TTCN-3

GUSZTÁV ADAMIS
ADAMIS@TMIT.BME.HU

I. INTRODUCTION TO TTCN-3

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)**

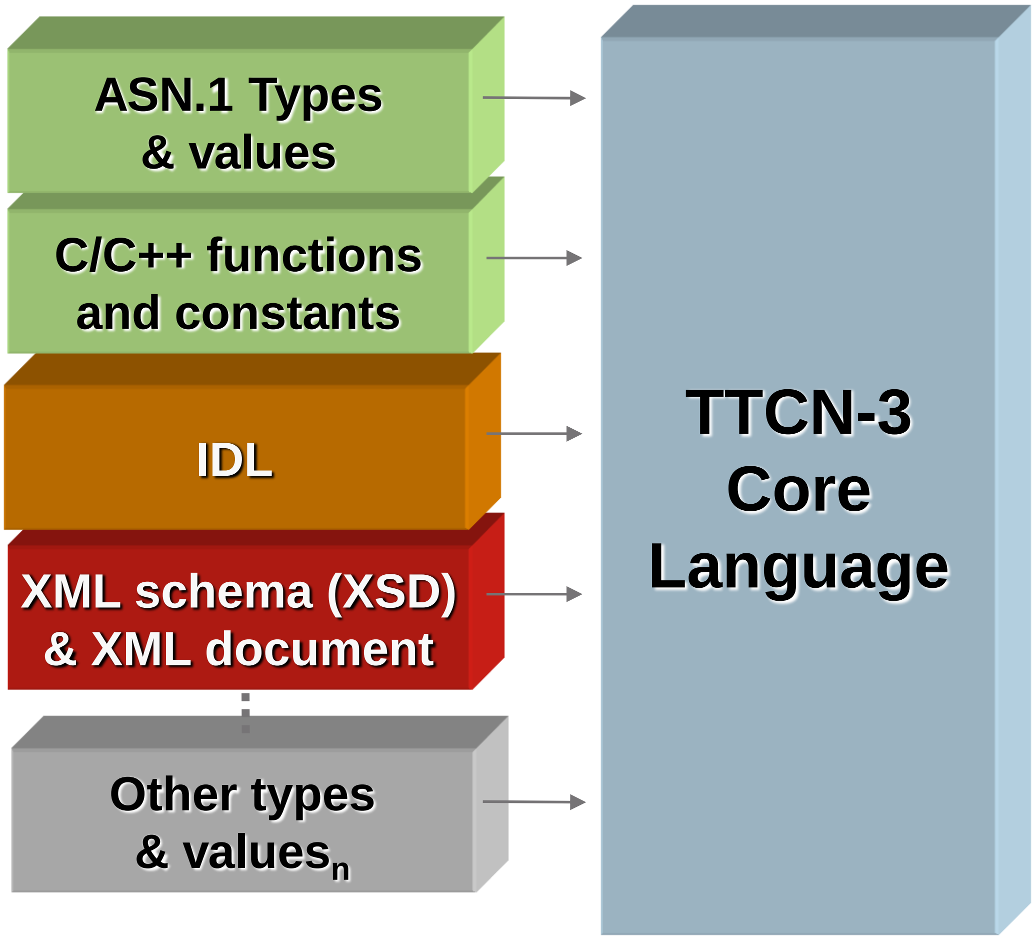
TTCN-2 TO TTCN-3 MIGRATION

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

TTCN-3 STANDARD DOCUMENTS

- Multi-part ETSI Standard v4.2.1 (2010)
  - 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/

