ETSI ES 203 119-1 V1.8.1 (2025-07)

ETSI STANDARD

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

Reference
RES/MTS-203119-1v1.8.1

Keywords
language, MBT, methodology, model, 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 - APE 7112B
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° w061004871

Contents

Intellectual Property Rights

Essential patents

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

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

Trademarks

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

DECT™, **PLUGTESTS™**, **UMTS™** and the ETSI logo are trademarks of ETSI registered for the benefit of its Members. **3GPP™**, **LTE™** and **5G™** logo are trademarks of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners. **oneM2M™** logo is a trademark of ETSI registered for the benefit of its Members and of the oneM2M Partners. **GSM**® and the GSM logo are trademarks registered and owned by the GSM Association.

Foreword

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

The present document is part 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)";

Part 5: "UML profile for TDL";

Part 6: "Mapping to TTCN-3";

Part 7: "Extended Test Configurations";

Part 8: "Textual Syntax";

Part 9: "Test Runtime Interfaces".

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 analysers 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 in the ETSI docbox.

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® formal/2013-06-01: "Documents Associated With Meta Object Facility™ (MOF™) Version 2.4.1".

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

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

[4] Void.

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

[6] ETSI ES 203 119-4: "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".

[8] ETSI ES 203 119-8: "Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 8: Textual Syntax".

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 may be useful in implementing an ETSI deliverable or add to the reader's understanding, but are not required for conformance to the present document.

[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] Void.

[i.3] Void.

[i.4] ETSI: "The TDL Open Source Project Website", last visited 20.12.2021.

3 Definition of terms, symbols and abbreviations

3.1 Terms

For the purposes of the present document, the following terms 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

effectively static: any data specification that is statically determined (also recursively for all parameter bindings) and can be considered constant during the execution

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

resolved data type: data type of a data use, determined by a referenced data instance, data type, parameter, return type of a function, or the context where the data use is defined

scope of behaviour: components participating in a behaviour

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

NOTE: Tester-input events in the present document are the following:

- Quiescence.
- TimeOut.
- An 'Interaction' with a 'Target' that in turn-via its 'GateReference'-refers to a 'ComponentInstance' in the role 'Tester'. If the source of an 'Interaction' is also a tester then it is not a tester-input event.

<undefined>: semantical concept denoting an undefined data value

3.2 Symbols

Void.

3.3 Abbreviations

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

ADT	Abstract Data Type
IEC	International Electrotechnical Commission
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
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 and only one atomic behaviour is executing at any time.
- 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 allows 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, as well as extensions and mappings to other languages are represented in 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.
- TDL-UP4TDL, part 5: Covers the standardized mapping of TDL to UML with the UML Profile for TDL.
- TDL-T3, part 6: Covers the standardized mapping all TDL to TTCN-3.
- TDL-TC, part 7: Covers all parts of concrete/abstract syntax and static/dynamic semantics of the TDL Extended Test Configurations extension.
- TDL-TX, part 8: Covers the normative concrete syntax of textual TDL.
- TDL-TRI, part 9: Covers the test runtime interfaces for TDL.

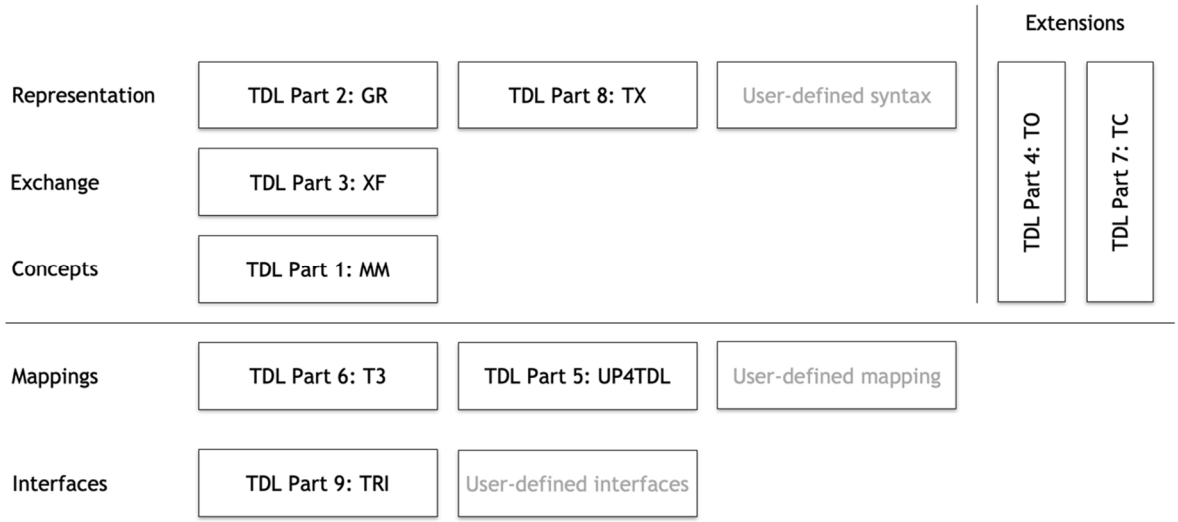

Figure 4.1: The TDL standards and their positioning

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, analysers 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).

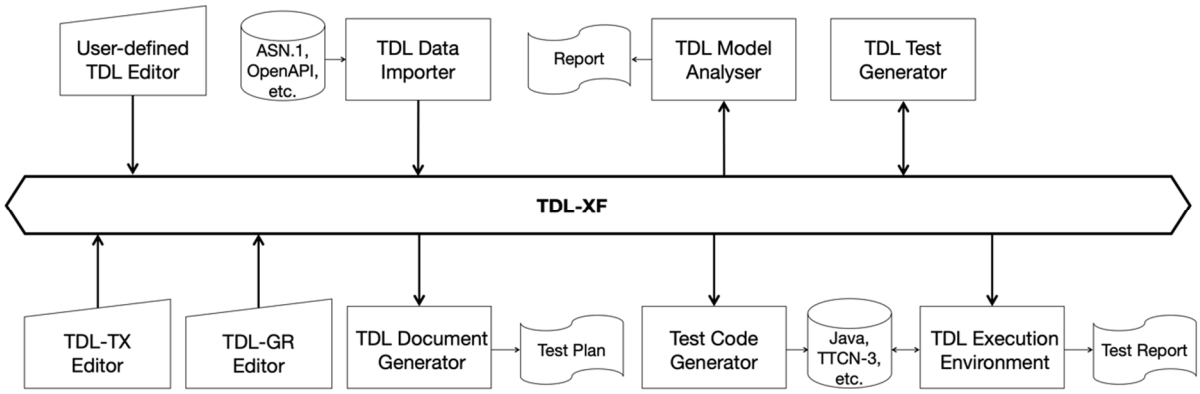

Figure 4.2: A scalable TDL tool architecture

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 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. For enumeration types, a paragraph "Literals" is used instead of the "Properties" paragraph to describe the enumeration literals and their meaning.

- Paragraph "Constraints" lists rules describing the static semantics of the concept, both in terms of informal descriptions and formally as OCL [3] 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 constraint 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 Element Operations

The following operations shall be provided in an implementation of the TDL meta-model in order to ensure the semantic integrity of TDL models. The operations are also used as reusable shortcuts for the specification of the formalized constraints and are required for their interpretation, in addition to the operations provided by the standard library of OCL [3]:

- Element **container**(): Element - applicable on any TDL 'Element', returns the 'Element' that contains the construct directly, except for the top level 'Element' which is not contained in any other 'Element'.

- Element **getParentTestDescription**(): TestDescription - applicable on any TDL 'Element', returns the 'TestDescription' that contains the construct directly or indirectly.

- DataUse **resolveDataType** (): DataType - applicable on any TDL 'DataUse', returns the 'DataType' resolved from the 'DataUse' after resolving all 'MemberReference's specified in 'reduction', or undefined if the data type cannot be resolved, which implies an invalid model.

- DataUse **resolveBaseDataType** (): DataType - applicable on any TDL 'DataUse', returns the 'DataType' resolved from the 'DataUse' without applying the 'MemberReference's specified in 'reduction', or undefined if the data type cannot be resolved, which implies an invalid model.

- DataUse **isEffectivelyStatic** (): Boolean - applicable on any TDL 'DataUse', returns the 'Boolean' 'true' if the 'DataUse' is a 'StaticDataUse' or a 'DataElementUse' that refers to a 'DataType', a 'DataInstance', or does not specify a 'dataElement', and all 'ParameterBinding's fulfil the same condition recursively.

- ParameterBinding **resolveParameterType** (): DataType - applicable on any TDL 'ParameterBinding', returns the 'DataType' resolved from the 'parameter' and the 'reduction' of the 'ParameterBinding', or undefined if the data type cannot be resolved, which implies an invalid model.

- Behaviour **isTesterInputEvent** (): Boolean - applicable on any TDL 'Behaviour', returns the 'Boolean' 'true' if the 'Behaviour' is a tester-input event as defined in the present document, and 'false' otherwise.

- Block **getParticipatingComponents** (): Set<ComponentInstance> - applicable on any TDL 'Block', returns all 'ComponentInstance's that participate in the 'Block' (as specified in clause 9.3.2).

- Behaviour **getParticipatingComponents** (): Set<ComponentInstance> - applicable on any TDL 'Behaviour', returns all 'ComponentInstance's that participate in the 'Behaviour'.

- Behaviour **getPermittedComponents** (): Set<ComponentInstance> - applicable on any TDL 'AlternativeBehaviour', 'OptionalBehaviour', or 'ExceptionalBehaviour', returns all 'ComponentInstance's that are permitted if the starting behaviour is specified. In case the starting behaviour is not specified all 'ComponentInstance's are returned.

- DataType **allConstraints** (): Set<Constraint> - applicable on any TDL 'DataType', returns all 'Constraints's that are associated with the 'DataType', including inherited ones.

- StructuredDataType **allMembers** (): Set<Member> - applicable on any TDL 'StructuredDataType', returns all 'Member's that are associated with the 'StructuredDataType', including inherited ones.

- GateType **allDataTypes** (): Set<DataType> - applicable on any TDL 'GateType', returns all 'DataTypes's that are associated with the 'GateType', including inherited ones.

- ComponentType **allGates** (): Set<GateInstance> - applicable on any TDL 'ComponentType', returns all 'GateInstance's that are associated with the 'ComponentType', including inherited ones.

- ComponentType **allTimers** (): Set<Timer> - applicable on any TDL 'ComponentType', returns all 'Timer's that are associated with the 'ComponentType', including inherited ones.

- ComponentType **allVariables** (): Set<Variable> - applicable on any TDL 'ComponentType', returns all 'Variable's that are associated with the 'ComponentType', including inherited ones.

- TestConfiguration **compatibleWith** (tc: TestConfiguration, cb: Set<ComponentInstanceBinding>): Boolean - applicable on any TDL 'TestConfiguration', returns 'true' if this 'TestConfiguration' is compatible with the provided 'TestConfiguration' 'tc' under the provided 'ComponentInstanceBinding's 'cb' between component instances of this 'TestConfiguration' and the TestConfiguration 'tc' (as specified in clause 9.4.11).

- Extension **isExtending** (e: PackageableElement): Boolean - applicable on any TDL 'Extension', returns 'true' if the value of 'extending' property of this 'Extension' is the same as the provided 'PackageableElement' 'e', or if the value of 'extending' property of this 'Extension' contains an 'Extension' that extends the provided 'PaclageableElement' 'e' (the condition is determined by calling this operation).

- PackageableElement **conformsTo** (e: PackageableElement): Boolean - applicable on any TDL 'PackageableElement', returns 'true' if this 'PackageableElement' is the same as the provided 'PackageableElement' 'e', or if this 'PackageableElement' contains an 'Extension' and that 'Extension' extends the provided 'PackageableElement' 'e' (the condition is determined by calling the isExtending() operation on the 'Extension').
- PackageableElement **conformsTo** (name: String): Boolean - applicable on any TDL 'PackageableElement', returns 'true' if this 'PackageableElement' conforms to a 'PackageableElement' with the provided 'name'.

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

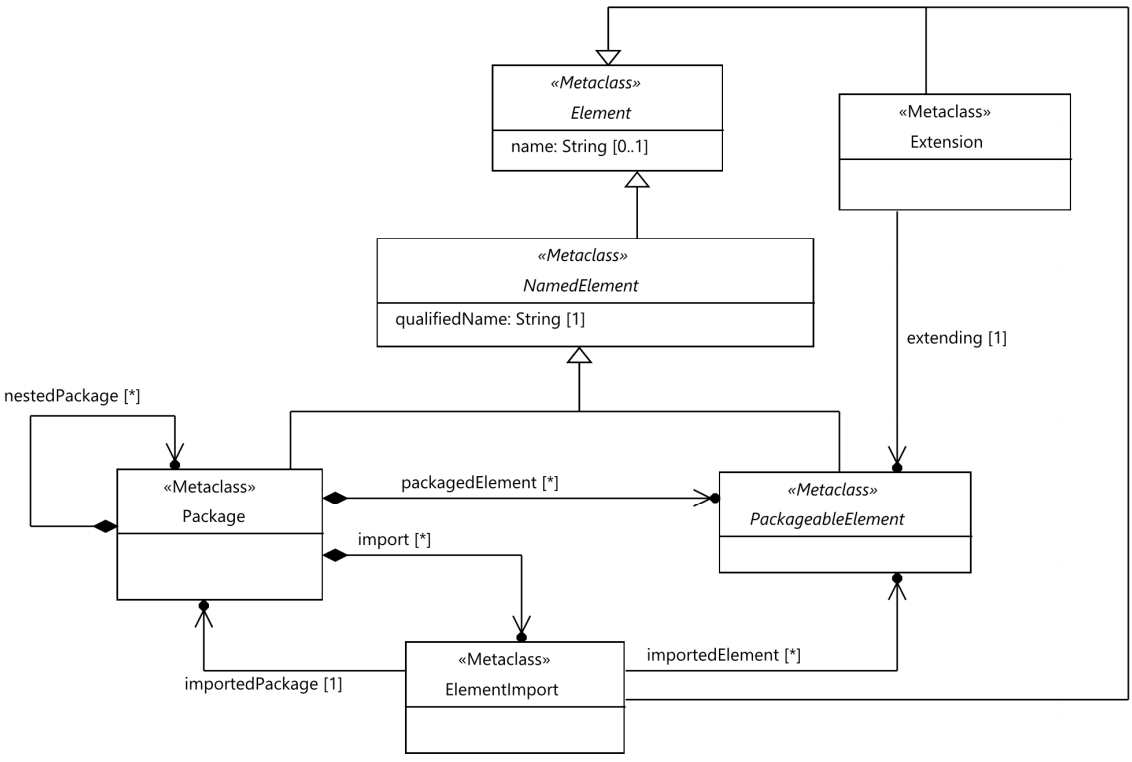

Figure 5.1: Foundational language concepts

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: **DistinguishableName**:
  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'. 'PackageableElement's that are defined within a containing 'Package' are not visible from within 'Package's nested in the 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->includes(e))

