ETSI ES 201 873-1 V4.10.1 (2018-05)

Methods for Testing and Specification (MTS);

The Testing and Test Control Notation version 3;

Part 1: TTCN‑3 Core Language

**ETSI Standard**

Reference

RES/MTS-201873-1v4A1

Keywords

language, methodology, testing, TTCN-3

***ETSI***

650 Route des Lucioles

F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C

Association à but non lucratif enregistrée à la

Sous-Préfecture de Grasse (06) N° 7803/88

***Important notice***

The present document can be downloaded from:  
<http://www.etsi.org/standards-search>

The present document may be made available in electronic versions and/or in print. The content of any electronic and/or print versions of the present document shall not be modified without the prior written authorization of ETSI. In case of any existing or perceived difference in contents between such versions and/or in print, the only prevailing document is the print of the Portable Document Format (PDF) version kept on a specific network drive within ETSI Secretariat.

Users of the present document should be aware that the document may be subject to revision or change of status. Information on the current status of this and other ETSI documents is available at <https://portal.etsi.org/TB/ETSIDeliverableStatus.aspx>

If you find errors in the present document, please send your comment to one of the following services:  
<https://portal.etsi.org/People/CommiteeSupportStaff.aspx>

***Copyright Notification***

No part may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm except as authorized by written permission of ETSI.  
The content of the PDF version shall not be modified without the written authorization of ETSI.  
The copyright and the foregoing restriction extend to reproduction in all media.

© ETSI 2018.

All rights reserved.

**DECT**TM, **PLUGTESTS**TM, **UMTS**TM and the ETSI logo are trademarks of ETSI registered for the benefit of its Members.  
**3GPP**TM and **LTE**TM are trademarks of ETSI registered for the benefit of its Members and  
of the 3GPP Organizational Partners.  
**oneM2M** logo is protected for the benefit of its Members.  
**GSM**® and the GSM logo are trademarks registered and owned by the GSM Association.

### 5.4.1 Formal parameters

#### 5.4.1.0 General

TTCN-3 modules, structured types, templates, functions, altsteps, and testcases may be defined incompletely, i.e. some entities (variables, templates, ports, timers, etc.) used by the above objects may not be resolved in the definition of the object. These objects are called parameterized objects. Formal entities replacing the unresolved entities in the parameterized object's definition are called formal parameters.

Formal parameters of parameterized templates, functions, altsteps, and testcases are defined in formal parameter lists. Formal parameters of modules are defined in module parameter definitions (see clause 8.2.1).

Formal parameters shall be **in**, **inout** or **out** parameters (see definitions in clause 3.1). If not stated otherwise, a formal parameter is an **in** parameter. For all these three sorts of parameter passing, the formal parameters can both be read and set (i.e. get new values being assigned) within the parameterized object. Formal parameters can be used directly as actual parameters for other parameterized objects, e.g. as actual parameters in function invocations or as actual parameters in template instances.

If parameters are passed by value (i.e. in case of **in** and **out** parameters), type compatibility rules specified in clause 6.3 apply. When parameters are passed by reference, strong typing is required. Both the actual and formal parameter shall be of the same type.

Formal **in** parameters may have default values. This default value is used when no actual parameter is provided.

NOTE 1: Although **out** parameters can be read within the parameterized object, they do not inherit the value of their actual parameter; i.e. they should be set before they are read.

Formal value or template parameters may be declared lazy using the **@lazy** modifier. The behaviour of lazy parameters is defined in clause 3.1, definition of lazy values or templates. See examples in clause 5.4.1.1.

Formal value or template parameters may be declared fuzzy using the **@fuzzy** modifier. The behaviour of lazy parameters is defined in clause 3.1, definition of fuzzy values or templates. See examples in clause 5.4.1.1.

Formal value or template parameters that are declared lazy or fuzzy can be additionally declared deterministic using the **@deterministic** modifier.

NOTE 2: The actual values of component variables used in the delayed evaluation of a lazy or fuzzy parameter may differ from their values at the time, when the parameterized function or alstep was called.

Assigning default values for lazy and fuzzy formal parameters does not change the parameters' semantics: when the default values are used as actual values for the parameters, they shall be evaluated the same way (i.e. delayed) as if an actual parameter was provided.

Lazy and fuzzy properties are valid only in the scope, where the parameters' names are visible. For example, if a fuzzy parameter is passed to a formal parameter declared without a modifier, it loses its fuzzy feature inside the called function. Similarly, if it is passed to a lazy formal parameter, it becomes lazy within the called function.

#### 5.4.1.1 Formal parameters of kind value

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

***Syntactical Structure***

[ ( **in** | **inout** | **out** ) ] [ ( **@lazy** | **@fuzzy ) [@deterministic]** ]

*Type* *ValueParIdentifier* [ArrayDef] [":=" ( *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 1: 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 2: 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 formal parameter's default value has to be compatible with the type of the parameter. The expression may be any expression that is well-defined at the beginning of the scope of the parameterized entity, but shall not refer to other parameters of the same parameter list.

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

g) Default values of port, timer or 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.

l) Only **function** and **altstep** definitions with the exception of functions or altsteps started as test component behaviour (see clause 21.3.2) may have formal parameters of a port, timer or default type or of a type that contains a direct or indirect element or field of a port, default or timer type.

m) Only **function**, **altstep** and **testcase** definitions may have formal parameters of a component type or of a type that contains a direct or indirect element or field of a component type.

n) If a lazy or fuzzy value parameter is used in deterministic contexts (i.e. during the evaluation of a snapshot or initialization of global non-fuzzy templates), it shall be declared @deterministic.

