### 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 and **set of**‑s, 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 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**. 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** or **set of** 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 uninitialized value.

When referencing an element of an uninitialized **record of** or **set of** value or field or omitted field (including omitting a field at a higher level of the embedding hierarchy) on the left hand side of an assignment, the reference shall recursively be expanded up to and including the depth of the referenced element as follows:

1. When expanding a value or value field of **record of** or **set of** type, the element referenced by the index notation shall be set to present and all elements with a smaller index shall be created with an uninitialized value.
2. Expansion of **record**, **union** and **set** values and intermediate fields shall follow the rules of item a) in clauses 6.2.1.1 and 6.2.5.1, and clause 6.2.2.1 correspondingly.
3. At the end of the expansion, the value at the right hand side of the assignment shall be assigned to the referenced element.

Uninitialized elements are permitted only in transient states (while the value remains invisible). Sending a **record of** or **set of** value with uninitialized elements shall cause an error.

NOTE 2: When using on the right hand side of an assignment for **record of-**s, **set of-**s, 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 two 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]]

#### 6.2.3.1 Nested type definitions

TTCN‑3 supports the definition of the aggregated type nested with the **record** **of** or **set** **of** definition. Both the definition of new structured types (**record**, **set**, **enumerated**, **set** **of** and **record** **of**) and the specification of subtype constraints are possible.

EXAMPLE:

 **type record of enumerated** { red, green, blue } ColorList;

 **type record length** (10) **of record length** (10) **of integer** Matrix;

 **type set of record** { **charstring** id, **charstring** val } GenericParameters;

#### 6.2.3.2 Referencing elements of record of and set of types

It is also allowed to reference the inner type of **record of** and **set of** types by using the index notation but with a dash. The notation *TypeId[-]*, where *TypeId* resolves to the name of a **record of** or **set of** type, references the inner type of *TypeId*. If the type definition restricts the element type of the **record of** or **set of** type, referencing the inner type of that type yields a type which contains all values from the constrained type.

EXAMPLE:

 //Provided the definitions below

 **type** **record** **of** **integer** MyRecordOfInt;

 **type** **record** **of** **record** {

 **integer** f1,

 **set** { **integer** s1, **boolean** s2 } f2

 } MyRecordOfRecord;

 **type** **record** **of** **record** **of** **integer** MyRecordOfRecordOfInt;

 **type** **record** **of** **record** {

 **integer** f1,

 **record** **of** **boolean** f2

 } MyRecordOfRecord2;

 // Referencing the inner integer type

 **type** MyRecordOfInt[-] MyInteger;

 **const** MyRecordOfInt[-] c\_MyInteger:= 5;

 // Referencing the nested record type

 **type** MyRecordOfRecord[-] MyInnerRecord;

 **const** MyRecordOfRecord[-] c\_MyRecord := { f1 = 5; f2 := { s1 := 0; s2 := **true** }}

 // Referencing the set type nested in the inner record

 **type** MyRecordOfRecord[-].f2 MyNestedSet;

 **const** MyRecordOfRecord[-].f2 c\_MySet := { s1 := 0; s2 := **true** }

 // Referencing the innermost boolean

 **type** MyRecordOfRecord[-].f2.s2 MyBoolean;

 **const** MyRecordOfRecord[-].f2.s2 c\_MyBool := **false**;

 // Referencing the inner record of

 **type** MyRecordOfRecordOfInt[-] MyInnerRecordOfInt;

 **const** MyRecordOfRecordOfInt[-] c\_MyInnerRecordOfInt := { 0, 1, 2, 3 };

 // Referencing the integer type within the inner record of

 **type** MyRecordOfRecordOfInt[-][-] MyInteger2;

 **const** MyRecordOfRecordOfInt[-][-] c\_MyInteger2 := 1;

 // Referencing the boolean type within the nested record

 **type** MyRecordOfRecord2[-].f2[-] MyInnermostBoolean;

 **const** MyRecordOfRecord2[-].f2[-] c\_MyInnermostBoolean := **true** ;

 **type** **record length** (5) **of** **record of integer** ConstrainedRecordOfInt (1 .. 10);

 **type** ConstrainedRecordOfInt[-] ConstrainedInt;

 // defines the type record of integer, where the integer values are restricted

 // to the range 1 .. 10 but the record of has no length restriction

### 6.2.7 Arrays

Arrays can be used in TTCN-3 as a shorthand notation to specify record of types. They may be specified also at the point of a variable declaration. Arrays may be declared as single or multi-dimensional. Array dimensions shall be specified using constant expressions, which shall evaluate to a positive **integer** values. Constants used in the constant expressions shall meet with the restrictions in clause 10.

EXAMPLE 1:

 **type integer** MyArrayType1[3]; // A type with 3 integer elements

 **type** **record** **length** (3) **of** **integer** MyRecordOfType1; // The corresponding record of

 **var** MyArrayType1 a1:= { 7, 8, 9 };

 **var** MyRecordOfType1 r1:= a1; // MyArrayType1 and MyRecordOfType1 are compatible

 **var** **integer** myArray1[3]:= r1; // Instantiates an integer array of 3 elements

 // with the index 0 to 2

 // being compatible to MyArrayType1 and MyRecordOfType1

 **var** **integer** myArray2[2][3]; // Instantiates a two-dimensional integer array of 2 × 3 elements // with indexes from (0,0) to (1,2)

Array elements are accessed by means of the index notation ([]), which shall specify a valid index within the array's range. Individual elements of multi-dimensional arrays can be accessed by repeated use of the index notation. An array or record of integer restricted to a single size can be used in the index notation as a short-hand for the repeated index notation. Accessing elements outside the array's range will cause a compile-time or test case error.

