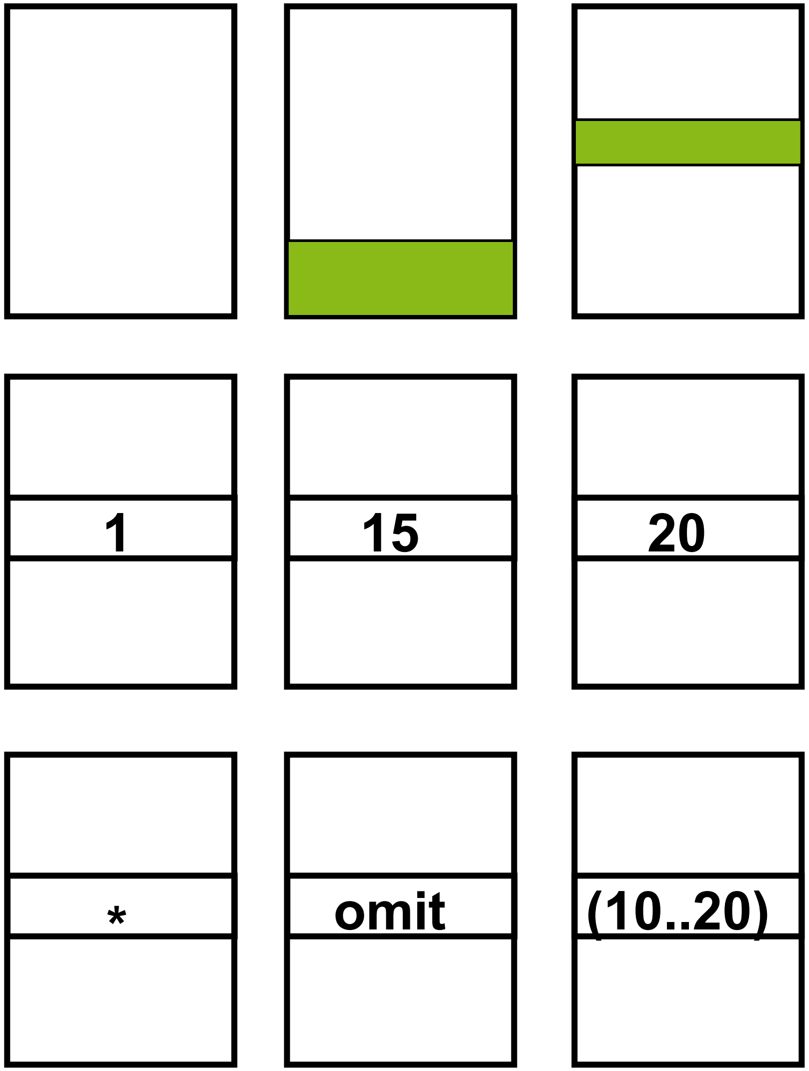
- Templates can be stored in so called template variables
- Template variable
  - may change its value several times
  - assignment and access to its elements are permitted (e.g. reference and index notation permitted)
  - must not be an operand of any TTCN-3 operators

```
control {
 var template integer vt := ?;
 var template MySetType vs :=
      { field1:= ?, field2 := true};
 vt := (1,2,3); // Ok
 vs.field1 := 2; // Ok
}
```

# 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

# TEMPLATE HIERARCHY – TYPICAL SITUATIONS



modified template

parametrized template

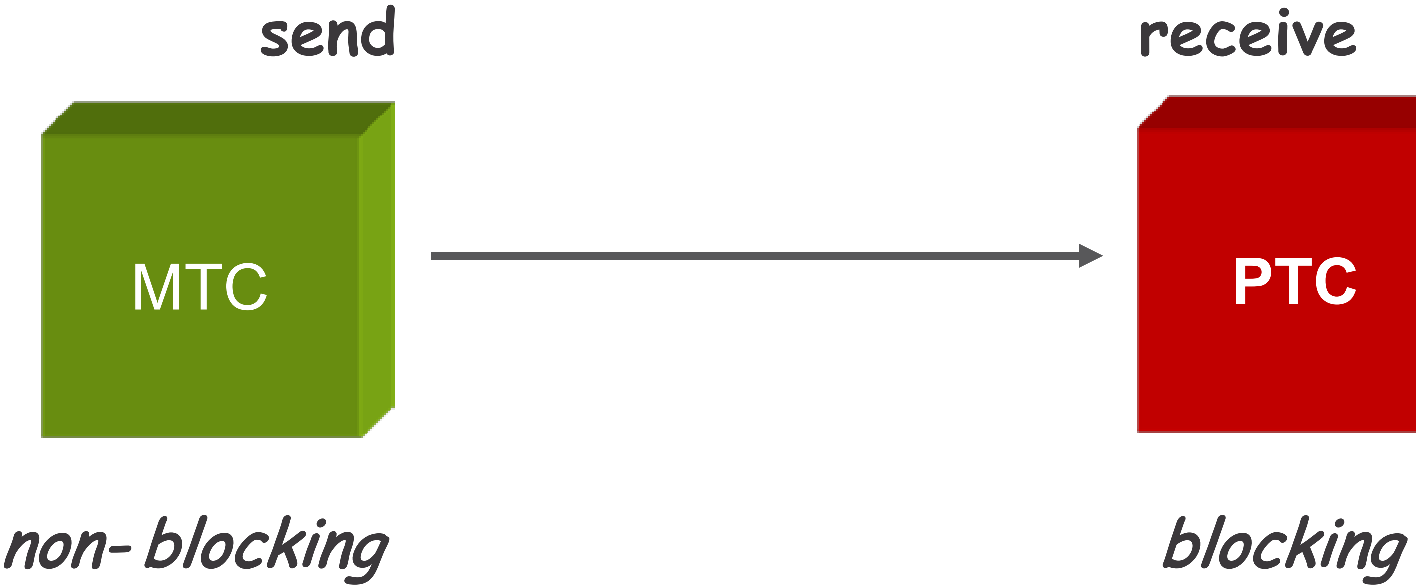
template parameter

# XIII. ABSTRACT COMMUNICATION OPERATIONS

ASYNCHRONOUS COMMUNICATION
SEND, RECEIVE, CHECK 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

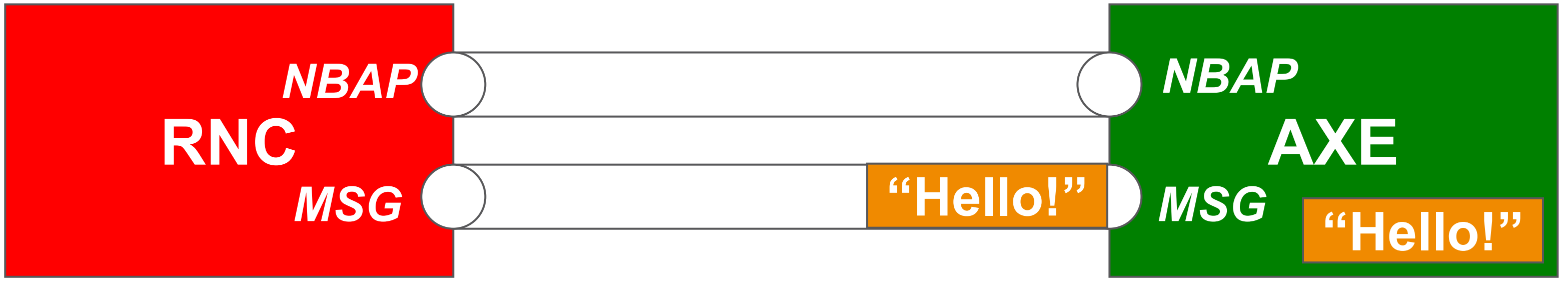
- *<PortId>*`.`**`receive`**`(`*<TemplateRef>*`)` or *<PortId>*`.`**`receive`**

