#### 6.2.1.1 Referencing fields of a record type

Elements of a **record** shall be referenced by the dot notation *TypeIdOrExpression.ElementId*, where *TypeIdOrExpression* resolves to the name of a structured type or an expression of a structured type such as variable, formal parameter, module parameter, constant, template, or function invocation. *ElementId* shall resolve to the name of a field in the structured type. Fields of record type definitions shall not reference themselves.

EXAMPLE 1:

MyVar1 := MyRecord1.myElement1;

// If a record is nested within another type then the reference may look like this

MyVar2 := MyRecord1.myElement1.myElement2;

EXAMPLE 2:

**type** **record** MyType

{

**integer** field1,

MyType.field2 field2 **optional**, // this circular reference is NOT ALLOWED

**boolean** field3

}

If a field in a **record** type or a subtype of a **record** type is referenced by the dot notation, the resulting type is the set of values allowed for that field imposed by the constraints of the field declaration itself (i.e. any constraints applied to the **record** type itself are ignored).

EXAMPLE 3:

**type** **record** MyType2

{

**integer** field1 (1 .. 10),

**charstring** field2 **optional**

}

**type** MyType2 MyType3 ({1, omit}, {2, "foo"}, {3, "bar"}) ;

**type** MyType3.field1 MyType4; // MyType4 is the integer type constrained to  
 // the values 1..10

**type** MyType3.field2 MyType5; // MyType5 is the charstring type

**type** MyType2.field1 MyType6; // MyType6 is the integer type constrained to  
 // the values 1..10

**type** MyType2.field2 MyType7; // MyType7 is the charstring type

Referencing a subfield of an uninitialized or omitted record field or value on the right hand side of an assignment shall cause an error.

EXAMPLE 4:

**type record** MyType4

{

**integer** field1 **optional**,

**record**

{

**integer** subfield1,

**integer** subfield2

} field2 **optional**

}

...

**var** MyType4 v\_rec := { field1 := 1, field2 := omit }

**var** integer v\_int := v\_rec.field2.subfield1;

// causes an error as v\_rec.field2 is omitted

When referencing a subfield of an uninitialized or omitted record field or value on the left hand side of an assignment, the record field or value is implicitly set to be present, it is expanded recursively up to and including the depth of the referenced subfield. During this expansion all unreferenced mandatory subfields shall be left undefined. When the assignment is used in a scope where the optional attribute is implicitly or explicitly set to "explicit omit", all unreferenced optional subfields shall be left undefined. When the assignment is used in a scope where the optional attribute is implicitly or explicitly set to "implicit omit", all unreferenced optional subfields shall be set to omit. After this expansion the value at the right hand side of the assignment shall be assigned to the referenced subfield.

EXAMPLE 5:

**var** MyType4 v\_rec;

v\_rec.field2.subfield1 := 5;

// after the assignment v\_rec is { field1 := <undefined>, field2 := { subfield1 := 5,

// subfield2 := <undefined> } }

### 6.2.3 Records and sets of single types

TTCN‑3 supports the specification of records and sets whose elements are all of the same type. These are denoted using the keyword **of**. These records and sets do not have element identifiers and can be considered similar to an ordered array and an unordered collection respectively.

NOTE 1: Subtyping of record of and set of types see in clause 6.2.13.

EXAMPLE 1:

**type** **set of** **boolean** MySetOfType; // is an unlimited set of boolean values

When the assignment notation is used for **record of**‑s, **set of**‑s and arrays, elements wished to be changed are identified explicitly and either a value or the not used symbol "-" can be assigned to them. Other fields, not referred to in the notation, shall remain unchanged. In particular, when specifying partial values (i.e. setting the value of only a subset of the fields) using the assignment notation, for example, at initialization, only the elements to be assigned values shall be specified: elements not mentioned are implicitly left uninitialized. It is also possible to leave fields explicitly unspecified using the not used symbol "-". When re-assigning a previously initialized value, using the not used symbol or just skipping a field or element in an assignment notation, will cause that field or element to remain unchanged.

