Methods for Testing and Specification (MTS);
The Test Description Language (TDL);
Part 8: Textual Syntax
## Contents

Intellectual Property Rights ................................................................................................................. 6

Foreword ........................................................................................................................................................................ 6

Modal verbs terminology ........................................................................................................................................... 6

1 Scope ........................................................................................................................................................................ 7

2 References .................................................................................................................................................................... 7
2.1 Normative references ..................................................................................................................................................... 7
2.2 Informative references .................................................................................................................................................. 7

3 Definition of terms, symbols and abbreviations .................................................................................................. 8
3.1 Terms ........................................................................................................................................................................ 8
3.2 Symbols ................................................................................................................................................................... 8
3.3 Abbreviations ............................................................................................................................................................... 8

4 Basic principles ........................................................................................................................................................... 9
4.1 Introduction .................................................................................................................................................................. 9
4.2 Document Structure .................................................................................................................................................... 9
4.3 Grammar Language ................................................................................................................................................... 9
4.3.1 Overview ................................................................................................................................................................. 9
4.3.2 Operators ............................................................................................................................................................... 10
4.3.3 Terminal rules and keywords ................................................................................................................................ 10
4.3.4 Production rules ................................................................................................................................................... 10
4.4 Conformance ............................................................................................................................................................. 11

5 General rules ............................................................................................................................................................. 11
5.1 Identities and references ............................................................................................................................................. 11
5.2 Models and importing .................................................................................................................................................. 11
5.3 Linking ....................................................................................................................................................................... 12
5.4 Alternative syntaxes ................................................................................................................................................... 12
5.5 Terminals .................................................................................................................................................................. 12
5.6 File format ................................................................................................................................................................. 14

6 Production Rules ........................................................................................................................................................ 14
6.1 Foundation ............................................................................................................................................................... 14
6.1.1 Element .................................................................................................................................................................. 14
6.1.2 NamedElement ..................................................................................................................................................... 15
6.1.3 ElementImport .................................................................................................................................................... 15
6.1.4 Package ............................................................................................................................................................... 15
6.1.5 PackageableElement .......................................................................................................................................... 16
6.1.6 Comment ............................................................................................................................................................... 16
6.1.7 Annotation ............................................................................................................................................................ 17
6.1.8 AnnotationType ................................................................................................................................................... 17
6.1.9 TestObjective ....................................................................................................................................................... 18
6.1.10 Extension ............................................................................................................................................................. 18
6.1.11 ConstraintType ................................................................................................................................................ 18
6.1.12 Constraint ........................................................................................................................................................... 19
6.2 Data ......................................................................................................................................................................... 19
6.2.1 DataResourceMapping ........................................................................................................................................ 19
6.2.2 DataElementMapping ........................................................................................................................................ 19
6.2.3 ParameterMapping .............................................................................................................................................. 20
6.2.4DataType .............................................................................................................................................................. 20
6.2.5 SimpleDataType ................................................................................................................................................ 20
6.2.6 SimpleDataInstance ......................................................................................................................................... 21
6.2.7 StructuredDataType ......................................................................................................................................... 21
6.2.8 Member ............................................................................................................................................................... 22
6.2.9 StructuredDataInstance .................................................................................................................................... 22
6.2.10 MemberAssignment ....................................................................................................................................... 22
6.2.11 CollectionDataType ....................................................................................................................................... 23
6.5.9 BoundedLoopBehaviour........................................................................................................ 44
6.5.8 CompoundBehaviour......................................................................................................... 44
6.5.7 SingleCombinedBehaviour.............................................................................................. 44
6.5.6 CombinedBehaviour......................................................................................................... 43
6.4.8 Connection..................................................................................................................... 40
6.4.7 GateReference.................................................................................................................. 40
6.4.5 ComponentInstance......................................................................................................... 39
6.4.4 ComponentType............................................................................................................... 39
6.4.3 GateInstance.................................................................................................................... 39
6.4.2 GateTypeKind.................................................................................................................. 38
6.4.1 GateType......................................................................................................................... 38
6.3.12 TimerStop..................................................................................................................... 37
6.3.11 TimerStart................................................................................................................... 37
6.3.8 Wait................................................................................................................................. 36
6.3.7 TimeOperation............................................................................................................... 36
6.3.6 Timer.............................................................................................................................. 35
6.3.5 TimeConstraint.............................................................................................................. 35
6.3.4 TimeLabelUseKind.......................................................................................................... 35
6.3.3 TimeLabel...................................................................................................................... 34
6.3.2 Time............................................................................................................................... 34
6.3.1 Time ................................................................................................................................ 34
6.2.35 FunctionCall.................................................................................................................. 32
6.2.34 DynamicDataUse.......................................................................................................... 31
6.2.33 LiteralValueUse............................................................................................................ 31
6.2.31 AnyValueOrOmit........................................................................................................... 30
6.2.30 AnyValue...................................................................................................................... 30
6.2.29 SpecialValueUse.......................................................................................................... 30
6.2.28 DataInstanceUse.......................................................................................................... 30
6.2.26 MemberReference......................................................................................................... 28
6.2.25 ParameterBinding.......................................................................................................... 28
6.2.24 DataUse....................................................................................................................... 27
6.2.23 EnumDataType............................................................................................................. 27
6.2.22 PredefinedFunction...................................................................................................... 26
6.2.21 UnassignedMemberTreatment .................................................................................. 26
6.2.20 Function........................................................................................................................ 26
6.2.19 Action.......................................................................................................................... 25
6.2.18 Variable...................................................................................................................... 25
6.2.17 FormalParameter........................................................................................................... 24
6.2.16 Parameter.................................................................................................................... 24
6.2.15 ParameterKind............................................................................................................. 24
6.2.14 ProcedureParameter.................................................................................................... 23
6.2.13 ProcedureSignature...................................................................................................... 23
6.2.12 CollectionDataInstance............................................................................................... 23
6.3 Test Behaviour................................................................................................................... 41
6.3.9 Quiescence................................................................................................................... 36
6.3.8 Wait............................................................................................................................... 36
6.3.7 TimeOperation............................................................................................................... 36
6.3.6 Timer.............................................................................................................................. 35
6.3.5 TimeConstraint.............................................................................................................. 35
6.3.4 TimeLabelUseKind.......................................................................................................... 35
6.3.3 TimeLabel...................................................................................................................... 34
6.3.2 Time............................................................................................................................... 34
6.3.1 Time ................................................................................................................................ 34
6.4 Test Configuration............................................................................................................... 38
6.4.1 GateType ....................................................................................................................... 38
6.4.2 GateTypeKind................................................................................................................ 38
6.4.3 GateInstance.................................................................................................................. 39
6.4.4 ComponentType............................................................................................................ 39
6.4.5 ComponentInstance....................................................................................................... 39
6.4.6 ComponentInstanceRole............................................................................................... 40
6.4.7 GateReference............................................................................................................... 40
6.4.8 Connection................................................................................................................... 40
6.4.9 TestConfiguration........................................................................................................... 41
6.5 Test Behaviour................................................................................................................... 41
6.5.1 TestDescription.............................................................................................................. 41
6.5.2 BehaviourDescription................................................................................................... 42
6.5.3 Behaviour...................................................................................................................... 42
6.5.4 Block.............................................................................................................................. 43
6.5.5 LocalExpression............................................................................................................ 43
6.5.6 CombinedBehaviour.................................................................................................... 43
6.5.7 SingleCombinedBehaviour.......................................................................................... 44
6.5.8 CompoundBehaviour................................................................................................... 44
6.5.9 BoundedLoopBehaviour............................................................................................... 44
Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The declarations pertaining to these essential IPRs, if any, are publicly available for ETSI members and non-members, and can be found in ETSI SR 000 314: "Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards", which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (https://ipr.etsi.org/).

Pursuant to the ETSI Directives including the ETSI IPR Policy, no investigation regarding the essentiality of IPRs, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

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

Foreword

This ETSI Standard (ES) has been produced by ETSI Technical Committee Methods for Testing and Specification (MTS).

The present document is part 8 of a multi-part deliverable. Full details of the entire series can be found in part 1 [1].

Modal verbs terminology

In the present document "shall", "shall not", "should", "should not", "may", "need not", "will", "will not", "can" and "cannot" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"must" and "must not" are NOT allowed in ETSI deliverables except when used in direct citation.
1 Scope

The present document specifies the concrete textual syntax of the Test Description Language (TDL). The intended use of the present document is to serve as the basis for the development of textual TDL tools and TDL specifications. The meta-model of TDL and the meanings of the meta-classes are described in ETSI ES 203 119-1 [1].

NOTE: OMG®, UML® and OCL™ are the trademarks of Object Management Group (OMG). Xtext™ is trademark of the Eclipse® Foundation. This information is given for the convenience of users of the present document and does not constitute an endorsement by ETSI of the products named.

2 References

2.1 Normative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

Referenced documents which are not found to be publicly available in the expected location might be found at https://docbox.etsi.org/Reference/.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are necessary for the application of the present document.


2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] Eclipse® Foundation: Xtext - The Grammar Language Website.

[i.2] ETSI TS 136 523-1 (V10.2.0): "LTE; Evolved Universal Terrestrial Radio Access (E-UTRA) and Evolved Packet Core (EPC); User Equipment (UE) conformance specification; Part 1: Protocol conformance specification (3GPP TS 36.523-1 version 10.2.0 Release 10)".

[i.3] ETSI TS 186 011-2: "Core Network and Interoperability Testing (INT); IMS NNI Interoperability Test Specifications (3GPP Release 10); Part 2: Test descriptions for IMS NNI Interoperability”.

[i.4] ETSI: TDL Open Source Project.

[i.5] ETSI ES 203 119-4: "Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 4: Structured Test Objective Specification (Extension)".

[i.6] ETSI ES 203 119-7: "Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 7: Extended Test Configurations".
3 Definition of terms, symbols and abbreviations

3.1 Terms

For the purposes of the present document, the terms given in ETSI ES 203 119-1 [1] and the following apply:

**derivation**: construction of an abstract syntactical structure, such as a model instance conforming to a meta-model, from a textual representation by applying the structural rules of a grammar, and potential mappings to the underlying meta-model

**(formal) grammar**: set of structural rules that define how to form valid strings from a language's alphabet that obey the syntax of the language

**non-terminal symbol**: placeholder for (groups of) other symbols that describe elements in a specified language

**(production) rule**: definition of a structured rule for the derivation of a non-terminal symbol based on other non-terminal symbols and terminal symbols

**terminal symbol**: symbols that appear explicitly in a specified language, such as a keyword, an identifier or other tokens

3.2 Symbols

Void.

3.3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

- EBNF: Extended Backus-Naur Form
- IMS: IP Multimedia Subsystem
- OCL: Object Constraint Language™
- OMG: Object Management Group®
- SUT: System Under Test
- TDL: Test Description Language
- TTCN-3: Testing and Test Control Notation version 3
- UML: Unified Modelling Language®
- URI: Uniform Record Identifier
4 Basic principles

4.1 Introduction

The meta-model of the Test Description Language (TDL) is specified in ETSI ES 203 119-1 [1]. The presentation format of the meta-model can be different according to the needs of the users or the requirements of the domain, where the TDL is applied. These presentation formats can either be text-oriented or graphic-oriented and may cover all the functionalities of the TDL meta-model or just a part of it, which is relevant to satisfy the needs of a specific application domain.

The present document specifies a concrete textual syntax that provides a textual representation for the commonly used functionality of the TDL meta-model. In the current version of the present document, certain parts, such as 'Comment's and 'Annotation's in 'DataUse' elements, are syntactically excluded. Syntactic specifications for these may be added in future versions of the present document as needed.