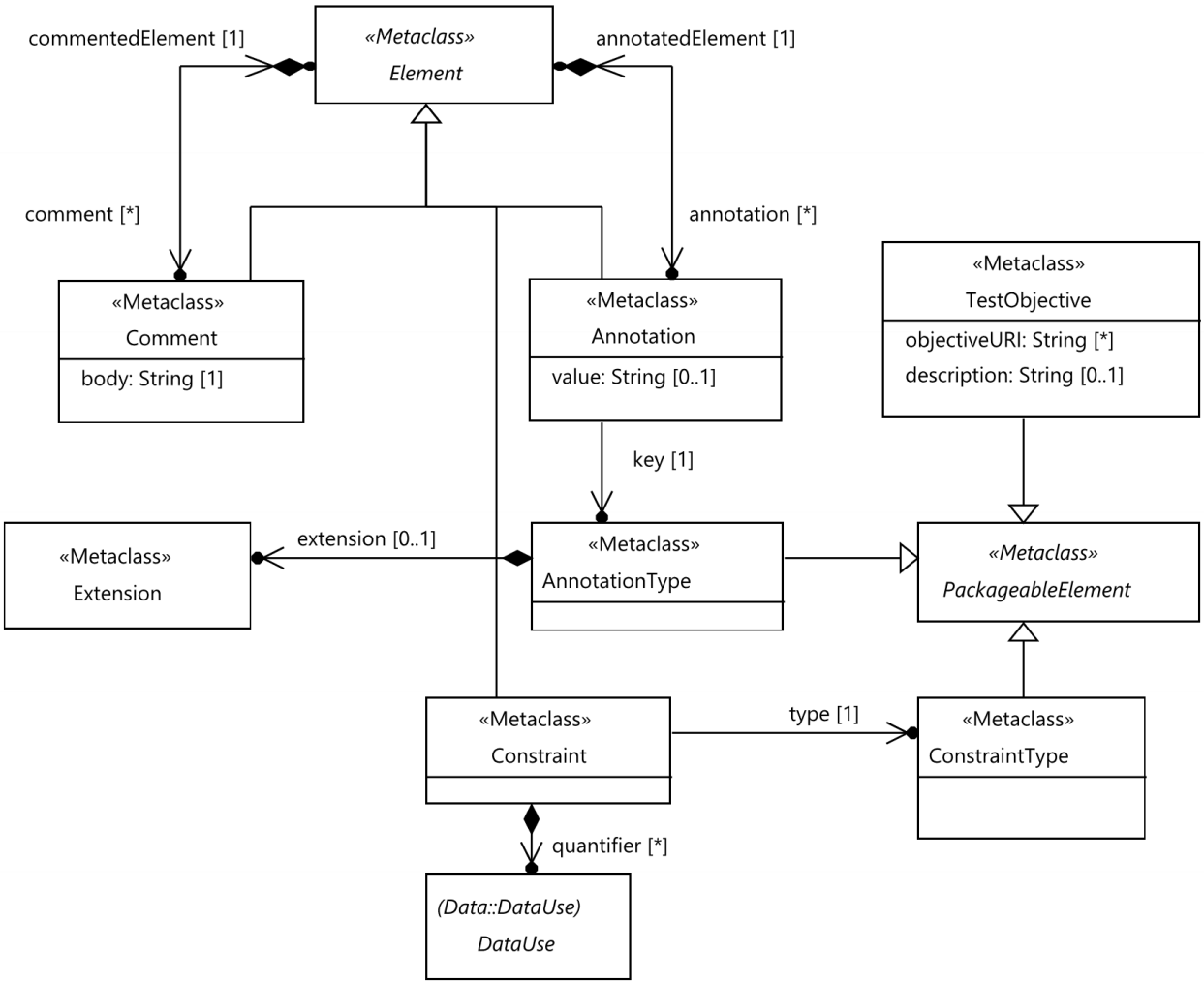

Figure 5.2: Miscellaneous elements

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. This property is deprecated in favour of the 'container()' operation.

- 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. This property is deprecated in favour of the 'container()' operation.

- 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.

An 'AnnotationType' may specialize another 'AnnotationType' via 'extension' property. The semantics of annotation type specialization are outside of the scope of TDL.

Generalization

- PackageableElement

Properties

- extension: Extension [0..1]
  Optional 'Extension' with reference to an 'AnnotationType' that this 'AnnotationType' specializes.

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.

5.2.10 Extension

Semantics

An 'Extension' shall declare an inheritance relation between two elements of the same meta-class. The meta-classes that support inheritance contain a property of type 'Extension'. The 'extending' property of 'Extension' shall refer to the element that is being extended.

Inheritance shall be transitive. That is, properties inherited by an element via 'extension' shall also be inherited by any element that extends this element. The properties that shall be acquired via inheritance are specified by the respective meta-classes. No properties shall be acquired by default.

NOTE: Name uniqueness constraints often prevent hiding or overriding inherited properties in extended elements.

Generalization

- Element

Properties

- extending: PackageableElement [1]
  The 'PackageableElement' that is extended.

Constraints

- **Inherit from element of the same meta-class**
  The element containing an 'Extension' and the element in the 'extending' property shall have the same meta-class.
  inv: **Extension**:
  self.container().oclType() = self.extending.oclType()

5.2.11 ConstraintType

Semantics

'ConstraintType's are used as classifiers for 'Constraint's. Predefined 'ConstraintType's with associated semantics for TDL are specified in clause 10.7. Additional domain or tool specific 'ConstraintType's may be created.

Generalization

- PackageableElement

Properties

There are no properties specified.

Constraints

There are no constraints specified.

5.2.12 Constraint

Semantics

A 'Constraint' is used to specify additional constraining properties for an 'Element'. Semantics and syntactical constraints for a 'Constraint' element are defined by its 'type'. The rules applied to an 'Element' by associated 'Constraint's are equivalent to syntactical constraints defined for meta-classes.

The meta-classes that support constraints shall contain a property of type 'Constraint'. Inheritance is specified for each meta-class individually.

NOTE: Constraints are similar to annotations. The distinction is introduced to support special syntaxes for constraints.

Generalization

- Element

Properties

- type: ConstraintType [1]
  The type that classifies this 'Constraint'.

- quantifier: DataUse [0..*] {ordered}
  Optional, 'type'-specific arguments for the 'Constraint'.

Constraints

- **Effectively static quantifiers**
  All 'DataUse's specified as 'quantifier's shall be effectively static.
  inv: **StaticQuantifiers**:
  self.quantifier->forAll(q | q.isEffectivelyStatic())

- **Empty arguments for quantifiers**
  The 'argument' sets for all 'DataUse's specified as 'quantifier's shall be empty.
  inv: **NoArgumentQuantifiers**:
  self.quantifier->forAll(q | q.argument->isEmpty())

- **Constraint applicability for 'union' and 'uniontype'**
  The predefined 'ConstraintType's 'union' and 'uniontype' shall be applied to 'StructuredDataType's only.
  inv: **ConstraintApplicabilityUnionUniontype**:
  (self.type.name = 'union' or self.type.name = 'uniontype')
  implies self.container().oclIsTypeOf(StructuredDataType)

- **Constraint applicability for 'range' and 'format'**
  The predefined 'ConstraintType's 'range' and 'format' shall be applied to 'SimpleDataType's and 'Member's with a 'SimpleDataType' 'dataType' only.
  inv: **ConstraintApplicabilityRangeFormat**:
  (self.type.name = 'range' or self.type.name = 'format')
  implies (self.container().oclIsTypeOf(SimpleDataType)
  or (self.container().oclIsTypeOf(Member)
  and self.container().oclAsType(Member).dataType.oclIsTypeOf(SimpleDataType)))

- **Constraint applicability for 'length', 'minLength', and 'maxLength'**
  The predefined 'ConstraintType's 'length', 'minLength', and 'maxLength' shall be applied to 'CollectionDataType's, 'SimpleDataType's and 'Member's with a 'SimpleDataType' or a 'CollectionDataType' 'dataType' only.
  inv: **ConstraintApplicabilityLength**:
  (self.type.name = 'length' or self.type.name = 'minLength' or self.type.name = 'maxLength')
  implies (self.container().oclIsTypeOf(SimpleDataType)
  or self.container().oclIsTypeOf(CollectionDataType)
  or (self.container().oclIsTypeOf(Member)
  and self.container().oclAsType(Member).dataType.oclIsTypeOf(SimpleDataType))
  and self.container().oclAsType(Member).dataType.oclIsTypeOf(CollectionDataType))

- **Quantifiers for 'length', 'minLength', and 'maxLength'**
  The predefined 'ConstraintType's 'length', 'minLength', and 'maxLength' shall be used with exactly one 'quantifier' resolved to an instance conforming to the predefined 'Integer' 'DataType'.
  inv: **ConstraintQuantifierLength**:
  (self.type.name = 'length' or self.type.name = 'minLength' or self.type.name = 'maxLength')
  implies (self.quantifier->size() = 1
  and self.quantifier->forAll(q | q.resolveDataType().conformsTo('Integer')))

- **Quantifiers for 'range'**
  The predefined 'ConstraintType' 'range' shall be used with exactly two 'quantifier's resolved to instance conforming to the predefined 'Integer' 'DataType'.
  inv: **ConstraintQuantifierRange**:
  (self.type.name = 'length')
  implies (self.quantifier->size() = 2
  and self.quantifier->forAll(q | q.resolveDataType().conformsTo('Integer')))

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

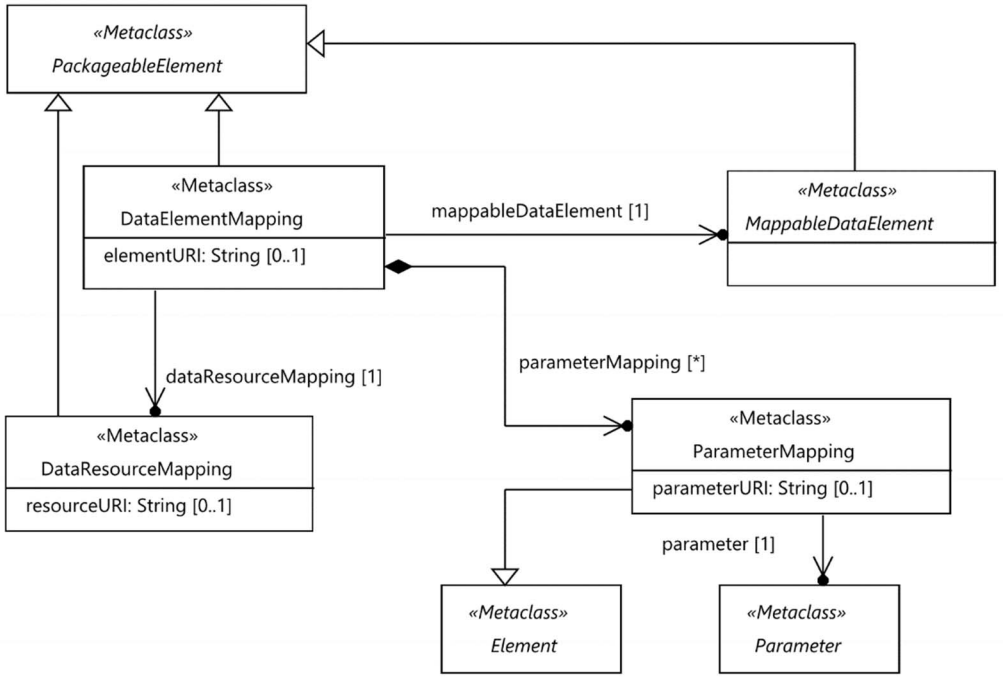

Figure 6.1: Data mapping concepts

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 'Member's (in the first case) or 'Parameters' (in the other cases) to concrete data representations explicitly.

If the 'DataElementMapping' refers to a 'StructuredDataType' that extends another 'StructuredDataType' then 'parameterMapping's from the 'DataElementMapping' that refers to the extended type are inherited. A 'member' from the extended 'StructuredDataType' may be mapped by means of a 'DataElementMapping' that refers to the extending type.

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.oclAsType(StructuredDataType).allMembers()->includes(p.parameter)))
  or (self.mappableDataElement.oclIsKindOf(Action)
  and self.parameterMapping->forAll(p |
  self.mappableDataElement.oclAsType(Action).formalParameter->includes(p.parameter)))

- **All parameters shall be mapped**
  If the 'mappableDataElement' refers to a 'StructuredDataType', an 'Action' or a 'Function' definition, all the 'Parameters' contained in the 'mappableDataElement' shall be mapped.
  inv: **ParameterMappings**:
  (self.mappableDataElement.oclIsKindOf(SimpleDataType) or
  self.mappableDataElement.oclIsKindOf(DataInstance)
  or (self.mappableDataElement.oclIsTypeOf(StructuredDataType)
  and self.mappableDataElement.oclAsType(StructuredDataType).member->forAll(p |
  self.parameterMapping->exists(m | m.parameter = p)))
  or (self.mappableDataElement.oclIsKindOf(Action)
  and self.mappableDataElement.oclAsType(Action).formalParameter->forAll(p |
  self.parameterMapping->exists(m | m.parameter = p))
  and self.parameterMapping->forAll(p |
  self.mappableDataElement.oclAsType(Action).formalParameter->includes(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' or 'ProcedureParameter' of a 'ProcedureSignature') to be mapped to a concrete data representation.

Constraints

There are no constraints specified.

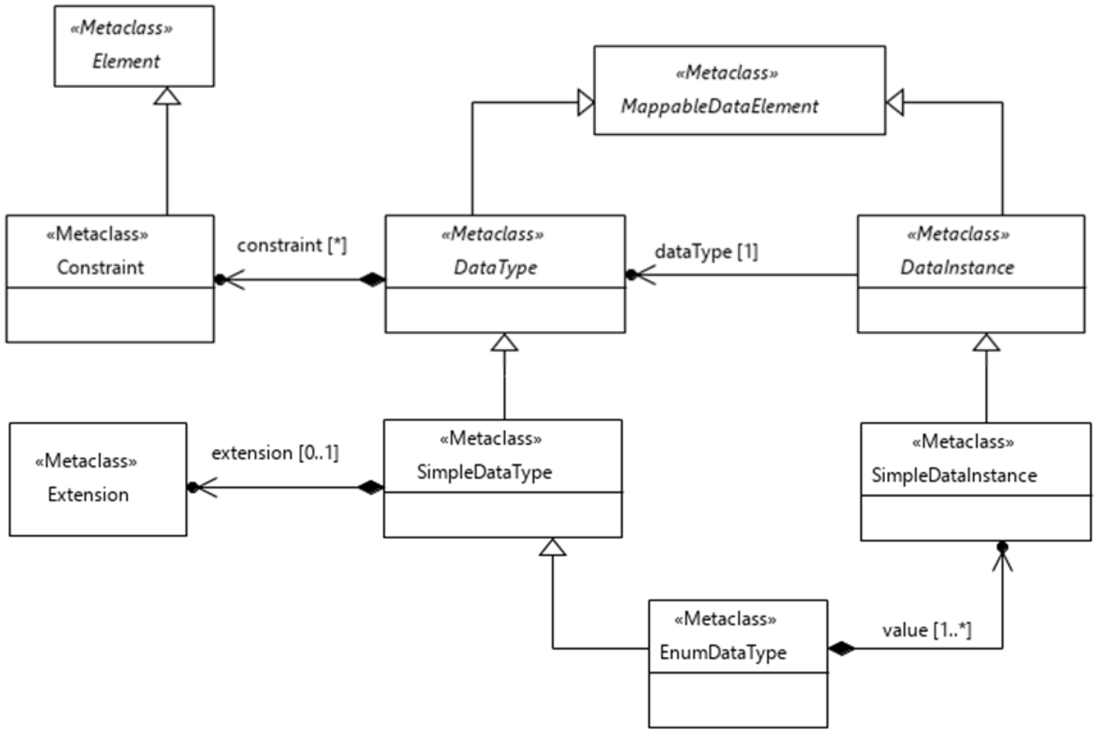

Figure 6.2: Basic data concepts and simple data

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 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

- constraint: Constraint [0..*] {ordered, unique}
  The contained ordered set of constraints that are associated with this data type.

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).

A 'SimpleDataType' may extend another 'SimpleDataType' via the 'extension' property. The 'SimpleDataType' shall acquire all 'Constraint's from the extended 'SimpleDataType' as well as any transitively inherited 'Constraint's from the extended 'SimpleDataType'.

Generalization

- DataType

Properties

- extension: Extension [0..1]
  Optional 'Extension' with a reference to a 'SimpleDataType' that this "SimpleDataType' extends.