  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

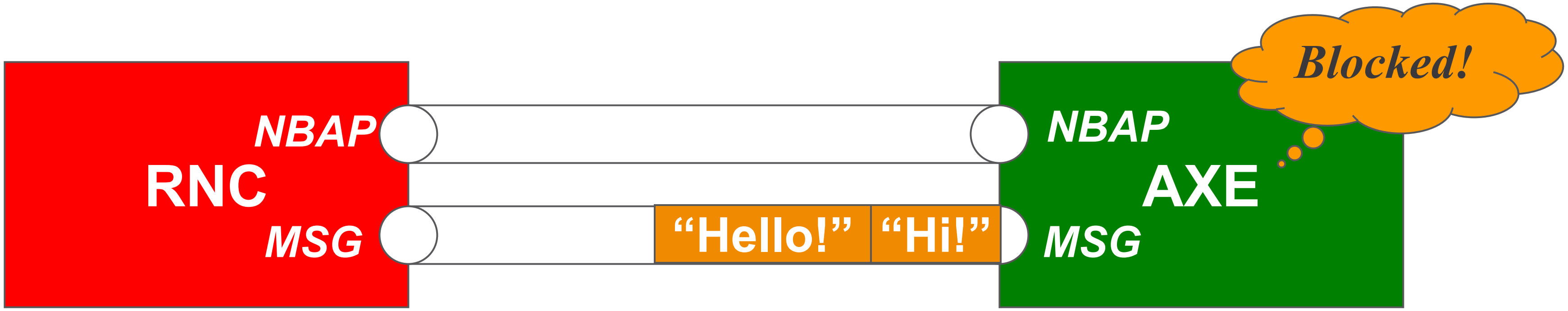
- Send and receive operations can be used only on connected ports
  - Sending or receiving on a port, which has neither connections nor mappings results in test case error
- The send operation is non-blocking
- The receive operation has blocking semantics
  (except if it is used within an alt or an interleave 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 of the port's queue, but extracts this *only if* the message matches the receive operation's template

# SEND AND RECEIVE EXAMPLES



```
MSG.send("Hello!");
```

```
MSG.receive("Hello!");
```



```
MSG.send ("Hi!");
MSG.send("Hello!");
```

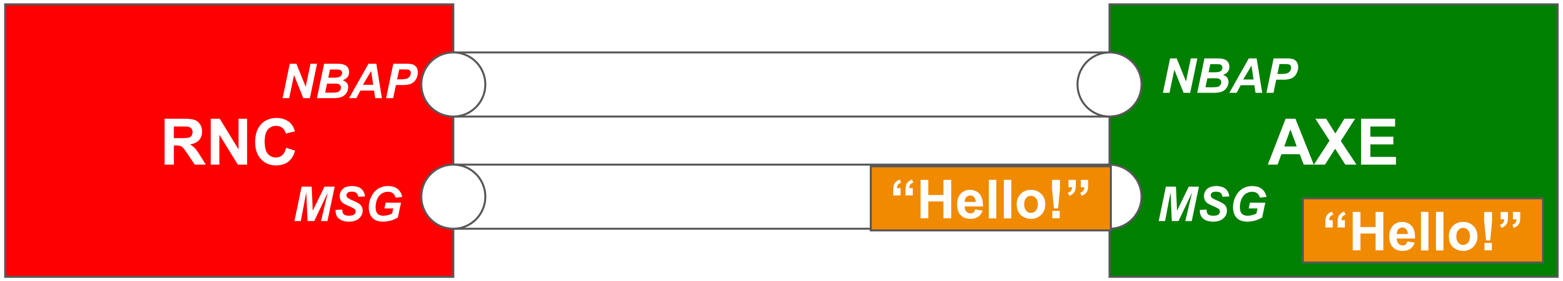
```
MSG.receive("Hello!");
```

# CHECK-RECEIVE AND TRIGGER VS. RECEIVE

- Check-receive operation blocks until a message is present in the port's queue, then it decides, if the 1st message of the port's queue matches our template or not;
  The message itself remains untouched on the top of the queue!
  - Usage:

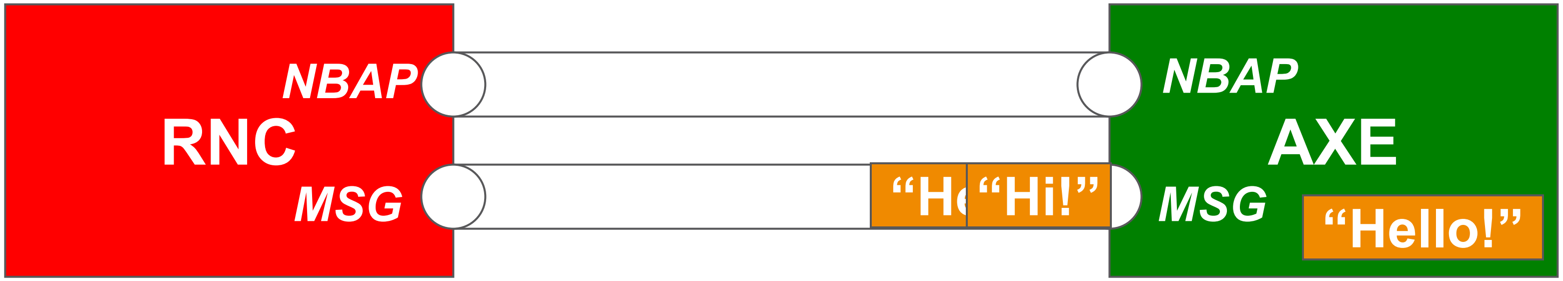
    ```
    <PortId>.check(receive(<TemplateRef>));
    <PortId>.check;
    any port.check;
    ```
- Trigger operation blocks until a message is arrived into the port's queue and extracts the 1st message from the queue:
  - If the top message meets the matching criteria → works like receive
  - Otherwise the message is dropped without any further action
  - Usage:
  - `<PortId>.trigger(<TemplateRef>);`
  - `<PortId>.trigger;` (equivalent to `<PortId>.receive;`)

