ETSI ES 203 119-1 V1.3.1 (2016-09)

ETSI STANDARD

Methods for Testing and Specification (MTS);
The Test Description Language (TDL);
Part 1: Abstract Syntax and Associated Semantics

Reference
RES/MTS-203119-1v1.3.1

Keywords
language, MBT, methodology, testing, TSS&TP, TTCN-3, UML

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° 7803/88

Contents

Intellectual Property Rights

IPRs essential or potentially essential to the present document may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is 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 IPR Policy, no investigation, 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.

Foreword

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

The present document is part 1 of a multi-part deliverable covering the Test Description Language, as identified below:

Part 1: "Abstract Syntax and Associated Semantics";

Part 2: "Graphical Syntax";

Part 3: "Exchange Format";

Part 4: "Structured Test Objective Specification (Extension)".

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 abstract syntax of the Test Description Language (TDL) in the form of a meta-model based on the OMG® Meta Object Facility™ (MOF) [1]. It also specifies the semantics of the individual elements of the TDL meta-model. The intended use of the present document is to serve as the basis for the development of TDL concrete syntaxes aimed at TDL users and to enable TDL tools such as documentation generators, specification analyzers and code generators.

The specification of concrete syntaxes for TDL is outside the scope of the present document. However, for illustrative purposes, an example of a possible textual syntax together with its application on some existing ETSI test descriptions are provided.

NOTE: OMG®, UML®, OCL™ and UTP™ are the trademarks of OMG (Object Management Group). 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 http://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.

[1] OMG®: "OMG Meta Object Facility™ (MOF) Core Specification V2.4.1", formal/2013-06-01.

NOTE: Available at http://www.omg.org/spec/MOF/2.4.1/.

[2] OMG®: "OMG Unified Modeling Language™ (OMG UML) Superstructure, Version 2.4.1", formal/2011-08-06.

NOTE: Available at http://www.omg.org/spec/UML/2.4.1/.

[3] OMG®: "Object Constraint Language™ (OCL), Version 2.4", formal/2014-02-03.

NOTE: Available at http://www.omg.org/spec/OCL/2.4/.

[4] Void.

[5] ETSI ES 203 119-3 (V1.2.1): "Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 3: Exchange Format".

[6] ETSI ES 203 119-4 (V1.2.1): "Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 4: Structured Test Objective Specification (Extension)".

[7] ISO/IEC 9646-1:1994: "Information technology - Open Systems Interconnection -- Conformance testing methodology and framework -- Part 1: General concepts".

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] ETSI ES 201 873-1 (V4.5.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language".

[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".

3 Definitions and abbreviations

3.1 Definitions

For the purposes of the present document, the following terms and definitions apply:

abstract syntax: graph structure representing a TDL specification in an independent form of any particular encoding

action: any procedure carried out by a component of a test configuration or an actor during test execution

actor: abstraction of entities outside a test configuration that interact directly with the components of that test configuration

component: active element of a test configuration that is either in the role tester or system under test

concrete syntax: particular representation of a TDL specification, encoded in a textual, graphical, tabular or any other format suitable for the users of this language

interaction: any form of communication between components that is accompanied with an exchange of data

meta-model: modelling elements representing the abstract syntax of a language

system under test (SUT): role of a component within a test configuration whose behaviour is validated when executing a test description

TDL model: instance of the TDL meta-model

TDL specification: representation of a TDL model given in a concrete syntax

test configuration: specification of a set of components that contains at least one tester component and one system under test component plus their interconnections via gates and connections

test description: specification of test behaviour that runs on a given test configuration

test verdict: result from executing a test description

tester: role of a component within a test configuration that controls the execution of a test description against the components in the role system under test

tester-input event: event that occurs at a component in the role tester and determines the subsequent behaviour of this tester component

<undefined>: semantical concept denoting an undefined data value

3.2 Abbreviations

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

ADT	Abstract Data Type
EBNF	Extended Backus-Naur Form
IEC	International Electrotechnical Commission
IMS	IP Multimedia Subsystem
ISO	International Organization for Standardization
MBT	Model-Based Testing
MOF	Meta-Object Facility™
OCL	Object Constraint Language™
OMG	Object Management Group®
SUT	System Under Test
TDD	Test Driven Development
TDL	Test Description Language
TTCN-3	Testing and Test Control Notation version 3
UML	Unified Modelling Language®
URI	Unified Resource Identifier
XML	eXtensible Markup Language

4 Basic Principles

4.1 What is TDL?

TDL is a language that supports the design and documentation of formal test descriptions that may be the basis for the implementation of executable tests in a given test framework, such as TTCN-3 [i.1]. Application areas of TDL that will benefit from this homogeneous approach to the test design phase include:

- Manual design of test descriptions from a test purpose specification, user stories in test driven development or other sources.
- Representation of test descriptions derived from other sources such as MBT test generation tools, system simulators, or test execution traces from test runs.

TDL supports the design of black-box tests for distributed, concurrent real-time systems. It is applicable to a wide range of tests including conformance tests, interoperability tests, tests of real-time properties and security tests based on attack traces.

TDL clearly separates the specification of tests from their implementation by providing an abstraction level that lets users of TDL focus on the task of describing tests that cover the given test objectives rather than getting involved in implementing these tests to ensure their fault detection capabilities onto an execution framework.

TDL is designed to support different abstraction levels of test specification. On one hand, the concrete syntax of the TDL meta-model may hide meta-model elements that are not needed for a declarative (more abstract) style of specifying test descriptions. For example, a declarative test description could work with the time operations *wait* and *quiescence* instead of explicit timers and operations on timers (see clause 9).

On the other hand, an imperative (less abstract or refined) style of a test description supported by a dedicated concrete syntax could provide additional means necessary to derive executable test descriptions from declarative test descriptions. For example, an imperative test description could include timers and timer operations necessary to implement the reception of SUT output at a tester component and further details. It is expected that most details of a refined, imperative test description can be generated automatically from a declarative test description. Supporting different levels of abstraction by a single TDL meta-model offers the possibility of working within a single language and using the same tools, simplifying the test development process that way.

4.2 Design Considerations

TDL makes a clear distinction between concrete syntax that is adjustable to different application domains and a common abstract syntax, which a concrete syntax is mapped to (an example concrete syntax is provided in annex B). The definition of the abstract syntax for a TDL specification plays the key role in offering interchangeability and unambiguous semantics of test descriptions. It is defined in the present document in terms of a MOF meta-model.

A TDL specification consists of the following major parts that are also reflected in the meta-model:

- A test configuration consisting of at least one tester and at least one SUT component and connections among them reflecting the test environment.
- A set of test descriptions, each of them describing one test scenario based on interactions between the components of a given test configuration and actions of components or actors. The control flow of a test description is expressed in terms of sequential, alternative, parallel, iterative, etc. behaviour.
- A set of data definitions that are used in interactions and as parameters of test description invocations.
- Behavioural elements used in test descriptions that operate on time.

Using these major ingredients, a TDL specification is abstract in the following sense:

- Interactions between tester and SUT components of a test configuration are considered to be atomic and not detailed further. For example, an interaction can represent a message exchange, a remote function/procedure call, or a shared variable access.
- All behavioural elements within a test description are totally ordered, unless it is specified otherwise. That is, there is an implicit synchronization mechanism assumed to exist between the components of a test configuration.
- The behaviour of a test description represents the expected, foreseen behaviour of a test scenario assuming an implicit test verdict mechanism, if it is not specified otherwise. If the specified behaviour of a test description is executed, the 'pass' test verdict is assumed. Any deviation from this expected behaviour is considered to be a failure of the SUT, therefore the 'fail' verdict is assumed.
- An explicit verdict assignment may be used if in a certain case there is a need to override the implicit verdict setting mechanism (e.g. to assign 'inconclusive' or any user-defined verdict values).
- The data exchanged via interactions and used in parameters of test descriptions are represented as values of an abstract data type without further details of their underlying semantics, which is implementation-specific.
- There is no assumption about verdict arbitration, which is implementation-specific. If a deviation from the specified expected behaviour is detected, the subsequent behaviour becomes undefined. In this case an implementation might stop executing the TDL specification.

A TDL specification represents a closed system of tester and SUT components. That is, each interaction of a test description refers to one source component and at least one target component that are part of the underlying test configuration a test description runs on. The actions of the actors (entities of the environment of the given test configuration) may be indicated in an informal way.

Time in TDL is considered to be global and progresses in discrete quantities of arbitrary granularity. Progress in time is expressed as a monotonically increasing function. Time starts with the execution of the first ('base') test description being invoked.

The elements in a TDL specification may be extended with tool, application, or framework specific information by means of annotations.

4.3 Principal Design Approach

The language TDL is designed following the meta-modelling approach which separates the language design into abstract syntax and concrete syntax on the one hand, and static semantics and dynamic semantics on the other hand. The abstract syntax of a language describes the structure of an expression defined in the language by means of abstract concepts and relationships among them, where a concrete syntax describes concrete representation of an expression defined in this language by means of textual, graphical, or tabular constructs which are mapped to concepts from the abstract syntax. The semantics describes the meaning of the individual abstract syntax concepts.

The realization of multiple representations by means of different syntactical notations for a single language requires a clear distinction between abstract syntax and concrete syntax. In a model-based approach to language design, the abstract syntax is defined by means of a meta-model. The meta-model of TDL defines the underlying structure of the abstract concepts represented by means of textual, graphical, or tabular constructs, without any restrictions on how these are expressed by means of e.g. keywords, graphical shapes, or tabular headings. The concrete syntax provides means for the representation of the abstract concepts in the form of textual, graphical, or tabular constructs and defines mappings between the concrete representations and the abstract concepts. This approach allow any concrete representation conforming to a given meta-model to be transformed into another representation conforming to that meta-model, such as graphical to textual, textual to tabular, tabular to graphical, etc. The transformations on the concrete syntax level have no impact on the semantics of the underlying abstract syntax concepts.

The semantics of a language is divided into static semantics and dynamic semantics. The static semantics defines further restrictions on the structure of abstract syntax concepts that cannot be expressed in syntax rules. The dynamic semantics defines the meaning of a syntactical concept when it is put into an execution environment.

The four pieces of the TDL design, concrete syntax, abstract syntax, static semantics, dynamic semantics, are mapped to the standards series of TDL as follows (see figure 4.1):

- TDL-MM, part 1: Covers abstract syntax, static semantics and dynamic semantics;
- TDL-GR, part 2: Covers concrete syntax of graphical TDL;
- TDL-XF, part 3: Covers concrete syntax of the XML-based TDL exchange format;
- TDL-TO, part 4: Covers all parts of concrete/abstract syntax and static/dynamic semantics of the TDL Test Objective extension.

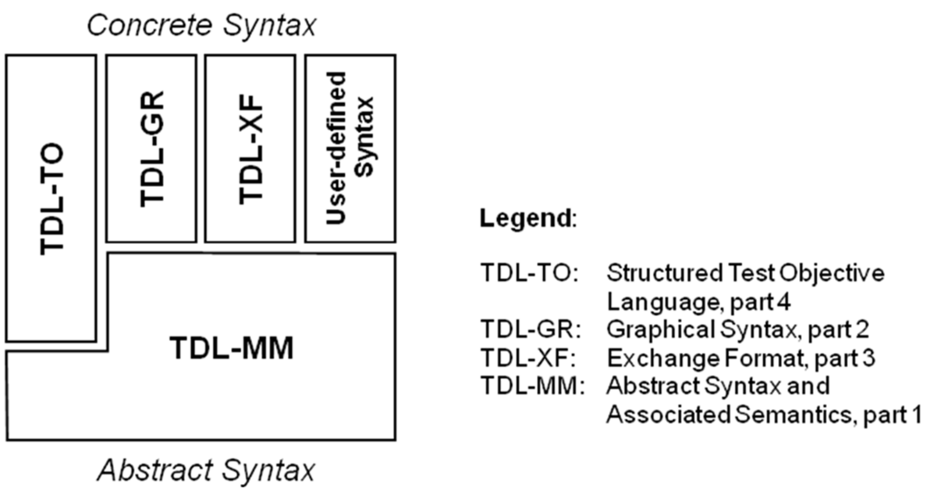

Figure 4.1: The TDL standards and their relation

This decomposition of the TDL language design into the different standard parts allows for the development of integrated and stand-alone tools: editors for TDL specifications in graphical, textual, and user-defined concrete syntaxes, analyzers of TDL specifications that check the consistency of TDL specifications, test documentation generators, test code generators to derive executable tests and others. In all cases the TDL exchange format [5] serves as the bridge between all TDL tools and to ensure tool interoperability (see figure 4.2).

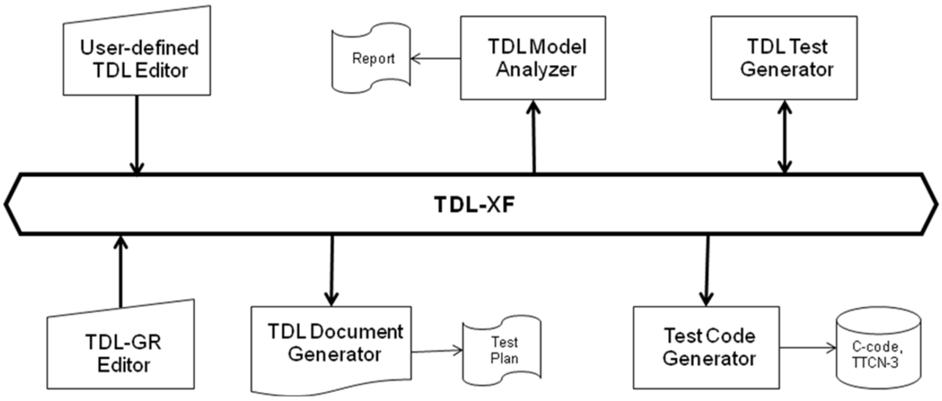

Figure 4.2: A scalable TDL tool architecture

4.4 Document Structure

The present document defines the TDL abstract syntax expressed as a MOF meta-model. The TDL meta-model offers language features to express:

- Fundamental concepts such as structuring of TDL specifications and tracing of test objectives to test descriptions (clause 5).
- Abstract representations of data used in test descriptions (clause 6).
- Concepts of time, time constraints, and timers as well as their related operations (clause 7).
- Test configurations, on which test descriptions are executed (clause 8).
- A number of behavioural operations to specify the control flow of test descriptions (clause 9).
- A set of predefined instances of the TDL meta-model for test verdict, time, data types and functions over them that may be extended further by a user (clause 10).

Each language feature clause contains a brief introduction to the concepts defined in that clause. A set of class diagrams defines the concepts associated with the feature. For each concept, properties and relationships are specified and visualized in the diagrams (figures in the present document). The defining instance of a concept (with icon and property compartment) appears only once in the set of diagrams. However, a concept may occur more than once in diagrams, in which case subsequent occurrences omit the icon and property compartment.

Besides the diagrams introducing the abstract syntax of the various TDL concepts formally, each clause is structured into the following paragraphs:

- Paragraph "Semantics" refers to the dynamic semantics of the concept defined in a declarative style hereafter. To emphasize the dynamic semantics aspect, sometimes the expression "at runtime" is used in the description. The description is augmented frequently with further explanations to ease reading interpretation of the document. These explanations are provided as NOTEs.
- Paragraph "Generalization" is derived from the abstract syntax diagram (figure) and lists the concept, which the defined concept is a specialization from. There is at most one generalization for any defined concept.
- Paragraph "Properties" is derived from the abstract syntax diagram (figure) and describes informally the meaning of the attributes that belong to the defined concept.
- Paragraph "Constraints" lists rules describing the static semantics of the concept, both in terms of informal descriptions and formally as OCL constraints.

4.5 Notational Conventions

In the present document, the following notational conventions are applied:

'element'	The name of an element or of the property of an element from the meta-model, e.g. the name of a meta-class.
«metaclass»	Indicates an element of the meta-model, which corresponds to the TDL concept in the abstract syntax, i.e. an intermediate node if the element name is put in *italic* or a terminal node if given in plain text.
«Enumeration»	Denotes an enumeration type.
/ name	The value with this name of a property or relation is derived from other sources within the meta-model.
[1]	Multiplicity of 1, i.e. there exists exactly one element of the property or relation.
[0..1]	Multiplicity of 0 or 1, i.e. there exists an optional element of the property or relation.
[*] or [0..*]	Multiplicity of 0 to many, i.e. there exists a possibly empty set of elements of the property or relation.
[1..*]	Multiplicity of one to many, i.e. there exists a non-empty set of elements of the property or relation.
{unique}	All elements contained in a set of elements shall be unique.
{ordered}	All elements contained in a set of elements shall be ordered, i.e. the elements form a list.
{readOnly}	The element shall be accessed read-only, i.e. shall not be modified. Used for derived properties.
inv [Name]:	Formal definition of a constraints by means of OCL [3], where [Name] is a placeholder for the unique constraint name.

Furthermore, the definitions and notations from the MOF 2 core framework [1] and the UML class diagram definition [2] apply.

4.6 OCL Constraints Requirements

In addition to the operations provided by the standard library of OCL, the formalized constraints rely on the following additional operations that serve as reusable shortcuts and shall be provided for the interpretation of the OCL constraints:

- OclAny **container**(): Element - applicable on any TDL 'Element', returns the 'Element' that contains the construct directly.
- OclAny **getTestDescription**(): TestDescription - applicable on any TDL 'Element', returns the 'TestDescription' that contains the construct directly or indirectly.
- DataUse **getDataType** (): DataType - applicable on any TDL 'DataUse', returns the 'DataType' resolved from the 'DataUse'.
- Behaviour **isTesterInputEvent** (): Boolean - applicable on any TDL 'Behaviour', returns the 'true' if the 'Behaviour' is a *tester-input event* as defined in this document, and 'false'otherwise.

4.7 Conformance

For an implementation claiming to conform to this version of the TDL meta-model, all features specified in the present document shall be implemented consistently with the requirements given in the present document. The electronic attachment in annex A may serve as a starting point for a TDL meta-model implementation conforming to the present document.

5 Foundation

5.1 Overview

The 'Foundation' package specifies the fundamental concepts of the TDL meta-model. All other features of the TDL meta-model rely on the concepts defined in this 'Foundation' package.

5.2 Abstract Syntax and Classifier Description

5.2.1 Element

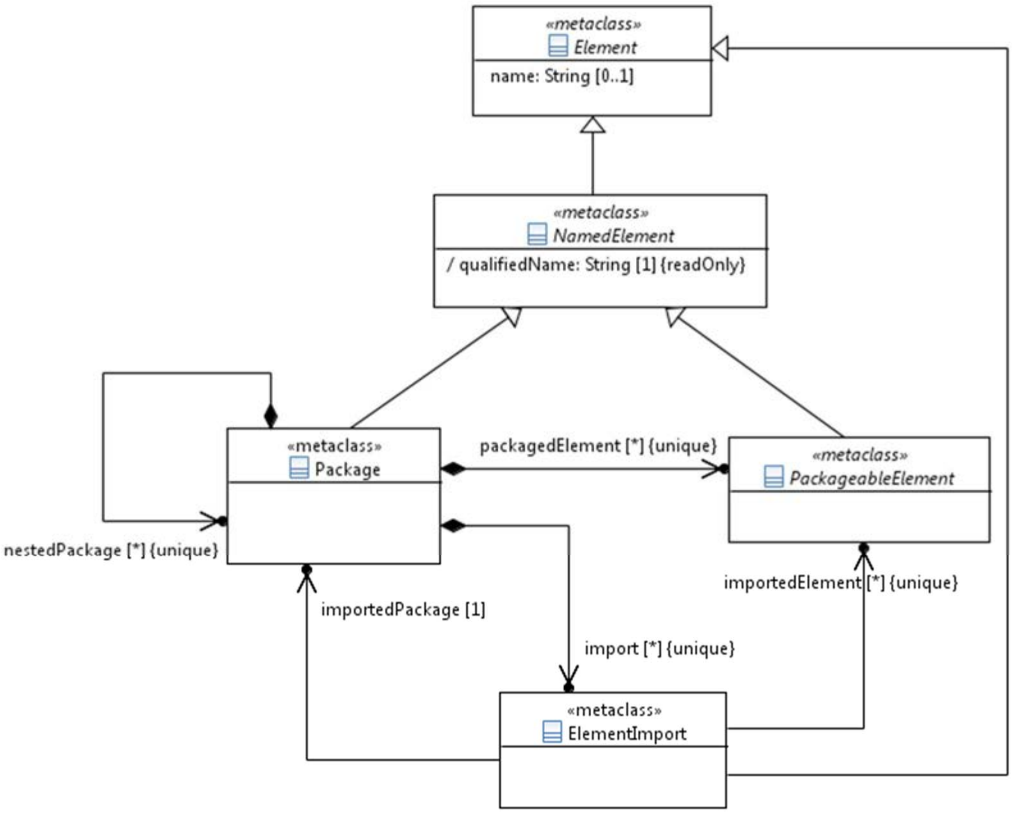

Figure 5.1: Foundational language concepts

Semantics

An 'Element' represents any constituent of a TDL model. It is the super-class of all other meta classes. It provides the ability to add comments and annotations. An 'Element' may contain any number of 'Comment's and 'Annotation's.

Generalization

There is no generalization specified.

Properties

- name: String [0..1]
  The name of the 'Element'. It may contain any character, including white-spaces. Having no name specified is different from an empty name (which is represented by an empty string).

- comment: Comment [0..*] {ordered, unique}
  The contained ordered set of 'Comment's attached to the 'Element'.

- annotation: Annotation [0..*] {ordered, unique}
  The contained ordered set of 'Annotation's attached to the 'Element'.

Constraints

There are no constraints specified.

5.2.2 NamedElement

Semantics

A 'NamedElement' represents any element of a TDL model that shall have a name and a qualified name.

The 'qualifiedName' is a compound name derived from the directly and all indirectly enclosing parent 'NamedElement's by concatenating the names of each 'NamedElement'. As a separator between the segments of a 'qualifiedName' the string '::' shall be used. The name of the root 'NamedElement' that (transitively) owns the 'NamedElement' shall always constitute the first segment of the 'qualifiedName'.

Generalization

- Element

Properties

- / qualifiedName: String [1] {readOnly}
  A derived property that represents the unique name of an element within a TDL model.

Constraints

- **Mandatory name**
  A 'NamedElement' shall have the 'name' property set and the 'name' shall be not an empty String.
  inv: **MandatoryName**:

  not self.name.oclIsUndefined() and self.name.size() > 0

- **Distinguishable qualified names**
  All qualified names of instances of 'NamedElement's shall be distinguishable within a TDL model.
  inv: **DistinquishableName**:

  NamedElement.allInstances()->one(e | e.qualifiedName = self.qualifiedName)

NOTE: It is up to the concrete syntax definition and tooling to resolve any name clashes between instances of the same meta-class in the qualified name.

5.2.3 PackageableElement

Semantics

A 'PackageableElement' denotes elements of a TDL model that may be contained in a 'Package'.

The visibility of a 'PackageableElement' is restricted to the 'Package' in which it is directly contained.
A 'PackageableElement' may be imported into other 'Package's by using 'ElementImport'. A 'PackageableElement' has no means to actively increase its visibility.

Generalization

- NamedElement

Properties

There are no properties specified.

Constraints

There are no constraints specified.

5.2.4 Package

Semantics

A 'Package' represents a container for 'PackageableElement's. A TDL model contains at least one 'Package', i.e. the root 'Package' of the TDL model. A 'Package' may contain any number of 'PackageableElement's, including other 'Package's.

A 'Package' constitutes a scope of visibility for its contained 'PackageableElement's. A 'PackageableElement' is only accessible within its owning 'Package' and within any 'Package' that directly imports it. 'PackageableElement's that are defined within a nested 'Package' are not visible from within its containing 'Package'.

A 'Package' may import any 'PackageableElement' from any other 'Package' by means of 'ElementImport'. By importing a 'PackageableElement', the imported 'PackageableElement' becomes visible and accessible within the importing 'Package'. Cyclic imports of packages are not permitted.

Generalization

- NamedElement

Properties

- packagedElement: PackageableElement [0..*] {unique}
  The set of 'PackageableElement's that are directly contained in the 'Package'.

- import: ElementImport [0..*] {unique}
  The contained set of import declarations.

- nestedPackage: Package [0..*] {unique}
  The contained set of 'Package's contained within this 'Package'.

Constraints

- **No cyclic imports**
  A 'Package' shall not import itself directly or indirectly.
  inv: **CyclicImports**:

  self.import->asOrderedSet()->closure(i | i.importedPackage.import)->forAll(i |

   i.importedPackage <> self)

5.2.5 ElementImport

Semantics

An 'ElementImport' allows importing 'PackageableElement's from arbitrary 'Package's into the scope of an importing 'Package'. By establishing an import, the imported 'PackageableElement's become accessible within the importing 'Package'.

Only those 'PackageableElement's that are directly contained in the exporting 'Package' may be imported via an 'ElementImport'. That is, the import of 'PackageableElement's is not transitive. After the import, all the imported elements become accessible within the importing 'Package'. The set of imported elements is declared via the 'importedElement' property.

If the set 'importedElement' is empty, it implies that all elements of the 'importedPackage' are imported.

Generalization

- Element

Properties

- importedPackage: Package [1]
  Reference to the 'Package' whose 'PackageableElement's are imported.

- importedElement: PackageableElement [0..*] {unique}
  A set of 'PackageableElement's that are imported into the context 'Package' via this 'ElementImport'.

Constraints

- **Consistency of imported elements**
  All imported 'PackageableElement's referenced by an 'ElementImport' shall be directly owned by the imported 'Package'.
  inv: **ConsistentImports**:

  self.importedElement->forAll(e | self.importedPackage.packagedElement->contains(e))

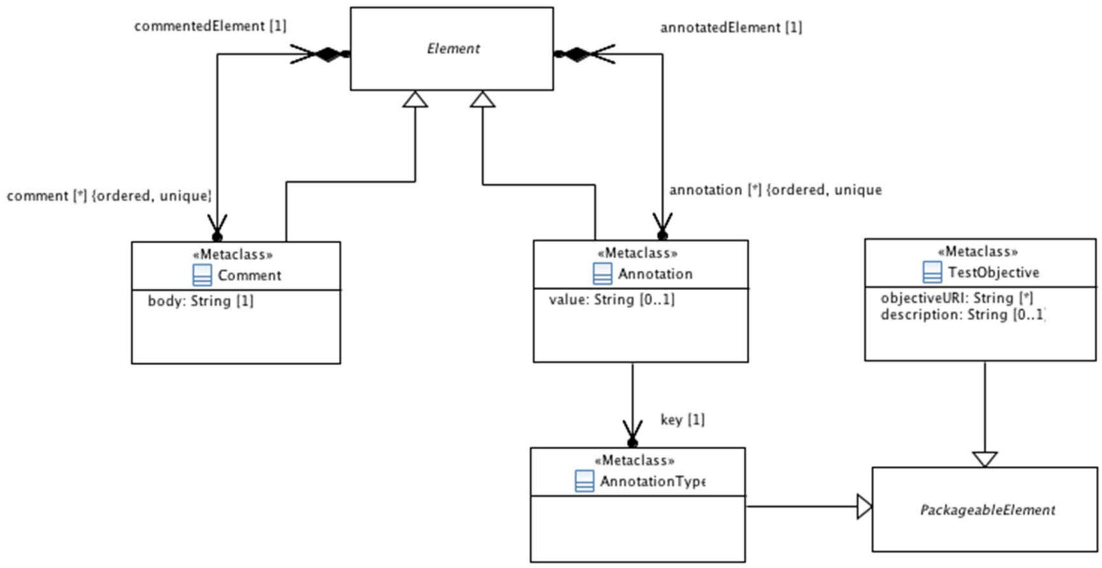

Figure 5.2: Miscellaneous elements

5.2.6 Comment

Semantics

'Comment's may be attached to 'Element's for documentation or for other informative purposes. Any 'Element', except for a 'Comment' or an 'Annotation', may contain any number of 'Comment's. The contents of 'Comment's shall not be used for adding additional semantics to elements of a TDL model.

Generalization

- Element

Properties

- commentedElement: Element [1]
  The 'Element' to which the 'Comment' is attached.

- body: String [1]
  The content of the 'Comment'.

Constraints

- **No nested comments**
  A 'Comment' shall not contain 'Comment's.
  inv: **CommentNestedComments**:

  self.comment->isEmpty()

- **No annotations to comments**
  A 'Comment' shall not contain 'Annotation's.
  inv: **CommentNestedAnnotations**:

  self.annotation->isEmpty()

5.2.7 Annotation

Semantics

An 'Annotation' is a means to attach user or tool specific semantics to any 'Element' of a TDL model, except to a 'Comment' and an 'Annotation' itself. An 'Annotation' represents a pair of a ('key', 'value') properties. Whereas the 'key' is mandatory for each 'Annotation', the 'value' might be left empty. This depends on the nature of the Annotation.

Generalization

- Element

Properties

- annotatedElement: Element [1]
  The 'Element' to which the 'Annotation' is attached.

- key: AnnotationType [1]
  Reference to the 'AnnotationType'.

- value: String [0..1]
  The 'value' mapped to the 'key'.

Constraints

- **No nested annotations**
  An 'Annotation' shall not contain 'Annotation's
  inv: **AnnotationNestedAnnotations**:

  self.annotation->isEmpty()

- **No comments to annotations**
  An 'Annotation' shall not contain 'Comment's.
  inv: **AnnotationNestedComments**:

  self.comment->isEmpty()

5.2.8 AnnotationType

Semantics

An 'AnnotationType' is used to define the 'key' of an 'Annotation'. It may represent any kind of user or tool specific semantics.

Generalization

- PackageableElement

Properties

There are no properties specified.

Constraints

There are no constraints specified.

5.2.9 TestObjective

Semantics

A 'TestObjective' specifies the reason for designing either a 'TestDescription' or a particular 'Behaviour' of a 'TestDescription'. A 'TestObjective' may contain a 'description' directly and/or refer to an external resource for further information about the objective.

The 'description' of a 'TestObjective' may be provided in natural language, or in a structured (i.e. machine-readable) format. The latter may be realized by means of the extension of TDL for the specification of structured test objectives defined in ETSI ES 203 119-4 [6].

Generalization

- PackageableElement

Properties

- description: String [0..1]
  A textual description of the 'TestObjective'.

- objectiveURI: String [0..*] {unique}
  A set of URIs locating resources that provide further information about the 'TestObjective'. These resources are typically external to a TDL model, e.g. part of requirements specifications or a dedicated test objective specification.

Constraints

There are no constraints specified.

6 Data

6.1 Overview

The 'Data' package describes all meta-model elements required to specify data and their use in a TDL model. It introduces the foundation for data types and data instances and distinguishes between simple data types and structured data types. The package also introduces parameters and variables and deals with the definition of actions and functions. It makes a clear separation between the definition of data types and data instances (clause 6.2) and their use in expressions (clause 6.3). The following main elements are described in this package:

- Elements to define data types and data instances, actions and functions, parameters and variables.

- Elements to make use of data elements in test descriptions, e.g. in guard conditions or data in interactions.

- Elements to allow the mapping of data elements (types, instances, actions, functions) to their concrete representations in an underlying runtime system.

For the purpose of defining the semantics of some data related meta-model elements, the semantical concept <undefined> is introduced denoting an undefined data value in a TDL model. The semantical concept <undefined> has no syntactical representation.

