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.

# 8 Modules

## 8.0 General

The principal building blocks of TTCN‑3 are modules. A module may define a fully executable test suite or just a library. A module may refer to the TTCN-3 language version and to package versions being used. A module consists of a (optional) definitions part, and a (optional) module control part.

NOTE: The term test suite is synonymous with a complete TTCN‑3 module containing test cases and a control part.

The transfer syntax of TTCN-3 modules shall be UTF-8, i.e. each character of the module shall be individually encoded and decoded according to the UCS Transformation Format 8 (UTF-8) as defined in annex R of ISO/IEC 10646 [2] and no characters not corresponding to any character of the module shall be present.

## 8.1 Definition of a module

A module is defined with the keyword **module**.

NOTE 1: The treatment of TTCN‑3 modules in files, repositories and alike is outside the scope of the present document.

***Syntactical Structure***

**module** *ModuleIdentifier* [ **language** *FreeText* { "," *FreeText* } ] "{"

 [ *ModuleDefinitionsPart* ]

 [ *ModuleControlPart* ]

"}"

***Semantic Description***

A TTCN‑3 module groups a set of (typically cohesive) TTCN‑3 definitions. TTCN‑3 modules have an explicit import interface to use definitions from other TTCN‑3 or non-TTCN‑3 modules. It is possible to hide definitions in a TTCN‑3 module (see clause 8.2.5). TTCN‑3 modules can be compiled/interpreted separately. They are reusable and parameterizable.

Module names are of the form of a TTCN‑3 identifier.

NOTE 2: The module identifier is the informal text name of the module.

In addition, a module specification can carry an optional attribute identified by the **language** keyword that identifies the edition of the TTCN‑3 language, in which the module is specified. The following language strings are to be used:

 "TTCN‑3:2001" - to be used with modules complying with V1.1.2 of the present document (see annex H)*.*
 "TTCN‑3:2003" - to be used with modules complying with V2.2.1 of the present document (see annex H).
 "TTCN‑3:2005" - to be used with modules complying with V3.1.1 of the present document (see annex H).
 "TTCN‑3:2007" - to be used with modules complying with V3.2.1 of the present document (see annex H).
 "TTCN‑3:2008" - to be used with modules complying with V3.3.2 of the present document (see annex H).
 "TTCN‑3:2008 Amendment 1" - to be used with modules complying with V3.4.1 of the present document
 (see annex H).
 "TTCN‑3:2009" - to be used with modules complying with V4.1.1 of the present document (see annex H).
 "TTCN‑3:2010" - to be used with modules complying with V4.2.1 of the present document (see annex H).
 "TTCN‑3:2011" - to be used with modules complying with V4.3.1 of the present document (see annex H).
 "TTCN‑3:2012" - to be used with modules complying with V4.4.1 of the present document (see annex H).
 "TTCN‑3:2013" - to be used with modules complying with V4.5.1 of the present document (see annex H).
 "TTCN‑3:2014" - to be used with modules complying with V4.6.1 of the present document (see annex H).
 "TTCN‑3:2015" - to be used with modules complying with V4.7.1 of the present document (see annex H).
 "TTCN‑3:2016" - to be used with modules complying with V4.8.1 of the present document (see annex H).
 "TTCN‑3:2017" - to be used with modules complying with V4.9.1 of the present document (see annex H).
 "TTCN‑3:2018" - to be used with modules complying with the present document.

Furthermore, the optional attribute identified by the **language** keyword may identify package versions being used by this module. The package tags are defined in ETSI ES 202 781 [i.11], ETSI ES 202 782 [i.14], ETSI ES 202 784 [i.12], and ETSI ES 202 785 [i.13]. The language identifier and the package identifier are to be written as a comma-separated list.

***Restrictions***

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

1. At most one language string per module shall be given to define the core language version in which the module is defined.
2. Per extension package, at most one extension package string of that extension package shall be used by a module.