# TRIGGER EXAMPLES



```
MSG.send("Hello!");
```

```
MSG.trigger("Hello!");
```



```
MSG.send ("Hi!");
MSG.send("Hello!");
```

```
MSG.trigger("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
PortRef.receive(MsgTemplate) -> sender Peer;
// extract MsgType message and save it with its sender
PortRef.trigger(MsgType:?) -> value MsgVar sender Peer;
```

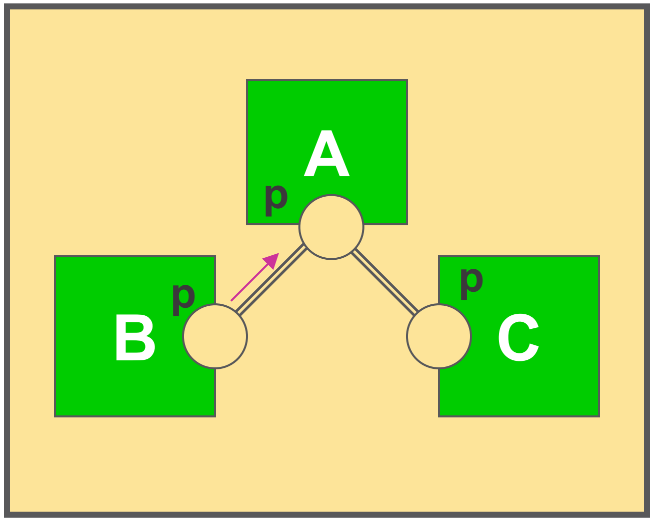
# SEND TO AND RECEIVE FROM

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

```
p.receive(TemplateRef) from B;
```

- How does component A send a message to system?

```
p.send(Msg) to C;
```



```
//send a reply for the previous message
p.receive (Request_Msg) -> sender CompVar;
p.send(Msg) to CompVar;
```

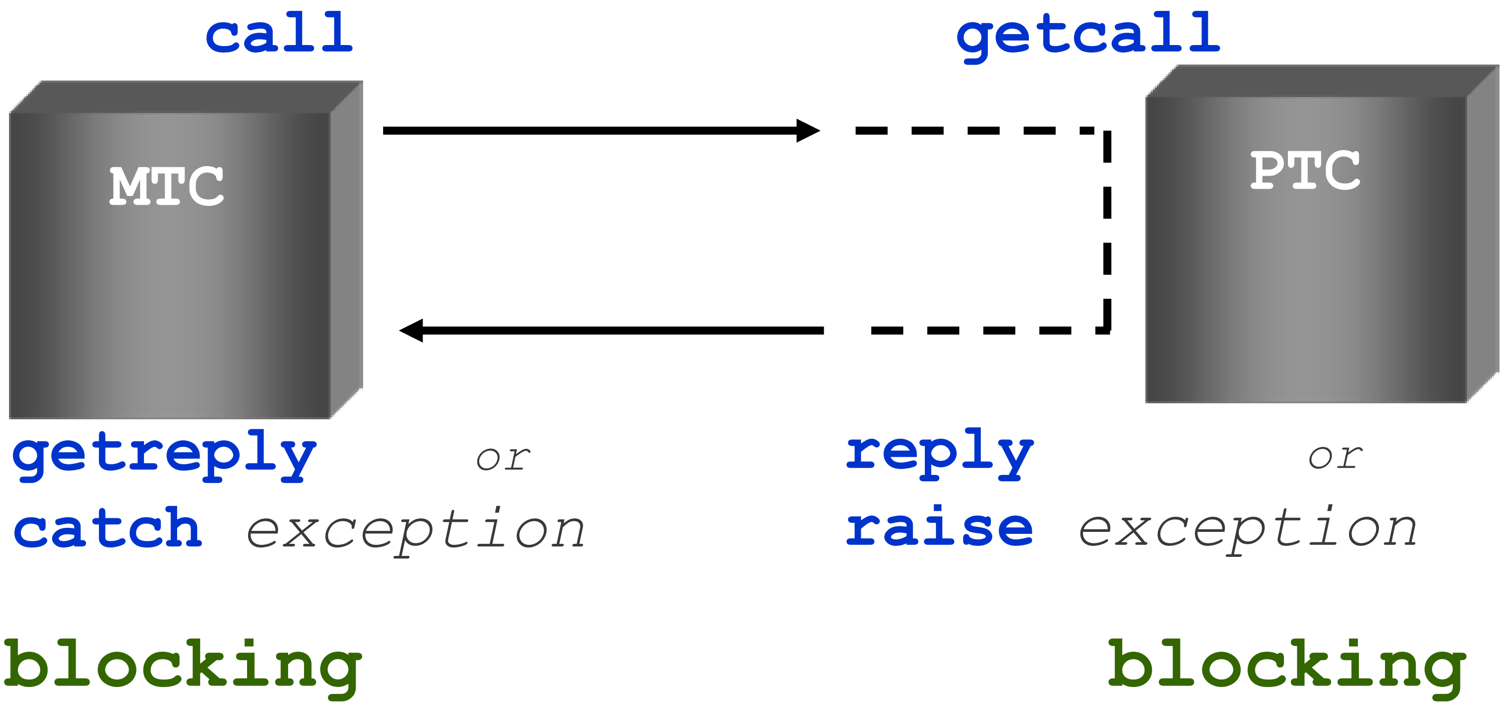
# EXAMPLES OF ASYNCHRONOUS COMMUNICATION OPERATIONS

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

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

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

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

any port.receive;

MyPort_PCO.check(receive(A < B)) from MyPeer;

MyPort_PCO.trigger(5) -> sender MyPeer;
```

# 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` |

# SYNCHRONOUS COMMUNICATION



call

getcall

MTC

PTC

getreply *or*
catch *exception*

reply *or*
raise *exception*

blocking

blocking

# EXAMPLES OF SYNCHRONOUS COMMUNICATION OPERATIONS

```
signature MyProc3 (out integer MyPar1, inout boolean MyPar2)
       return integer
       exception (charstring);
// Call of MyProc3
MyPort.call(MyProc3:{ -, true }, 5.0) to MyPartner {
  [] MyPort.getreply(MyProc3:{?, ?}) -> value MyResult param
                              (MyPar1Var,MyPar2Var) { }
  [] MyPort.catch(MyProc3, “Problem occured”) {
              setverdict(fail); stop; }
 [] MyPort.catch(timeout) {
              setverdict(inconc); stop; }
}
// Reply and exception to an accepted call of MyProc3
MyPort.reply(MyProc3:{5,MyVar} value 20);
MyPort.raise(MyProc3, “Problem occured”);
```

# SUMMARY OF SYNCHRONOUS COMMUNICATION OPERATIONS

| Operation | Keyword |
|---|---|
| Invoke (remote) procedure call | `call` |
| Reply to a (remote) procedure call | `reply` |
| Raise an exception | `raise` |
| Accept (remote) procedure call | `getcall` |
| Handle response from a previous call | `getreply` |
| Catch exception (from called entity) | `catch` |
| Check reply or exception | `check` |

# 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
```