6.2 Data Definition - Abstract Syntax and Classifier Description

6.2.1 DataResourceMapping

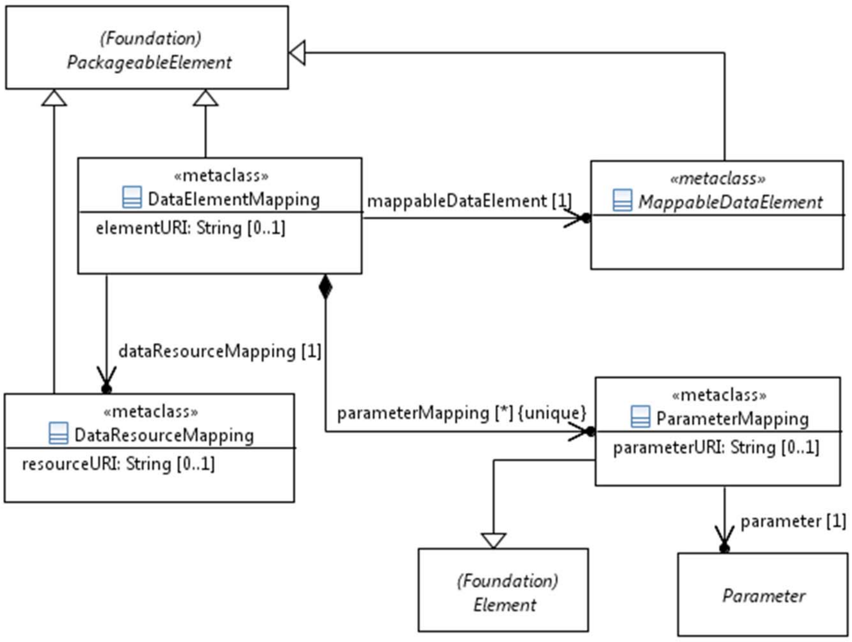

Figure 6.1: Data mapping concepts

Semantics

A 'DataResourceMapping' specifies a resource, in which the platform-specific representation of a 'DataType' or a 'DataInstance', i.e. their representation in a concrete data type system, is located as identified in the 'resourceURI' property. The 'DataResourceMapping' thus connects a TDL model with resources and artefacts that are outside of the scope of TDL.

Generalization

- PackageableElement

Properties

- resourceURI: String [0..1]
  Location of the resource that contains concrete data definitions. The location shall resolve to an unambiguous name.

Constraints

There are no constraints specified.

6.2.2 MappableDataElement

Semantics

A 'MappableDataElement' is the super-class of all data-related elements that may be mapped to a platform-specific representation by using a 'DataResourceMapping' and a 'DataElementMapping'. Each 'MappableDataElement' may be mapped to any number of concrete representations located in different resources. However the same 'MappableDataElement' shall not be mapped more than once to different concrete representations in the same 'DataResourceMapping'.

Generalization

- PackageableElement

Properties

There are no properties specified.

Constraints

There are no constraints specified.

6.2.3 DataElementMapping

Semantics

A 'DataElementMapping' specifies the location of a single concrete data definition within an externally identified resource (see clause 6.2.1). The location of the concrete data element within the external resource is described by means of the 'elementURI' property. A 'DataElementMapping' maps arbitrary data elements in a TDL model to their platform-specific counterparts.

If the 'DataElementMapping' refers to a 'StructuredDataType', an 'Action', or a 'Function', it is possible to map specific 'Members' (in the first case) or 'Parameters' (in the other cases) to concrete data representations explicitly.

Generalization

- PackageableElement

Properties

- elementURI: String [0..1]
  Location of a concrete data element within the resource referred in the referenced 'DataResourceMapping'. The location shall resolve to an unambiguous name within the resource.

- dataResourceMapping: DataResourceMapping [1]
  The 'DataResourceMapping' that specifies the URI of the external resource containing the concrete data element definitions.

- mappableDataElement: MappableDataElement [1]
  Refers to a 'MappableDataElement' that is mapped to its platform-specific counterpart identified in the 'elementURI'.

- parameterMapping: ParameterMapping [0..*] {unique}
  The set of 'Member's of a 'StructuredDataType' or 'FormalParameter's of an 'Action' or 'Function' that are mapped.

Constraints

- **Restricted use of 'ParameterMapping'**
  A set of 'ParameterMapping's may only be provided if 'mappableDataElement' refers to a 'StructuredDataType', an 'Action' or a 'Function' definition and the 'mappableDataElement' contains the mapped 'Parameters'.
  inv: **ParameterMappingType**:

  self.parameterMapping.size() = 0

  or (self.mappableDataElement.oclIsTypeOf(StructuredDataType)

  and self.parameterMapping->forAll(p |

  self.mappableDataElement.member->contains(p.parameter)))

  or (self.mappableDataElement.oclIsKindOf(Action)

  and self.parameterMapping->forAll(p |

  self.mappableDataElement.formalParameter->contains(p.parameter)))

6.2.4 ParameterMapping

Semantics

A 'ParameterMapping' is used to provide a mapping of 'Member's of a 'StructuredDataType' or 'FormalParameter's of an 'Action' or a 'Function'. It represents the location of a single concrete data element within the resource according to the 'DataResourceMapping', which the containing 'DataElementMapping' of the 'ParameterMapping' refers to. The location within the resource is described by means of the 'memberURI' property.

Generalization

- Element

Properties

- memberURI: String [0..1]
  Location of a concrete data element within the resource referred indirectly via the 'DataElementMapping' in the 'DataResourceMapping'. The location shall resolve to an unambiguous name within the resource.

- parameter: Parameter [1]
  Refers to the 'Parameter' ('Member' of a 'StructuredDataType' or 'FormalParameter' of an 'Action' or a 'Function') to be mapped to a concrete data representation.

Constraints

There are no constraints specified.

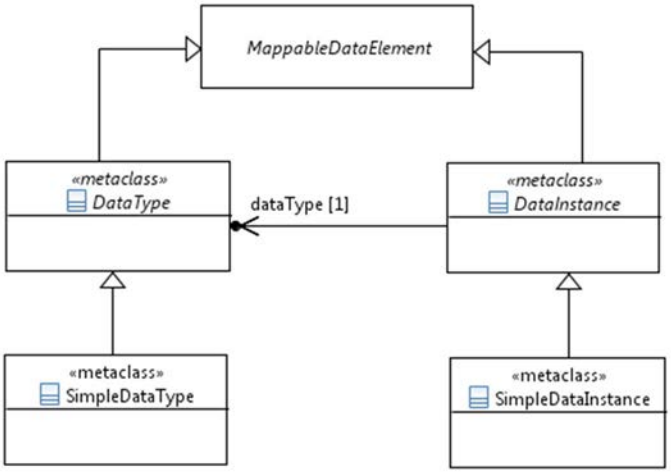

Figure 6.2: Basic data concepts and simple data

6.2.5 DataType

Semantics

A 'DataType' is the super-class of all type-related concepts. It is considered as abstract in several dimensions:

1) It is an abstract meta-class that is concretized by 'SimpleDataType' and 'StructuredDataType'.

2) It is abstract regarding its structure (simple or structured), semantics and operations that may be performed on on it. It, thus, shall be considered as an abstract data type (ADT).

3) It is abstract with respect to its manifestation in a concrete data type system.

A 'DataType' may be mapped to a concrete data type definition contained in a resource, which is external to the TDL model.

Generalization

- MappableDataElement

Properties

There are no properties specified.

Constraints

There are no constraints specified.

6.2.6 DataInstance

Semantics

A 'DataInstance' represents a symbolic value of a 'DataType'.

Generalization

- MappableDataElement

Properties

- dataType: DataType [1]
  Refers to the 'DataType', which this 'DataInstance' is a value of.

Constraints

There are no constraints specified.

6.2.7 SimpleDataType

Semantics

A 'SimpleDataType' represents a 'DataType' that has no internal structure. It resembles the semantics of ordinary primitive types from programming languages such as Integer or Boolean.

A set of predefined 'SimpleDataType's is provided by TDL by default (see clause 10.2).

Generalization

- DataType

Properties

There are no properties specified.

Constraints

There are no constraints specified.

6.2.8 SimpleDataInstance

Semantics

A 'SimpleDataInstance' represents a symbolic value of a 'SimpleDataType'. This symbolic value may denote either one specific value or a set of values in a concrete type system (the latter is similar to the notion of template in TTCN-3, see clause 15 in ETSI ES 201 873-1 [i.1]).

EXAMPLE: Assuming the 'SimpleDataType' Integer, 'SimpleDataInstance's of this type can be specified as Strings: "0", "1", "2", "max", "[-10..10]" etc. These symbolic values need to be mapped to concrete definitions of an underlying concrete type system to convey a specific meaning.

Generalization

- DataInstance

Properties

There are no properties specified.

Constraints

- **SimpleDataInstance shall refer to SimpleDataType**
  The inherited reference 'dataType' from 'DataInstance' shall refer to instances of 'SimpleDataType' solely.
  inv: **SimpleDataInstanceType**:

  self.dataType.oclIsKindOf(SimpleDataType)

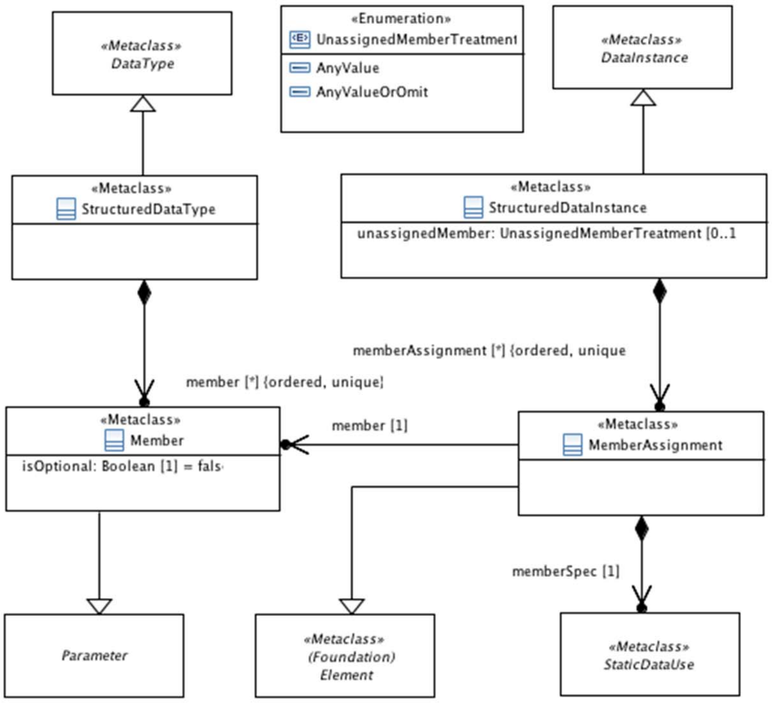

Figure 6.3: Structured data type and instance

6.2.9 StructuredDataType

Semantics

A 'StructuredDataType' represents a 'DataType' with an internal structure expressed by the concepts of 'Member's. It resembles the semantics of a complex data type in XML Schema, a record in TTCN-3 or a class in Java.

Generalization

- DataType

Properties

- member: Member [0..*] {ordered, unique}
  The contained ordered set of individual elements of the 'StructuredDataType'.

Constraints

- **Different member names in a structured data type**
  All 'Member' names of a 'StructuredDataType' shall be distinguishable.
  inv: **DistinguishableMemberNames**:

  self.container().member->one(e | e.name = self.name)

6.2.10 Member

Semantics

A 'Member' specifies a single constituent of the internal structure of a 'StructuredDataType'. It may be specified as an optional or a mandatory constituent. By default, all 'Member's of a 'StructuredDataType' are mandatory.

An optional member of a structured data type has an impact on the use of 'StructuredDataInstance's of this type (see clause 6.3.1).

Generalization

- Parameter

Properties

- isOptional: Boolean [1] = false
  If set to 'true' it indicates that the member is optional within the containing 'StructuredDataType'.

Constraints

There are no constraints specified.

6.2.11 StructuredDataInstance

Semantics

A 'StructuredDataInstance' represents a symbolic value of a 'StructuredDataType'. It contains 'MemberAssignment's for none, some or all 'Member's of the 'StructuredDataType'. This allows initializing the 'Member's with symbolic values.

If a 'StructuredDataInstance' has no 'MemberAssignment' for a given 'Member' of its 'StructuredDataType', it is assumed that the 'Member' has the value <undefined> assigned to it.

The optional 'unassignedMember' property may be used to override the semantics of unassigned 'Member's for the 'StructuredDataInstance'. If the 'unassignedMember' property is provided, then unassigned 'Member's shall be treated according to the semantics of the provided 'UnassignedMemberTreatment'. It is applied recursively.

Generalization

- DataInstance

Properties

- memberAssignment: MemberAssignment [0..*] {ordered, unique}
  Refers to the contained list of 'MemberAssignment's, which are used to assign values to 'Member's.

- unassignedMember: UnassignedMemberTreatment [0..1]
  Optional indication of how unassigned 'Members' shall be interpreted.

Constraints

- **StructuredDataInstance shall refer to StructuredDataType**
  The inherited reference 'dataType' from 'DataInstance' shall refer to instances of 'StructuredDataType' solely.
  inv: **StructuredDataInstance**:

  ```
  self.dataType.oclIsTypeOf(StructuredDataType)
  ```

- **'Member' of the 'StructuredDataType'**
  The referenced 'Member' shall be contained in the 'StructuredDataType' that the 'StructuredDataInstance', which contains this 'MemberAssignment', refers to.
  inv: **ExistingMemberOfDataType**:

  ```
  self.memberAssignment->forAll(a | self.dataType.member->contains(a.member))
  ```

6.2.12 MemberAssignment

Semantics

A 'MemberAssignment' specifies the assignment of a symbolic value to a 'Member' of a 'StructuredDataType'.

Generalization

- Element

Properties

- member: Member [1]
  Refers to the 'Member' of the 'StructuredDataType' definition that is referenced via the 'dataType' property of the 'StructuredDataInstance'.

- memberSpec: StaticDataUse [1]
  The contained 'StaticDataUse' specification for the referenced 'Member'. The symbolic value of this 'StaticDataUse' will be assigned to the 'Member'.

Constraints

- **Type of a 'memberSpec' and 'Member' shall coincide**
  The 'DataType' of the 'StaticDataUse' of 'memberSpec' shall coincide with the 'DataType' of the 'Member' of the 'MemberAssignment'.
  inv: **MatchingMemberDataType**:

  ```
  self.memberSpec.getDataType() = self.member.dataType
  ```

- **Restricted use of 'OmitValue' for optional 'Member's only**
  A non-optional 'Member' shall have a 'StaticDataUse' specification assigned to it that is different from 'OmitValue' and 'AnyValueOrOmit'.
  inv: **OmitValueUse**:

  ```
  (self.memberSpec.oclIsTypeOf(OmitValue) or self.memberSpec.oclIsTypeOf(AnyValueOrOmit))
  implies self.member.isOptional = true
  ```

- **Static data use in structured data**
  If the 'memberSpec' refers to a 'StructuredDataInstance', all of its 'ParameterBinding's shall refer to 'StaticDataUse'.
  inv: **StructuredDataInstanceUse**:

  ```
  self.memberSpec.argument->forAll(a |
    a.dataUse.oclIsKindOf(StaticDataUse))
  and self.memberSpec.argument->closure(a | a.dataUse.argument)->forAll(a |
      a.dataUse.oclIsKindOf(StaticDataUse))
  ```

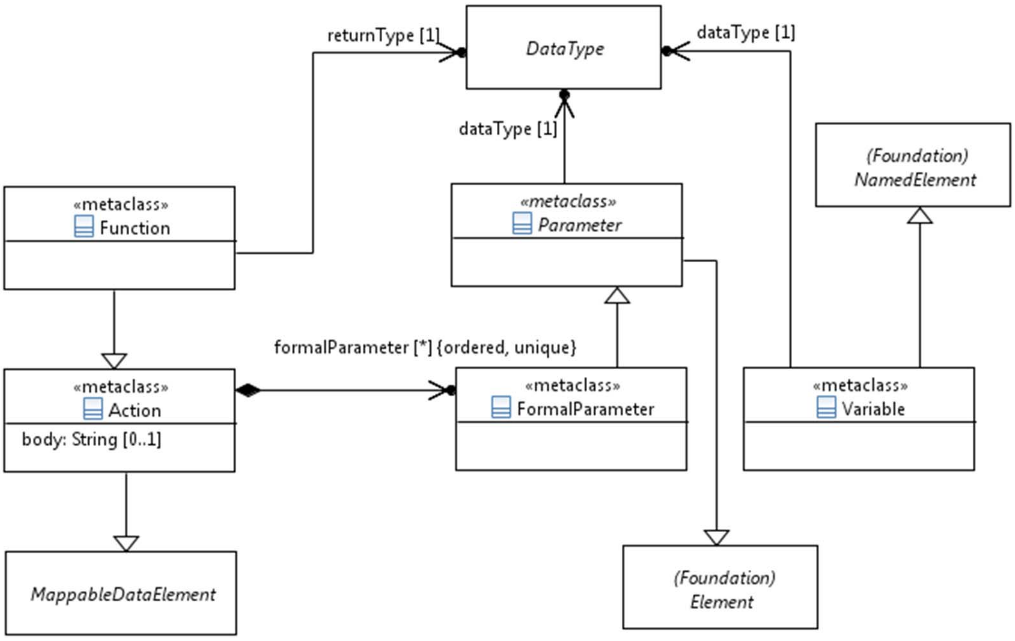

Figure 6.4: Action, function, parameter and variable

6.2.13 Parameter

Semantics

A 'Parameter' is used to define some common operations over 'FormalParameter' and 'Member' such as data mapping and assignments.

Generalization

- Element

Properties

- dataType: DataType [1]
  Refers to the 'DataType', which the 'Parameter' may be bound to.

Constraints

There are no constraints specified.

6.2.14 FormalParameter

Semantics

A 'FormalParameter' represents the concept of a formal parameter as known from programming languages.

Generalization

- Parameter

Properties

There are no properties specified.

Constraints

There are no constraints specified.

6.2.15 Variable

Semantics

A 'Variable' is used to denote a component-wide local variable. When it is defined, which occurs when the 'ComponentInstance' that is assumed to hold this variable is created (see clause 8.2.4), the 'Variable' has the value <undefined> assigned to it.

Generalization

- NamedElement

Properties

- dataType: DataType [1]
  Refers to the 'DataType' of 'DataInstance's, which the 'Variable' shall be bound to.

Constraints

There are no constraints specified.

6.2.16 Action

Semantics

An 'Action' is used to specify any procedure, e.g. a local computation, physical setup or manual task. The interpretation of the 'Action' is outside the scope of TDL. That is, its semantics is opaque to TDL. The implementation of an 'Action' may be provided by means of a 'DataElementMapping'.

An 'Action' may be parameterized. Actual parameters are provided in-kind. That is, executing an 'Action' does not change the values of the parameters provided; execution of an 'Action' is side-effect free.

Generalization

- MappableDataElement

Properties

- body: String [0..1]
  An informal, textual description of the 'Action' procedure.

- formalParameter: FormalParameter [0..*] {ordered, unique}
  The ordered set of contained 'FormalParameter's of this 'Action'.

Constraints

There are no constraints specified.

6.2.17 Function

Semantics

A 'Function' is a special kind of an 'Action' that has a return value. 'Function's are used to express calculations over 'DataInstance's within a 'TestDescription' at runtime. The execution of a 'Function' is side-effect free. That is, a 'Function' does not modify any passed or accessible 'DataInstance's or 'Variable's of the 'TestDescription'. The value of a 'Function' is defined only by its return value.

Generalization

- Action

Properties

- returnType: DataType [1]
  The 'DataType' of the 'DataInstance' that is returned when the 'Function' finished its calculation.

Constraints

There are no constraints specified.

6.2.18 UnassignedMemberTreatment

Semantics

'UnassignedMemberTreatment' shall be used in the definition or use of a 'StructuredDataInstance' in order to override how unassigned 'Members' shall be treated.

Generalization

There is no generalization specified.

Literals

- AnyValue
  Unassigned 'Members' shall be interpreted as 'AnyValue'.

- AnyValueOrOmit
  Unassigned 'Members' shall be interpreted as 'AnyValue'. Unassigned optional 'Members' shall be interpreted as 'AnyValueOrOmit'.

Constraints

There are no constraints specified.

6.3 Data Use - Abstract Syntax and Classifier Description

6.3.1 DataUse

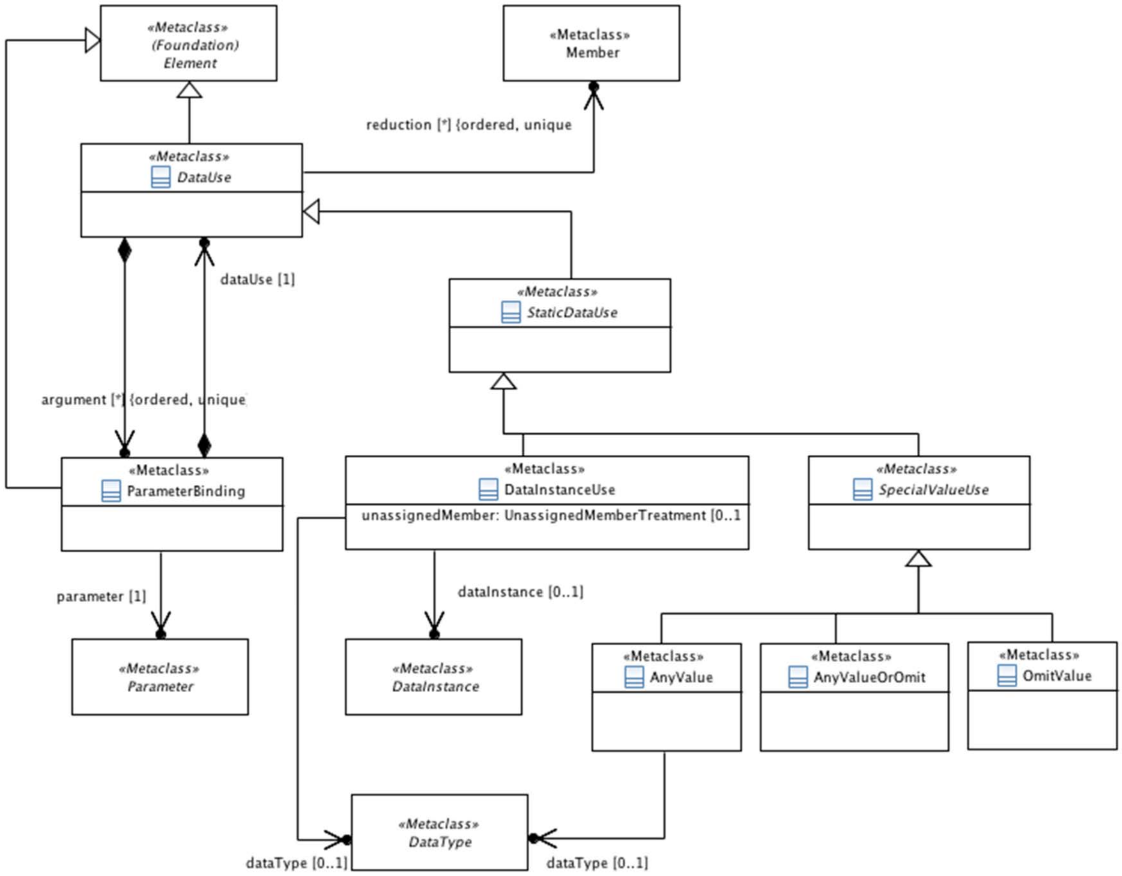

Figure 6.5: Data use concepts and static data use

Semantics

A 'DataUse' denotes an expression that evaluates to a 'DataInstance' of a given 'DataType'. Thus, a 'DataUse' delivers the symbolic value that may be used in assignments and invocations. Sub-classes of 'DataUse' are used in specific situations, e.g. to invoke a 'Function' or refer to a 'DataInstance'. The decision on what a 'DataUse' refers to is made by the concrete sub-classes. This is called the *context* of a 'DataUse'.

A 'DataUse' offers the capability to be parameterized. This is achieved by the use of a 'ParameterBinding'.

In case that the context of a 'DataUse' evaluates to a 'StructuredDataInstance', it is possible to specify a *location expression* over nested 'StructuredDataInstance's in order to reduce the 'DataUse' to the symbolic value contained in a potentially nested 'Member'. This is called *reduction*. The reduction is semantically equivalent to the dot-notation typically found in programming languages, e.g. in Java or TTCN-3, in order to navigate from a context object, i.e. the 'StructuredDataInstance', which this 'DataUse' evaluates to at runtime, to a specific location. The starting point of a location expression is the implicitly or explicitly referenced 'StructuredDataInstance' obtained after the 'DataUse' has been evaluated at runtime. The first element of the 'reduction' has to be a 'Member' of the context 'StructuredDataInstance'. In case that a 'Member' in the reduction list represents a 'SimpleDataType', no more 'Member's shall occur in the location expression after this 'Member'.

Generalization

- Element

Properties

- argument: ParameterBinding [0..*] {ordered, unique}
  The contained ordered set of 'ParameterBinding's that handles the assignment of symbolic values to 'Parameter's or 'Member's depending on the respective context of this 'DataUse'.

- reduction: Member [0..*] {ordered, unique}
  Location expression that refers to potentially nested 'Member's of a 'StructuredDataType'. Each 'Member' of the ordered set represents one fragment of the location expression. The location expression is evaluated after all 'argument' assignments have been put into effect.

Constraints

- **Occurrence of 'argument' and 'reduction'**
  Only in case of a ‘FunctionCall' both the 'argument' list and the ‘reduction' list may be provided, otherwise either the 'argument' list, the ’reduction' list, or none of them shall be provided.
  inv: **ArgumentReductionLists**:

  self.argument.isEmpty() or self.reduction.isEmpty() or self.oclIsTypeOf(FunctionCall)

- **Structured data types in 'reduction' set**
  A 'Member' at index *i* of a 'reduction' shall be contained in the 'StructuredDataType' of the 'Member' at index (*i* - 1) of that 'reduction'.
  inv: **ReductionMembers**:

  not self.getDataType().isKindOf(StructuredDataType)

  or self.reduction->isEmpty()

  or self.getDataType().member->contains(self.reduction->first())

  and self.reduction->select(m | self.reduction->indexOf(m) > 0)->forAll(m |

  self.reduction->at(self.reduction->indexOf(m)-1).dataType.isKindOf(StructuredDataType)

  and self.reduction->at(self.reduction->indexOf(m)-1).dataType.member->contains(m))

6.3.2 ParameterBinding

Semantics

A 'ParameterBinding' is used to assign a 'DataUse' specification to a 'FormalParameter' or a 'Member' of a 'StructuredDataType'.

If an 'OmitValue' is assigned to a non-optional 'Member' at runtime, the resulting semantics is kept undefined in TDL and needs to be resolved outside the scope of the present document.

NOTE: A typical treatment of the above case in an implementation would be to raise a runtime error.

Generalization

- Element

Properties

- dataUse: DataUse [1]
  Refers to the contained 'DataUse' specification whose symbolic value shall be assigned to the 'Parameter'.

- parameter: Parameter [1]
  Refers to the parameter, which gets the symbolic value of a 'DataUse' specification assigned to.

Constraints

- **Matching data type**
  The provided 'DataUse' shall match the 'DataType' of the referenced 'Parameter'.
  inv: **ParameterBindingTypes**:

  self.dataUse.getDataType() = self.parameter.dataType

- **Use of a 'StructuredDataInstance' with non-optional 'Member's**
  A non-optional 'Member' of a 'StructuredDataType' shall have a 'DataUse' specification assigned to it that is different from 'OmitValue' or 'AnyValueOrOmit'.
  inv: **OmitValueParameter**:

  self.parameter.oclIsTypeOf(Member) and self.parameter.oclAsType(Member).isOptional = false

  implies not self.dataUse.oclIsTypeOf(OmitValue) and not self.dataUse.oclIsTypeOf(AnyValueOrOmit)

6.3.3 StaticDataUse

Semantics

A 'StaticDataUse' specification denotes an expression that evaluates to a symbolic value that does not change during runtime, in other words, a constant.

Generalization

- DataUse

Properties

There are no properties specified.

Constraints

There are no constraints specified.

6.3.4 DataInstanceUse

Semantics

A 'DataInstanceUse' specifies a 'DataInstance' in a data usage context. It shall refer to a 'SimpleDataInstance', a 'StructuredDataInstance', or it shall provide 'ParameterBinding's as arguments in case no 'DataInstance' is referenced. An optional reference to a 'DataType' shall be provided if the 'DataInstanceUse' is used as the argument of 'Interaction' and no 'DataInstance' is provided.

In case it refers to a 'StructuredDataInstance', its value may be modified inline by providing arguments as 'ParameterBinding's. This allows replacing the current value of the referenced 'Member' with a new value evaluated from the provided 'DataUse' specification.

In case it does not refer to a 'DataInstance', a value for a 'StructuredDataInstance' of the 'DataType' inferred from the context in which it is used may be specified inline by providing arguments as 'ParameterBinding's. The 'DataType' of the 'DataInstance' is inferred from the 'DataType' of the 'Member', 'Parameter', 'FormalParameter', or 'Variable'.

If a referenced 'StructuredDataInstance' has no 'MemberAssignment' for a given 'Member' of its 'StructuredDataType', it is assumed that the 'Member' has the value <undefined> assigned to it. The optional 'unassignedMember' property may be used to override the semantics of unassigned 'Member's for the referenced 'StructuredDataInstance' in the usage context. If the 'unassignedMember' property is provided, then unassigned 'Member's shall be treated according to the semantics of the provided 'UnassignedMemberTreatment'. It is applied recursively. This also applies to inline specification of 'StructuredDataInstance's in case the 'DataInstanceUse' does not refer to a 'DataInstance'.

Generalization

- StaticDataUse

Properties

- dataInstance: DataInstance [0..1]
  Optional reference to a 'DataInstance' that is used in this 'DataUse' specification.

- dataType: DataType [0..1]
  Optional reference to a 'DataType' if the 'DataInstanceUse' is used as the argument of 'Interaction' and no 'DataInstance' is provided.

- unassignedMember: UnassignedMemberTreatment [0..1]
  Optional indication of how unassigned 'Members' shall be interpreted.

Constraints

- **'DataInstance' reference or non-empty 'argument'**
  Either a 'dataInstance' or a non-empty 'argument' set shall be specified.
  inv: **DataInstanceOrArguments**:

  not self.dataInstance.oclIsUndefined() or not self.argument->isEmpty()

- **'DataType' provided only in 'Interaction' with no 'DataInstance' reference**
  The 'dataType' property shall be provided only if the 'DataInstanceUse' is directly contained in an 'Interaction' and no 'DataInstance' is referenced.
  inv: **DataTypeInInteraction**:

  (self.container().oclIsTypeOf(Interaction)

  and self.dataInstance.oclIsUndefined()

  and not self.dataType.oclIsUndefined())

  or (self.container().oclIsTypeOf(Interaction)

  and self.dataType.oclIsUndefined()

  and not self.dataInstance.oclIsUndefined())

  or (self.dataType.oclIsUndefined()

  and not self.container().oclIsTypeOf(Interaction))

6.3.5 SpecialValueUse

Semantics

A 'SpecialValueUse' is the super-class of all 'StaticDataUse' specifications that represent predefined wildcards instead of values.

Generalization

- StaticDataUse

Properties

There are no properties specified.

Constraints

- **Empty 'argument' and 'reduction' sets**
  The 'argument' and 'reduction' sets shall be empty.
  inv: **SpecialValueArgumentReduction**:

  self.reduction->isEmpty() and self.argument->isEmpty()

6.3.6 AnyValue

Semantics

An 'AnyValue' denotes an unknown symbolic value from the set of all possible values of 'DataType's which are compatible in the context in which 'AnyValue' is used. The set of all possible values is not restricted to values explicitly specified as 'DataInstance's in a given TDL model. It excludes the 'OmitValue' and the <undefined> value.

Its purpose is to be used as a placeholder in the specification of a data value when the actual value is not known or irrelevant. When used in certain contexts, such as 'MemberAssignment', there is only one 'DataType' for the set of possible values, which shall be inferred from the context. When 'AnyValue' is used directly as an argument of an 'Interaction', under certain circumstances there may be multiple 'DataType's that are compatible in the context. In this case, a 'DataType' may be specified explicitly to restrict the acceptable 'DataInstance's to the ones of the specified 'DataType' only. Otherwise, 'AnyValue' is a placeholder for the 'DataInstance's of any of the compatible 'DataType's.

Generalization

- SpecialValueUse

Properties

- dataType: DataType [0..1]
  Refers to the optionally declared 'DataType' of the 'AnyValue'.

Constraints

There are no constraints specified.

6.3.7 AnyValueOrOmit

Semantics

An 'AnyValueOrOmit' denotes an unknown symbolic value from the union set of 'AnyValue' and 'OmitValue'.