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

EXAMPLE IN CORE LANGUAGE

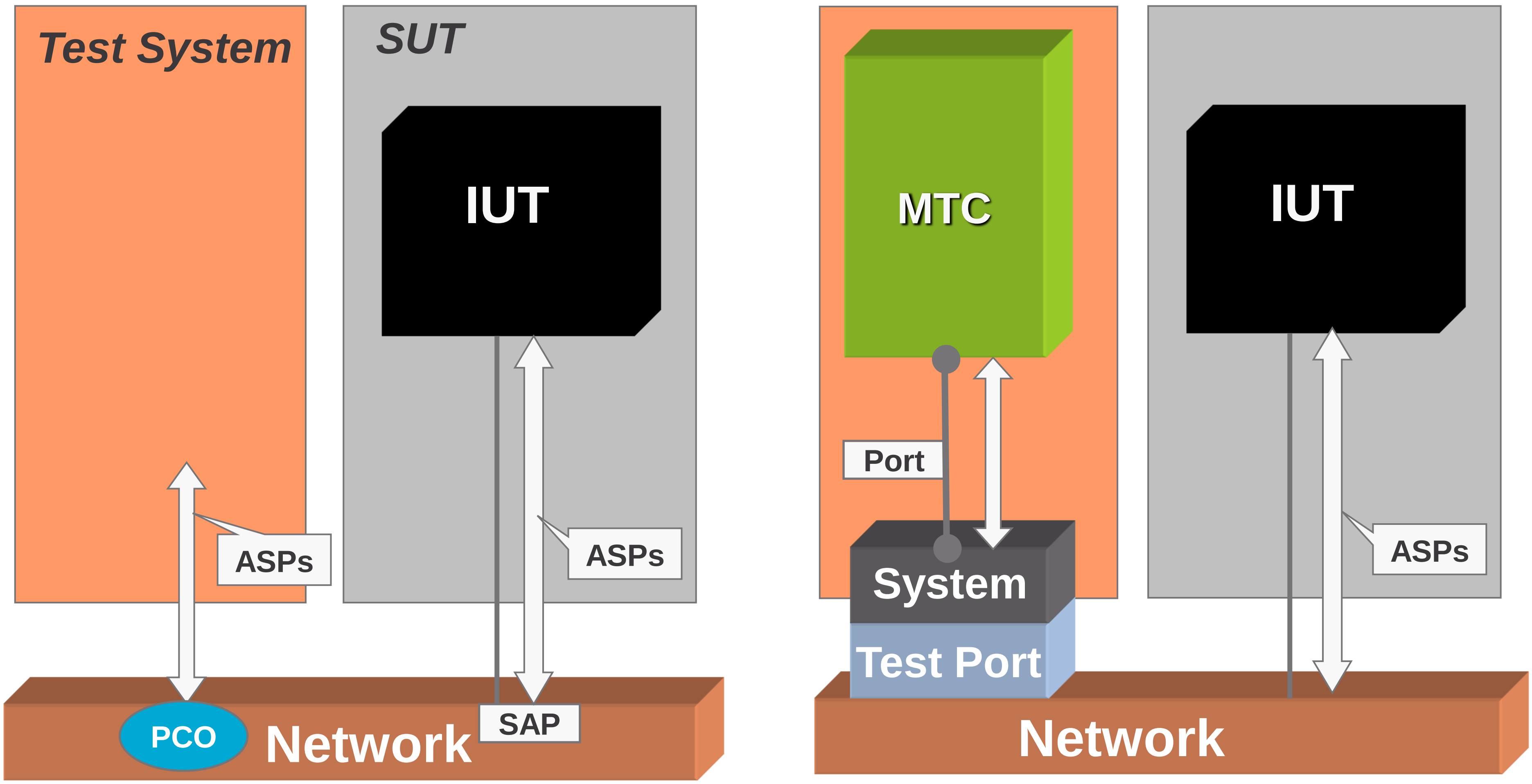
```
function PO49901(integer FL) runs on MyMTC
{
        L0.send(A_RL3(FL, CREF1, 16));
        TAC.start;
        alt {
        [] L0.receive(A_RC1((FL+1) mod 2)) {
                TAC.stop;
                setverdict(pass);
           }
        [] TAC.timeout {
                setverdict(inconc);
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

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

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

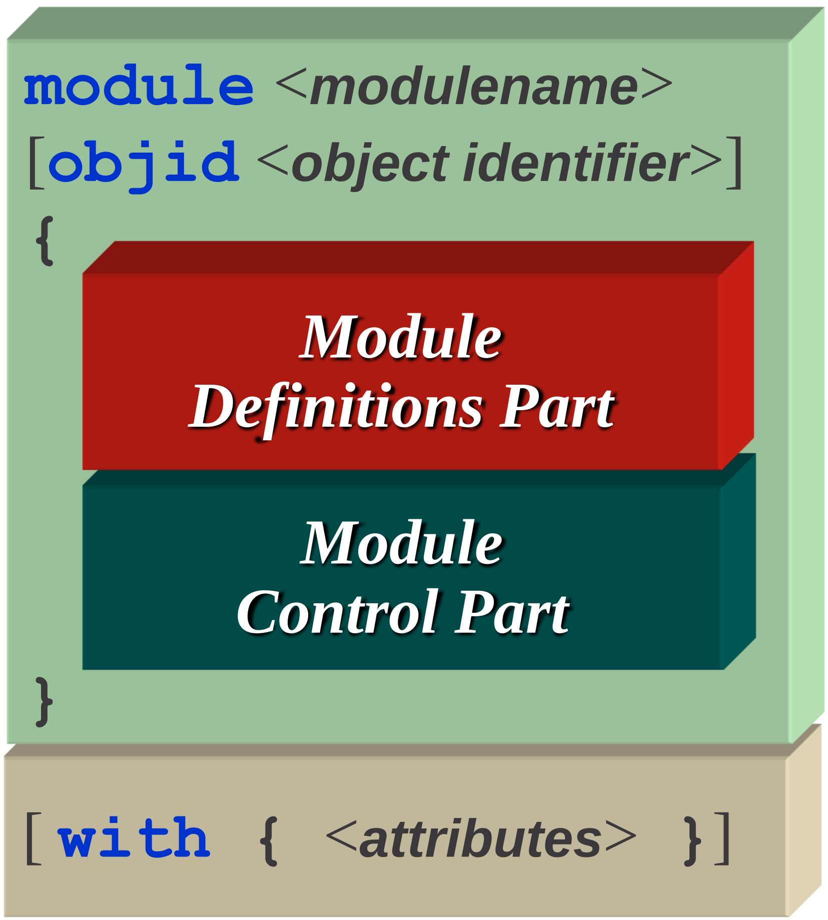
+ Abstract Data Types
+ Templates and powerful matching mechanisms
+ Event handling
+ Timer management
+ Verdict management
+ Abstract (synchronous and asynchronous) communication
+ Concurrency
+ Test specific constructions: alt, interleave, default, altstep

TEST ARRANGEMENT AND ITS TTCN-3 MODEL



II. TTCN-3 MODULE STRUCTURE

TTCN-3 MODULES

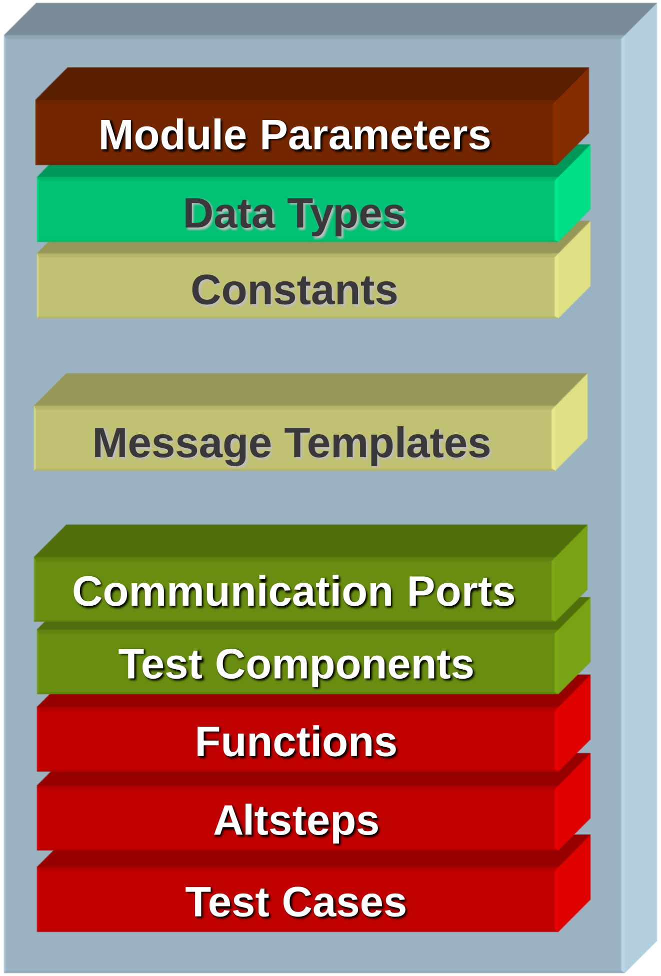


- Module – Top-level unit of TTCN-3
- A test suite consists of one or more modules
- A module contains a **module definitions** and an (optional) **module control part**.
- Modules can have run-time parameters → module parameters
- Modules can have **attributes**

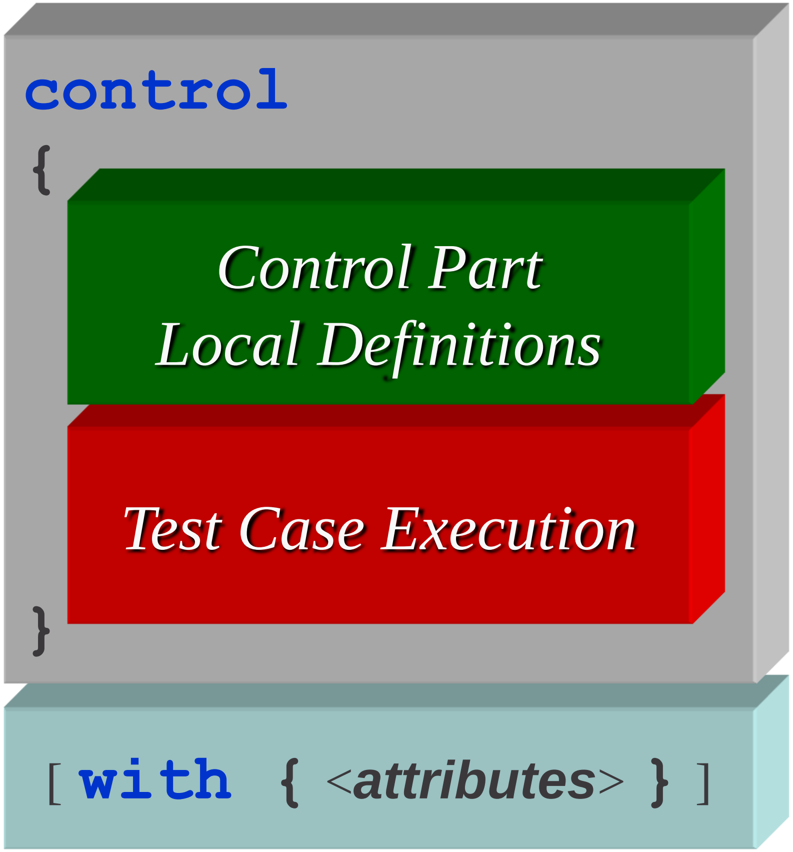
MODULE DEFINITIONS PART

Definitions in module definitions part are globally visible within the module

- **Module parameters** are external parameters, which can 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**

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() );
  }
}
```

III. TYPE SYSTEM

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 (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();
}
```

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
}
```

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

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}

// 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 };
```

STRUCTURED TYPES – `enumerated`

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

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

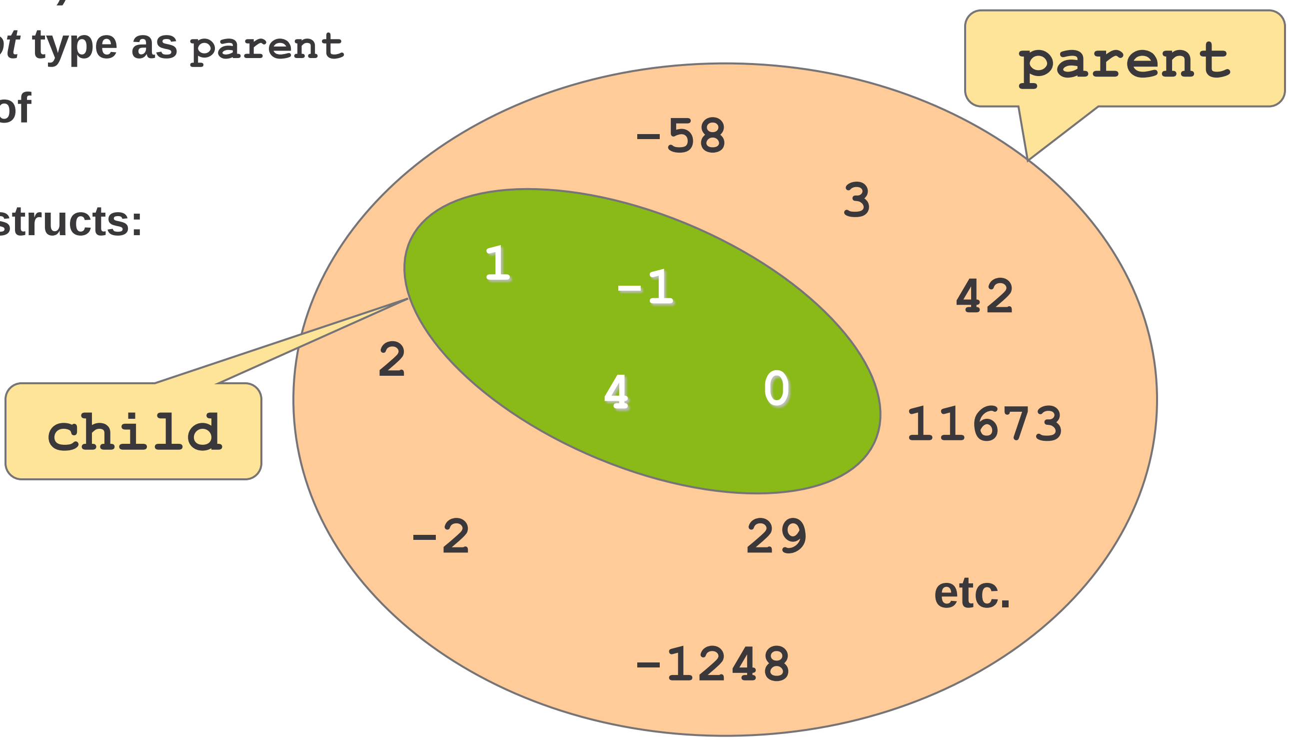
- **Enumeration items (literals):**
  - **Must have a locally (not globally) unique identifier**
- **Shall only be reused within other structured type definitions**
  - **Must not collide with local or global identifiers**
  - **Distinct integer values may optionally be associated with enumeration items**

