# 3 Definitions and abbreviations

## 3.1 Definitions

For the purposes of the present document, the terms and definitions given in Recommendation ITU‑T X.290 [5], Recommendation ITU‑T X.292 [3] and the following apply:

**actual parameter:** value, expression, template or name reference (identifier) to be passed as parameter to the invoked entity (function, test case, altstep, etc.) as defined at the place of invoking

**assignment notation:** notation that can be used for record, set, record of and set of values, where the fields or the elemens to which a value is assigned are identified explicitly within a pair of curly brackets ("{" and "}") by the field names or the positions of the elements

**basic types:** set of predefined TTCN‑3 types described in clauses 6.1.0 and 6.1.1 of the present document

NOTE: Basic types are referenced by their names.

**communication port:** abstract mechanism facilitating communication between test components

NOTE: A communication port is modelled as a FIFO queue in the receiving direction. Ports can be message‑based or procedure-based.

**compatible type:** TTCN‑3 is not strongly typed but the language does require type compatibility

NOTE: Variables, constants, templates, etc. have compatible types if conditions in clause 6.3 are met.

**completely initialized:** values and templates of simple types are completely initialized if they are partially initialized

NOTE: Values and templates of structured types and arrays are completely initialized if all their fields and elements are completely initialized. In case of an empty record or set type, the value "{}" completely initializes the instances of this type. In case of record of, set of, and array values and templates, this means at least the first n elements are initialized, where n is the minimal length imposed by the type length restriction or array definition (thus in case of n equals 0, the value "{}" also completely initializes a record of, a set of or an array).

**component constant:** constant defined in a component type

**component port:** port defined in a component type

**component template:** template defined in a component type

**component timer:** timer defined in a component type

**component variable:** variable defined in a component type

**data types:** common name for simple basic types, basic string types, structured types, the special data type anytype and all user defined types based on them

NOTE: See table 3 of the present document.

**defined types (defined TTCN‑3 types):** set of all predefined TTCN‑3 types (basic types, all structured types, the type anytype, the address, port and component types and the default type) and all user-defined types declared either in the module or imported from other TTCN‑3 modules

**deterministic function:** function that for the same input in the in and inout parameters always yields the same output both for the return result as well as the inout and out parameters

NOTE 1: A non-deterministic function is one that is not deterministic.

NOTE 2: In general, it cannot be decided if a function is deterministic or not. However, a function can be specified to be deterministic, i.e. the function is supposed to be deterministic. In this case, a violation of the determinism can be detected and handled accordingly. The handling however is tool-specific.

**dynamic parameterization:** form of parameterization, in which actual parameters are dependent on runtime events

EXAMPLE: The value of the actual parameter is a value received during runtime or depends on a received value by a logical relation.

**exception:** in cases of procedure-based communication, an exception (if defined) is raised by an answering entity if it cannot answer a remote procedure call with the normal expected response

**formal parameter:** typed name or typed template reference (identifier) not resolved at the time of the definition of an entity (function, test case, altstep, etc.) but at the time of invoking it

NOTE: Actual values or templates (or their names) to be used at the place of formal parameters are passed from the place of invoking the entity (see also the definition of actual parameter).

**fuzzy value or template:** If a value or template instance is declared to be fuzzy, the expression, initializing or partly initializing it (including actual parameters passed to in formal parameters), is subject to lazy evaluation. During execution, this expression is re-evaluated each time when the fuzzy object is referenced, except when at the left hand side of an assignment or passing it to a fuzzy or lazy formal parameters. The result of this (re)evaluation is used as the actual value or template of the fuzzy instance. When new content is assigned to a fuzzy instance or to its subpart, the right hand side of the assignment is subject to lazy evaluation again.

**global visibility:** attribute of an entity (module parameter, constant, template, etc.) that its identifier can be referenced anywhere within the module where it is defined including all functions, test cases and altsteps defined within the same module and the control part of that module

**implementation conformance statement (ICS):** See Recommendation ITU‑T X.290 [5].

**implementation extra information for testing (IXIT):** See Recommendation ITU‑T X.290 [5].

**implementation under test (IUT):** See Recommendation ITU‑T X.290 [5].

**in parameterization:** kind of parameterization where the value of the actual parameter (the argument) is assigned to the formal parameter when the parameterized object is invoked, but the value of the formal parameter is not passed back to the actual parameter when the invoked object completes

NOTE 1: In **in** parameterization, parameters are passed by value.

NOTE 2: The arguments are evaluated before the parameterized object is entered.