The present document specifies the TDL textual file format, where the textual representations of the instances of the TDL meta-classes may be placed. A textual representation may contain keywords, delimiters, and textual labels within a defined structure. The rules, how these structures shall be interpreted, are described by means of Extended Backus-Naur Form (EBNF)-like expressions. In particular, in addition to the syntactical structure, the EBNF-like expressions also indicate how the textual labels and structures are mapped to the TDL meta-model.

4.2 Document Structure

The present document specifies the concrete textual syntax of the Test Description Language (TDL).

Clause 5 specifies general rules for the specification and use of the TDL textual file format.

Clause 6 specifies the concrete production rules defined for the TDL meta-classes (the meta-model of TDL and the meanings of the meta-classes are described in ETSI ES 203 119-1 [1]):

- Foundation (clause 6.1)
- Data (clause 6.2)
- Time (clause 6.3)
- Test Configuration (clause 6.4)
- Test Behaviour (clause 6.5)

At the end of the present document several examples illustrating the features of the TDL Textual Syntax can be found.

4.3 Grammar Language

4.3.1 Overview

The rules that define the textual syntax of the TDL are described in present document using the grammar language of the Xtext framework. In addition to defining the lexical structure of the TDL syntax the grammar language also provides means for mapping those textual constructs to the TDL meta-model. Additional rules such as identity resolution and linking are described where applicable to provide complete mapping of textual TDL to the TDL model.

The grammar of textual TDL is composed of a number of grammar rules organized in a tree. The grammar structure follows the logical structure of the TDL meta-model and the root of the grammar is the 'Package' production rule. Production rules are used to construct model objects and assign values to the properties of those objects. Production rules consist of keywords (character literals) and calls to production rules, data type rules and terminal rules (which correspond to tokens of text).

The following clauses describe the syntax of the grammar language. See Xtext documentation for further details [i.1].
4.3.2 Operators

Various operators are used in grammar rule definitions to specify the order and cardinality of keywords and rule calls. Terminal rule specific operators are used to express various textual constructs. Production rule specific operators are used to define assignments and cross-references.

Following operators are used in all rule definitions:

- '?' indicates that preceding construct shall occur 0 or 1 time;
- '*' indicates that preceding construct shall occur 0 or more times;
- '+' indicates that preceding construct shall occur 1 or more times;
- '|' is used between alternative constructs; and
- '(' and ')' are used to group constructs defined in between.

Following operators are used in terminal rule definitions:

- '!' is used to negate a construct;
- '->' is used to indicate that everything is ignored until the following construct is detected;
- '..' is used between characters to define a range; and
- '.' denotes any character.

Following operators are used in production rule definitions:

- '=' is used to define a simple assignment of a right hand construct to a property on the left;
- '+=' is used for assigning (adding to) multi-valued property;
- '?=' is used for assigning the value 'true' to a Boolean property on the condition that the right hand side construct is present; and
- '[', '|', and ']' are used to define a cross-reference.

Various special symbols are included in the grammar definitions of production rules that are included solely as implementation detail (to help the generation of a parser for textual TDL) and do not alter the definition of the syntax. Such symbols include '->' and '=>'.

4.3.3 Terminal rules and keywords

Lexical tokens in the TDL grammar are either keywords of character sequences that are matched and consumed by terminal rules during parsing. In the grammar definition, keywords are placed between apostrophes (').

Terminal rule declarations start with the keyword 'terminal' followed by the rule name (in upper-case letters by convention). The rule name is followed by 'returns' keyword and the reference to a data type that is used for creating a value using the consumed token.

The definition of the rule starts with a colon (':') and ends with a semi-colon (';'). Terminal rule definitions consist of terminal rule calls (indicated by rule name), characters and operators.

**EXAMPLE: Terminal INT returns EInt: ('0'..'9')+;**

Some terminal rules (such as comments and whitespace) are defined as hidden in TDL grammar and corresponding text shall be allowed anywhere in textual TDL (outside of tokens).

4.3.4 Production rules

Production rules are used to create model objects or data values. The rules that return a data type instead of a meta-class are known as data type rules.
Production rule declarations start with the rule name followed by 'returns' keyword and the reference to the meta-class that defines the object that is produced by the rule. The definition of the rule starts with a colon (':') and ends with a semi-colon (';'). Production rule definitions consist of rule calls, keywords and operators.

**EXAMPLE 1:** Comment returns tdl::Comment:

```plaintext
'Note:' body=EString
```

An assignment is defined as a property name followed by an assignment operator (see clause 4.3.2) followed by a rule call (name of production or data type rule) or a cross-reference. A cross-reference is defined as a meta-class reference followed by '|' and a terminal rule call that defines the format for the identifier. The cross-reference definition is placed between square brackets ('[' and ']').

**EXAMPLE 2:** Annotation returns tdl::Annotation:

```plaintext
'@' key=[tdl::AnnotationType|Identifier]
(':' value=EString)?
```

Production rule calls may also be used without assignment. In that case the model object that is returned from the calling rule is the one that is created in the called rule.

Production rules may be created as fragments by prefixing the declaration with the 'fragment' keyword. In that case the rule does not produce an object by itself but rather assigns to properties of the object that is created in the calling rule. Fragment rules are always unassigned.

### 4.4 Conformance

For an implementation claiming to conform to this version of the TDL Concrete Textual Syntax, all features specified in the present document and in ETSI ES 203 119-1 [1] shall be implemented consistently with the requirements given in the present document and ETSI ES 203 119-1 [1].

### 5 General rules

#### 5.1 Identities and references

In TDL models, references between objects are based on unique identifiers that are generated by the modelling framework and stored in model files. Such identifiers are generally hidden from the user. In textual TDL, all attributes shall be part of the text document and the use of such identifiers is not feasible.

In textual TDL, objects are identified by 'name' or 'qualifiedName' property. The allowed values for the 'name' property are restricted by the terminal rule 'ID' (see clause 5.5). The exception to this rule is made for objects that are predefined in TDL and are mapped to special symbols in textual TDL (such as AnyValue).

If the 'name' property shall have a value that is equal to a keyword in textual TDL then that value shall be prefixed with '^' in the text.

#### 5.2 Models and importing

TDL objects stored in a single file are collectively referred to as model. Both the TDL model and textual TDL allow single 'Package' object as the root of the model. Thus, logically the root package of a TDL file is a TDL model.

Naming of textual TDL files and the location of those files is out of the scope of the present document. Implementations of the textual TDL shall provide means to make TDL models available for importing.

Imported 'Package's shall be referred to by the value of the 'qualifiedName' property.
5.3 Linking

Linking refers to the phase in the compilation process of textual TDL where name-based cross-references are resolved to actual objects that they represent. By default, linking utilizes object identities as described in clause 5.1.

In some cases where explicit cross-references are not required by the grammar rules, the linking may apply context specific logic to assign references to object properties. Such cases are described in the relevant clauses.

5.4 Alternative syntaxes

Although the keywords are specified with certain case (lower-case or title-case) in the present document, the case itself is not prescribed. Therefore, an implementation can be case-insensitive as well. It is recommended that users apply a consistent case nonetheless.

The delimiters for 'Block's and other constructs are specified in an abstract manner with the 'BEGIN' and 'END' terminal symbols. While the default assumption is that these terminal symbols are mapped to left and right braces ('{' and '}'), referred to as 'brace-based' syntax, an alternative implementation using white space indentation is also possible, where synthetic delimiters for the beginning and end of indented parts shall be used instead, referred to as 'indentation-based' syntax. Besides the replacement of the 'BEGIN' and 'END' symbols, no other differences shall be present between implementations of the 'brace-based' and 'indentation-based' syntax. Left and right braces ('{' and '}') shall be used in certain contexts even within the 'indentation-based' syntax, e.g. for 'TimeConstraint's and data-related 'Constraint's.

The examples in the present document conform to the default assumption. Additional examples illustrating the indentation-based syntax are included in Annex B.

5.5 Terminals

The base terminal symbol definitions include the following:

```plaintext
terminal ID: 'a'.. 'z'|'A'..'Z'|'_'( 'a'.. 'z'|'A'..'Z'|'_' | '0'.. '9' )*;
terminal INT returns EInt: ('0'.. '9' )+;
terminal STRING: "" ( '\' . /* 'b'|'t'|'n'|'f'|'r'|'u'|""|""|'\' */  | !( '\'| '"') )* "" |
                  "" ( '\' . /* 'b'|'t'|'n'|'f'|'r'|'u'|""|""|'\' */  | !( '\'| '"') )* "";
terminal ML_COMMENT : '/*' -> '*/';
terminal SL_COMMENT : '//' !( '
'| '' )* (''? '
')?;
terminal WS : ( ' '| '	' | '' | '
')+;
terminal ANY_OTHER: .;
terminal TRUE : 'true';
terminal FALSE : 'false';
terminal BEGIN: '{';
terminal END: '}';
```

The 'WS', 'MLCOMMENT', and 'SLCOMMENT' tokens shall be hidden.

For the indentation-based syntax variant, the 'BEGIN' and 'END' terminal symbols are redefined to the following (with 'synthetic:BEGIN' and 'synthetic:END' representing an increase and a decrease in the indentation, respectively):

```plaintext
@Override
terminal BEGIN: 'synthetic:BEGIN';  // increase indentation
@Override
terminal END: 'synthetic:END';      // decrease indentation
```

In addition to the terminal symbols, data type parser rules for context-sensitive 'pseudo-terminals' include the following:

```plaintext
EString:
  STRING
;```
The 'LBrace' and 'RBrace' rules differentiate the use of left '{' and right '}' braces in certain contexts (e.g. 'Constraint's and 'TimeConstraint's) from their use as delimiters in the brace-based variant of the syntax. For the indentation-based variant of the syntax, these rules shall be overridden as follows:

```java
// Retain Braces even in indentation-based
@Override
LBrace: '{'

@Override
RBrace: '{'

// for both indented and un-indented blocks within parentheses
@Override
LParen: '(' BEGIN?