EXAMPLE 2:

**var** MyRecordOfType MyVariable := {

[0] := '111'B,

[1] := '101'B,

[2] := -

}

MyVariable := { '10111'B, -, - };

// after this, MyVariable contains:

// { '10111'B, '101'B /\* unchanged \*/, <undefined> /\* unchanged \*/ }

MyVariable :=

{

[1] := '010'B,

}

// after this, MyVariable contains:

// { '10111'B/\* unchanged \*/, '010'B, <undefined>/\* unchanged \*/ }

MyVariable :=

{

[0] := -,

[1] := '001'B**,**

[2] := -

}

// after this, MyVariable contains:

// { '10111'B/\* unchanged \*/, '001'B, <undefined> /\* unchanged \*/}

When using the value list notation, all elements in the structure shall be specified either with a value or the not used symbol "-". The first member of the list is assigned to the first element, the second list member is assigned to the second element, etc. No empty assignment is allowed (e.g. two commas, the second immediately following the first or only with white space between them). Elements to be left out of the assignment shall be explicitly skipped in the list by use of the not-used-symbol "-". Already initialized elements left without a corresponding list member in a value list notation (i.e. at the end of a list) are becoming uninitialized. In this way, a value with initialized elements can be made empty by using the empty value list notation ("{}").

Indexed value notations can be used on both the right-hand side and left-hand side of assignments. For multi-dimensional arrays or nested record of or set of types, an array or record of integer can be used as a short-hand notation for a nested index notation. The index notation, when used on the right hand side, refers to the value of the identified element of a **record of** or a **set of** or array. When it is used at the left hand side, only the value of the identified single element is changed, values assigned to other elements already remain unchanged. The index of the first element shall be zero and the index value shall not exceed the limitation placed by length subtyping.

If the value of the element indicated by the index at the right-hand of an assignment is undefined (uninitialized), this shall cause a semantic or runtime error. Referencing an identified element of an uninitialized or omitted record of, set of or array field or value on the right hand side of an assignment shall cause an error.

If an indexing operator at the left-hand side of an assignment refers to a non-existent element, the value at the right-hand side is assigned to the element and all elements with an index smaller than the actual index and without assigned value are created with an undefined value. When referencing an identified element of an uninitialized or omitted record of, set of or array field or value on the left hand side of an assignment, the record of, set of or array field or value is set to be present and it is expanded recursively up to and including the depth of the referenced element. During this expansion all unreferenced elements shall be left undefined. After this expansion the value at the right hand side of the assignment shall be assigned to the referenced element. Undefined elements are permitted only in transient states (while the value remains invisible). Sending a **record of** or **set of** value with undefined elements shall cause a test case error.

NOTE 2: When using on the right hand side of an assignment for **record of-**s, **set of-**s or arrays, the assignment notation and the indexed notation have similar effect, with the exception that the assignment notation is able to address multiple elements in one notation, while the index notation is able to address a single element only.

EXAMPLE 3:

// Given

**type** **record of integer** MyRecordOf;

**type** **record** **of** MyRecordOf RoRoI;

**var integer** MyVar;

// Using the value list notation

**var** MyRecordOf MyRecordOfVar := { 0, 1, 2, 3, 4 };

// The same record of, defined with the assignment notation  
 **var** MyRecordOf MyRecordOfVarAssignment := {  
 [0] := 0,  
 [1] := 1,  
 [2] := 2,

[3] := 3,  
 [4] := 4

};

**var** RoRoI v\_recof;

// Using an indexed notation

MyVar := MyRecordOfVar[0]; // the first element of the "record of" value (integer 0)

// is assigned to MyVar

// Indexed values are permitted on the left-hand side of assignments as well:

MyRecordOfVar[1] := MyVar; // MyVar is assigned to the second element

// value of MyRecordOfVar is { 0, 0, 2, 3, 4 }

// The assignment

MyRecordOfVar := { 0, 1, -, 2 };

// will change the value of MyRecordOfVar to{ 0, 1, 2 <unchanged>, 2};  
 // Note, that the 3rd element would be undefined if had no previous assigned value.

// The assignment

MyRecordOfVar[6] := 6;

// will change the value of MyRecordOfVar to

// { 0, 1, 2 , 2, <uninitialized>, <uninitialized>, 6 };  
 // Note the 5th and 6th elements (with indexes 4 and 5) had no assigned value before this

// last assignment and are therefore undefined.