Its purpose is to be used as a placeholder in the specification of a data value when the actual value is not known or irrelevant.

NOTE: 'AnyValueOrOmit' is semantically equivalent to 'AnyValue' if applied on mandatory 'Member's of a 'StructuredDataType'.

Generalization

- SpecialValueUse

Properties

There are no properties specified.

Constraints

There are no constraints specified.

6.3.8 OmitValue

Semantics

An 'OmitValue' denotes a symbolic value indicating that a concrete value is not transmitted in an 'Interaction' at runtime. Outside an 'Interaction' it carries no specific meaning.

NOTE: The typical use of an 'OmitValue' is its assignment to an optional 'Member' that is part of a 'StructuredDataType' definition.

Generalization

- SpecialValueUse

Properties

There are no properties specified.

Constraints

There are no constraints specified.

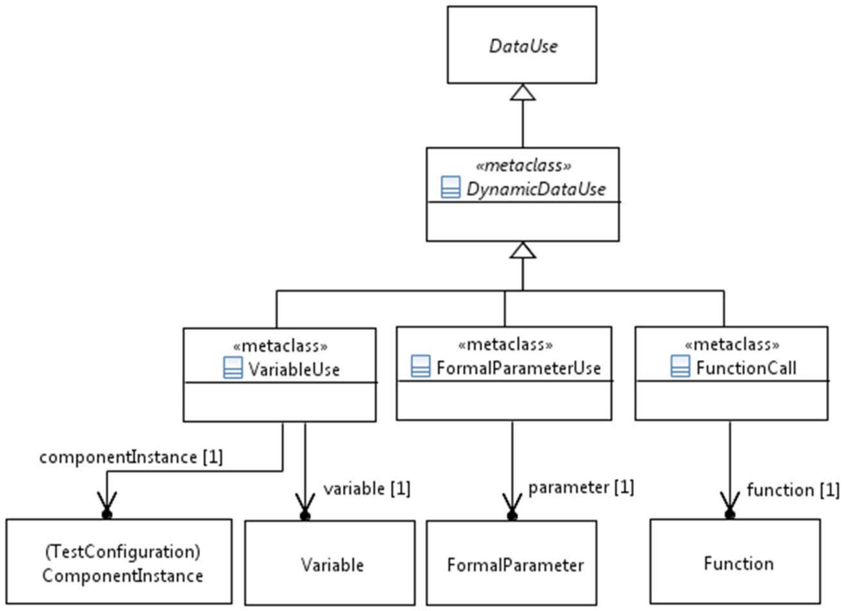

Figure 6.6: Dynamic data use

6.3.9 DynamicDataUse

Semantics

A 'DynamicDataUse' is the super-class for all symbolic values that are evaluated at runtime.

Generalization

- DataUse

Properties

There are no properties specified.

Constraints

There are no constraints specified.

6.3.10 FunctionCall

Semantics

A 'FunctionCall' specifies the invocation of a 'Function' with its arguments.

If the invoked 'Function' has declared 'FormalParameter's the corresponding arguments shall be specified by using 'ParameterBinding'.

If a 'reduction' is provided, it applies to the return value of the 'Function', which implies that the return value is of 'StructuredDataType'.

Generalization

- DynamicDataUse

Properties

- function: Function [1]
  Refers to the function being invoked.

Constraints

- **Matching parameters**
  All 'FormalParameter's of the invoked 'Function' shall be bound.
  inv: **FunctionCallParameters**:

  self.function.formalParameter->forAll(p | self.argument->exists(a | a.parameter = p))

6.3.11 FormalParameterUse

Semantics

A 'FormalParameterUse' specifies the access of a symbolic value stored in a 'FormalParameter' of a 'TestDescription'.

Generalization

- DynamicDataUse

Properties

- parameter: FormalParameter [1]
  Refers to the 'FormalParameter' of the containing 'TestDescription' being used.

Constraints

There are no constraints specified.

6.3.12 VariableUse

Semantics

A 'VariableUse' denotes the use of the symbolic value stored in a 'Variable'.

Generalization

- DynamicDataUse

Properties

- variable: Variable [1]
  Refers to the 'Variable', whose symbolic value shall be retrieved.

- componentInstance: ComponentInstance [1]
  Refers to the 'ComponentInstance' that references the 'Variable' via its 'ComponentType'.

Constraints

- **Local variables of tester components only**
  All variables used in a 'DataUse' specification via a 'VariableUse' shall be local to the same 'componentInstance' and the 'componentInstance' shall be in the role 'Tester'.
  inv: **VariableUseComponentRole**:

  self.componentInstance.type.variable->contains(self.variable)

  and self.componentInstance.role.name = 'Tester'

7 Time

7.1 Overview

The 'Time' package defines the elements to express time, time constraints, timers and operations over time and timers.

7.2 Abstract Syntax and Classifier Description

7.2.1 Time

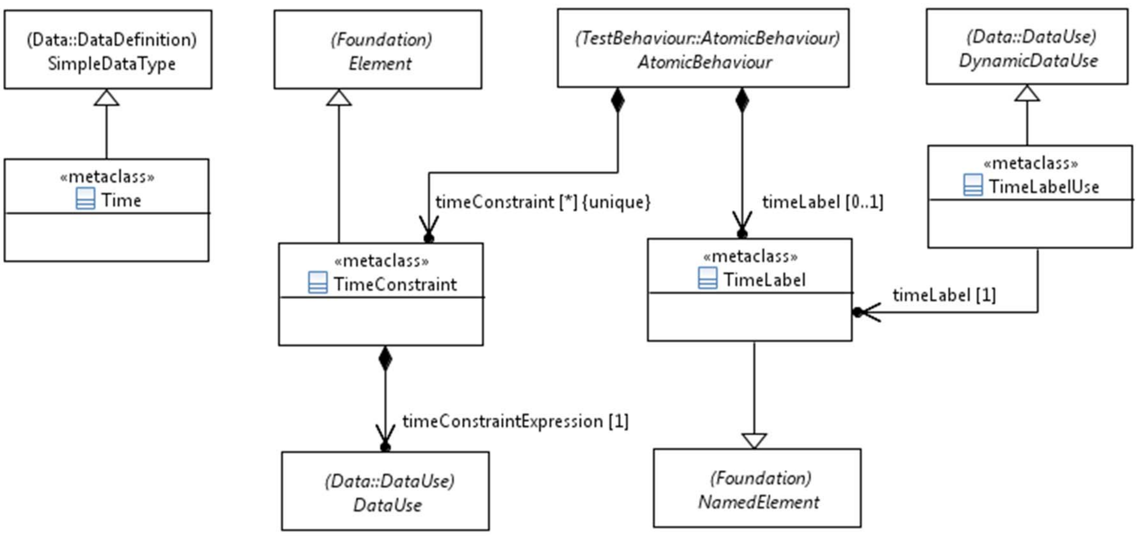

Figure 7.1: Time, time label and time constraint

Semantics

A 'Time' element extends the 'SimpleDataType' and is used to measure time and helps expressing time-related concepts in a TDL model.

Time in TDL is considered to be global and progresses in discrete quantities of arbitrary granularity. Time starts with the execution of the first 'TestDescription' being invoked. Progress in time is expressed as a monotonically increasing function, which is outside the scope of TDL.

A time value is expressed as a 'SimpleDataInstance' of an associated 'Time' 'SimpleDataType'. The way how a time value is represented, e.g. as an integer or a real number, is kept undefined in TDL and may be defined by the user via a 'DataElementMapping'.

The 'name' property of the 'Time' element expresses the granularity of time measurements. TDL defines the predefined instance 'Second' of the 'Time' data type, which measures the time in the physical unit seconds. See clause 10.4.

NOTE: When designing a concrete syntax from the TDL meta-model, it is recommended that the 'Time' data type can be instantiated at most once by a user and the same 'Time' instance is used in all 'DataUse' expressions within a TDL model; let it be the predefined instance 'Second' or a user-defined instance. This assures a consistent use of time-related concepts throughout the TDL model.

Generalization

- SimpleDataType

Properties

There are no properties specified.

Constraints

There are no constraints specified.

7.2.2 TimeLabel

Semantics

A 'TimeLabel' is a symbolic name attached to an 'AtomicBehaviour' that represents an ordered list of timestamps of execution of this atomic behaviour. A 'TimeLabel' allows the expression of time constraints (see subsequent clauses). It is contained in the 'AtomicBehaviour' that produces the timestamps at runtime.

If the atomic behaviour the 'TimeLabel' is attached to is executed once, the 'TimeLabel' contains only a single timestamp. Otherwise, if the atomic behaviour is executed iteratively, e.g. within a loop, the 'TimeLabel' represents a list of timestamps. In the latter case, some functions are predefined that return a single timestamp from this list (see clause 10.5.3). To enable the definition of these functions, it is assumed that all 'TimeLabel's belong to the predefined data type 'TimeLabelType' (see clause 10.2.3).

There is no assumption being made when the timestamp is taken: at the start or the end of the 'AtomicBehaviour' or at any other point during its execution. It is however recommended to have it consistently defined in an implementation of the TDL model.

Generalization

- NamedElement

Properties

There are no properties specified.

Constraints

There are no constraints specified.

7.2.3 TimeLabelUse

Semantics

A 'TimeLabelUse' enables the use of a time label in a 'DataUse' specification. The most frequent use of that will be within a 'TimeConstraint' expression.

Generalization

- DynamicDataUse

Properties

- timeLabel: TimeLabel [1]
  Refers to the time label being used in the 'DataUse' specification.

Constraints

There are no constraints specified.

7.2.4 TimeConstraint

Semantics

A 'TimeConstraint' is used to express a time requirement for an 'AtomicBehaviour'. The 'TimeConstraint' is usually formulated over one or more 'TimeLabel's. A 'TimeConstraint' constrains the execution time of the 'AtomicBehaviour' that contains this 'TimeConstraint'.

If the 'AtomicBehaviour' is a *tester-input event*, the 'TimeConstraint' is evaluated after this 'AtomicBehaviour' happened. If it evaluates to Boolean 'true' it implies a 'pass' test verdict; otherwise a 'fail' test verdict. In other cases of 'AtomicBehaviour', the 'TimeConstraint' is evaluated before its execution. Execution is blocked and keeps blocking until the 'TimeConstraint' evaluates to Boolean 'true'.

Generalization

- Element

Properties

- timeConstraintExpression: DataUse [1]
  Defines the time constraint over 'TimeLabel's as an expression of predefined type 'Boolean'.

Constraints

- **Time constraint expression of type Boolean**
  The expression given in the 'DataUse' specification shall evaluate to predefined type 'Boolean'.
  inv: **TimeConstraintType**:

  self.timeConstraintExpression.getDataType().name = 'Boolean'

- **Use of local variables only**
  The expression given in the 'DataUse' specification shall contain only 'Variable's that are local to the 'AtomicBehaviour' that contains this time constraint. That is, all 'Variable's shall be referenced in the 'ComponentInstance' that executes the 'AtomicBehaviour'.
  inv: **TimeConstraintVariables**:

```
(not self.timeConstraintExpression.oclIsKindOf(VariableUse)
 or (self.container().oclIsKindOf(Interaction)
 and (self.container().sourceGate.component = self.timeConstraintExpression.componentInstance
   or self.container().target->forAll(t |
        t.targetGate.component = self.timeConstraintExpression.componentInstance)))
 or (self.container().oclIsKindOf(ActionBehaviour)
 and not self.container().componentInstance.oclIsUndefined()
 and self.container().componentInstance = self.timeConstraintExpression.componentInstance))


and self.timeConstraintExpression.argument->forAll(a |
     not a.dataUse.oclIsKindOf(VariableUse)
     or (self.container().oclIsKindOf(Interaction)
     and (self.container().sourceGate.component = a.dataUse.componentInstance
       or self.container().target->forAll(t |
            t.targetGate.component = a.dataUse.componentInstance)))
     or (self.container().oclIsKindOf(ActionBehaviour)
     and not self.container().componentInstance.oclIsUndefined()
     and self.container().componentInstance = self.timeConstraintExpression.componentInstance))


and self.timeConstraintExpression.argument->closure(a | a.dataUse.argument)->forAll(a |
     not a.dataUse.oclIsKindOf(VariableUse)
     or (self.container().oclIsKindOf(Interaction)
     and (self.container().sourceGate.component = a.dataUse.componentInstance
       or self.container().target->forAll(t |
            t.targetGate.component = a.dataUse.componentInstance)))
     or (self.container().oclIsKindOf(ActionBehaviour)
     and not self.container().componentInstance.oclIsUndefined()
     and self.container().componentInstance = self.timeConstraintExpression.componentInstance))
```

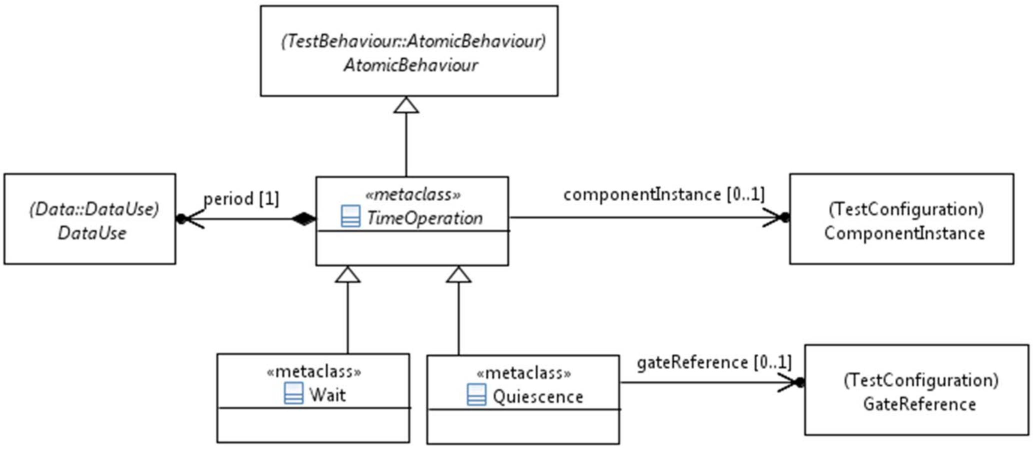

Figure 7.2: Time operations

7.2.5 TimeOperation

Semantics

A 'TimeOperation' summarizes the two possible time operations that may occur at a 'Tester' 'ComponentInstance': 'Wait' and 'Quiescence'.

Generalization

- AtomicBehaviour

Properties

- period: DataUse [1]
  The 'period' defines the time duration of the 'TimeOperation'.

- componentInstance: ComponentInstance [0..1]
  The 'ComponentInstance', to which the 'TimeOperation' is associated.

Constraints

- **Time operations on tester components only**
  A 'TimeOperation' shall be performed only on a 'ComponentInstance' in the role 'Tester'.
  inv: **TimeOperationComponentRole**:

  (not self.componentInstance.oclIsUndefined()

  and self.componentInstance.role.name = 'Tester')

  or (self.oclIsTypeOf(Quiescence)

  and not self.gateReference.oclIsUndefined()

  and self.gateReference.component.role.name = 'Tester')

- **'Time' data type for period expression**
  The 'DataUse' expression assigned to the 'period' shall evaluate to a data instance of the 'Time' data type.
  inv: **TimePeriodType**:

  self.period.getDataType().oclIsKindOf(Time)

7.2.6 Wait

Semantics

A 'Wait' defines the time duration that a 'Tester' component instance waits before performing the next behaviour.

Any input arriving at the 'Tester' component during 'Wait' at runtime is handled by the following behaviour and is not a violation of the test description. The specific mechanism of implementing 'Wait' is not specified.

NOTE: 'Wait' is implemented typically by means of a timer started with the given 'period' property. After the timeout, the 'Tester' component continues executing the next behaviour.

Generalization

- TimeOperation

Properties

There are no properties specified.

Constraints

There are no constraints specified.

7.2.7 Quiescence

Semantics

A 'Quiescence' is called a *tester-input event* and defines the time duration, during which a 'Tester' component shall expect no input from a 'SUT' component at a given gate reference (if 'Quiescence' is associated to a gate reference) or at all the gate references the 'Tester' component instance contains of (if 'Quiescence' is associated to a component instance).

When a 'Quiescence' is executed, the 'Tester' component listens to 'Interaction's that occur at the defined gate reference(s). If such an 'Interaction' occurs during the defined 'period' (time duration), the test verdict is set to 'fail'; otherwise to 'pass'.

Input arriving during 'Quiescence' that matches an 'Interaction' of an alternative block in 'AlternativeBehaviour' or 'ExceptionalBehaviour' is allowed and not a violation of the test description. A similar statement holds for the use of 'Quiescence' in 'ParallelBehaviour'.

If 'Quiescence' occurs as the first behaviour element in an alternative block of an 'AlternativeBehaviour' or 'ExceptionalBehaviour', then its behaviour is defined as follows. The measurement of the quiescence duration starts with the execution of the associated alternative or exceptional behaviour. The check for the absence of an 'Interaction' occurs only if none of the alternative blocks have been selected.

NOTE: 'Quiescence' is implemented typically by means of a timer with the given 'period' property and listening at the indicated gate reference(s). The occurrence of the timeout indicates the end of a 'Quiescence' with verdict 'pass'.

Generalization

- TimeOperation

Properties

- gateReference: GateReference [0..1]
  The 'GateReference', to which the 'Quiescence' is associated.

Constraints

- **Exclusive use of gate reference or component instance**
  If a 'GateReference' is provided, a 'ComponentInstance' shall be not provided and vice versa.
  inv: **QuiescenceTarget**:

  self.gateReference.oclIsUndefined() or self.componentInstance.oclIsUndefined()

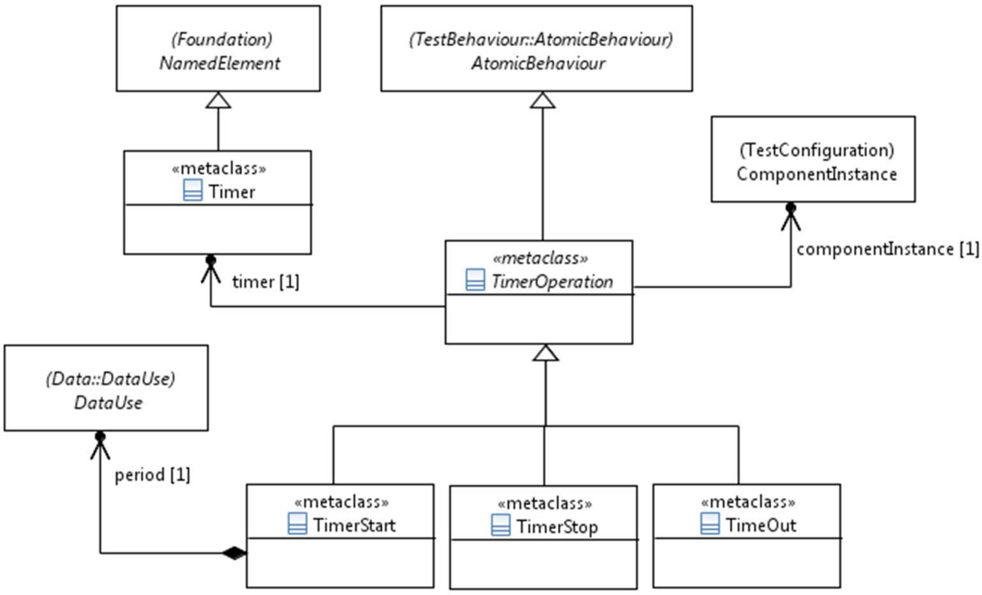

Figure 7.3: Timer and timer operations

7.2.8 Timer

Semantics

A 'Timer' defines a timer that is used to measure time intervals. A 'Timer' is contained within a 'ComponentType' assuming that each 'ComponentInstance' of the given 'ComponentType' has its own local copy of that timer at runtime. Each 'Timer' is initialized as *idle* at runtime.

Generalization

- NamedElement

Properties

There are no properties specified.

Constraints

There are no constraints specified.

7.2.9 TimerOperation

Semantics

A 'TimerOperation' operates on an associated 'Timer'. It is an element that summarizes the operations on timers: timer start, timeout and timer stop.

Generalization

- AtomicBehaviour

Properties

- timer: Timer [1]
  This property refers to the 'Timer' on which the 'TimerOperation' operates.

- componentInstance: ComponentInstance [1]
  The 'ComponentInstance', to which the 'TimerOperation' is associated.

Constraints

- **Timer operations on tester components only**
  A 'TimerOperation' shall be performed only on a 'ComponentInstance' in the role 'Tester'.
  inv: **TimerOperationComponentRole**:

  self.componentInstance.role.name = 'Tester'

7.2.10 TimerStart

Semantics

A 'TimerStart' operation starts a specific timer and the state of that timer becomes *running*. If a running timer is started, the timer is stopped implicitly and then (re-)started.

Generalization

- TimerOperation

Properties

- period: DataUse [1]
  Defines the duration of the timer from start to timeout.

Constraints

- **'Time' data type for period expression**
  The 'DataUse' expression assigned to the 'period' shall evaluate to a data instance of the 'Time' data type.
  inv: **TimerPeriodType**:

  self.period.getDataType().oclIsKindOf(Time)

7.2.11 TimerStop

Semantics

A 'TimerStop' operation stops a running timer. If an idle timer is stopped, then no action shall be taken. After performing a 'TimerStop' operation on a running timer, the state of that timer becomes *idle*.

Generalization

- TimerOperation

Properties

There are no properties specified.

Constraints

There are no constraints specified.

7.2.12 TimeOut

Semantics

A 'TimeOut' is called a *tester-input event* and is used to specify the occurrence of a timeout event when the period set by the 'TimerStart' operation has elapsed. At runtime, the timer changes from *running* state to *idle* state.

Generalization

- TimerOperation

Properties

There are no properties specified.

Constraints

There are no constraints specified.

8 Test Configuration

8.1 Overview

The 'Test Configuration' package describes the elements needed to define a 'TestConfiguration' consisting of tester and SUT components, gates, and their interconnections represented as 'Connection's. A 'TestConfiguration' specifies the structural foundations on which test descriptions may be built upon. The fundamental units of a 'TestConfiguration' are the 'ComponentInstance's. Each 'ComponentInstance' specifies a functional entity of the test system. A 'ComponentInstance' may either be a (part of a) tester or a (part of an) SUT. That is, both the tester and the SUT may be decomposed, if required. The communication exchange between 'ComponentInstance's is established through interconnected 'GateInstance's via 'Connection's and 'GateReference's. To offer reusability, TDL introduces 'ComponentType's and 'GateType's.

8.2 Abstract Syntax and Classifier Description

8.2.1 GateType

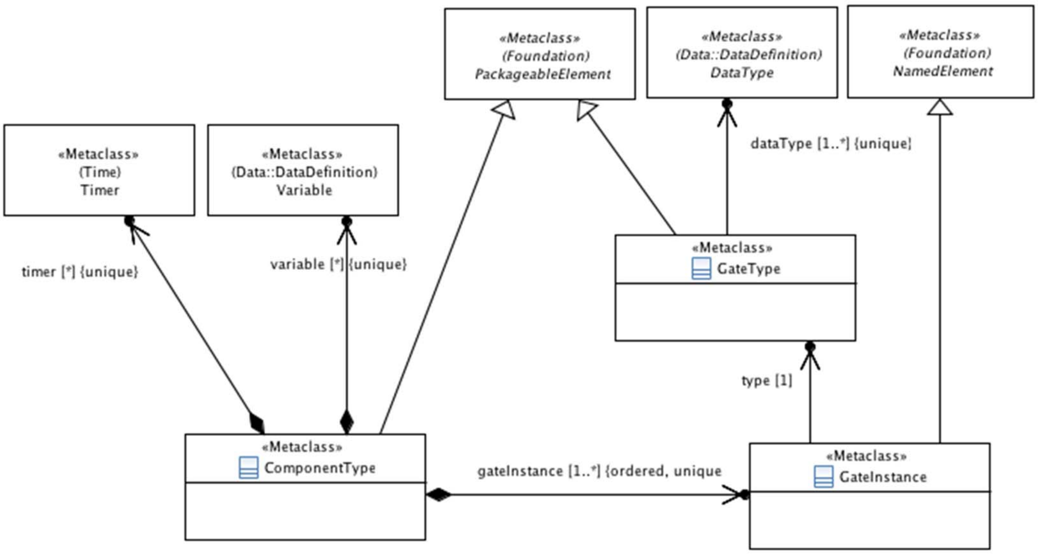

Figure 8.1: Component and gate type

Semantics

A 'GateType' represents a type of communication points, called 'GateInstance's, for exchanging information between 'ComponentInstance's. A 'GateType' specifies the 'DataType's that can be exchanged via 'GateInstance's of this type in both directions.

Generalization

- PackageableElement

Properties

- dataType: DataType [1..*] {unique}
  The 'DataType's that can be exchanged via 'GateInstance's of that 'GateType'. The arguments of 'Interactions' shall adhere to the 'DataType's that are allowed to be exchanged.

Constraints

There are no constraints specified.

8.2.2 GateInstance

Semantics

A 'GateInstance' represents an instance of a 'GateType'. It is the means to exchange information between connected 'ComponentInstance's. A 'GateInstance' is contained in a 'ComponentType'.

Generalization

- NamedElement

Properties

- type: GateType [1]
  The 'GateType' of the 'GateInstance'.

Constraints

There are no constraints specified.

8.2.3 ComponentType

Semantics

A 'ComponentType' specifies the type of one or several functional entities, called 'ComponentInstance's, that participate in a 'TestConfiguration'. A 'ComponentType' contains at least one 'GateInstance' and may contain any number of 'Timer's and 'Variable's.

Generalization

- PackageableElement

Properties

- gateInstance: GateInstance [1..*] {ordered, unique}The 'GateInstance's used by 'ComponentInstance's of that 'ComponentType'.
- timer: Timer [0..*] {unique}
  The 'Timer's owned by the 'ComponentType'.
- variable: Variable [0..*] {unique}
  The 'Variable's owned by the 'ComponentType'.

Constraints

There are no constraints specified.

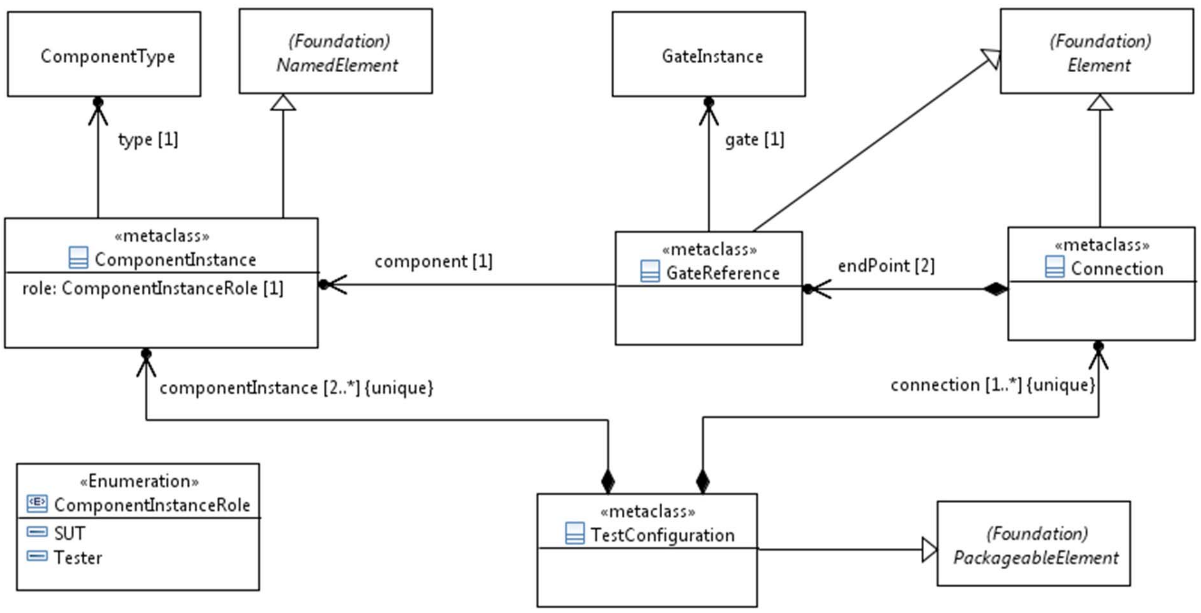

Figure 8.2: Test configuration

8.2.4 ComponentInstance

Semantics

A 'ComponentInstance' represents an active, functional entity of the 'TestConfiguration', which contains it. Its main purpose is to exchange information with other connected components via 'Interaction's. It acts either in the role of a 'Tester' or an 'SUT' component.

A 'ComponentInstance' derives the 'GateInstance's, 'Timer's, and 'Variable's from its 'ComponentType' for use within a 'TestDescription'. However, component-internal 'Timer's and 'Variable's shall be only used in 'TestDescription's if the role of the component is of 'Tester'. When a 'ComponentInstance' is created, a 'Timer' shall be in the *idle* state (see clause 7.2.8) and a 'Variable' shall have the value <undefined> (see clause 6.2.15).

Generalization

- NamedElement

Properties

- type: ComponentType [1]
  The 'ComponentType' of this 'ComponentInstance'.

- role: ComponentInstanceRole [1]
  The role that the 'ComponentInstance' plays within the 'TestConfiguration'. It can be either 'Tester' or 'SUT'.

Constraints

There are no constraints specified.

8.2.5 ComponentInstanceRole

Semantics

'ComponentInstanceRole' specifies the role of a 'ComponentInstance', whether it acts as a 'Tester' or as an 'SUT' component.

Generalization

There is no generalization specified.

Literals

- SUT
  The 'ComponentInstance' assumes the role 'SUT' in the enclosing 'TestConfiguration'.

- Tester
  The 'ComponentInstance' assumes the role 'Tester' in the enclosing 'TestConfiguration'.

Constraints

There are no constraints specified.

8.2.6 GateReference

Semantics

A 'GateReference' is an endpoint of a 'Connection', which it contains. It allows the specification of a connection between two 'GateInstance's of different component instances in unique manner (because 'GateInstance's are shared between all 'ComponentInstance's of the same 'ComponentType').

Generalization

- Element

Properties

- component: ComponentInstance [1]
  The 'ComponentInstance' that this 'GateReference' refers to.

- gate: GateInstance [1]
  The 'GateInstance' that this 'GateReference' refers to.

Constraints

- **Gate instance of the referred component instance**
  The referred 'GateInstance' shall be contained in the 'ComponentType' of the referred 'ComponentInstance'.
  inv: **GateInstanceReference**:

  self.component.type.gateInstance->contains(self.gate)

8.2.7 Connection

Semantics

A 'Connection' defines a communication channel for exchanging information between 'ComponentInstance's via 'GateReference's. It does not specify or restrict the nature of the communication channel that is eventually used in an implementation. For example, a 'Connection' could refer to an asynchronous communication channel for the exchange of messages or it could rather refer to a programming interface that enables the invocation of functions.

A 'Connection' is always bidirectional and point-to-point, which is assured by defining exactly two endpoints, given as 'GateReference's. A 'Connection' can be established between any two different 'GateReference's acting as 'endPoint' of this connection. That is, self-loop 'Connection's that start and end at the same 'endPoint' are not permitted.

A 'Connection' can be part of a point-to-multipoint communication relation. In this case, the same pair of 'GateInstance'/'ComponentInstance' occurs multiple times in different 'Connection's. However, multiple connections between the same two pairs of 'GateInstance'/'ComponentInstance' are not permitted in a 'TestConfiguration' (see clause 8.2.8).

Generalization

- Element

Properties

- endPoint: GateReference [2]
  The two 'GateReference's that form the endpoints of this 'Connection'.

Constraints

- **Self-loop connections are not permitted**
  The 'endPoint's of a 'Connection' shall not be the same. Two endpoints are the same if both, the referred 'ComponentInstance's and the referred 'GateInstance's, are identical.
  inv: **NoSelfLoop**:

  self.endPoint->forAll(e1 | self.endPoint->one(e2 | e1.gate = e2.gate

    and e1.component = e2.component))

- **Consistent type of a connection**
  The 'GateInstance's of the two 'endPoint's of a 'Connection' shall refer to the same 'GateType'.
  inv: **ConsistentConnectionType**:

  self.endPoint->at(0).gate.type = self.endPoint->at(1).gate.type

8.2.8 TestConfiguration

Semantics

A 'TestConfiguration' specifies the communication infrastructure necessary to build 'TestDescription's upon. As such, it contains all the elements required for information exchange: 'ComponentInstance's and 'Connection's.