@Override
RParen: ')' END? '
```

The redefinition of the 'LParen' and 'RParen' with optional 'BEGIN' and 'END' tokens enables the use of indentation in blocks within parentheses in the indentation-based variant as all indentation is semantically relevant. In case indentation needs to be optionally allowed in other cases, a similar pattern can be applied for further tailoring of the indentation-based syntax variant.
5.6 File format

No assumptions are made about the file format at present. For practical purposes, certain conventions regarding the naming of files using the indentation-based and brace-based variants of the syntax are recommended, e.g. using different file endings or "extensions".

6 Production Rules

6.1 Foundation

6.1.1 Element

Concrete Textual Notation

```plaintext
fragment AnnotationFragment returns tdl::Element:
  [annotation+=Annotation] *
```

```plaintext
fragment AnnotationCommentFragment returns tdl::Element:
  [comment+=Comment] *
  [annotation+=Annotation] *
 ;
```

```plaintext
fragment NameFragment returns tdl::Element:
  'Name:' name=Identifier
 ;
```

```plaintext
fragment WithCommentFragment returns tdl::Element:
  'with'
  BEGIN
    [comment+=Comment]+
  END
 ;
```

```plaintext
fragment WithNameFragment returns tdl::Element:
  'with'
  BEGIN
    NameFragment
  END
 ;
```

Comments

This is an abstract metaclass, therefore no textual representation is defined for the element. The concrete textual notation represents reusable fragments that can be embedded in the concrete textual notation of metaclasses inheriting from this metaclass.

The different fragments are used in different contexts.

Examples

```plaintext
Note: "Example test objective"
@Example

with {
  Note: "Comment on nested package"
}

with {
  Name: an Optional Name For Element Without Mandatory Name
}
```
6.1.2 NamedElement

Concrete Textual Notation
Void.

Comments
This is an abstract metaclass, therefore no textual representation is defined for the element.

Examples
Void.

6.1.3 ElementImport

Concrete Textual Notation

```plaintext
ElementImport returns tdl::ElementImport:
   AnnotationCommentFragment
   'Import'|
   [\*] |
   | [importedElement+=\[tdl::PackageableElement | Identifier\]]* |
   | [\*] |
   | 'from' importedPackage=[\[tdl::Package | QIdentifier\]] |
```

Comments
No comments.

Examples
Import all from NestedPackage
Import NestedAnnotation from NestedPackage

6.1.4 Package

Concrete Textual Notation

```plaintext
Package returns tdl::Package:
   AnnotationCommentFragment
   'Package' name=Identifier
   [BEGIN |
   ^import+=ElementImport]* |
   [packagedElement+=PackageableElement]* |
   [nestedPackage+=Package]* |
   END]? |
```

Comments
'Annotation's applied to the 'Package' shall be defined within the 'Package' or imported in the 'Package' from other 'Package's even as the applicable 'Annotation's appear on the "outside" of the 'Package'.
Examples

```java
@NestedAnnotation
package Foundation {
    note: "Example imports from nested / other package"
    import all from NestedPackage
    import NestedAnnotation from NestedPackage

    note: "Annotate examples"
    annotation Example

    note: "Annotate standardized constructs"
    annotation Standard

    package NestedPackage {
        annotation NestedAnnotation
    }

    with {
        note: "Comment on nested package"
    }
}

with {
    note: "Comment on foundation"
}
```

6.1.5 PackageableElement

Concrete Textual Notation

`packageableElement returns std::PackageableElement:
| AnnotationType | TestObjective
| ConstraintType |
| DataSourceMapping | DataElementMapping
| SimpleDataType | SimpleDataInstance
| StructuredDataType | StructuredDataInstance
| CollectionDataType | CollectionDataInstance
| ProcedureSignature
| Action | Function
| PredefinedFunction
| EnumDataType
| Time
| ComponentType | GateType
| TestConfiguration
| TestDescription
|`

Comments

This is an abstract metaclass, the textual representation depends on the concrete types indicated as alternative derivations.

Examples

Void.

6.1.6 Comment

Concrete Textual Notation

`comment returns std::comment:
| "Note" [name=NIdentifier]? |
| "body" = EString`

;;
Comments

In different contexts, depending on the fragments being used, a 'Comment' may be defined before the 'Element' or within a 'with' block.

'Comment's are syntactically excluded from certain constructs, e.g. 'DataUse'.

Examples

```
Note: "Example test objective"
Objective TO_Foundation
```

### 6.1.7 Annotation

Concrete Textual Notation

```
Annotation returns all::Annotation:
    '@' key=[all::AnnotationType [identifier]
       | "value=estring"]?
;
```

Comments

'Annotation's are syntactically excluded from certain constructs, e.g. 'DataUse'.

Examples

```
@Example
Objective TO_Foundation
```

### 6.1.8 AnnotationType

Concrete Textual Notation

```
AnnotationType returns all::AnnotationType:
    AnnotationCommentFragment
    'Annotation' name=identifier
    | 'extends' extension=Extension ?
;
```

Comments

No comments.

Examples

```
Note: "Annotate examples"
Annotation Example

Note: "Annotate standardized constructs"
Annotation Standard
```
6.1.9  TestObjective

Concrete Textual Notation

TestObjective returns tdl::TestObjective:
   AnnotationCommentFragment
   'Objective'  name=Identifier
   BEGIN
      ['Description': description=EString]?
      ['References': objectiveURI+=EString (', objectiveURI+=EString)*]?
   END?

Comments
No comments.

Examples

Objective TO_Foundation {
   Description: "Illustrate the definition of a test objectives"
   References: "This package.", "A base document", "Another source"
}

6.1.10  Extension

Concrete Textual Notation

Extension returns tdl::Extension:
   extending=[tdl::PackageableElement|Identifier]

Comments
'Annotation's and 'Comment's are syntactically excluded.

Examples

Structure Post {
   String title,
   String date
}

Structure TaggedPost extends Post {
   Tags tags
}

6.1.11  ConstraintType

Concrete Textual Notation

ConstraintType returns tdl::ConstraintType:
   AnnotationCommentFragment
   'Constraint'  name=Identifier

Comments
No comments.
Examples

```text
@Example
Constraint HexString
@Example
Constraint DateTime
@Standard
Constraint Length
```

6.1.12 Constraint

Concrete Textual Notation

```text
Constraint returns tdl::Constraint:
  type=[tdl::ConstraintType | Identifier] (':' quantifier+=DataUse (',' quantifier+=DataUse)* )?;
```

Comments

'Annotation's and 'Comment's are syntactically excluded.

Examples

```text
Collection Posts {Length: 10} of Post
```

6.2 Data

6.2.1 DataResourceMapping

Concrete Textual Notation

```text
DataResourceMapping returns tdl::DataResourceMapping:
  AnnotationCommentFragment
  'Use' resourceURI=EString
  'as' name=Identifier;
```

Comments

No comments.

Examples

```text
Note: "Use external resource for the mapping"
Use "API.yaml" as API
```

6.2.2 DataElementMapping

Concrete Textual Notation

```text
DataElementMapping returns tdl::DataElementMapping:
  AnnotationCommentFragment
  'Map' mappableDataElement=[tdl:: | Identifier] ('to' elementURI=EString )?
  'in' dataResourceMapping=[tdl::DataResourceMapping | Identifier] 'as' name=Identifier
  [BEGIN
    parameterMapping+=ParameterMapping (',' parameterMapping+=ParameterMapping)*
  END]*;
```
6.2.3 ParameterMapping

Concrete Textual Notation

ParameterMapping returns idl::ParameterMapping:
   AnnotationCommentFragment
   parameter=[idl::Parameter|Identifier]'->' parameterURI=EString
;

Comments
No comments.

Examples

   title -> 'post::title'
   date -> 'post::date'

6.2.4 DataType

Concrete Textual Notation

fragment ConstraintFragment returns idl::DataType:
   [LBrace constraint+=Constraint RBrace]*
;

Comments
This is an abstract metaclass, therefore no textual representation is defined for the element. The concrete textual notation represents reusable fragments that can be embedded in the concrete textual notation of metaclasses inheriting from this metaclass.

Examples
Void.

6.2.5 SimpleDataType

Concrete Textual Notation

SimpleDataType returns idl::SimpleDataType:
   AnnotationCommentFragment
   'Type' name=Identifier
   ConstraintFragment
   | 'extends' extension=Extension
;

Comments

Comments

No comments.

Examples

```c
@Standard
type String
@Standard
type Integer
@Standard
type Verdict
```

### 6.2.6 SimpleDataInstance

Concrete Textual Notation

```c
SimpleDataInstance returns tdl::SimpleDataInstance:
   AnnotationCommentFragment
dataType=[tdl::DataType|Identifier]
   name=Identifier
;```

Comments

No comments.

Examples

```c
Verdict fail
String authToken
Integer sessionId
```

### 6.2.7 StructuredDataType

Concrete Textual Notation

```c
StructuredDataType returns tdl::StructuredDataType:
   AnnotationCommentFragment
   'Structure' name=Identifier
   ConstraintFragment
   ('extends' extension+=Extension (',' extension+=Extension)*)?
   LParen (member+=Member (',' member+=Member)*)? RParen
;```

Comments

No comments.

Examples

```c
Structure Post
   String title,
   String date
;```
6.2.8 Member

Concrete Textual Notation

Member returns id: Member:
   AnnotationCommentFragment
   [asOptional?="optional"]?
   dataType=[id: DataType | Identifier]
   name=Identifier
   [LBrace constraint+=Constraint RBrace]*
;

Comments
No comments.

Examples

String title
   Note: "Constraint for members"

String date [DateString]
   Note: "Optional members"

6.2.9 StructuredDataInstance

Concrete Textual Notation

StructuredDataInstance returns id: StructuredDataInstance:
   AnnotationCommentFragment
   dataType=[id: DataType | Identifier]
   name=Identifier
   ["<" unassignedMember=UnassignedMemberTreatment ">"]? LParen [memberAssignment+=MemberAssignment (',' memberAssignment+=MemberAssignment)*]? RParen
;

Comments
No comments.

Examples

Post firstReport <?>
   title = "first report"
   date = "today"

6.2.10 MemberAssignment

Concrete Textual Notation

MemberAssignment returns id: MemberAssignment:
   AnnotationCommentFragment
   member=[id: Member | Identifier]
   '=' memberSpec=DataUse
;

Comments
No comments.
Examples

```plaintext
title = 'first report'
date = firstReport.date
tags = ?
```

### 6.2.11 CollectionDataType

Concrete Textual Notation

```plaintext
CollectionDataType returns tdl::CollectionDataType:
   AnnotationCommentFragment
   'Collection' name=Identifier
   ConstraintFragment
   'of' itemType=[tdl::DataType|Identifier]

```

Comments
No comments.

Examples

```plaintext
Collection Posts of Post
Collection Tags of Tag
```

### 6.2.12 CollectionDataInstance

Concrete Textual Notation

```plaintext
CollectionDataInstance returns tdl::CollectionDataInstance:
   AnnotationCommentFragment
dataType=[tdl::DataType|Identifier]
   name=Identifier
   [unassignedMember=UnassignedMemberTreatment]?
   ']' item+=DataUse (',' item+=DataUse)* '

```

Comments
No comments.

Examples

```plaintext
Tags usefulReportsFilter [useful, report]
Posts allPosts [
   new Post(title="first post", date="yesterday"),
   new Post(title="second post", date="today"),
   firstReport,
   secondReport
]
```

### 6.2.13 ProcedureSignature

Concrete Textual Notation

```plaintext
ProcedureSignature returns tdl::ProcedureSignature:
   AnnotationCommentFragment
   'Signature' name=Identifier
   LParen parameter+=ProcedureParameter (',' parameter+=ProcedureParameter)* RParen

```

ETSI
Comments
No comments.

Examples

```c
Signature publish (in Post post, out Integer postId)
```

6.2.14 ProcedureParameter

Concrete Textual Notation

```
ProcedureParameter returns tdl::ProcedureParameter:
  AnnotationFragment
  kind=ParameterKind
dataType=[tdl::DataType | Identifier]
  name=Identifier
  WithCommentFragment?
;
```

Comments
No comments.

Examples

```
in Post post
out Integer postId
```

6.2.15 ParameterKind

Concrete Textual Notation

```
enum ParameterKind returns tdl::ParameterKind:
  In = "in" | Out = "out" | Exception = "exception"
;
```

Comments
No comments.

Examples

Void.

6.2.16 Parameter

Concrete Textual Notation

Void.

Comments

This is an abstract metaclass, therefore no textual representation is defined for the element.

Examples

Void.
6.2.17  FormalParameter

Concrete Textual Notation

FormalParameter returns $\text{idl::FormalParameter}$:

\begin{verbatim}
    AnnotationFragment
dataType=[idl::DataType | Identifier]
    name=Identifier
    WithCommentFragment?
\end{verbatim}

Comments
No comments.

