#### 5.4.1.1 Formal parameters of kind value

Values of all basic types, all user-defined types, address type, component type, and default can be passed as value parameters.

***Syntactical Structure***

[ ( **in** | **inout** | **out** ) ] [ **@lazy** | **@fuzzy** ] *Type* *ValueParIdentifier* [":=" ( *Expression* | "-" ) ]

***Semantic Description***

Value formal parameters can be used within the parameterized object the same way as values, for example in expressions.

Value formal parameters may be in, inout or out parameters. The default for value formal parameters is **in** parameterization which may optionally be denoted by the keyword **in**. Using of inout or out kind of parameterization shall be specified by the keywords **inout** or **out** respectively.

In parameters may have a default value, which is given by an expression assigned to the parameter. Formal parameters of modified templates may inherit the default values from the corresponding parameters of their parent templates; this shall explicitly be denoted by using a dash (don't change) symbol at the place of the modified template parameters' default value.

NOTE: If functions are used for the initialization of default values of **in** parameters, it is strongly advised to avoid side effects during the evaluation of default values. Side effects may cause non-deterministic test executions.They can be avoided, e.g., by adhering to the rules defined in clause 16.1.4.

TTCN‑3 supports value parameterization according to the following rules:

* the language element **module** allows *static* value parameterization to support test suite parameters, i.e.this parameterization may or may not be resolvable at compile-time but shall be resolved by the commencement of runtime (i.e. *static* at runtime). This means that, at runtime, module parameter values are globally visible but not changeable (see more details in clause 8.2);
* the language elements **template**, **testcase,** **altstep** and **function** support *dynamic* value parameterization (i.e. this parameterization shall be resolved at runtime).

NOTE: Component and default references are also handled as value parameters. In the case of component references, the corresponding component type is the type of the formal parameter. In the case of default references the TTCN-3 type **default** is the type of the formal parameter.

***Restrictions***

a) Language elements which cannot be parameterized are: **const**, **var**, **timer**, **control**, **record** **of**, **set** **of**, **enumerated**, **port**, **component** and subtype definitions, **group** and **import**.

b) Formal value parameters of templates, and of altsteps activated as defaults (see clause 20.5.2) shall always be **in** parameters.

c) Restrictions on module parameters are given in clause 8.2.

d) Default values can be provided for **in** parameters only.

e) The expression of the formal parameters' default value has to be compatible with the type of the parameter. The expression shall not refer to elements of the component type of the optional **runs on** clause. The expression shall not refer to other parameters of the same parameter list. The expression shall not contain the invocation of functions with a **runs on** clause.

f) Default values of component type formal parameters shall be one of the special values **null, mtc, self**, or **system**.

g) Default values of default type formal parameters shall be the special value **null**.

h) The dash (don't change) symbol shall be used with formal parameters of modified templates only (see also clause 15.5).

i) For formal value parameters of templates the restrictions specified in clause 15 shall apply.

j) Only in parameters can be declared lazy or fuzzy.

k) When parameters are referenced (e.g. in assignments, expressions, template bodies, etc.), the rules for variables shall apply.

***Examples***

EXAMPLE 1: In, out and inout formal parameters

**function** MyFunction1(**in boolean** MyReferenceParameter){ … };

// MyReferenceParameter is an in value parameter. The parameter can be read. It can also be set // within the function, however, the assignment is local to the function only

**function** MyFunction2(**inout boolean** MyReferenceParameter){ … };

// MyReferenceParameter is an inout value parameter. The parameter can be read and set

// within the function - the assignment is not local

**function** MyFunction3(**out template boolean** MyReferenceParameter){ … };

// MyReferenceParameter is an out value parameter. The parameter can be set within the function,   
 // the assignment is not local. It can also be read, but only after it has been set.

EXAMPLE 2: Reading and setting parameters

**type** **record** MyMessage {

**integer** f1,

**integer** f2

}

**function** f\_MyMessage (**integer** p\_int) **return** MyMessage {

**var** **integer** f1, f2;

f1 := f\_mult2 (p\_int);

// parameter p\_int is passed on; as the parameter of the called function f\_mult2 is

// defined as an inout parameter, it passes back the changed value for p\_int,

f2 := p\_int;

**return** {f1, f2};

}

**function** f\_mult2 (**inout** **integer** p\_integer) **return** **integer** {

p\_integer := 2 \* p\_integer;

// the value of the formal parameter is changed; this new value is passed back when

// f\_mult2 completes

**return** p\_integer-1

}

**testcase** tc\_01 () **runs** **on** MTC\_PT {

...

P1.**send** (f\_MyMessage(5))

// the value sent is { f1 := 9 , f2 := 10 }

...