***Examples***

EXAMPLE 1: In, out and inout formal parameters

**function** f\_myFunction1(**in boolean** p\_myReferenceParameter){ … };

// p\_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** f\_myFunction2(**inout boolean** p\_myReferenceParameter){ … };

// p\_myReferenceParameter is an inout value parameter. The parameter can be read and set

// within the function - the assignment is not local

**function** f\_myFunction3(**out template boolean** p\_myReferenceParameter){ … };

// p\_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** v\_f1, v\_f2;

v\_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,

v\_f2 := p\_int;

**return** {v\_f1, v\_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\_v := p\_int1 + p\_int2;

**return** v\_v;

}

**function** f\_f () {

**var** **integer** v\_w;

v\_w := f\_comp(1); // same as calling f\_comp(1,3);

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

…

}

type **component** Comp { **var** **integer** i := 0 }

**function** g(**integer** x := f\_comp(i)) **runs on** Comp return **integer** {

// reference to i from Comp is allowed in default value of parameter x

**return** x;

}

**function** h(**integer** y := g()+i) **runs** **on** Comp {

// reference to g is allowed because it has a compatible runs on clause as h

}

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** vc\_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 vc\_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 vc\_int equals 2 is called

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

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

}

**function** f\_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 {

vc\_int := 0;

f\_myLazyFuzzy (f\_double(vc\_int), f\_double(vc\_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\_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

}

#### 5.4.1.2 Formal parameters of kind template

Template kind parameters are used to pass templates into parameterizable objects.

***Syntactical Structure***

[ **in** | **inout** | **out** ] **template** [ *Restriction* ] [ ( **@lazy** | **@fuzzy ) [@deterministic]** ]

*Type* *ValueParIdentifier* [[ArrayDef](file:///C:\\Users\\ethgry\\AppData\\Local\\Microsoft\\Windows\\Temporary%20Internet%20Files\\Content.IE5\\0EG46CRK\\CR7496-v1.docx" \l "TArrayDef)] [":=" ( *TemplateInstance* | "-" )]

***Semantic Description***

Template parameters can be defined for templates, functions, altsteps, and test cases.

To enable a parameterized object to accept templates or matching symbols as actual parameters, the extra keyword **template** shall be added before the type field of the corresponding formal parameter. This makes the parameter a template parameter and in effect extends the allowed actual parameters for the associated type to include the appropriate set of matching attributes (see annex B) as well as the normal set of values.

Formal template parameters can be used within the parameterized object the same way as templates and template variables.

Formal template parameters may be in, inout or out parameters. The default for formal template parameters is **in** parameterization.

In parameters may have a default template, which is given by a template instance assigned to the parameter. Formal template parameters of modified templates may inherit their default templates 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 parameter's default template. If a default template is used, it is evaluated in the scope of the parameterized entity, not the scope of the actual parameter list.

Formal template parameters can be restricted to accept actual parameters containing a restricted set of matching mechanisms only. Such limitations can be expressed by the restrictions **omit**, **present**, and **value**. The restriction **template (omit)** can be replaced by the shorthand notation **omit**. The meaning of the restrictions is explained in clause 15.8.

***Restrictions***

a) Only **function, testcase**, **altstep** and **template** definitions may have formal template parameters.

b) Formal template parameters of **templates** and of **altstep**s activated as defaults (see clause 20.5.2) shall always be **in** parameters.

c) Default templates can be provided for in parameters only.

d) The default template instance has to be compatible with the type of the parameter. The template instance may be any template expression that is well-defined at the beginning of the scope of the parameterized entity, but shall not refer to other parameters in the same parameter list.

e) Default templates of component type formal parameters shall be built from the special values **null, mtc, self**, or **system**.

f) Restrictions specified in clause 15 shall apply.

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

h) Only in template parameters can be declared lazy or fuzzy.

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

j) If a lazy or fuzzy template parameter is used in deterministic contexts (i.e. during the evaluation of a snapshot or initialization of global non-fuzzy templates), it shall be declared @deterministic.

### 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 or in a mixed notation where the first parameters are given in list notation and additional parameters in assignment notation.

***Syntactical Structure***

( *Expression* | // for value parameter

*TemplateInstance* | // for template 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, value variables, invocations of 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 **out** formal value parameters shall be (template) variables, formal (template) parameters (of in, inout or out parameterization) or references to elements of (template) variables or formal (template) parameters of structured types. Furthermore it is allowed to use the dash symbol "-" as an actual **out** parameter, signifying that a possible result for that parameter will not be passed back.

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 1: 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, invocations of 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 template variables, formal template parameters or references to elements of template variables or formal template parameters of structured types. Furthermore it is allowed to use the dash symbol "-" as an actual **out** parameter, signifying that a possible result for that parameter will not be passed back.

Actual parameters that are passed to **inout** formal template parameters shall be template variables or formal 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. The value is passed back to the actual parameter only if within the invoked object a value is assigned to it. If no value is assigned, the actual parameter remains unchanged when the invoked object completes.

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).