***Examples***

 **module** MyTestSuite **language** "TTCN‑3:2003"

 { … }

## 8.2 Module definitions part

### 8.2.0 General

The module definitions part specifies the top-level definitions of the module and may import visible identifiers from other modules. Visibility rules are given in clause 8.2.5. Scope rules for declarations made in the module definitions part and imported declarations are given in clause 5.3. Those language elements which may be defined in a TTCN‑3 module are listed in table 1. Every definition can be associated with attributes using the with statement defined in clause 27. Visible module definitions may be imported by other modules.

***Syntactical Structure***

 {

 [ *Visibility* ] (

 *TypeDef* |

 *ConstDef* |

 *TemplateDef* |

 *ModuleParDef* |

 *FunctionDef* |

 *SignatureDef* |

 *TestcaseDef* |

 *AltstepDef* |

 *ImportDef* |

 *GroupDef* |

 *ExtFunctionDef* |

 *FriendDef*

 ) [ *WithStatement* ]

 [ ";" ]

 }+

***Semantic Description***

Definitions in the module definitions part may be made in any order.

Such definitions, i.e. top-level definitions outside of other scope units, are globally visible within the module. They may be used elsewhere in the module. This includes identifiers imported from other modules.

Declarations of dynamic language elements such as variables or timers shall only be made in the control part, test cases, functions, altsteps or component types.

TTCN‑3 does not support the declaration of variables in the module definitions part, i.e. global variables cannot be defined in TTCN‑3. However, variables defined in a test component type may be used by all test cases, functions, etc. running on components of that component type and variables defined in the control part provide the ability to keep their values independently of test case execution.

***Restrictions***

No specific restrictions in addition to the general static rules of TTCN‑3 given in clause 5.

***Examples***

 **module** MyModule

 { // This module contains definitions only

 :

 **const** **integer** MyConstant := 1;

 **type record** MyMessageType { … }

 :

 **function** TestStep(){ … }

 **:**

 }

### 8.2.1 Module parameters

Module parameters define a set of values that are supplied by the test environment at runtime. Module parameters do not change their value during test execution. They can be used on right hand side of assignments, in expressions, in actual parameters, and in template definitions, but not within type definitions.

***Syntactical Structure***

Single type, single module parameter form:

[ *Visibility* ] **modulepar** *ModuleParType* *ModuleParIdentifier* [ ":=" *ConstantExpression* ] ";"

Single type, multiple module parameter form:

[ *Visibility* ] **modulepar** *ModuleParType*

{ *ModuleParIdentifier* [ ":=" *ConstantExpression* ] **","** }

 *ModuleParIdentifier* [ ":=" *ConstantExpression* ] ";"

***Semantic Description***

Module parameters behave as global constants at runtime. For module parameterization, TTCN-3 only supports value parameterization which has to be resolved static at start of runtime.

Module parameters allow to customize a TTCN‑3 test suite for a specific IUT, test setup or test campaign. Module parameters are declared by specifying the type and listing their identifiers following the keyword **modulepar**.

It is allowed to specify default values for module parameters. This shall be done by an assignment in the module parameter list. A default value can merely be assigned at the place of the declaration of the module parameter.

If the test system does not provide an actual runtime value for a module parameter, the default value shall be used during test execution, otherwise the actual value provided by the test system. Actual runtime values shall be literals only.

If functions are used for the initialization of module parameters, it is strongly advised to adhere to the rules defined in clause 16.1.4. Not following these rules may cause non-deterministic test executions.

Visible module parameters can be imported.

Optional fields of record and set module parameters or module parameter fields can be initialized explicitly or implicitly. For implicit initialization of the optional fields of a module parameter or a module parameter field, an **optional** attribute with the value "**implicit omit**" (see clause 27.7) shall be associated with it either directly or via the attribute distribution (scoping) mechanism (see clause 27.1.1).

***Restrictions***

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

a) During test execution these values shall be treated as constants.

b) Module parameters shall not be of port, default, timer or component type and shall not be of a structured type that contains a sub-element of port or timer type at any level of nesting.

c) A module parameter shall only be of type address if the address type is explicitly defined within the associated module.

d) Module parameters shall be declared within the module definition part only.

e) More than one occurrence of module parameters declaration is allowed but each parameter shall be declared only once (i.e. redefinition of the module parameter is not allowed).

f) The constant expression for the default value of a module parameter shall respect the limitations given in clause 16.1.4.

g) Module parameters shall not be used in type or array definitions.

h) All sub-elements of **component** or **default** type of a default value of a module parameter shall be initialized with the special value **null**.