Examples

@Encrypted
Post post
@Unique
Integer postId

6.2.18  Variable

Concrete Textual Notation

Variable returns $\text{idl::Variable}$:

\begin{verbatim}
    AnnotationCommentFragment
    'variable' dataType=[idl::DataType | Identifier]
    name=Identifier
    WithCommentFragment?
\end{verbatim}

Comments
No comments.

Examples

variable Binary authToken

6.2.19  Action

Concrete Textual Notation

Action returns $\text{idl::Action}$:

\begin{verbatim}
    AnnotationCommentFragment
    'Action' name=Identifier
    (LParen formalParameter+=FormalParameter ( ',' formalParameter+=FormalParameter)* RParen )?
    (':' body=EString)?
\end{verbatim}

Comments
No comments.

Examples

Action reset
Action clean: "Cleaning procedure: Wash hands, wear mask and gloves, open windows."
Action reload(Posts posts): "Reloading procedure: Clear all posts, reset, reload posts."
6.2.20  Function

Concrete Textual Notation

\[
\text{Function returns id::Function: AnnotationCommentFragment}
\]
\[
\text{"Function" name=Identifier
[LParen formalParameter+=FormalParameter (',' formalParameter+=FormalParameter)* RParen]?}
\]
\[
\text{"returns" returnType=[tdl::DataType|Identifier]
[':' body=EString]?}
\]

Comments
No comments.

Examples

\[
\text{Function categorizeReport(Post post, Tags tags) returns Post: "Categorize with text mining"}
\]

6.2.21  UnassignedMemberTreatment

Concrete Textual Notation

\[
\text{enum UnassignedMemberTreatment returns id::UnassignedMemberTreatment:}
\]
\[
\text{AnyValue = '?' | AnyValueOrOmit = '*'}
\]

Comments
No comments.

Examples

\[
\text{Post firstReport <?> |
  title = "first report",
  date = "today"}
\]

6.2.22  PredefinedFunction

Concrete Textual Notation

\[
\text{PredefinedFunction returns id::PredefinedFunction: AnnotationCommentFragment}
\]
\[
\text{"Predefined"}
\]
\[
\text{| name=PredefinedIdentifierBinary
| name=PredefinedIdentifierNot
| name=PredefinedIdentifierSize}
\]
\[
\text{| "returns" returnType=[tdl::DataType|Identifier]?}
\]

PredefinedIdentifierBinary returns ecore::EString:

\[
\text{'+' | '-' | '*' | '/' | 'mod'
| '>=' | '<=' | '==' | '!=' | 'and' | 'or' | 'xor'}
\]

PredefinedIdentifierNot returns ecore::EString:

\[
\text{"not"}
\]

PredefinedIdentifierSize returns ecore::EString:

\[
\text{"size"}
\]
Comments
The 'PredefinedFunction's shall be provided as a standard library.

Examples

```
Predefined ==
Predefined !=
Predefined +
```

6.2.23 EnumDataType

Concrete Textual Notation

```c
EnumDataType returns idl::EnumDataType:
    AnnotationCommentFragment
    "Enumerated" name=Identifier
    BEGIN
        value+=SimpleDataInstance | value+=SimpleDataInstance |
    END
```

Comments
No comments.

Examples

```
Enumerated Tag {
    Tag useful,
    Tag interesting,
    Tag report
}
```

6.2.24 DataUse

Concrete Textual Notation

```c
DataUse returns idl::DataUse:
    DataElementUse
    | StaticDataUse
    | DynamicDataUse

    fragment ReductionFragment returns idl::DataUse:
        |->reduction+=CollectionReference | |
        |* reduction+=MemberReference | |

    fragment ParameterBindingFragment returns idl::DataUse:
        |Paren [argument+=ParameterBinding | * argument+=ParameterBinding |] | RParen
```

Comments
This is an abstract metaclass, the textual representation depends on the concrete types indicated as alternative derivations. The reusable fragments can be embedded in the concrete textual notation of metaclasses inheriting from this metaclass.

'Annotation's and 'Comment's, as well as the 'name' property, are syntactically excluded from all concrete types.
Examples
Void.

6.2.25 ParameterBinding

Concrete Textual Notation

ParameterBinding returns tdl::ParameterBinding:
  parameter=tdl::Parameter|Identifier
  '=' dataUse=DataUse;

Comments
No comments.

Examples
Void.

6.2.26 MemberReference

Concrete Textual Notation

MemberReference returns tdl::MemberReference:
  member=tdl::Member|Identifier
  [->'[' collectionIndex=DataUse ']' ]?;

CollectionReference returns tdl::MemberReference:
  '[' collectionIndex=DataUse ']';

Comments
The 'CollectionReference' derivation is applicable in case only a collection reference is needed, for example, immediately after a 'DataUse' with a type resolving to a 'CollectionDataType'.

Examples
Post memberPost |
  title = randomPosts[1].title |

6.2.27 StaticDataUse

Concrete Textual Notation

StaticDataUse returns tdl::StaticDataUse:
  DataInstanceUse
  | SpecialValueUse
  | LiteralValueUse;

Comments
This is an abstract metaclass, the textual representation depends on the concrete types indicated as alternative derivations.
Examples
Void.

6.2.28 DataInstanceUse

Concrete Textual Notation

DataInstanceUse returns tdl::DataInstanceUse:

- "instance" dataInstance=[tdl::DataInstance | Identifier]
  ParameterBindingFragment?
  UnassignedFragment?
  ReductionFragment

- "an" "instance" tdI::structuredDataType=[tdl::StructuredDataType | Identifier]
  ParameterBindingFragment | CollectionItemFragmentDataInstanceUse
  UnassignedFragment?

- "an" "instance" UnassignedFragment?
  ParameterBindingFragment | CollectionItemFragmentDataInstanceUse

fragment UnassignedFragment returns tdl::DataInstanceUse:

- '<' unassignedMember=UnassignedMemberTreatment '>

fragment CollectionItemFragmentDataInstanceUse returns tdl::DataInstanceUse:

- '[' item+=DataUse (',' item+=DataUse)* ']'

Comments
No comments.

Examples

Test illustrateDataInstanceUse(Post parameterPost) uses base {
  //anonymous instance
  client::http sends an instance of Post(title = "anonymous post") to server::http
  //defined instance
  client::http sends instance examplePost(title = "overridden title") to server::http
  //defined parameter
  client::http sends parameter parameterPost(title = "overridden title") to server::http
  //value returned from function
  client::http sends instance returned from fetchPost(id = 1) to server::http
  //anonymous collection including all of the above and truly anonymous instances
  client::http sends new Posts{
    an instance of Post(title = "anonymous post"),
    instance examplePost(title = "overridden title"),
    parameter parameterPost(title = "overridden title"),
    instance returned from fetchPost(id = 1),
    an instance (title = "truly anonymous without type specification")
  } to server::http
}
6.2.29 SpecialValueUse

Concrete Textual Notation

SpecialValueUse returns tdl::SpecialValueUse:
  OmitValue | AnyValue | AnyValueOrOmit;

Comments
This is an abstract metaclass, the textual representation depends on the concrete types indicated as alternative derivations.

Examples
Void.

6.2.30 AnyValue

Concrete Textual Notation

AnyValue returns tdl::AnyValue:
  name= '?'
  [LBrace dataType=[tdl::DataType | Identifier] RBrace]?
;

Comments
No comments.

Examples
Void.

6.2.31 AnyValueOrOmit

Concrete Textual Notation

AnyValueOrOmit returns tdl::AnyValueOrOmit:
  name= '*'
;

Comments
No comments.

Examples
Void.

6.2.32 OmitValue

Concrete Textual Notation

OmitValue returns tdl::OmitValue:
  name= 'omit'
;
Comments
No comments.

Examples
Void.

6.2.33 LiteralValueUse

Concrete Textual Notation

LiteralValueUse returns td::LiteralValueUse:
|value=STRING | intValue=BIGINTEGER | boolValue=BOOLEAN|
  |
  LBrace dataType=[td::DataType | Identifier] RBrace
  |ParameterBindingFragment | ReductionFragment|
  |
  ;

Comments
No comments.

Examples

client::authToken = "101010"
client::authToken = 1234
client::loggedIn = true
client::failAfter = 5 {sec}
client::decodedPostWithOverriddenTitle = "E242A4D4'\h'" {Post}title = "new title"
client::decodedTitle = "E242A4D4'\h'" {Post}title
client::decodedPost = "[E242A4D4'\h,F2A2A2D3'\h']" {Posts}[1]

6.2.34 DynamicDataUse

Concrete Textual Notation

DynamicDataUse returns td::DynamicDataUse:
FunctionCall
| FormalParameterUse
| VariableUse
| PredefinedFunctionCall
| TimeLabelUse
;

Comments
This is an abstract metaclass, the textual representation depends on the concrete types indicated as alternative derivations.

Examples
Void.
6.2.35 FunctionCall

Concrete Textual Notation

```
FunctionCall returns idl::FunctionCall:
    'instance' 'returned' 'from' function=[idl::Function|Identifier]
    ParameterBindingFragment
    ReductionFragment
```

Comments
No comments.

Examples
```
client::authToken = instance returned from generateToken(seed = 12)
```

6.2.36 FormalParameterUse

Concrete Textual Notation

```
FormalParameterUse returns idl::FormalParameterUse:
    'parameter' parameter=[idl::FormalParameter|Identifier]
    ParameterBindingFragment | ReductionFragment
```

Comments
No comments.

Examples
```
client::encodedToken = retrieveToken(parameter tokenID)
```

6.2.37 VariableUse

Concrete Textual Notation

```
VariableUse returns idl::VariableUse:
    componentInstance=[idl::ComponentInstance|Identifier]
        '::' variable=[idl::Variable|Identifier]
    ParameterBindingFragment | ReductionFragment
```

Comments
No comments.

Examples
```
client::authToken = "101010"
client::encodedToken = encodeToken(client::authToken)
```
6.2.38 PredefinedFunctionCall

Concrete Textual Notation

PredefinedFunctionCall returns tdl::PredefinedFunctionCall:
  PredefinedFunctionCall be
  PredefinedFunctionCallNot
  PredefinedFunctionCallBinary

PredefinedFunctionCallSize returns tdl::PredefinedFunctionCall:
  functions[tdl::PredefinedFunction] PredefinedIdentifierSize
  LParen actualParameters+=DataUse RParen

PredefinedFunctionCallNot returns tdl::PredefinedFunctionCall:
  functions[tdl::PredefinedFunction] PredefinedIdentifierNot
  LParen actualParameters+=DataUse RParen

PredefinedFunctionCallBinary returns tdl::PredefinedFunctionCall:
  LParen actualParameters+=DataUse
  functions[tdl::PredefinedFunction] PredefinedIdentifierBinary
  actualParameters+=DataUse
  RParen

Comments
No comments.