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)

- **SimpleDataInstance container in EnumDataType**
  A 'SimpleDataInstance' whose 'dataType' property refers to an 'EnumDataType' shall be contained in that 'EnumDataType'.
  inv: **EnumDataInstanceContainment**:
  not self.dataType.oclIsKindOf(EnumDataType) or self.oclContainer() = self.dataType

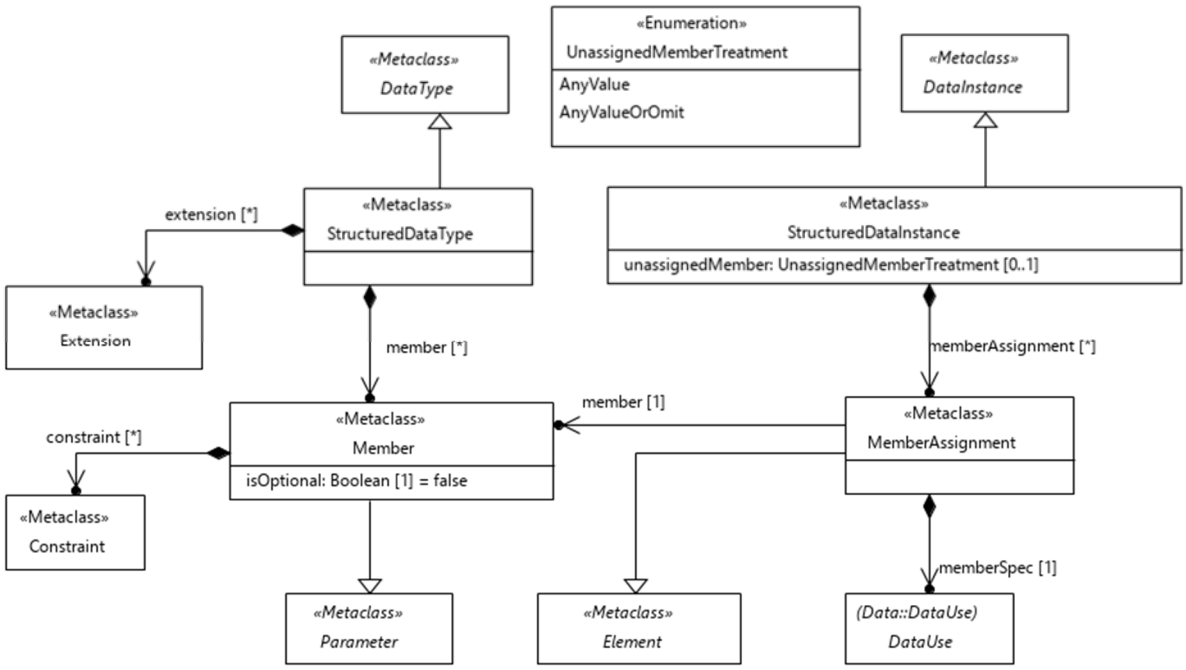

Figure 6.3: Structured data type and instance

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.

A 'StructuredDataType' may extend other 'StructuredDataType's via the 'extension' property. The 'StructuredDataType' shall acquire all 'Member's and 'Constraint's from extended 'StructuredDataType's as well as any transitively inherited 'Members's and 'Constraint's from the extended 'StructuredDataType's.

The inheritance mechanism may be altered by adding 'Constraint's of predefined types to a 'StructuredDataType' (see clause 10.7).

Generalization

- DataType

Properties

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

- extension: Extension [0..*]
  Optional 'Extension's with references to one or more 'StructuredDataType's that this 'StructuredDataType' extends.

Constraints

- **Different member names in a structured data type**
  All 'Member' names of a 'StructuredDataType' (including the names of inherited 'Members') shall be distinguishable.
  inv: **DistinguishableMemberNames**:
  self.allMembers()->isUnique(e | e.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 of the present document).

Generalization

- Parameter

Properties

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

- constraint: Constraint [0..*] {ordered, unique}
  The contained ordered set of 'Constraint's that are applied to this member.

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', except when the 'DataUse's specify their own 'UnassignedMemberTreatment'. It is applied recursively.

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

- 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. If not specified, the default value 'undefined' is assumed.

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 or inherited by the 'StructuredDataType' that the 'StructuredDataInstance', which contains this 'MemberAssignment', refers to.
  inv: **ExistingMemberOfDataType**:
  self.memberAssignment->forAll(a | self.dataType.oclAsType(StructuredDataType).allMembers()->includes(a.member))

- **Unique assignments for each 'Member' of the 'StructuredDataType'**
  There shall be at most one 'memberAssignment' for each 'Member' of the 'StructuredDataType' that the 'StructuredDataInstance', which contains this 'MemberAssignment', refers to.
  inv: **UniqueMemberAssignments**:
  self.memberAssignment->isUnique(m | m.member)

- '**union' constraint on the type of the 'StructuredDataInstance'**
  If the 'dataType' of the 'StructuredDataInstance' has the predefined constraint 'union' then the 'memberAssignment' shall not contain more than one 'MemberAssignment'.
  inv: **StructuredDataInstanceUnionConstraint**:
  not self.dataType.allConstraints()->exists(c | c.type.name = 'union')
  or self.memberAssignment->size() <= 1

- **'uniontype' constraint on the type of the 'StructuredDataInstance'**
  If the 'dataType' of 'StructuredDataInstance' has the predefined constraint 'uniontype' then there shall only be 'MemberAssignment' for the 'Member's of the 'dataType' itself or of at most one of the 'StructuredDataType's which the 'dataType' is extending.
  inv: **StructuredDataInstanceUniontypeConstraint**:
  not self.dataType.allConstraints()->exists(c | c.type.name = 'uniontype')
  or self.memberAssignment->forAll(m | self.dataType.oclAsType(StructuredDataType).member->includes(m)
  or self.dataType.oclAsType(StructuredDataType).extension->one(e |
  e.extending.oclAsType(StructuredDataType).allMembers()->includes(m)))

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 contained or inherited 'Member' of the 'StructuredDataType' definition that is referenced via the 'dataType' property of the 'StructuredDataInstance'.

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

Constraints

- **Type of a 'memberSpec' shall conform to the type of the 'member'**
  The 'DataType' of the 'DataUse' of 'memberSpec' shall conform to the 'DataType' of the 'Member' of the 'MemberAssignment'.
  inv: **MatchingMemberDataType**:
  self.memberSpec.resolveDataType().conformsTo(self.member.dataType)

- **Restricted use of 'OmitValue' for optional 'Member's only**
  A non-optional 'Member' shall have a 'DataUse' 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 'memberSpec'**
  The 'memberSpec' and all of its 'ParameterBinding's shall be effectively static.
  inv: **StaticDataInMemberSpec:**
  self.memberSpec.isEffectivelyStatic()

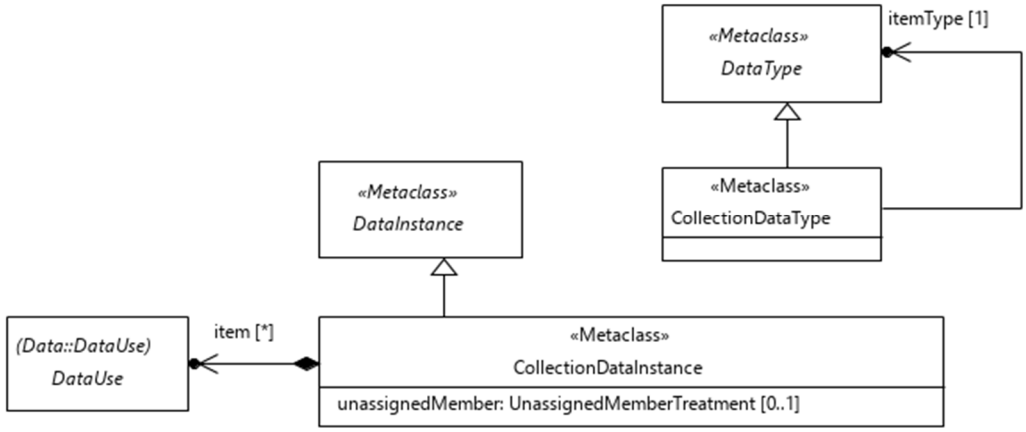

Figure 6.4: Collection data

6.2.13 CollectionDataType

Semantics

A 'CollectionDataType' defines a type for collections of 'DataInstance's of a specific 'DataType'.

Generalization

- DataType

Properties

- itemType: DataType [1]
  The type of the items that shall be contained in collections corresponding to this 'CollectionDataType'.

Constraints

- **No multidimensional collections**
  The 'itemType' shall not be an instance of 'CollectionDataType'.
  inv: **NoMultidimensionalCollections**:
  not self.itemType.oclIsKindOf(CollectionDataType)

6.2.14 CollectionDataInstance

Semantics

A 'CollectionDataInstance' represents an ordered set of symbolic values of a 'DataType'. The type of items is defined by item type of the associated 'CollectionDataType'. The 'CollectionDataInstance' contains 'DataUse's that represent those values. The optional 'unassignedMember' property may be used to override the semantics of unassigned 'Member's for 'StructuredDataInstance's referenced as items of the 'CollectionDataInstance'. If the 'unassignedMember' property is provided, then unassigned 'Member's shall be treated according to the semantics of the provided 'UnassignedMemberTreatment', except when the 'DataUse's specify their own 'UnassignedMemberTreatment'. It is applied recursively.

NOTE: Items of a 'CollectionDataInstance' are not mappable via index references but only as independent 'DataInstance's.

Generalization

- DataInstance

Properties

- item: DataUse [0..*] {ordered}
  List of contained 'DataUse's that define items in this collection.

- unassignedMember: UnassignedMemberTreatment [0..1]
  Optional indication of how unassigned 'Members' shall be interpreted. If not specified, the default value 'undefined' is assumed.

Constraints

- **CollectionDataInstance shall refer to CollectionDataType**
  The inherited reference 'dataType' from 'DataInstance' shall refer to instances of 'CollectionDataType' solely.
  inv: **CollectionDataInstanceType**:
  self.dataType.oclIsKindOf(CollectionDataType)

- **Type of items in the 'CollectionDataInstance'**
  The items in 'CollectionDataInstance' shall conform to the 'itemType' of the 'CollectionDataType' that is defined as the 'dataType' of this 'CollectionDataInstance'.
  inv: **CollectionDataInstanceItemType**:
  self.item->forAll(i |
  i.resolveDataType().conformsTo(self.dataType.oclAsType(CollectionDataType).itemType))

- **Static data use in 'item'**
  The DataUse's in 'item' and all of the respective 'ParameterBinding's shall be effectively static.
  inv: **StaticDataInItem**:
  self.item->forAll(i | i.isEffectivelyStatic())

- **Length constraint of the 'CollectionDataInstance'**
  If the 'dataType' 'CollectionType' contains the predefined constraint 'length' then the length of this 'CollectionDataInstance' shall be equal to the 'quantifier' of that 'Constraint'.
  inv: **CollectionDataInstanceLengthConstraint**:
  This constraint cannot be expressed formally.

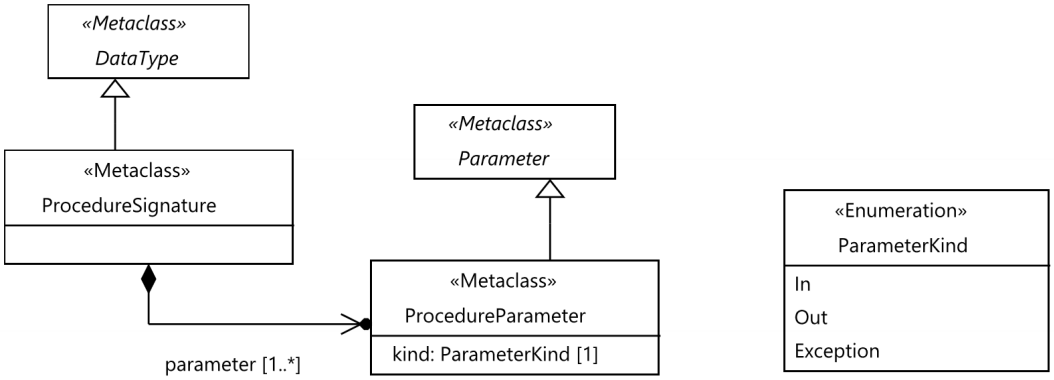

Figure 6.5: Procedure and procedure parameter

6.2.15 ProcedureSignature

Semantics

A 'ProcedureSignature' is a specification of a remote procedure call signature. A 'ProcedureSignature' specifies one or more input and output parameters as well as any exceptional values that can be returned.

Generalization

- DataType

Properties

- parameter: ProcedureParameter [1..*]
  Ordered set of formal parameters of the 'ProcedureSignature'.

Constraints

There are no constraints specified.

6.2.16 ProcedureParameter

Semantics

A 'ProcedureParameter' is a declaration of input or output of a 'ProcedureSignature'.

Generalization

- Parameter

Properties

- kind: ParameterKind [1]
  Specifies the direction and nature of the 'ProcedureParameter'.

Constraints

There are no constraints specified.

6.2.17 ParameterKind

Semantics

A 'ParameterKind' specifies the direction and nature of a 'ProcedureParameter'.

Generalization

There is no generalization specified.

Literals

- In
  Indicates that the 'ProcedureParameter' shall be used only in 'ProcedureCall's which are not replies to other 'ProcedureCall's.

- Out
  Indicates that the 'ProcedureParameter' shall be used only in 'ProcedureCall's which are replies to other 'ProcedureCall's.

- Exception
  Indicates that the 'ProcedureParameter' shall be used only in 'ProcedureCall's which are replies to other 'ProcedureCall's, to indicate exceptions.

Constraints

There are no constraints specified.

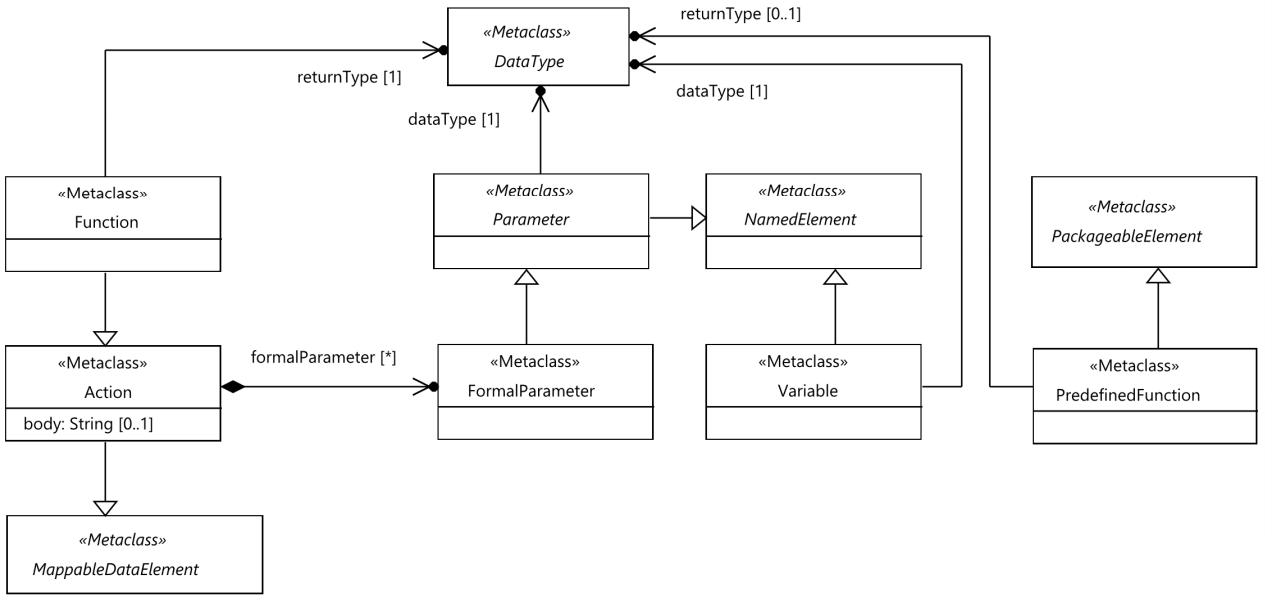

Figure 6.6: Action, function, parameter and variable

6.2.18 Parameter

Semantics

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

Generalization

- NamedElement

Properties

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

Constraints

There are no constraints specified.

6.2.19 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.

6.2.20 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 8.2.5), 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.

6.2.21 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.

6.2.22 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.

6.2.23 UnassignedMemberTreatment

Semantics

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

Generalization

There is no generalization specified.

Literals

- undefined
  Unassigned 'Members' shall be interpreted as <undefined>. This is the assumed default value.
- AnyValue
  Unassigned 'Members' shall be interpreted as 'AnyValue'.
- AnyValueOrOmit
  Unassigned non-optional 'Members' shall be interpreted as 'AnyValue'. Unassigned optional 'Members' shall be interpreted as 'AnyValueOrOmit'.

Constraints

There are no constraints specified.

6.2.24 PredefinedFunction

Semantics

A 'PredefinedFunction' is a 'PackageableElement' that has a return value. 'PredefinedFunction's provide means for expressing generic calculations over 'DataInstance's within a 'TestDescription' at runtime. The formal parameters of each 'PredefinedFunction' are specified on a higher level, and thus not bound to specific 'DataType's. The 'DataType' of the returned value may be optionally specified for each 'PredefinedFunction', otherwise the specification of the 'PredefinedFunction' shall describe how the 'DataType' of the returned value shall be derived from the actual parameters of the 'PredefinedFunction'. 'PredefinedFunction's as specified in clause 10.5 shall be implemented by tools. Similar to a 'Function', the execution of a 'PredefinedFunction' is side-effect free.