***Examples***

 **module** MyTestSuiteWithParameters

 {

 // single type, single module parameter, which is per default public

 **modulepar** **boolean** PX\_Par0 := **true**;

 // single type, multiple module parameters with an explicit public visibility

 **public** **modulepar** **integer** PX\_Par1, PX\_Par2 := 1 + **char2int**("a");

 ...

 }

### 8.2.2 Groups of definitions

In the module definitions part, definitions can be collected in named groups. Grouping is done to aid readability and to add logical structure to the module if required. If necessary, the dot notation shall be used to identify sub-groups within the group hierarchy uniquely, e.g. for the import of a specific sub-group.

***Syntactical Structure***

[ **public** ] **group** *GroupIdentifier* "{"

{ *ModuleDefinition* [ ";" ] }

"}"

***Semantic Description***

A group of definitions can be specified wherever a single definition is allowed. Groups may be nested, i.e. groups may contain other groups. This allows the test suite specifier to structure, among other things, collections of test data or functions describing test behaviour.

Groups and nested groups have no scoping. Please note however, attributes given to a group by an associated with statement apply to all elements of a group (see clause 27). Import statements may import groups so that all visible elements of a group are imported (see clause 8.2.3.3).

***Restrictions***

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

1. Group identifiers across the whole module need not necessarily be unique. However, top-level group identifiers and all group identifiers of subgroups of a single group shall be unique.
2. Only **public** visibility can be defined for groups as they are always public.

***Examples***

 **module** MyModule {

 :

 // A collection of definitions

 **group** myGroup {

 **const** **integer** c\_myConst:= 1;

 :

 **type record** MyMessageType { … };

 **group** myGroup1 { // Sub-group with definitions

 **type** **record** AnotherMessageType { … };

 **const** **boolean** c\_myBoolean := **false**

 }

 }

// A group of altsteps

 **group** myStepLibrary {

 **group** myGroup1 { // Sub-group with the same name as the sub-group with definitions

 **altstep** a\_myStep11() { … }

 **altstep** a\_myStep12() { … }

 :

 **altstep** a\_myStep1n() { … }

 }

 **group** myGroup2 {

 **altstep** a\_myStep21() { … }

 **altstep** a\_myStep22() { … }

 :

 **altstep** a\_myStep2n() { … }

 }

 }

 :

 }

 // An import statement that imports myGroup1 within myStepLibrary

 **import** **from** MyModule {

 **group** myStepLibrary.myGroup1

 }

### 8.2.3 Importing from modules

#### 8.2.3.0 General

It is possible to re-use visible definitions specified in different modules using the **import** statement. Every definition in a TTCN‑3 module has an associated visibility, which is by default **public** (see clause 8.2.5).

NOTE: Groups are **public** only. Importing a group means that only the visible elements of the group are being imported.

#### 8.2.3.1 General format of import

An import statement can be used anywhere in the module definitions part.

***Syntactical Structure***

[ *Visibility* ] **import** **from** *ModuleId*  *[***->** *LocalModuleName]*

 (

 ( **all** [ **except** "{" *ExceptSpec* "}" ] )

 |

 ( "{" *ImportSpec* "}" )

 )

[ ";" ]

***Semantic Description***

TTCN‑3 supports the import of the following definitions: module parameters, user defined types, signatures, constants, data templates, signature templates, functions, external functions, altsteps and test cases. Each definition has a *name* (defines the identifier of the definition, e.g. a function name), a *specification* (e.g. a type specification or a signature of a function) and in the case of functions, altsteps and test cases an associated *behaviour description*. In addition, import statements of one module can be explicitly imported by another module (see clause 8.2.3.7). Only definitions or import statements visible from the importing module can be imported (see clause 8.2.5).

In contrast to module definitions, which are by default public, import statements are by default private.