It is not necessary that all 'ComponentInstance's contained in a 'TestConfiguration' are actually connected via 'Connection's. But for any 'TestConfiguration' at least the semantics of a minimal test configuration shall apply, which comprises one 'Tester' component and one 'SUT' component that are connected via one 'Connection'.

Generalization

- PackageableElement

Properties

- componentInstance: ComponentInstance [2..*] {unique}
  The 'ComponentInstance's of the 'TestConfiguration'.

- connection: Connection [1..*] {unique}
  The 'Connection's of the 'TestConfiguration' over which 'Interaction's are exchanged.

Constraints

- **'TestConfiguration' and components roles**
  A 'TestConfiguration' shall contain at least one 'Tester' and one 'SUT' 'ComponentInstance'.
  inv: **ComponentRoles**:

  self.componentInstance->exists(c | c.role.name = 'Tester')

  and self.componentInstance->exists(c | c.role.name = 'SUT')

- **Only 'Connection's between own 'ComponentInstance's**
  A 'TestConfiguration' shall only contain 'Connection's between gates of its own 'ComponentInstance's.
  inv: **OwnedComponents**:

  self.connection->forAll(c |

  self.componentInstance->contains(c.endPoint->at(0).component)

  and self.componentInstance->contains(c.endPoint->at(1).component))

- **Minimal 'TestConfiguration'**
  Each 'TestConfiguration' shall specify at least one 'Connection' that connects a 'GateInstance' of a 'ComponentInstance' in the role 'Tester' with a 'GateInstance' of a 'ComponentInstance' in the role 'SUT'.
  inv: **MinimalTestConfiguration**:

```
self.connection->exists(c |
  (c.endPoint->at(0).component.role.name = 'Tester'
 and c.endPoint->at(1).component.role.name = 'SUT')
 or (c.endPoint->at(0).component.role.name = 'SUT'
 and c.endPoint->at(1).component.role.name = 'Tester'))
```

- **At most one connection between any two 'GateInstance'/'ComponentInstance' pairs**
  Given the set of 'Connection's contained in a 'TestConfiguration'. There shall be no two 'Connection's containing 'GateReference's that in turn refer to identical pairs of 'GateInstance'/'ComponentInstance'.
  inv: **UniqueConnections**:

```
self.connection->forAll(c1 | self.connection->one(c2 |
  (c1.endPoint->at(0).component = c2.endPoint->at(0).component
 and c1.endPoint->at(0).gate = c2.endPoint->at(0).gate
 and c1.endPoint->at(1).component = c2.endPoint->at(1).component
 and c1.endPoint->at(1).gate = c2.endPoint->at(1).gate)

 or (c1.endPoint->at(1).component = c2.endPoint->at(0).component
 and c1.endPoint->at(1).gate = c2.endPoint->at(0).gate
 and c1.endPoint->at(0).component = c2.endPoint->at(1).component
 and c1.endPoint->at(0).gate = c2.endPoint->at(1).gate)))
```

9 Test Behaviour

9.1 Overview

The 'TestBehaviour' package defines all elements needed to describe the behaviour of a test description.

9.2 Test Description - Abstract Syntax and Classifier Description

9.2.1 TestDescription

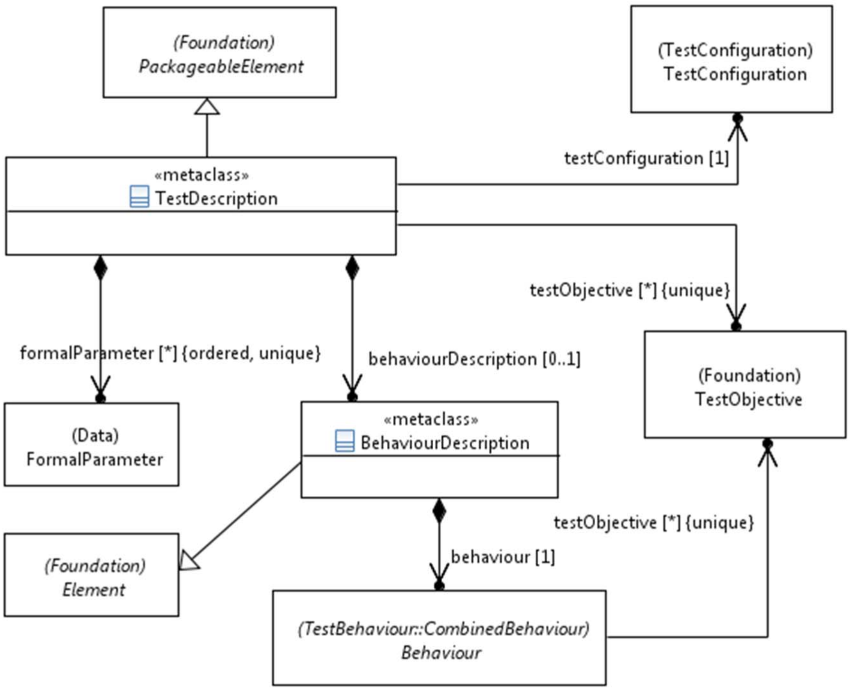

Figure 9.1: Test description

Semantics

A 'TestDescription' is a 'PackageableElement' that may contain a 'BehaviourDescription' defining the test behaviour based on ordered 'AtomicBehaviour' elements. It may also refer to 'TestObjective' elements that it realizes.

A 'TestDescription' is associated with exactly one 'TestConfiguration' that provides 'ComponentInstance's and 'GateInstance's to be used in the behaviour.

A 'TestDescription' may contain 'FormalParameter' that are used to pass data to behaviour.

If a 'TestDescription' with formal parameters is invoked within another 'TestDescription', actual parameters are provided via a 'TestDescriptionReference' (see clause 9.4.8). The mechanism of passing arguments to a 'TestDescription' that is invoked by a test management tool is not defined.

Generalization

- PackageableElement

Properties

- testConfiguration: TestConfiguration [1]
  Refers to the 'TestConfiguration' that is associated with the 'TestDescription'.

- behaviourDescription: BehaviourDescription [0..1]
  The actual behaviour of the test description in terms of 'Behaviour' elements.

- formalParameter: FormalParameter [0..*] {ordered, unique}
  The formal parameters that shall be substituted by actual data when the 'TestDescription' is invoked.

- testObjective: TestObjective [0..*]
  The 'TestObjective's that are realized by the 'TestDescription'.

Constraints

There are no constraints specified.

9.2.2 BehaviourDescription

Semantics

A 'BehaviourDescription' contains the behaviour of a 'TestDescription'.

Generalization

- Element

Properties

- behaviour: Behaviour [1]
  The contained root 'Behaviour' of the 'TestDescription'.

Constraints

There are no constraints specified.

9.3 Combined Behaviour - Abstract Syntax and Classifier Description

9.3.1 Behaviour

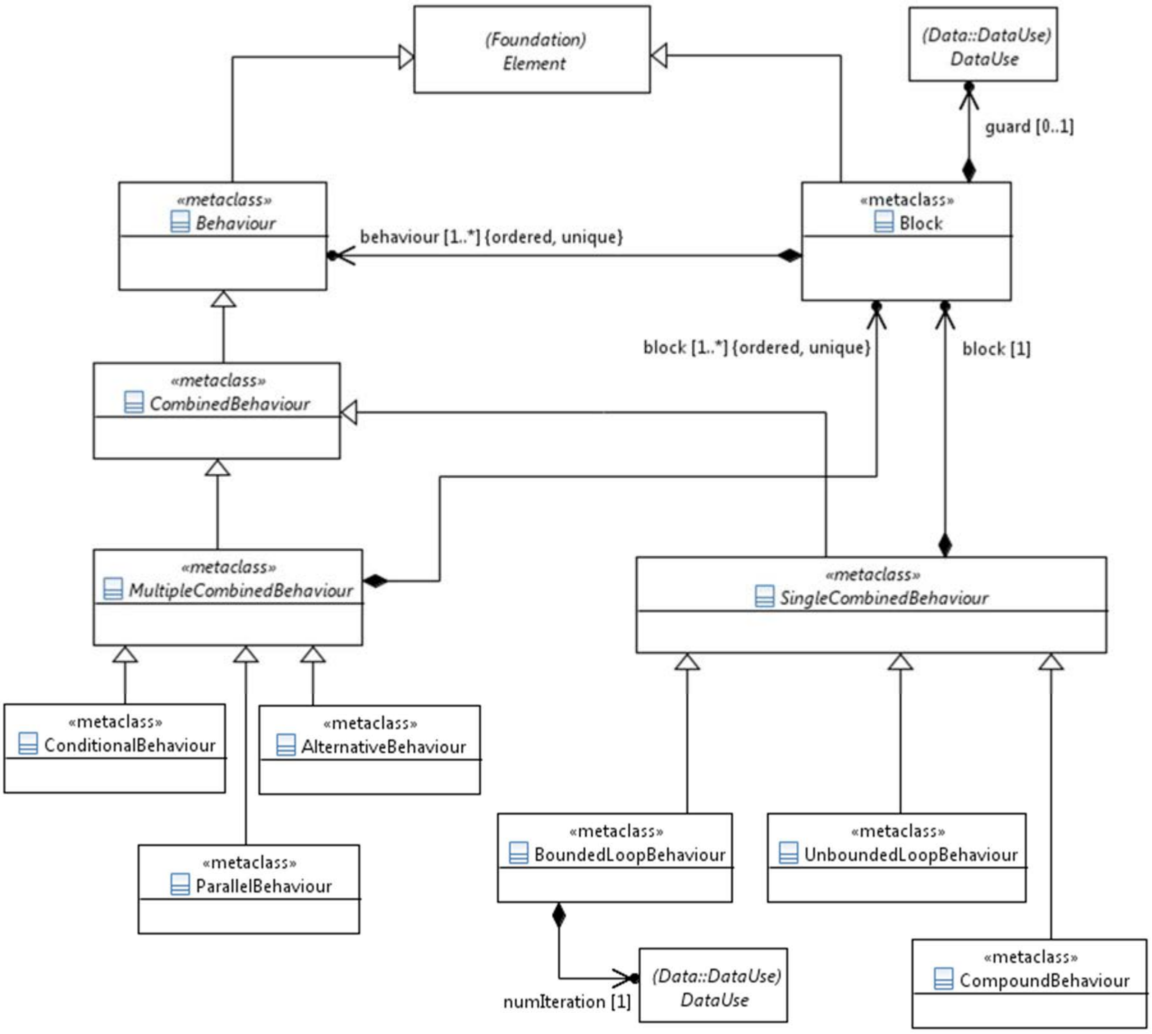

Figure 9.2: Combined behaviour concepts

Semantics

A 'Behaviour' is a constituent of the 'BehaviourDescription' of a 'TestDescription'. It represents the super-class for any concrete behavioural units a 'BehaviourDescription' is composed of. It offers the capability to refer to 'TestObjective's to enable traceability among 'TestObjective's and any concrete subclass of 'Behaviour'.

If a 'Behaviour' references a 'TestObjective', the 'Behaviour' is considered to realize/cover that 'TestObjective'.

Generalization

- Element

Properties

- testObjective: TestObjective [0..*] {unique}
  A set of 'TestObjective's that are realized by the 'Behaviour'.

Constraints

There are no constraints specified.

9.3.2 Block

Semantics

A 'Block' serves as a container for behavioural units that are executed sequentially. If a 'Block' has a 'guard', it shall only be executed if that guard evaluates to Boolean 'true'. If a 'Block' has no 'guard', it is equivalent to a 'guard' that evaluates to 'true'.

Generalization

- Element

Properties

- behaviour: Behaviour [1..*] {unique, ordered}
  The ordered set of 'Behaviour's that describe the sequentially executed units of 'Behaviour' contained in the 'Block'.
- guard: DataUse [0..1]
  An expression, whose type shall resolve to the predefined 'DataType' 'Boolean'.

Constraints

- **Guard shall evaluate to Boolean**
  The type of 'guard' shall be 'Boolean'.
  inv: **GuardType**:

  self.guard.oclIsUndefined() or self.guard.getDataType().name = 'Boolean'

9.3.3 CombinedBehaviour

Semantics

A 'CombinedBehaviour' is a behavioural constituent over all 'ComponentInstance's and 'GateReference's defined in the associated 'TestConfiguration' the containing 'TestDescription' operates on.

Additionally, a 'CombinedBehaviour' may contain any number of ordered 'PeriodicBehaviour's and 'ExceptionalBehaviour's that are evaluated in combination with the directly defined behaviour of the 'CombinedBehaviour'.

Generalization

- Behaviour

Properties

- periodic: PeriodicBehaviour [0..*] {unique, ordered}
  The ordered set of 'PeriodicBehaviour's attached to this 'CombinedBehaviour'.
- exceptional: ExceptionalBehaviour [0..*] {unique, ordered}
  The ordered set of 'ExceptionalBehaviour's attached to this 'CombinedBehaviour'.

Constraints

There are no constraints specified.

9.3.4 SingleCombinedBehaviour

Semantics

A 'SingleCombinedBehaviour' contains a single 'Block' of 'Behaviour'.

Generalization

- CombinedBehaviour

Properties

- block: Block [1]
  The 'Block' that is contained in the 'SingleCombinedBehaviour'.

Constraints

There are no constraints specified.

9.3.5 CompoundBehaviour

Semantics

A 'CompoundBehaviour' serves as a container for sequentially ordered 'Behaviour's. Its purpose is to group or structure behaviour, for example to describe the root behaviour of a 'TestDescription' or enable the assignment of 'PeriodicBehaviour's and/or 'ExceptionalBehaviour's.

Generalization

- SingleCombinedBehaviour

Properties

There are no properties specified.

Constraints

There are no constraints specified.

9.3.6 BoundedLoopBehaviour

Semantics

A 'BoundedLoopBehaviour' represents a recurring execution of the contained behaviour 'Block'. It has the same semantics as a *for-loop* statement in programming languages, i.e. the 'Block' shall be executed as many times as is determined by the 'numIteration' property.

The evaluation of the 'numIteration' expression happens once at the beginning of the 'BoundedLoopBehaviour'. For dynamically evaluated loop conditions, the 'UnboundedLoopBehaviour' shall be used.

The concrete mechanism of counting is not defined.

Generalization

- SingleCombinedBehaviour

Properties

- numIteration: DataUse [1]
  An expression that determines how many times the 'Block' of a 'BoundedLoopBehaviour' shall be executed.

Constraints

- **No guard constraint**
  The 'Block' of a 'BoundedLoopBehaviour' shall not have a 'guard'.
  inv: **BoundedGuard**:

  self.block.guard.oclIsUndefined()

- **Iteration number shall be countable and positive**
  The expression assigned to the 'numIteration' property shall evaluate to a countable 'SimpleDataInstance' of an arbitrary user-defined data type, e.g. a positive Integer value.
  inv: **LoopIteration**:

  This constraint cannot be expressed in OCL.

9.3.7 UnboundedLoopBehaviour

Semantics

An 'UnboundedLoopBehaviour' represents a recurring execution of the contained behaviour 'Block'. It has the same semantics as a *while-loop* statement in programming languages, i.e. the 'Block' shall be executed as long as the 'guard' of the 'Block' evaluates to Boolean 'true'. If the 'Block' has no guard condition, it shall be executed an infinite number of times, unless it contains a 'Break' or a 'Stop'.

Generalization

- SingleCombinedBehaviour

Properties

There are no properties specified.

Constraints

There are no constraints specified.

9.3.8 MultipleCombinedBehaviour

Semantics

A 'MultipleCombinedBehaviour' contains at least one potentially guarded 'Block' (in case of 'ConditionalBehaviour') or at least two ordered and potentially guarded 'Block's (in case of 'AlternativeBehaviour' or 'ParallelBehaviour').

Generalization

- CombinedBehaviour

Properties

- block: Block [1..*] {unique, ordered}
  The contained ordered list of 'Block's that specifies the behaviour of the 'MultipleCombinedBehaviour'.

Constraints

There are no constraints specified.

9.3.9 AlternativeBehaviour

Semantics

An 'AlternativeBehaviour' shall contain two or more 'Block's, each of which starting with a distinct *tester-input event* (see definition in clause 3.1).

Guards of all blocks are evaluated at the beginning of an 'AlternativeBehaviour'. Only blocks with guards that evaluate to Boolean 'true' are active in this 'AlternativeBehaviour'. If none of the guards evaluates to 'true', none of the 'Block's are executed, i.e. execution continues with the next 'Behaviour' following this 'AlternativeBehaviour'.

Only one of the alternative 'Block's will be executed. The evaluation algorithm of an alternative 'Block' at runtime is a step-wise process:

1) All guards are evaluated and only those 'Block's, whose guards evaluated to 'true' are collected into an ordered set of potentially executable 'Block's.
2) The tester-input event of each potentially executable 'Block' is evaluated in the order, in which the 'Block's are specified.
3) The first 'Block' with an executable tester-input event is entered; the tester-input event itself and the subsequent 'Behaviour' of this 'Block' are executed.

Generalization

- MultipleCombinedBehaviour

Properties

There are no properties specified.

Constraints

- **Number of 'Block's**
  An 'AlternativeBehaviour' shall contain at least two 'Block's.
  inv: **AlternativeBlockCount**:

  self.block->size() > 1

- **First behaviour of 'Block's**
  Each block of an 'AlternativeBehaviour' shall start with a *tester-input event*.
  inv: **FirstBlockBehaviour**:

  self.block->forAll(b | b.behaviour->first().isTesterInputEvent())

9.3.10 ConditionalBehaviour

Semantics

A 'ConditionalBehaviour' represents an alternative choice over a number of 'Block's. A 'ConditionalBehaviour' is equivalent to an *if-elseif-else* statement in programming languages, e.g. *select-case* statement in TTCN-3.

Only one of the alternative 'Block's will be executed. The evaluation algorithm of an alternative 'Block' at runtime is a step-wise process:

1) The guards of the specified 'Block's are evaluated in the order of their definition.

2) The first 'Block', whose guard is evaluated to 'true', is entered and the 'Behaviour' of this 'Block' is executed.

If none of the guards evaluates to 'true', none of the 'Block's are executed, i.e. execution continues with the next 'Behaviour' following this 'ConditionalBehaviour'.

NOTE: Typically, 'Block's are specified with a 'guard'. If a guard is missing, it is equivalent to a guard that evaluates to 'true' (see clause 9.3.2). The latter case is also known as the *else* branch of an *if-elseif-else* statement in a programming language. Blocks specified after this *else* block would never be executed.

Generalization

- MultipleCombinedBehaviour

Properties

There are no properties specified.

Constraints

- **Guard for 'ConditionalBehaviour' with single block**
  If there is only one 'Block' specified, it shall have a 'guard'.
  inv: **ConditionalFirstGuard**:

  self.block->size() > 1 or not self.block->first().guard.oclIsUndefined()

- **Possible else block for 'ConditionalBehaviour' with multiple blocks**
  All 'Block's specified, except the last one, shall have a 'guard'.
  inv: **ConditionalLastGuard**:

  self.block->size() = 1

  or self.block->forAll(b | b = self.block->last() or not b.guard.oclIsUndefined())

9.3.11 ParallelBehaviour

Semantics

A 'ParallelBehaviour' represents the parallel execution of 'Behaviour's contained in the multiple 'Block's. That is, the relative execution order of the 'Behaviour's among the different 'Block's is not specified. The execution order of 'Behaviour's within the same 'Block' shall be kept as specified, even though it might be interleaved with 'Behaviour's from other parallel 'Block's.

'Block's may have guards. Guards of all blocks are evaluated at the beginning of a 'ParallelBehaviour'. Only blocks with guards that evaluate to Boolean 'true' are executed in this 'ParallelBehaviour'. If none of the guards evaluates to 'true', none of the 'Block's are executed, i.e. execution continues with the next 'Behaviour' following this 'ParallelBehaviour'.

The 'ParallelBehaviour' terminates when the all 'Block's are terminated.

Generalization

- MultipleCombinedBehaviour

Properties

There are no properties specified.

Constraints

- **Number of blocks in 'ParallelBehaviour'**
  There shall be at least two 'Block's specified.
  inv: **ParallelBlockCount**:

  self.block->size() > 1

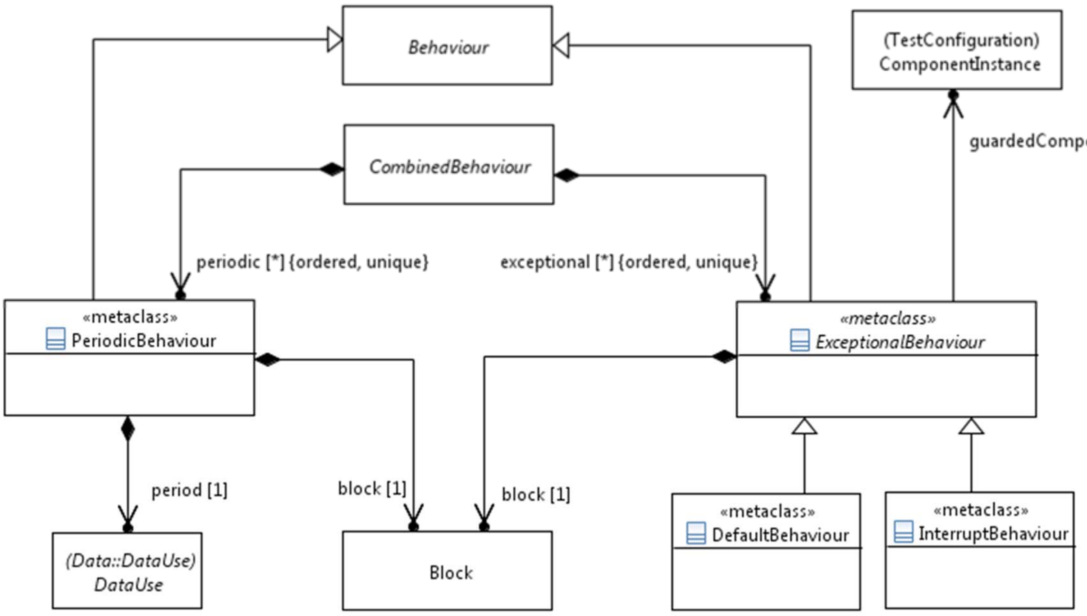

Figure 9.3: Exceptional and periodic behaviour

9.3.12 ExceptionalBehaviour

Semantics

'ExceptionalBehaviour' is optionally contained within a 'CombinedBehaviour'. It is a 'Behaviour' that consists of one 'Block' that shall have no guard and shall start with a *tester-input event* (see definition in clause 3.1).

An 'ExceptionalBehaviour' may specify the 'ComponentInstance' that it guards. This allows restricting the possible situations when the 'Behaviour' of the 'CombinedBehaviour' containing this 'ExceptionalBehaviour' is executed. In this case only those 'Behaviour's that are defined in the scope of the 'guardedComponent' force the 'ExceptionalBehaviour' to be activated.

An 'ExceptionalBehaviour' defines 'Behaviour' that is an alternative to every 'Interaction' directly or indirectly contained in the enclosing 'CombinedBehaviour' that matches one of the following two conditions:

- If no 'guardedComponent' reference is present, an interaction whose target 'GateInstance' is associated to a 'ComponentInstance' with the role of 'Tester'.

- If a 'guardedComponent' reference is present, an interaction whose target 'GateInstance' is associated to the same 'ComponentInstance' as referenced by the 'guardedComponent' property.

In case of more than one 'ExceptionalBehaviour' is attached to the same 'CombinedBehaviour', the corresponding 'AlternativeBehaviour' would contain the 'Blocks' of all the attached 'ExceptionalBehaviour's in the same order.

An 'ExceptionalBehaviour' can be either a 'DefaultBehaviour' or an 'InterruptBehaviour'.

Generalization

- Behaviour

Properties

- block: Block [1]
  The contained 'Block' that specifies the 'Behaviour' of the 'ExceptionalBehaviour'.

- guardedComponent: ComponentInstance [0..1]
  Reference to a 'ComponentInstance' with role 'Tester', for which the 'ExceptionalBehaviour' is to be applied.

Constraints

- **First 'AtomicBehaviour' in block allowed**
  Each block of an 'ExceptionalBehaviour' shall start with a *tester-input event*.
  inv: **FirstExceptionalBehaviour**:

  ```
  self.block.behaviour->first().isTesterInputEvent()
  ```

- **Guarded component shall be a 'Tester' component**
  The 'guardedComponent' shall refer to a 'ComponentInstance' with the role of 'Tester'.
  inv: **ExceptionalGuardedComponent**:

  ```
  self.guardedComponent.oclIsUndefined() or self.guardedComponent.role.name = 'Tester'
  ```

9.3.13 DefaultBehaviour

Semantics

A 'DefaultBehaviour' is a specialization of an 'ExceptionalBehaviour'.

If a 'DefaultBehaviour' of the 'CombinedBehaviour', which it is attached to, becomes executable and the 'Behaviour' defined in the 'Block' of the 'DefaultBehaviour' subsequently completes execution, the execution of the 'CombinedBehaviour' continues with the next 'Behaviour' that follows the 'Behaviour' that caused the execution of the 'DefaultBehaviour'.

Generalization

- ExceptionalBehaviour

Properties

There are no properties specified.

Constraints

There are no constraints specified.

9.3.14 InterruptBehaviour

Semantics

An 'InterruptBehaviour' is a specialization of an 'ExceptionalBehaviour'.

If an 'InterruptBehaviour' of the 'CombinedBehaviour', which it is attached to, becomes executable and the 'Behaviour' defined in the 'Block' of the 'InterruptBehaviour' subsequently completes execution, the execution of the 'CombinedBehaviour' continues with the same 'Behaviour' that caused the execution of the 'InterruptBehaviour'.

Generalization

- ExceptionalBehaviour

Properties

There are no properties specified.

Constraints

There are no constraints specified.

9.3.15 PeriodicBehaviour

Semantics

A 'PeriodicBehaviour' defines a 'Behaviour' in a single 'Block' that is executed periodically in parallel with the 'CombinedBehaviour' it is attached to. The recurrence interval of the execution is specified by its 'period' property. If the execution of the contained 'Block' takes longer than the specified period, the semantics of the resulting behaviour is unspecified.

The execution of 'PeriodicBehaviour' terminates if the 'CombinedBehaviour', which it is attached to, terminates.

Generalization

- Behaviour

Properties

- block: Block [1]
  The contained 'Block', whose 'Behaviour' is executed periodically in parallel with the 'Behaviour' of the 'CombinedBehaviour', which this 'PeriodicBehaviour' is attached to.

- period: DataUse [1]
  The recurrence interval of executing the behaviour of the 'Block' specified by the 'block' property.

Constraints

- **'Time' data type for period expression**
  The 'DataUse' expression assigned to the 'period' shall evaluate to a data instance of the 'Time' data type.
  inv: **PeriodType**:

  self.period.getDataType().oclIsKindOf(Time)

9.4 Atomic Behaviour - Abstract Syntax and Classifier Description

9.4.1 AtomicBehaviour

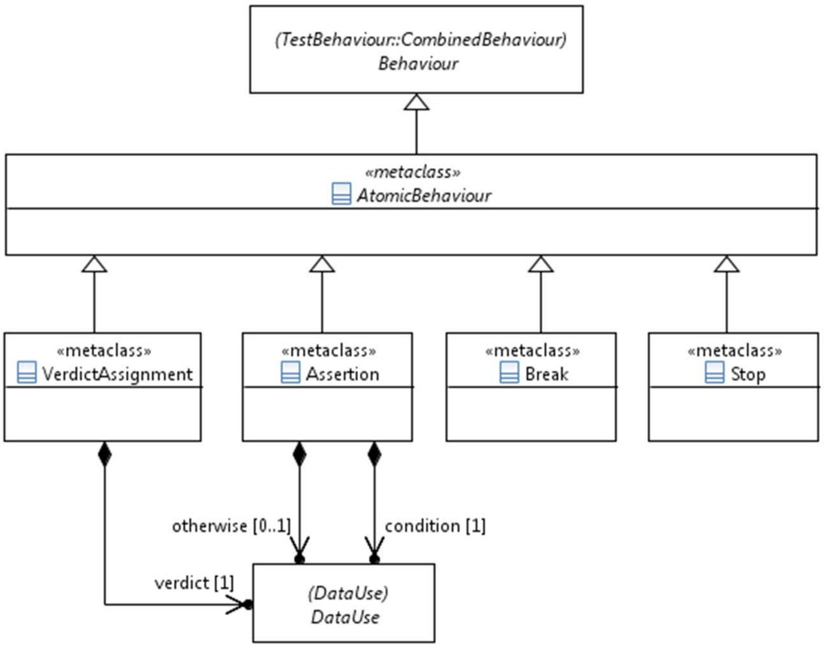

Figure 9.4: Global atomic behaviour concepts

Semantics

An 'AtomicBehaviour' defines the simplest form of behavioural activity of a 'TestDescription' that cannot be decomposed further.

An 'AtomicBehaviour' can have a 'TimeLabel' that holds the timestamp of this behaviour when it is executed (see clause 7.2.2). In addition, an 'AtomicBehaviour' may contain a list of 'TimeConstraint' expressions that affect its execution time (see clause 7.2.4).

Generalization

- Behaviour

Properties

- timeLabel: TimeLabel [0..1]
  Refers to the time label contained in the 'AtomicBehaviour'.

- timeConstraint: TimeConstraint [0..*] {unique}
  Refers to a contained list of 'TimeConstraint's that determines the execution of the given 'AtomicBehaviour' by means of time constraint expressions.

Constraints

There are no constraints specified.

9.4.2 Break

Semantics

A 'Break' terminates the execution of the behavioural 'Block', in which the 'Break' is contained. Execution continues with the 'Behaviour' that follows afterwards. In case of 'ParallelBehaviour', a 'Break' terminates only the execution of its own 'Block', but does not affect the execution of the other parallel 'Block'(s).

Generalization

- AtomicBehaviour

Properties

There are no properties specified.

Constraints

There are no constraints specified.

9.4.3 Stop

Semantics

'Stop' is used to describe an explicit and immediate stop of the execution of the entire 'TestDescription' that was initially invoked. No further behaviour shall be executed beyond a 'Stop'. In particular, a 'Stop' in a referenced (called) 'TestDescription' shall also stop the behaviour of the referencing (calling) 'TestDescription'(s).

Generalization

- AtomicBehaviour

Properties

There are no properties specified.

Constraints

There are no constraints specified.

9.4.4 VerdictAssignment

Semantics

The 'VerdictAssignment' is used to set the verdict of the test run explicitly. This might be necessary if the implicit verdict mechanism described below is not sufficient.

By default, the test description specifies the expected behaviour of the system. If an execution of a test description performs the expected behaviour, the verdict is set to 'pass' implicitly. If a test run deviates from the expected behaviour, the verdict 'fail' will be assigned to the test run implicitly. Other verdicts, including 'inconclusive' and user-definable verdicts, need to be set explicitly within a test description.

Generalization

- AtomicBehaviour

Properties

- verdict: StaticDataUse [1]
  Stores the value of the verdict to be set.

Constraints

- **Verdict of type 'Verdict'**
  The 'verdict' shall evaluate to a, possibly predefined, instance of a 'SimpleDataInstance' of data type 'Verdict'.
  inv: **VerdictType**:

  self.verdict.getDataType().name = 'Verdict'

9.4.5 Assertion

Semantics

An 'Assertion' allows the specification of a test 'condition' that needs to evaluate to 'true' at runtime for a passing test, in which case the implicit test verdict is set to 'pass'. If the 'condition' is not satisfied, the test verdict is set to 'fail' or to the optionally specified verdict given in 'otherwise'.

Generalization

- AtomicBehaviour

Properties

- condition: DataUse [1]
  Refers to the test condition that is evaluated.
- otherwise: StaticDataUse [0..1]
  Refers to the value of the verdict to be set if the assertion fails.