Generalization

- PackageableElement

Properties

- returnType: DataType [0..1]
  Optionally specified 'DataType' of the 'DataInstance' that is returned when the 'PredefinedFunction' finished its calculation.

Constraints

There are no constraints specified.

6.2.25 EnumDataType

Semantics

An 'EnumDataType' specifies a limited set of 'SimpleDataInstance's. No other 'SimpleDataInstance's shall exist that refer to an 'EnumDataType' than the ones that are contained by it. 'EnumDataType' shall not have extensions and it shall not be extended.

Generalization

- SimpleDataType

Properties

- value: SimpleDataInstance [1..*]
  The set of allowed values of this enumerated type.

Constraints

- **No extensions for EnumDataType**
  The 'extension' property of an 'EnumDataType' shall be empty.
  inv: **EnumDataTypeExtensions**:
  self.extension.oclIsUndefined()

6.3 Data Use - Abstract Syntax and Classifier Description

6.3.1 DataUse

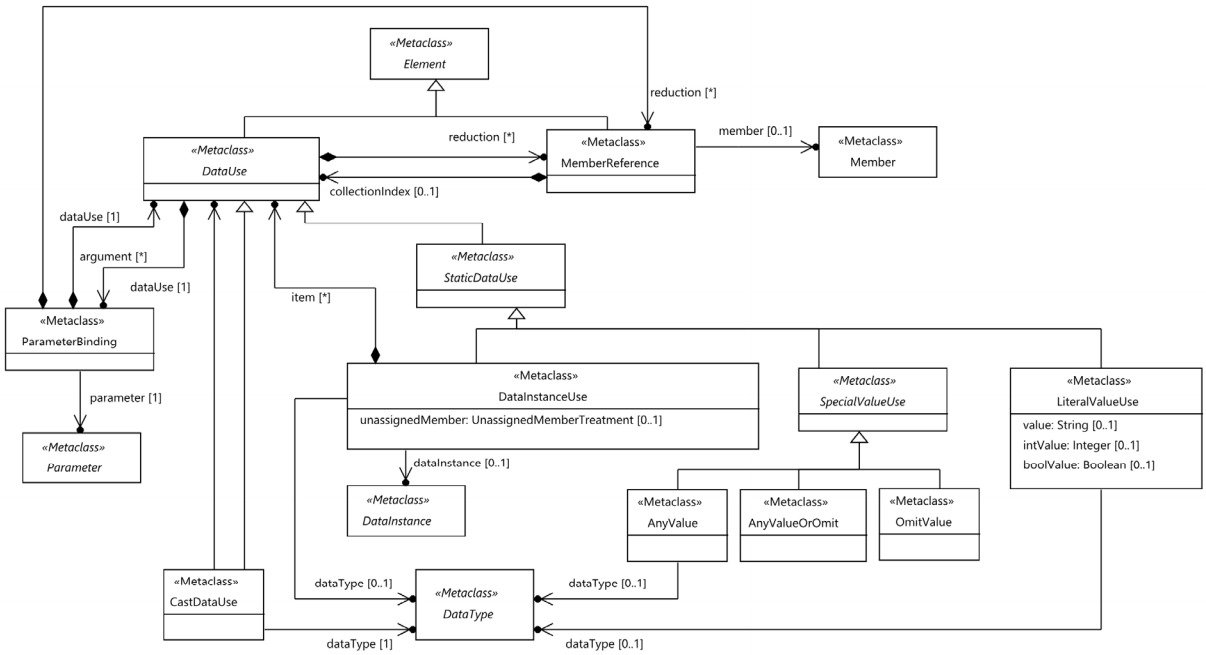

Figure 6.7: Data use concepts and static data use

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. Subclasses 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 subclasses.

A 'DataUse' is used in a scope that is defined by the containing 'AtomicBehaviour' or 'LocalExpression'. The scope specifies a set of 'componentInstance's whose properties are accessible for the 'DataUse'. Additional limitations are specified in the following clauses.

A 'DataUse' offers the capability to be parameterized. This is achieved by the use of a 'ParameterBinding'. The scope for 'DataUse's in 'argument' is the same as the scope for this 'DataUse'.

In case the 'DataUse' evaluates to a 'StructuredDataInstance' or 'CollectionDataInstance', a *location expression* over nested 'StructuredDataInstance's and 'CollectionDataInstances' may be specified in order to reduce the 'DataUse' to the symbolic value contained in a potentially nested 'Member' or item in a collection. This is called *reduction*. The reduction is semantically equivalent to the dot-notation or array access expression typically found in programming languages, e.g. in Java or TTCN-3, in order to navigate from the top level object, i.e. the 'DataInstance', which this 'DataUse' evaluates to at runtime, to a specific location. The starting point of a location expression is the implicitly or explicitly referenced 'DataInstance' obtained after the 'DataUse' has been evaluated at runtime.

Multiple 'MemberReference's may be used incrementally. If the first 'MemberReference' in the list specifies collectionIndex then its member shall be unspecified. In this case the value for which the 'MemberReference' applies to shall be a collection.

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 concrete subclass of this 'DataUse'.

- reduction: MemberReference [0..*] {ordered, unique}
  Location expression that refers to potentially nested 'Member's of a 'StructuredDataType' and collection item indexes. Each contained 'MemberReference' 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', or a 'DataElementUse' which refers to a 'Function', 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)
  or (self.oclIsTypeOf(DataElementUse)
  and self.oclAsType(DataElementUse).dataElement.oclIsTypeOf(Function))

- **Members in 'reduction' list**
  The 'Member' referenced by the 'MemberReference' at index *i* of a 'reduction' shall be contained in or inherited by the 'StructuredDataType' of the 'Member' referenced by the 'MemberReference' at index (*i* - 1) of that 'reduction' or the 'StructuredDataType' of this 'DataUse' if the 'MemberReference' is the first element of the 'reduction'.
  inv: **ReductionMembers**:

```
self.reduction->isEmpty()
  or self.reduction->reject(r | self.reduction->indexOf(r) = 1)
  ->iterate(r; acc = Sequence{Sequence{self.resolveBaseDataType(), self.reduction->at(1)}}
    | acc->including(Sequence{self.reduction->at(self.reduction->indexOf(r) - 1)->asSequence()
      ->reject(r|r.member.oclIsUndefined()).member.dataType->including(self.resolveBaseDataType())->at(1), r
    }))
  ->reject(tr | tr->at(1).oclIsUndefined() or tr->at(2).oclAsType(MemberReference).member.oclIsUndefined())
  ->iterate(tr; acc = Sequence{} | acc->including(Sequence{tr->at(1)->asSequence()
    ->select(t|t.oclIsKindOf(CollectionDataType)).oclAsType(CollectionDataType).itemType->including(tr->at(1))->at(1), tr->at(2)
  }))
  ->forAll(tr | tr->at(1).oclAsType(StructuredDataType).allMembers()->includes(tr->at(2).oclAsType(MemberReference).member))
```

- **Collection index of first 'reduction'**
  If the 'member' is not specified for the first element of the 'reduction' then the type of this 'DataUse' shall be 'CollectionDataType'.
  inv: **ReductionMembers**:

```
self.reduction->isEmpty()
  or not self.reduction->at(1).member.oclIsUndefined()
  or self. resolveBaseDataType().oclIsKindOf(CollectionDataType)
```

6.3.2 ParameterBinding

Semantics

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

In case the 'dataType' of the 'parameter' is a 'StructuredDataType' or a 'CollectionDataType', a *location expression* over the 'Member's of the 'StructuredDataType' and the items of 'CollectionDataType' may be specified in order to *reduce* the assignment of the symbolic value of the 'DataUse' specification to a nested element of the 'parameter'.

The original value assigned to the 'parameter' shall be completely overwritten by the 'dataUse'. This shall occur even if the specified 'dataUse' is not fully specified, i.e. some parameter or member values have been omitted or are undefined.

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 shall be assigned the symbolic value of a 'dataUse' specification.

- reduction: MemberReference [0..*] {ordered, unique}
  Location expression that refers to potentially nested 'Member's of a 'StructuredDataType' and collection item indexes. Each contained 'MemberReference' of the ordered set represents one fragment of the location expression. The location expression is applied on the 'parameter' to specify a potentially nested 'Member' which shall be assigned the symbolic value of the 'dataUse' specification.

Constraints

- **Members in 'reduction' list of parameter**
  The 'Member' referenced by the 'MemberReference' at index *i* of a 'reduction' shall be contained in or inherited by the 'StructuredDataType' of the 'Member' referenced by the 'MemberReference' at index (*i* - 1) of that 'reduction' or the 'StructuredDataType' of the 'parameter' if the 'MemberReference' is the first element of the 'reduction'.
  inv: **ReductionMembers**:

```
self.reduction->isEmpty()
    or self.reduction->reject(r | self.reduction->indexOf(r) = 1)
    ->iterate(r; acc = Sequence{Sequence{self.parameter.dataType, self.reduction->at(1)}}
        | acc->including(Sequence{self.reduction->at(self.reduction->indexOf(r) - 1)->asSequence()
            ->reject(r|r.member.oclIsUndefined()).member.dataType->including(self.parameter.dataType)-
>at(1), r
        }))
    ->reject(tr | tr->at(1).oclIsUndefined() or tr-
>at(2).oclAsType(MemberReference).member.oclIsUndefined())
    ->iterate(tr; acc = Sequence{} | acc->including(Sequence{tr->at(1)->asSequence()
        ->select(t|t.oclIsKindOf(CollectionDataType)).oclAsType(CollectionDataType).itemType-
>including(tr->at(1))->at(1), tr->at(2)
    }))
    ->forAll(tr | tr->at(1).oclAsType(StructuredDataType).allMembers()->includes(tr-
>at(2).oclAsType(MemberReference).member))
```

- **Collection index of first 'reduction' of parameter**
  If the 'member' is not specified for the first element of the 'reduction' then the type of the 'parameter' shall be 'CollectionDataType'.
  inv: **ReductionMembers**:

```
self.reduction->isEmpty()
    or not self.reduction->at(1).member.oclIsUndefined()
    or self.parameter.dataType.oclIsKindOf(CollectionDataType)
```

- **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 MemberReference

Semantics

A 'MemberReference' points to a single data value within a structure or a collection by specifying an index of a collection item or a member of a structured type or both. The 'collectionIndex' shall be any 'DataUse' that resolves to an integral value at runtime. Member shall be a contained or inherited 'Member' of the 'StructuredDataType' of the structured data value that the 'MemberReference' applies to. If both member and collectionIndex are specified then member is applied first and the collectionIndex is used to access the item in the collection assigned to that member.

Generalization

- Element

Properties

- member: Member [0..1]
  A 'Member' of a 'StructuredDataType' that is the type of the value to which this 'MemberReference' applies to.

- collectionIndex: DataUse [0..1]
  The value of the index expression defines the item in the collection which is selected for use.

Constraints

- **Collection index expressions for collections only**
  If the 'member' is specified and the type of the 'member' is not 'CollectionDataType' then the collectionIndex shall be undefined.
  inv: **CollectionIndex**:
  self.collectionIndex.oclIsUndefined()
  or self.member.oclIsUndefined()
  or self.member.dataType.oclIsKindOf(CollectionDataType)

- **Either member or collection index is required**
  Either the member or collectionIndex shall be specified.
  inv: **MemberOrReduction**:
  not self.member.oclIsUndefined() or not self.collectionIndex.oclIsUndefined()

6.3.4 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.

6.3.5 DataInstanceUse

Semantics

A 'DataInstanceUse' specifies a 'DataInstance' in a data usage context. It may refer to a 'SimpleDataInstance', a 'StructuredDataInstance', or a 'CollectionDataInstance'. In case no 'DataInstance' is referenced, it shall provide 'ParameterBinding's as arguments or 'DataUse's as items, depending on whether the resolved 'DataType' is a 'StructuredDataType' or a 'CollectionDataType', respectively. An optional reference to a 'DataType' shall be provided if the 'DataInstanceUse' is used as the argument of 'Message' and no 'DataInstance' is provided. In other contexts the optional reference to a 'DataType' may be provided for clarity.

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's with new values evaluated from the provided 'DataUse' specifications. The inline modification is applicable only in the context where it is specified. The value of the original 'StructuredDataInstance' remains unchanged.

In case it refers to a 'CollectionDataInstance', the items in the referenced collection cannot be modified inline.

In case it does not refer to a 'DataInstance', a value for an anonymous 'DataInstance' specified inline by providing arguments as 'ParameterBinding's or by providing 'DataInstance's as items for an anonymous 'CollectionDataInstance'. The 'DataType' for the anonymous 'DataInstance' is either specified explicitly by means of the optional 'dataType' property, or implicitly, inferred from the context in which it is used, from the 'DataType' of the 'Member', 'Parameter', 'FormalParameter', or 'Variable' to which it is assigned. Since 'Extension's are not considered for implicitly inferred 'DataType's, if a 'DataType' extending the implicitly inferred 'DataType' is to be used, the extending 'DataType' shall be specified explicitly by means of the 'dataType' property in an anonymous 'DataInstance'.

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 to all contained 'Member's.

If a referenced 'CollectionDataInstance' contains 'StructuredDataInstance's as items which have no 'MemberAssignment's for 'Member's of the respective 'StructuredDataType's, it is assumed that the 'Member's have the value <undefined> assigned to them. The optional 'unassignedMember' property may be used to override the semantics of all unassigned 'Member's for all items in the referenced 'CollectionDataInstance' recursively.

NOTE: 'DataInstanceUse' may be deprecated in future editions of the present document in favour of 'DataElementUse'.

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. If not specified, the default value 'undefined' is assumed.

- item: DataUse [0..*] {ordered}
  List of contained 'DataUse's that define items in an anonymous 'DataInstance'.

Constraints

- **'DataInstance' reference or non-empty 'argument' or non-empty 'item'**
  If a 'dataInstance' is not specified, either a non-empty 'argument' set or a non-empty 'item' set shall be specified.
  inv: **DataInstanceOrArgumentsOrItems**:
  not self.dataInstance.oclIsUndefined() or not self.argument->isEmpty() or not self.item->isEmpty()

- **Valid 'DataType' for items**
  The items in the 'item' property shall conform to the 'itemType' of the resolved 'CollectionDataType'.
  inv: **DataTypeOfItemsInDataInstance**
  self.item->forAll(i |
  i.resolveDataType().conformsTo(self.resolveDataType().oclAsType(CollectionDataType).itemType))

- **No 'item' if 'dataInstance' is specified**
  The 'item' property shall be empty if the 'dataInstance' property is specified.
  inv: **NoItemWithDataInstance**
  not self.dataInstance.oclIsUndefined() implies self.item->isEmpty()

6.3.6 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()

6.3.7 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]
  Optional reference to a 'DataType'. This property is deprecated in favour of 'CastDataUse'.

Constraints

There are no constraints specified.

6.3.8 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.

6.3.9 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.

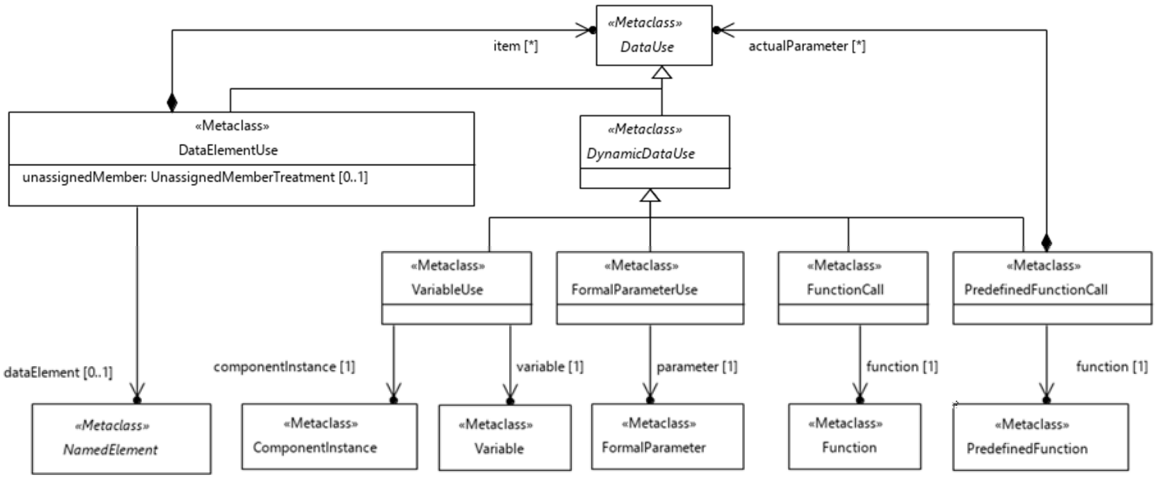

Figure 6.8: Dynamic data use

6.3.10 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.