NOTE 2: Because inout parameters are passed by reference and component variables are effectively also accessed by reference within a called function or altstep, passing parts of a structured component variable as an actual inout parameter may have confusing effects inside the parameterized behaviour: changing either the inout parameter or the component variable may also change the other simultaneously, which might break the intended algorithm. For this reason, such situations should be avoided.

When a formal parameter is an **out** parameter or 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.

The actual parameters are evaluated in the order of their appearance. If for some formal parameters, no actual parameter has been provided, if they are **out** parameters, the dash symbol "-" and for **in** parameters their default values are taken. Default values are evaluated after the evaluation of the actual parameters and the order of their evaluation corresponds to their order in the formal parameter list.

NOTE 3: If assignment notation has been used for the actual parameter list, the order of the evaluation of actual parameters may differ from the order of the parameters in the 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 **inout** parameter and for each **in** parameter without a default there shall be an actual parameter. The actual parameter of a formal **out** parameter or **in** 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 are **out** parameters or have default values and are left out. The number of actual parameters in the list notation shall not exceed the number of parameters in the formal parameter list.

b) Void.

c) When using assignment notation, each formal parameter shall be assigned an actual parameter at most once. For each assigned actual parameter there shall exist a corresponding formal parameter of the same name. 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) For **in** formal parameters, the type of the actual parameter shall be compatible with the type of the formal parameter. For **out** formal parameters, the type of the formal parameter shall be compatible with the type of the actual parameter. Strong typing is required for **inout** formal (parameters passed by reference). For **in** formal template parameters, the template restriction of the actual parameter shall not be less restrictive than the one of the formal parameter. For **out** formal template parameters, the template restriction of the actual parameter shall not be more restrictive than the one of the formal parameter. For **inout** formal template parameters, the template restriction of the actual and the formal parameter shall be the same.

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**, **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.

NOTE 4: **signature** objects also have formal parameters, see clauses 15.2 and 22.3 for their handling.

h) Void.

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 elements 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.

o) If the mixed notation is used, no value list notation shall be used following the first assignment notation and the parameters given in assignment notation shall not assign parameters that already have an actual parameter given in list notation.

p) Actual parameters passed to fuzzy or lazy formal parameters shall fulfill the restrictions imposed on content of functions used in special places given in [16.1.4](#_16.1.4_Invoking_functions).

***Examples***

EXAMPLE 1: Formal and actual parameter lists have to match

// A function definition with a formal parameter list

**function** f\_myFunction(**integer** p\_formalPar1, **boolean** p\_formalPar2, **bitstring** p\_formalPar3) { … }

// A function call with an actual parameter list

f\_myFunction(123, **true**,'1100'B);

// A function call with assignment notation for actual parameters

f\_myFunction(p\_formalPar1 := 123, p\_formalPar3 := '1100'B, p\_formalPar2 := **true**);

EXAMPLE 2: In parameters

**function** f\_myFunction(**in** **template** MyTemplateType p\_myValueParameter){ … };

// p\_myValueParameter is in parameter, the in keyword is optional

// A function call with an actual parameter

f\_myFunction(m\_myGlobalTemplate);

EXAMPLE 3: Inout and out parameters

**function** f\_myFunction(**inout boolean** p\_myReferenceParameter){ … };

// p\_myReferenceParameter is an inout parameter

// A function call with an actual parameter

f\_myFunction(v\_myBooleanVariable);

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

**function** f\_myFunction(**out template boolean** p\_myReferenceParameter){ … };

// p\_myReferenceParameter is an out parameter

// A function call with an actual parameter

f\_myFunction(v\_myBooleanVariable);

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

f\_myFunction(**-**); // the outcoming value is not assigned to a variable

**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 := v\_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.

**function** f\_changeAndIncrement(**out** **integer** p\_e, **in** **integer** p\_v, **inout** **integer** p\_i) {

p\_i := p\_i + 1;

p\_e := p\_v;

}

:

**var** **integer** v\_i := 0;

f\_changeAndIncrement(v\_roi[v\_i], 3, v\_i); // increments p\_i, but still assigns 3 to v\_roi[0]

EXAMPLE 4: A side effect caused by passing part of a component variable as inout parameter

**type** **component** MyComp {

**var** ROI v\_rec := { 0, 1 }

}

**testcase** TC() **runs on** MyComp {

f\_test(v\_rec[1]) // passing 2nd element of component variable as inout parameter

**log**(v\_rec); //will log { 2 , 2 }

}

**function** f\_test(**inout** **integer** p\_int) **runs on** MyComp {

v\_rec := { 2 }; // now, isbound(p\_int) == false

p\_int := 2; // now, v\_rec == { 2, 2 }

}

EXAMPLE 5: Empty parameter lists

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

**function** f\_myFunction(){ … }

// and shall be called as

f\_myFunction();

// A template definition with a default value for a formal parameter written as

**template** MyRecord m\_mytemplate (**integer** p\_myValue:= 1):= { … }

// may be used without actual parameter list (i.e. the default value is used)

myPCO**.send**(m\_mytemplate)

EXAMPLE 6: Nested parameter lists

// Given the message definition

**type record** MyMessageType

{

**integer** field1,

**charstring** field2,

**boolean** field3

}

// A message template might be

**template** MyMessageType mw\_myTemplate(**integer** p\_myValue) :=

{

field1 := p\_myValue,

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

field3 := **true**

}

// A test case parameterized with a template might be

**testcase** TC\_001(**template** MyMessageType p\_rxMsg) **runs on** PTC1 **system** TS1 {

:

myPCO.**receive**(p\_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**(TC\_001(mw\_myTemplate(7)));