Constraints

- **Boolean condition**
  The 'condition' shall evaluate to predefined 'DataType' 'Boolean'.
  inv: **AssertionOtherwise**:

  self.condition.getDataType().name = 'Boolean'

- **Otherwise of type 'Verdict'**
  The 'otherwise' shall evaluate to a, possibly predefined, instance of a 'SimpleDataInstance' of data type 'Verdict'.
  inv: **AssertionVerdict**:

  self.otherwise.oclIsUndefined() or self.otherwise.getDataType().name = 'Verdict'

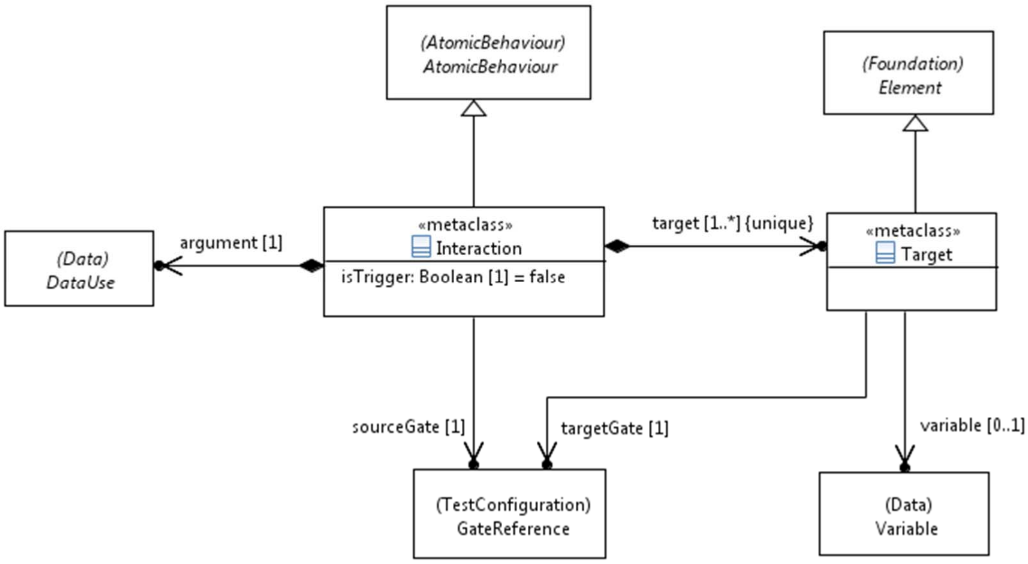

Figure 9.5: Interaction behaviour

9.4.6 Interaction

Semantics

An 'Interaction' is a representation of any information exchanged between connected components. An 'Interaction' is an 'AtomicBehaviour', i.e. it cannot be decomposed into smaller behavioural activities. It is also directed, i.e. the information being exchanged is sent by a component via the 'sourceGate' and received by one or many components via the other 'targetGate's (point-to-point and point-to-multipoint communication, see clause 8.2.7).

NOTE 1: In a concrete realization, an interaction can represent typically one of the following options, among others:

- Message-based communication: The data of an interaction argument represents a message being sent (from 'sourceGate') and received (by 'targetGate').
- Procedure-based communication: The data of an interaction argument represents a remote function call being initiated (from 'sourceGate') and invoked (at 'targetGate') or its return values being transmitted back.
- Shared variable access: The data of an interaction argument represents a shared variable being read ('sourceGate' is the gate of the component that owns this variable, 'targetGate' is the gate of the reading component) or written ('sourceGate' is the gate of the component that wants to change the value of a shared variable, 'targetGate' is the gate of the component that owns this variable).

An 'Interaction' with a 'Target' that in turn—via its 'GateReference'—refers to a 'ComponentInstance' in the role 'Tester' is called a *tester-input event*.

The 'argument' property of an 'Interaction' refers to the expected data value being exchanged. Executing an 'Interaction' implies that this expected data value occurs at runtime among the participating components and the implicit test verdict 'pass' shall be set. If the expected value does not occur, i.e. either the interaction with the expected value does not occur at all within an arbitrary time or an interaction with different value occurs, the test verdict 'fail' shall be set.

NOTE 2: The time period to wait for the specified interaction to occur is defined outside the scope of the present document.

If an 'Interaction' is a trigger 'Interaction' ('isTrigger' property is set), execution of the 'Interaction' terminates only if the expected data occurred (test verdict 'pass') or the expected data did not occur within an arbitrary time (test verdict 'fail'). Intermediate 'Interaction'(s) with data values that do not match the expected value are discarded during the execution of that trigger 'Interaction'.

The 'DataUse' specification, which the 'argument' refers to, can contain 'Variable's of 'ComponentInstance's participating in this 'Interaction'. Use of a 'Variable' in an 'argument' specification implies the use of its value. Additionally, placeholders such as 'AnyValue' or 'AnyValueOrOmit' can be used if the concrete value is not known or irrelevant (see clauses 6.3.6 and 6.3.7).

NOTE 3: How the <undefined> value within the 'DataUse' specification of 'argument' is resolved is outside of the scope of the present document.

To store the actual data of an 'Interaction' received at the 'Target' side at runtime, a 'Variable' with the same data type as the 'argument' specification can be used, provided that the 'Variable' is local to the same 'ComponentInstance' that is also referred to in the 'targetGate'.

NOTE 4: If the 'Variable' refers to a 'StructuredDataType', the non-optional 'Member's of this data type can be assigned values only that are different from 'OmitValue'; see clause 6.3.2.

Generalization

- AtomicBehaviour

Properties

- isTrigger: Boolean [1] = false
  If set to 'true', this property denotes a trigger interaction that is successful only if a matching 'argument' has occurred in this interaction. Previously occurring unmatched 'argument's are discarded.

- argument: DataUse [1]
  Refers to a 'DataUse' that is taken as the argument (data value) of this interaction.

- sourceGate: GateReference [1]
  Refers to a 'GateReference' that acts as the source of this interaction.

- target: Target [1..*] {unique}
  Refers to a contained list of 'Target' 'GateReference's of different component instances. If the list contains more than one element, it implies point-to-multipoint communication.

Constraints

- **Gate references of an interaction shall be connected**
  The 'GateReference's that act as source or target(s) of an 'Interaction' shall be interconnected by a 'Connection' which is contained in the 'TestConfiguration' referenced by the 'TestDescription' containing the 'Interaction'.
  inv: **ConnectedInteractionGates**:

```
self.target.forAll(t |
  self.getTestDescription().testConfiguration.connection->exists(c |
    (c.endPoint->at(0).gate = self.sourceGate.gate
   and c.endPoint->at(0).component = self.sourceGate.component
   and c.endPoint->at(1).gate = t.targetGate.gate
   and c.endPoint->at(1).component = t.targetGate.component)

   or (c.endPoint->at(1).gate = self.sourceGate.gate
   and c.endPoint->at(1).component = self.sourceGate.component
   and c.endPoint->at(0).gate = t.targetGate.gate
   and c.endPoint->at(0).component = t.targetGate.component)))
```

- **Type of interaction argument**
  The 'DataUse' specification referred to in the 'argument' shall match one of the 'DataType's referenced in the 'GateType' definition of the 'GateInstance's referred to by the source and target 'GateReference's of the 'Interaction'.
  inv: **InteractionArgumentAndGateType**:

```
  (self.argument.oclIsKindOf(AnyValue)
 and self.argument.dataType.oclIsUndefined())
 or (self.sourceGate.gate.type.dataType->includes(self.argument.getDataType())
 and self.target.forAll(t | t.targetGate.gate.type.dataType->includes(self.argument.getDataType())))
```

- **Use of variables in the 'argument' specification**
  The use of 'Variable's in the 'DataUse' specification shall be restricted to 'Variable's of 'ComponentInstance's that participate in this 'Interaction' via the provided 'GateReference's.
  inv: **InteractionArgumentVariableUse**:

```
  (not self.argument.oclIsKindOf(VariableUse)
  or (self.sourceGate.component = self.argument.componentInstance
   or self.target->forAll(t | t.targetGate.component = self.argument.componentInstance)))


and self.argument.argument->forAll(a |
      not a.dataUse.oclIsKindOf(VariableUse)
    or (self.sourceGate.component = a.dataUse.componentInstance
     or self.target->forAll(t | t.targetGate.component = a.dataUse.componentInstance)))


and self.argument.argument->closure(a | a.dataUse.argument)->forAll(a |
      not a.dataUse.oclIsKindOf(VariableUse)
    or (self.sourceGate.component = a.dataUse.componentInstance
     or self.target->forAll(t | t.targetGate.component = a.dataUse.componentInstance)))
```

- **Matching data type for 'argument' and 'variable'**
  The 'DataUse' specification of the 'argument' and the referenced 'Variable' of any 'Target' shall refer to the same 'DataType'.
  inv: **InteractionArgumentAndVariableType**:

```
  self.target.forAll(t | t.variable.oclIsUndefined()
    or t.variable.dataType = self.argument.getDataType())
```

- **'DataType' for 'AnyValue' as 'argument'**
  If 'AnyValue' is used as 'argument' and of the 'Interaction' and the optional 'ParameterBinding's are provided, the 'dataType' of 'AnyValue' shall be specified.
  inv: **AnyValueDataType**:

```
  self.argument.oclIsKindOf(AnyValue)
and (self.argument.argument.isEmpty() or not self.argument.dataType.oclIsUndefined())
```

9.4.7 Target

Semantics

A 'Target' holds the 'GateReference' that acts as target for the 'Interaction', which in turn contains this 'Target', and an optional 'Variable' that stores the received data value from this 'Interaction'.

Generalization

- Element

Properties

- targetGate: GateReference [1]
  Refers to the 'GateReference' that acts as target for an interaction.

- variable: Variable [0..1]
  Refers to a 'Variable' that stores the received data value from the 'Interaction'.

Constraints

- **Variable and target gate of the same component instance**
  The referenced 'Variable' shall exist in the same 'ComponentType' as the 'GateInstance' that is referred to by the 'GateReference' of the 'targetGate'.
  inv: **TargetComponent**:

  self.variable.oclIsUndefined()

  or self.targetGate.component.type.variable->contains(self.variable)

- **Variable of a tester component only**
  If a 'Variable' is specified, the 'ComponentInstance' referenced by 'targetGate' shall be in the role 'Tester'.
  inv: **TargetVariableComponentRole**:

  self.variable.oclIsUndefined() or self.targetGate.component.role.name = 'Tester'

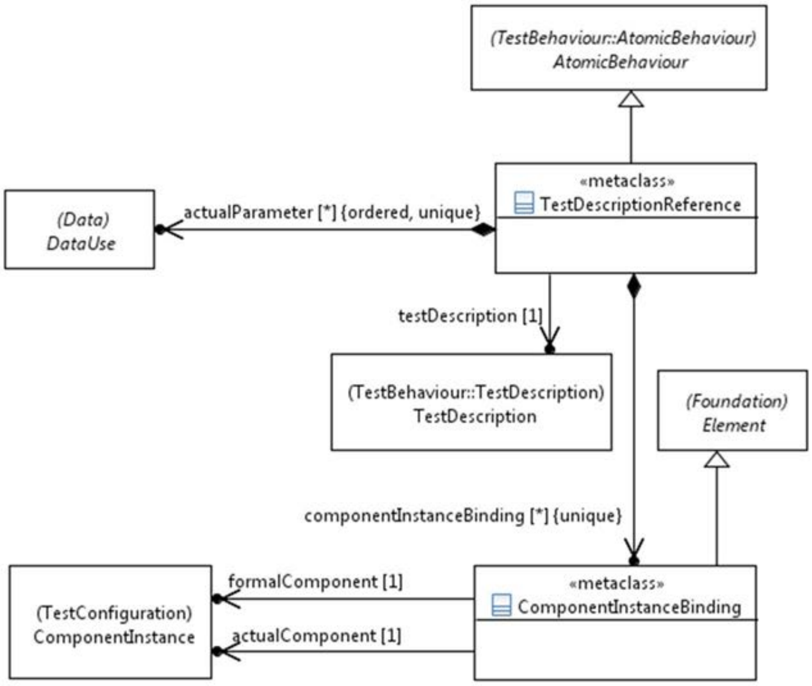

Figure 9.6: Test description reference

9.4.8 TestDescriptionReference

Semantics

A 'TestDescriptionReference' is used to describe the invocation of the behaviour of a test description within another test description. The invoked behaviour is executed in its entirety before the behaviour of the invoking test description is executed further. A 'TestDescriptionReference' has a possibly empty list of actual parameters which is passed to the referenced 'TestDescription'. It also has an optional list of bindings between component instances of the involved test configurations that shall be present if the test configurations of the referencing (invoking) and referenced (invoked) test descriptions are different.

If the 'TestConfiguration' of the invoked 'TestDescription' is different from the one of the invoking 'TestDescription', it shall be compatible with it. The compatibility rule is defined below. In case of different test configurations, 'ComponentInstance's contained in the 'TestConfiguration' of the invoked 'TestDescription' will be substituted with 'ComponentInstance's of the 'TestConfiguration' of the invoking 'TestDescription'. Substitution is implicit when both test configurations coincide. Explicit substitution is defined using the 'ComponentInstanceBinding'.

Generalization

- AtomicBehaviour

Properties

- testDescription: TestDescription [1]
  Refers the test description whose behaviour is invoked.

- actualParameter: DataUse [0..*] {ordered}
  Refers to an ordered set of actual parameters passed to the referenced test description.

- componentInstanceBinding: ComponentInstanceBinding [0..*] {unique}
  Defines explicit bindings between 'ComponentInstance's from 'TestConfiguration' of invoking 'TestDescription' and those from the 'TestConfiguration' of the invoked 'TestDescription'.

Constraints

- **Number of actual parameters**
  The number of actual parameters in the 'TestDescriptionReference' shall be equal to the number of formal parameters of the referenced 'TestDescription'.
  inv: **ParameterCount**:

  ```
  self.actualParameter->size() = self.testDescription.formalParameter->size()
  ```

- **No use of variables in actual parameters**
  The 'DataUse' expressions used to describe actual parameters shall not contain variables directly or indirectly.
  inv: **NoVariables**:

  ```
  self.actualParameter.forAll(p |
    not p.oclIsKindOf(VariableUse)
  and p.argument->forAll(a | not a.dataUse.oclIsKindOf(VariableUse))
  and p.argument->closure(a | a.dataUse.argument)->forAll(a |
      not a.dataUse.oclIsKindOf(VariableUse)))
  ```

- **Matching parameters**
  The actual parameter *AP*[*i*] of index *i* in the ordered list of 'actualParameter's shall match 'DataType' of the 'FormalParameter' *FP*[*i*] of index *i* in the ordered list of formal parameters of the referenced 'TestDescription'.
  inv: **TestDescriptionParameters**:

  ```
  self.actualParameter.forAll(ap |
    self.testDescription.formalParameter->size() > self.actualParameter->indexOf(ap)
  and ap.getDataType() =
      self.testDescription.formalParameter->at(self.actualParameter.indexOf(ap)).dataType)
  ```

- **Restriction to 1:1 component instance bindings**
  If component instance bindings are provided, the component instances referred to in the bindings shall occur at most once for the given test description reference.
  inv: **UniqueComponentBindings**:

  ```
  self.componentInstanceBinding->isEmpty()
  or self.componentInstanceBinding->forAll(b |
    self.componentInstanceBinding->one(c |
      c.formalComponent = b.formalComponent or c.actualComponent = b.actualComponent))
  ```

- **Compatible test configurations**
  The test configuration *TConf2* of the referenced (invoked) test description shall be compatible with the test configuration *TConf1* of the referencing (invoking) test description under the provision of a list of bindings between component instances in *TConf1* and *TConf2*. Compatibility is then defined in the following terms:

  - All component instances in *TConf2* can be mapped to component instances of *TConf1*.
    A component instance *B* of test configuration *TConf2* can be mapped to a component instance *A* of test configuration *TConf1*if and only if:

    a) there is a binding pair (*A*, *B*) provided;

    b) *A* and *B* refer to the same component type; and

c) *A* and *B* have the same component instance role {SUT, Tester} assigned.

- All connections between component instances in *TConf2* exist also between the mapped component instances in *TConf1* and the type of a connection in *TConf2* equals the type of the related connection in *TConf1*.
  Two connections of the two test configurations are equal if and only if the same gate instances are used in the definition of the gate references of the connections.

NOTE 1: The compatibility between test configurations is defined asymmetrically. That is, if *TConf2* is compatible with *TConf1*, it does not imply that *TConf1*is compatible with *TConf2*. If *TConf2* is compatible with *TConf1*, it is said that *TConf2* is a sub-configuration of *TConf1* under a given binding.

NOTE 2: If two test configurations are equal, then they are also compatible.

inv: **CompatibleConfiguration**:

```
  (self.getTestDescription().testConfiguration = self.testDescription.testConfiguration
 and self.componentInstanceBinding->isEmpty())
 or (self.testDescription.testConfiguration.connection->forAll(c |
     self.componentInstanceBinding->exists(i |
       i.formalComponent = c.endPoint->at(0).component)
   and self.componentInstanceBinding->exists(i |
       i.formalComponent = c.endPoint->at(1).component)
   and self.getTestDescription().testConfiguration.connection->select(p |
       (p.endPoint->at(0).component = self.componentInstanceBinding->select(i |
         i.formalComponent = c.endPoint->at(0).component)->first().actualComponent
       and p.endPoint->at(1).component = self.componentInstanceBinding->select(i |
           i.formalComponent = c.endPoint->at(1).component)->first().actualComponent)
     or (p.endPoint->at(1).component = self.componentInstanceBinding->select(i |
         i.formalComponent = c.endPoint->at(0).component)->first().actualComponent
       and p.endPoint->at(0).component = self.componentInstanceBinding->select(i |
           i.formalComponent =
             c.endPoint->at(1).component)->first().actualComponent))->exists(m |
               (m.endPoint->at(0).gate.type = c.endPoint->at(0).gate.type
              and m.endPoint->at(1).gate.type = c.endPoint->at(1).gate.type)
              or (m.endPoint->at(0).gate.type = c.endPoint->at(1).gate.type
              and m.endPoint->at(1).gate.type = c.endPoint->at(0).gate.type))))
```

9.4.9 ComponentInstanceBinding

Semantics

The 'ComponentInstanceBinding' is used with the 'TestDescriptionReference' in case when the 'TestConfiguration' of the invoked 'TestDescription' differs from that of the invoking 'TestDescription'. It specifies that a (formal) 'ComponentInstance' in the invoked 'TestDescription' will be substituted with an (actual) 'ComponentInstance' from the invoking 'TestDescription'.

Additional rules and semantics are defined in clause 9.4.8.

Generalization

- Element

Properties

- formalComponent: ComponentInstance [1]
  Refers to a 'ComponentInstance' contained in the 'TestConfiguration' of the invoked 'TestDescription'.

- actualComponent: ComponentInstance [1]
  Refers to a 'ComponentInstance' contained in the 'TestConfiguration' of the invoking 'TestDescription'.

Constraints

- **Matching component types**
  Both, the formal and the actual component instances, shall refer to the same 'ComponentType'.
  inv: **BindingComponentTypes**:

  self.formalComponent.type = self.actualComponent.type

- **Matching component instance roles**
  Both, the formal and the actual component instances, shall have the same 'ComponentInstanceRole' assigned to.
  inv: **BindingComponentRoles**:

  self.formalComponent.role = self.actualComponent.role

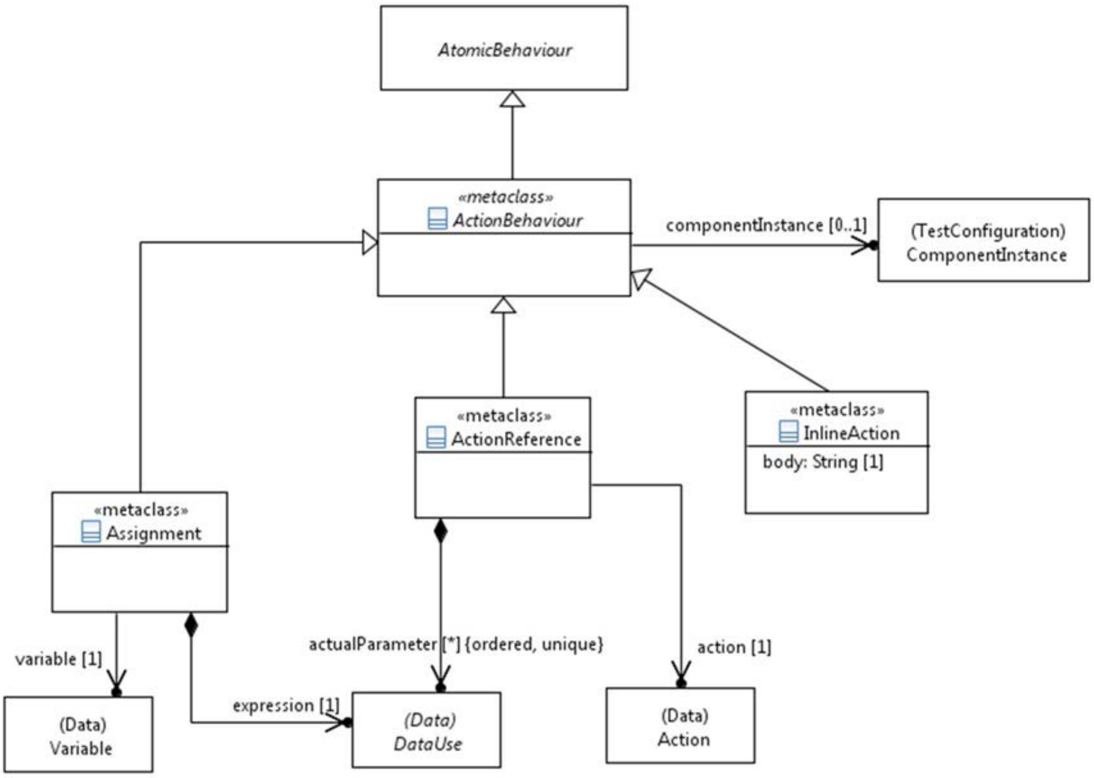

Figure 9.7: Action behaviour concepts

9.4.10 ActionBehaviour

Semantics

'ActionBehaviour' is a refinement of 'AtomicBehaviour' and a super-class for 'ActionReference', 'InlineAction' and 'Assignment'.

It may refer to a 'Tester' 'ComponentInstance' that specifies the location, on which the 'ActionBehaviour' is executed. If no reference to a 'ComponentInstance' is given, the 'ActionBehaviour' is executed in the global scope of the associated 'TestConfiguration'.

Generalization

- AtomicBehaviour

Properties

- componentInstance: ComponentInstance [0..1]
  Refers to a 'ComponentInstance' from the 'TestConfiguration', on which the 'ActionBehaviour' is performed.

Constraints

- **'ActionBehaviour' on 'Tester' components only**
  The 'ComponentInstance' that an 'ActionBehaviour' refers to shall be of role 'Tester'.
  inv: **ActionBehaviourComponentRole**:

  self.componentInstance.oclIsUndefined() or self.componentInstance.role.name = 'Tester'

9.4.11 ActionReference

Semantics

An 'ActionReference' invokes an 'Action'. It may carry a list of 'DataUse' specifications to denote actual parameters of this 'Action'.

Generalization

- ActionBehaviour

Properties

- action: Action [1]
  Refers to the 'Action' to be executed.

- actualParameter: DataUse [0..*] {ordered, unique}
  Refers to an ordered set of actual parameters passed to the referenced action.

Constraints

- **Matching parameters**
  The actual parameter *AP*[*i*] of index *i* in the ordered set of 'actualParameter's shall match the 'DataType' of the 'FormalParameter' *FP*[*i*] of index *i* in the ordered set of formal parameters of the referenced 'Action'.
  inv: **ActionParameters**:

```
self.actualParameter->forAll(ap | ap.getDataType() =
    self.action.formalParameter->at(self.actualParameter->indexOf(ap)).dataType)
```

9.4.12 InlineAction

Semantics

An 'InlineAction' denotes the execution of an informally defined action. The semantics of its execution is outside the scope of TDL.

Generalization

- ActionBehaviour

Properties

- body: String [1]
  The action described as free text.

Constraints

There are no constraints specified.

9.4.13 Assignment

Semantics

An 'Assignment' denotes the assignment of a value that is expressed as a 'DataUse' specification to a variable within a component instance.

Generalization

- ActionBehaviour

Properties

- variable: Variable [1]
  Refers to the variable that is assigned the data value resulting from evaluating the 'expression'.

- expression: DataUse [1]
  Refers to the 'DataUse' specification, which is evaluated at runtime and whose value is assigned to the referenced 'Variable'.

Constraints

- **Known component instance**
  The property 'componentInstance' shall be set to identify the 'Variable' in this 'Assignment'.
  inv: **AssignmentComponent**:

  not self.componentInstance.oclIsUndefined()

- **Matching data type**
  The provided 'DataUse' expression shall match the 'DataType' of the referenced 'Variable'.
  inv: **AssignmentDataType**:

  self.expression.getDataType() = self.variable.dataType

10 Predefined TDL Model Instances

10.1 Overview

This clause lists the predefined element instances for various meta-model elements that shall be a part of a standard-compliant TDL implementation. It is not specified how these predefined instances are made available to the user. However, it is implied that in different TDL models predefined instances with the same name are semantically equivalent. This statement implies further that predefined instances shall not be overwritten with different instances of the same name, but with a different meaning.

10.2 Predefined Instances of the 'SimpleDataType' Element

10.2.1 Boolean

The predefined 'SimpleDataType' 'Boolean' denotes the common Boolean data type with the two values (instances of 'SimpleDataInstance') 'true' and 'false' to denote truth values (see clause 10.3) and support logical expressions.

No assumptions are made about how 'Boolean' is implemented in an underlying concrete type system.

10.2.2 Verdict

The predefined 'SimpleDataType' 'Verdict' denotes the data type that holds the possible test verdicts of a 'TestDescription' (see clause 10.3). The 'Verdict' allows the definition of functions that use this data type as an argument or as the return type.

No assumptions are made about how 'Verdict' is implemented in an underlying concrete type system.

10.2.3 TimeLabelType

The predefined 'SimpleDataType' 'TimeLabelType' denotes the data type that holds all instances of 'TimeLabel' elements defined in a 'BehaviourDescription' of a 'TestDescription'. Its mere purpose is to enable the definition of functions over time labels; some of them are predefined (see clause 10.5.2).

No assumptions are made about how 'TimeLabelType' is implemented in an underlying concrete type system.

10.3 Predefined Instances of 'SimpleDataInstance' Element

10.3.1 true

The predefined 'SimpleDataInstance' 'true' shall be associated with the 'SimpleDataType' 'Boolean' (see clause 10.2.1). It denotes one of the two truth values with the usual meaning.

10.3.2 false

The predefined 'SimpleDataInstance' 'false' shall be associated with the 'SimpleDataType' 'Boolean' (see clause 10.2.1). It denotes one of the two truth values with the usual meaning.

10.3.3 pass

The predefined 'SimpleDataInstance' 'pass' shall be associated with the predefined 'SimpleDataType' 'Verdict' (see clause 10.2.2). It denotes the valid behaviour of the SUT as observed by the tester in correspondence to the definition in ISO/IEC 9646-1 [7].

10.3.4 fail

The predefined 'SimpleDataInstance' 'fail' shall be associated with the predefined 'SimpleDataType' 'Verdict' (see clause 10.2.2). It denotes the invalid behaviour of the SUT as observed by the tester in correspondence to the definition in ISO/IEC 9646-1 [7].

10.3.5 inconclusive

The predefined 'SimpleDataInstance' 'inconclusive' shall be associated with the predefined 'SimpleDataType' 'Verdict' (see clause 10.2.2). It denotes behaviour of the SUT as observed by the tester in cases when neither 'pass' nor 'fail' verdict can be given in correspondence to the definition in ISO/IEC 9646-1 [7].

10.4 Predefined Instances of 'Time' Element

10.4.1 Second

The predefined instance 'Second' of the 'Time' element denotes a data type that represents the physical quantity time measured in seconds. Values of this time data type, i.e. instances of 'SimpleDataInstance', denote a measurement of time with the physical unit second.

No assumptions are made about how 'Second' is implemented in an underlying concrete type system.

10.5 Predefined Instances of the 'Function' Element

10.5.1 Overview

In this clause, the predefined functions are provided in one of the following two syntax forms:

- Prefix notation: <function name>: <parameter type>, <parameter type>, ... → <return type>

- Infix notation: _<function name>_: <parameter type>, <parameter type> → <return type>

The <parameter type> and <return type> names from above refer to (predefined) instance names of meta-model elements. If arbitrary instances are supported, the function **instanceOf**(<element>) shall provide such an arbitrary instance of the given meta-model element.

No assumptions are made about how these functions are implemented in an underlying concrete type system.

10.5.2 Functions of Return Type 'Boolean'

The following functions of return type 'Boolean' shall be predefined:

- _==_: **instanceOf**(DataInstance), **instanceOf**(DataInstance) → Boolean
  Denotes equality of any two data instances of arbitrary, but same data type.
- _!=_: **instanceOf**(DataInstance), **instanceOf**(DataInstance) → Boolean
  Denotes inequality of any two data instances of arbitrary, but same data type.
- _and_: Boolean, Boolean → Boolean
  Denotes the standard logical AND operation.
- _or_: Boolean, Boolean → Boolean
  Denotes the standard logical OR operation.
- not: Boolean → Boolean
  Denotes the standard logical NOT operation.

10.5.3 Functions of Return Type 'TimeLabelType'

The following functions of return type 'TimeLabelType' shall be predefined. Their purpose is to identify unique occurrences of a time label if it occurs in an iterative behaviour, e.g. within bounded or unbounded loops. All functions listed below will return the time label itself if they are applied to time labels that are outside of iterative behaviour:

- first: TimeLabelType → TimeLabelType
  Returns the first occurrence of a time label in an iterative behaviour.
- last: TimeLabelType → TimeLabelType
  Returns the last occurrence of a time label in an iterative behaviour.
- prev: TimeLabelType → TimeLabelType
  Returns the occurrence of a time label in the previous iteration. The previous occurrence of a time label in the first iteration shall be equal to the first occurrence of this time label.

10.5.4 Functions of Return Type of Instance of 'Time'

The following functions of return type of instance of the 'Time' meta-model element shall be predefined:

- _+_: **instanceOf**(Time), **instanceOf**(Time) → **instanceOf**(Time)
  Returns the sum of two time values of the same time data type, i.e. all parameters of the function definition shall refer to the same instance of the 'Time' element as data type.
- obs: TimeLabelType → **instanceOf**(Time)
  Returns the timestamp of a time label attached to an atomic behaviour instance, i.e. the time point when this behavioural activity is observed. The timestamp is returned as a time value of the given time data type.
- span: TimeLabelType , TimeLabelType → **instanceOf**(Time)
  Returns the time span between two time labels attached to two atomic behaviour instances, i.e. the elapsed time between the two behavioural activities. The time span is returned as a value of the given time data type.

Annex A (informative): Technical Representation of the TDL Meta-Model

The technical representation of the TDL meta-model is included as an electronic attachment es_20311901v010301p0.zip which accompanies the present document. The purpose of this annex is to serve as a possible starting point for implementing the TDL meta-model conforming to the present document. See the readme contained in the zip file for details.

Annex B (informative): Examples of a TDL Concrete Syntax

B.1 Introduction

The applicability of the TDL meta-model that is described in the main part of the present document depends on the availability of TDL concrete syntaxes that implement the meta-model (abstract syntax). Such a TDL concrete syntax can then be used by end users to write TDL specifications. Though a concrete syntax will be based on the TDL meta-model, it can implement only parts of the meta-model if certain TDL features are not necessary to handle a user's needs.

This annex illustrates an example of a possible TDL concrete syntax in a textual format that supports all features of the TDL meta-model, called "TDLan". Three examples are outlined below; two examples translated from existing test descriptions taken from ETSI TS 136 523-1 [i.2] and ETSI TS 186 011-2 [i.3] as well as an example illustrating some of the TDL data parameterization and mapping concepts. The examples are accompanied by a complete reference description of the textual syntax of TDLan given in EBNF.