```
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!**

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`) $\subseteq$ 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**

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");
```

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;
```

IV. CONSTANTS, VARIABLES, MODULE PARAMETERS

CONSTANT DEFINITIONS

- `const` keyword

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

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 }
```

- **Optionally, an initial value may be assigned to a variable**

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

- **Iteration counter of for loops**

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

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_myPar1 := 0;
    // module parameter w/o default value
    modulepar octetstring tsp_myPar2;
}
```

V. PROGRAM STATEMENTS AND OPERATORS

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

Statement	Keyword or symbol
Expression Condition (Boolean expression)	e.g. `2*f1(v1,c2)+1` e.g. `x+y<z`
Assignment (not an operator!)	*LHS* `:=` *RHS* e.g. `v := { 1, f2(v1) }`

Print entries into log	`log(a);`
	`log(a, …);`
	`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*>) {
	`case` (<*template*>) { <*statement*> }
	[`case` (<*template-list*>) { <*statement*> }]
	...
	[`case else` { <*statement*> }]
	}

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

OPERATORS (1)

Category	Operation	Format	Type of operands and result
Arithmetical	Addition	$+op$ or $op_1 + op_2$	op, op_1, op_2, *result*: `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; *result*: `boolean`
	Not equal	$op_1 \, != op_2$	
	Less than	$op_1 < op_2$	op_1, op_2: `integer, float, enumerated;` *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	

VI. TIMERS

TIMERS

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

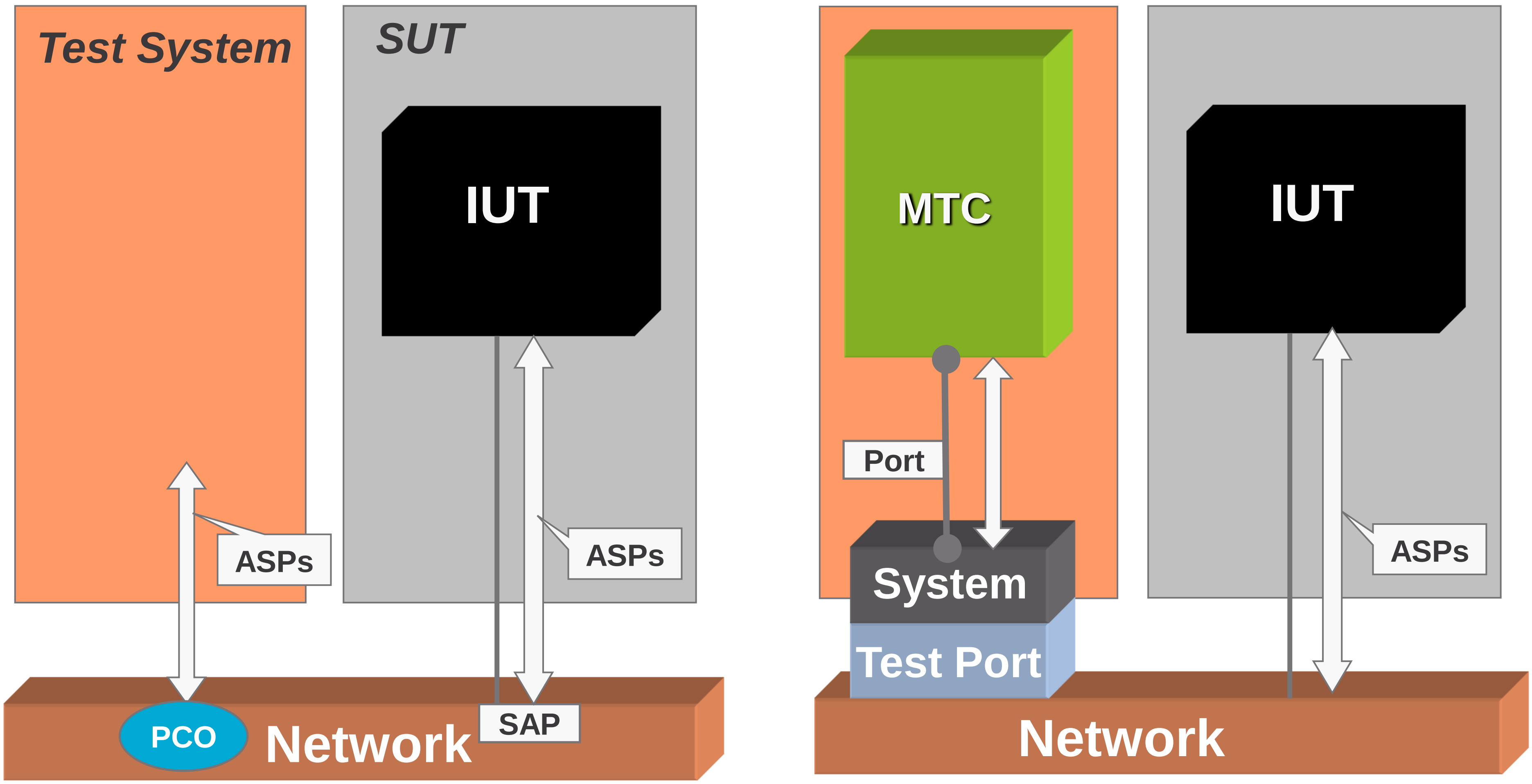
```
timer T := 5.0;
T.start;  // or T.start(2.0);
T.timeout; // block until timer expiry
```

- 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

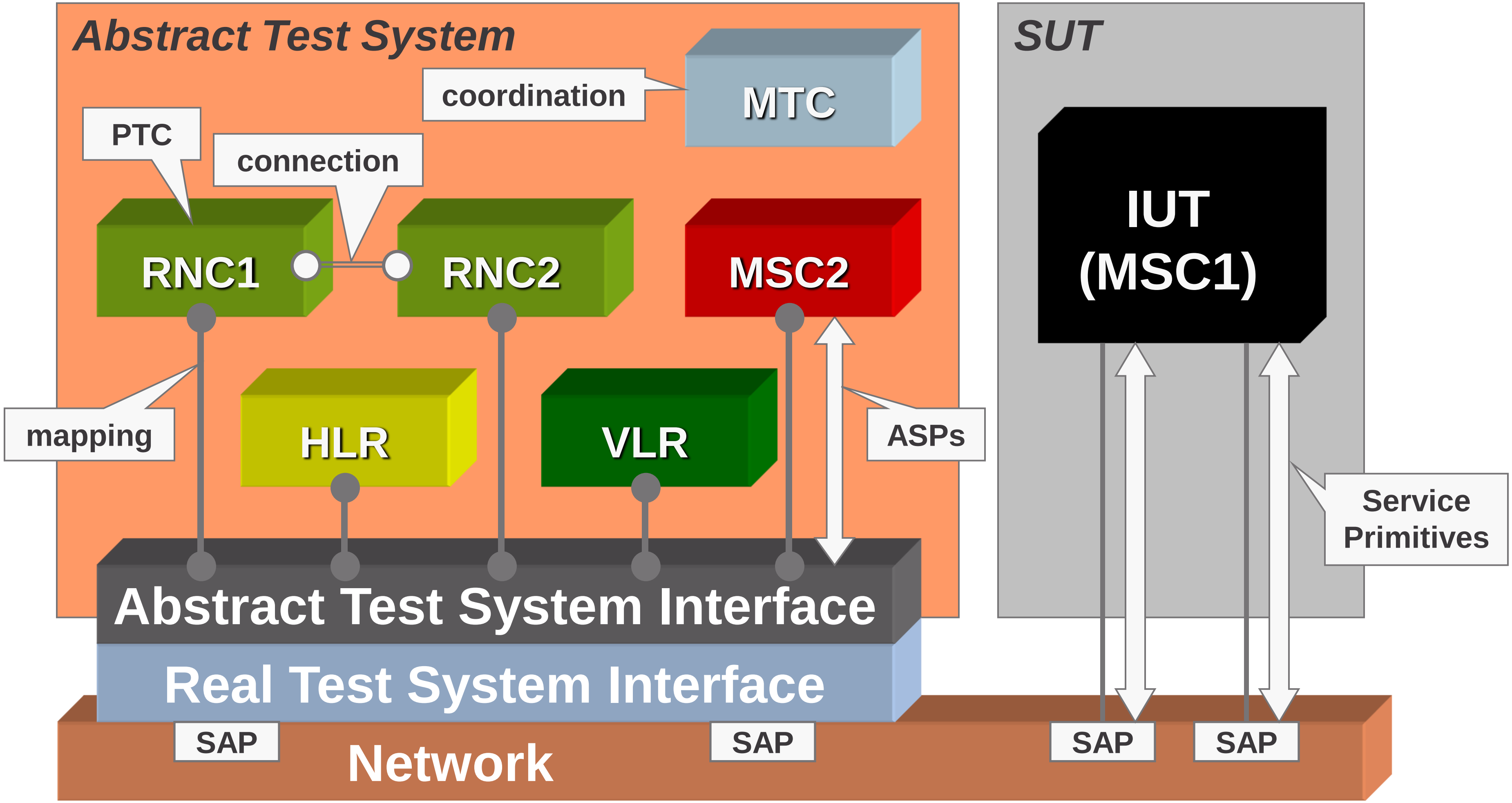
```
T.stop;
// stopping all timers in scope
all timer.stop;
```

VII. TEST CONFIGURATION

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



TTCN-3 VIEW OF TESTING – DISTRIBUTED TESTER



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!)**

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).