}

EXAMPLE 3: Function with default value for parameter

**function** f\_comp (**in integer** p\_int1, **in** **integer** p\_int2 := 3) **return** **integer** {

**var** **integer** v := p\_int1 + p\_int2;

:

**return** v;

}  
  
 **function** f () {

**var** **integer** w;  
 …

w := f\_comp(1); // same as calling f\_comp(1,3);  
 w := f\_comp(1,2); // value 2 is taken for parameter p\_int2 and not its default value 3

…

}

EXAMPLE 4: Direct passing of formal parameters to functions

**function** f\_MyFunc2(**in** **bitstring** p\_refPar1, **inout** **integer** p\_refPar2) **return** **integer** {

:

}

**function** f\_MyFunc1(**inout bitstring** p\_refPar1, **out** **integer** p\_refPar2) **return** **integer** {

:

**return** f\_MyFunc2(p\_refPar1, p\_refPar2);

}

// p\_refPar1 and p\_refPar2 can be passed directly to a function invocation

EXAMPLE 5: Lazy and fuzzy parameters

**type component** MyComp { **var integer** v\_int }

**function** f\_MyLazyFuzzy(**in** **@lazy** **integer** p\_lazy, **in** **@fuzzy** **integer** p\_fuzzy) **runs on** MyComp {

//When called from MyCalling:

v\_int := 1;

**log**(p\_lazy); //will log 2 as function double with actual parameter v\_int equals 1 is called

//here; 2 is stored in p\_lazy (also, function double stores 2 in v\_int)

**log**(p\_lazy); //will log 2 again as p\_lazy is not re-evaluated

**log**(p\_fuzzy);//will log 4 as function double with actual parameter v\_int equals 2 is called

// here (also, function double stores 4 in v\_int)

**log**(p\_fuzzy) //will log 8 as function double is re-evaluated with actual parameter 4

}

**function double** (**in integer** p\_in) **runs on** MyComp **return integer**{

p\_in := 2\* p\_in;

v\_int := p\_in;

**return** p\_in

}

**testcase** tc\_MyCalling() **runs on** MyComp {

v\_int := 0;

f\_MyLazyFuzzy (double(v\_int), double(v\_int) )

}

EXAMPLE 6: Difference between passing by value and passing by reference

**function** f\_byValue (**in integer** p\_int1, **in** **integer** p\_int2) {

p\_int2 := p\_int2 + 1;

log(p\_int1);

log(p\_int2);

}

**function** f\_byReference (**inout integer** p\_int1, **inout** **integer** p\_int2) {

p\_int2 := p\_int2 + 1;

log(p\_int1);

log(p\_int2);

}

**function** f () {

**var** **integer** v\_int := 1;

f\_byValue(v\_int, v\_int); // prints 1 and 2

log(v\_int); // prints 1

f\_byReference(v\_int, v\_int); // prints 2 and 2

log(v\_int); // prints 2

}

### 5.4.2 Actual parameters

Values, templates, timers and/or ports can be passed into parameterized TTCN-3 objects as actual parameters. Actual parameters can be provided both as a list in the same order as the formal parameters as well as in an assignment notation explicitly using the associated formal parameter names.

***Syntactical Structure***

( *Expression* | // for value parameter

*TemplateInstance* | // for template parameter

*TimerRef* | // for timer parameter

*Port* | // for port parameter

"-" ) | // to skip a parameter with default  
 *ParameterId* ":=" ( Expression | TemplateInstance | TimerRef | Port ) )

***Semantic Description***

Actual parameters that are passed by value to **in** formal value parameters shall be variables, literal values, module parameters, constants, variables, value returning (external) functions, formal value parameters (of in, inout or out parameterization) of the current scope or expressions composed of the above.

Actual parameters that are passed to **inout** formal value parameters shall be variables or formal value parameters (of in, inout or out parameterization) or references to elements of variables or formal value parameters of structured types.

NOTE: Reference to a string element cannot be passed by reference as string types are not structured types.

Actual parameters that are passed to **in** formal template parameters shall be literal values, module parameters, constants, variables, value or template returning (external) functions, formal value parameters (of in, inout or out parameterization) of the current scope or expressions composed of the above, as well as templates, template variables or formal template parameters (of in, inout or out parameterization) of the current scope.

Actual parameters that are passed to **out** formal template parameters shall be variables, template variables, formal value parameters, formal template parameters or references to elements of variables, template variables, formal value parameters or formal template parameters of structured types.

Actual parameters that are passed to **inout** formal template parameters shall be variables, template variables, formal value or template parameters (of in, inout or out parameterization) of the current scope or references to elements of (template) variables or formal (template) parameters of structured types.

When actual parameters that are passed to **in** formal value or template parameters contain a value or template reference, rules for using references on the right hand side of assignments apply. When actual parameters that are passed to **inout** and **out** formal value or template parameters contain a value or template reference, rules for using references on the left hand side of assignments apply.