EXAMPLE 1a:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Name | Specification | Behaviour description |
| **function** | f\_myFunction | (**inout** MyType1 p\_myPar) **return** MyType2**runs on** MyCompType | { **const** MyType3 c\_myConst := …; : // further behaviour} |

|  |  |  |  |
| --- | --- | --- | --- |
|  | Specification | Name | Specification |
| **type** | **record** | MyRecordType | { MyType4 field1, **integer** field2 } |

|  |  |  |  |
| --- | --- | --- | --- |
|  | Specification | Name | Specification |
| **template** | MyType5 | m\_myTemplate | := { field1 := 1, field2 := c\_myConst, // c\_myConst is a module constant field3 := PX\_ModulePar // PX\_ModulePar is module parameter} |

Behaviour descriptions have no effect on the import mechanism, because their internals are considered to be invisible to the importer when the corresponding functions, altsteps or test cases are imported. Thus, they are not considered in the following descriptions.

The specification part of an importable definition contains *local definitions* (e.g. field names of structured type definitions or values of enumerated types) and *referenced definitions* (e.g. references to type definitions, templates, constants or module parameters). For the examples above, this means:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Name | Local definitions | Referenced definitions |
| **function** | f\_myFunction | p\_myPar | MyType1, MyType2, MyCompType |
| **type** | MyRecordType | field1, field2 | MyType4, integer |
| **template** | m\_myTemplate |  | MyType5, field1, field2, field3, c\_myConst, PX\_ModulePar |

NOTE 1: The local definitions column refers to identifiers only that are newly defined in the importable definition. Values assigned to individual fields of importable definitions, e.g. in template definitions, may also be considered as local definitions, but they are not important for the explanation of the import mechanism.

NOTE 2: The referenced definitions field1, field2 and field3 of template MyTemplate are the field names of MyType5, i.e. they are referenced via MyType5.

Referenced definitions are also importable definitions, i.e. the source of a referenced definition can again be structured into a name and a specification part and the specification part also contains local and referenced definitions. In other words, an importable definition may be built up recursively from other importable definitions.

The TTCN‑3 import mechanism is related to the local and referenced definitions used in the specification part of the importable definitions. Table 8 specifies the possible local and referenced definitions of importable definitions.

Table 8: Possible local and referenced definitions of importable definitions

|  |  |  |
| --- | --- | --- |
| Importable Definition | Possible Local Definitions | Possible Referenced Definitions |
| Module parameter |  | Module parameter type |
| User-defined type (for all) |  |  |
| * enumerated type
 | Concrete values |  |
| * structured type
 | Field names, nested type definitions | Field types |
| * port type
 |  | Message types, signatures |
| * component type
 | Constant names, variable names, timer names and port names | Constant types, variable types, port types |
| Signature | Parameter names | Parameter types, return type, types of exceptions |
| Constant |  | Constant type |
| Data Template | Parameter names | Template type, parameter types, constants, module parameters, functions |
| Signature template |  | Signature definition, constants, module parameters functions |
| Function | Parameter names | Parameter types, return type, component type (**runs on** clause) |
| External function | Parameter names | Parameter types, return type |
| Altstep | Parameter names | Parameter types, component type (**runson** clause) |
| Test case | Parameter names | Parameter types, component types (**runs on**- and **system** clause) |
| NOTE 1: For the import of import statements see clause 8.2.3.7.NOTE 2: For the import of groups see clause 8.2.3.3. |

The TTCN‑3 import mechanism distinguishes between the *identifier of a referenced definition* and the *information necessary for the usage of a referenced definition* within the imported definition. For the usage, the identifier of a referenced definition is not required and therefore not imported automatically.

EXAMPLE 1b: Differentiation between *information necessary for the usage* and the identifier

 **module** A {

 **type** **record** MyRec1 {

 **integer** field1,

 **charstring** field2

 }

 }

 **module** B {

 **import** **from** A **all**;

 **type** **record** MyRec2 {

 MyRec1 myField1,

 // "myField1" is the local definition, "MyRec1" is a referenced definition;

 // the *name* "MyRec1" shall be imported in this case as is directly referenced

 **boolean** myField2

 }

 }

 **module** C {

 **import** **from** B **all**;

 **const** MyRec2 c\_myRec2 := {

 myField1 := { field1 := 5, field2 := "A" },

 // to define myField1 of MyRec2 the name "MyRec1" is not needed, the

 // *information necessary for the usage* is its type information,

 // i.e. names and types of its fields field1 and field2

 // which is embeddded in the imported definition of MyRec2

 myField2 := **true**

 }

 }

If an imported definition has attributes (defined by means of a **with** statement) then the attributes shall also be imported. The mechanism to change attributes of imported definitions is explained in clause 27.1.3.