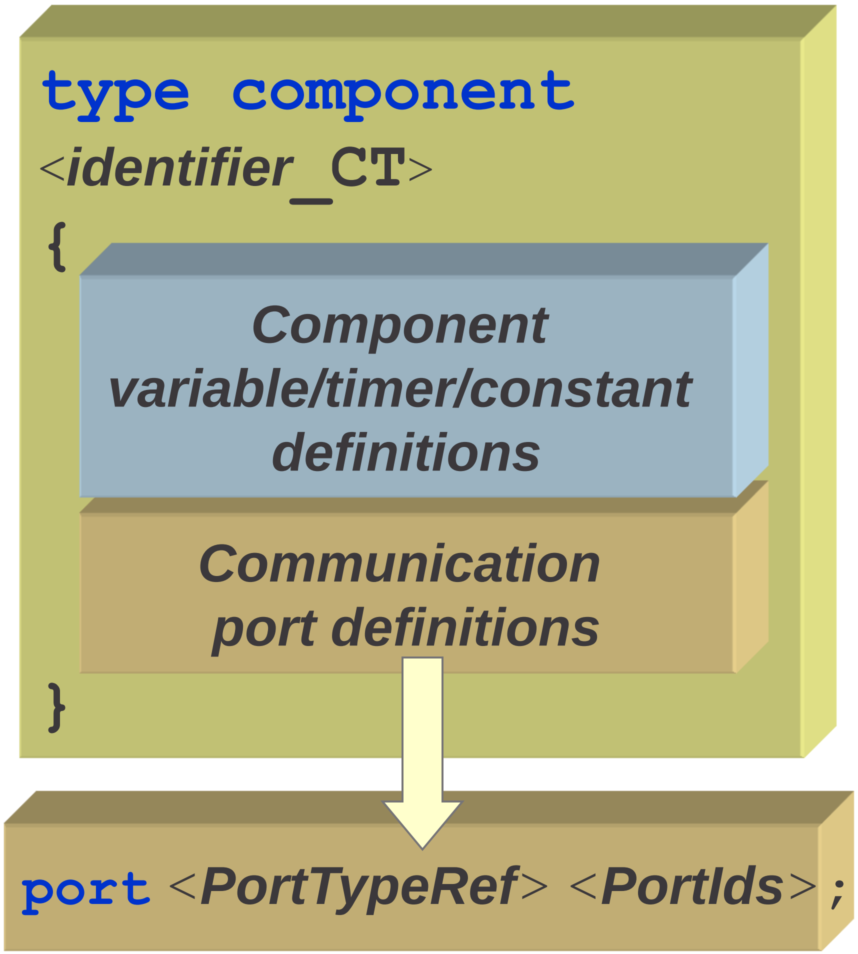
`ASP_TxType` messages can only be sent.

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

TEST COMPONENTS

- Test components are the building blocks of test configurations
- Components execute test behavior
- Three 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



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
  - local copies in component instances

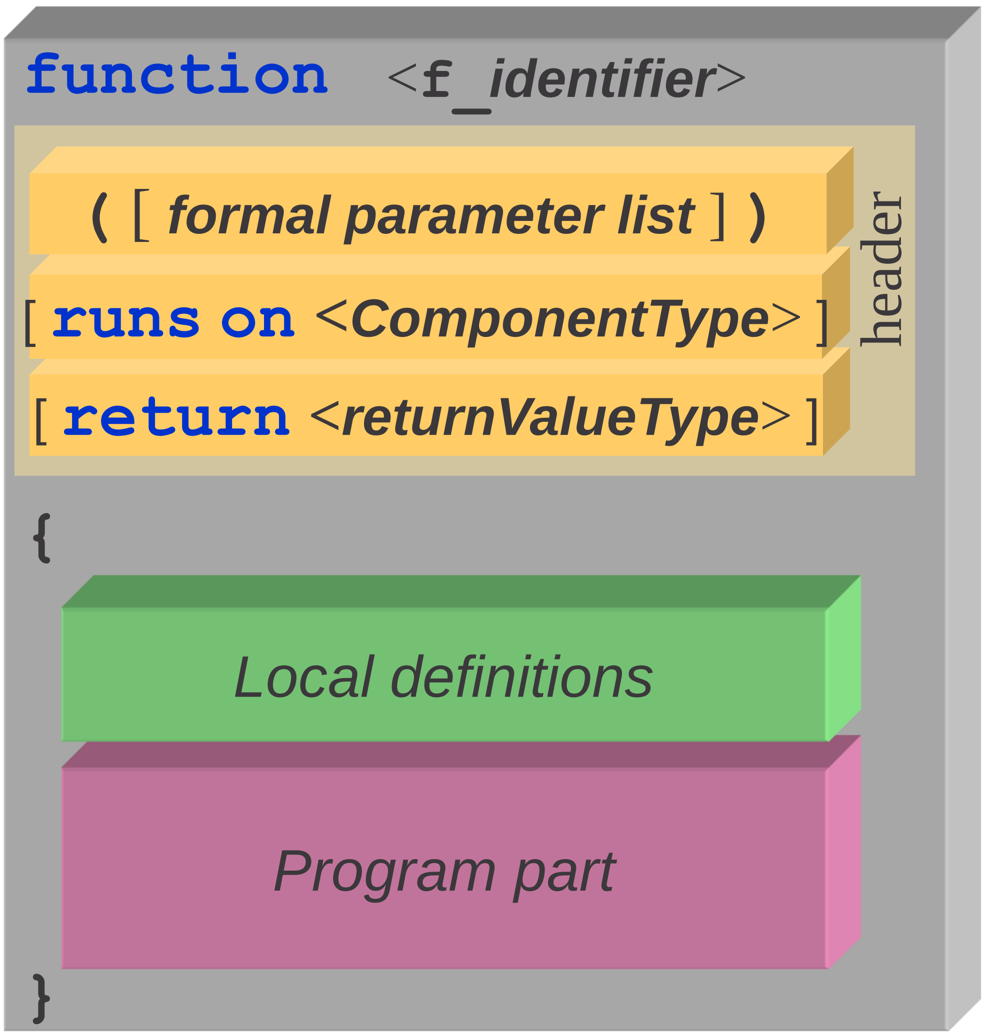
COMPONENT TYPE DEFINITION (EXAMPLE)

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 charstring Control_PCO;
 // component-wide definitions:
 const bitstring     c_MyConst := '1001'B;
 var integer         v_MyVar;
 timer               T_MyTimer := 1.0;
 }
```

VIII. FUNCTIONS AND TESTCASES

FUNCTION DEFINITION



- 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

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
}
```