MyRecordOfVar[4] := 4; MyRecordOfVar[5] := 5;

// will complete MyRecordOfVar to the fully defined value { 0, 1, 2 , 2, 4 , 5 , 6 };

// Expansion of uninitialized record of value:

v\_recof[1][2] := 0;

// after the assignment v\_recof is { <undefined>, { <undefined>, <undefined>, 0 } }

// Pls. Note the difference between the to index assignment notations the following

// example:

**var** MyRecordOf ix := { 0,1,2 }

ix := { [3] := 2\*ix[2]+1 }

// the value of ix is: { 0, 1, 2, 5 }

// The same result can be achieved by using an index notation on the left hand side of

// the assignment:

**var** MyRecordOf ix := { 0,1,2 }

ix[3] := 2\*ix[2]+1

// the value of ix is: { 0, 1, 2, 5 }

NOTE 3: The index notation makes it possible e.g. to copy **record of** values element by element in a for loop. For example, the function below reverses the elements of a **record of** value:

**function** reverse(**in** MyRecordOf src) **return** MyRecordOf  
 {  
 **var** MyRecordOf dest;  
 **var integer** i, srcLength := **lengthof** (src);  
 **for**(i := 0; i < srcLength; i := i + 1) {  
 dest[srcLength - 1 - i] := src[i];  
 }  
 **return** dest;  
 }

Embedded **record of** and **set of** types will result in a data structure similar to multidimensional arrays (see clause 6.2.7).

EXAMPLE 4:

// Given **type** **record of** **integer** MyBasicRecordOfType;  
 **type** **record of** MyBasicRecordOfTypeMy2DRecordOfType;

// Then, the variable myRecordOfArray will have similar attributes to a two-dimensional array:  
 **var** My2DRecordOfType myRecordOfArray;  
 // and reference to a particular element would look like this  
 // (value of the second element of the third 'MyBasicRecordOfType' construct)   
 myRecordOfArray [2][1] := 1;

//with

**var** **integer** i[2] := { 1, 2 };  
 myRecordOfArray [i] := 2;

// is the same as assigning element myRecordOfArray[i[0]][i[1]]

#### 6.2.5.1 Referencing fields of a union type

Alternatives of a **union** type shall be referenced by the dot notation *TypeIdOrExpression.AlternativeId*, where *TypeIdOrExpression* resolves to the name of a union type or an expression of a union type such as variable, formal parameter, module parameter, constant, template, or function invocation. *AlternativeId* shall resolve to the name of an alternative in the union type. Alternatives of union type definitions shall not reference themselves.

EXAMPLE 1:

MyVar5 := MyUnion1.myChoice1;

// If a union type is nested in another type then the reference may look like this

MyVar6 := MyRecord1.myElement1.myChoice2;

// Note, that the union type, of which the field with the identifier 'myChoice2' is referenced,

// is embedded in a record type

If an alternative in a union type or a subtype of a union type is referenced by the dot notation, the resulting type is the set of values allowed for that alternative imposed by the constraints of the alternative declaration itself (i.e. any constraints applied to the union type itself are ignored).

When an alternative of a union type is referenced on the right hand side of an assignment an error shall occur if the referenced alternative is not the currently chosen alternative or if the referenced union field or value is omitted or uninitialized.

EXAMPLE 2:

**type** **union** MyUnion2

{

**integer** choice1,

**charstring** choice2

}

**type** **record** MyRecordWithUnion

{

MyUnion2 field1 **optional**

}

...

**var** MyUnion2 v\_un := { choice1 := 1 }

**var** integer v\_int1 := v\_un.choice2; // causes an error as v\_un.choice2 is not chosen

**var** MyRecordWithUnion v\_rec := { field1 := **omit** }

**var** integer v\_int2 := v\_rec.field1.choice1; // causes an error as v\_rec.field1 is omitted

When referencing an alternative of a union type on the left hand side of an assignment, the referenced alternative becomes the chosen one. This rule is applied recursively if the reference contains alternatives of nested unions, choosing all the referenced alternatives. If the referenced alternative belongs to an uninitialized or omitted union field or value, the field or value is implicitly set to be present and the referenced alternative is chosen. After that the value at the right hand side of the assignment shall be assigned to the referenced alternative.

EXAMPLE 3:

**type** **union** MyUnion3

{

**integer** choice1,

**union**

{

**bitstring** subchoice1,

**charstring** subchoice2

} choice2

}

...

**var** MyUnion3 v\_un2 := { choice1 := 1 }

**var** MyRecordWithUnion v\_rec2 := { field1 := omit }

v\_un2.choice2.subchoice2 := "Hello!";

// after the assignment v\_un2 is { choice2 := { subchoice2 := "Hello!" } }

v\_rec2.field1.choice1 := 10; // after the assignment v\_rec2 is { field1 := { choice1 := 10 } }