6.3.11 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's.

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

NOTE: 'FunctionCall' may be deprecated in future editions of the present document in favour of 'DataElementUse'.

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))

6.3.12 FormalParameterUse

Semantics

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

NOTE: 'FormalParameterUse' may be deprecated in future editions of the present document in favour of 'DataElementUse'.

Generalization

- DynamicDataUse

Properties

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

Constraints

There are no constraints specified.

6.3.13 VariableUse

Semantics

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

In case of local ordering, the scope for a 'DataInstance' shall be equal to the 'componentInstance'.

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.allVariables()->includes(self.variable)
  and self.componentInstance.role = ComponentInstanceRole::Tester

6.3.14 PredefinedFunctionCall

Semantics

A 'PredefinedFunctionCall' specifies the invocation of a 'PredefinedFunction' with its arguments.

The actual parameters corresponding to the 'FormalParameter's of the invoked 'PredefinedFunction' as specified in clause 10.5 shall be provided in the 'PredefinedFunctionCall' by using 'ParameterBinding's.

The scope for 'DataUse's in 'actualParameters' is the same as the scope for this 'PredefinedFunctionCall'.

Generalization

- DynamicDataUse

Properties

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

- actualParameters: DataUse[0..*] {ordered}
  Contained ordered set of actual parameters passed to the predefined function.

Constraints

- **Compatible actual parameters**
  The number and type of actual parameters shall be compatible with the formal parameters of the invoked 'PredefinedFunction' according to the specification of the 'PredefinedFunction'.
  inv: **PredefinedFunctionCallParameters**:
  This constraint cannot be expressed formally.

- **Empty 'argument' and 'reduction' sets**
  The 'argument' and 'reduction' sets shall be empty.
  inv: **PredefinedFunctionCallArgumentReduction**:
  self.reduction->isEmpty() and self.argument->isEmpty()

6.3.15 LiteralValueUse

Semantics

A 'LiteralValueUse' specifies an inline literal that shall be specified in either string, integer or Boolean format. Any other formats shall be encoded as strings. The encoding of such data is outside the scope of TDL.

A 'DataType' may be specified if it is required by the context in which this 'LiteralValueUse' is used.

If a 'DataType' is not specified, the data type of the literal shall be resolved based on the context ('MemberAssignment', 'ParameterBinding') in which this 'LiteralValueUse' is used. In the absence contextual information, the data type shall be resolved to 'Integer' if the 'intValue' property is specified, 'Boolean' if the 'boolValue' property is specified, or 'String' if the 'value' property is specified. If the specified or resolved 'DataType' is a 'StructuredDataType', the 'reduction' and 'argument' properties may be used to access or override specific parts of the encoded literal, respectively.

Generalization

- StaticDataUse

Properties

- value: String [0..1]
  The literal as a string.
- intValue: Integer [0..1]
  The literal as an integer.
- boolValue: Boolean [0..1]
  The literal as a Boolean.
- dataType: DataType [0..1]
  Optional reference to a 'DataType'. This property is deprecated in favour of 'CastDataUse'.

Constraints

- **Exactly one value specification**
  There shall be exactly one value specification, where either the 'value', or the 'intValue', or the 'boolValue' property is specified, but not more than one of them.
  inv: **SpecifiedLiteralValue**:
  not self.value.oclIsUndefined()
  xor not self.intValue.oclIsUndefined()
  xor not self.boolValue.oclIsUndefined()
- **Empty 'argument' and 'reduction' sets if not 'dataType'**
  If 'dataType' is not specified then the 'argument' and 'reduction' sets shall be empty.
  inv: **LiteralValueArgumentReduction**:
  (self.dataType.oclIsUndefined())
  implies (self.reduction->isEmpty() and self.argument->isEmpty())
- **Integer type for integer value**
  If 'intValue' is specified then the 'dataType' is either unspecified or the specified 'DataType' is instance of 'Time' or conforms to predefined type 'Integer'.
  inv: **LiteralValueIntType**:
  not self.intValue.oclIsUndefined()
  implies (self.dataType.oclIsUndefined() or self.dataType.oclIsKindOf(Time) or self.dataType.conformsTo('Integer'))
- **Boolean type for Boolean value**
  If 'boolValue' is specified then the 'dataType' is either unspecified or the specified 'DataType' conforms to predefined type 'Boolean'.
  inv: **LiteralValueBoolType**:
  not self.boolValue.oclIsUndefined()
  implies (self.dataType.oclIsUndefined() or self.dataType.conformsTo('Boolean'))

6.3.16 DataElementUse

Semantics

A 'DataElementUse' denotes a generic and unified expression that evaluates to a 'DataInstance' of a given 'DataType'. A 'DataElementUse' unifies the specification of symbolic values that may be used in assignments and invocations, including referencing specified 'DataInstance's, 'FormalParameter's, and 'Function's to be called. Additionally, by referencing 'StructuredDataType's and 'CollectionDataType's, anonymous 'DataInstance's of such 'DataType's can be specified inline. If a 'StructuredDataType' or a 'CollectionDataType' can be inferred from the context, an anonymous 'DataInstance's of such 'DataType' can also be specified inline without explicitly indicating the 'DataType' as the 'dataElement' for the 'DataElementUse'. All constraints for the corresponding kind of data element are applicable. While the property type for the 'dataElement' is the generic 'NamedElement' in order to be able to refer to all the different kinds of data elements, only 'DataType's, 'DataInstance's, 'FormalParameter's, and 'Function's shall be referenced as 'dataElement's in 'DataElementUse's.

In case it refers to a 'StructuredDataInstance', a 'FormalParameter', or a 'Function', the corresponding value may be modified inline by providing arguments as 'ParameterBinding's. This allows replacing the current value of the referenced 'Member's with new values evaluated from the provided 'DataUse' specifications. The inline modification is applicable only in the context where it is specified. The value of the original 'StructuredDataInstance', 'FormalParameter', or returned from the execution of the 'Function' remains unchanged.

In case it refers to a 'CollectionDataInstance', the items in the referenced collection cannot be modified inline.

In case it does not refer to a data element or in case it refers to a 'StructuredDataType', a value for an anonymous 'DataInstance' is specified inline by providing arguments as 'ParameterBinding's or by providing 'DataInstance's as items for an anonymous 'CollectionDataInstance'. The 'DataType' for the anonymous 'DataInstance' is either specified explicitly by means of the 'dataElement' property referring to a 'StructuredDataType', or implicitly, inferred from the context in which it is used, from the 'DataType' of the 'Member', 'Parameter', 'FormalParameter', or 'Variable' to which it is assigned. Since 'Extension's are not considered for implicitly inferred 'DataType's, if a 'DataType' extending the implicitly inferred 'DataType' is to be used, the extending 'DataType' shall be specified explicitly by means of the 'dataElement' property in an anonymous 'DataInstance'.

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 to all contained 'Member's.

If a referenced 'CollectionDataInstance' contains 'StructuredDataInstance's as items which have no 'MemberAssignment's for 'Member's of the respective 'StructuredDataType's, it is assumed that the 'Member's have the value <undefined> assigned to them. The optional 'unassignedMember' property may be used to override the semantics of all unassigned 'Member's for all items in the referenced 'CollectionDataInstance' recursively.

If a referenced 'Function' has declared 'FormalParameter's the corresponding arguments shall be specified by using 'ParameterBinding's. If a 'reduction' is provided when a 'Function' is referenced, it applies to the return value of the 'Function', which implies that the return value is of 'StructuredDataType' or 'CollectionDataType'.

NOTE: Other specialized subclasses of 'DataUse', such as 'FormalParameterUse', 'FunctionCall', and 'DataInstanceUse' may be deprecated in future editions of the present document.

Generalization

- DataUse

Properties

- dataElement: NamedElement [0..1]
  The literal as a string.

- item: DataUse [0..*] {ordered}
  List of contained 'DataUse's that define items in an anonymous 'DataInstance'.

- unassignedMember: UnassignedMemberTreatment [0..1]
  Optional indication of how unassigned 'Members' shall be interpreted. If not specified, the default value 'undefined' is assumed.

Constraints

- **'DataElement' reference or non-empty 'argument' or non-empty 'item'**
  If a 'dataElement' is not specified, or if the 'dataElement' is resolved to a 'StructuredDataType' or a 'CollectionDataType', either a non-empty 'argument' set or a non-empty 'item' set shall be specified.
  inv: **DataInstanceOrArgumentsOrItemsInDataElementUse**:
  not (self.dataElement.oclIsUndefined() and self.argument->isEmpty() and self.item->isEmpty())
  and (self.dataElement.oclIsKindOf(StructuredDataInstance)
  or not (self.resolveDataType().oclIsKindOf(StructuredDataType)
  and self.argument->isEmpty()))
  and (self.dataElement.oclIsKindOf(CollectionDataInstance)
  or not (self.resolveDataType().oclIsKindOf(CollectionDataType)
  and self.item->isEmpty()))

- **Valid 'DataType' for items**
  The items in the 'item' property shall conform to the 'itemType' of the resolved 'CollectionDataType'.
  inv: **DataTypeOfItemsInDataInstance**
  self.item->forAll(i |
  i.resolveDataType().conformsTo(self.resolveDataType().oclAsType(CollectionDataType).itemType))

- **Only 'item' if the resolved data type is 'CollectionDataType'**
  The 'item' property shall be non-empty if the 'dataElement' property is resolved to a 'CollectionDataType'.
  inv: **ItemOnlyWithCollectionDataType**
  (((self.dataElement.oclIsKindOf(CollectionDataType))
  or self.dataElement.oclIsUndefined())
  and not self.item->isEmpty())
  or self.item->isEmpty()

- **Matching parameters for 'Function's**
  All 'FormalParameter's shall be bound if the 'dataElement' refers to a 'Function'.
  inv: **FunctionCallParameters**:
  not self.dataElement.oclIsTypeOf(Function)
  or self.dataElement.oclAsType(Function).formalParameter->forAll(p | self.argument->exists(a | a.parameter = p))

6.3.17 CastDataUse

Semantics

A 'CastDataUse' denotes a generic and unified expression that changes the resolved 'DataType' of a 'DataUse' to the 'DataType' specified in the 'dataType' property. The specified 'DataType' shall extend the resolved 'DataType' of the referenced 'dataUse'.

Generalization

- DataUse

Properties

- dataUse: DataUse [1]
  The contained 'DataUse' to be cast.

- dataType: DataType [1]
  The 'DataType' to cast to.

Constraints

- **Compatible 'DataType's**
  The specified 'dataType' shall directly or indirectly extend the resolved 'DataType' of the specified 'dataUse'.
  inv: **CompatibleDataTypesInCastDataUse**:
  self.dataType.conformsTo(self.dataUse.resolveDataType())

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

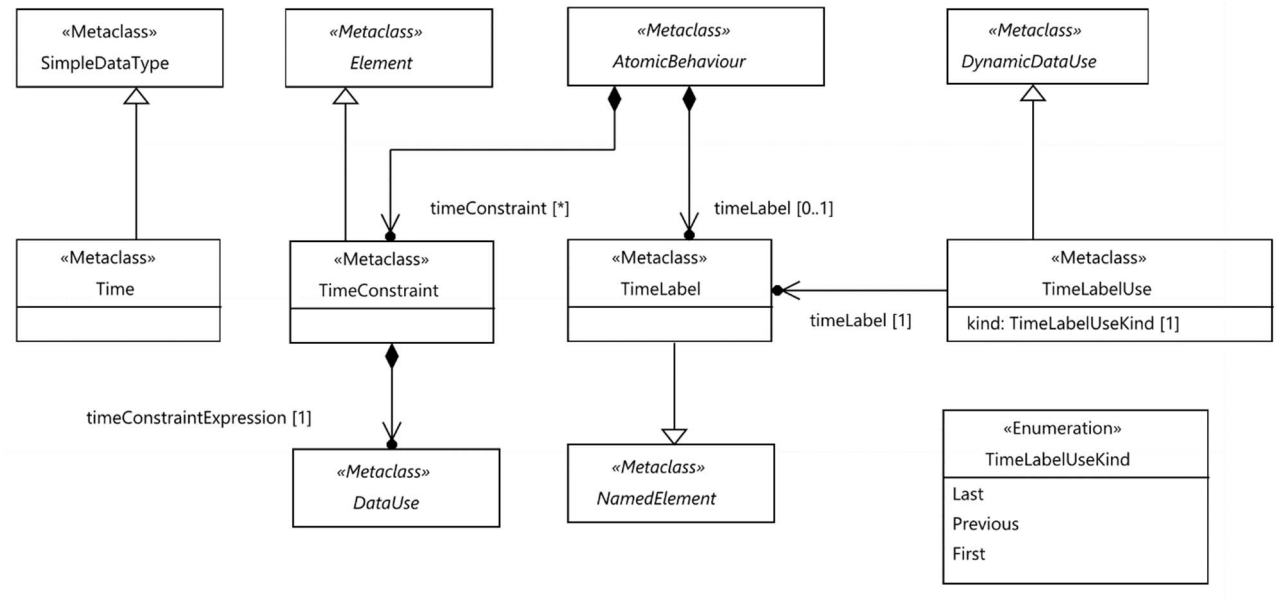

Figure 7.1: Time, time label and time constraint

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 contains the first, last, and previous 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 'AtomicBehaviour' the 'TimeLabel' is attached to is executed only once, all three timestamps are identical. Otherwise, if the atomic behaviour is executed iteratively, e.g. within a loop, the timestamps contained in the 'TimeLabel' are continuously updated. When a 'TimeLabel' is used, the desired timestamp shall be specified together with the 'TimeLabel'.

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. However, it is 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. The 'kind' of 'TimeLabelUse' specifies which of the timestamps of a 'TimeLabel' shall be used in a concrete expression.

The scope of the 'AtomicBehaviour' that contains the 'TimeLabel' used by a 'TimeLabelUse' and the scope for that 'TimeLabelUse' shall contain at least one common tester component.

Generalization

- DynamicDataUse

Properties

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

- kind: TimeLabelUseKind [1]
  Refers to the kind of time label use, specifying which kind of time label shall be used.

Constraints

- **Empty 'argument' and 'reduction' sets**
  The 'argument' and 'reduction' sets shall be empty.
  inv: **TimeLabelArgumentReduction**:
  self.reduction->isEmpty() and self.argument->isEmpty()

- '**TimeLabel's only within the same 'TestDescription' when local ordering is used**
  When local ordering is used, 'TimeLabel's shall only be used within the same test description.
  inv: **TimeLabelLocallyOrdered**:
  self.getParentTestDescription().isLocallyOrdered = true
  or self.timeLabel.getParentTestDescription() = self.getParentTestDescription()

7.2.4 TimeLabelUseKind

Semantics

'TimeLabelUseKind' specifies the kind of a 'TimeLabelUse', whether it shall access the 'first', 'previous', or 'last' timestamp of the execution of an 'AtomicBehaviour'.

Generalization

There is no generalization specified.

Literals

- last
  The corresponding 'TimeLabelUse' shall refer to the timestamp of the last occurrence of the 'AtomicBehaviour' containing the 'TimeLabel' referenced by the 'TimeLabelUse'.

- previous
  The corresponding 'TimeLabelUse' shall refer to the timestamp of the previous occurrence of the 'AtomicBehaviour' containing the 'TimeLabel' referenced by the 'TimeLabelUse'.

- first
  The corresponding 'TimeLabelUse' shall refer to the timestamp of the first occurrence of the 'AtomicBehaviour' containing the 'TimeLabel' referenced by the 'TimeLabelUse'.

Constraints

There are no constraints specified.

7.2.5 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'. If both a 'TimeLabel' and a 'TimeConstraint' are defined within the same 'AtomicBehaviour', then the 'TimeLabel' is always evaluated before the 'TimeConstraint'.