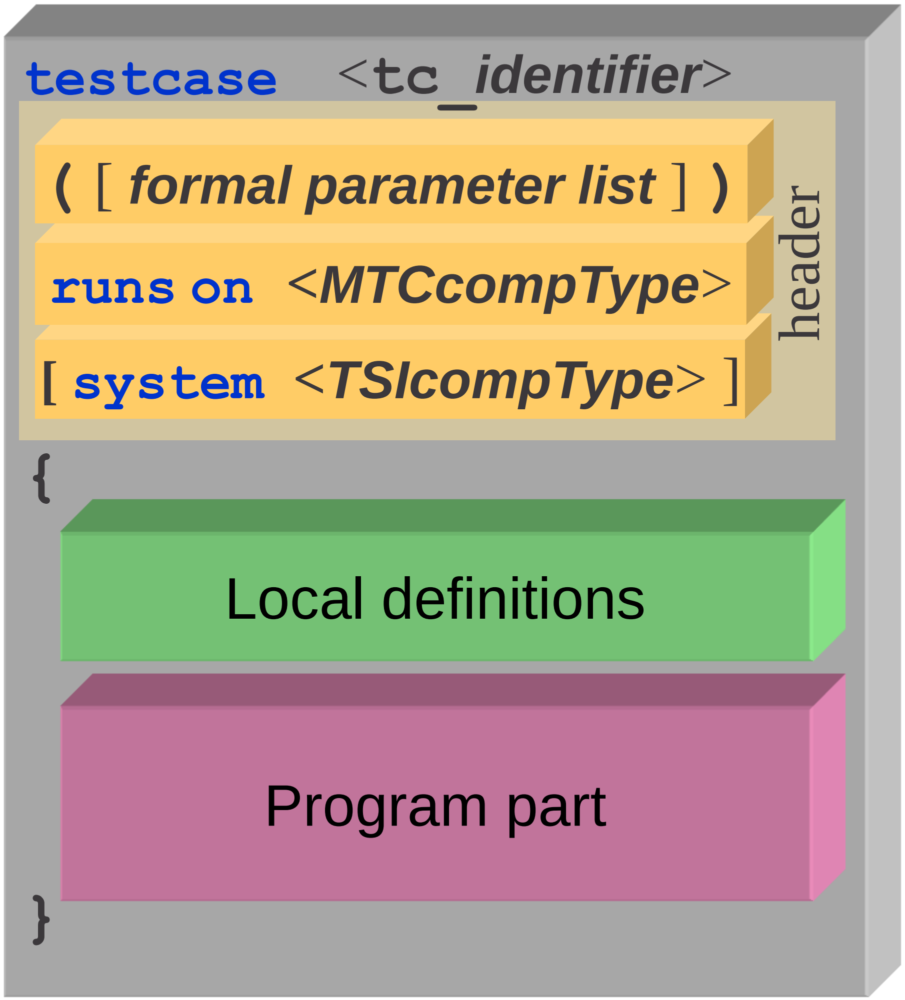
The function below uses the ports defined in `MyCompType_CT`

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

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

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

```
 vl_MyVerdict := execute(tc_TestCaseName(), 5.0);
```

IX. VERDICTS

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());
```

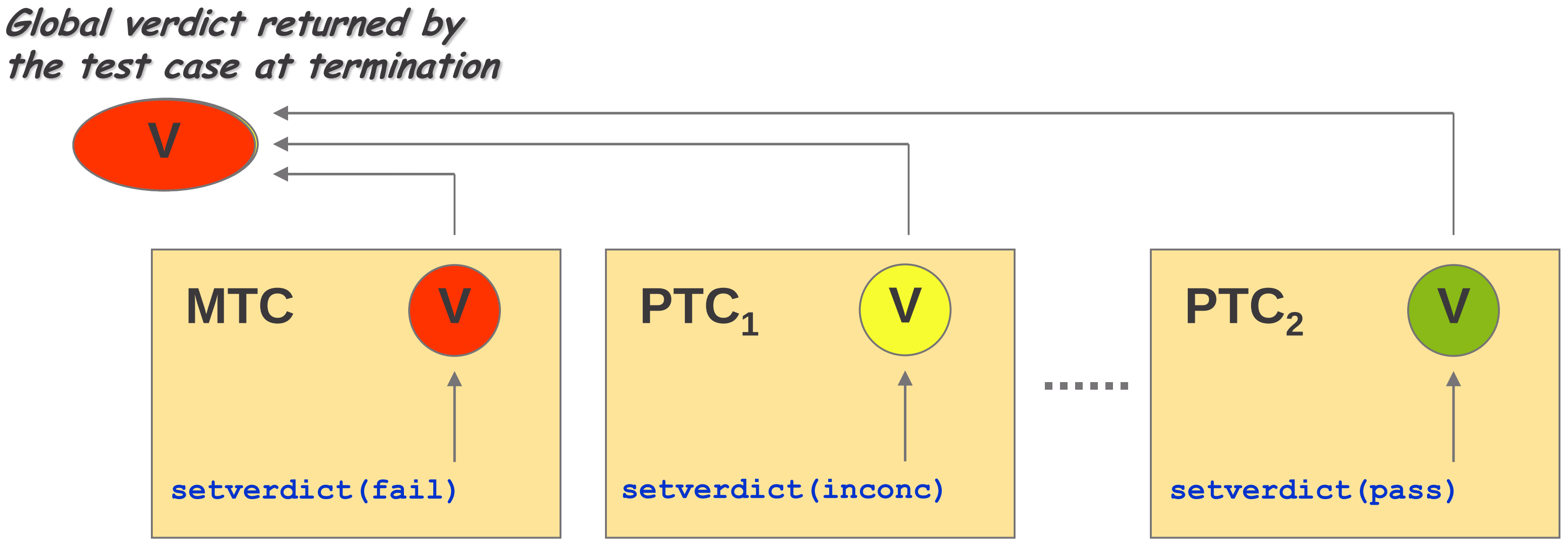
BUILT-IN VERDICT

- **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"**
- **`error` is set by the run-time environment in case of dynamic error**

```
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 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.



X. CONFIGURATION OPERATIONS

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
```

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.
```

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
  - 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));
```

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

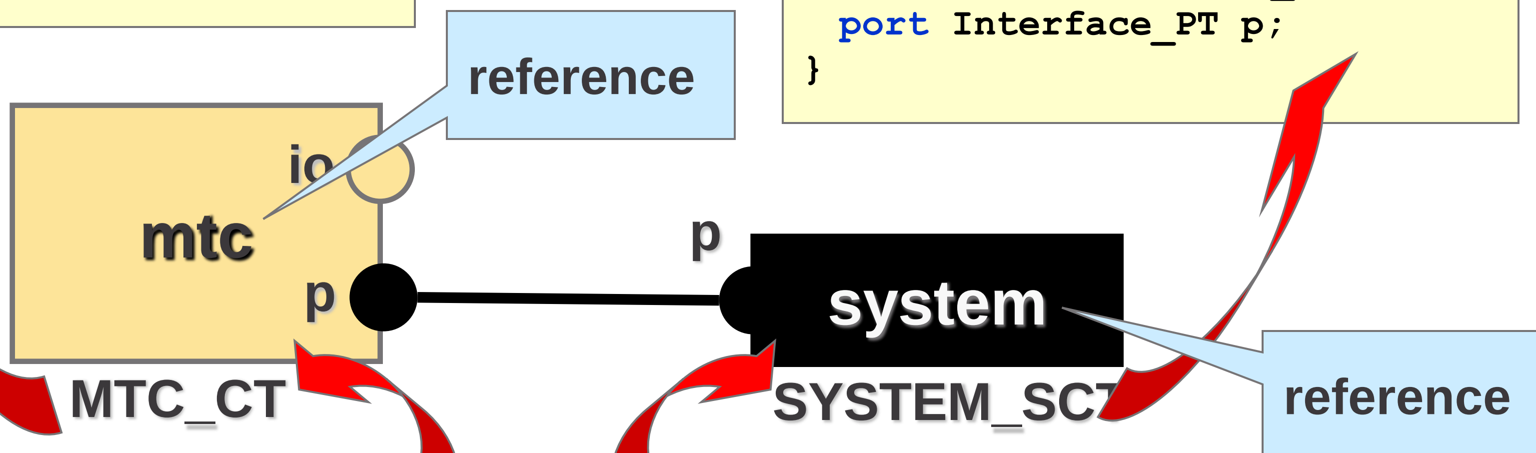
```
vc_A.done; // blocks execution until vc_A terminates