The values of **out** formal parameters are passed to the actual parameters in the same order as is the order of formal parameters in the definition of the parameterized TTCN-3 object.

Actual parameters that are passed to formal timer parameters shall be component timers, local timers or formal timer parameters of the current scope.

Actual parameters that are passed to formal port parameters shall be component ports or formal port parameters of the current scope.

It is allowed to pass elements of structured values or templates (record, set, record of, set of, union and anytype values or templates) by reference. Modification of parameters passed this way affects the original structured value or template. Before passing the actual parameter, the rules for referencing the element on the left hand side of assignments are applied, expanding the structured value so that the referenced element becomes accessible (see clauses 6.2 and 15.6 for more details).

When a formal parameter has been defined with a default value or template, respectively, then it is not necessary to provide an actual parameter. In such a case the default value or template is taken as actual parameter. All parameters are evaluated in the order of their appearance in the corresponding formal parameter list.

The empty brackets for instances of parameterized templates that have only parameters with default values are optional when no actual parameters are provided, i.e. all formal parameters use their default values.

***Restrictions***

a) When using list notation, the order of elements in the actual parameter list shall be the same as their order in the corresponding formal parameter list. For each formal parameter without a default there shall be an actual parameter. The actual parameter of a formal parameter with default value can be skipped by using dash "-" as actual parameter. An actual parameter can also be skipped by just leaving it out if no other actual parameter follows in the actual parameter list - either because the parameter is last or because all following formal parameters have default values and are left out.

b) Either list notation or assignment notation shall be used in a single parameter list. They shall not be mixed.

c) When using assignment notation, each formal parameter shall be assigned an actual parameter at most once. For each formal parameter without default value, there shall be an actual parameter. In order to use the default value of a formal parameter, no assignment for this specific parameter shall be provided.

d) The type of each actual parameter shall be compatible with the type of each corresponding formal parameter. Strong typing is required for parameters passed by reference.

e) Actual parameters passed to restricted formal template parameters shall obey the restrictions given in clause 15.8.

f) All parameterized entities specified as an actual parameter shall have their own parameters resolved in the top‑level actual parameter list.

g) If the formal parameter list of TTCN‑3 objects **function**, **testcase**, **signature, altstep** or **external** **function** is empty, then the empty parentheses shall be included both in the declaration and in the invocation of that object. In all other cases the empty parentheses shall be omitted.

h) Restrictions on the use of signature parameters are given in clauses 15.2 and 22.3.

i) Restrictions on parameters passed to altsteps are given in clauses 16.2.1 and 20.5.2.

j) Unless specified differently in the relevant clause(s), actual parameters passed to **in** or **inout** formal parameters shall be at least partially initialized (for an exemption see e.g. clause 16.1.2 of the present document).

k) Functions, called by actual parameters passed to fuzzy or lazy formal parameters of the calling function, shall not have inout or out formal parameters. The called functions may use other functions with inout or out parameters internally.

l) Actual parameters passed to **out** and **inout** parameters shall not be references to lazy or fuzzy variables.

m) Whenever a value or template of a record, set, union, record of, set of, array and anytype type is passed as an actual parameter to an inout parameter, none of the fields or elemens of this structured value or template shall be passed as an actual parameter to another inout parameter of the same parameterized TTCN-3 object. This restriction applies recursively to all sub-elements of the structured value or template in any level of nesting.

n) If two or more actual parameters passed to **inout** parameters of the same parameterized TTCN-3 object contain a reference to distinct alternatives of the same union or anytype value, an error shall be produced.

***Examples***

EXAMPLE 1: Formal and actual parameter lists have to match

// A function definition with a formal parameter list

**function** MyFunction(**integer** FormalPar1, **boolean** FormalPar2, **bitstring** FormalPar3) { … }

// A function call with an actual parameter list

MyFunction(123, **true**,'1100'B);

// A function call with assignment notation for actual parameters

MyFunction(FormalPar1 := 123, FormalPar3 := '1100'B, FormalPar2 := **true**);

EXAMPLE 2: In parameters

**function** MyFunction(**in** **template** MyTemplateType MyValueParameter){ … };

// MyValueParameter is in parameter, the in keyword is optional

// A function call with an actual parameter

MyFunction(MyGlobalTemplate);

EXAMPLE 3: Inout and out parameters

**function** MyFunction(**inout boolean** MyReferenceParameter){ … };

// MyReferenceParameter is an inout parameter

// A function call with an actual parameter

MyFunction(MyBooleanVariable);

// The actual parameter can be read and set within the function

**function** MyFunction(**out template boolean** MyReferenceParameter){ … };

// MyReferenceParameter is an out parameter

// A function call with an actual parameter

MyFunction(MyBooleanVariable);

// The actual parameter is initially unbound, but can be set and read within the function.