In case of locally ordered 'TestDescription's, the 'TimeLabel's that are used in the expression shall be contained in an 'AtomicBehaviour' that is local to one of the 'ComponentInstance's of the 'AtomicBehaviour' that the 'TimeConstraint' is contained in. That is, there shall be at least one 'ComponentInstance' that participates in both the behaviour containing 'TimeConstraint' and the behaviour containing every 'TimeLabel' that is used in the 'timeConstraintExpression'.

NOTE: The participation of components in behaviours can be determined based on the component participation rules specified in clause 9.2.1, and the corresponding operation 'getParticipatingComponents' specified in clause 4.6.

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.resolveDataType().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 'TimeConstraint'. That is, all 'VariableUse's shall reference the 'ComponentInstance's which participate in the 'AtomicBehaviour'.
  inv: **TimeConstraintVariables**:
  self.timeConstraintExpression.argument
  ->closure(a | a.dataUse.argument)->collect(pb | pb.dataUse)
  ->union(self.timeConstraintExpression.argument)
  ->select(du | du.oclIsKindOf(VariableUse))
  ->forAll(du | self.container().oclAsType(Behaviour).getParticipatingComponents()->includes(
  du.oclAsType(VariableUse).componentInstance))

- **Use of local time labels only**
  In case of locally ordered 'TestDescription', the expression given in the 'DataUse' specification shall contain only 'TimeLabel's that are local to the 'AtomicBehaviour' that contains this 'TimeConstraint'. That is, all 'TimeLabel's shall be contained in 'AtomicBehaviour's involving the 'ComponentInstance's which participate in the 'AtomicBehaviour' that contains this 'TimeConstraint.
  inv: **TimeConstraintTimeLabels**:
  self.timeConstraintExpression.argument
  ->closure(a | a.dataUse.argument)->collect(pb | pb.dataUse)
  ->union(self.timeConstraintExpression.argument)
  ->select(du | du.oclIsKindOf(TimeLabelUse))
  ->forAll(du | self.container().oclAsType(Behaviour).getParticipatingComponents()->includesAll(

  du.oclAsType(TimeLabelUse).timeLabel.container().oclAsType(Behaviour).getParticipatingComponents()))

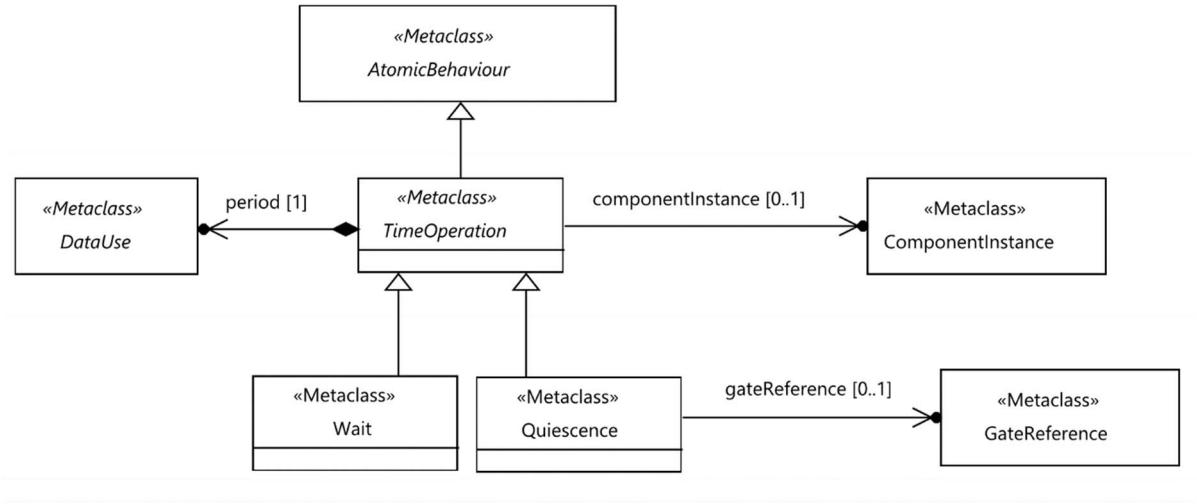

Figure 7.2: Time operations

7.2.6 TimeOperation

Semantics

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

In case of local ordering, the scope of a 'TimeOperation' is the 'componentInstance'. If the 'componentInstance' is not specified then the scope of the 'timeOperation' is the scope of the containing 'Block' and it does not affect the scope of the 'Block'.

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

- **Component required in locally ordered test description**
  If the 'TimeOperation' is contained in a locally ordered 'TestDescription' then the 'componentInstance' shall be specified.
  inv: **TimeOperationComponent**:
  self.getParentTestDescription().isLocallyOrdered
  implies not self.componentInstance.oclIsUndefined()

- **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 = ComponentInstanceRole::Tester)
  or (self.oclIsTypeOf(Quiescence)
  and not self.oclAsType(Quiescence).gateReference.oclIsUndefined()
  and self.oclAsType(Quiescence).gateReference.component.role = ComponentInstanceRole::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.resolveDataType().oclIsKindOf(Time)

7.2.7 Wait

Semantics

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

Any input arriving at the 'Tester' 'componentInstance' 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.

7.2.8 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). If 'gateReference' is specified then the scope of the 'Quiescence' is the component of that 'GateReference'.

When a 'Quiescence' is executed, the 'Tester' component listens to 'Interaction's that occur at the defined gate reference(s). If no unhandled 'Interaction' occurs during the defined 'period' (time duration), the 'Quiescence' completes successfully.

Input arriving during 'Quiescence' that matches an 'Interaction' of an alternative block in 'AlternativeBehaviour' or 'ExceptionalBehaviour' is considered handled and does not affect the outcome of the 'Quiescence'. 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.

If 'Quiescence' occurs as the first behaviour element in an alternative block of an 'InterruptBehaviour', upon the execution of the corresponding alternative block, the 'Quiescence' is reset. That is, the corresponding block may be executed again repeatedly as long as no other alternative block can be executed for the duration of the 'Quiescence'.

In case there are multiple alternative blocks of 'InterruptBehaviour's in which the first behaviour element is a 'Quiescence' all of them are operating independently. That is, if one 'InterruptBehaviour' starting with a 'Quiescence' is executed, only that 'Quiescence' is reset. Other 'InterruptBehaviour's starting with a 'Quiescence' are not affected.

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 not be provided and vice versa.
  inv: **QuiescenceTarget**:
  self.gateReference.oclIsUndefined() xor self.componentInstance.oclIsUndefined()

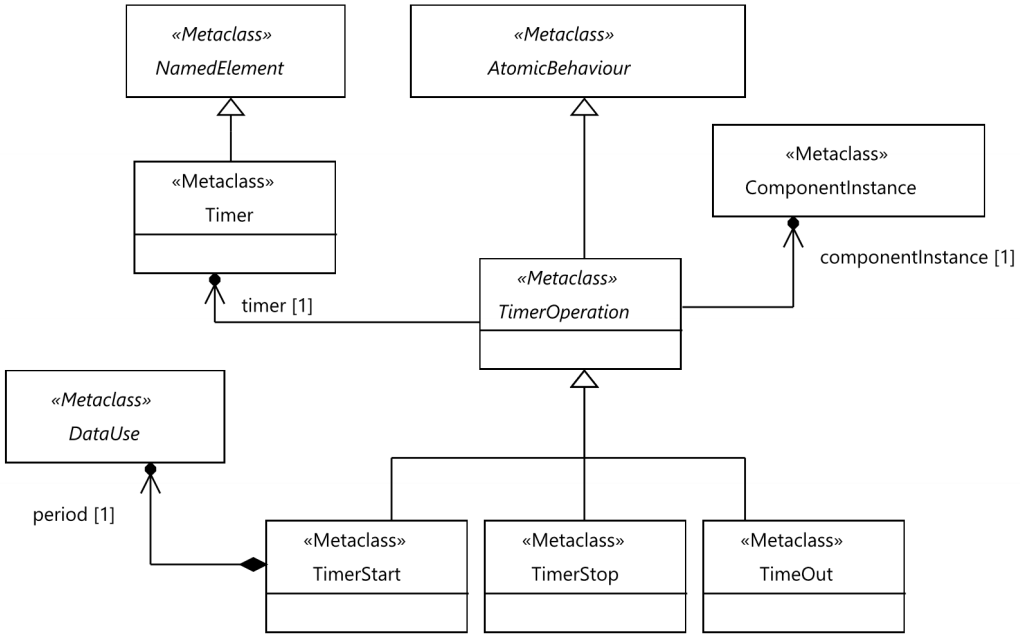

Figure 7.3: Timer and timer operations

7.2.9 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.

7.2.10 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. The scope of a 'TimerOperation' is the 'componentInstance'.

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 = ComponentInstanceRole::Tester

7.2.11 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.resolveDataType().oclIsKindOf(Time)

7.2.12 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.

7.2.13 TimeOut

Semantics

A 'TimeOut' is called a tester-input event and is used to specify the occurrence of a timeout event. The successful execution of a 'TimeOut' implies that the period set by the 'TimerStart' operation elapses. 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

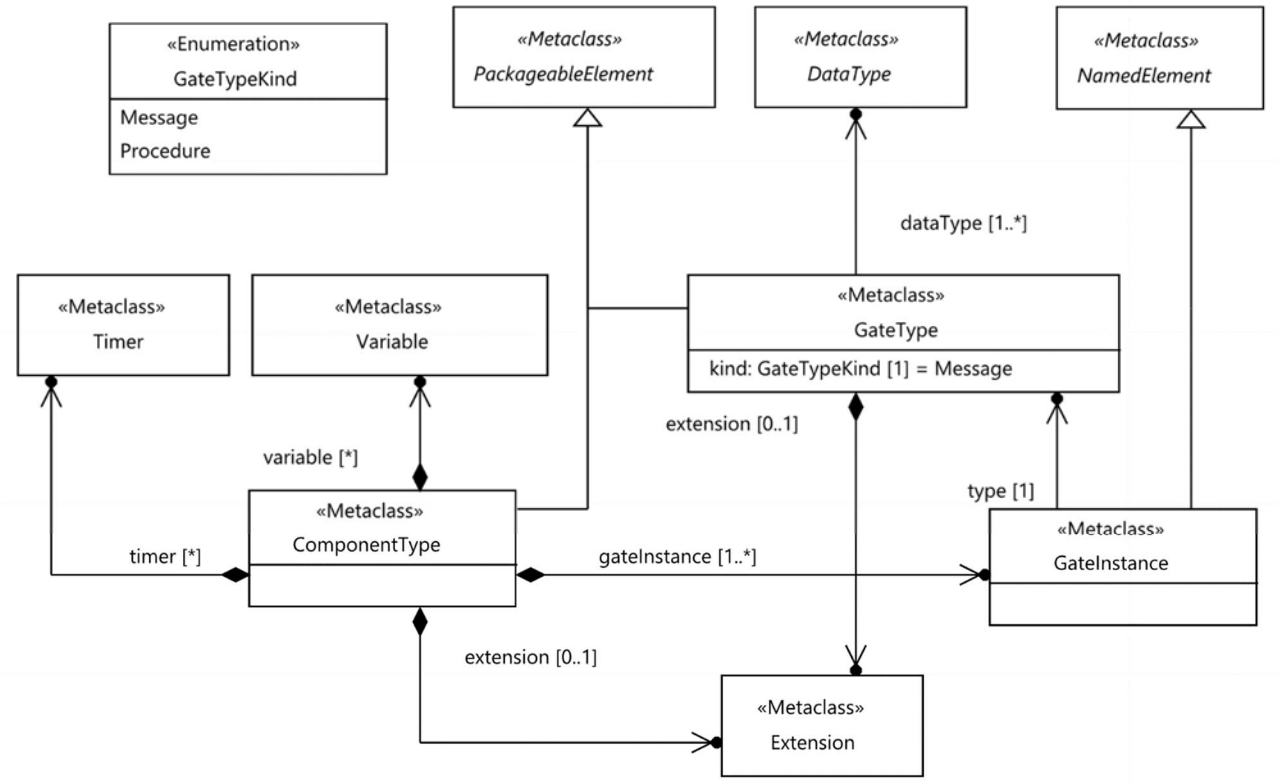

Figure 8.1: Component and gate type

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.

NOTE 1: Only the top-level data types for the messages or procedure calls which are used as arguments need to be specified as part of the gate type definition. That is, the member data types of such top-level data types need not be specified in the gate type definition unless this member data type is also a top-level data type in another message or procedure associated to the gate type definition.

NOTE 2: When a 'DataType' is allowed by a 'GateType' then all 'DataType's that extend it are implicitly allowed. However, execution frameworks may require explicit declaration of all 'DataType's regardless of extension relations.

A 'GateType' may extend another 'GateType' via the 'extension' property. As a result, the 'GateType' shall acquire 'dataType's from the extended 'GateType' transitively.

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.

- kind: GateTypeKind [1] = 'Message'
  Indicates whether the 'GateType' shall be used for 'Message' or 'Procedure' 'Interaction's.

- extension: Extension [0..1]
  Optional 'Extension' with reference to a 'GateType' that this 'GateType' extends.

Constraints

- **Compatible 'DataType's.**
  The 'DateType's specified for the 'GateType' shall correspond the kind of the 'GateType'. For 'GateType' of kind 'Procedure' only 'ProcedureSignature's shall be specified as data types. For 'GateType' of kind 'Message' only 'StructuredDataType's, 'SimpleDataType's and 'CollectionDataType's shall be specified as data types.
  inv: **GateType**:
  (self.kind = GateTypeKind::Procedure and self.allDataTypes()->forAll(t | t.oclIsTypeOf(ProcedureSignature)))
  or (self.kind = GateTypeKind::Message and self.allDataTypes()->forAll(t | t.oclIsTypeOf(StructuredDataType)
  or t.oclIsKindOf(SimpleDataType) or t.oclIsTypeOf(CollectionDataType)))

8.2.2 GateTypeKind

Semantics

'GateTypeKind' specifies the kind of a 'GateType', whether it shall be used for 'Message'-based or 'Procedure'-based interactions.

Generalization

There is no generalization specified.

Literals

- Message
  The 'GateType' shall be used only for 'Message' 'Interaction's involving simple, structured, and collection 'DataTypes'.

- Procedure
  The 'GateType' shall be used only for 'ProcedureCall' 'Interaction's involving 'ProcedureSignature' 'DataTypes'.

Constraints

There are no constraints specified.

8.2.3 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.

8.2.4 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.

A 'ComponentType' may extend another 'ComponentType' via the 'extension' property. The 'ComponentType' shall acquire 'gateInstance's, 'timer's and 'variable's from the extended 'ComponentType' transitively.

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'.
- extension: Extension [0..1]
  Optional 'Extension' with reference to a 'ComponentType' that this 'ComponentType' extends.

Constraints

There are no constraints specified.

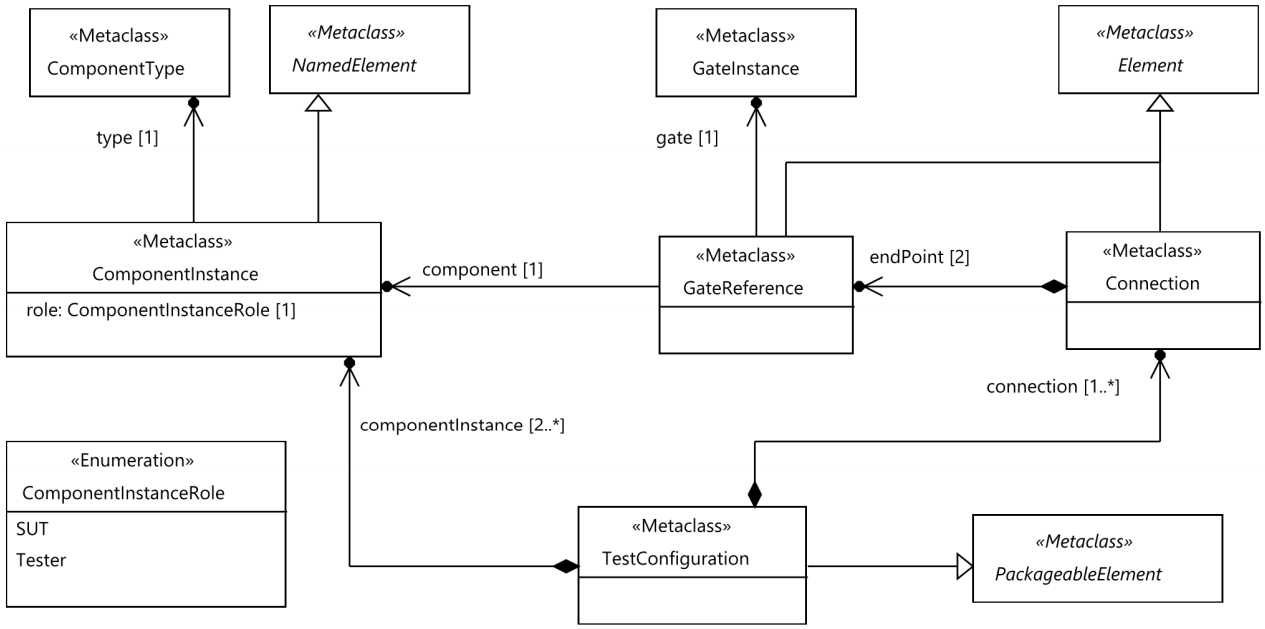

Figure 8.2: Test configuration

8.2.5 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' with role 'Tester' is referred to as a *tester 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 'Tester'. When a 'ComponentInstance' is created, a 'Timer' shall be in the *idle* state (see clause 7.2.9) and a 'Variable' shall have the value <undefined> (see clause 6.2.20).