NOTE 3: If the module has global attributes they are associated to definitions without these attributes.

The use of **import** on single definitions, groups of definitions, definitions of the same kind, etc. may lead to situations where the *same definition is referred to more than once*. Such cases shall be resolved by the system and definitions shall be imported only once.

NOTE 4: The mechanisms to resolve such ambiguities, e.g. overwriting and sending warnings to the user, are outside the scope of the present document and should be provided by TTCN‑3 tools.

All **import** statements and definitions within import statements are considered to be treated independently one after the other in the order of their appearance.

All TTCN‑3 modules shall have their own name space in which all definitions shall be uniquely identified. *Name clashes* may occur due to import, e.g. import from different modules. Name clashes shall be resolved using qualified name(s) for the imported definition(s), i.e. prefixing the imported definition (which causes the name clash) by the identifier of the module in which it has been defined; the prefix and the identifier shall be separated by a dot ("."). If the type of the component referenced in a connection operation is known (either when the component reference is a variable or value returned from a function or the type is defined the runs on, mtc or system clause of the calling function), the referenced port declaration shall be present in this component type.

There is one exception to this rule: when **in the context** of an enumerated type (see clause 6.2.4), an enumerated value is clashing with the name of a definition in the importing module, the enumerated value shall take precedence and the definition in the importing module shall be referenced by using its qualified name (see example 5 below in this clause).

In cases where there are no ambiguities the prefixing need not (but may) be present when the imported definitions are used. When the definition is referenced in the same module where it is defined, the module identifier of the module (the current module) also may be used for prefixing the identifier of the definition. For the latter case, prefixing shall only be used for definitions with global visibility for the module.

It is allowed to rename a module name during its import. The new name will be visible only in the importing module.

***Restrictions***

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

a) An import statement shall only be used in the module definitions part and not be used within a control part, function definition, and alike.

b) Only top-level visible definitions of a module may be imported. Definitions which are top-level but invisible to the importing module or which occur at a lower scope (e.g. local constants defined in a function) shall not be imported.

c) A definition is imported together with its name and all local definitions.

NOTE 5: A local definition, e.g. a field name of a user-defined record type or an enumerated value, has only meaning in the context of the definitions in which it is defined, e.g. a field name of a record type can only be used to access a field of the record type and not outside this context.

 In particular, importing an enumerated type does not impose the restriction given in clause 6.2.4 on global names defined in the importing module.

d) A definition is imported together with all information of referenced definitions that are necessary for the usage of the imported definition, independent of the visibility of the referenced definitions (see clause 8.2.5).

NOTE 6: If module C imports a definition from module B that uses a type reference defined in module A, the corresponding information necessary for the usage of that type is automatically imported into module C (see example 6 below in this clause). Identifiers of referenced definitions are not automatically imported.

 In particular, if module C imports global value or template definitions (e.g. constants, module parameters, templates) or local definitions (e.g. formal parameters of templates, functions, etc., or constants and variables of component types) of an enumerated type from module B, the enumerated values of this type (i.e. the identifiers) are implicitly and automatically imported to module C. That is, the enumerated values are known when an enumerated value or template is used in module C (e.g. when an actual parameter is passed or a value is assigned to a component variable). Note that this implicit importing does not impose the restriction given in clause 6.2.4 on global names defined in module C.

e) If the referenced definitions are wished to be used in the importing module, they shall be explicitly imported either directly from its source module or indirectly by importing the import statements of a module importing it (see clause 8.2.3.7).

f) When importing a function, altstep or test case the corresponding behaviour specifications and all definitions used inside the behaviour specifications remain invisible for the importing module.

g) The language specification (see clause 8.1) of the import statement shall not override the language specification of the importing module.

h) The language specification of the import statement shall be identical to the language specification of the source module from which definitions are imported (see clause 8.2.3.8) provided a language specification is defined in the source module. If not, the language specification in the import statement is taken as the language specification of the source module. If the source module uses however language concepts not being part of that language specification, this causes an error for the import statement.

i) If an imported module is renamed in the import clause, the original module name is not imported and cannot be used for referencing the imported module. The imported module can be referenced using the new local name only.

j) If an imported module is renamed in the import clause, the new local name of the module shall be unique in the scope of the importing module.