**type** **record** **of** **integer** RoI;

**function** f\_swapElements (**inout integer** p\_int1, **inout** **integer** p\_int2) {

**var** **integer** v\_tmp := p\_int1;

p\_int1 := p\_int2;

p\_int2 := p\_tmp;

}

**function** f\_testReferences (**inout** RoI p\_roi, **inout** **integer** p\_elem) { … }

:

**var** RoI v\_roi := { 0, 1, 2, 3, 4, 5 };

f\_swapElements(v\_roi[0], v\_roi[5]); // after the function call, v\_roi is { 5, 1, 2, 3, 4, 0 }

f\_testReferences(v\_roi, v\_roi[2]); // produces an error as elements of v\_roi are not allowed

// to be passed by reference if the parent structure (v\_roi) is passed by reference too.

EXAMPLE 4: Empty parameter lists

// A function definition with an empty parameter list shall be written as

**function** MyFunction(){ … }

// and shall be called as

MyFunction();

// A record definition with an empty parameter list shall be written as

**type record** MyRecord { … }

// and shall be used as

**template** MyRecord Mytemplate := { … }

EXAMPLE 5: Nested parameter lists

// Given the message definition

**type record** MyMessageType

{

**integer** field1,

**charstring** field2,

**boolean** field3

}

// A message template might be

**template** MyMessageType MyTemplate(**integer** MyValue) :=

{

field1 := MyValue,

field2 := **pattern** "abc\*xyz",

field3 := **true**

}

// A test case parameterized with a template might be

**testcase** TC001(**template** MyMessageType RxMsg) **runs on** PTC1 **system** TS1 {

:

MyPCO.**receive**(RxMsg);

}

// When the test case is called in the control part and the parameterized template is

// passed as an actual parameter, the template's actual parameters shall be provided

**control**

{ :

**execute**(TC001(MyTemplate(7)));

:

}

EXAMPLE 6: A typical use case for lazy parameterization

**modulepar** **boolean** logMessage := true;

**function** logMsg(**@lazy** **charstring** complex) {

**if** (logMessage) {

**log**(complex);

}

}

**function** computeComplexMessage() **return charstring** {

// some complicated computation

}

logMsg(computeComplexMessage()); // computeComplexMessage() is only invoked if

// logMessage is false

EXAMPLE 7: Actual parameters passed to lazy and fuzzy formal parameters

**type** **record** MyMessage { **integer** id, **float** number }

**type** **port** MyPortType **message** { **inout** MyMessage }

**type** **component** MyMTC {

**var** **integer** v\_id;

**port** MyPortType P;

}

**testcase** TC\_shootingMessages () **runs** **on** MyMTC {

**connect**(**self**:P,**self**:P);

sendLazy({v\_id, rnd()}); //note that at this point v\_id is unintialized yet

sendFuzzy({v\_id, rnd()})

}

**function** sendLazy(@lazy MyMessage pdu) **runs** **on** MyMTC {

**for** (v\_id := 1; v\_id<9; v\_id:=v\_id+1){

P.**send**(pdu); // the actual parameter passed to the formal parameter pdu is evaluated only in

// the first loop;let say rnd() returns 0.924946; the message { 1, 0.924946 } is

// sent out 8 times

}

**setverdict**(**pass**,"messages has been sent out")

}

**function** sendFuzzy(@fuzzy MyMessage pdu) **runs** **on** MyMTC {

**for** (v\_id := 1; v\_id<9; v\_id:=v\_id+1){

P.**send**(pdu); // the actual parameter passed to the formal parameter pdu is evaluated in each

// loop; let say rnd() returns 0.924946, 0.680497, 0.630836, 0.648681, 0.428501,

// 0.262539, 0.646990, 0.265262 in subsuent calls; the messages 1, 0.924946 },

// {{ 2, 0.680497 }, { 3, 0.630836 }, { 4, 0.648681 }, { 5, 0.428501 },

// { 6, 0.262539 }, { 7, 0.646990 } and { 8, 0.265262 } are sent out in sequence

}

**setverdict**(**pass**,"messages has been sent out")

}

EXAMPLE 8: Order of out parameters

**function** f\_initValues (**out** **integer** p\_par1, **out** **integer** p\_par2) {

p\_par1 := 1**;**

p\_par2 := 2;

}

**function** f(){

**var** **integer** v\_var1;

f\_initValues(p\_par2 := v\_var1, p\_par1 := v\_var1);

// After this function call, v\_var1 will contain 2, as parameters are assigned in

// the same order as in the definition of the f\_initValues function. Thus p\_par1 is

// assigned first to v\_var1 and p\_par2 after that ovewriting the previous value.

}