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.

8.2.6 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.

8.2.7 GateReference

Semantics

A 'GateReference' is an endpoint of a 'Connection', in which it is contained. It allows the specification of a connection between two 'GateInstance's of different 'ComponentInstance's in a unique manner as '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.allGates()->includes(self.gate)

8.2.8 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 8.2.9).

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.gate.type->asSet()->size() = 1

8.2.9 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 = ComponentInstanceRole::Tester)
  and self.componentInstance->exists(c | c.role = ComponentInstanceRole::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->includesAll(c.endPoint.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.component.role->includesAll(Set{ComponentInstanceRole::SUT, ComponentInstanceRole::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 |
  not c1.endPoint->reject(ep1 | not c2.endPoint->exists(ep2 |
  ep1.component = ep2.component and ep1.gate = ep2.gate
  ))->isEmpty()))

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

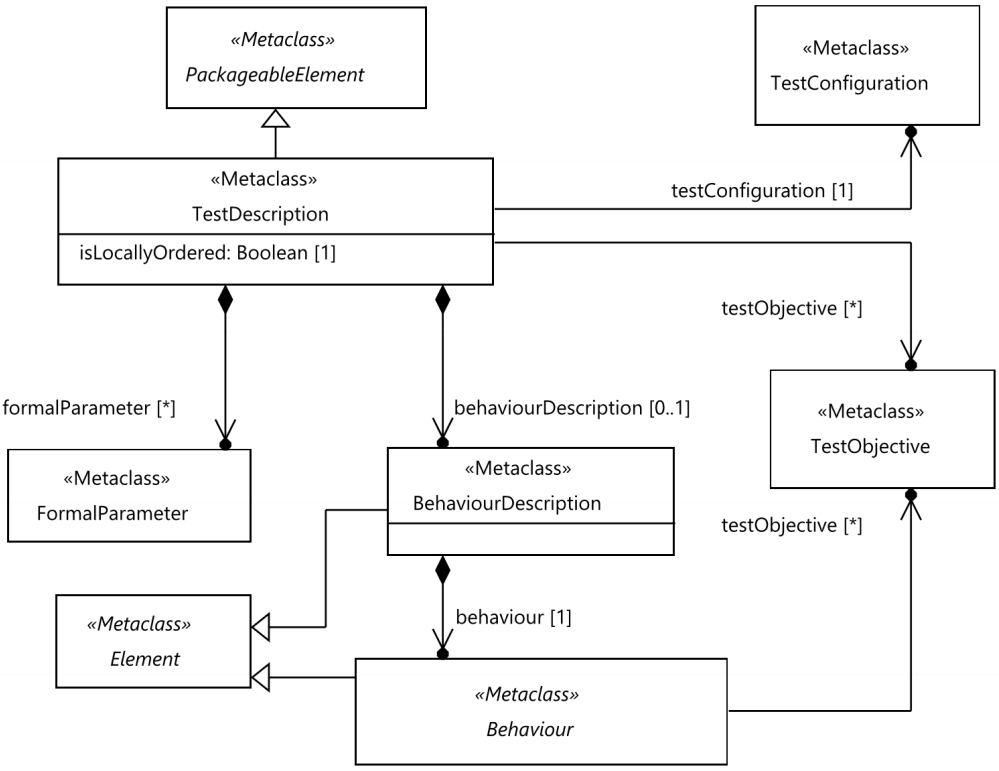

Figure 9.1: Test description

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 'GateReference's to be used in the behaviour.

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

If a 'TestDescription' with formal parameters is invoked within another 'TestDescription', actual parameters are provided via a 'TestDescriptionReference' (see clause 9.4.11). The mechanism of passing arguments to a 'TestDescription' that is invoked by a test management tool is not defined.

The 'isLocallyOrdered' property, set to 'false' by default, enables the specification of 'TestDescriptions' that override the assumption of total ordering of all contained 'Behaviour's and 'Block's. If set to 'true', the default semantics of total ordering of all 'Behaviour's within the 'TestDescription' is changed to local ordering within each 'ComponentInstance' for the 'TestDescription'.

In case of local ordering, 'Behaviour's are executed in the scope of specific 'ComponentInstance's. The order of contained 'Behaviour's is only determined for the individual 'ComponentInstance's. There is no assumption of implicit synchronization between the 'Behaviour's that execute in the different 'ComponentInstance's and 'Behaviour's may execute simultaneously.

In case of total ordering, the scope of a 'Behaviour' consists of all 'ComponentInstance's of the 'TestConfiguration'. In case of local ordering, the scope is defined by the 'ComponentInstance's directly participating in an 'AtomicBehaviour' or the union of scopes of each 'Block' that the 'Behaviour' contains. For 'Block's the scope is the union of the scopes of the contained 'Behaviour's. The individual definitions of scope are specified in the following clauses.

Local ordering implies that properties of components, such as timers and variables, are only available for a single 'ComponentInstance'. For a conditional 'Behaviour' with multiple components in its scope, the condition shall be specified for every tester 'ComponentInstance' separately. This enables the use of component variables in those conditions. 'AtomicBehaviour's that are not inherently tied to components may be associated with 'ComponentInstance's to limit their scope and thus provide access to component's properties. If 'ComponentInstance' is not specified then the 'AtomicBehaviour' shall execute independently on all 'ComponentInstance's that are in the scope of the containing 'Block'.

NOTE: A 'TestDescription' may be annotated with the predefined annotation 'Master' to indicate that the 'TestDescription' is an entry point for test execution (see clause 10.6.1). Any number of 'TestDescription's may be annotated with the 'Master' annotation. Also, no restrictions or provisions regarding the referencing of a 'TestDescription' annotated with the 'Master' annotation is defined. Restrictions may be added external to TDL, within tools, or depending on the specific context.

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'.
- isLocallyOrdered: Boolean [1] = false
  If set to 'true', the default semantics of total ordering of all behaviours within the test description is changed to local ordering for 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

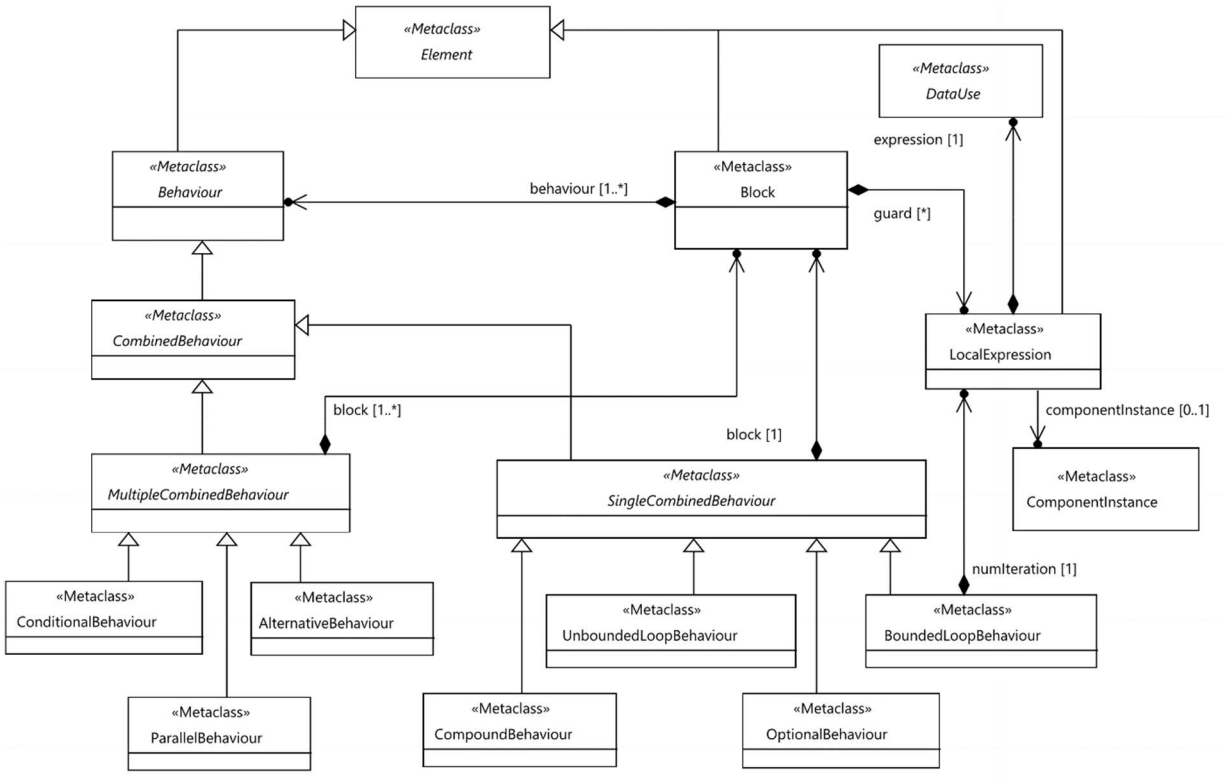

Figure 9.2: Combined behaviour concepts

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'.

In case of a locally ordered 'TestDescription' either a guard shall be specified for every tester 'ComponentInstance' in the scope of the 'Block' or the 'Block' shall not have a guard at all (except when specified otherwise).

In case of local ordering, the scope of a 'Block' is the union of the scopes of 'Behaviour's contained in 'behaviour'.

NOTE: In case of a locally ordered test description, the behaviour of each participating component is determined solely based on its local guard condition. Determining the compatibility of guards associated with different components is outside of the scope of the present document.

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: LocalExpression [0..*]
  A potentially scoped 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 ->forAll(g | g.expression.resolveDataType().conformsTo('Boolean'))

- **No directly contained 'ExceptionalBehaviour's and 'PeriodicBehaviour's**
  A 'Block' shall not contain 'ExceptionalBehaviour's and 'PeriodicBehaviour's.
  inv: **AllowedBehaviourTypes**:
  self.behaviour->forAll(b |
  not b.oclIsTypeOf(ExceptionalBehaviour) and not b.oclIsTypeOf(PeriodicBehaviour))

- **Guard for each participating tester in locally ordered test descriptions**
  If the 'Block' is contained in a locally ordered 'TestDescription' then a guard shall be specified for every participating 'ComponentInstance' in the associated 'TestConfiguration' that has the role 'Tester' or there shall be no guards at all.
  inv: **GuardsForParticipatingComponents**:
  self.guard->size() = 0
  or self.getParticipatingComponents()->reject(c | c.role = ComponentInstanceRole::SUT)
  ->forAll(c | self.guard->exists(ex | ex.componentInstance = c))
  or not self.getParentTestDescription().isLocallyOrdered

- **Single guard in totally ordered test description**
  If the 'Block' is contained in a totally ordered 'TestDescription' then there shall not be more than one guard.
  inv: **SingleTotalGuard**:
  self.getParentTestDescription().isLocallyOrdered or self.guard->size() <= 1

9.3.3 LocalExpression

Semantics

In locally ordered 'TestDescription's, some data items, such as 'Variable's, shall only be used within the scope of the 'ComponentInstance' that owns those items. A 'LocalExpression' allows to limit the scope of an expression by associating it with a 'ComponentInstance' and by this enable the use of the items that are local to this component in the expression.

Generalization

- Element

Properties

- expression: DataUse [1]
  An expression that specifies the value.

- componentInstance: ComponentInstance [0..1]
  The 'ComponentInstance' that provides the scope for the expression.

Constraints

- **Local expressions in locally ordered test descriptions have 'ComponentInstance' specified**
  If the 'LocalExpression' is contained in a locally ordered 'TestDescription' then the componentInstance shall be specified.
  inv: **LocalExpressionComponent**:
  ```
  self.getParentTestDescription().isLocallyOrdered
  implies not self.componentInstance.oclIsUndefined()
  ```

- **Only local variables and time labels in case of locally ordered test description**
  If the componentInstance is specified then all 'Variable's and 'TimeLabel's used in the expression shall be local to that 'ComponentInstance'.
  inv: **LocalVariablesAndTimersInExpression**:
  ```
  self.expression.argument
    ->closure(a | a.dataUse.argument)->collect(pb | pb.dataUse)
    ->union(self.expression.argument)
    ->including(self.expression)
    ->select(du | du.oclIsKindOf(VariableUse))
    ->forAll(du | du.oclAsType(VariableUse).componentInstance = self.componentInstance)
  and self.expression.argument
    ->closure(a | a.dataUse.argument)->collect(pb | pb.dataUse)
    ->union(self.expression.argument)
    ->including(self.expression)
    ->select(du | du.oclIsKindOf(TimeLabelUse))
    ->forAll(du | du.oclAsType(TimeLabelUse).timeLabel.container().oclAsType(Behaviour)
      .getParticipatingComponents()->includes(self.componentInstance))
  ```

9.3.4 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.

9.3.5 SingleCombinedBehaviour

Semantics

A 'SingleCombinedBehaviour' contains a single 'Block' of 'Behaviour'. In case of local ordering, the scope of a 'SingleCombinedBehaviour' is the union of the scopes of the 'Block's contained in 'block', 'periodic', and 'exceptional'.

Generalization

- CombinedBehaviour

Properties

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

Constraints

There are no constraints specified.

9.3.6 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.

9.3.7 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.

In case of a locally ordered 'TestDescription', a numIteration shall be specified for every 'ComponentInstance' in the scope of the 'BoundedLoopBehaviour'.

NOTE: In case of locally ordered test description, the behaviour of each participating component is determined solely based on its local iteration condition. Determining the compatibility of conditions associated with different components is outside of the scope of the present document.

Generalization

- SingleCombinedBehaviour

Properties

- numIteration: LocalExpression [1..*]
  A potentially scoped 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->isEmpty()

- **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**:
  self.numIteration->forAll(e | e.expression.resolveDataType().conformsTo('Integer'))

- **Iteration count in locally ordered test descriptions**
  If the 'BoundedLoopBehaviour' is contained in a locally ordered 'TestDescription' then a numIteration shall be specified for every participating 'ComponentInstance' that has the role 'Tester'.
  inv: **IterationCountsForParticipatingComponents**:
  self.block.getParticipatingComponents()->reject(c | c.role = ComponentInstanceRole::SUT)
  ->forAll(c | self.numIteration->exists(ex | ex.componentInstance = c))
  or not self.getParentTestDescription().isLocallyOrdered

- **Single numIteration in totally ordered test description**
  If the 'BoundedLoopBehaviour' is contained in a totally ordered 'TestDescription' then there shall be exactly one numIteration.
  inv: **SingleTotalIterationCount**:
  self.getParentTestDescription().isLocallyOrdered or self.numIteration->size() = 1

9.3.8 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.

9.3.9 OptionalBehaviour

Semantics

An 'OptionalBehaviour' specifies a 'Block' of inter-tester communication where the decision to execute the behaviour is decided by one tester and the other tester is able to continue regardless of whether the behaviour is executed or not. An 'OptionalBehaviour' shall start with a tester-to-tester 'Interaction'. For the source 'ComponentInstance' of that 'Interaction' the 'OptionalBehaviour' is semantically equivalent to a 'CompoundBehaviour'. For the target 'ComponentInstance's of that 'Interaction' the 'OptionalBehaviour' shall be treated in the same way as an 'InterruptBehaviour'.