***Examples***

EXAMPLE 1: Selected import examples

**module** MyModuleA

{ :

 // Scope of the imported definitions is global to MyModuleA

 **import** **from** MyModuleB **all**; // import of all definitions from MyModuleB

 **import from** MyModuleC { // import of selected definitions from MyModuleC

 **type** MyType1, MyType2; // import of types MyType1 and MyType2

 **template** **all** // import of all templates

}

 :

  **function** f\_myBehaviourC()

 {

 // import cannot be used here

 :

 }

 **:**

 **control**

{

 // import cannot be used here

:

}

}

EXAMPLE 2: Use of imported definitions and visibility of definitions referenced by them

 **module** ModuleONE {

 **modulepar** **integer** ModPar1 := …;

 **type** **record** RecordType\_T1 {

 **integer** Field1\_T1,

  **:**

 }

 **type** **record** RecordType\_T2 {

 RecordType\_T1 Field1\_T2,

 :

 }

 **const** **integer** c\_myConst := …;

 **template** RecordType\_T2 m\_t2 (RecordType\_T1 p\_tempParT2):= { // parameterized template

 Field1\_T2 := …,

 :

 }

 } // end module ModuleONE

 **module** ModuleTWO {

 **import** **from** ModuleONE {

 **template** m\_t2

 }

 // Only the names m\_T2 and p\_tempParT2 will be visible in ModuleTWO. Please note, that

 // the identifier p\_tempParT2 can only be used when modifying m\_t2. All information

 // necessary for the usage of m\_t2, e.g. for type checking purposes, are imported

 // for the referenced definitions RecordType\_T1, Field1\_T2, etc., but their identifiers are

 // not visible in ModuleTWO.

 // This means, e.g. it is not possible to use the constant c\_myConst or to declare a

 // variable of type RecordType\_T1 or RecordType\_T2 in ModuleTWO without explicitly importing
 // these types.

 **import** **from** ModuleONE {

 **modulepar** ModPar2

 }

 // The module parameter ModPar2 of ModuleONE is imported from ModuleONE and

 // can be used like an integer constant

 } // end module ModuleTWO

 **module** ModuleTHREE {

 **import** **from** ModuleONE **all**; // imports all definitions from ModuleONE

 **type port** MyPortType **message** {

 **inout** RecordType\_T2 // Reference to a type defined in ModuleONE

 }

 **type component** MyCompType {

 **var integer** v\_myComponentVar := ModPar2;

 // Reference to a module parameter of ModuleONE

 **:**

 }

 **function** f\_myFunction () **return** **integer** {

 **return** c\_myConst // Reference to a module constant of ModuleONE

 }

 **testcase** TC\_MyTestCase (**out** RecordType\_T2 p\_myPar) **runs** **on** MyCompType {

 :

 MyPort.**send**(m\_t2); // Sending a template defined in ModuleONE

 :

 }

 } // end ModuleTHREE

 **module** ModuleFOUR {

 **import from** ModuleTHREE {

 **testcase** TC\_MyTestCase

 }

 // Only the name TC\_MyTestCase will be visible and usable in ModuleFOUR.

 // Type information for RecordType\_T2 is imported via ModuleTHREE from ModuleONE and

 // Type information for MyCompType is imported from ModuleTHREE. All definitions

 // used in the behaviour part of TC\_MyTestCase remain hidden for the user of ModuleFOUR.