Examples
assert (size(allPosts)== 4 )
assert not (client::authenticated)

6.2.39 DataElementUse

Concrete Textual Notation

DataElementUse returns tdl::DataElementUse:
  | dataElement=[tdl::NamedElement|Identifier]
  UnassignedFragmentNamedElement?
  ParameterBindingFragment?
  ReductionFragment
  | ['new' dataElement=[tdl::DataType|Identifier]]?
  UnassignedFragmentNamedElement?
  (ParameterBindingFragment | CollectionItemFragment)

fragment UnassignedFragmentNamedElement returns tdl::DataElementUse:
  '<' UnassignedMember=UnassignedMemberTreatment '>'

fragment CollectionItemFragment returns tdl::DataElementUse:
  '[' item+=DataUse [',' item+=DataUse] ']'

Comments
If no 'dataElement' is specified, or if the specified 'dataElement' is a 'DataType', 'ParameterBinding's or 'Collection' items shall be specified. Otherwise, 'ParameterBinding's and/or 'MemberReference's may be specified.
Examples

```c
test illustrateDataElementUse|Post parameter Post| uses base {
  //anonymous instance
  client::http sends new Post(title = "anonymous post") to server::http
  //defined instance
  client::http sends examplePost(title = "overridden title") to server::http
  //defined parameter
  client::http sends parameterPost(title = "overridden title") to server::http
  //Value returned from function
  client::http sends fetchPost(id = 1) to server::http
  //anonymous collection including all of the above and truly anonymous instances
  client::http sends new Posts[
    new Post(title = "anonymous post"),
    examplePost(title = "overridden title"),
    parameterPost(title = "overridden title"),
    fetchPost(id = 1),
    [title = "truly anonymous without type specification!"
  ] to server::http
}
```

6.3 Time

6.3.1 Time

Concrete Textual Notation

```c
Time returns std::Time:
  AnnotationCommentFragment
  "Time" name=Identifier
  ;
```

Comments

No comments.

Examples

- `Time seconds`
- `Time milliseconds`

6.3.2 TimeLabel

Concrete Textual Notation

```c
TimeLabel returns std::TimeLabel:
  name=Identifier "='now'"
  ;
```

Comments

'Annotation's and 'Comment's are syntactically excluded.

Examples

- `publicationTime = now`
6.3.3 TimeLabelUse

Concrete Textual Notation

```plaintext
TimeLabelUse returns tdl::TimeLabelUse:
   '@' timelabel=[id::TimeLabel|Identifier]
   '.' kind=TimeLabelUseKind;
```

Comments

Assignment of the 'reduction' and 'argument' properties is syntactically excluded.

Examples

```plaintext
@publicationTime
@publicationTime.last
```

6.3.4 TimeLabelUseKind

Concrete Textual Notation

```plaintext
enum TimeLabelUseKind returns tdl::TimeLabelUseKind:
   Last = 'last' | Previous = 'previous' | First = 'first'
```

Comments

No comments.

Examples

Void.

6.3.5 TimeConstraint

Concrete Textual Notation

```plaintext
TimeConstraint returns tdl::TimeConstraint:
   timeConstraintExpression=DataUse
```

Comments

'Annotation's and 'Comment's are syntactically excluded.

Examples

Void.

6.3.6 Timer

Concrete Textual Notation

```plaintext
Timer returns tdl::Timer:
   AnnotationCommentFragment
   'timer' name=Identifier
```

ETS
Comments
No comments.

Examples
			timer global

6.3.7 TimeOperation

Concrete Textual Notation

TimeOperation returns tdI::TimeOperation:
	Wait | Quiescence

Comments
This is an abstract metaclass, the textual representation depends on the concrete types indicated as alternative derivations.

Examples
Void.

6.3.8 Wait

Concrete Textual Notation

Wait returns tdI::Wait:
	AtomicPrefixFragment
	"wait" "for" period=DataUse
	| "on" componentInstance=[tdI::ComponentInstance | Identifier]

Comments
No comments.

Examples
	wait for 10
	wait for 10 on client

6.3.9 Quiescence

Concrete Textual Notation

Quiescence returns tdI::Quiescence:
	AtomicPrefixFragment
	"quiet" "for" period=DataUse
	| "on" componentInstance=[tdI::ComponentInstance | Identifier]
	| "gate" gateReference=[tdI::GateReference | GateIdentifier]

Comments
No comments.
Example

- quiet for 5
- quiet for 5 on server
- quiet for 5 on gate server::http

6.3.10 TimerOperation

Concrete Textual Notation

TimerOperation returns tdl::TimerOperation:
  TimerStart | TimerStop | TimeOut

Comments

This is an abstract metaclass, the textual representation depends on the concrete types indicated as alternative derivations.

Examples

Void.

6.3.11 TimerStart

Concrete Textual Notation

TimerStart returns tdl::TimerStart:
  AtomicPrefixFragment
  "start" componentInstance=[tdl::ComponentInstance|Identifier]
  ":" timer=[tdl::Timer|Identifier]
  "for" period=DataUse

Comments

No comments.

Examples

  start client::global for 10

6.3.12 TimerStop

Concrete Textual Notation

TimerStop returns tdl::TimerStop:
  AtomicPrefixFragment
  "stop" componentInstance=[tdl::ComponentInstance|Identifier]
  ":" timer=[tdl::Timer|Identifier]

Comments

No comments.

Examples

  stop client::global
6.3.13 TimeOut

Concrete Textual Notation

```plaintext
TimeOut returns tdl::TimeOut:
  AtomicPrefixFragment
  "timeout" on componentInstance=\[tdl::ComponentInstance | Identifier\]
  ;

Comments
No comments.

Examples
```
timeout on client::global
```

6.4 Test Configuration

6.4.1 GateType

Concrete Textual Notation

```plaintext
GateType returns tdl::GateType:
  AnnotationCommentFragment
  kind=GateTypeKind
   'Gate' name=Identifier
   'extends' extension=Extension? 
   'accepts' dataType+=\[tdl::DataType | Identifier\] ( ',' dataType+=\[tdl::DataType | Identifier\])*

Comments
No comments.

Examples
```
Message Gate HTTP accepts Post, Posts
Message Gate HTTPS extends HTTP accepts Binary
Procedure Gate RPC accepts publish
```

6.4.2 GateTypeKind

Concrete Textual Notation

```plaintext
enum GateTypeKind returns tdl::GateTypeKind:
  Message = 'Message' | Procedure = 'Procedure'

Comments
No comments.

Examples

Void.
6.4.3 GateInstance

Concrete Textual Notation

GateInstance returns tdl::GateInstance:
  AnnotationCommentFragment
  'gate' type=[tdl::GateType | Identifier]
  name=Identifier
  WithCommentFragment?

Comments
No comments.

Examples
  gate HTTP http
  gate RPC rpc

6.4.4 ComponentType

Concrete Textual Notation

ComponentType returns tdl::ComponentType:
  AnnotationCommentFragment
  'Component' name=Identifier
  ('extends' extension=Extension)?
  BEGIN
    |timer+=Timer | variable+=Variable | gateInstance+=GateInstance|*
  END

Comments
No comments.

Examples
  Component Node {
    timer global
    variable Binary authToken
    variable Integer lastPostId
    gate HTTP http
    gate RPC rpc
  }

  Component SecureNode extends Node {
    gate HTTPS https
  }

6.4.5 ComponentInstance

Concrete Textual Notation

ComponentInstance returns tdl::ComponentInstance:
  AnnotationCommentFragment
  type=[tdl::ComponentType | Identifier]
  name=Identifier
  'as' role=ComponentInstanceRole

Comments
No comments.

Examples
Node server as SUT
Node client as Tester

6.4.6 ComponentInstanceRole

Concrete Textual Notation

```cpp
enum ComponentInstanceRole returns tdl::ComponentInstanceRole:
  SUT = 'SUT' | Tester = 'Tester';
```

Comments
No comments.

Examples
Void.

6.4.7 GateReference

Concrete Textual Notation

```cpp
GateReference returns tdl::GateReference:
  [name=GRIdentifier =]?,
  component=[tdl::ComponentInstance | Identifier] 
  .
  gate=[tdl::GateInstance | Identifier];
```

Comments
No comments.

Examples
Void.

6.4.8 Connection

Concrete Textual Notation

```cpp
Connection returns tdl::Connection:
  AnnotationCommentFragment
  'connect' endPoint+=GateReference
  'to' endPoint+=GateReference
  WithNameFragment?;
```

Comments
No comments.
Examples

```plaintext
connect client::http to server::http
connect cRPC=client::rpc to sRPC=server::rpc
```

### 6.4.9 TestConfiguration

**Concrete Textual Notation**

```plaintext
TestConfiguration returns tdl::TestConfiguration:
  AnnotationCommentFragment
  'Configuration' name=Identifier
  BEGIN
    componentInstance+=ComponentInstance (',' componentInstance+=ComponentInstance)*
    | connection+=Connection)*
  END
```

**Comments**

No comments.

**Examples**

```plaintext
Configuration base {
  Node server as SUT,
  Node client as Tester,
  connect client::http to server::http,
  connect cRPC=client::rpc to sRPC=server::rpc
}
```

### 6.5 Test Behaviour

#### 6.5.1 TestDescription

**Concrete Textual Notation**

```plaintext
TestDescription returns id::TestDescription:
  TDPrefixFragment
  ['Test' 'Description' | isLocallyOrdered? 'Test']
  name=Identifier
  [LParen formalParameter+=FormalParameter (',' formalParameter+=FormalParameter)* RParen ]?
  'uses' testConfiguration=[tdl::TestConfiguration|Identifier]?
  (behaviourDescription=BehaviourDescription)?

fragment TDPrefixFragment returns id::TestDescription:
  TDObjectiveFragment?
  AnnotationCommentFragment

fragment TDObjectiveFragment returns id::TestDescription:
  'Objective:' testObjective+=[tdl::TestObjective|Identifier]
  | [',' testObjective+=[tdl::TestObjective|Identifier]]*
```

**Comments**

No comments.
6.5.2 BehaviourDescription

Concrete Textual Notation

BehaviourDescription returns tdl::BehaviourDescription:
    behaviour = Behaviour

Comments

'Annotation's and 'Comment's, as well as the 'name' property, are syntactically excluded from 'BehaviourDescription'.

Examples

Void.

6.5.3 Behaviour

Concrete Textual Notation

Behaviour returns tdl::Behaviour:
    CombinedBehaviour | AtomicBehaviour

fragment WithBehaviourFragment returns tdl::Behaviour:
    'with'
    BEGIN
        NameFragment?
        ObjectiveFragment?
        [comment+=Comment]*
    END

fragment ObjectiveFragment returns tdl::Behaviour:
    'Objective:' testObjective+=
        tdl::TestObjective | Identifier
        [', testObjective+=
        tdl::TestObjective | Identifier]*

Comments

The reusable fragments can be embedded in the concrete textual notation of metaclasses inheriting from this metaclass.

Examples

Void.
6.5.4 Block

Concrete Textual Notation

Block returns tdl::Block:
  \[
  [\text{guard}=\text{LocalExpression} \ | \ ! \text{guard}=\text{LocalExpression}]^* \]\;
BEGIN
  \text{behaviour}+=\text{Behaviour}+
END
;

Comments

'Annotation's and 'Comment's, as well as the 'name' property, are syntactically excluded from 'Block's. 'Annotation's and 'Comment's can be assigned to the containing 'CombinedBehaviour's.

Examples

Void.

6.5.5 LocalExpression

Concrete Textual Notation

LocalExpression returns tdl::LocalExpression:
  \text{expression}=\text{DataUse}
  \ | \ \text{on}^* \text{componentInstance}=\{\text{dl}::\text{ComponentInstance} \ | \ \text{Identifier}\}^* \text{?} ?
  ;

LocalLoopExpression returns tdl::LocalExpression:
  \text{expression}=\text{DataUse} \ \text{times}
  \ | \ \text{on}^* \text{componentInstance}=\{\text{dl}::\text{ComponentInstance} \ | \ \text{Identifier}\}^* \text{?} ?
  ;

Comments

The 'LocalLoopExpression' derivation is only used within 'BoundedLoopBehaviour's. 'Annotation's and 'Comment's, as well as the 'name' property, are syntactically excluded from 'LocalExpression's.

Examples

Void.

6.5.6 CombinedBehaviour

Concrete Textual Notation

CombinedBehaviour returns tdl::CombinedBehaviour:
  \{SingleCombinedBehaviour \ | \ MultipleCombinedBehaviour\} 
  =>\text{WithCombinedFragment}?
  ;

fragment \text{WithCombinedFragment} returns tdl::CombinedBehaviour:
  \text{with}^* \text{BEGIN}
  \text{NameFragment}^* \text{ObjectiveFragment}^* \text{(comment}+=\text{Comment}{)*}
  \text{periodic}+=\text{PeriodicBehaviour}{)*}
  \text{exceptional}+=\text{ExceptionalBehaviour}{)*}
  \text{END}
  ;
Comments
This is an abstract metaclass, the textual representation depends on the concrete types indicated as alternative derivations. The 'WithCombinedFragment' is always assigned to the innermost 'CombinedBehaviour'.

Examples
Void.

6.5.7 SingleCombinedBehaviour

Concrete Textual Notation

SingleCombinedBehaviour returns tdl::SingleCombinedBehaviour:
  CompoundBehaviour
  | BoundedLoopBehaviour
  | UnboundedLoopBehaviour
  | OptionalBehaviour
;

Comments
This is an abstract metaclass, the textual representation depends on the concrete types indicated as alternative derivations.

Examples
Void.

6.5.8 CompoundBehaviour

Concrete Textual Notation

CompoundBehaviour returns tdl::CompoundBehaviour:
  AnnotationFragment
  block=Block
;

Comments
No comments.

Examples

@Example
  [ "some expression" ] {
    perform action: "reload"
  }

@Example
  {
    perform action: "reload"
  }

6.5.9 BoundedLoopBehaviour

Concrete Textual Notation

BoundedLoopBehaviour returns tdl::BoundedLoopBehaviour:
  AnnotationFragment
  "repeat" numIteration+=LocalLoopExpression | "repeat" numIteration+=LocalLoopExpression |*
  block=Block
;
6.5.10 UnboundedLoopBehaviour

Concrete Textual Notation