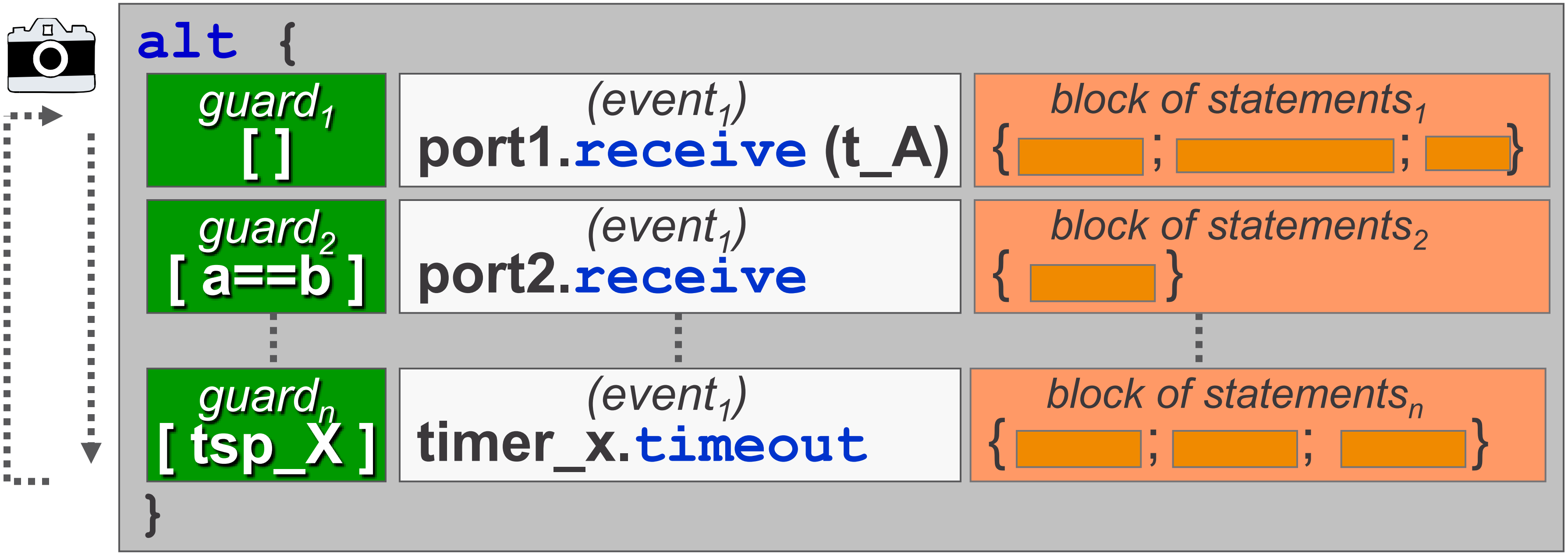
# ALTERNATIVE EXECUTION BEHAVIOR EXAMPLES

- 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 */ }
}
```

# 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



alt {

*guard$_1$* [ ] *(event$_1$)* port1.receive (t_A) *block of statements$_1$* { ; ; }

*guard$_2$* [ a==b ] *(event$_1$)* port2.receive *block of statements$_2$* { }

*guard$_n$* [ tsp_X ] *(event$_1$)* timer_x.timeout *block of statements$_n$* { ; ; }

}

# FORMAT OF ALTERNATIVES

- Guard condition enables or disables the alternative:
  - Usually empty: `[]` equivalent to `[true]`
  - Can contains a condition (boolean expression): `[x > 0]`
  - Occasionally the else keyword: `[else]` → `else` branch
    - but it makes the semantics completely different!
- Blocking operation (event):
  - Any of `receive`, `trigger`, `getcall`, `getreply`, `catch`, `check`, `timeout`, `done` or `killed`
  - `altstep` invocation → `altstep`
  - May be empty only in `[else]` guard
- Statement block:
  - Describes actions to be executed on event occurrence
  - Optional: can be empty ( i.e. `{}` 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 “the same 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!

# 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 :

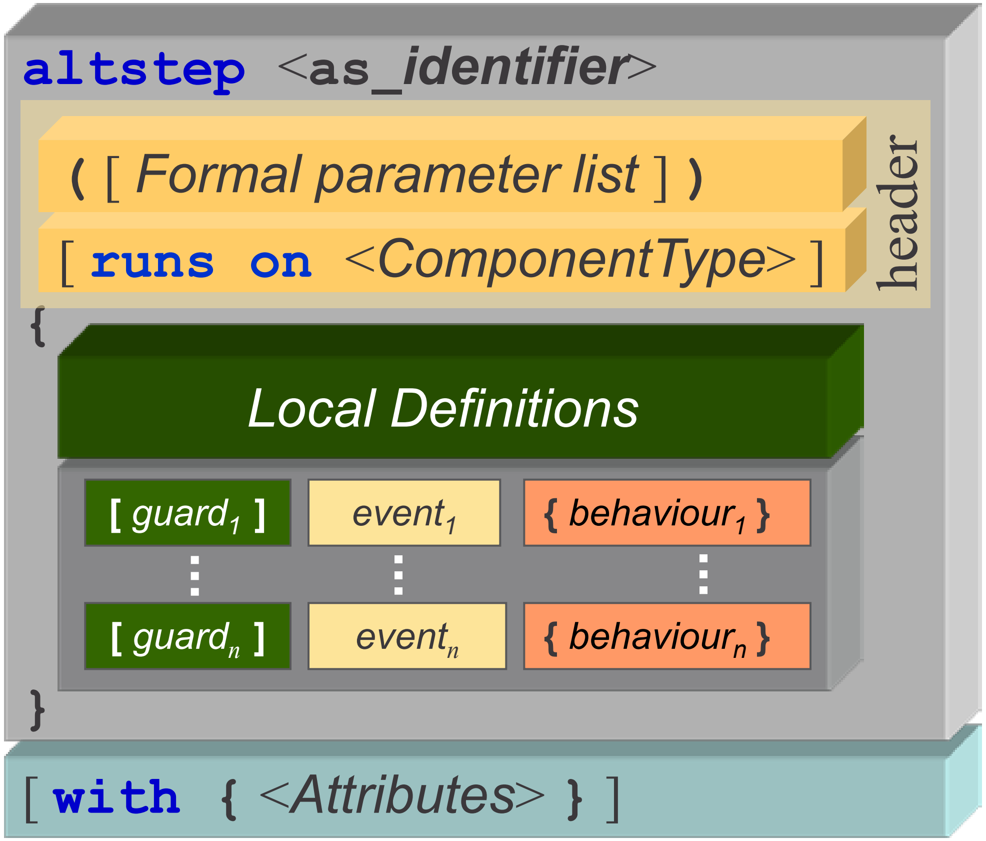
```
P.send(req)
T.start;
// …
alt {
[] P.receive(resp)   { /* actions to do and exit alt */ }
[] P.receive(keep_alive) { /* handle keep alive message */
                            repeat }
[] any port.receive { /* handle unexpected event */ }
[] T.timeout        { /* handle timer expiry and exit */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