:

}

EXAMPLE 7: A typical use case for lazy parameterization

**modulepar** **boolean** PX\_LOG\_MESSAGE := **true**;

**function** f\_logMsg(**@lazy** **charstring** p\_complex) {

**if** (PX\_LOG\_MESSAGE) {

**log**(p\_complex);

}

}

**function** f\_computeComplexMessage() **return charstring** {

// some complicated computation

}

f\_logMsg(f\_computeComplexMessage()); // f\_computeComplexMessage() is only invoked if

// PX\_LOG\_MESSAGE is true

EXAMPLE 8: 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** vc\_id;

**port** MyPortType p;

}

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

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

f\_sendLazy({vc\_id, **rnd**()}); //note that at this point vc\_id is unintialized yet

f\_sendFuzzy({vc\_id, **rnd**()})

}

**function** f\_sendLazy(**@lazy** MyMessage p\_pdu) **runs** **on** MyMTC {

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

p.**send**(p\_pdu); // the actual parameter passed to the formal parameter p\_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** f\_sendFuzzy(**@fuzzy** MyMessage p\_pdu) **runs** **on** MyMTC {

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

p.**send**(pdu); // the actual parameter passed to the formal parameter p\_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 subsequent 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 9: Order of out parameters

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

p\_par1 := 1**;**

p\_par2 := 2;

}

**function** f\_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 overwriting the previous value.

}

EXAMPLE 10: Skipped actual parameters

**function** f\_skip (**out** **integer** p\_par1, **in** **integer** p\_par2 := 2) {

p\_par1 := 1 + p\_par2**;**

}

**function** f\_f(){

// the following statements all have the same semantics :

f\_skip (-,-); // p\_par2 is initialized with default value 2 and

// the result of p\_par1 is not assigned to any variable

f\_skip (p\_par1 := -, p\_par2 := -);

f\_skip (p\_par2 := -); // skip p\_par1

f\_skip (-) ; // skip p\_par2 because it is the last

f\_skip () ; // skip p\_par1 because all following are also skipped

}

EXAMPLE 11: Mixed notation

**function** f\_mixed (**out** **integer** p\_par1, **in** **integer** p\_par2 := 2, **inout** **integer** p\_par3) {

p\_par1 := 1 + p\_par2**;**

}

**function** f\_f(){

**var** **integer** v := 0;

// the following statements all have the same semantics:

f\_mixed(-,2,v);

f\_mixed(-,p\_par2 := 2, p\_par3 := v);

f\_mixed(-,-,p\_par3 := v);

f\_mixed(-,p\_par3 := v, p\_par2 := 2);

// not allowed:

f\_mixed(-,2,p\_par3 := v, p\_par2 := 5); // p\_par2 is already assigned in list notation

}

# 11 Declaring variables

## 11.0 General

TTCN-3 variables are statically typed variables. Variables are either value variables to store values or template variables to store templates.

Variables can be of simple basic types, basic string types, structured types, special data types (including subtypes derived from these types) as well as address, component or default, port or timer types.

Variables can be declared and used in the module control part, test cases, functions and altsteps. Additionally, variables can be declared in component type definitions. These variables can be used in test cases, altsteps and functions which are running on a given component type.

Variables can be declared lazy using the **@lazy** modifier.

Alternatively, variables can be declared fuzzy using the **@fuzzy** modifier.

If a variable is declared fuzzy or lazy they can additionally declared with the **@deterministic** modifier to indicate that when used in a deterministic evaluation context, any evaluation of the variable would have no side effect and would yield the same result.

Lazy and fuzzy features are valid only in the scope, where the variables' names are visible. For example, if a fuzzy variable is passed to a formal parameter declared without a modifier, it loses its fuzzy feature inside the called function. Similarly, if it is passed to a lazy formal parameter, it becomes lazy within the called function.

Whenever a lazy or fuzzy variable is assigned, the TE is required to save the lexical environment (the set of directly or indirectly referenced values and templates) valid at the time of the assignment, so that it is possible to resolve the expression at the time of evaluation of the lazy or fuzzy value or template. If the assignment was made on a lower scope than the evaluation, saving the lexical environment extends lifetime of the referenced variables defined on that lower scope.

***Example***

**var** **@fuzzy** **integer** v\_fuzzy := 1;

**var integer** v\_var;

**var** **boolean** v\_condition := **true**;

**if** (v\_condition) {

**var integer** v\_local := 0;

v\_fuzzy := v\_local;

v\_local := 10;

}

// although v\_local is no longer valid at this point, v\_fuzzy still evaluates to 10 because

// the lexical environment is available to the fuzzy variable:

v\_var := v\_fuzzy;

## 11.1 Value variables

A TTCN-3 value variable stores values. It is declared by the **var** keyword followed by a type identifier and a variable identifier. An initial value can be assigned at variable declaration.

It may be used at the right hand side as well as at the left hand side of assignments, in expressions, following the **return** keyword in bodies of functions with a return clause in their headers and may be passed to both value and template-type formal parameters.