```
UnboundedLoopBehaviour returns tdl::UnboundedLoopBehaviour:
   AnnotationFragment
   'while' block=Block
```

Comments
No comments.

Examples
```
while ["some expression"] {
   perform action: "reload"
}
while ["some expression" on client, "other expression" on server] {
   perform action: "reload"
}
```

6.5.11 OptionalBehaviour

Concrete Textual Notation

```
OptionalBehaviour returns tdl::OptionalBehaviour:
   AnnotationFragment
   "optionally" block=Block
```

Comments
No comments.

Examples
```
optionally {
   perform action: "reload"
}
```
6.5.12 MultipleCombinedBehaviour

Concrete Textual Notation

```
MultipleCombinedBehaviour returns tdl::MultipleCombinedBehaviour:
  ConditionalBehaviour
  | AlternativeBehaviour
  | ParallelBehaviour;
```

Comments

This is an abstract metaclass, the textual representation depends on the concrete types indicated as alternative derivations.

Examples

Void.

6.5.13 ConditionalBehaviour

Concrete Textual Notation

```
ConditionalBehaviour returns tdl::ConditionalBehaviour:
  AnnotationFragment
  'if' block+=Block
  | => ['else' block+=Block]
  | | => ['else' 'if' block+=Block]*
  | | | => ['else' block+=Block]??
```

Comments

The 'Block's are identified by the preceding keywords, where the first 'Block' is referred to as an 'if' 'Block' and the following 'Block's are referred to as 'else' or 'else if' 'Blocks', respectively. An 'else' 'Block' shall always be attached to the innermost 'if' 'Block'.

Examples

```
if [ "some expression"] {
  perform action: "reload"
}

if [ "some expression" on client, "other expression" on server] {
  perform action: "reload"
} else if [ "some expression" on client, "other expression" on server] {
  perform action: "backup"
} else {
  perform action: "query"
}
```

6.5.14 AlternativeBehaviour

Concrete Textual Notation

```
AlternativeBehaviour returns tdl::AlternativeBehaviour:
  AnnotationFragment
  'alternatively' block+=Block
  | | 'or' block+=Block
```

Comments
No comments.

Examples

alternatively {
    server::http sends "error" to client::http
    perform action: "reload"
} or {
    timeout on client::global
    perform action: "reload conditionally"
}

alternatively ['some expression'] {
    server::http sends "error" to client::http
    perform action: "reload"
} or ['some other expression'] {
    timeout on client::global
    perform action: "reload conditionally"
}

alternatively ['some expression' on client, 'other expression' on server] {
    server::http sends "error" to client::http
    perform action: "reload"
} or {
    timeout on client::global
    perform action: "reload conditionally"
}

6.5.15 ParallelBehaviour

Concrete Textual Notation

ParallelBehaviour returns tdl::ParallelBehaviour:
    Annotation Fragment
    'run' block+=Block
    ['in' 'parallel' to block+=Block]
    ['and' block+=Block]*

Comments
No comments.

Examples

run {
    perform action: 'reload'
} in parallel to {
    perform action: 'backup'
} and {
    perform action: 'query'
}

run ['some expression' on client, 'other expression' on server] {
    perform action: 'reload'
} in parallel to {
    perform action: 'backup'
} and ['some expression' on client, 'other expression' on server] {
    perform action: 'query'
}
6.5.16 ExceptionalBehaviour

Concrete Textual Notation

```cpp
ExceptionalBehaviour returns td::ExceptionalBehaviour:
    DefaultBehaviour | InterruptBehaviour
```

Comments

This is an abstract metaclass, the textual representation depends on the concrete types indicated as alternative derivations.

Examples

Void.

6.5.17 DefaultBehaviour

Concrete Textual Notation

```cpp
DefaultBehaviour returns td::DefaultBehaviour:
    AnnotationFragment
        'default'
        | 'on' guardedComponent=[td::ComponentInstance | Identifier]?
    block=Block
        WithBehaviourFragment?
```

Comments

No comments.

Examples

```cpp
while [ "some expression" on client, "other expression" on server] {
    perform action: "query"
} with {
    Note: "Applies to combined behaviour"
    default {
        server::http sends "error" to client::http
        perform action: "reload"
    } with {
        Note: "Applies to default"
    }
    default on client {
        timeout on client::global
        perform action: "reload"
    }
}
```

6.5.18 InterruptBehaviour

Concrete Textual Notation

```cpp
InterruptBehaviour returns td::InterruptBehaviour:
    AnnotationFragment
        'interrupt'
        | 'on' guardedComponent=[td::ComponentInstance | Identifier]?
    block=Block
        WithBehaviourFragment?
```
Comments
No comments.

Examples

```java
while ["some expression" on client, "other expression" on server] {
    perform action: "query"
} with {
    interrupt on client ["some condition"] {
        timeout on client::global
        perform action: "reload conditionally"
    }
}
```

6.5.19 PeriodicBehaviour

Concrete Textual Notation

```plaintext
PeriodicBehaviour returns tdl::PeriodicBehaviour:
    AnnotationFragment
    'every'
    [period+=LocalExpression |" period+=LocalExpression |"]
    block=Block
    WithBehaviourFragment?

; Comments
No comments.

Examples

```java
while ["some expression" on client, "other expression" on server] {
    perform action: "query"
} with {
    every 5 {sec} ["some expression" on client, "other expression" on server] {
        perform action: "reload"
    }
}
```

6.5.20 AtomicBehaviour

Concrete Textual Notation

```plaintext
AtomicBehaviour returns tdl::AtomicBehaviour:
    |TimerOperation
    |TimeOperation
    |Break | Stop
    |VerdictAssignment | Assertion
    |Interaction
    |TestDescriptionReference
    |ActionBehaviour
    |Assignment)
    WithAtomicFragment?

fragment AtomicPrefixFragment returns tdl::AtomicBehaviour:
    ObjectiveFragment?
    AnnotationCommentFragment

fragment WithAtomicFragment returns tdl::AtomicBehaviour:
    with
    BEGIN
    TimeLabelFragment?
    TimeConstraintFragment?
```
END

fragment TimeLabelFragment returns tdl::AtomicBehaviour:  
  timeLabel=TimeLabel

;  

fragment TimeConstraintFragment returns tdl::AtomicBehaviour:  
  LBrace timeConstraint+=TimeConstraint ( ',' timeConstraint+=TimeConstraint )* RBrace

Comments

The reusable fragments can be embedded in the concrete textual notation of metaclasses inheriting from this metaclass.

Examples

Objective: CheckAuthToken

client::http sends new Post() to server::http
  with {
    publicationTime=now
  }

Objective: CheckAuthToken

server::http sends authToken to client::http

  client::http sends parameter cleanPost to server::http

  server::http sends 'OK' to client::http
  with {
    clean Time=now
    { [\(@clean Time-@publicationTime\) <= 2 [sec]]}
  }

6.5.21 Break

Concrete Textual Notation

Break returns tdl::Break:  
  {tdl::Break}  
  AtomicPrefixFragment
  "break"

Comments

No comments.

Examples

break

6.5.22 Stop

Concrete Textual Notation

Stop returns tdl::Stop:  
  {tdl::Stop}  
  AtomicPrefixFragment
  "terminate"

;
Comments

No comments.

Examples

`terminate`

6.5.23 VerdictAssignment

Concrete Textual Notation

```
VerdictAssignment returns id::VerdictAssignment:
  AtomicPrefixFragment
  'set "verdict" to verdict=DataUse
```

Comments

No comments.

Examples

```
set verdict to fail
```

6.5.24 Assertion

Concrete Textual Notation

```
Assertion returns id::Assertion:
  {id::Assertion}
  AtomicPrefixFragment
  'assert' condition=DataUse
      | 'on' componentInstance=[id::ComponentInstance | Identifier]?
      | 'otherwise' otherwise=DataUse]?
```

Comments

No comments.

Examples

```
assert [client::authToken==referenceToken]
assert [client::authToken==referenceToken] on client
assert [client::authToken=="101010"] otherwise fail
```

6.5.25 Interaction

Concrete Textual Notation

```
Interaction returns id::interaction:
  Message | ReceiveMessage | ProcedureCall | ProcedureCallResponse
```

Comments

This is an abstract metaclass, the textual representation depends on the concrete types indicated as alternative derivations.
Examples

6.5.26 Message

Concrete Textual Notation

Message returns id::Message:
  AtomicPrefixFragment
  sourceGate=[tdl::GateReference | GRIdentifier]
  "sends" | (isTrigger? "triggers")?
  argument=DataUse
  "to" target=TargetMessage | "," target=TargetMessage*;

ReceiveMessage returns id::Message:
  AtomicPrefixFragment
  target+=ReceiveTargetMessage
  "receives" ?isTrigger? "trigger"?
  argument=DataUse
  "from" sourceGate=[tdl::GateReference | GRIdentifier]

Comments

A 'ReceiveMessage' can be used as an alternative way to specify a 'Message' by switching the source and target gates. Only a single 'Target' can be specified when within a 'ReceiveMessage'. If multiple 'Target's need to be specified, the 'Message' derivation shall be used instead.

Examples

Note: "Single target"
server::http sends new Binary to client::http

Note: "Multiple targets"
server::http sends new Binary to client::http, bridge::http

Note: "Single target assignment"
server::http sends new Binary to
  client::http where it is assigned to authToken

Note: "Multiple targets with assignment"
server::http sends new Binary to
  client::http where it is assigned to authToken,
  bridge::http where it is assigned to authToken

Note: "Alternative notation without assignment"
client::http receives new Binary from server::http

Note: "Alternative notation with assignment"
authToken = client::http receives new Binary from server::http

6.5.27 Target

Concrete Textual Notation

TargetMessage returns id::Target:
  targetGate=[tdl::GateReference | GRIdentifier]
  |valueAssignment+=ValueAssignmentMessage?