# THE ELSE GUARD

- Guard contains else and blocking event is absent
- Execution continues with the else branch, when none of the previous alternatives satisfied at first snapshot
- Consequently, an alt with else:
  - takes only a single snapshot → never blocks execution
  - does not wait for any declared event to happen
  - goes on immediately with the actions of the event, which happened before taking the snapshot or jumps to statement block of else branch

```
alt { // 1 snapshot is taken here
[] A.receive(x) { /* extract x if available in A */ }
[] any port.receive { /* remove anything */ }
[else] { /* continue here when none of above applied */ }
} // end of alt
```

# STRUCTURING ALTERNATIVE BEHAVIOR – ALTSTEP



**altstep** <as_*identifier*>

( [ *Formal parameter list* ] )

[ **runs on** <*ComponentType*> ]

header

{

*Local Definitions*

[ *guard$_1$* ] *event$_1$* { *behaviour$_1$* }

⋮ ⋮ ⋮

[ *guard$_n$* ] *event$_n$* { *behaviour$_n$* }

}

[ **with** { <*Attributes*> } ]

- 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 WAYS TO USE ALTSTEP

- Direct invocation:
  - Expands dynamically to an `alt` statement

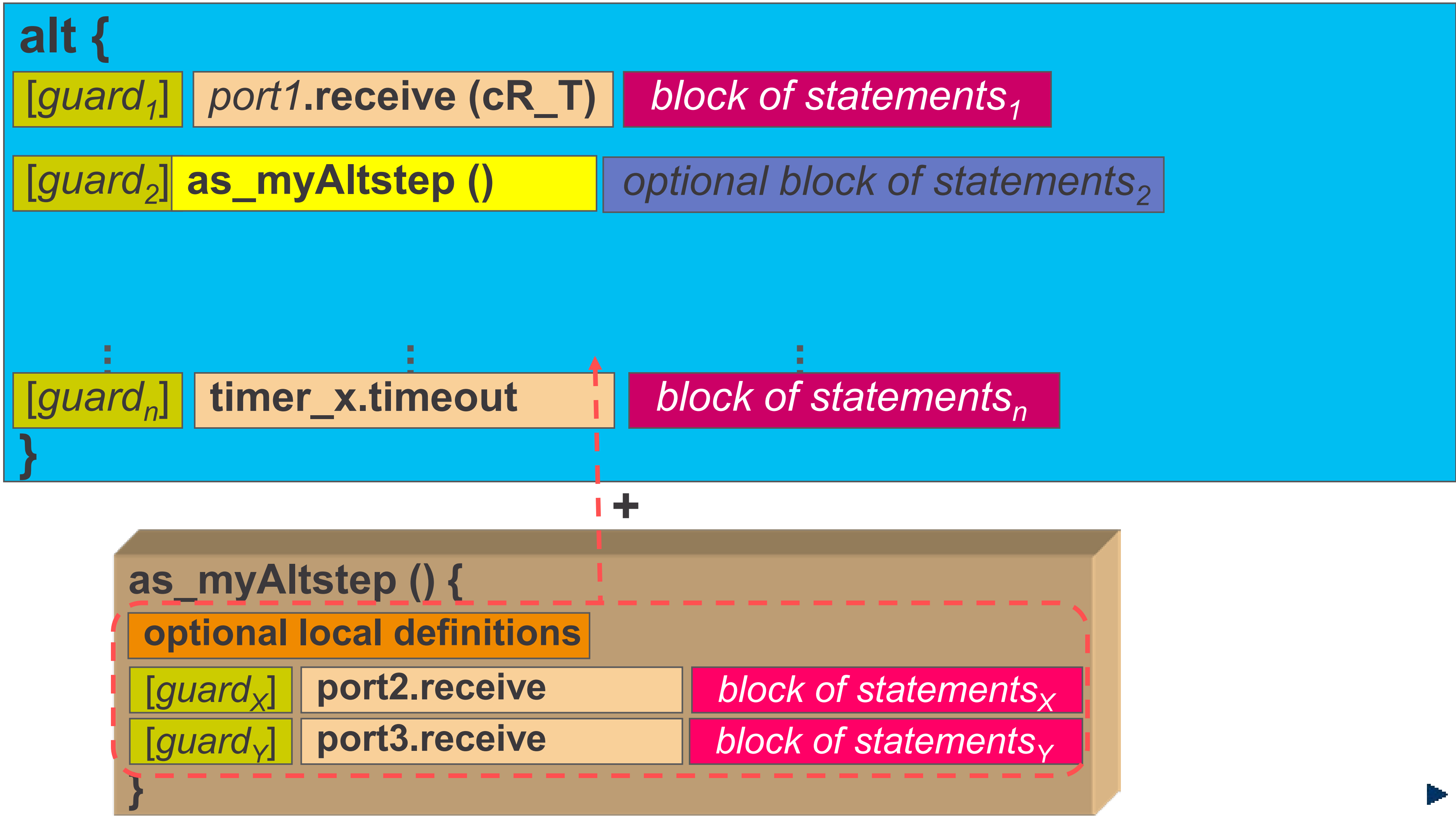
- Dynamic invocation from alt statement:
  - Attaches further alternatives to the place of invocation

- Default activation:
  - Automatic attachment of activated `altstep` branches to the end of each `alt`/blocking operation