***Syntactical Structure***

**var** [ ( **@lazy** | **@fuzzy )** [ **@deterministic** ] ] *Type*

{ *VarIdentifier* [ *ArrayDef* ] [ ":=" *Expression* ] [ "," ] }+ [ ";" ]

***Semantic Description***

A value variable associates a name with the location of a value. A value variable may change its value during test execution several times. A value can be assigned several times to a value variable. The value variable can be referenced multiple times in a TTCN-3 module.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clause 5, the following restrictions apply:

a) *Expression* shall be of type *Type*.

b) Value variables shall store values only.

c) Value variables shall not be declared or used in a module definitions part (i.e. global variables are not supported in TTCN‑3).

d) Use of uninitialized value variables at other places than the left hand side of assignments, in return statements, or as actual parameters passed to formal parameters shall cause an error.

e) The initialization or assignment of a fuzzy or lazy variable shall not contain function calls of functions with inout or out parameters. The called functions may use other functions with inout or out parameters internally.

f) If a lazy or fuzzy value variable is used in deterministic contexts (i.e. during the evaluation of a snapshot or initialization of global non-fuzzy templates), it shall be declared @deterministic and the expression assigned to the variable shall fulfill the restrictions imposed on content of functions used in special places given in [16.1.4](#_16.1.4_Invoking_functions).

g) The expression assigned to a lazy or fuzzy variable might contain a direct or indirect reference to this variable. Evaluation of such an expression shall cause a dynamic error.

h) Using the dot notation (see clauses 6.2.1.1, 6.2.2.1 and 6.2.5.1) and index notation (see clauses 6.2.7 and 6.2.7) for referencing a field, alternative or element of an **address** value, which actual value is **null** shall cause an error.

i) The expression shall evaluate to a value, which is at least partially initialized.

***Examples***

**var** **integer** v\_myVar0;

**var** **integer** v\_myVar1 := 1;

**var** **boolean** v\_myVar2 := **true**, v\_myVar3 := **false**;

**var** @**lazy** **integer** v\_myLazyVar1 := v\_myVar1+1;

**var** **timer** v\_timer1;

**timer** t\_myTimer1; v\_myVar1 := 2;

v\_myVar1 := v\_myLazyVar1; // v\_myLazyVar1 evaluates to 2 + 1

v\_myLazyVar1 := v\_myLazyVar1 + 1;

v\_myVar1 := v\_myLazyVar1; // causes an error as v\_myLazyVar1 references itself

v\_timer1 := t\_myTimer1;

## 11.2 Template variables

A TTCN-3 template variable stores templates. They are declared by the **var** **template** keyword followed by a type identifier and a variable identifier. An initial content can be assigned at declaration. In addition to values, template variables may also store matching mechanisms (see clause 15.7).

Template variables may be used on the right hand side as well as on the left hand side of assignments, following the **return** keyword in bodies of functions defining a template-type return value in their headers and may be passed as actual parameters to template-type formal parameters. It is also allowed to assign a template instance to a template variable or a template variable field.

***Syntactical Structure***

**var template** [ *restriction* ] [ (**@lazy** | **@fuzzy) [ @deterministic ]** ]

*Type* { *VarIdentifier* [ *ArrayDef* ] ":=" *TemplateBody* [ "," ] }+ [ ";" ]

***Semantic Description***

A template variable associates a name with the location of a template or a value (as every value is also a template).   
A template variable may change its template during test execution several times. A template or value can be assigned several times to a template variable. The template variable can be referenced multiple times in a TTCN-3 module.

The content of a template variable can be restricted to the matching mechanisms specific value and omit in the same way as formal template parameters, see clause 5.4.1.2. The restriction **template (omit)** can be replaced by the shorthand notation **omit**.

NOTE 1: String and list type templates can be concatenated, see clause 15.11.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clause 5, the following restrictions apply:

a) Template variables shall not be declared or used in a module definitions part (i.e. global variables are not supported in TTCN‑3).

b) When used on the right hand side of assignments template variables shall not be operands of TTCN‑3 operators (see clause 7.1) and the variable on the left hand side shall be a template variable too.

c) When accessing element of template variables either on the left hand side or on the right hand side of assignments, the rules given in clause 15.6 shall apply.

NOTE 2: While it is not allowed to directly apply TTCN‑3 operations to template variables, it is allowed to use the dot notation and the index notation to inspect and modify template variable fields.

d) Use of uninitialized template variables at other places than the left hand side of assignments, in return statements, or as actual parameters passed to formal parameters shall cause an error.

e) Void.

f) If the template variable is restricted, then the template used to initialize it shall contain only the matching mechanisms as described in clause 15.8.

g) Template variables, similarly to global and local templates, shall be fully specified in order to be used in sending and receiving operations.

h) Restrictions on templates in clause 15 shall apply.

i) The initialization or assignment of a fuzzy or lazy variable shall not contain function calls of functions with inout or out parameters. The called functions may use other functions with inout or out parameters internally.

j) If a lazy or fuzzy template variable is used in deterministic contexts (i.e. during the evaluation of a snapshot or initialization of global non-fuzzy templates), it shall be declared **@deterministic** and the template body assigned to the variable shall fulfill the restrictions imposed on content of functions used in special places given in [16.1.4](#_16.1.4_Invoking_functions).

k) Using the dot notation (see clauses 6.2.1.1, 6.2.2.1 and 6.2.5.1) and index notation (see clause 6.2.7) for referencing a field, alternative or element of an **address** value, which actual value is **null** shall cause an error.

l) The template body at the right-hand side of the assignment symbol shall evaluate to a value or template, which is type compatible with the variable being declared.

m) The template body at the right-hand side of the assignment symbol shall evaluate to an object that is at least partially initialized.