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

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

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)
}
```

XI. DATA TEMPLATES

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

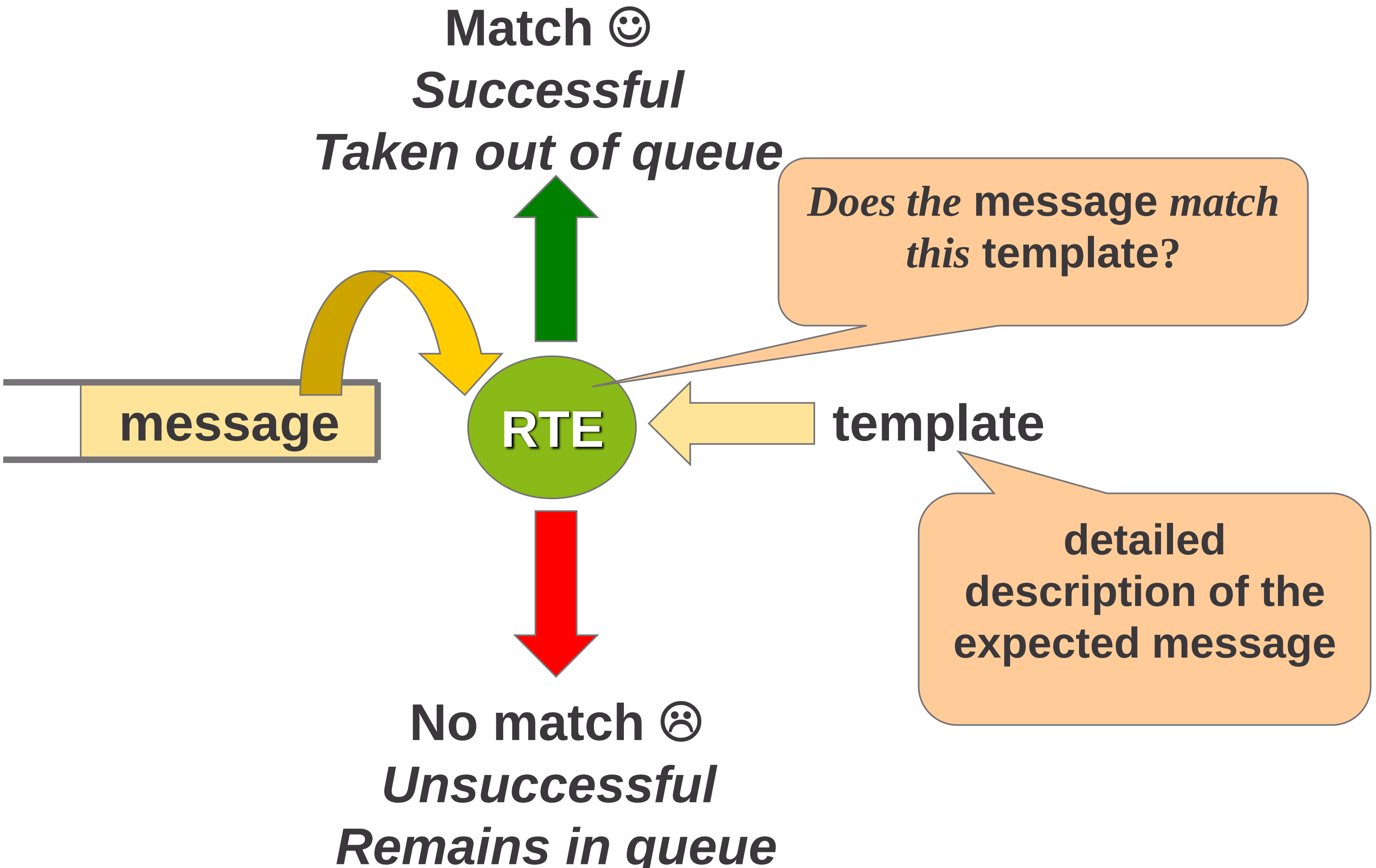
DATA TEMPLATES

- **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



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 similar to variable definition**
  - **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()`

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");
```

ANY VALUE TEMPLATE – ?
ANY VALUE OR NONE TEMPLATE – *

- **Match all valid values for the concerned template field type;**
- **`? –` Does not match when the optional field is omitted;**
- **`* –` 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 `*` or `?` field can *NOT* be sent**

```
// If both fields are optional:

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,))#(,)”;
```

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 := ?
};
```

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
 }
```

TEMPLATE PARAMETERIZATION (1)

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

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

TEMPLATE PARAMETERIZATION (2)

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

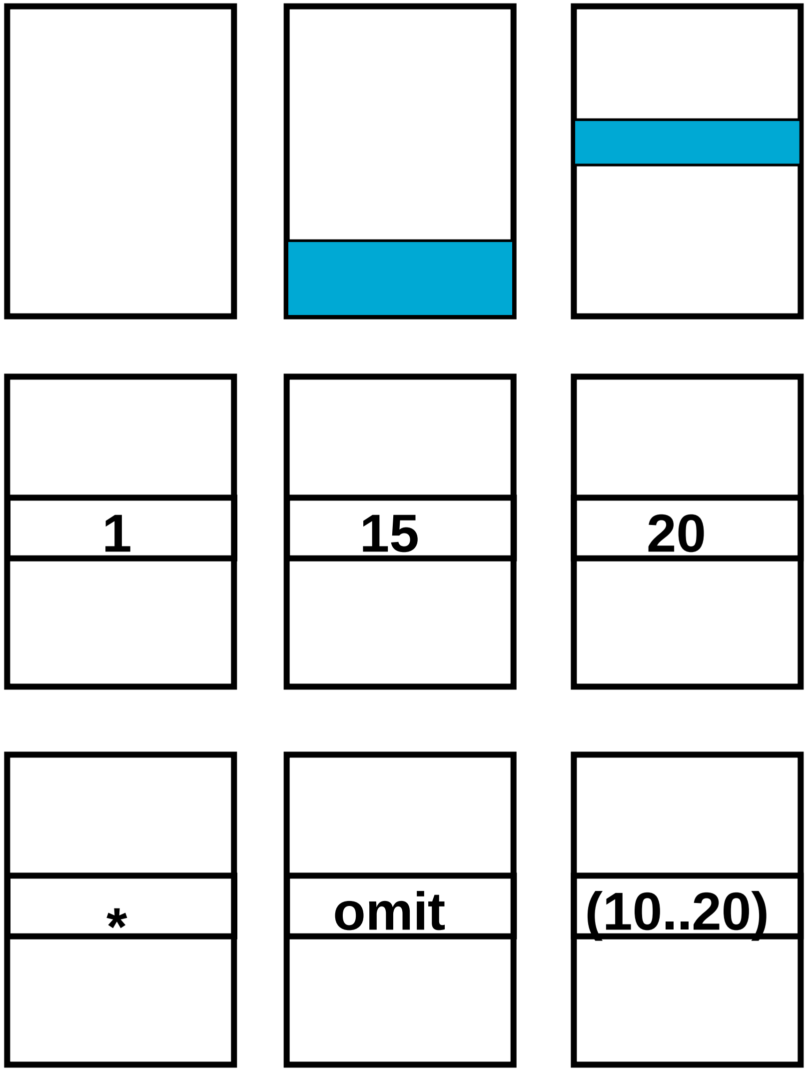
```
// Template-type parameterization
template integer tr_Int := ( (3..6), 88, 555) );
template MyIEType tr_TemplPm(template integer pl_int) :=
  { f1 := 1, f2 := pl_int }

// 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!

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

XII. ABSTRACT COMMUNICATION OPERATIONS

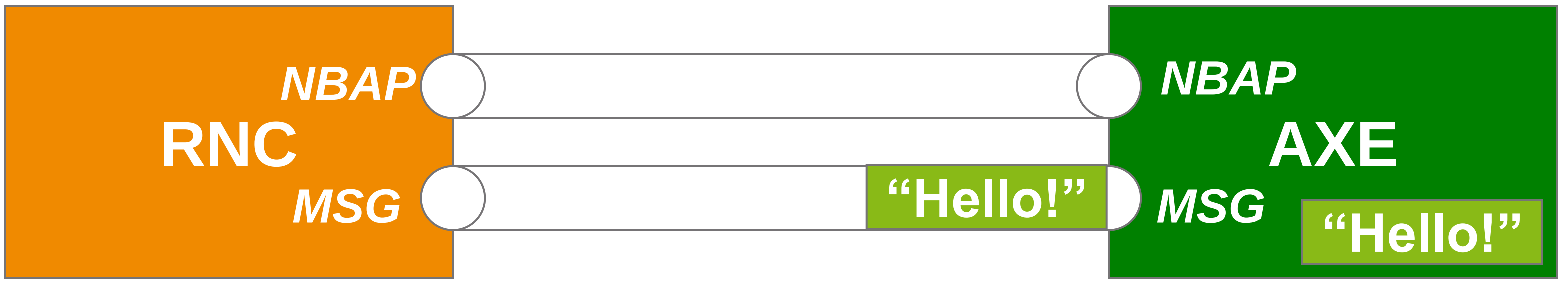
ASYNCHRONOUS COMMUNICATION

`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, <u>specific value</u> template (i.e. send template) reference or expression

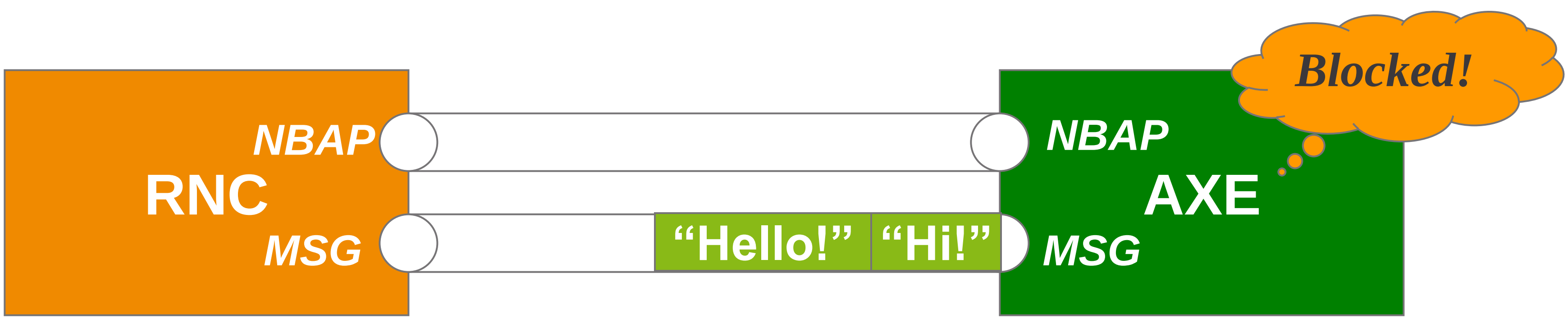
- <*PortId*>`.receive(`<*TemplateRef*>`)` or <*PortId*>`.receive`
  where <*PortId*> is the name of a `message` port containing an `in` or `inout` definition for the type of <*TemplateRef*> and <*TemplateRef*> can be:
  - Literal value; constant, variable, template (even with matching mechanisms) reference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

SEND AND RECEIVE EXAMPLES




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

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




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

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

VALUE AND SENDER REDIRECT

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

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

var MsgType MsgVar;
var CompRef Peer;

// save message matched by MsgTemplate into MsgVar
PortRef.receive(MsgTemplate) -> value MsgVar;
// obtain sender of message
PortRef.receive(MsgTemplate) -> sender Peer;
// extract MsgType message and save it with its sender
```