# USING 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…
}

// … has the same effect as
testcase tc_101() runs on My_CT {
  alt {
  [] PCO.receive(4) {…}
  [] PCO.receive(tr_Msg) {…}
  }
}
```

# USING ALTSTEP – INVOCATION IN ALT



**alt {**

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

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

⋮ ⋮ ⋮

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

**}**

**+**

**as_myAltstep () {**

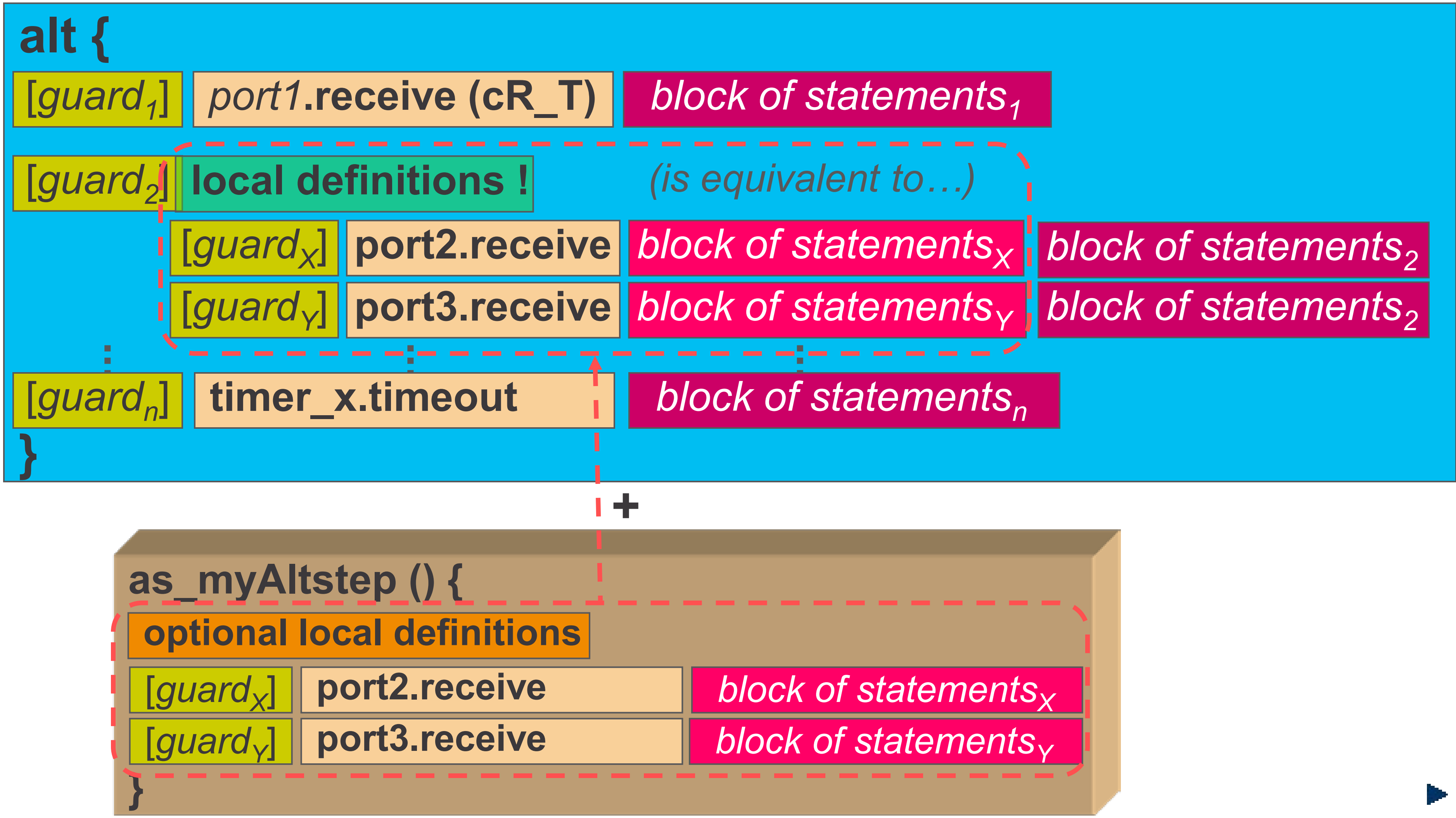
**optional local definitions**

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

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

**}**

# USING ALTSTEP – INVOCATION IN ALT



alt {

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

[*guard*$_2$] local definitions ! *(is equivalent to…)*

[*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*$_Y$

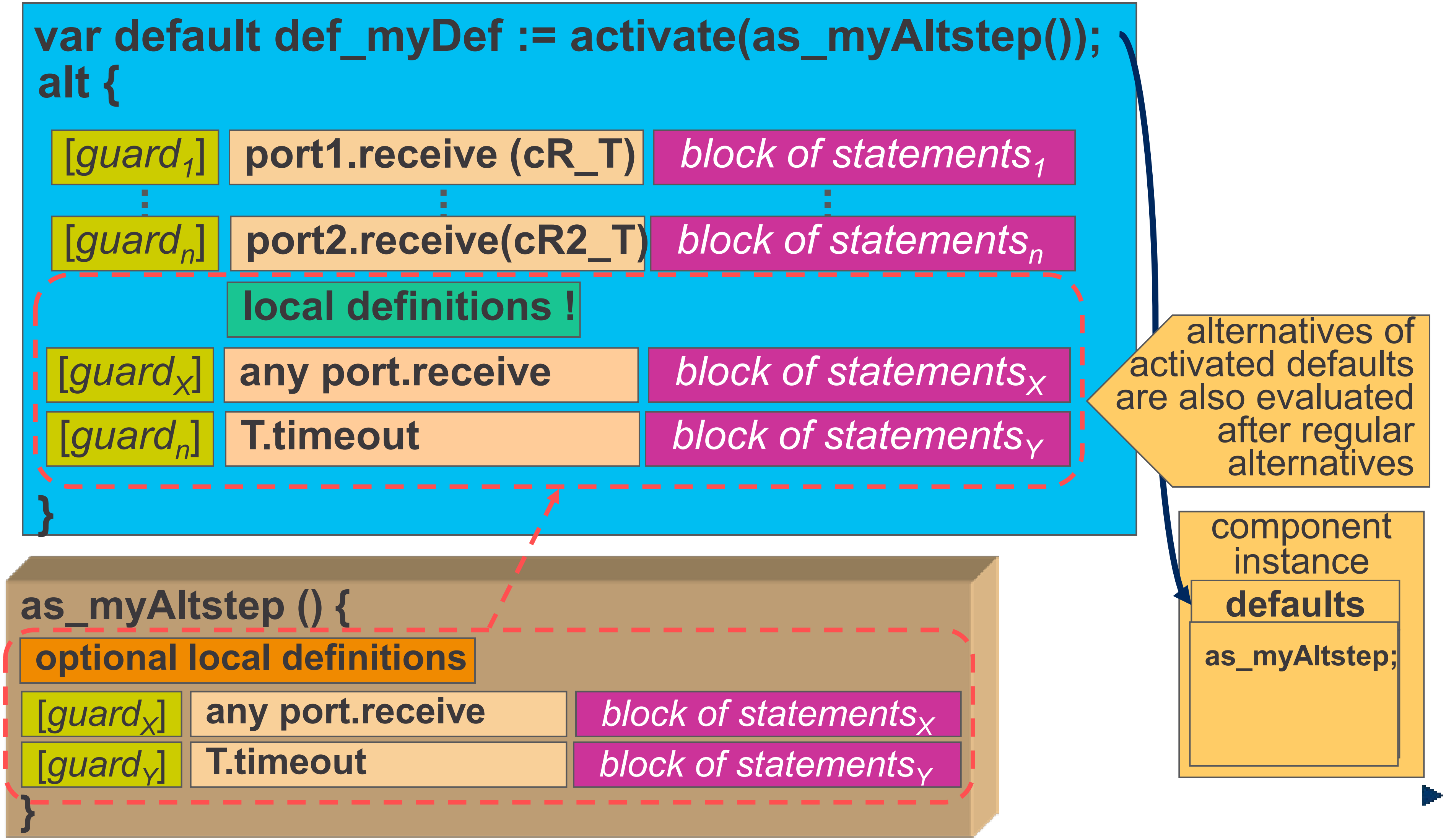
}

# MOTIVATION - DEFAULTS

- Error handling at the end of each **alt** instruction
  - Collect these alternatives into an **altstep**
  - Activate as **default**
  - Automatically copied to the end of each **alt**