# 15 Declaring templates

## 15.0 General

Templates are used to either transmit a set of distinct values or to test whether a set of received values matches the template specification. Templates can be defined globally or locally.

Templates provide the following possibilities:

1. they are a way to organize and to re-use test data, including a simple form of inheritance;
2. they can be parameterized;
3. they allow matching mechanisms;
4. they can be used with either message-based or procedure-based communications.

Within a template values, ranges and matching attributes can be specified and then used in both message-based and procedure-based communications. Templates may be specified for any TTCN‑3 type or procedure signature. The type‑based templates are used for message-based communications and the signature templates are used in procedure‑based communications.

A template can be declared fuzzy using the **@fuzzy** modifier.

NOTE 1: Using a fuzzy template from a non-fuzzy template causes evaluation of the fuzzy template. Thus, for unparameterized non-fuzzy templates, the result of the used fuzzy templates will stay the same for every usage.

A fuzzy template can be declared deterministic using the @deterministic modifier. A deterministic template shall be evaluated to the same result in the same deterministic evaluation context whenever it is evaluated.

A modified template declaration (see clause 15.5) specifies only the fields to be changed from the base template, i.e. it is a partial specification.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clause 5, the following restrictions apply:

1. Templates shall not be of **default,** port or timer type.
2. Templates shall not be of a structured type that contains fields of **default**, port or timer type on any level of nesting.

NOTE 2: The **anytype** type does not include the **default** type, port and timer types (see clause 6.2.6), so that restriction b) does not apply to anytype templates.

1. The expression or template body initializing a template shall evaluate to a value or template, which is type compatible with the template being declared.
2. The expression or template body initializing a template shall evaluate to a value or a template that is at least partially initialized or to a matching mechanism.
3. The body of a fuzzy template shall not contain function calls of functions with inout or out parameters. The called functions may use other functions with inout or out parameters internally.
4. Fuzzy features are valid only in the scope, where the templates' names are visible. For example, if a fuzzy template is passed to a formal template parameter declared without a modifier, it loses its fuzzy feature inside the called function.
5. For a fuzzy template that is declared deterministic the template body shall fulfill the restrictions imposed on content of functions used in special places given in [16.1.4](#_16.1.4_Invoking_functions).

## 15.3 Global and local templates

TTCN-3 allows defining global templates and local templates.

***Syntactical Structure***

**template** [ *restriction* ] [ **@fuzzy** [ @deterministic ] ] *Type* *TemplateIdentifier*

["(" *TemplateFormalParList* ")"]

[ **modifies** *TemplateRef* ] ":=" *TemplateBody*

NOTE: The optional restriction part is covered by clause 15.8.

## 15.5 Modified templates

In cases where small changes are needed to specify a new template, it is possible to specify a modified template. A modified template specifies modifications to particular fields of the original template, either directly or indirectly. As well as creating explicitly named modified templates, TTCN‑3 allows the definition of in-line modified templates.

***Syntactical Structure***

Global or local modified template:

**template** [*restriction*] [ **@fuzzy** [ @deterministic ]] *Type* *TemplateIdentifier*

["(" *TemplateFormalParList* ")"]

**modifies** *TemplateRef* ":=" *TemplateBody*

### 16.1.4 Invoking functions from specific places

If value returning functions are called in receiving communication operations (in templates, template fields, in-line templates as actual parameters or when evaluating the port expression), in timeout operations (when evaluating the timer expression), in test component operations (in guards or events of alt statements or altsteps, see clause 20.2), or in initializations of altstep local definitions (see clause 16.2), the following operations shall not be present in functions called in the cases specified above, in order to avoid side effects that cause changing the state of the component or the actual snapshot and to prevent different results of subsequent evaluations on an unchanged snapshot:

1. All component operations, i.e. **create**, **start**(component), **stop**(component), **kill**, **running**(component), **alive,** **done** and **killed** (see notes 1, 3, 4 and 6).
2. All port operations, i.e. **start**(port), **stop**(port), **halt**, **clear**, **checkstate**, **send**, **receive**, **trigger**, **call**, **getcall**, **reply**, **getreply**, **raise**, **catch**, **check**, **connect**, **disconnect**, **map** and **unmap** (see notes 1, 2, 3, 4 and 6).
3. The **action** operation (see notes 2 and 6).
4. All timer operations, i.e. **start**(timer), **stop**(timer), **running**(timer), **read**, **timeout** (see notes 4 and 6).
5. Calling non-deterministic external functions, i.e. external functions where the resulting values for actual inout or out parameters or the return value may differ for different invocations with the same actual in and inout parameters (see notes 4 and 6).
6. Calling the **rnd** predefined function (see notes 4 and 6).
7. Changing of component variables, i.e. using component variables on the left-hand side of assignments, and in the instantiation of **out** and **inout** parameters (see notes 4 and 6).
8. Calling the **setverdict** operation (see notes 4 and 6).
9. Activation and deactivation of defaults, i.e. the **activate** and **deactivate** statements (see notes 5 and 6).
10. Calling functions and deterministic external functions with **out** or **inout** parameters (see notes 7 and 8).
11. Calling functions and external functions with **@fuzzy** formal parameters and variables (see notes 4 and 9).
12. The **setencode** operation (see note 8 and clause 27.9).
13. Referencing lazy or fuzzy variables, parameters or templates that have not been declared deterministic.