NOTE 3: Only the values of the arguments are passed and changes to the arguments within the invoked object have no effect on the arguments as seen by the invoking object.

**index notation:** notation to access individual elements of record of, set of, array and string values or templates, where the element to be accessed is identified explicitly by an index value enclosed in square brackets ("[" and "]") which specifies the position of that element within the referenced value or template and the index value is either an integer value, array of integers or record of integers

NOTE: Integer values used for indexing (either directly or as elements of the record of or array values) always lie within the index range of the type of the referenced value or template. Except for those arrays which are defined with an explicit index range, the index range always has 0 as the index for the first element.

**initialization:** a value or template, or a value or template field is initialized when a content is first assigned to it. The assignment may be explicit at the declaration of the given object, in which case the same restrictions apply as for the right-hand side of the assignment operation, or at first use on the left-hand side of an assignment, or may be implicit. Implicit initialization is when a yet uninitialized object is passed as actual parameter to an out formal parameter of a directly called testcase, function or altstep returns with a non-uninitialized value or template that is assigned to the actual parameter; or when module parameters not initialized in the TTCN-3 code get their runtime values before test suite execution.

**inout parameterization:** kind of parameterization that uses passing by reference, i.e. when the parameterized object is invoked, the formal parameter is linked with the actual parameter and gets direct access to the same data content that is currently represented by the actual parameter

NOTE 1: The invoked object uses the actual parameter directly, so that all changes made in the formal parameter become immediately effective on the actual parameter. If the same actual parameter is passed to two distinct formal parameters, a change in one formal parameter becomes immediately effective in the other one (and in the actual parameter).

NOTE 2: Inout parameters can be used for functions, altsteps, and test cases only, if not restricted by further rules, e.g. altsteps activated as defaults.

**known types:** set of all TTCN‑3 predefined types, types defined in a TTCN‑3 module and types imported into that module from other TTCN‑3 modules or from non-TTCN‑3 modules

**lazy evaluation:** Lazy evaluation means that evaluation of an expression is delayed during execution until the value or template instance, to which the result of the evaluation should have been assigned or passed to as actual parameter, is first referenced at an other place than the left hand side of an assignment or an actual parameter passed to a fuzzy or lazy formal parameter. During execution, this delayed evaluation is carried out at the first actual reference, even when the result is to be used in an expression that is also subject to lazy evaluation. For the evaluation the actual values at the time of the evaluation to be used (not the actual values at the time of the assignment or parameter passing). This implies that components of the expression may be uninitialized at the time, when execution reaches the assignment or parameter passing, but may be initialized by the time of the evaluation that can lead to successful evaluation. If, by the time of the evaluation, execution has left the scope unit, in which one or more components of the expression is defined, the actual values of the component(s) at the time of leaving the scope unit are to be stored for the purpose of the delayed evaluation (but only for that, i.e. the values are not accessible for the user).

**lazy value or template:** A value or template instance is called lazy, when the expression, initializing or partly initializing it (including actual parameters passed to in formal parameters), is subject to lazy evaluation. When, during execution, the delayed (lazy) evaluation is taking place, its result is stored in the lazy value or template and the lazy instance is used further on like ordinary values and templates, until the next use of the lazy variable or parameter on the left hand side of an assignment. When a new content is assigned to a lazy instance or to its subpart, the right hand side of the assignment is subject to lazy evaluation again. If, during execution, no expression referencing the lazy object is evaluated, the lazy value or template instance is never evaluated.

**left hand side (of assignment):** value or template variable identifier or a field name of a structured type value or template variable (including array index if any), which stands left to an assignment symbol (:=)

NOTE: A constant, module parameter, timer, structured type field name or a template header (including template type, name and formal parameter list) standing left of an assignment symbol (:=) in declarations and or a modified template definitions are out of the scope of this definition as not being part of an assignment.

**local visibility:** attribute of an entity (constant, variable, etc.) that its identifier can be referenced only within the function, test case or altstep where it is defined

**main test component (MTC):** See Recommendation ITU‑T X.292 [3].

**out parameterization:** kind of parameterization where the actual parameter's content (the argument) is not passed to the formal parameter when the parameterized object is invoked, but the value of the formal parameter is passed back to the actual parameter when the invoked object completes

NOTE 1: In **out** parameterization, parameters are passed by value.

NOTE 2: Out parameters can be used for functions, altsteps, and test cases only, if not restricted by further rules, e.g. **altstep**s activated as defaults.