ReceiveTargetMessage returns id::Target:
  |valueAssignment+=ReceiveValueAssignmentMessage?
  targetGate=[tdl::GateReference | GRIdentifier]
TargetProcedure returns \texttt{id::Target}:
\texttt{targetGate=[\texttt{id::GateReference}\mid\texttt{GRIdentifier}]}\texttt{|valueAssignment+=ValueAssignmentProcedure \mid valueAssignment+=ValueAssignmentProcedure*}? ;

Comments

The alternative derivations are used in the respective contexts.

'Annotation's and 'Comment's, as well as the 'name' property, are syntactically excluded from all concrete types.

Examples

\texttt{Void}.

6.5.28 ValueAssignment

Concrete Textual Notation

ValueAssignmentMessage returns \texttt{id::ValueAssignment}:
\texttt{where"it"is"assigned"to'variable=[\texttt{id::Variable}\mid\texttt{Identifier}]};

ReceiveValueAssignmentMessage returns \texttt{id::ValueAssignment}:
\texttt{variable=[\texttt{id::Variable}\mid\texttt{Identifier}] =};

ValueAssignmentProcedure returns \texttt{id::ValueAssignment}:
\texttt{where"parameter=[\texttt{id::Parameter}\mid\texttt{Identifier}]"is"assigned"to'variable=[\texttt{id::Variable}\mid\texttt{Identifier}]};

Comments

The alternative derivations are used in the respective contexts.

'Annotation's and 'Comment's, as well as the 'name' property, are syntactically excluded from all concrete types.

Examples

\texttt{where it is assigned to authToken}
\texttt{where postId is assigned to lastPostId}

6.5.29 ProcedureCall

Concrete Textual Notation

ProcedureCall returns \texttt{id::ProcedureCall}:
\texttt{AtomicPrefixFragment [name=Identifier \mid]? \texttt{sourceGate=[\texttt{id::GateReference}\mid\texttt{GRIdentifier}] \texttt{\textquoteleft calls\textquoterightness} \texttt{signature=[\texttt{id::ProcedureSignature}\mid\texttt{Identifier}] \texttt{LParen argument+=ParameterBinding \mid argument+=ParameterBinding*} \texttt{RParen \textquoteleft on\textquoterightness} target+=TargetProcedure} ;}

ProcedureCallResponse returns \texttt{id::ProcedureCall}:
\texttt{AtomicPrefixFragment \texttt{[replyTo=[\texttt{id::ProcedureCall}\mid\texttt{Identifier}] \mid] \texttt{sourceGate=[\texttt{id::GateReference}\mid\texttt{GRIdentifier}] \texttt{\textquoteleft responds\textquoterightness} \texttt{with\textquoterightness} \texttt{signature=[\texttt{id::ProcedureSignature}\mid\texttt{Identifier}] \texttt{LParen argument+=ParameterBinding \mid argument+=ParameterBinding*} \texttt{RParen \textquoteleft to\textquoterightness} target+=TargetProcedure} ;}

\textbf{ETSI}
Comments

A 'ProcedureCallResponse' shall be used to specify a response after a 'ProcedureCall'. The 'ProcedureCall' shall be specified with an assigned 'name'-property within the 'WithCombinedFragment'.

Examples

publishCall: client::rpc calls publish [post=firstReport] on server::rpc
publishCall: server::rpc responds with publish [postId=1] to client::rpc
publishCall: server::rpc responds with publish [postId=1] to client::rpc
where postId is assigned to lastPostid

6.5.30 TestDescriptionReference

Concrete Textual Notation

TestDescriptionReference: returns [tdl::TestDescriptionReference:
  AtomicPrefixFragment 'execute' testDescription=[tdl::TestDescription] identifier |
  [LParen argument+=ParameterBinding ',' argument+=ParameterBinding RParen ]? |
  BEGIN componentInstanceBinding+=ComponentInstanceBinding |
  END ]? ;

Comments

No comments.

Examples

execute publishAll
execute publishClean |
  cleanPost = new Post<?> [title = "Cleaner post"],
  authRequest = "00111001"
| execute publishClean |
  cleanPost = new Post<?> [title = "Cleanest post"],
  authRequest = parameter authRequest |
  | client -> client,
  server -> server |

6.5.31 ComponentInstanceBinding

Concrete Textual Notation

ComponentInstanceBinding: returns [tdl::ComponentInstanceBinding:
  AnnotationCommentFragment formalComponent=[tdl::ComponentInstance] identifier |
  actualComponent=[tdl::ComponentInstance] identifier ]

Comments

No comments.

Examples

client -> webClient
server -> webServer
6.5.32 ActionBehaviour

Concrete Textual Notation

```plaintext
ActionBehaviour returns tdl::ActionBehaviour:
    ActionReference | InlineAction
;
```

Comments

This is an abstract metaclass, the textual representation depends on the concrete types indicated as alternative derivations.

Examples

Void.

6.5.33 ActionReference

Concrete Textual Notation

```plaintext
ActionReference returns tdl::ActionReference:
    AtomicPrefixFragment
    "perform" action=[tdl::Action | Identifier]
    (LParen argument+=ParameterBinding ( ',' argument+=ParameterBinding )* RParen )?
    ("on" componentInstance=[tdl::ComponentInstance | Identifier])?
;
```

Comments

No comments.

Examples

```plaintext
perform reset
perform reset on server
perform reload(posts = allPosts) on server
```

6.5.34 InlineAction

Concrete Textual Notation

```plaintext
InlineAction returns tdl::InlineAction:
    AtomicPrefixFragment
    "perform" "action" ':' body=EString
    ("on" componentInstance=[tdl::ComponentInstance | Identifier])?
;
```

Comments

No comments.

Examples

```plaintext
perform action : "Call administrator"
perform action : "Call administrator" on client
```
6.5.35 Assignment

Concrete Textual Notation

Assignment returns id: Assignment:
   AtomicPrefixFragment
   variable=VariableUse
   '=' expression=DataUse
;

Comments

No comments.

Examples

    client::authToken = "101010"
    client::authToken = instance returned from generateToken[seed = 12]
    client::authToken = generateToken[seed = 12]
Annex A (informative):
Technical Representation of the Complete Textual Syntax

The technical representation of the complete specification of the textual syntax is available in the TDL Open Source Project (TOP) [i.4]. The technical representation also includes the specification of the textual syntax for the Structured Test Objective [i.5] and Extended Test Configurations [i.6] TDL extensions.
Annex B (informative):
Examples

B.0 Overview

This annex provides several examples to illustrate how the different elements of the TDL Textual Syntax can be used and demonstrates the applicability of TDL in several different areas.

The first example in clause B.1 demonstrates the usage of data-related concepts. It showcases the indentation-based syntax variant.

The second example in clause B.2 shows a scenario when a 'Tester' performs a test scenario on one interface of the 'SUT'. The example is taken from ETSI TS 136 523-1 [i.2]. It showcases the brace-based syntax variant.

The third example in clause B.3 provides an example for interoperability testing in IMS. The example is taken from ETSI TS 186 011-2 [i.3]. It is illustrated by means of the indentation-based syntax.

The examples are also available online as part of the TDL Open-Source Project (TOP) [i.4], both using the brace-based and indentation-based variants of the syntax.

B.1 Illustration of Data Use

This example describes some of the concepts related to data and data mapping in TDL by means of the TDL Textual Syntax. It illustrates how data instances can be parameterized, mapped to concrete data entities specified in an external resource, e.g. a TTCN-3 file, or to a runtime URI where dynamic concrete data values might be stored by the execution environment during runtime in order to facilitate some basic data flow of dynamic values between different interactions. The example considers a scenario where the SUT is required to generate and maintain a session ID between subsequent interactions and an alternative realization where data flow is expressed with variables.

/*
 * Copyright (c) ETSI 2022.
 * This software is subject to copyrights owned by ETSI. Non-exclusive permission is hereby granted, free of charge, to copy, reproduce and amend this file under the following conditions: It is provided "as is", without warranty of any kind, expressed or implied.
 * ETSI shall never be liable for any claim, damages, or other liability arising from its use or inability of use. This permission does not apply to any documentation associated with this file for which ETSI keeps all rights reserved. The present copyright notice shall be included in all copies of whole or part of this file and shall not imply any sub-license right.
 *
 */

//A manually constructed example illustrating the data mapping concepts

package DataExample

//Alternatively the predefined verdicts may be used as well

type Verdict
Verdict PASS
Verdict FAIL

//Test objectives

objective CHECK_SESSION_ID_IS_MAINTAINED

description "Check whether the session ID is maintained after the first response."

//Data definitions

type SESSION_ID
SESSION_ID SESSION_ID_1
SESSION_ID SESSION_ID_2
Structure MSG |
| optional SESSION_ID session |
| MSG REQUEST_SESSION_ID |
| session = omit |
| MSG RESPONSE |
| session = ? |
| MSG MESSAGE |
| session = ? |

//Data mappings
//Load resource.ttcn3
Use "resource.ttcn3" as TTCN_MAPPING
Map MSG to "record_message" in TTCN_MAPPING as MSG_mapping
    session -> "session_id"
Map REQUEST_SESSION_ID to "template_message_request" in TTCN_MAPPING as REQUEST_mapping
Map RESPONSE to "template_response" in TTCN_MAPPING as RESPONSE_mapping
Map MESSAGE to "template_message" in TTCN_MAPPING as MESSAGE_mapping

//Map types and instances to TTCN-3 records and templates, respectively
//located in the used TTCN-3 file]
Use "runtime://sessions/" as RUNTIME_MAPPING
//Map session ID data instances to locations within the runtime URI
Map SESSION_ID_1 to "id_1" in RUNTIME_MAPPING as SESSION_ID_1_mapping
Map SESSION_ID_2 to "id_2" in RUNTIME_MAPPING as SESSION_ID_2_mapping

//Gate type definitions
Message Gate defaultGT accepts MSG

//Component type definitions
Component defaultCT
    gate defaultGT g

Configuration defaultTC
    defaultCT UE as SUT,
    defaultCT withVariable SS as Tester,
    connect SS::g to UE::g

//Test description definition
Test Description exampleTD uses defaultTC
    Note: "Tester requests a session id"
    SS::g sends REQUEST_SESSION_ID to UE::g
    Note: "SUT responds with a session id that is assigned to the URI provided by the execution environment"
    UE::g sends RESPONSE | session = SESSION_ID_1 | to SS::g
    Note: "Tester sends a message with the session id from the runtime URI"
    SS::g sends MESSAGE | session = SESSION_ID_1 | to UE::g
    alternatively
    Note: "SUT responds with the same session id"
    UE::g sends RESPONSE | session = SESSION_ID_1 | to SS::g
    set verdict to PASS
    or
    Note: "SUT responds with a new session id"
    UE::g sends RESPONSE | session = SESSION_ID_2 | to SS::g
    set verdict to FAIL

    Objective: CHECK_SESSION_ID_IS_MAINTAINED

//Alternative approach with variables

//Component type definitions
Component defaultCTwithVariable
    variable MSG v
    gate defaultGT v

Configuration defaultTCwithVariables
    defaultCT UE as SUT,
    defaultCTwithVariable SS as Tester,
connect SS::g to UE::g