B.2 A 3GPP Conformance Example in Textual Syntax

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

```
//Translated from [i.5], Section 7.1.3.
TDLan Specification Layer_2_DL_SCH_Data_Transfer {
  //Procedures carried out by a component of a test configuration
  //or 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 TS 36.508 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
  Annotation PROCEDURE ;      //Informal textual description of a test step
  Annotation PRECONDITION ;   //Identify pre-condition behaviour
  Annotation PREAMBLE ;       //Identify preamble behaviour.

  //Test objectives (copied verbatim from source document)
  Test Objective TP1 {
    from : "36523-1-a20_s07_01.doc::7.1.3.1.1 (1)" ;
    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 }
                  }" ;
  }
  Test Objective TP2 {
    from : "36523-1-a20_s07_01.doc::7.1.3.1.1 (2)" ;
```

```
  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 }
           }" ;
}

//Relevant data definitions
Type PDU;
PDU mac_pdu ;

Type ACK ;
ACK harq_ack ;

Type C_RNTI;
C_RNTI ue;
C_RNTI unknown;

Type PDCCH (optional c_rnti of type C_RNTI);
PDCCH pdcch;

Type CONFIGURATION;
CONFIGURATION RRCConnectionReconfiguration ;

//User-defined time units
Time SECONDS;
SECONDS five;

//Gate type definitions
Gate Type defaultGT accepts ACK, PDU, PDCCH, C_RNTI, CONFIGURATION ;

//Component type definitions
Component Type defaultCT having {
  gate g of type defaultGT;
}

//Test configuration definition
Test Configuration defaultTC {
  create Tester SS of type defaultCT;
  create SUT UE of type defaultCT ;
  connect UE.g to SS.g ;
}

//Test description definition
Test Description TD_7_1_3_1 uses configuration defaultTC {
  //Pre-conditions and preamble from the source document
  perform action preCondition with { PRECONDITION ; } ;
  perform action preamble  with { PREAMBLE ; } ;

  //Test sequence
  SS.g sends pdcch (c_rnti=ue) to UE.g with {
    STEP : "1" ;
    PROCEDURE : "SS transmits a downlink assignment
           including the C-RNTI assigned to
           the UE" ;
  } ;
  SS.g sends mac_pdu to UE.g with {
    STEP : "2" ;
    PROCEDURE : "SS transmits in the indicated
           downlink assignment a RLC PDU in
           a MAC PDU" ;
  } ;
  UE.g sends harq_ack to SS.g with {
    STEP : "3" ;
    PROCEDURE : "Check: Does the UE transmit an
           HARQ ACK on PUCCH?" ;
    test objectives : TP1 ;
  } ;
  set verdict to PASS ;
  SS.g sends pdcch (c_rnti=unknown) to UE.g with {
    STEP : "4" ;
    PROCEDURE : "SS transmits a downlink assignment
             to including a C-RNTI different from
           the assigned to the UE" ;
  } ;
```

```
    SS.g sends mac_pdu to UE.g with {
      STEP : "5" ;
      PROCEDURE : "SS transmits in the indicated
             downlink assignment a RLC PDU in
             a MAC PDU" ;
    } ;

    //Interpolated original step 6 into an alternative behaviour,
    //covering both the incorrect and the correct behaviours of the UE
    alternatively {
      UE.g sends harq_ack to SS.g ;
        set verdict to FAIL ;
    } or {
        gate SS.g is quiet for five ;
      set verdict to PASS ;
    } with {
            STEP : "6" ;
            PROCEDURE : "Check: Does the UE send any HARQ ACK
                       on PUCCH?" ;
            test objectives : TP2 ;
    }
  } with {
    Note  : "Note 1: For TDD, the timing of ACK/NACK is not
         constant as FDD, see Table 10.1-1 of TS 36.213." ;
  }
} with {
  Note : "Taken from 3GPP TS 36.523-1 V10.2.0 (2012-09)" ;
  TITLE : "Correct handling of DL assignment / Dynamic case" ;
}
```

B.3 An IMS Interoperability Example in Textual Syntax

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

```
//Translated from [i.6], Section 4.5.1.
TDLan Specification IMS_NNI_General_Capabilities {
    //Procedures carried out by a component of a test configuration
    //or an actor during test execution
    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 [16]
                 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)
    Test Objective TP_IMS_4002_1 {
        //Location in source document
        from : "ts_18601102v030101p.pdf::4.5.1.1 (CC 1)" ;
        //Further reference to another document
        from : "ETSI TS 124 229 [1], clause 4.2A, paragraph 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 }
                       }" ;
}
Test Objective UC_05_I {
    //Only a reference to corresponding section in the source document
    from : "ts_18601102v030101p.pdf::4.4.4.2" ;
}

//Relevant data definitions
Type MSG (optional TCP of type CONTENT);
MSG MESSAGE ;
MSG DING ;
MSG DELIVERY_REPORT ;
MSG M_200_OK

Type CONTENT ;
CONTENT tcp;

Time seconds;
seconds default_timeout;

//Gate type definitions.
Gate Type defaultGT accepts MSG, CONTENT ;

//Component type definitions
//In this case they may also be reduced to a single component type
Component Type USER having {
    gate g of type defaultGT ;
}
Component Type UE having {
    gate g of type defaultGT ;
}
Component Type IMS having {
    gate g of type defaultGT ;
}
Component Type IBCF having {
    gate g of type defaultGT ;
}

//Test configuration definition
Test Configuration CF_INT_CALL {
  create Tester USER_A of type USER;
  create Tester UE_A of type UE;
  create Tester IMS_A of type IMS;
  create Tester IBCF_A of type IBCF;
  create Tester IBCF_B of type IBCF;
  create SUT IMS_B of type IMS;
  create Tester UE_B of type UE;
  create Tester USER_B of type USER;
    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 configuration CF_INT_CALL {
        //Pre-conditions from the source document
        perform action preConditions with { PRECONDITION ; };

        //Test sequence
        USER_A.g sends MESSAGE to UE_A.g with { STEP : "1" ; } ;
        UE_A.g sends MESSAGE to IMS_A.g with { STEP : "2" ; } ;
        IMS_A.g sends MESSAGE to IBCF_A.g with { STEP : "3" ; } ;
        IBCF_A.g sends MESSAGE to IBCF_B.g with { STEP : "4" ; } ;
        IBCF_B.g sends MESSAGE (TCP = tcp) to IMS_B.g with { STEP : "5" ; } ;
        IMS_B.g sends MESSAGE to UE_B.g with { STEP : "6" ; } ;
        UE_B.g sends DING to USER_B.g with { STEP : "7" ; } ;
        UE_B.g sends M_200_OK to IMS_B.g with { STEP : "8" ; } ;
        IMS_B.g sends M_200_OK to IBCF_B.g with { STEP : "9" ; } ;
        IBCF_B.g sends M_200_OK to IBCF_A.g with { STEP : "10" ; } ;
        IBCF_A.g sends M_200_OK to IMS_A.g with { STEP : "11" ; } ;
```

```
            IMS_A.g sends M_200_OK to UE_A.g with { STEP : "12" ; } ;
            alternatively {
                UE_A.g sends DELIVERY_REPORT to USER_A.g with { STEP : "13" ; } ;
            } or {
                gate USER_A.g is quiet for default_timeout;
            }
        } with {
            SUMMARY : "IMS network shall support SIP messages greater than
                            1 500 bytes" ;
        }
} with {
    Note : "Taken from ETSI TS 186 011-2 [i.3] V3.1.1 (2011-06)" ;
    TITLE : "SIP messages longer than 1 500 bytes" ;
}
```

B.4 An Example Demonstrating TDL Data Concepts

This example describes some of the concepts related to data and data mapping in TDL by means of the proposed 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 using a similar test configuration as defined for the first example in clause B.2, and an alternative realization where data flow is expressed with variables.

```
//A manually constructed example illustrating the data mapping concepts
TDLan Specification DataExample {
    //User-defined verdicts
    //Alternatively the predefined verdicts may be used as well
    Type Verdict ;
    Verdict PASS ;
    Verdict FAIL ;

    //Test objectives
    Test Objective CHECK_SESSION_ID_IS_MAINTAINED {
        //Only a description
        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 ;

  Type MSG (optional session of type SESSION_ID);
  MSG REQUEST_SESSION_ID(session = omit);
  MSG RESPONSE(session = ?);
  MSG MESSAGE(session = ?);

  //Data mappings

  //Load resource.ttcn3
  Use "resource.ttcn3" as TTCN_MAPPING ;

  //Map types and instances to TTCN-3 records and templates, respectively
  //(located in the used TTCN-3 file)
  Map MSG to "record_message" in TTCN_MAPPING as MSG_mapping with {
    session mapped to "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 ;

  //Use a runtime URI for dynamic data available at runtime, such as
  //session IDs
  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
  Gate Type defaultGT accepts MSG , SESSION_ID;
```

```
//Component type definitions
Component Type defaultCT having {
  gate g of type defaultGT ;
}

//Test configuration definition
Test Configuration defaultTC {
  create SUT UE of type defaultCT;
  create Tester SS of type defaultCT;
  connect SS.g to UE.g ;
}

//Test description definition
Test Description exampleTD uses configuration defaultTC {
  //Tester requests a session id
  SS.g sends REQUEST_SESSION_ID to UE.g ;
      //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 ;
      //Tester sends a message with the session id
      //from the runtime URI
      SS.g sends MESSAGE (session=SESSION_ID_1) to UE.g ;

      alternatively {
          //SUT responds with the same session id
          UE.g sends RESPONSE (session=SESSION_ID_1) to SS.g ;
          set verdict to PASS;
      } or {
          //SUT responds with a new session id
          UE.g sends RESPONSE (session=SESSION_ID_2) to SS.g ;
          set verdict to FAIL;
      } with {
          test objectives : CHECK_SESSION_ID_IS_MAINTAINED ;
      }
}

//Alternative approach with variables

//Component type definitions
Component Type defaultCTwithVariable having {
  variable v of type MSG;
  gate g of type defaultGT ;
}

  //Test configuration definition
Test Configuration defaultTCwithVariables {
  create SUT UE of type defaultCT;
  create Tester SS of type defaultCTwithVariable;
  connect SS.g to UE.g ;
}

Test Description exampleTD uses configuration defaultTC {
  //Tester requests a session id
  SS.g sends REQUEST_SESSION_ID to UE.g ;

      //SUT responds with a response message containing a session ID
      //The response could contain any of the known session IDs
      //The received response is stored in the variable v of the SS
      UE.g sends RESPONSE to SS.g where it is assigned to v;

      //Tester sends a message with the session ID
      //from the response stored in the variable v of the SS
      SS.g sends MESSAGE(session=SS->v.session) to UE.g ;

      alternatively {
          //SUT responds with the same session ID that is stored in
          //the variable v of the SS from the previous response
          UE.g sends RESPONSE(session=SS->v.session) to SS.g ;
          set verdict to PASS;
      } or {
          //SUT responds with a any session ID, including the one from the
          //previous response stored in v. The ordering of evaluation will
          //always select the first alternative in that case. Alternatively
          //a function can be defined and called that checks explicitly that
          //a the specific session ID from the previous response stored in v
          //is not received e.g.
          // UE.g sends RESPONSE(session=not(SS->v.session)) to SS.g;
```

```
            UE.g sends RESPONSE to SS.g ;
            set verdict to FAIL;
        } with {
            test objectives : CHECK_SESSION_ID_IS_MAINTAINED ;
        }
    }
}
```

B.5 TDL Textual Syntax Reference

B.5.1 Conventions for the TDLan Syntax Definition

This annex describes the grammar of the used concrete textual syntax in the Extended Backus-Naur Form (EBNF) notation. The EBNF representation is generated from a reference implementation of the TDL meta-model. The EBNF representation can be used either as a concrete syntax reference for TDL end users or as input to a parser generator tool. Table B.1 defines the syntactic conventions used in the definition of the EBNF rules. To distinguish this concrete textual syntax from other possible concrete textual syntax representations, it is referred to as "TDLan". This proposed syntax is complete in the sense that it covers the whole TDL meta-model.

Table B.1: Syntax definition conventions used

::=	is defined to be
abc	the non-terminal symbol abc
abc xyz	abc followed by xyz
abc | xyz	alternative (abc or xyz)
[abc]	0 or 1 instance of abc
{abc}+	1 or more instances of abc
{abc}	0 or more instances of abc
'a'-'z'	all characters from a to z
(...)	denotes a textual grouping
'abc'	the terminal symbol abc
;	production terminator
\	the escape character

B.5.2 TDL Textual Syntax EBNF Production Rules

```
              TDLSpec  ::=  'TDLan Specification' EString '{' [ ElementImport { ElementImport
                            } ] [ PackageableElement { PackageableElement } ] [ Package {
                            Package } ] '}' [ 'with' '{' [ Comment { Comment } ] [ Annotation
                            { Annotation } ] '}' ] ;
               Action  ::=  Action_Impl | Function ;
      ActionReference  ::=  'perform' 'action' EString [ '(' DataUse { ',' DataUse } ')' ] [
                            'on' EString ] [ 'with' '{' [ Comment { Comment } ] [ Annotation {
                            Annotation } ] [ 'test objectives' ':' EString { ',' EString } ';'
                            ] [ 'name' EString ] [ 'time' 'label' TimeLabel ] [ 'time'
                            'constraints' ':' TimeConstraint { ',' TimeConstraint } ';' ] '}'
                            ] ';' ;
          Action_Impl  ::=  'Action' EString [ '(' FormalParameter { ',' FormalParameter } ')'
                            ] [ ':' String0 ] [ 'with' '{' [ Comment { Comment } ] [
                            Annotation { Annotation } ] '}' ] ';' ;
 AlternativeBehaviour  ::=  'alternatively' Block { 'or' Block } [ 'with' '{' [ Comment {
                            Comment } ] [ Annotation { Annotation } ] [ 'test objectives' ':'
                            EString { ',' EString } ';' ] [ 'name' EString ] [
                            PeriodicBehaviour { PeriodicBehaviour } ] [ ExceptionalBehaviour {
                            ExceptionalBehaviour } ] '}' ] ;
           Annotation  ::=  EString [ ':' String0 ] [ 'with' '{' [ Comment { Comment } ] [
                            Annotation { Annotation } ] EString '}' ] ';' ;
       AnnotationType  ::=  'Annotation' EString [ 'with' '{' [ Comment { Comment } ] [
                            Annotation { Annotation } ] '}' ] ';' ;
       AnyValueOrOmit  ::=  '*' [ 'with' '{' [ 'reduction' '(' EString { ',' EString } ')' ] [
                            Comment { Comment } ] [ Annotation { Annotation } ] [ 'name'
                            EString ] '}' ] ;
             AnyValue  ::=  '?' [ ':' EString ] [ 'with' '{' [ 'reduction' '(' EString { ','
                            EString } ')' ] [ Comment { Comment } ] [ Annotation { Annotation
                            } ] '}' ] ;
```

```
        ParameterBinding  ::=  EString '=' DataUse [ 'with' '{' [ Comment { Comment } ] [
                               Annotation { Annotation } ] [ 'name' EString ] '}' ] ;
               Assertion  ::=  'assert' DataUse [ 'otherwise' 'set verdict' 'to' DataUse ] [
                               'with' '{' [ Comment { Comment } ] [ Annotation { Annotation } ] [
                               'test objectives' ':' EString { ',' EString } ';' ] [ 'name'
                               EString ] [ 'time' 'label' TimeLabel ] [ 'time' 'constraints' ':'
                               TimeConstraint { ',' TimeConstraint } ';' ] '}' ] ';' ;
              Assignment  ::=  [ EString '->' ] EString '=' DataUse [ 'with' '{' [ Comment {
                               Comment } ] [ Annotation { Annotation } ] [ 'test objectives' ':'
                               EString { ',' EString } ';' ] [ 'name' EString ] [ 'time' 'label'
                               TimeLabel ] [ 'time' 'constraints' ':' TimeConstraint { ','
                               TimeConstraint } ';' ] '}' ] ';' ;
               Behaviour  ::=  TimerStart
                                | TimerStop
                                | TimeOut
                                | Wait
                                | Quiescence
                                | PeriodicBehaviour
                                | AlternativeBehaviour
                                | ParallelBehaviour
                                | BoundedLoopBehaviour
                                | UnboundedLoopBehaviour
                                | ConditionalBehaviour
                                | CompoundBehaviour
                                | DefaultBehaviour
                                | InterruptBehaviour
                                | VerdictAssignment
                                | Assertion
                                | Stop
                                | Break
                                | Assignment
                                | InlineAction
                                | ActionReference
                                | TestDescriptionReference
                                | Interaction ;
    BehaviourDescription  ::=  Behaviour [ 'with' '{' [ Comment { Comment } ] [ Annotation {
                               Annotation } ] [ 'name' EString ] '}' ] ;
                   Block  ::=  [ '[' DataUse ']' ] '{' [ Comment { Comment } ] [ Annotation {
                               Annotation } ] [ 'name' EString ] Behaviour { Behaviour } '}' ;
                 Boolean  ::=  'true' | 'false' ;
    BoundedLoopBehaviour  ::=  'repeat' DataUse 'times' Block [ 'with' '{' [ Comment { Comment }
                               ] [ Annotation { Annotation } ] [ 'test objectives' ':' EString {
                               ',' EString } ';' ] [ 'name' EString ] [ PeriodicBehaviour {
                               PeriodicBehaviour } ] [ ExceptionalBehaviour {
                               ExceptionalBehaviour } ] '}' ] ;
                   Break  ::=  'break' [ 'with' '{' [ Comment { Comment } ] [ Annotation {
                               Annotation } ] [ 'test objectives' ':' EString { ',' EString } ';'
                               ] [ 'name' EString ] [ 'time' 'label' TimeLabel ] [ 'time'
                               'constraints' ':' TimeConstraint { ',' TimeConstraint } ';' ] '}'
                               ] ';' ;
                 Comment  ::=  'Note' EString ':' String0 [ 'with' '{' [ Comment { Comment } ] [
                               Annotation { Annotation } ] '}' ] ';' ;
       ComponentInstance  ::=  'create' ComponentInstanceRole EString 'of type' EString [ 'with'
                               '{' [ Comment { Comment } ] [ Annotation { Annotation } ] '}' ]
                               ';' ;
ComponentInstanceBinding  ::=  'bind' EString 'to' EString [ 'with' '{' [ Comment { Comment } ] [
                               Annotation { Annotation } ] [ 'name' EString ] '}' ] ;
           ComponentType  ::=  'Component Type' EString 'having' '{' { Timer } { Variable } {
                               GateInstance } '}' [ 'with' '{' [ Comment { Comment } ] [
                               Annotation { Annotation } ] '}' ] ;
       CompoundBehaviour  ::=  Block [ 'with' '{' [ Comment { Comment } ] [ Annotation {
                               Annotation } ] [ 'test objectives' ':' EString { ',' EString } ';'
                               ] [ 'name' EString ] [ PeriodicBehaviour { PeriodicBehaviour } ] [
                               ExceptionalBehaviour { ExceptionalBehaviour } ] '}' ] ;
    ConditionalBehaviour  ::=  'if' Block [ ( ( 'else' Block ) ) | ( { 'else' 'if' Block } | (
                               'else' Block ) ) ] [ 'with' '{' [ Comment { Comment } ] [
                               Annotation { Annotation } ] [ 'test objectives' ':' EString { ','
                               EString } ';' ] [ 'name' EString ] [ PeriodicBehaviour {
                               PeriodicBehaviour } ] [ ExceptionalBehaviour {
                               ExceptionalBehaviour } ] '}' ] ;
              Connection  ::=  'connect' GateReference 'to' GateReference [ 'with' '{' [ Comment
                               { Comment } ] [ Annotation { Annotation } ] [ 'as' EString ] '}' ]
                               ';' ;
      DataElementMapping  ::=  'Map' EString [ 'to' String0 ] 'in' EString [ 'as' EString ] [
                               'with' '{' { ParameterMapping } [ Comment { Comment } ] [
                               Annotation { Annotation } ] '}' ] ';' ;
            DataInstance  ::=  SimpleDataInstance_Impl | StructuredDataInstance ;
```

```
     DataInstanceUse  ::=  ( ( ( NumberAsIdentifier | ( EString | UnassignedMemberTreatment )
                           )
                            | [ ( '(' | ParameterBinding | { ',' ParameterBinding } | ')' ) |
                           { '.' | EString } ] ) | ( 'new'
                            | EString
                            | UnassignedMemberTreatment
                            | ( '(' ParameterBinding { ',' ParameterBinding } ')' ) ) | (
                           UnassignedMemberTreatment
                            | ( '(' ParameterBinding { ',' ParameterBinding } ')' ) ) ) [
                           'with' '{' [ 'name' EString ] [ Comment { Comment } ] [ Annotation
                           { Annotation } ] '}' ] ;
 DataResourceMapping  ::=  'Use' String0 [ 'as' EString ] [ 'with' '{' [ Comment { Comment }
                           ] [ Annotation { Annotation } ] '}' ] ';' ;
            DataType  ::=  SimpleDataType_Impl
                            | StructuredDataType
                            | Time ;
             DataUse  ::=  DataInstanceUse
                            | FunctionCall
                            | FormalParameterUse
                            | TimeLabelUse
                            | VariableUse
                            | AnyValue
                            | AnyValueOrOmit
                            | OmitValue ;
    DefaultBehaviour  ::=  'default' [ 'on' EString ] Block [ 'with' '{' [ Comment { Comment
                           } ] [ Annotation { Annotation } ] [ 'test objectives' ':' EString
                           { ',' EString } ';' ] [ 'name' EString ] '}' ] ;
             EString  ::=  ID ;
          EStringDot  ::=  ID '.' ID ;
       ElementImport  ::=  'Import' ( 'all' | ( EString | { ',' EString } ) ) 'from' EString
                           [ 'with' '{' [ Comment { Comment } ] [ Annotation { Annotation } ]
                           EString '}' ] ';' ;
ExceptionalBehaviour  ::=  DefaultBehaviour | InterruptBehaviour ;
            Function  ::=  'Function' EString '(' [ FormalParameter { ',' FormalParameter } ]
                           ')' 'returns' EString [ ':' String0 ] [ 'with' '{' [ Comment {
                           Comment } ] [ Annotation { Annotation } ] '}' ] ';' ;
        FunctionCall  ::=  'instance' 'returned' 'from' EString '(' [ ParameterBinding { ','
                           ParameterBinding } ] ')' { '.' EString } [ 'with' '{' [ 'name'
                           EString ] [ Comment { Comment } ] [ Annotation { Annotation } ]
                           '}' ] ;
        GateInstance  ::=  'gate' EString 'of type' EString [ 'with' '{' [ Comment { Comment
                           } ] [ Annotation { Annotation } ] '}' ] ';' ;
       GateReference  ::=  EString '.' EString [ 'with' '{' [ Comment { Comment } ] [
                           Annotation { Annotation } ] [ 'name' EString ] '}' ] ;
            GateType  ::=  'Gate Type' EString 'accepts' EString { ',' EString } [ 'with' '{'
                           [ Comment { Comment } ] [ Annotation { Annotation } ] '}' ] ';' ;
        InlineAction  ::=  'perform' 'action' ':' String0 [ 'on' EString ] [ 'with' '{' [
                           Comment { Comment } ] [ Annotation { Annotation } ] [ 'test
                           objectives' ':' EString { ',' EString } ';' ] [ 'name' EString ] [
                           'time' 'label' TimeLabel ] [ 'time' 'constraints' ':'
                           TimeConstraint { ',' TimeConstraint } ';' ] '}' ] ';' ;
         Interaction  ::=  EStringDot ( 'sends' | 'triggers' ) DataUse 'to' Target { ','
                           Target } [ 'with' '{' [ Comment { Comment } ] [ Annotation {
                           Annotation } ] [ 'test objectives' ':' EString { ',' EString } ';'
                           ] [ 'name' EString ] [ 'time' 'label' TimeLabel ] [ 'time'
                           'constraints' ':' TimeConstraint { ',' TimeConstraint } ';' ] '}'
                           ] ';' ;
             Trigger  ::=  'triggers' ;
  InterruptBehaviour  ::=  'interrupt' [ 'on' EString ] Block [ 'with' '{' [ Comment {
                           Comment } ] [ Annotation { Annotation } ] [ 'test objectives' ':'
                           EString { ',' EString } ';' ] [ 'name' EString ] '}' ] ;
 MappableDataElement  ::=  SimpleDataType_Impl
                            | SimpleDataInstance_Impl
                            | StructuredDataType
                            | StructuredDataInstance
                            | Action_Impl
                            | Function
                            | Time ;
              Member  ::=  [ 'optional' ] EString 'of type' EString [ 'with' '{' [ Comment {
                           Comment } ] [ Annotation { Annotation } ] '}' ] ;
            Optional  ::=  'optional' | 'mandatory' ;
    MemberAssignment  ::=  EString '=' StaticDataUse [ 'with' '{' [ Comment { Comment } ] [
                           Annotation { Annotation } ] [ 'name' EString ] '}' ] ;
    ParameterMapping  ::=  EString [ 'mapped' 'to' String0 ] [ 'as' EString ] [ 'with' '{' [
                           Comment { Comment } ] [ Annotation { Annotation } ] '}' ] ';' ;
```

```
           OmitValue  ::=  'omit' [ 'with' '{' [ 'argument' '{' ParameterBinding { ','
                           ParameterBinding } '}' ] [ 'reduction' '(' EString { ',' EString }
                           ')' ] [ Comment { Comment } ] [ Annotation { Annotation } ] [
                           'name' EString ] '}' ] ;
             Package  ::=  'Package' EString '{' [ ElementImport { ElementImport } ] [
                           PackageableElement { PackageableElement } ] [ Package { Package }
                           ] '}' [ 'with' '{' [ Comment { Comment } ] [ Annotation {
                           Annotation } ] '}' ] ;
  PackageableElement  ::=  AnnotationType
                            | TestObjective
                            | DataResourceMapping
                            | DataElementMapping
                            | SimpleDataType_Impl
                            | SimpleDataInstance_Impl
                            | StructuredDataType
                            | StructuredDataInstance
                            | Action_Impl
                            | Function
                            | ComponentType
                            | GateType
                            | Time
                            | TestConfiguration
                            | TestDescription ;
  ParallelBehaviour   ::=  'run' Block { 'in' 'parallel' 'to' Block } [ 'with' '{' [ Comment
                           { Comment } ] [ Annotation { Annotation } ] [ 'test objectives'
                           ':' EString { ',' EString } ';' ] [ 'name' EString ] [
                           PeriodicBehaviour { PeriodicBehaviour } ] [ ExceptionalBehaviour {
                           ExceptionalBehaviour } ] '}' ] ;
           Parameter  ::=  Member | FormalParameter ;
     FormalParameter  ::=  EString 'of type' EString [ 'with' '{' [ Comment { Comment } ] [
                           Annotation { Annotation } ] '}' ] ;
        TimeLabelUse  ::=  'time' 'label' EString [ 'with' '{' [ 'argument' '{'
                           ParameterBinding { ',' ParameterBinding } '}' ] [ 'reduction' '('
                           EString { ',' EString } ')' ] [ 'name' EString ] [ Comment {
                           Comment } ] [ Annotation { Annotation } ] '}' ] ;
  FormalParameterUse  ::=  'parameter' EString [ '(' ParameterBinding { ',' ParameterBinding
                           } ')' ] { '.' EString } [ 'with' '{' [ 'name' EString ] [ Comment
                           { Comment } ] [ Annotation { Annotation } ] '}' ] ;
   PeriodicBehaviour  ::=  'every' DataUse Block [ 'with' '{' [ Comment { Comment } ] [
                           Annotation { Annotation } ] [ 'test objectives' ':' EString { ','
                           EString } ';' ] [ 'name' EString ] '}' ] ;
          Quiescence  ::=  ( ( 'component' | EString ) | ( 'gate' | EStringDot ) ) 'is'
                           'quiet' 'for' DataUse [ 'with' '{' [ Comment { Comment } ] [
                           Annotation { Annotation } ] [ 'test objectives' ':' EString { ','
                           EString } ';' ] [ 'name' EString ] [ 'time' 'label' TimeLabel ] [
                           'time' 'constraints' ':' TimeConstraint { ',' TimeConstraint } ';'
                           ] '}' ] ';' ;
SimpleDataInstance_Impl  ::=  EString ( EString | NumberAsIdentifier ) [ 'with' '{' [ Comment {
                           Comment } ] [ Annotation { Annotation } ] '}' ] ';' ;
 SimpleDataType_Impl  ::=  'Type' EString [ 'with' '{' [ Comment { Comment } ] [ Annotation {
                           Annotation } ] '}' ] ';' ;
       StaticDataUse  ::=  DataInstanceUse
                            | AnyValue
                            | AnyValueOrOmit
                            | OmitValue ;
                Stop  ::=  'terminate' [ 'with' '{' [ Comment { Comment } ] [ Annotation {
                           Annotation } ] [ 'test objectives' ':' EString { ',' EString } ';'
                           ] [ 'name' EString ] [ 'time' 'label' TimeLabel ] [ 'time'
                           'constraints' ':' TimeConstraint { ',' TimeConstraint } ';' ] '}'
                           ] ';' ;
             String0  ::=  STRING ;
StructuredDataInstance  ::=  EString EString UnassignedMemberTreatment '(' [ MemberAssignment {
                           ',' MemberAssignment } ] ')' [ 'with' '{' [ Comment { Comment } ]
                           [ Annotation { Annotation } ] '}' ] ';' ;
  StructuredDataType  ::=  'Type' EString [ '(' Member { ',' Member } ')' ] [ 'with' '{' [
                           Comment { Comment } ] [ Annotation { Annotation } ] '}' ] ';' ;
              Target  ::=  EStringDot [ 'where it is' 'assigned' 'to' EString ] [ 'with' '{'
                           [ Comment { Comment } ] [ Annotation { Annotation } ] [ 'name'
                           EString ] '}' ] ;
   TestConfiguration  ::=  'Test Configuration' EString '{' ComponentInstance {
                           ComponentInstance } Connection { Connection } '}' [ 'with' '{' [
                           Comment { Comment } ] [ Annotation { Annotation } ] '}' ] ;
     TestDescription  ::=  'Test Description' EString [ '(' FormalParameter { ','
                           FormalParameter } ')' ] 'uses' 'configuration' EString (
                           BehaviourDescription | ';' ) [ 'with' '{' [ Comment { Comment } ]
                           [ Annotation { Annotation } ] [ 'test objectives' ':' EString {
                           ',' EString } ';' ] '}' ] ;
```

```
TestDescriptionReference  ::=  'execute' EString [ '(' DataUse { ',' DataUse } ')' ] [ 'with' '{'
                               [ 'bindings' '{' ComponentInstanceBinding { ','
                               ComponentInstanceBinding } '}' ] [ Comment { Comment } ] [
                               Annotation { Annotation } ] [ 'test objectives' ':' EString { ','
                               EString } ';' ] [ 'name' EString ] [ 'time' 'label' TimeLabel ] [
                               'time' 'constraints' ':' TimeConstraint { ',' TimeConstraint } ';'
                               ] '}' ] ';' ;
           TestObjective  ::=  'Test Objective' EString '{' [ 'from' ':' String0 ';' { 'from' ':'
                               String0 ';' } ] [ 'description' ':' String0 ';' ] '}' [ 'with' '{'
                               [ Comment { Comment } ] [ Annotation { Annotation } ] '}' ] ;
                    Time  ::=  'Time' EString [ 'with' '{' [ Comment { Comment } ] [ Annotation {
                               Annotation } ] '}' ] ';' ;
          TimeConstraint  ::=  EString DataUse [ 'with' '{' [ Comment { Comment } ] [ Annotation
                               { Annotation } ] '}' ] ;
               TimeLabel  ::=  EString [ 'with' '{' [ Comment { Comment } ] [ Annotation {
                               Annotation } ] '}' ] ';' ;
                 TimeOut  ::=  EString '.' EString 'times' 'out' [ 'with' '{' [ Comment { Comment
                               } ] [ Annotation { Annotation } ] [ 'test objectives' ':' EString
                               { ',' EString } ';' ] [ 'name' EString ] [ 'time' 'label'
                               TimeLabel ] [ 'time' 'constraints' ':' TimeConstraint { ','
                               TimeConstraint } ';' ] '}' ] ';' ;
                   Timer  ::=  'timer' EString [ 'with' '{' [ Comment { Comment } ] [ Annotation
                               { Annotation } ] '}' ] ';' ;
              TimerStart  ::=  'start' EString '.' EString 'for' DataUse [ 'with' '{' [ Comment {
                               Comment } ] [ Annotation { Annotation } ] [ 'test objectives' ':'
                               EString { ',' EString } ';' ] [ 'time' 'label' TimeLabel ] [
                               'time' 'constraints' ':' TimeConstraint { ',' TimeConstraint } ';'
                               ] [ 'name' EString ] '}' ] ';' ;
               TimerStop  ::=  'stop' EString '.' EString [ 'with' '{' [ Comment { Comment } ] [
                               Annotation { Annotation } ] [ 'test objectives' ':' EString { ','
                               EString } ';' ] [ 'name' EString ] [ 'time' 'label' TimeLabel ] [
                               'time' 'constraints' ':' TimeConstraint { ',' TimeConstraint } ';'
                               ] '}' ] ';' ;
  UnboundedLoopBehaviour  ::=  'repeat' Block [ 'with' '{' [ Comment { Comment } ] [ Annotation {
                               Annotation } ] [ 'test objectives' ':' EString { ',' EString } ';'
                               ] [ 'name' EString ] [ PeriodicBehaviour { PeriodicBehaviour } ] [
                               ExceptionalBehaviour { ExceptionalBehaviour } ] '}' ] ;
                Variable  ::=  'variable' EString 'of type' EString [ 'with' '{' [ Comment {
                               Comment } ] [ Annotation { Annotation } ] '}' ] ';' ;
             VariableUse  ::=  EString '->' EString [ '(' ParameterBinding { ',' ParameterBinding
                               } ')' ] { '.' EString } [ 'with' '{' [ 'name' EString ] [ Comment
                               { Comment } ] [ Annotation { Annotation } ] '}' ] ;
        PredefinedVerdict  ::=  'Verdict' ;
        VerdictAssignment  ::=  'set verdict' 'to' DataUse [ 'with' '{' [ Comment { Comment } ] [
                               Annotation { Annotation } ] [ 'test objectives' ':' EString { ','
                               EString } ';' ] [ 'name' EString ] [ 'time' 'label' TimeLabel ] [
                               'time' 'constraints' ':' TimeConstraint { ',' TimeConstraint } ';'
                               ] '}' ] ';' ;
                    Wait  ::=  ( 'component' EString ) 'waits' 'for' DataUse [ 'with' '{' [
                               Comment { Comment } ] [ Annotation { Annotation } ] [ 'test
                               objectives' ':' EString { ',' EString } ';' ] [ 'name' EString ] [
                               'time' 'label' TimeLabel ] [ 'time' 'constraints' ':'
                               TimeConstraint { ',' TimeConstraint } ';' ] '}' ] ';' ;
      NumberAsIdentifier  ::=  ['-'] INT [ '.' INT ] ;
   ComponentInstanceRole  ::=  ( 'SUT' | 'Tester' ) ;
UnassignedMemberTreatment ::=  ( '?' | '*' ) ;
                      ID  ::=  ( ['^'] ( 'a'-'z' | 'A'-'Z' | '_' ) { 'a'-'z' | 'A'-'Z' | '_' |
                               '0'-'9' } ) ;
                     INT  ::=  '0'-'9' ;
```

Annex C (normative): TDL - UML Mapping

C.0 Structure of UML Profile for TDL

This annex describes how concepts of the TDL meta model can be mapped to UML creating a UML Profile for TDL, called UP4TDL.