NOTE 3: An **out** formal parameter is uninitialized (unbound) when the invoked object is entered.

NOTE 4: 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.

**parallel test component (PTC):** See Recommendation ITU‑T X.292 [3].

**partially initialized:** values are partially initialized if a concrete value has been assigned to it or to at least one of its fields or elements

NOTE 1: A template variable is initialized if a matching mechanism has been assigned to it or to at least one of its fields or elements, directly or indirectly via expansion (see clause 15.6). A template is initialized if a matching mechanism has been assigned to it, directly or indirectly via expansion (see clause 15.6).

NOTE 2: TTCN-3 data objects (both value and template) are initialized if they or at least one of their fields or elements have been used on the left hand side of an assignment (including initial value assignment at declaration), except of uninitialized **record**s and **set**s, when the assignment does not change any of its fields. Note that for example an empty pair of curly brackets used for a record value may change the optional fields of the value via the implicit omit mechanism (see clause 27.7).

**passing by reference:** ability to link an actual parameter with a formal parameter of a function, altstep or test case and to control its actual value within the function, altstep or test case by using the formal parameter reference, i.e. no copy of the data content is made and the actual and formal parameters share the same data content

**passing by value:** ability to make a copy of a data content of an actual or formal parameter before passing it to a formal or actual parameter, i.e the actual and formal parameters do not share the same data content

**port parameterization:** ability to pass a port as an actual parameter into a parameterized object via a port parameter

NOTE: This actual port parameter is added to the specification of that object and may complete it.

**qualified name:** TTCN-3 elements can be identified unambiguously by qualified names

NOTE: For modules, the qualified name is the <module name>. For global definitions such as testcases, functions, etc., the qualified name is <module name>.<definition name>. For control, the qualified name is <module name>.control. For local definitions, such as variables, local templates, etc. within a global definition, the qualified name is <module name>.<global definition name>.<local definition name>.

**right hand side (of assignment):** expression, template reference or signature parameter identifier which stands right to an assignment symbol (:=)

NOTE: Expressions and template references standing right of an assignment symbol (:=) in constant, module parameter, timer, template or modified template declarations are out of the scope of this definition as not being part of an assignment.

**root type:** root types of types derived from TTCN-3 basic types are the respective basic types

NOTE 1: The root type of user defined record types is **record**, the root type of user defined record of and array types is **record of**, the root type of user defined set types is **set**, the root type of user defined set of types is **set of**. The root type of user defined union types is **union** and the root type of anytypes is **anytype**. The root types of special configuration types are **default** or **component,** respectively. Port types do not have a root type.

NOTE 2: As **address** is more a predefined type name than a distinct type with its own properties, the root type of an **address** type and all of its derivatives are the same, as the root type was, if the type was defined with a name different from **address**.

**static parameterization:** form of parameterization, in which actual parameters are independent of runtime events; i.e. known at compile time or in case of module parameters are known by the start of the test suite execution

NOTE 1: A static parameter is to be known from the test suite specification, (including imported definitions), or the test system is aware of its value before execution time.

NOTE 2: All types are known at compile time, i.e. are statically bound.

**strong typing:** strict enforcement of type compatibility by type name equivalence with no exceptions

**system under test (SUT):** See Recommendation ITU‑T X.290 [5].

**template:** TTCN-3 data objects are values or templates by definition. A TTCN‑3 template identifies a subset of the values of its type (where the subset may contain a single instance of the type, several instances or all instances) or the matching mechanism **omit**. Templates are defined by global and local templates, template variable definitions, or formal template parameters. Any of those are templates from the point of view of their usage, irrespective of their actual content; for example, a template variable containing a specific value is a template.

**template parameterization:** ability to pass a template as an actual parameter into a parameterized object via a template parameter

NOTE 1: This actual template parameter is added to the specification of that object and may complete it.

NOTE 2: Values passed to formal template parameters are considered to be in-line templates (see clause 15.4).

**test behaviour:** (or behaviour) test case or a function started on a test component when executing an **execute** or a **start** component statement and all functions and altsteps called recursively

NOTE: During a test case execution each test component has its own behaviour and hence several test behaviours may run concurrently in the test system (i.e. a test case can be seen as a collection of test behaviours).