## 19.10 The Return statement

The **return** statement terminates execution of functions or altsteps.

***Syntactical Structure***

**return** [ *Expression* | *TemplateInstance* ]

***Semantic Description***

The **return** statement terminates execution of a function or altstep and returns control to the point from which the function or altstep was called. When used in functions, a **return** statement may be optionally associated with a return value or template.

TTCN-3 allows optional statement blocks that may follow altstep calls within **alt** statements. If there is a statement block, the **return** statement returns control to the beginning of this statement block and the statement block is executed before the **alt** statement is left. If there is no statement block, test execution continues with the first statement following the **alt** statement.

The return value or template is first evaluated before returning.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 16, the following restrictions apply:

1. The return statement shall not be present in testcase definitions.

## 20.4 The Interleave statement

The **interleave** statement allows to specify the interleaved occurrence and handling of receiving events including **done**, **killed**, **timeout**, **receive**, **trigger**, **getcall**, **getreply**, **catch** and **check.**

***Syntactical Structure***

**interleave** "{"

{ "[]" ( *TimeoutStatement* |

*ReceiveStatement* |

*TriggerStatement* |

*GetCallStatement* |

*CatchStatement* |

*CheckStatement* |

*GetReplyStatement* |

*DoneStatement* |

*KilledStatement* ) *StatementBlock*

}

"}"

***Semantic Description***

The **interleave** statement allows to specify the interleaved occurrence and handling of the statements **done**, **killed**, **timeout**, **receive**, **trigger**, **getcall**, **getreply**, **catch** and **check**.

Interleaved behaviour can always be replaced by an equivalent set of nested **alt** statements. The procedures for this replacement and the operational semantics of interleaving are described in part 4 of the TTCN‑3 standard (ETSI ES 201 873‑4 [1]).

The rules for the evaluation of an interleaving statement are the following:

1. Whenever a reception statement is executed, the following non-reception statements are subsequently executed until the next reception statement is reached, a **break** statement is reached, or the interleaved sequence ends.

NOTE 1: Reception statements are TTCN‑3 statements which may occur in sets of alternatives, i.e. **receive**, **check**, **trigger**, **getcall**, **getreply**, **catch**, **done,** **killed** and **timeout**. Non-reception statements denote all other non-control-transfer statements which can be used within the **interleave** statement.

1. If none of the alternatives of the **interleave** statement can be executed, the default mechanism will be invoked. This means, according to the semantics of the default mechanism, the actual snapshot will be used to evaluate those altsteps that have been activated before entering the **interleave** statement.

NOTE 2: The complete semantics of the default mechanism within an **interleave** statement is given by replacing the **interleave** statement by an equivalent set of nested **alt** statements. The default mechanism applies for each of these **alt** statements.

1. The evaluation then continues by taking the next snapshot if no **break** statement was encountered.
2. The evaluation of the **interleave** statement is terminated if a **break** statement is executed.

The operational semantics of interleaving are fully defined in part 4 of the TTCN‑3 standard (ETSI ES 201 873‑4 [1]).

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 16, the following restrictions apply:

1. Control transfer statements **activate**, **deactivate**, **repeat**, all calls of altsteps and (direct and indirect) calls of user-defined functions, which include reception statements, shall not be present in **interleave** statements.
2. In addition, it is not allowed to guard branches of an **interleave** statement with Boolean expressions (i.e. the '[ ]' shall always be empty). It is also not allowed to specify **else** branches in interleaved behaviour.
3. An **interleave** statement used within the module control part shall only contain **timeout** statements.
4. The restricted use of the control transfer statements **for**, **while**, **do-while**, and **goto** within **interleave** statements is allowed under the following conditions:
   1. The loop statements **for**, **while**, and **do-while** can be used within statements blocks that do not contain reception statements.
   2. The **goto** statement can be used for defining jumps with statements blocks that do not contain reception statements and for specifying jumps out of **interleave** statements.

# 21 Configuration Operations

## 21.0 General

Configuration operations are used to set up and control test components and their connections. They are summarized in table 20.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clause 5, the following restrictions apply:

1. These operations shall only be used in:

* TTCN‑3 test cases;
* behaviours invoked directly or indirectly from a test case or from a behaviour started on a ptc.

1. They shall not be present in:

* the module control part;
* functions or altsteps invoked directly or indirectly from the module control part;
* declarations inside component type definitions; or
* functions invoked directly or indirectly from declarations inside component type definitions.