EXAMPLE 2:

 MyArray1[1] := 5;
 MyArray2[1][2] := 12;

 MyArray1[4] := 12; // ERROR: index shall be between 0 and 2
 MyArray2[3][2] := 15; // ERROR: first index shall be 0 or 1

Array dimensions may also be specified using ranges (with inclusive boundaries only). In such cases, the lower and upper values of the range define the lower and upper index values. Such an array is corresponding to a record of with a fixed length restriction computed as the difference between upper and lower index bound plus 1 and indexing starting from the lower bound of the array definition.

EXAMPLE 3:

 **type integer** MyArrayType2[2 .. 5]; // A type with 4 integer elements, indices starting with 2

 **type** **record** **length** (4) **of** **integer** MyRecordOfType2; // The corresponding record of

 **var** **integer** MyArray3[1 .. 5]; // Instantiates an integer array of 5 elements

 // with the index 1 to 5

 MyArray3[1] := 10; // Lowest index

 MyArray3[5] := 50; // Highest index

 **var** **integer** MyArray4[1 .. 5][2 .. 3 ]; // Instantiates a two-dimensional integer array of
 // 5 × 2 elements with indexes from (1,2) to (5,3)

NOTE: It is not possible to define an array type with a variable amount of elements. Neither is it possible to define an unlimited array with a lower bound on the array index.

The values of array elements shall be compatible with the corresponding variable or type declaration. Values may be assigned individually by a value list notation or indexed notation or more than one or all at once by a value list notation or index assignment notation. When the value list notation is used, the first value of the list is assigned to the first element of the array (the element with index 0 or the lower bound if an index range has been given), the second value to the next element, etc. Elements to be left out from the assignment shall be explicitly skipped in the list by using dash. For using the assignment notation for arrays, the rules described in 6.2.3 are valid for arrays as well.

Indexed value notation can be used on both the right-hand side and left-hand side of assignments. The index of the first element shall be zero or the lower bound if an index range has been given. The index shall not exceed the limitations given by either the length or the upper bound of the index. If the value of the element indicated by the index at the right‑hand of an assignment is undefined or if the index notation is applied to an uninitialized or omitted array value on the right hand side of an assignment, an error shall be generated. Sending an array value with undefined elements shall cause an error. All elements in an array value that are not set explicitly are undefined. When referencing an element of an uninitialized array value or field or omitted field on the left hand side of an assignment, the rules for record of values specified in 6.2.3 apply.

For assigning values to multi-dimensional arrays, each dimension that is assigned shall resolve to a set of values enclosed in curly braces. When specifying values for multi-dimensional arrays, the leftmost dimension corresponds to the outermost structure of the value, and the rightmost dimension to the innermost structure. The use of array slices of multi-dimensional arrays, i.e. when the number of indexes of the array value is less than the number of dimensions in the corresponding array definition, is allowed. Indexes of array slices shall correspond to the dimensions of the array definition from left to right (i.e. the first index of the slice corresponds to the first dimension of the definition). Slice indexes shall conform to the related array definition dimensions.

EXAMPLE 4:

 MyArray1[0]:= 10;

 MyArray1[1]:= 20;

 MyArray1[3]:= 30;

 // or using an value list

 MyArray1:= {10, 20, **-**, 30};

 MyArray4:= {{1, 2}, {3, 4}, {5, 6}, {7, 8}, {9, 10}};

 // the array value is completely defined

 **var** **integer** MyArray5[2][3][4] :=

 {
 {
 {1, 2, 3, 4}, // assigns a value to MyArray5 slice [0][0]
 {5, 6, 7, 8}, // assigns a value to MyArray5 slice [0][1]
 {9, 10, 11, 12} // assigns a value to MyArray5 slice [0][2]
 }, // end assignments to MyArray5 slice [0]
 {
 {13, 14, 15, 16}, {17, 18, 19, 20}, {21, 22, 23, 24}

 } // assigns a value to MyArray5 slice [1]

 };

 MyArray4[2] := {20, 20};

 // yields {{1, 2}, {3, 4}, {20, 20}, {7, 8}, {9, 10}};
 MyArray5[1] := { {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}};
 // yields {{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}},
 // {{0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}}};

 MyArray5[0][2] := {3, 3, 3, 3};

 // yields {{{1, 2, 3, 4}, {5, 6, 7, 8}, {3, 3, 3, 3}},

 // {{0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}}};

 **var integer** MyArrayInvalid[2][2];

 MyArrayInvalid := { 1, 2, 3, 4 }

 // causes an error as the dimension of the value notation

 // does not correspond to the dimensions of the definition

 MyArrayInvalid[2] := { 1, 2 }

 // causes an error as the index of the slice should be 0 or 1