**test case:** See Recommendation ITU‑T X.290 [5].

**test case error:** See Recommendation ITU‑T X.290 [5].

**test suite:** set of TTCN‑3 modules that contains a completely defined set of test cases, optionally supplemented with one or more TTCN‑3 control parts

**test system:** See Recommendation ITU‑T X.290 [5].

**test system interface:** test component that provides a mapping of the ports available in the (abstract) TTCN‑3 test system to those offered by the SUT

**timer parameterization:** ability to pass a timer as an actual parameter into a parameterized object via a timer parameter

NOTE: This actual timer parameter is added to the specification of that object and may complete it.

**type compatibility:** language feature that allows to use values, expressions or templates of a given type as actual values of another type

EXAMPLE: At assignments, as actual parameters at calling a function, referencing a template, etc. or as a return value of a function.

**type context:** "In the context of a type" means that at least one object involved in the given TTCN-3 action (an assignment, operation, parameter passing, etc.) identifies a concrete type unambiguously

NOTE: Either directly (e.g. an in-line template) or by means of a typed TTCN-3 object (e.g. via a constant, variable, formal parameter, etc.).

**unqualified name:** unqualified name of a TTCN-3 element is its name without any qualification

**user-defined type:** type that is defined by subtyping of a basic type or declaring a structured type

NOTE: User-defined types are referenced by their identifiers (names).

**value:** TTCN-3 data objects are values or templates by definition. A TTCN‑3 value is an instance of its type

NOTE: Values are defined by module parameters, constants, value variables, or formal value parameters. Any of those are value objects from the point of view of their usage. A template containing only specific value matching - though referring to a single instance of its type - is not a value object, but is a template object.

**value list notation:** notation that can be used for record, set, record of and set of values, where the values of the subsequent fields or elements are listed within a pair of curly brackets ("{" and "}"), without an explicit identification of the field name or element position

**value notation:** notation by which an identifier is associated with a given value or range of a particular type

NOTE: Values may be constants or variables.

**value parameterization:** ability to pass a value as an actual parameter into a parameterized object via a value parameter

NOTE: This actual value parameter is added to the specification of that object and may complete it.

## 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** ] *Type* *VarIdentifier* [ *ArrayDef* ] [ ":=" *Expression* ]

{ [ "," ] *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 lazy or fuzzy value variables are used in deterministic contexts (i.e. during the evaluation of a snapshot or initialization of global non-fuzzy templates), the same restrictions apply to all functions used in the value assigned to the variable as for functions described in clause 16.1.4.

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 6.2.3 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 (therefore, in particular, it shall not evaluate to **omit**).

***Examples***

**var** **integer** MyVar0;

**var** **integer** MyVar1 := 1;

**var** **boolean** MyVar2 := **true**, MyVar3 := **false**;

**var** @**lazy** **integer** MyLazyVar1 := MyVar1+1;

MyVar1 := 2;

MyVar2 := MyLazyVar1; // MyLazyVar1 evaluates to 2 + 1

MyLazyVar1 := MyLazyVar1 + 1;

MyVar2 := MyLazyVar1; // causes an error as MyLazyVar1 references itself

## 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** [ **@lazy** | **@fuzzy** ] [ *restriction* ] *Type* *VarIdentifier* [ *ArrayDef* ] ":=" *TemplateBody*

{ [ "," ] *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 lazy or fuzzy template variables are used in deterministic contexts (i.e. during the evaluation of a snapshot or initialization of global non-fuzzy templates), the same restrictions apply to all functions used in the template body assigned to the variable as for functions described in clause 16.1.4.

k) Using the dot notation (see clauses 6.2.1.1, 6.2.2.1 and 6.2.5.1) and index notation (see clauses 6.2.3 and 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 The template body at the right-hand side of the assignment symbol shall evaluate to an object that is at least partially initialized.

***Examples***

**var** **template** **integer** MyVarTemp1 := ?;

**var** **template** MyRecord MyVarTemp2 := { field1 := **true**, field2 := \* },  
MyVarTemp3 := { field1 := **?**, field2 := MyVarTemp1 };

**var** **template** @**fuzzy** **float** FuzzTemp1 := rnd(); // evaluated on every usage

**var** **template** @**fuzzy** MyRecord FuzzTemp2 := { rnd() < 0.5, float2int(rnd()) }

**var** **template** @**lazy** **float** LazyTemp1 := FuzzTemp1; // evaluates FuzzTemp1

**var** **template** @lazy MyRecord LazyTemp2 :=

{ LazyTemp1 < 0.5, float2int(FuzzTemp1) } // evaluates LazyTemp1 and FuzzTemp1

LazyTemp2.field1 := true; // evaluates LazyTemp2 and overwrites field1 with true

# 15 Declaring templates

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 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:

a) Templates shall not be of **default** or port type.

b) Templates shall not be of a structured type that contains fields of **default** or port type on any level of nesting.