```
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) }
}
```

# USING ALTSTEP – ACTIVATED AS DEFAULT



var default def_myDef := activate(as_myAltstep());
alt {

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

[*guard*$_n$] port2.receive(cR2_T) *block of statements*$_n$

local definitions !

[*guard*$_X$] any port.receive *block of statements*$_X$

[*guard*$_n$] T.timeout *block of statements*$_Y$

}

alternatives of activated defaults are also evaluated after regular alternatives

component instance

**defaults**

as_myAltstep;

as_myAltstep () {

optional local definitions

[*guard*$_X$] any port.receive *block of statements*$_X$

[*guard*$_Y$] T.timeout *block of statements*$_Y$

}

# ACTIVATION OF ALTSTEP TO DEFAULTS

- Altsteps can be used as default operations:
  - **activate**: appends an **altstep** with given actual parameters to the current default context, returns a unique default reference
  - **deactivate**: removes the given default reference from the context

```
altstep as1() runs on CT {
[] any port.receive { setverdict(fail)}
[] any timer.timeout { setverdict(inconc)}
}

var default d1:= activate(as1());
 ...
deactivate(d1);
```

- Defaults can be 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 be entirely controlled run-time
- Defaults have no effect within an alt, which contains an else guard!

# STANDALONE RECEIVING STATEMENTS VS. ALT

- 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 (!!)

- 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) }
}
```

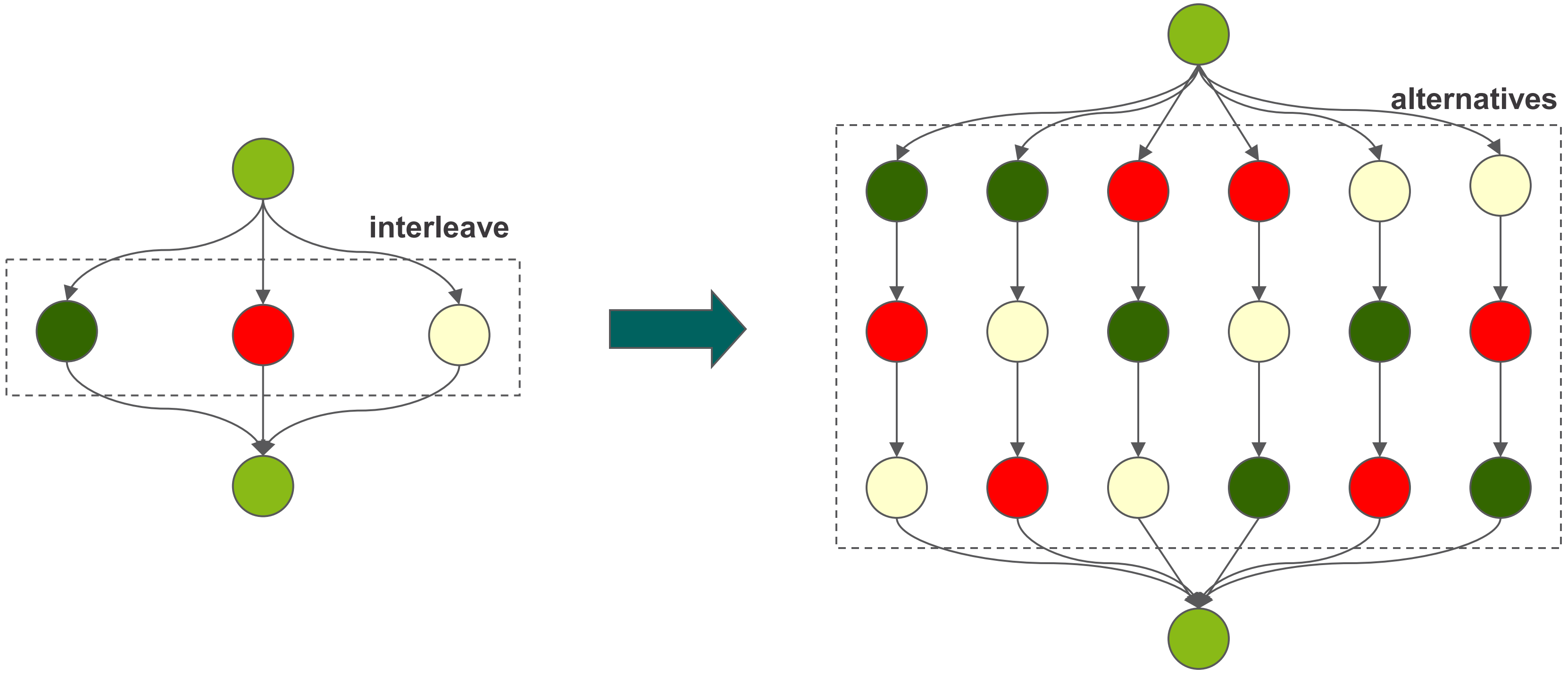
# MULTIPLE DEFAULTS

- Default branches are appended in the opposite order of their activation to the end of alt, therefore the most recently activated default branch comes before of the previously activated one(s)

```
altstep as1() runs on CT {
[] T.timeout { setverdict(inconc) }
}
altstep as2() runs on CT {
[] any port.receive { setverdict(fail) }
}
altstep as3() 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
```