**Test Description** example TD uses default TC with variables

- **Note**: "Tester requests a session id"
- **Note**: "SUT responds with a session id that is assigned to the URI provided by the execution environment"
- **UE::g** sends RESPONSE to SS::g where it is assigned to
- **Note**: "Tester sends a message with the session id from the runtime URI"
- **SS::g** sends MESSAGE | session = SS::v.session | to UE::g
- **Note**: "SUT responds with the same session id"
- **UE::g** sends RESPONSE | session = SS::v.session | to SS::g
- **Note**: "SUT responds with a new session id"
- **UE::g** sends RESPONSE to SS::g
- **set verdict to PASS**
- **Objective**: CHECK_SESSION_ID_IS_MAINTAINED

**B.2 Interface Testing**

This example describes one possible way to translate clause 7.1.3.1 from ETSI TS 136 523-1 [i.2] into the brace-based variant of the TDL Textual Syntax, by mapping the concepts from the representation in the source document to the corresponding concepts in the TDL meta-model. The example has been enriched with additional information, such as explicit data definitions and test configuration details for completeness where applicable.

```/*
Copyright (c) ETSI 2022.
This software is subject to copyrights owned by ETSI. Non-exclusive permission is hereby granted, free of charge, to copy, reproduce and amend this file under the following conditions: It is provided "as is", without warranty of any kind, expressed or implied.
ETSI shall never be liable for any claim, damages, or other liability arising from its use or inability of use. This permission does not apply to any documentation associated with this file for which ETSI keeps all rights reserved. The present copyright notice shall be included in all copies of whole or part of this file and shall not imply any sub-license right.
*/
//Translated from ETSI TS 136 523-1 [i.2], clause 7.1.3.

**Note**: "Taken from ETSI TS 136 523-1 [i.2] V10.2.0 (2012-09)"

**@TITLE**: "Correct handling of DL assignment / Dynamic case"

**Package** Layer_2_DL_SCH_Data_Transfer {

**//Procedures carried out by a component of a test configuration**
**//for an actor during test execution**

**Action** preCondition : "Pre-test Conditions:
RRC Connection Reconfiguration"

**Action preamble**: "Preamble:
The generic procedure to get UE in test state loopback
Activated State 4 according to ETSI TS 136 508 [i.10] clause 4.5
is executed, with all the parameters as specified in the
procedure except that the RLC SDU size is set to return no
data in uplink.
[reference corresponding behavior once implemented]"

**//User-defined verdicts**
**//Alternatively the predefined verdicts may be used as well**
**Type Verdict**
- Verdict PASS
- Verdict FAIL

**//User-defined annotation types**
**Annotation** TITLE //test description title
**Annotation** STEP //Step identifiers in source documents
```
// Test objectives (copied verbatim from source document)

Objective TP1 {
    Description: "with {
        UE in E-UTRA RRC_CONNECTED state
    } ensure that {
        when {
            UE receives downlink assignment on the PDCCH
            for the UE's C-RNTI and receives data in the
            associated subframe and UE performs HARQ
            operation
        } then {
            UE sends a HARQ feedback on the HARQ process
        }"
    }
    References: "36523-1-a20_s07_01.doc::7.1.3.1.1 |1 |
}

Objective TP2 {
    Description: "with {
        UE in E-UTRA RRC_CONNECTED state
    } ensure that {
        when {
            UE receives downlink assignment on the PDCCH
            with a C-RNTI unknown by the UE and data is
            available in the associated subframe
        } then {
            UE does not send any HARQ feedback on the
            HARQ process
        }"
    }
    References: "36523-1-a20_s07_01.doc::7.1.3.1.1 [2]"
}

// Relevant data definitions

Type PDU
    PDU mac_pdu
Type ACK
    ACK harq_ack
Type C_RNTI
    C_RNTI ue
    C_RNTI unknown
Structure PDCCH |
    optional C_RNTI c_rnti
    PDCCH pdcch |

// User-defined time units

Time sec

// Gate type definitions

Message Gate defaultGT accepts ACK,PDU,PDCCH,C_RNTI

// Component type definitions

Component defaultCT {
    gate defaultGT g
}

// Test configuration definition

Configuration defaultTC {
    defaultCT SystemSimulator as Tester,
    defaultCT UserEquipment as SUT,
    connect UE= UserEquipment::g to SS= SystemSimulator::g
}
Test Description TD_7_1_3_1 uses default TC

//Pre-conditions and preamble from the source document
@PRECONDITION
perform preCondition
@PREAMBLE
perform preamble

//Test sequence
@STEP : "1"
@PROCEDURE : "SS transmits a downlink assignment including the C-RNTI assigned to the UE"
SS sends pdcch \( |_{c\_rnti=ue} \) to UE
@STEP : "2"
@PROCEDURE : "SS transmits in the indicated downlink assignment a RLC PDU in a MAC PDU"
SS sends mac_pdu to UE
Objective: TP1
@STEP : "3"
@PROCEDURE : "Check: Does the UE transmit an HARQ ACK on PUCCH?"
UE sends harq_ack to SS
set verdict to PASS
@STEP : "4"
@PROCEDURE : "SS transmits a downlink assignment to including a C-RNTI different from the assigned to the UE"
SS sends pdcch \( |_{c\_rnti=unknown} \) to UE
@STEP : "5"
@PROCEDURE : "SS transmits in the indicated downlink assignment a RLC PDU in a MAC PDU"
SS sends mac_pdu to UE
//Interpolated original step 6 in an alternative behaviour, covering both the incorrect and the correct behaviours of the UE
@STEP : "6"
@PROCEDURE : "Check: Does the UE send any HARQ ACK on PUCCH?"
alternatively{
    UE sends harq_ack to SS
    set verdict to FAIL
} or {
    quiet for 5 \( \) sec on gate SS
    set verdict to PASS
} with {
    Objective: TP2
}
}

Note : "Note 1: For TDD, the timing of ACK/NACK is not constant as FDD, see Table 10.1-1 of ETSI TS 136 213 [i.9],"
B.3 Interoperability Testing

This example describes one possible way to translate clause 4.5.1 from ETSI TS 186 011-2 [i.3] into the TDL Textual Syntax, by mapping the concepts from the representation in the source document to the corresponding concepts in the TDL meta-model. The example has been enriched with additional information, such as explicit data definitions and test configuration details for completeness where applicable.

/*
Copyright (c) ETSI 2022.

This software is subject to copyrights owned by ETSI. Non-exclusive permission is hereby granted, free of charge, to copy, reproduce and amend this file under the following conditions: It is provided "as is", without warranty of any kind, expressed or implied.

ETSI shall never be liable for any claim, damages, or other liability arising from its use or inability of use. This permission does not apply to any documentation associated with this file for which ETSI keeps all rights reserved. The present copyright notice shall be included in all copies of whole or part of this file and shall not imply any sub-license right.
*/

//Translated from ETSI TS 186 011-2 [i.3], clause 4.5.1.

Note: "Taken from ETSI TS 186 011-2 [i.3] V3.1.1 (2011-06)"

@TITLE : "SIP messages longer than 1 500 bytes"

Package IMS_NNI_General_Capabilities

//Procedures carried out by a component of a test configuration

Action preConditions : "Pre-test conditions:
- HSS of IMS_A and of IMS B is configured according to table 1
- UE_A and UE_B have IP bearers established to their respective IMS networks as per clause 4.2.1
- UE_A and IMS_A configured to use TCP for transport
- UE_A is registered in IMS_A using any user identity
- UE_B is registered user of IMS_B using any user identity
- MESSAGE request and response has to be supported at II-NNI
  [ETSI TS 129 165 [i.7]
  see tables 6.1 and 6.3]"

//User-defined verdicts
//Alternatively the predefined verdicts may be used as well

Type Verdict
Verdict PASS
Verdict FAIL

//User-defined annotation types

Annotation TITLE //test description title
Annotation STEP //Step identifiers in source documents
Annotation PROCEDURE //Informal textual description of a test step
Annotation PRECONDITION //Identify pre-condition behaviour
Annotation PREAMBLE //Identify preamble behaviour.
Annotation SUMMARY //Informal textual description of test sequence

//Test objectives (copied verbatim from source document)

Objective TP_IMS_4002_1
Description: "Ensure that {
  when (UE_A sends a MESSAGE to UE_B
  containing a Message_Body greater than 1 300 bytes)
  then (IMS_B receives the MESSAGE containing the
  Message_Body greater than 1 300 bytes)
}"

References: "ts_18601102v030101p.pdf::4.5.1.1 [CC 1]"
,"ETSI TS 124 229 [i.8], clause 4.2A, paragraph 1"

Objective UC_OC_1
References: "ts_18601102v030101p.pdf::4.4.4.2"

//Relevant data definitions

Structure MSG
  optional CONTENT TCP
  MSG MESSAGE ||
MSG DING ||
MSG DELIVERY_REPORT ||
MSG M_200_OK ||
Type CONTENT
CONTENT tcp
Time SECONDS
SECONDS default_timeout

//Gate type definitions.
Message Gate defaultGT accepts MSG,CONTENT

//Component type definitions
//In this case they may also be reduced to a single component type
Component USER
gate defaultGT g
Component UE
gate defaultGT g
Component IMS
gate defaultGT g
Component IBCF
gate defaultGT g

//Test configuration definition
Configuration CF_INT_CALL
USER USER_A as Tester,
UE UE_A as Tester,
IMS IMS_A as Tester,
IBCF IBCF_A as Tester,
IBCF IBCF_B as Tester,
IMS IMS_B as SUT,
UE UE_B as Tester,
USER USER_B as Tester,
connect USER_A::g to UE_A::g,
connect UE_A::g to IMS_A::g,
connect IMS_A::g to IBCF_A::g,
connect IBCF_A::g to IBCF_B::g,
connect IBCF_B::g to IMS_B::g,
connect IMS_B::g to UE_B::g,
connect UE_B::g to USER_B::g

//Test description definition
Test Description TD_IMS_MESS_0001 uses CF_INT_CALL
@SUMMARY: "IMS network shall support SIP messages greater than 1500 bytes"

//Pre-conditions from the source document
@PRECONDITION
perform preConditions

//Test sequence
@STEP: "1"
USER_A::g sends MESSAGE to UE_A::g
@STEP: "2"
UE_A::g sends MESSAGE to IMS_A::g
@STEP: "3"
IMS_A::g sends MESSAGE to IBCF_A::g
@STEP: "4"
IBCF_A::g sends MESSAGE to IBCF_B::g
@STEP: "5"
IBCF_B::g sends MESSAGE | TCP = tcp | to IMS_B::g
@STEP: "6"
IMS_B::g sends MESSAGE to UE_B::g
@STEP: "7"
UE_B::g sends DING to USER_B::g
@STEP: "8"
UE_B::g sends M_200_OK to IMS_B::g
@STEP: "9"
IMS_B::g sends M_200_OK to IBCF_B::g
@STEP: "10"
IBCF_B::g sends M_200_OK to IBCF_A::g
@STEP: "11"
IBCF_A::g sends M_200_OK to IMS_A::g
@STEP: "12"
IMS_A::g sends M_200_OK to UE_A::g
Alternatively

@STEP: '13'

UE_A::g sends DELIVERY_REPORT to USER_A::g

or

quiet for default timeout on gate USER_A::g
### History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Event</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1.1.1</td>
<td>May 2022</td>
<td>Publication</td>
<td></td>
</tr>
<tr>
<td>V1.2.1</td>
<td>August 2023</td>
<td>Membership Approval Procedure</td>
<td>MV 20231023: 2023-08-24 to 2023-10-23</td>
</tr>
<tr>
<td>V1.2.1</td>
<td>October 2023</td>
<td>Publication</td>
<td></td>
</tr>
</tbody>
</table>