NOTE 2: The **anytype** type does not include the **default** type nor port types (see clause 6.2.6), so that restriction b) does not apply to anytype templates.

c) The expression or template body initializing a template shall evaluate to a value or template, which is type compatible with the template being declared.

d) 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.

e) 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.

f) 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.

***Examples***

**type** **record** MyRecord {

**default** def

}

**type** **union** MyUnion {

**integer** choice1,

MyRecord choice2

}

**template** MyUnion t\_integerChosen := { choice1 := 5 }

// shall cause an error as the type MyUnion contains MyRecord, which includes

// a field of default type.

**external** **function** garble(**charstring** str) **return** str;

**template @fuzzy charstring t\_fuzzy := garble("foobar"); // every usage of t\_fuzzy re-evaluates**

**// the function call**

## 19.1 Assignments

Values or templates may be assigned to variables or template variables (see clause 11). This is indicated by the symbol ":=".

***Syntactical Structure***

*VariableRef* ":=" ( *Expression* | *TemplateBody* )

***Semantic Description***

During execution of an assignment, the right-hand side of the assignment shall evaluate to a value or template that is at least partially initialized. The effect of an assignment is to bind the variable to the value of the expression or to a template. Assignments are processed from left to right, i.e. expressions in the left hand side are evaluated before those in the right hand side. The evaluations obey the operator precedence defined in table 6. Unless the assignment is to a lazy or fuzzy variable or parameter, the right hand side is evaluated completely before the resulting value or template is bound to the evaluated left-hand side of the assignment. Whenever assignments are used within the right hand side of an assignment (due to assignment notation), these rules apply recursively.

A structured value on the right-hand side of the assignment shall be assigned completely to the variable on the left-hand side of the assignment, If a partially initialized value is assigned to a completely initialized variable, fields uninitialized at the right-hand side of the assignment shall also become uninitialized at the left-hand side.

When a direct or indirect element or field of a lazy or fuzzy variable is assigned, the variable is also evaluated as much as necessary before assignment, i.e. if an ancestor of that element or field is initialized with a function call, it shall be evaluated. Thus, if the variable is fully assigned, it does not need to be evaluated before assignment.

NOTE: If a sub-field or sub-element of a fuzzy variable is assigned that has an ancestor which was formerly assigned a function call, this function call will be evaluated once before the assignment and replaced by its result inside the variable. Thus, the other sub-fields and sub-elements of that ancestor, apart from the field or element being assigned become non-fuzzy.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 15, the following restrictions apply:

1. The right‑hand side of an assignment shall evaluate to a value or template, which is type compatible with the variable at the left-hand side of the assignment.
2. When the right‑hand side of the assignment evaluates to a template (global or local template, in-line template, template variable or a matching mechanism), the variable at the left hand side shall be a template variable.
3. The right‑hand side of an assignment shall evaluate to an object that is at least partially initialized.
4. If the left-hand side of the assignment is a reference to a non-optional value object (i.e. a value definition, a mandatory field, a record/set of/array element, a union alternative, a value parameter), the right-hand side shall not be a reference to an omitted field or the omit symbol.
5. Using a reference to an omitted field in the right-hand side of the assignment has the same effect as using the **omit** keyword.

***Examples***

EXAMPLE 1:

MyVariable := (x + y - increment(z))\*3;

EXAMPLE 2:

**type** **record** MyRecord {

**record** { **float** x, **float** y } c,

**integer** a

}

**var** @lazy MyRecord r := {

c := computeC(),

a := computeA()

} // not evaluated here

r.c.x := computeX(); // first replaces field c with result of computeC(),

// then replaces field c.x with unevaluated computeX()

// field while c.y remains fixed; field a remains unevaluated

EXAMPLE 3:

**type** **record** MyRecord {

**charstring** field1,

**charstring** field2,

**charstring** field3

}

**var** MyRecord v\_MyList1;

**var** MyRecord v\_MyList2;

v\_MyList1 := {"value1", "value2", "value3" }; // v\_MyList1 is completely initialized

v\_MyList2.field2 := "newvalue"; // v\_MyList2 is partilly initialized

// field1 and field3 remain uninitialized

v\_MyList1 := v\_MyList2; // v\_MyList1 become partially initialized,

// field2 has the value "newvalue"

// field1 and field3 are uninitialized