The stereotypes representing concepts from Foundation section of TDL metamodel are directly stored in the UP4TDL Profile, while other concepts are stored in various additional included Packages.

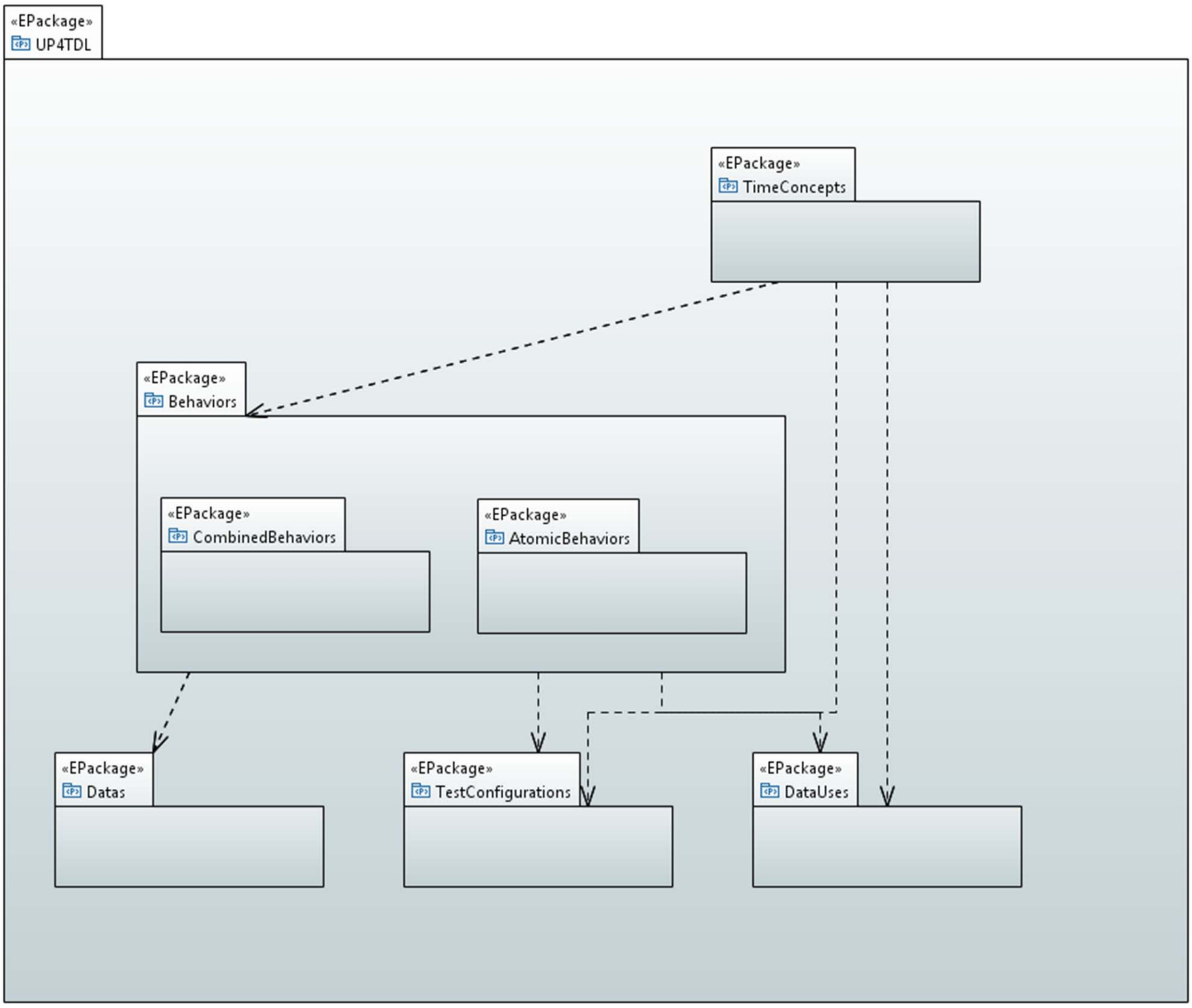

Figure C.1: Structure of UMLProfile4TDL

The following clauses describe the content of each package. The subchapters describe how the TDL metamodel elements can be mapped to UML.

C.1 Foundation

C.1.0 Overview

Most concepts of the Foundation Package are directly mapped to UML meta-classes. Exeptions are:

- TDL::Element : a stereotype is created for allowing elements to have Annotations;
- TDL::Annotations and TDL::TestObjective for which there is no equivalent concept in UML.

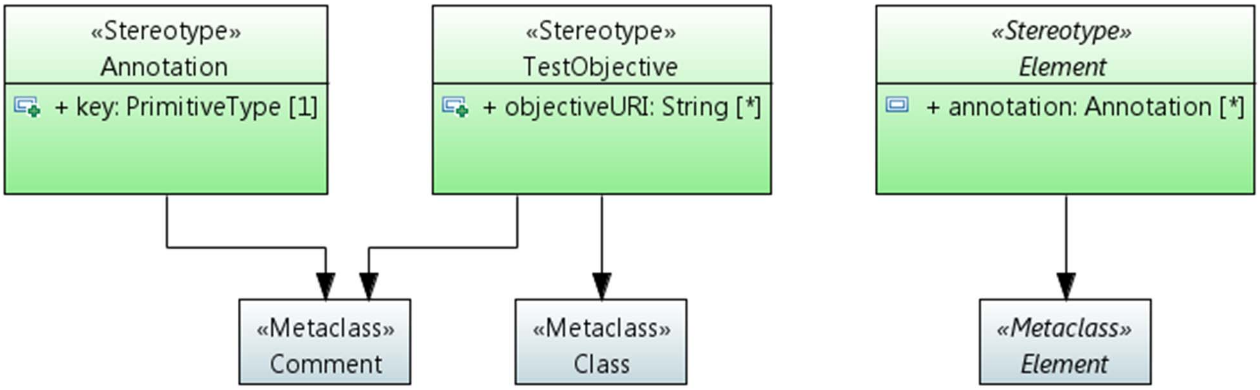

Figure C.1.1: Foundational language concepts

C.1.1 Element

Extended UML Meta-Class

- UML::Element

Generalization

None.

Properties

- <<UP4TDL::Element>>
- TDL::Element.comment := Computed as the set of Comment whose *annotatedElement* Property contains this element.
- TDL::Element. annotation := UP4TDL::Element.annotation : new (derived) property computed as the set of Comment with stereotype Annotation applied whose *annotatedElement* property contains this Element.

Constraints

None.

C.1.2 NamedElement

Extended UML Meta-Class

- Direct mapping without a stereotype to UML::NamedElement

Generalization

None.

Properties

- TDL::NamedElement.qualifiedName := UML::NamedElement.qualifiedName

Constraints

None.

C.1.3 PackageableElement

Extended UML Meta-Class

- Direct mapping without a stereotype to UML::PackageableElement

Generalization

None.

Properties

None.

Constraints

None.

C.1.4 Package

Extended UML Meta-Class

- Direct mapping without a stereotype to UML::Package

Generalization

None.

Properties

- TDL::Package.packagedElement:= UML::Package.packagedElement
- TDL::Package.import := UML::Package.elementImport
- TDL::Package.nestedPackage := UML::Package.nestedPackage

Constraints

None.

C.1.5 ElementImport

Extended UML Meta-Class

- Direct mapping without a stereotype to UML::ElementImport

Generalization

None.

Properties

- TDL::ElementImport is resolved in UML by a set of UML::ElementImport
- TDL::ElementImport.importedPackage := UML::ElementImport.importedElement.namespace
- TDL::ElementImport.importedElement := UML::ElementImport. importedElement

Constraints

None.

Model to Model transformation advice

- In UML, an ElementImport can import exactly one element. This implies that for one TDL::ElementImport, the equivalent model in UP4TDL can have several UML::ElementImport

C.1.6 Comment

Extended UML Meta-Class

- Direct mapping without a stereotype to UML::Comment

Generalization

- None

Properties

- TDL::Comment.commentedElement := UML::Comment.annotatedElement
- TDL::Comment.body := UML::Comment.body

Constraints

None.

C.1.7 Annotation

Extended UML Meta-Class

- UML::Comment

Generalization

None.

Properties

- <<UP4TDL::Annotation>>
- TDL::Annotation.key :=UP4TDL::Annotation.key (new property)
- TDL::Annotation.value := UP4TDL::Annotation.base_Comment.body

- TDL::Annotation.annotatedElement := UP4TDL::Annotation.base_Comment.annotatedElement

Constraints

None.

C.1.8 AnnotationType

Extended UML Meta-Class

- Direct mapping without a stereotype to UML::PrimitiveType

Generalization

None.

Properties

None.

Constraints

None.

C.1.9 TestObjective

Extended UML Meta-Class

- UML::Comment
- UML::Class

Generalization

None.

Properties

- <<UP4TDL::TestObjective>>
- TDL::TestObjective.description := UP4TDL::TestObjective.description
- TDL::TestObjective.objectiveURI := UP4 TDL::TestObjective.objectiveURI (new property)

Constraints

None.

C.2 Data

C.2.1 Data Definition

C.2.1.0 Overview

TDL Data Mapping-related concepts are mapped to stereotypes in UML.

TDL::Function concept is mapped to a stereotype to make it possible to add the constraint thata Function has exactly one return parameter.

All the other TDL Data Definitonconcepts are directly mapped to UML meta-classes without stereotypes.

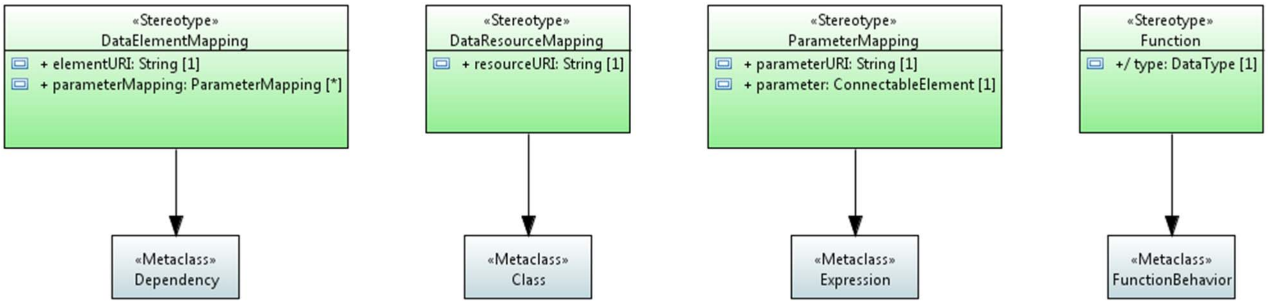

Figure C.2.1: Data Definition concepts

C.2.1.1 DataResourceMapping

Extended UML Meta-Class

- UML::Class

Generalization

None.

Properties

- <<UP4TDL::DataresourceMapping>>
- TDL::DataResourceMapping.resourceURI := UP4TDL::DataresourceMapping.reourceURI (new property)

Constraints

None.

C.2.1.2 MappableDataElement

This MetaClass shall not be mapped.

C.2.1.3 DataElementMapping

Extended UML Meta-Class

- UML::Dependency

Generalization

None.

Properties

- <<UP4TDL::DataElementMapping>>
- TDL::DataElementMapping.elementURI :=UP4TDL::DataElementMapping.elementURI (new property)

- TDL::DataElementMapping.dataResourceMapping := UP4TDL::DataElementMapping.base_AssociationClass.memberEnd[0]
- TDL::DataElementMapping.mappableDataElement := UP4TDL::DataElementMapping.base_AssociationClass.memberEnd[1]
- TDL::DataElementMapping.parameterMapping := UP4TDL::DataElementMapping.parameterMapping (new property)

Constraints

None.

C.2.1.4 ParameterMapping

Extended UML Meta-Class

- UML::Expression

Generalization

None.

Properties

- <<UP4TDL::ParameterMapping>>
- TDL::ParameterMapping.memberURI := UP4TDL::ParameterMapping.memberURI (new property)
- TDL::ParameterMapping.parameter := UP4TDL::ParameterMapping.parameter (new property)

Constraints

None.

C.2.1.5 DataType

Extended UML Meta-Class

- Direct mapping without a stereotype to UML::Classifier

Generalization

None.

Properties

None.

Constraints

None.

C.2.1.6 DataInstance

Extended UML Meta-Class

- Direct mapping without a stereotype to UML::InstanceSpecification

Generalization

None.

Properties

- TDL::DataInstance.dataType := UML::InstanceSpecification.classifier

Constraints

- TDL data instance shall have only one classifier

C.2.1.7 SimpleDataType

Extended UML Meta-Class

- Direct mapping without a stereotype to UML::PrimitiveType

Generalization

None.

Properties

None.

Constraints

None.

C.2.1.8 SimpleDataInstance

Extended UML Meta-Class

- Direct mapping without a stereotype to UML::InstanceSpecification

Generalization

None.

Properties

None.

Constraints

None.

Model to Model Transformation advice:

- TDL::SimpleDataInstance and TDL::StructuredDataInstance are mapped to the same UML concept : UML::InstanceSpecification. To know wether an InstanceSpecification represents a TDL::SimpleDataInstance or a TDL::StructuredDataInstance, one shall look at UML::InstanceSpecification.classifier. If it is a PrimitiveType, then the InstanceSpecification represents a TDL::SimpleDataInstance, otherwise, it represents a TDL::StructuredDataInstance

C.2.1.9 StructuredDataType

Extended UML Meta-Class

- Direct mapping without a stereotype to UML::DataType

Generalization

None.

Properties

- TDL::StructuredDataType.member := UML::DataType.ownedAttribute

Constraints

None.

C.2.1.10 Member

Extended UML Meta-Class

- Direct mapping without a stereotype to UML::Property

Generalization

None.

Properties

None.

Constraints

None.

Model to Model Transformation advice:

- TDL::Members correspond to Properties that are owned by a DataType

C.2.1.11 StructuredDataInstance

Extended UML Meta-Class

- Direct mapping without a stereotype to UML::InstanceSpecification

Generalization

None.

Properties

- TDL::StructuredDataInstance.memberAssignment:= UML::InstanceSpecification.slot

Constraints

None.

Model to Model Transformation advice:

- TDL::SimpleDataInstance and TDL::StructuredDataInstance are mapped to the same UML concept : UML::InstanceSpecification. To know wether an InstanceSpecification represents a TDL::SimpleDataInstance or a TDL::StructuredDataInstance, one shall look at UML::InstanceSpecification.classifier. If it is a PrimitiveType, then the InstanceSpecification represents a TDL::SimpleDataInstance, otherwise, it represents a TDL::StructuredDataInstance

C.2.1.12 MemberAssignment

Extended UML Meta-Class

- Direct mapping without a stereotype to UML::Slot

Generalization

None.

Properties

- TDL::MemberAssignment.memberSpec := UML::Slot.value
- TDL::MemberAssignment.member := UML::Slot.definingFeature

Constraints

None.

C.2.1.13 Parameter

Extended UML Meta-Class

- Direct mapping without a stereotype to ConnectableElement

Generalization

None.

Properties

- TDL::Parameter.dataType:= UML::ConnectableElement.type

Constraints

None.

C.2.1.14 FormalParameter

Extended UML Meta-Class

- Direct mapping without a stereotype to UML::Parameter

Generalization

None.

Properties

None.

Constraints

None.

C.2.1.15 Variable

Extended UML Meta-Class

- Direct mapping without a stereotype to UML::Property

Generalization

None.

Properties

- TDL::Variable.dataType := UML.Property.type

Constraints

None.

C.2.1.16 Action

Extended UML Meta-Class

- Direct mapping without a stereotype to UML::OpaqueBehavior

Generalization

None.

Properties

- TDL::Action.body := UML::OpaqueBehavior.body
- TDL::Action.formalParameter := UML::OpaqueBehavior.ownedParameter

Constraints

None.

C.2.1.17 Function

Extended UML Meta-Class

- UML::FunctionBehavior

Generalization

None.

Properties

- TDL::Function.returnType := UP4TDL::Function.returnType (Derived property computed as the type of the out parameter of the underlying behaviour)

Constraints

- There shall be exactly one parameter of a Function that is of kind return.

C.2.2 Data Use

C.2.2.0 Overview

Most of the Data Use concepts require a stereotype. In most of the cases these stereotypes extend the UML::Expression meta-class, except for DataUse, which extends ValueSpecification.

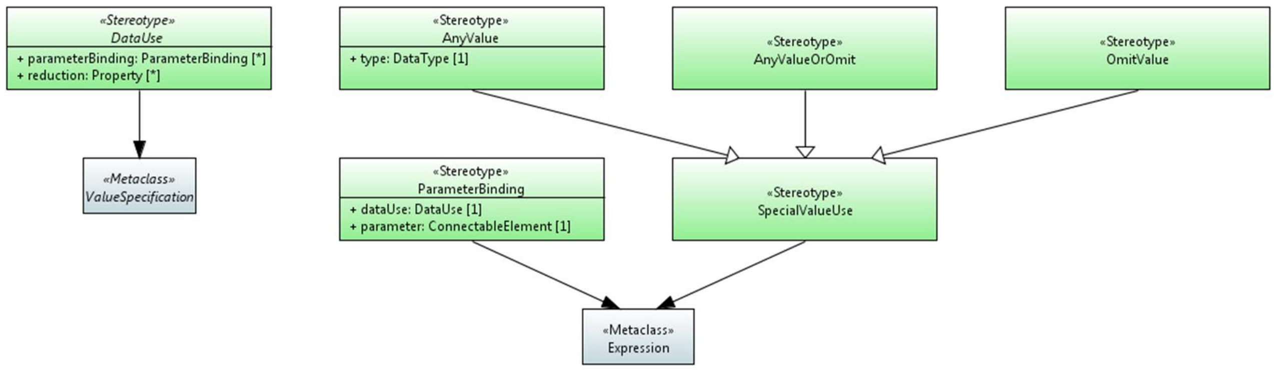

Figure C.2.2: Data use concepts Part 1

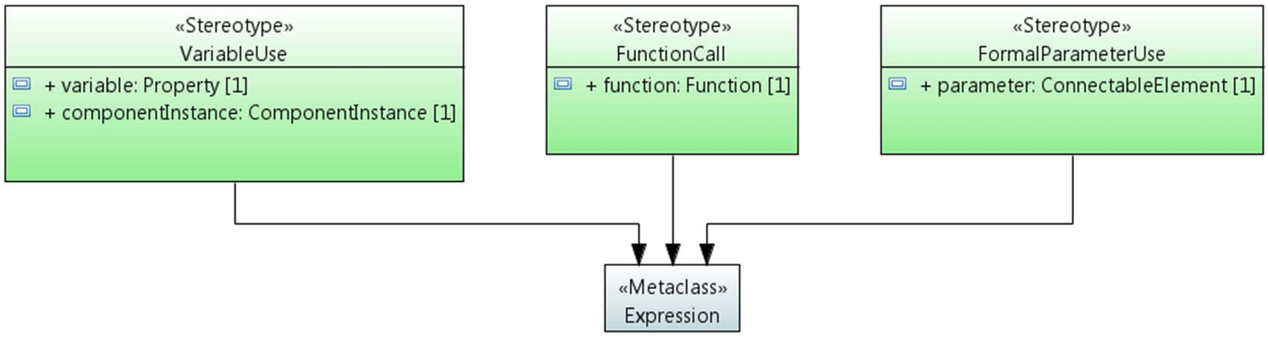

Figure C.2.3: Data use concepts Part 2

C.2.2.1 DataUse

Extended UML Meta-Class

- UML::ValueSpecification. This is a required extension.

Generalization

None.

Properties

- <<UP4TDL::DataUse>>
- TDL::DataUse.argument := UP4TDL::DataUse.argument (new property)
- TDL::DataUse.reduction := UP4TDL::DataUse.reduction (new property)

Constraints

None.

C.2.2.2 ParameterBinding

Extended UML Meta-Class

- UML::Expression

Generalization

None.

Properties

- <<UP4TDL::ParameterBinding>>
- TDL::ParameterBinding.dataUse := UML::Expression.operand
- TDL::ParameterBinding.parameter := UP4TDL::ParameterBinding.parameter (new property)

Constraints

None.

C.2.2.3 StaticDataUse

Extended UML Meta-Class

N/A.

Generalization

N/A.

Properties

N/A.

Constraints

None.

C.2.2.4 DataInstanceUse

Extended UML Meta-Class

- Direct mapping without a stereotype to UML::InstanceValue

Generalization

None.

Properties

- TDL::DataInstanceUse.dataInstance := UML::InstanceValue.instance

Constraints

None.

C.2.2.5 SpecialValueUse

Extended UML Meta-Class

- UML::Expression

Generalization

None.

Properties

- <<UP4TDL::SpecialValueUse>>

Constraints

None.

C.2.2.6 AnyValue

Extended UML Meta-Class

- UML::Expression

Generalization

- UP4TDL::SpecialValueUse

Properties

- <<UP4TDL::AnyValue>>
- TDL:: AnyValue.type := UP4TDL:: AnyValue.type (new property)

Constraints

None.

C.2.2.7 AnyValueOrOmit

Extended UML Meta-Class

- UML::Expression

Generalization

- UP4TDL::SpecialValueUse

Properties

- <<UP4TDL::AnyValueOrOmit>>

Constraints

None.

C.2.2.8 OmitValue

Extended UML Meta-Class

- UML::Expression

Generalization

- UP4TDL::SpecialValueUse

Properties

- <<UP4TDL::OmitValue>>

Constraints

None.

C.2.2.9 DynamicDataUse

Extended UML Meta-Class

N/A.

Generalization

N/A.

Properties

N/A.

Constraints

None.

C.2.2.10 FunctionCall

Extended UML Meta-Class

- UML::Expression

Generalization

None.

Properties

- <<UP4TDL::FunctionCall>>
- TDL::FunctionCall.function := UP4TDL::FunctionCall.function (new property)

Constraints

None.

C.2.2.11 FormalParameterUse

Extended UML Meta-Class

- UML::Expression

Generalization

None.

Properties

- <<UP4TDL::FormalParameterUse>>
- TDL::FormalParameterUse.parameter = UP4TDL::FormalParameterUse.parameter (new property)

Constraints

None.

C.2.2.12 VariableUse

Extended UML Meta-Class

- UML::Expression

Generalization

None.

Properties

- <<UP4TDL::VariableUse>>
- TDL::VariableUse.variable := UP4TDL::VariableUse.variable (new property)
- TDL::VariableUse.componentInstance := UP4TDL::VariableUse.componentInstance (new property)

Constraints

None.

C.3 Time

C.3.0 Overview

All Time-related TDL concepts require stereotypes as shown in figures C.3.1 to C.3.3.

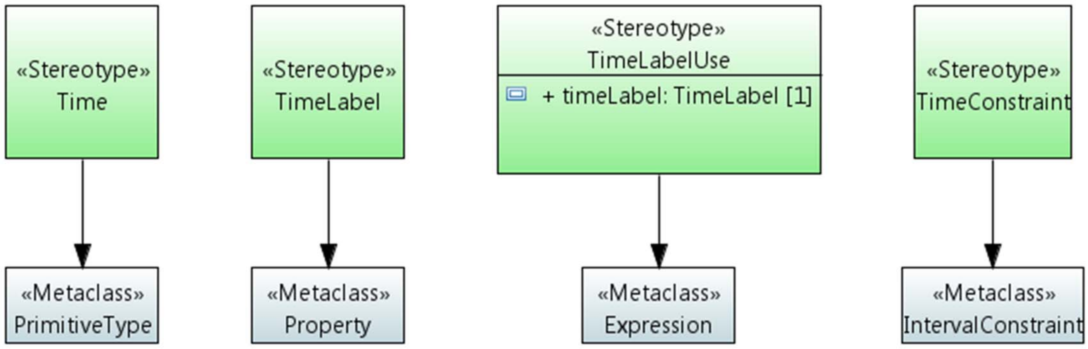

Figure C.3.1: General time-related concepts

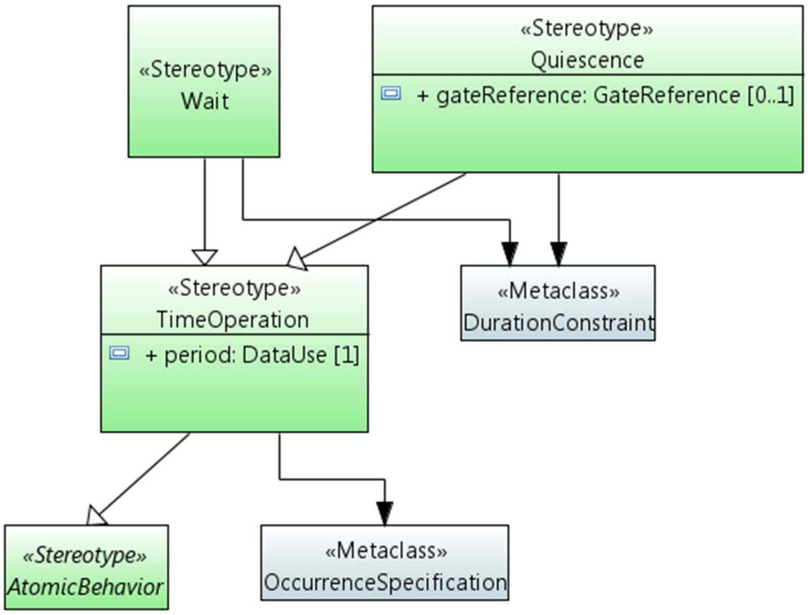

Figure C.3.2: Time operations

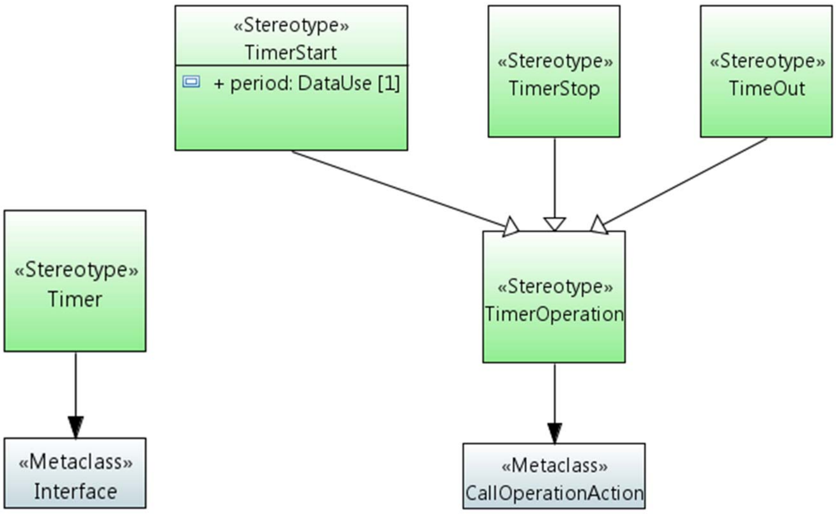

Figure C.3.3: Timer and Timer Operations

C.3.1 Time

Extended UML Meta-Class

- UML::PrimitiveType (by generalization)

Generalization

None.

Properties

- <<UP4TDL::Time>>

Constraints

None.

C.3.2 TimeLabel

Extended UML Meta-Class

- UML::Property

Generalization

None.

Properties

- <<UP4TDL::TimeLabel>>

Constraints

None.

C.3.3 TimeLabelUse

Extended UML Meta-Class

- UML::Expression (by generalization)

Generalization

None.

Properties

- <<UP4TDL::TimeLabelUse>>
- TDL::TimeLabelUse.timeLabel := UP4TDL::TimeLabelUse.timeLabel (new property)

Constraints

None.

C.3.4 TimeConstraint

Extended UML Meta-Class

- UML::IntervalConstraint

Generalization

None.

Properties

- <<UP4TDL::TimeConstraint>>
- TDL::TimeConstraint. timeConstraintExpression := UP4TDL::TimeConstraint. timeConstraintExpression (new property)

Constraints

None.

C.3.5 TimeOperation

Extended UML Meta-Class

- UML::OccuranceSpecification

Generalization

- UP4TDL::AtomicBehavior

Properties

- <<UP4TDL::TimeOperation>>
- TDL::TimeOperation.period := UP4TDL::TimeOperation .period (new property)

- TDL::TimeOperation.ComponentInstance:= UP4TDL::ComponentInstance.base_OccurenceSpecification.covered.represents

Constraints

None.

C.3.6 Wait

Extended UML Meta-Class

- UML::DurationConstraint

Generalization

- UP4TDL::TimeOperation

Properties

- << UP4TDL::Wait>>

Constraints

None.