 } // end ModuleFOUR

EXAMPLE 3: Handling of name clashes

 **module** MyModuleA {

 :

 **type** **bitstring** MyTypeA;

 **import from** SomeModuleC {

 **type** MyTypeA, // Where MyTypeA is of type character string

 MyTypeB // Where MyTypeB is of type character string

 }

 **:**

 **control** {

 :

 **var** SomeModuleC.MyTypeA v\_myVar1 := "Test String"; // Prefix shall be used

 **var** MyTypeA v\_myVar2 := '10110011'B; // This is the original MyTypeA

 :

 **var** MyTypeB v\_myVar3 := "Test String"; // Prefix need not be used …

 **var** SomeModuleC.MyTypeB v\_myVar3 := "Test String"; // … but it can be if wished

 :

}

 }

NOTE 7: Definitions with the same name defined in different modules are always assumed to be different, even if the actual definitions in the different modules are identical. For example, importing a type that is already defined locally, even with the same name, would lead to two different types being available in the module.

EXAMPLE 4: Renaming imported module

 **module** MyModuleA {

 **import from** VeryLongModuleNameB -> ShortNameB {

 **type** MyTypeA, // Where MyTypeA is of type character string

 }

 **:**

 **control** {

 :

 **var** ShortNameB.MyTypeA v\_myVar1 := "Test String"; // Is correct

 **var** VeryLongModuleNameB MyTypeA v\_myVar2 := "Test String"; // Causes an error

 // as the original module name cannot be used for referencing if the

 // imported module has been renamed.

 }

 }

EXAMPLE 5: Name clash between enumerated values and global definitions

 **module** A {

 **type** **enumerated** MyEnumType {enumX, enumY}

 **type** **enumerated** MyEnumType2 {enumY, enumZ}

 }

 **module** B {

 **import** **from** A **all**;

 **const** MyEnumType enumY := enumX; // this is not allowed as enumerated values restrict

 // global names (see clause 6.2.4)

 **const** MyEnumType2 enumX := enumY;// this is allowed as MyEnumtype2 does not contain enumX

 **const** MyEnumType enumZ := enumX; // allowed as MyEnumType does not contain enumZ

 }

 **module** C {

 **import** **from** A **all**;

 **import** **from** B **all**;

 **const** **integer** enumZ := 0;

 **const** **integer** enumY := 1;

 **const** MyEnumType2 enumX := enumY;

 **modulepar** MyEnumType PX\_MyModulePar1 := enumY

 // the default value of the module parameter will be the value enumY, as the type of

 // PX\_MyModulePar1 creates the context of MyEnumType and in this context enumerated values

 // take precedence over global definition names; note that for the same context reason there

 // is no name clash between the enumerated values defined in MyEnumType and in MyEnumType2

 **modulepar** MyEnumType PX\_MyModulePar2 := B.enumZ

 // the default value of the module parameter will be the value enumX, as the prefix

 // identifies the constant definition enumZ unambiguously, which has the value enumX

 **modulepar** **integer** PX\_IntegerPar := enumZ;

 // the default value of the module parameter will be 0 as this assignment is not in the

 // context of an enumerated type, hence no name clash occurs

 **modulepar** MyEnumType PX\_MyModulePar3 := C.enumX

 // causes an error as PX\_MyModulePar3 and the constant enumX in module C has different types

 }

EXAMPLE 6: Importing local definitions transitively

 **module** A {

 **type enumerated** MyEnumType { enumX, enumY, enumZ}

 **type record** MyRec { **integer** a, **integer** b }

 **type component** MyComp { **var** MyRec v\_rec := { a := 5 } }

 }

 **module** B {

 **import** **from** A **all**;

 **modulepar** MyEnumType PX\_MyModulePar := enumY;

 **type component** MyCompUser **extends** MyComp {}

 }

 **module** C {

 **import from** B **all**;

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

 **if** (PX\_MyModulePar == enumY) {

 // the enumerated value enumY is known in C without explicitly importing it from A

 **setverdict**(**pass**)

 }

 **if** (v\_rec.a == 5) {

 v\_rec.b := v\_rec.a;

 // Both the variable name v\_rec and the record field names are known in C without

 // explicitly importing them from A

 **setverdict** (**pass**)

 }

 }

 }