Table 20: Overview of TTCN‑3 configuration operations

| Operation | Explanation | Syntax Examples |
| --- | --- | --- |
| **Connection Operations** | | |
| connect | Connects the port of one test component to the port of another test component | **connect**(ptc1:p1, ptc2:p2); |
| disconnect | Disconnects two or more connected ports | **disconnect**(ptc1:p1, ptc2:p2); |
| map | Maps the port of one test component to the port of the test system interface | **map**(ptc1:q, **system**:sutPort1); |
| unmap | Unmaps two or more mapped ports | **unmap**(ptc1:q, **system**:sutPort1); |
| **Test Component Operations** | | |
| create | Creation of a normal or alive test component, the distinction between normal and alive test components is made during creation (MTC behaves as a normal test component) | Non-alive test components:  **var** PTCType c := PTCType.**create**;  Alive test components:  **var** PTCType c := PTCType.**create** **alive**; |
| start | Starting test behaviour on a test component, starting a behaviour does not affect the status of component variables, timers or ports | c.**start**(PTCBehaviour()); |
| stop | Stopping test behaviour on a test component | c.**stop**; |
| kill | Causes a test component to cease to exist | c.**kill**; |
| alive | Returns true if the test component has been created and is ready to execute or is executing already a behaviour; otherwise returns false | **if** (c.**alive**) … |
| running | Returns true as long as the test component is executing a behaviour; otherwise returns false | if (c.**running**) … |
| done | Checks whether the function running on a test component has terminated | c.**done**; |
| killed | Checks whether a test component has ceased to exist | c.**killed** { … } |
| **Test Case Operations** | | |
| stop | Terminates the test case with the test verdict **error** | **testcase**.**stop** ( … ); |
| **Reference Operations** | | |
| mtc | Gets the reference to the MTC | **connect**(**mtc**:p, ptc:p); |
| system | Gets the reference to the test system interface | **map**(c:p, **system**:sutPort); |
| self | Gets the reference to the test component that executes this operation | **self**.**stop**; |

#### A.1.6.1.3 Template definitions

TemplateDef ::= [TemplateKeyword](#TTemplateKeyword) [[TemplateRestriction](#TTemplateRestriction)] [[FuzzyModifier](#TFuzzyModifier) [DeterministicModifier]]

[BaseTemplate](#TBaseTemplate) [[DerivedDef](#TDerivedDef)] [AssignmentChar](#TAssignmentChar) [TemplateBody](#TTemplateBody)

BaseTemplate ::= ([Type](#TType) | [Signature](#TSignature)) [Identifier](#TIdentifier) ["(" [TemplateOrValueFormalParList](#TTemplateOrValueFormalParList)

")"]

### A.1.6.3 Local definitions

#### A.1.6.3.1 Variable instantiation

VarInstance ::= [VarKeyword](#TVarKeyword) (([([LazyModifier](#TLazyModifier) | [FuzzyModifier](#TFuzzyModifier)) [DeterministicModifier]]

[Type](#TType) [VarList](#TVarList)) |

(([TemplateKeyword](#TTemplateKeyword) | [RestrictedTemplate](#TRestrictedTemplate))

[([LazyModifier](#TLazyModifier) | [FuzzyModifier](#TFuzzyModifier)) [DeterministicModifier]]

[Type](#TType) [TempVarList](#TTempVarList)))

VarList ::= [SingleVarInstance](#TSingleVarInstance) {"," [SingleVarInstance](#TSingleVarInstance)}

SingleVarInstance ::= [Identifier](#TIdentifier) [[ArrayDef](#TArrayDef)] [[AssignmentChar](#TAssignmentChar) [Expression](#TExpression)]

VarKeyword ::= "var"

TempVarList ::= [SingleTempVarInstance](#TSingleTempVarInstance) {"," [SingleTempVarInstance](#TSingleTempVarInstance)}

SingleTempVarInstance ::= [Identifier](#TIdentifier) [[ArrayDef](#TArrayDef)] [[AssignmentChar](#TAssignmentChar) [TemplateBody](#TTemplateBody)]

ValueRef ::= [Identifier](#TIdentifier) [[ExtendedFieldReference](#TExtendedFieldReference)]

### A.1.6.7 Parameterization

InParKeyword ::= "in"

OutParKeyword ::= "out"

InOutParKeyword ::= "inout"

FormalValuePar ::= [[InParKeyword](#TInParKeyword) | [InOutParKeyword](#TInOutParKeyword) | [OutParKeyword](#TOutParKeyword)]

[([LazyModifier](#TLazyModifier) | [FuzzyModifier](#TFuzzyModifier)) [DeterministicModifier]]

[Type](#TType) [Identifier](#TIdentifier) [[ArrayDef](file:///C:\\Users\\ethgry\\AppData\\Local\\Microsoft\\Windows\\Temporary%20Internet%20Files\\Content.IE5\\0EG46CRK\\CR7496-v1.docx" \l "TArrayDef)] [":=" ([Expression](#TExpression) | [Minus](#TMinus))]

FormalTemplatePar ::= [[InParKeyword](#TInParKeyword) | [OutParKeyword](#TOutParKeyword) | [InOutParKeyword](#TInOutParKeyword)]

([TemplateKeyword](#TTemplateKeyword) | [RestrictedTemplate](#TRestrictedTemplate))

[([LazyModifier](#TLazyModifier) | [FuzzyModifier](#TFuzzyModifier)) [DeterministicModifier]]

[Type](#TType) [Identifier](#TIdentifier) [[ArrayDef](file:///C:\\Users\\ethgry\\AppData\\Local\\Microsoft\\Windows\\Temporary%20Internet%20Files\\Content.IE5\\0EG46CRK\\CR7496-v1.docx" \l "TArrayDef)] [":=" ([TemplateInstance](#TTemplateInstance) | [Minus](#TMinus))]

RestrictedTemplate ::= [OmitKeyword](#TOmitKeyword) | ([TemplateKeyword](#TTemplateKeyword) [TemplateRestriction](#TTemplateRestriction))

TemplateRestriction ::= "(" ([OmitKeyword](#TOmitKeyword) |

[ValueKeyword](#TValueKeyword) |

[PresentKeyword](#TPresentKeyword)

) ")"