C.3.7 Quiescence

Extended UML Meta-Class

- UML::DurationConstraint

Generalization

- UP4TDL::TimeOperation

Properties

- << UP4TDL::Quiescence>>
- TDL::Quiescence.gateReference := UP4TDL::Quiescence. gateReference (new property)

Constraints

None.

C.3.8 Timer

Extended UML Meta-Class

- UML::Interface

Generalization

None.

Properties

None.

Constraints

None.

C.3.9 TimerOperation

Extended UML Meta-Class

- UML::CallOperationAction

Generalization

None.

Properties

- <<UP4TDL::TimerOperation>>
- TDL::TimerOperation.timer := UP4TDL::TimerOperation.timer (new property)
- TDL::TimerOperation.componentInstance := UP4TDL::TimerOperation..componentInstance (new property)

Constraints

None.

C.3.10 TimerStart

Extended UML Meta-Class

- UML::CallOperationAction

Generalization

- UP4TDL::TimerOperation

Properties

- <<UP4TDL::TimerStart>>
- TDL::TimerStart.period := UP4TDL::TimerStart.period (new property)

Constraints

None.

C.3.11 TimerStop

Extended UML Meta-Class

- UML::CallOperationAction

Generalization

- UP4TDL::TimerOperation

Properties

- <<UP4TDL::TimerStop>>

Constraints

None.

C.3.12 TimeOut

Extended UML Meta-Class

- UML::CallOperationAction

Generalization

- UP4TDL::TimerOperation

Properties

- <<UP4TDL::TimeOut>>

Constraints

None.

C.4 Test Configuration

C.4.0 Overview

The TDL concepts shown on figures C.4.1 and C.4.2 require stereotypes in UP4TDL. The other Test Configuration-related concepts are directly mapped.

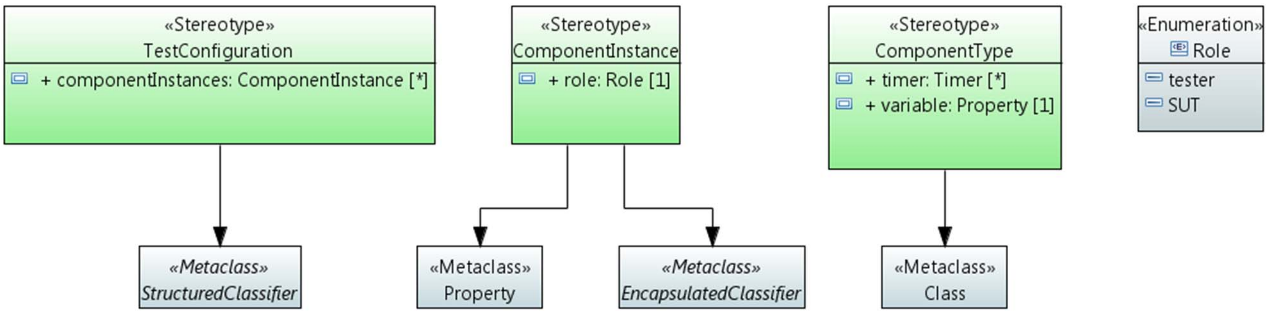

Figure C.4.1: Test Configuration concepts Part 1

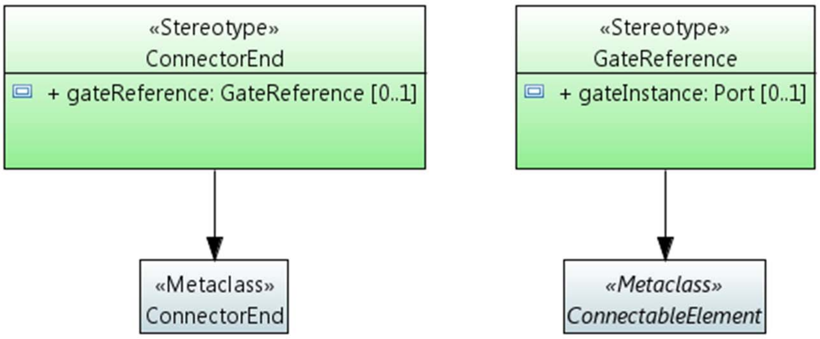

Figure C.4.2: Test Configuration concepts Part 2

C.4.1 GateType

Extended UML Meta-Class

- Direct mapping without a stereotype to UML::DataType or UML::Interface containing several datatype (for the case of multipleDataType handled by the GateInstance)

Generalization

None.

Properties

- TDL::GateType.dataType := UML::Interface.ownedAttribute.type[1..*] {unique}

Constraints

None.

C.4.2 GateInstance

Extended UML Meta-Class

- Direct mapping without a stereotype to UML::Port

Generalization

None.

Properties

- TDL::GateInstance.gateType := UML::Port.type

Constraints

None.

C.4.3 ComponentType

Extended UML Meta-Class

- UML::Class

Generalization

None.

Properties

- <UP4TDL::ComponentType>>
- TDL::ComponentType.timer:=UP4TDL::ComponentType.timer (new property)
- TDL::ComponentType.variable := UP4TDL::ComponentType.base_Class.ownedProperty
- TDL::ComponentType.gateInstance := UP4TDL::ComponentType.base_Class.ownedPorts

Constraints

None.

C.4.4 ComponentInstance

Extended UML Meta-Class

- UML::Property
- UML::Component

Generalization

None.

Properties

- <<UP4TDL::ComponentInstance>>
- TDL::ComponentInstance.type: ComponentType:= UP4TDL::ComponentInstance.base_Property.type
- TDL::ComponentInstance.role := UP4TDL::ComponentInstance.role (.new property)

Constraints

- Type of ComponentInstance shall be ComponentType.

C.4.5 ComponentInstanceRole

Extended UML Meta-Class

- ComponentInstanceRole is modelled as not a meta-class, but as an instance of a UML::Enumeration

Generalization

None.

Properties

None.

Literals

- SUT
  UML::EnumerationLiteral <<SUT>>

- Tester
  UML::EnumerationLiteral <<Tester>>

Constraints

None.

C.4.6 GateReference

The TDL::GateReference concept directly mapped to the UML::ConnectorEnd concept.

Extended UML Meta-Class

- Direct mapping to UML::ConnectorEnd

Generalization

None.

Properties

- TDL::GateReference.component := UMLConnectorEnd.partWithPort (new property)

- TDL::GateReference.gate := UML::ConnectorEnd.role

Constraints

None.

C.4.7 Connection

Extended UML Meta-Class

- Direct mapping without a stereotype to UML::Connector

Generalization

None.

Properties

- TDL::Connection.endPoint := UML::Connector.end

Constraints

None.

C.4.8 TestConfiguration

Extended UML Meta-Class

- UML::StructuredClassifier

Generalization

None.

Properties

- <<UP4TDL::TestConfiguration>>
- TDL::TestConfiguration.component := UP4TDL::TestConfiguration.componentInstance (new derived property computed as the subset of ownedAttribute that have stereotype componentInstance applied on)
- TDL::TestConfiguration.connector := UP4TDL::TestConfiguration.base_StructuredClassifier.ownedConnector

Constraints

None.

C.5 Test Behaviour

C.5.1 Test Description

C.5.1.0 Overview

TDL::TestDescription is mapped to UML::BehavioredClassifier, while there is no need to map TDL::BehaviourDescription.

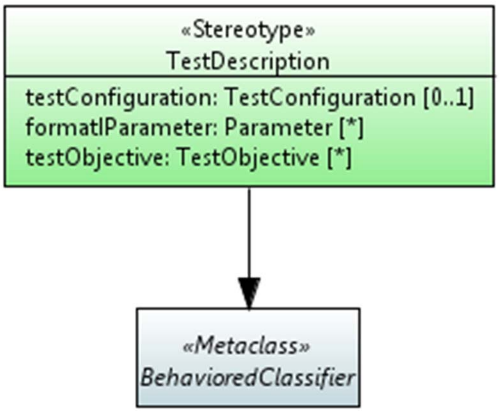

Figure C.5.1: Test Description

C.5.1.1 TestDescription

Extended UML Meta-Class

- UML::BehavioredClassifier

Generalization

None.

Properties

- << UP4TDL::TestDescription>>

- TDL::TestDescription.testConfiguration := UP4TDL::TestDescription.testConfiguration (new property)
- TDL::TestDescription.formalParameter := UP4TDL::TestDescription.parameter (new property)
- TDL::TestDescription.behaviourDescription := UP4TDL::TestDescription.base_BehavioredClassifier.classifierBehavior
- TDL::TestDescription.testObjective := UP4TDL::TestDescription.testObjective (new property)

Constraints

None.

C.5.1.2 BehaviourDescription

Extended UML Meta-Class

- Direct mapping without a stereotype to UML::InteractionFragment.

Generalization

None.

Properties

None.

Constraints

None.

Model to Model transformation advice

- The BehaviourDescription is the first retrieved as the classifierBehavior of the TestDescription

C.5.2 Combined Behaviour

C.5.2.0 Overview

Behaviour is mapped to UML::InteractionFragment, CombinedBehaviours are mapped to CombinedFragment, Blocks are mapped to UML::InteractionOperand. They are required extensions. SingleCombinedBehaviour and MultipleCombinedBehaviour are not to be mapped, but all of their sub-classes are mapped to UML::CombinedFragment.

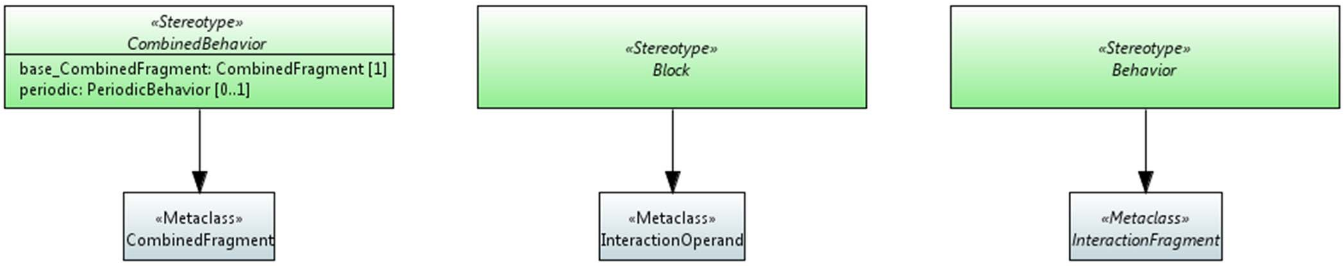

Figure C.5.2: Behaviour, CombinedBehaviour and Block Concepts

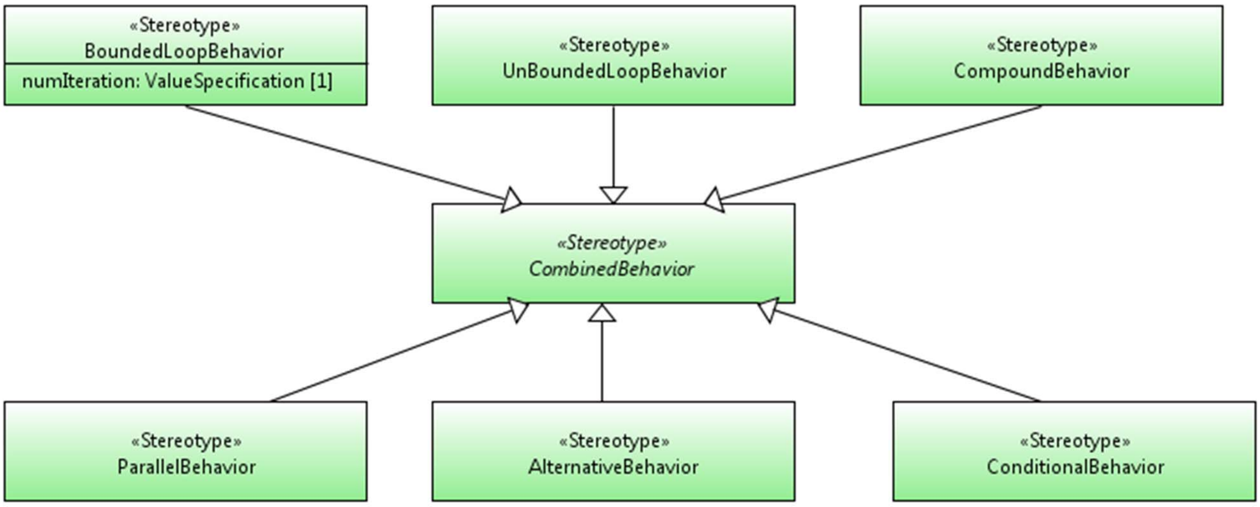

Figure C.5.3: Single and Multiple Combined Behaviour Concepts

C.5.2.1 Behaviour

Extended UML Meta-Class

- UML::InteractionFragment. This is a required extension.

Generalization

None.

Properties

- <<UP4TDL::Behavior>>
- TDL::Behaviour.testObjective := UP4TDL::Behavior.testObjective

Constraints

None.

C.5.2.2 Block

Extended UML Meta-Class

- UML::InteractionOperand. This is a required extension

Generalization

None.

Properties

- <<UP4TDL::Block>>
- TDL::Block.behaviour := UP4TDL::Block.base_InteractionOperand.fragment
- TDL::Block.guard := UP4TDL::Block.base_InteractionOperand.guard.specification

Constraints

None.

C.5.2.3 CombinedBehaviour

Extended UML Meta-Class

- UML::CombinedFragment

Generalization

- UP4TDL::Behavior

Properties

- <<UP4TDL::CombinedBehavior>>
- TDL::CombinedBehaviour.periodic := UP4TDL::CombinedBehavior.periodic (new property)
- TDL::CombinedBehaviour.exceptional := UP4TDL::CombinedBehavior. exceptional (new property)

Constraints

None.

C.5.2.4 SingleCombinedBehaviour

This MetaClass shall not be mapped. The constraints to on the number of owned blocks is transferred to the actual Combined Behaviours (i.e. CompoundBehaviour, BoundedLoopBehaviour and UnboundedLoopBehaviour). This is done in order to limitate the number of stereotypes and the stereotype hierarchy lenght.

C.5.2.5 CompoundBehaviour

Extended UML Meta-Class

- UML::CombinedFragment (by generalization) with one InteractionOperand (block)

Generalization

- UP4TDL::CombinedBehavior

Properties

- << UP4TDL::CombinedBehavior>>

Constraints

None.

C.5.2.6 BoundedLoopBehaviour

Extended UML Meta-Class

- UML::CombinedFragment (by generalization) with one InteractionOperand (block)

Generalization

- UP4TDL::CombinedBehavior

Properties

- << UP4TDL::BoundedLoopBehavior>>
- TDL::BoundedLoopBehaviour.numIteration := UP4TDL::BoundedLoopBehavior.numIteration (new property)

Constraints

None.

C.5.2.7 UnboundedLoopBehaviour

Extended UML Meta-Class

- UML::CombinedFragment (by generalization) with one InteractionOperand (block)

Generalization

- UP4TDL::CombinedBehavior

Properties

- <<UP4TDL::UnboundedLoopBehavior>>

Constraints

None.

C.5.2.8 MultipleCombinedBehaviour

This MetaClass shall not be mapped. The constraints to on the number of owned blocks is transferred to the actual Combined Behavious. This is done in order to limitate the number of stereotypes and the stereotype hierarchy lenght.

C.5.2.9 AlternativeBehaviour

Extended UML Meta-Class

- UML::CombinedFragment (by generalization) with at least 2 InteractionOperand (block)

Generalization

- UP4TDL::CombinedBehavior

Properties

- <<UP4TDL::AlternativeBehavior >>

Constraints

None.

C.5.2.10 ConditionalBehaviour

Extended UML Meta-Class

- UML::CombinedFragment (by generalization) with one or moreInteractionOperand (block)

Generalization

- UP4TDL::CombinedBehavior

Properties

- <<UP4TDL::ConditionalBehavior >>

Constraints

None.

C.5.2.11 ParallelBehaviour

Extended UML Meta-Class

- UML::CombinedFragment (by generalization) with one or moreInteractionOperand (block)

Generalization

- UP4TDL::CombinedBehavior

Properties

- <<UP4TDL::ParallelBehavior >>

Constraints

None.

C.5.2.12 ExceptionalBehaviour

Extended UML Meta-Class

- UML::CombinedFragment (by generalization) with one InteractionOperand (block)

Generalization

- UP4TDL::CombinedBehavior

Properties

- <<UP4TDL::ExceptionalBehavior >>
- TDL:: ExceptionalBehaviour.block := UP4TDL:: ExceptionalBehavior.base_CombinedFragment.operand
- TDL:: ExceptionalBehaviour.guardedComponent := UP4TDL::ExceptionalBehavior. guardedComponent (new property)

Constraints

None.

C.5.2.13 DefaultBehaviour

Extended UML Meta-Class

- UML::CombinedFragment (by generalization)

Generalization

- UP4TDL::ExceptionalBehavior

Properties

- << UP4TDL::DefaultBehavior>>

Constraints

None.

C.5.2.14 InterruptBehaviour

Extended UML Meta-Class

- UML::CombinedFragment (by generalization)

Generalization

- UP4TDL::ExceptionalBehavior

Properties

- << UP4TDL::InterruptBehavior>>

Constraints

None.

C.5.2.15 PeriodicBehaviour

Extended UML Meta-Class

- UML::CombinedFragment (by generalization)

Generalization

- UP4TDL::CombinedBehavior

Properties

- << UP4TDL::PeriodicBehavior>>
- TDL:: PeriodicBehaviour.block := UP4TDL::PeriodicBehavior.base_CombinedFragment.operand
- TDL:: PeriodicBehaviour.period := UP4TDL:: PeriodicBehaviour (new property)

Constraints

None.

C.5.3 Atomic Behaviour

C.5.3.0 Overview

All TDL::AtomicBehaviour-related concepts require stereotypes, as shown in figures C.5.3 to C.5.6.

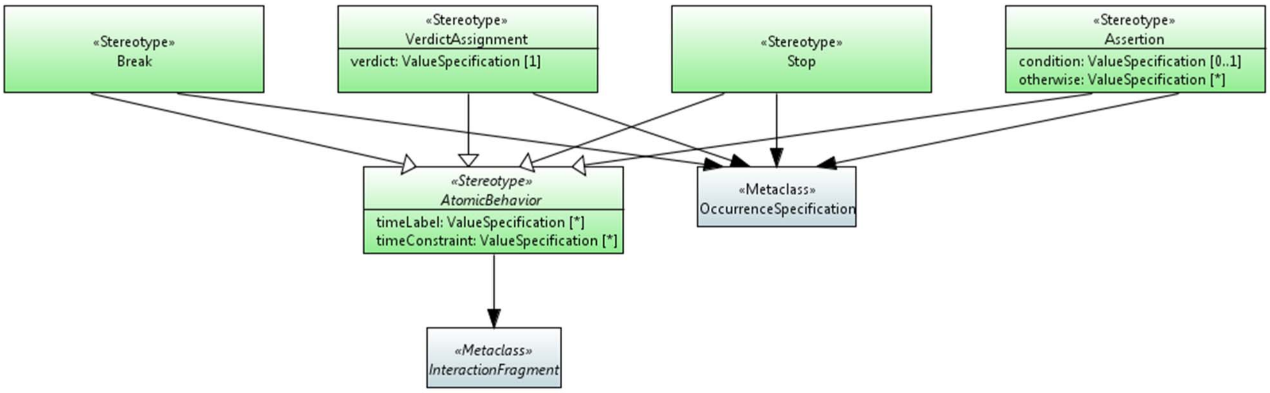

Figure C.5.4: Global Atomic Behaviour concepts

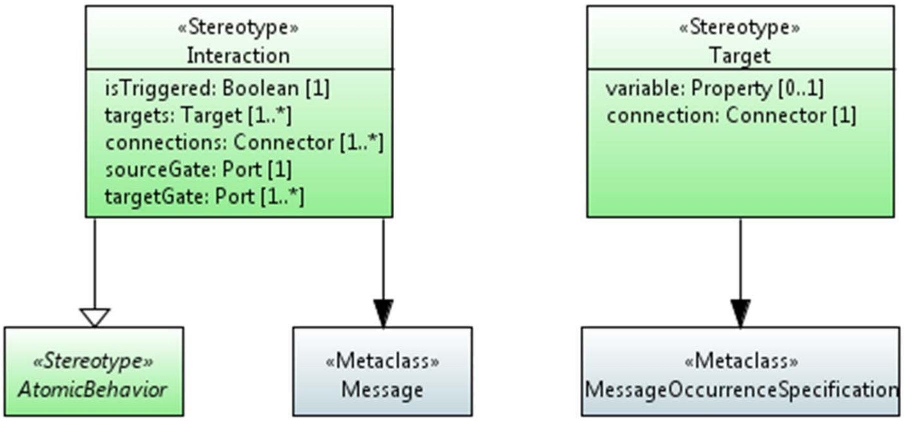

Figure C.5.5: Interaction behaviour

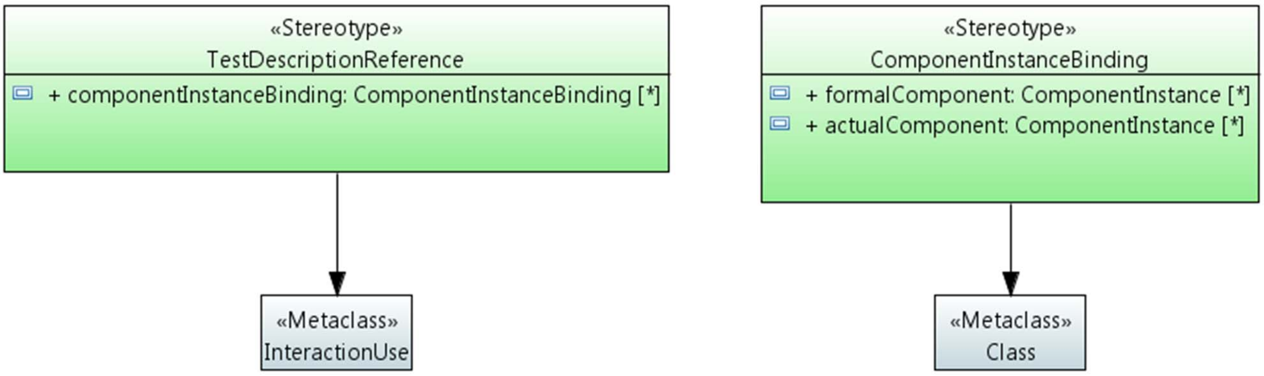

Figure C.5.6: Test Description Reference

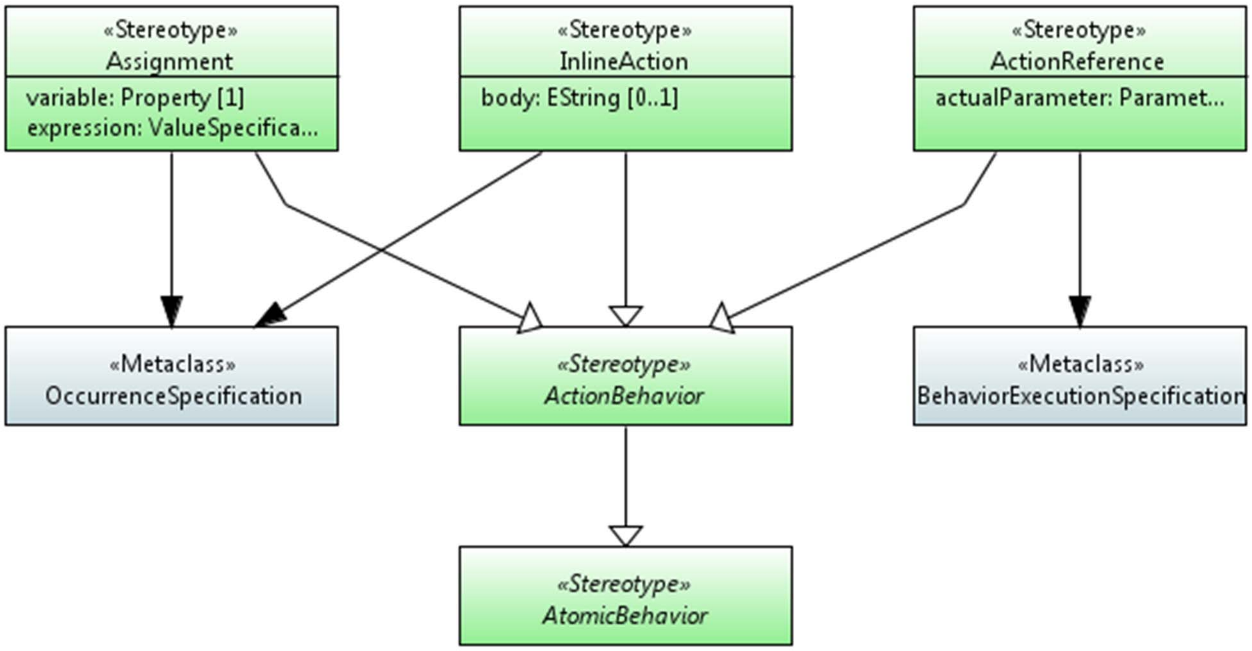

Figure C.5.7: Action behaviour concepts

C.5.3.1 AtomicBehaviour

Extended UML Meta-Class

- UML::InteractionFragment

Generalization

- None

Properties

- <<UP4TDL::AtomicBehavior>>
- TDL::AtomicBehaviour.timeLabel :=- UP4TDL::AtomicBehavior.timeLabel (new property)
- TDL::AtomicBehaviour.timeConstraint := UP4TDL::AtomicBehavior.timeConstraint (new property)

Constraints

None.

C.5.3.2 Break

Extended UML Meta-Class

- UML::OccurrenceSpecification

Generalization

- UP4TDL::AtomicBehavior

Properties

- << UP4TDL::Break>>

Constraints

None.

C.5.3.3 Stop

Extended UML Meta-Class

- UML::OccurrenceSpecification

Generalization

- UP4TDL::AtomicBehavior

Properties

- << UP4TDL::Stop>>

Constraints

None.

C.5.3.4 VerdictAssignment

Extended UML Meta-Class

- UML::OccurrenceSpecification
- UML::StateInvariant

Generalization

- UP4TDL::AtomicBehavior

Properties

- << UP4TDL::VerdictAssignment>>
- TDL::VerdictAssignment.verdict := UP4TDL::VerdictAssignment.verdict

Constraints

None.

C.5.3.5 Assertion

Extended UML Meta-Class

- UML::OccurrenceSpecification
- UML::StateInvariant

Generalization

- UP4TDL::AtomicBehavior

Properties

- << UP4TDL::Assertion>>
- TDL::Assertion.condition := UP4TDL::Assertion.condition (new property)
- TDL::Assertion.otherwise := UP4TDL::Assertion.otherwise (new property)

Constraints

None.

C.5.3.6 Interaction

Extended UML Meta-Class

- UML::Message

Generalization

- UP4TDL::AtomicBehaviour

Properties

- <<UP4TDL::Interaction>>
- TDL::Interaction.isTrigger := UP4TDL::Interaction.isTrigger (new property)
- TDL::Interaction.argument := UP4TDL::Interaction.base_Message.argument
- TDL::Interaction.target := UP4TDL::Interaction.targets ; (the first is UP4TDL::Interaction.base_Message.receiveEvent)
- UP4TDL::Interaction contains a set of connector UP4TDL::Interaction.connection (new derived property computed as the union of all connectors of targets of the interactions) that allows the derivation of the source gate and the target gates
- TDL::Interaction.sourceGate : UP4TDL::Interaction.sourceGate (new derived property computed from as the single sending Port)
- TDL::Interaction.targetGate:= UP4TDL::Interaction.targetGate (new derived property computed as the union of all receiving Ports)

Constraints

None.

C.5.3.7 Target

Extended UML Meta-Class

- UML::(Message)OccurenceSpecification

Generalization

None.

Properties

- <<UP4TDL::Target>>

- TDL::Target.variable := UP4TDL::Target.variable
- TDL::Target.targetGate := UP4TDL::Target retrieved from UP4TDL::Target.connection (see Model to model Transformation advice)

Constraints

None.

Model to model Transformation Advice

- UP4TDL::Target contains a property connection of type UML::Connector. TDL::Target.targetGate *t* is the Retrieved from a UP4TDL::Target as follow:
  - It is the end of UP4TDL::Target.end *ce* such that : *ce.partWithPort* is the same as *t.covered.represents.*

C.5.3.8 TestDescriptionReference

Extended UML Meta-Class

- UML::InteractionUse

Generalization

None.

Properties

- <<UP4TDL::TestDescriptionReference>>
- TDL::TestDescriptionReference.testDescription := UP4TDL::TestDescriptionReference.base_InteractionUse.referTo
- TDL::TestDescriptionReference.actualParameter := UP4TDL::TestDescriptionReference.base_InteractionUse.argument
- TDL::TestDescriptionReference.componentInstanceBinding := UP4TDL::TestDescriptionReference.componentInstanceBinding (new property)

Constraints

None.

C.5.3.9 ComponentInstanceBinding

Extended UML Meta-Class

- UML::Class

Generalization

None.

Properties

- <<UP4TDL::ComponentInstanceBinding>>
- TDL::ComponentInstanceBinding.formalComponent := UP4TDL::ComponentInstanceBinding.formalComponent (new property)

- TDL::ComponentInstanceBinding.actualComponent := UP4TDL::ComponentInstanceBinding.actualComponent (new property)

Constraints

None.

C.5.3.10 ActionBehaviour

Extended UML Meta-Class

- UML::InstanceFragment (by generalization)

Generalization

- UP4TDL::AtomicBehavior

Properties

- <<UP4TDL::ActionBehavior>>
- TDL::ActionBehaviour.componentInstance := retrieved from the lifeline that this UP4TDL::ActionBehavior is covers. (see Model to model Transformation advice)

Constraints

None.

Model to model transformation advice

- The componentInstance is retrieved on the child concepts
- For ActionReference:
  - TDL::ActionBehaviour.componentInstance = UP4TDL::AcionReference.base_BehaviorExecutionSpecification.start.covered.represents
- For InlineAction:
  - TDL::ActionBehaviour.componentInstance = InlineAction.base_OccurenceSpecification.covered.represents.
- For Assignment:
  - TDL::ActionBehaviour.componentInstance = Assignment.base_OccurenceSpecification.covered.represents.

C.5.3.11 ActionReference

Extended UML Meta-Class

- UML::BehaviorExecutionSpecification
- UML::ActionExecutionSpecification

Generalization

- UP4TDL::ActionBehavior

Properties

- << UP4TDL::ActionReference >>
- TDL::ActionReference.action := UP4TDL::ActionReference.base_BehaviorExecutionSpecification.behavior
- TDL::ActionReference.actualParameter := UP4TDL::ActionReference.actualParameter (new property)

Constraints

None.

C.5.3.12 InlineAction

Extended UML Meta-Class

- UML::OccurrenceSpecification

Generalization

- UP4TDL::ActionBehaviour

Properties

- <<UP4TDL::InlineAction>>
- TDL::InlineAction.body := UP4TDL::InlineAction.body (new property)

Constraints

None.

C.5.3.13 Assignment

Extended UML Meta-Class

- UML::OccurrenceSpecification

Generalization

- UP4TDL::ActionBehaviour

Properties

- <<UP4TDL::Assignment>>
- TDL::Assignment.variable := UP4TDL::Assignment.variable
- TDL::Assignment.expression := UP4TDL::Assignment.expression

Constraints

None.

Annex D (informative): Bibliography

ETSI ES 202 553 (V1.2.1): "Methods for Testing and Specification (MTS); TPLan: A notation for expressing Test Purposes".

ISO/IEC/IEEE 29119-3:2013: "Software and Systems Engineering - Software Testing; Part 3: Test Documentation".

OMG®: "UML Testing Profile (UTP) V1.2", formal/2013-04-03.

ETSI ES 203 119-2 (V1.2.1): "Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 2: Graphical Syntax".

History

Document history		
V1.1.1	April 2014	Publication as ETSI ES 203 119
V1.2.1	June 2015	Publication
V1.3.1	July 2016	Membership Approval Procedure MV 20160911: 2016-07-13 to 2016-09-12
V1.3.1	September 2016	Publication