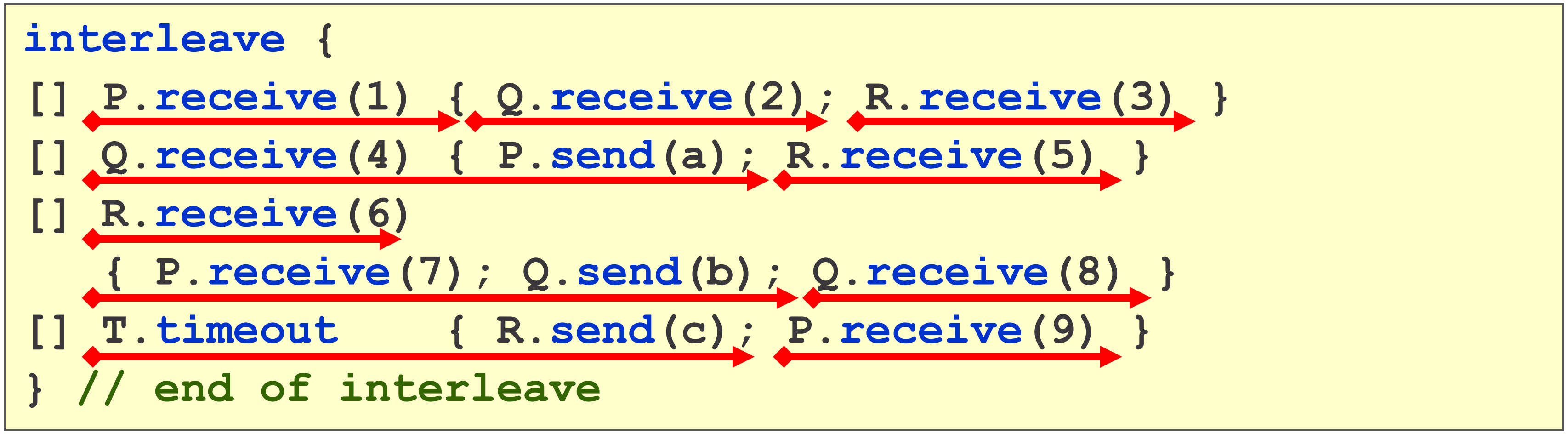
# INTERLEAVED BEHAVIOR

- Specifies the interleaved handling of events
- Alternative events can occur in any order but exactly once
- Can be modeled with a number of alt statements



# SAMPLE INTERLEAVE STATEMENT

- Difference from alt:
  - All events must happen exactly once
  - Alternative execution (i.e. snapshot semantics) applies within statement block as well
  - Execution may continue on different branch when an operation blocks the actual one and resume later from the same place



```
interleave {
[] P.receive(1)  { Q.receive(2); R.receive(3) }
[] Q.receive(4)  { P.send(a); R.receive(5) }
[] R.receive(6)
    { P.receive(7); Q.send(b); Q.receive(8) }
[] T.timeout     { R.send(c); P.receive(9) }
} // end of interleave
```

# INTERLEAVE RESTRICTIONS

- Guard must be empty
- No control statements (`for`, `while`, `do-while`, `goto`, `stop`, `repeat`, `return`) permitted in interleave branches
- No `activate`/`deactivate`, no `altstep` invocation
- No call of functions including communication operations

# OVERVIEW OF BEHAVIORAL CONTROL STATEMENTS

| Statement | Keyword or symbol |
|---|---|
| Sequential behaviour | `…; …; …` |
| Alternative behaviour | `alt { … }` |
| Interleaved behaviour | `interleave { … }` |
| Activate default | `activate` |
| Deactivate default | `deactivate` |
| Returning control | `return` |
| Repeating an alt, altstep or default | `repeat` |

# XV. SAMPLE TEST CASE IMPLEMENTATION

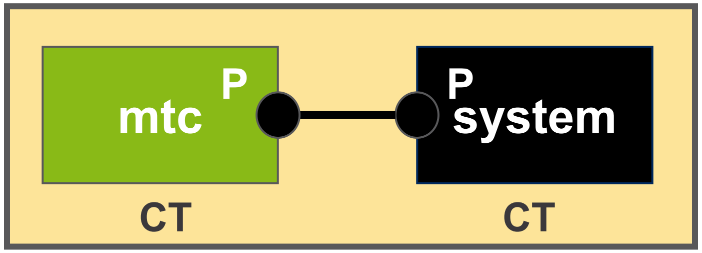
TEST PURPOSE IN MSC

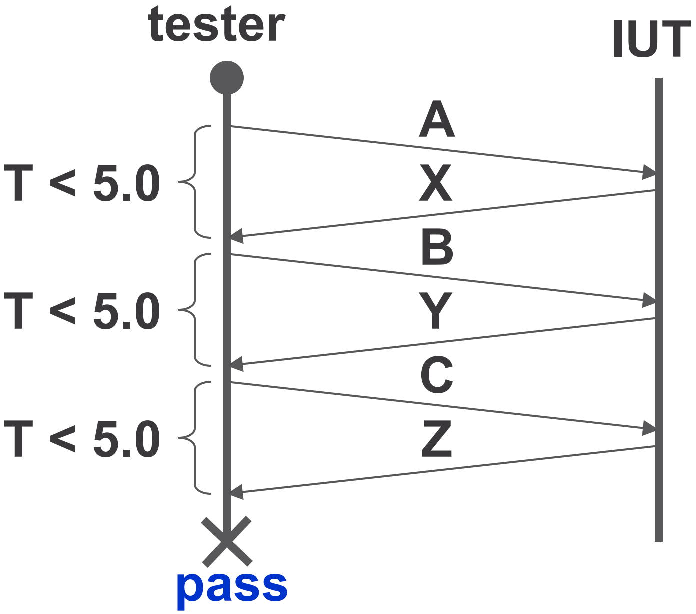
TEST CONFIGURATION

MULTIPLE IMPLEMENTATIONS

CONTENTS

# SAMPLE TEST CASE IMPLEMENTATION





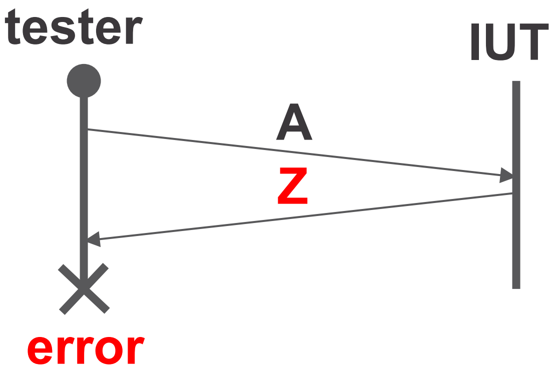
- Single component test configuration
- Test purpose defined by MSC:
  - Simple request-response protocol
  - Answer time less than 5 s
  - Result is pass for displayed operation, otherwise the verdict shall be fail

# FIRST IMPLEMENTATION WITHOUT TIMING CONSTRAINTS

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

```
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);
}
```

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



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

# SOUND IMPLEMENTATION

```
testcase test2() runs on CT {
   timer T:=5.0; map(mtc:P, system:P);
   P.send(a); T.start;
   alt {
   [] P.receive(x) {setverdict(pass)}
   [] P.receive {setverdict(fail)}
   [] T.timeout {setverdict(inconc)}
   }
   P.send(b); T.start;
   alt {
   [] P.receive(y) {setverdict(pass)}
   [] P.receive {setverdict(fail)}
   [] T.timeout {setverdict(inconc)}
   }
   P.send(c); T.start;
   alt {
   [] P.receive(z) {setverdict(pass)}
   [] P.receive {setverdict(fail)}
   [] T.timeout {setverdict(inconc)}
   }
}
```

```
type port PT message {
   out A, B, C;
   in  X, Y, Z;
}
type component CT {
   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(inconc)}
}
```

```
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 of defaults
- Compact solution employing defaults for handling incorrect IUT behavior

ERICSSON