XIII. BEHAVIORAL STATEMENTS

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

- <u>Go for the alternative that happens earliest</u>!
- 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

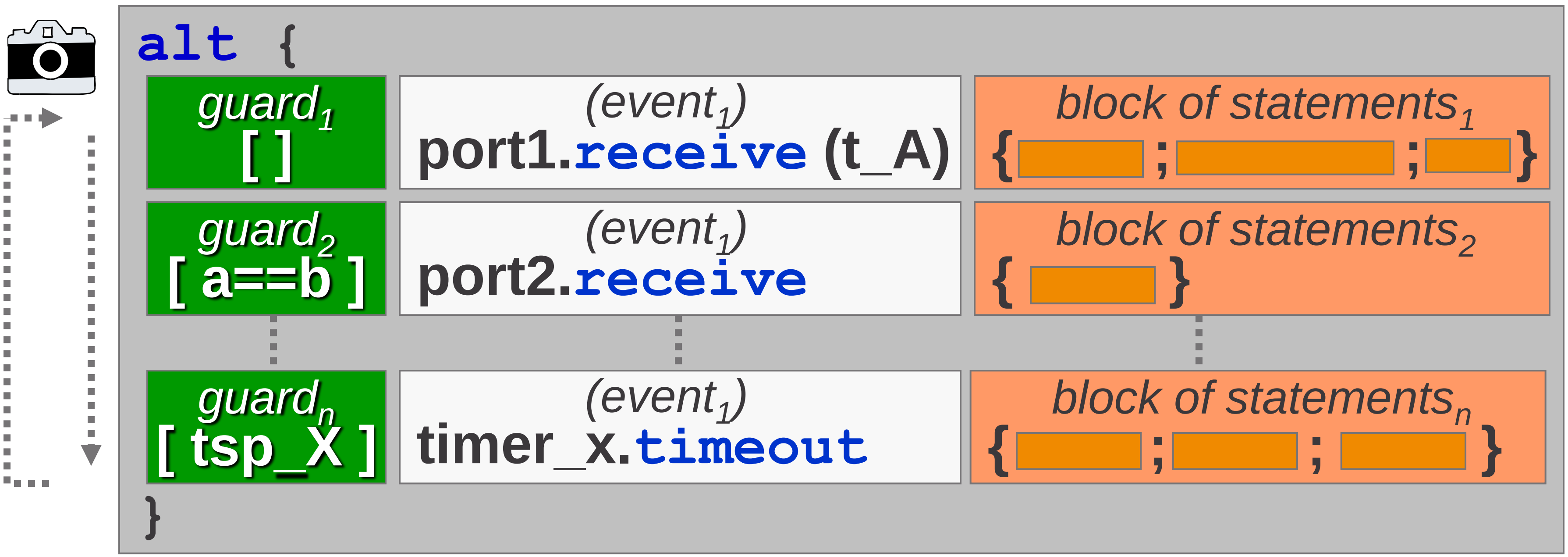
    … in order to satisfy soundness criterion
- All alternatives inside `alt` are blocking operations

- The format of `alt` statement:

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

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

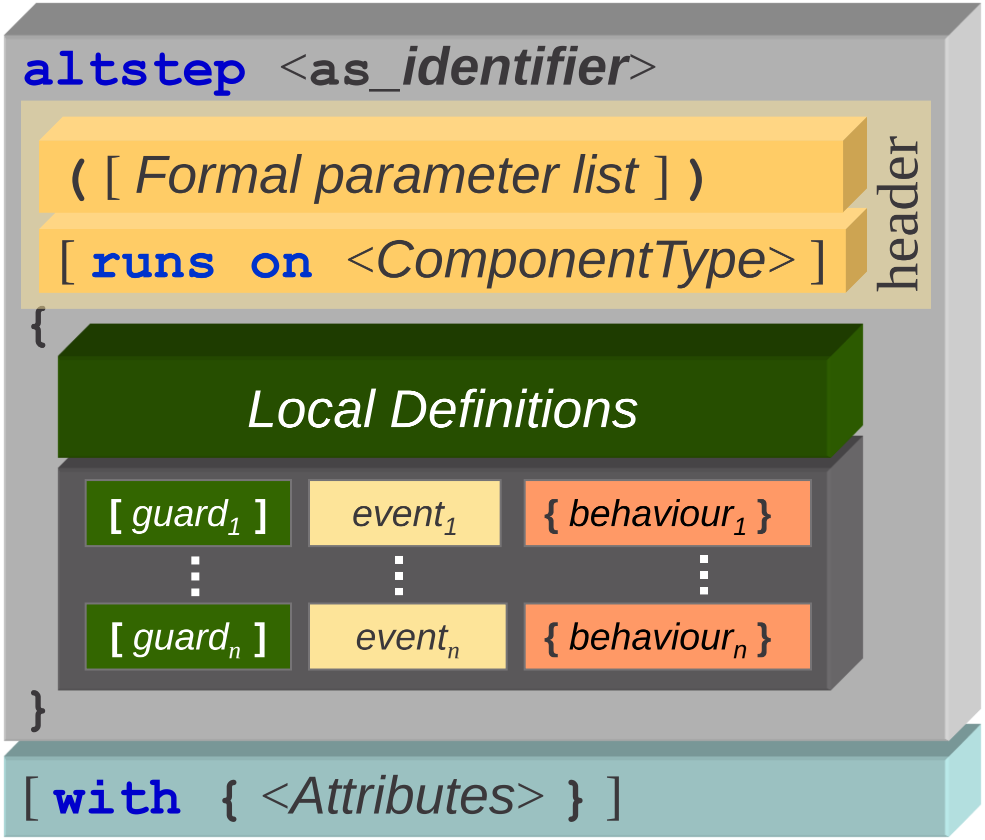


ALTERNATIVE EXECUTION BEHAVIOR EXAMPLES

- **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 */ }
}
```

STRUCTURING ALTERNATIVE BEHAVIOR – `altstep`



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

THREE 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 () {

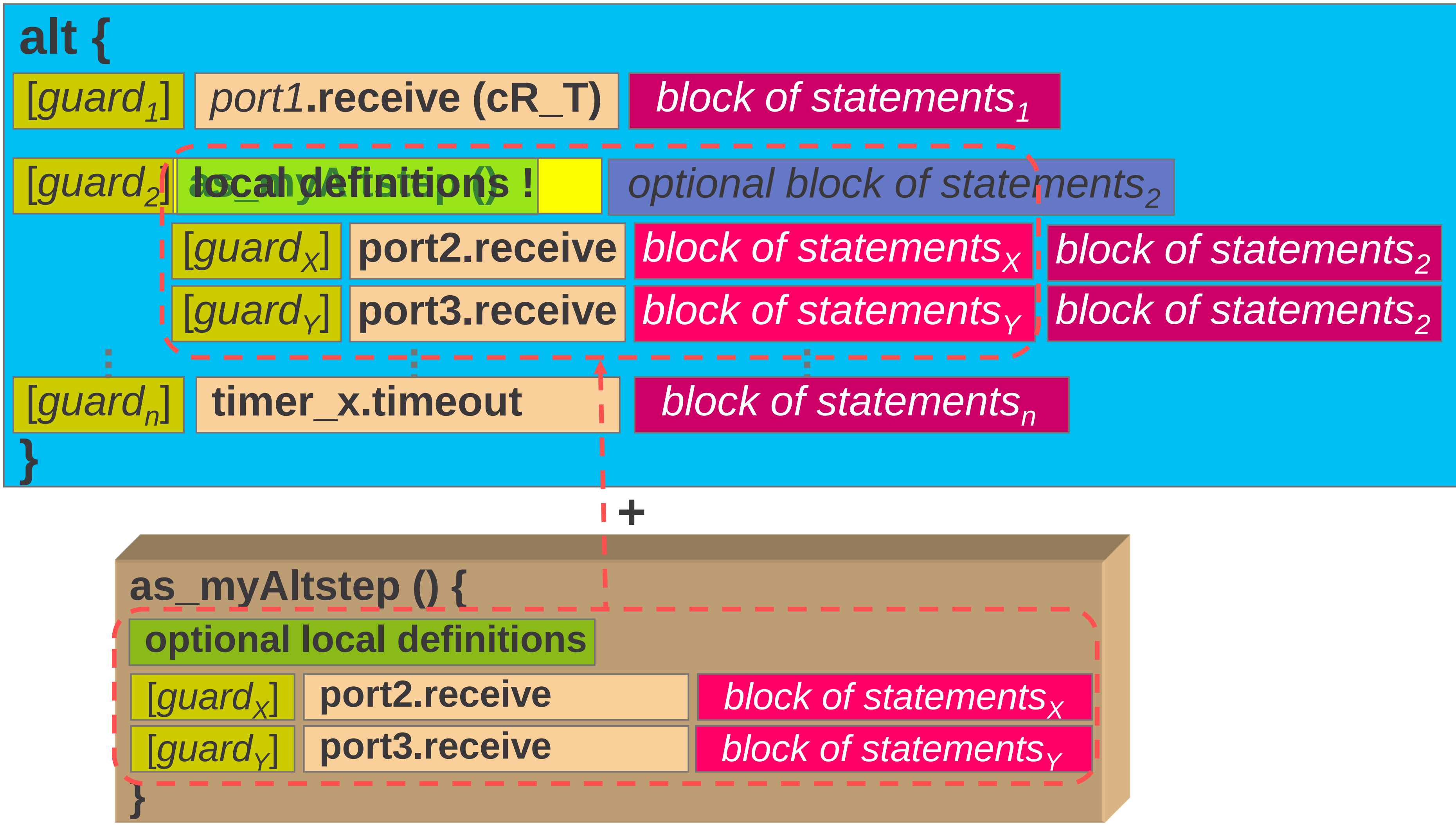
optional local definitions

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

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

}

USING altstep – INVOCATION IN alt

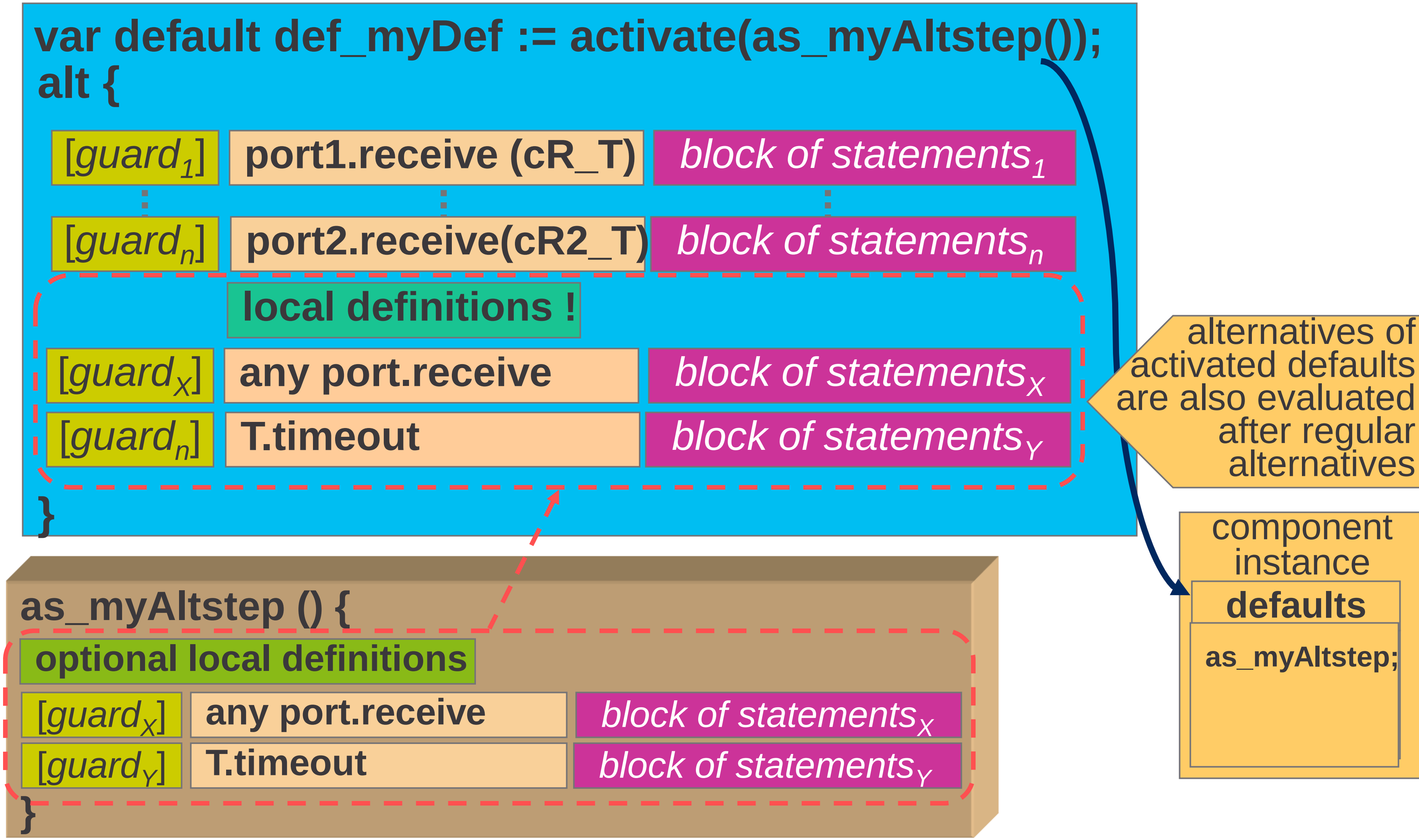


MOTIVATION - DEFAULTS

- 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



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**

STANDALONE RECEIVING STATEMENTS VS. alt

- Default context contains a list of altsteps that is implicitly appended:
  - After all stand-alone blocking receive/timeout/done … operations (!!)

- Any standalone receiving statement (receive, done, killed, 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!**

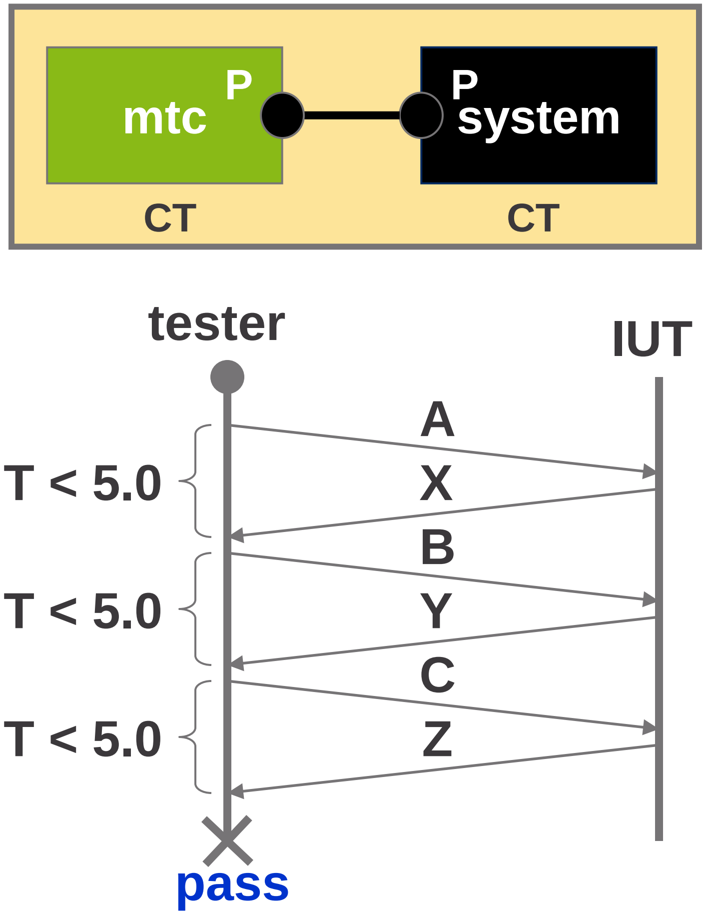
```
var default d := activate(myAltstep(2));
MyTimer.timeout;
```

- **… is equivalent to:**

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

XIV. SAMPLE TEST CASE IMPLEMENTATION

SAMPLE TEST CASE IMPLEMENTATION



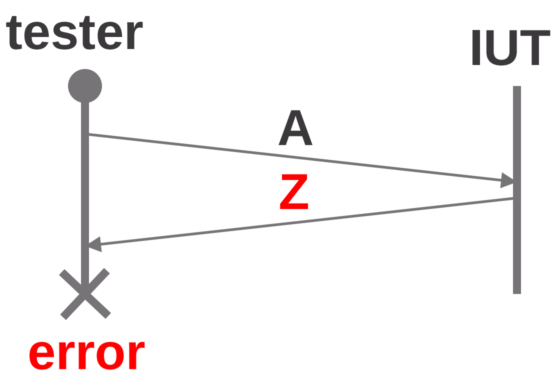
- Single component test configuration
- Test purpose defined by MSC:
  - Simple request-response protocol
  - Answer time less than 5 s
  - Result is pass for displayed operation, otherwise the verdict shall be fail

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