In case of locally ordered 'TestDescription', the scope of the contained 'Block' shall be limited to the source and target(s) of the starting 'Interaction' of the block. Other 'OptionalBehaviour's may be added to the block of an 'OptionalBehaviour' if the source of the starting 'Interaction' in the block of the contained 'OptionalBehaviour's is a tester that participates in the containing 'OptionalBehaviour'.

An 'OptionalBehaviour' shall be disabled:

- at the source tester:
  - if the block of the 'OptionalBehaviour' has a guard and that guard evaluates to 'false';
  - after the behaviour in the block of the 'OptionalBehaviour' has completed execution;
- at the target tester(s):
  - after the behaviour in the block of the 'OptionalBehaviour' has completed execution;
  - after the execution of the first tester-input event following the 'OptionalBehaviour', whose source is the same 'ComponentInstance' as the source of the 'OptionalBehaviour';
  - the containing 'TestDescription' terminates.

Generalization

- SingleCombinedBehaviour

Properties

There are no properties specified.

Constraints

- **First 'AtomicBehaviour' in block allowed**
  The block of an 'OptionalBehaviour' shall start with a tester-to-tester 'Interaction'.
  inv: **OptionalBehaviourStart**:
  self.block.behaviour->first().oclIsKindOf(Interaction)
  and self.block.behaviour->first().oclAsType(Interaction)
  ->collect(i | i.sourceGate.component->union(i.target.targetGate.component))
  ->forAll(c | c.role = ComponentInstanceRole::Tester)

- **No other testers except the participants of starting 'Interaction' within 'OptionalBehaviour' in locally ordered 'TestDescription'**
  If an 'OptionalBehaviour' is included in a locally ordered 'TestDescription' then no other tester 'ComponentInstance' shall participate in the block of the 'OptionalBehaviour' than the source and target of the starting 'Interaction' except when being a target of the starting 'Interaction' in a nested 'OptionalBehaviour'.
  inv: **OptionalBehaviourParticipation**:
  let initial = self.block.behaviour->first().oclAsType(Interaction),
  initialComponents = initial->collect(i | i.sourceGate.component->union(i.target.targetGate.component)),
  optionals = self.block->closure(
  b | b.behaviour
  ->select(oclIsKindOf(SingleCombinedBehaviour)).oclAsType(SingleCombinedBehaviour).block
  ->union(b.behaviour
  ->select(oclIsKindOf(MultipleCombinedBehaviour)).oclAsType(MultipleCombinedBehaviour).block)
).behaviour->select(oclIsKindOf(OptionalBehaviour)).oclAsType(OptionalBehaviour),
  optionalTargets = optionals.block->select(b | b.behaviour->select(i | i.oclIsKindOf(Interaction))->first().oclAsType(Interaction).target.targetGate.component)
  in
  self.block.getParticipatingComponents()
  ->forAll(c | initialComponents->includes(c) or optionalTargets->includes(c))
  or not self.getParentTestDescription().isLocallyOrdered

9.3.10 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').

In case of local ordering, the scope of a 'MultipleCombinedBehaviour' is the union of the scopes of the 'Block's contained in 'block', 'periodic', and 'exceptional'.

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.

9.3.11 AlternativeBehaviour

Semantics

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

If the 'AlternativeBehaviour' is contained in a locally ordered 'TestDescription' then:

- all of the starting tester-input events shall occur on the same 'ComponentInstance' (target-of-alt);
- the scope of the 'Block' shall not contain tester components other than the target-of-alt.

A 'Block' in an 'AlternativeBehaviour' can only contain 'OptionalBehaviour'(s) in which the source of the first 'Interaction' is the target-of-alt.

NOTE: The contained 'OptionalBehaviour'(s) may contain additional 'OptionalBehaviour'(s) as is specified in clause 9.3.9.

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 a successful 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())

- **Same component if locally ordered**
  If the containing 'TestDescription' is locally ordered then all 'Block's shall start with a tester-input event of the same 'ComponentInstance'.
  inv: **AlternativeBlocksComponent**:
  let initial = self.block.behaviour->first() in
  Set{}
  ->including(initial->select(oclIsKindOf(Interaction)).oclAsType(Interaction).target.targetGate.component)
  ->including(initial->select(oclIsKindOf(Quiescence)).oclAsType(Quiescence).componentInstance)
  -> including(initial->select(oclIsKindOf(TimeOut)).oclAsType(TimeOut).componentInstance)
  ->size() = 1 or not self.getParentTestDescription().isLocallyOrdered

- **Tester participating in locally ordered case**
  If the 'AlternativeBehaviour' is contained in a locally ordered 'TestDescription' then no other tester 'ComponentInstance' shall participate in any block than the target of the first tester-input event and 'ComponentInstance's participating in blocks of contained 'OptionalBehaviour's.
  inv: **AlternativeBehaviourParticipation**:
  ```
  let initial = self.block.behaviour->first(),
  targetComponent = Set{}
  ->including(initial->select(oclIsKindOf(Interaction)).oclAsType(Interaction).target.targetGate.component)
  ->including(initial->select(oclIsKindOf(Quiescence)).oclAsType(Quiescence).componentInstance)
  -> including(initial->select(oclIsKindOf(TimeOut)).oclAsType(TimeOut).componentInstance),
  nonOptionalBlocks = self.block->closure(
    b | b.behaviour->reject(oclIsKindOf(OptionalBehaviour))
      ->select(oclIsKindOf(SingleCombinedBehaviour)).oclAsType(SingleCombinedBehaviour).block
    ->union(b.behaviour->reject(oclIsKindOf(OptionalBehaviour))
      ->select(oclIsKindOf(MultipleCombinedBehaviour)).oclAsType(MultipleCombinedBehaviour).block)
  )
  in
  targetComponent->includesAll(
    nonOptionalBlocks.getParticipatingComponents()->reject(c | c.role = ComponentInstanceRole::SUT))
  or not self.getParentTestDescription().isLocallyOrdered
  ```

- **OptionalBehaviour in locally ordered case**
  A block of an 'AlternativeBehaviour' if the containing 'TestDescription' is locally ordered, shall only contain 'OptionalBehaviour'(s) whose source 'ComponentInstance' is the same as the target of the first tester-input event of that 'Block'.
  inv: **OptionalAlternativeBehaviour**:
  ```
  let initial = self.block.behaviour->first(),
  targetComponent = Set{}
  ->including(initial->select(oclIsKindOf(Interaction)).oclAsType(Interaction).target.targetGate.component)
  ->including(initial->select(oclIsKindOf(Quiescence)).oclAsType(Quiescence).componentInstance)
  -> including(initial->select(oclIsKindOf(TimeOut)).oclAsType(TimeOut).componentInstance)
  in
  self.block.behaviour->select(oclIsKindOf(OptionalBehaviour)) .oclAsType(OptionalBehaviour).block
    ->first().oclAsType(Interaction).sourceGate.component->forAll(c | targetComponent->includes(c))
  or not self.getParentTestDescription().isLocallyOrdered
  ```

9.3.12 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 1: 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-else* statement in a programming language. Blocks specified after this *else* block would never be executed.

NOTE 2: In case of locally ordered test description, the behaviour of each participating component is determined solely based on its local condition. Determining the compatibility of conditions associated with different components is outside of the scope of the present document.

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 self.block->first().guard->size() > 1

- **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 b.guard ->size() > 0)

9.3.13 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 all the '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

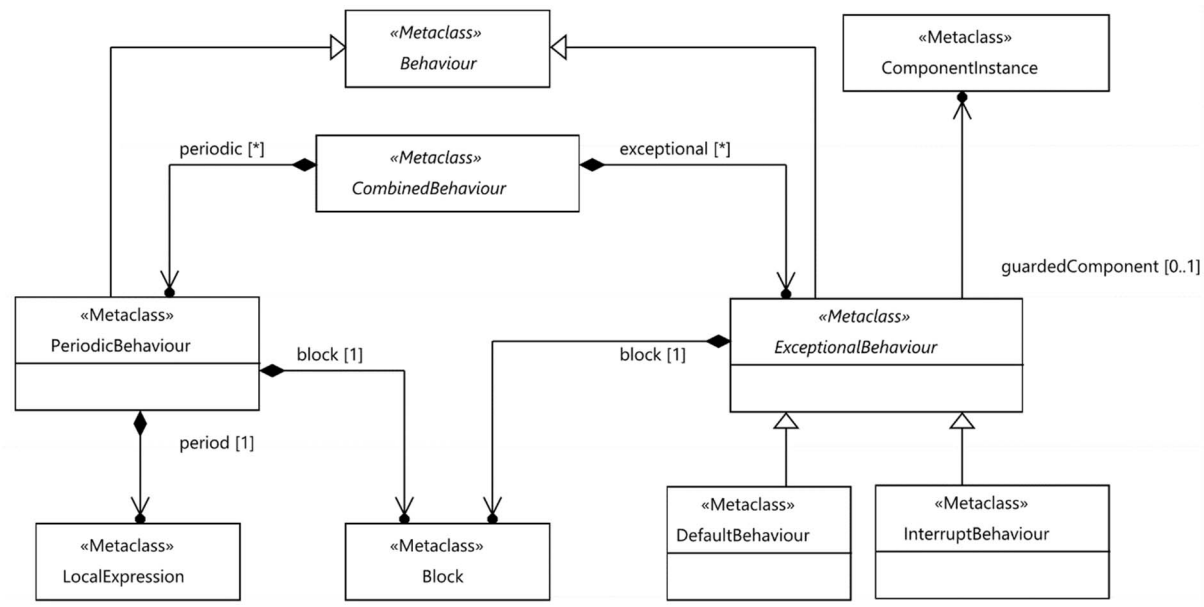

Figure 9.3: Exceptional and periodic behaviour

9.3.14 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 term in clause 3.1).

In case of local ordering, the scope of an 'ExceptionalBehaviour' is the scope of the 'block'.

An 'ExceptionalBehaviour' defines 'Behaviour' that is an alternative to every tester-input event directly or indirectly contained in the 'Block's of the containing 'CombinedBehaviour'.

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 behaviours whose scope includes the 'guardedComponent' force the 'ExceptionalBehaviour' to be activated.

If the 'ExceptionalBehaviour' is contained in a locally ordered 'TestDescription' then the following rules shall apply:

- The 'ExceptionalBehaviour' shall be an alternative only to those tester-input events whose target is the same as the starting tester-input event of the exceptional block.
- The scope of the 'block' shall not contain tester 'ComponentInstance's other than the target of the first tester-input event.

NOTE 1: Other 'OptionalBehaviour'(s) may be added as specified in clause 9.3.9.

- A 'Block' in an 'ExceptionalBehaviour' can only contain 'OptionalBehaviour'(s) whose source 'ComponentInstance' is the same as the target of the first tester-input event of that 'Block'.

NOTE 2: The contained 'OptionalBehaviour'(s) may contain additional 'OptionalBehaviour'(s) as is specified in clause 9.3.9.

In case of more than one 'ExceptionalBehaviour' is attached to the same 'CombinedBehaviour', the implied 'AlternativeBehaviour' would contain the 'Blocks' of all the attached 'ExceptionalBehaviour's in the same order. In case the 'CombinedBehaviour' is contained within another 'CombinedBehaviour' with 'ExceptionalBehaviour's attached to it, the 'ExceptionalBehaviour's of the containing 'CombinedBehaviour' apply to the contained 'CombinedBehaviour' as well, where the 'ExceptionalBehaviour's of the contained 'CombinedBehaviour' have precedence over the 'ExceptionalBehaviour's of the containing 'CombinedBehaviour'.

In case an 'ExceptionalBehaviour' is attached to a 'CombinedBehaviour' which contains a 'TestDescriptionReference', the 'ExceptionalBehaviour' also applies to the behaviour of the referenced 'TestDescription'. The semantics is identical to that of nested 'CombinedBehaviour's, that is 'ExceptionalBehaviour's defined within the referenced 'TestDescription' have precedence over 'ExceptionalBehaviour's defined for the 'CombinedBehaviour' containing the 'TestDescriptionReference'.

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**
  The 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 = ComponentInstanceRole::Tester

- **Same component if locally ordered and guarded component present**
  If the containing 'TestDescription' is locally ordered and guardedComponent is specified then the 'Block's shall start with tester-input event of the same 'ComponentInstance' as specified in guardedComponent.
  inv: **ExceptionalGuardedandTargetComponent**:
  let initial = self.block.behaviour->first(),
  targetComponent = Set{}
  ->including(initial->select(oclIsKindOf(Interaction)).oclAsType(Interaction).target.targetGate.component)
  ->including(initial->select(oclIsKindOf(Quiescence)).oclAsType(Quiescence).componentInstance)
  -> including(initial->select(oclIsKindOf(TimeOut)).oclAsType(TimeOut).componentInstance)
  in
  guardedComponent->includesAll(targetComponent)
  or not self.getParentTestDescription().isLocallyOrdered

- **Tester participating in locally ordered case**
  If the 'ExceptionalBehaviour' is contained in a locally ordered 'TestDescription' then no other tester 'ComponentInstance' shall participate in any block than the target of the first tester-input event and 'ComponentInstance's participating in blocks of contained 'OptionalBehaviour's .
  inv: **ExceptionalBehaviourParticipation**:
  ```
  let initial = self.block.behaviour->first(),
  targetComponent = Set{}
  ->including(initial->select(oclIsKindOf(Interaction)).oclAsType(Interaction).target.targetGate.component)
  ->including(initial->select(oclIsKindOf(Quiescence)).oclAsType(Quiescence).componentInstance)
  -> including(initial->select(oclIsKindOf(TimeOut)).oclAsType(TimeOut).componentInstance),
  nonOptionalBlocks = self.block->closure(
    b | b.behaviour->reject(oclIsKindOf(OptionalBehaviour))
      ->select(oclIsKindOf(SingleCombinedBehaviour)).oclAsType(SingleCombinedBehaviour).block
  ->union(b.behaviour->reject(oclIsKindOf(OptionalBehaviour))
      ->select(oclIsKindOf(MultipleCombinedBehaviour)).oclAsType(MultipleCombinedBehaviour).block)
  )
  in
  targetComponent->includesAll(
    nonOptionalBlocks.getParticipatingComponents()->reject(c | c.role = ComponentInstanceRole::SUT))
  or not self.getParentTestDescription().isLocallyOrdered
  ```

- **OptionalBehaviour in locally ordered case**
  A block of an 'ExceptionalBehaviour' if the containing 'TestDescription' is locally ordered, shall only contain 'OptionalBehaviour'(s) whose source 'ComponentInstance' is the same as the target of the first tester-input event of that 'Block'.
  inv: **OptionalExceptionalBehaviour**:
  ```
  let initial = self.block.behaviour->first(),
  targetComponent = Set{}
  ->including(initial->select(oclIsKindOf(Interaction)).oclAsType(Interaction).target.targetGate.component)
  ->including(initial->select(oclIsKindOf(Quiescence)).oclAsType(Quiescence).componentInstance)
  -> including(initial->select(oclIsKindOf(TimeOut)).oclAsType(TimeOut).componentInstance)
  in
  self.block.behaviour->select(oclIsKindOf(OptionalBehaviour)).oclAsType(OptionalBehaviour).block
    ->first().oclAsType(Interaction).sourceGate.component->forAll(c | targetComponent->includes(c))
  or not self.getParentTestDescription().isLocallyOrdered
  ```

9.3.15 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.

9.3.16 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.

9.3.17 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.

In case of locally ordered 'TestDescription', if a period is specified, it shall be specified for every tester component in the scope of the 'PeriodicBehaviour'. In case of local ordering, the scope of a 'PeriodicBehaviour' is the scope of the 'block'.

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: LocalExpression [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->forAll(e | e.expression.resolveDataType().oclIsKindOf(Time))

- **Period for each tester in locally ordered test descriptions**
  If the 'PeriodicBehaviour' is contained in a locally ordered 'TestDescription' then a period shall be specified for every 'ComponentInstance' that has the role 'Tester' and for which there is a behaviour in the contained 'Block'.
  inv: **PeriodForParticipatingComponents**:
  self.block.getParticipatingComponents()->reject(c | c.role = ComponentInstanceRole::SUT)
  ->forAll(c | self.period->exists(ex | ex.componentInstance = c))
  or not self.getParentTestDescription().isLocallyOrdered

9.4 Atomic Behaviour - Abstract Syntax and Classifier Description

9.4.1 AtomicBehaviour

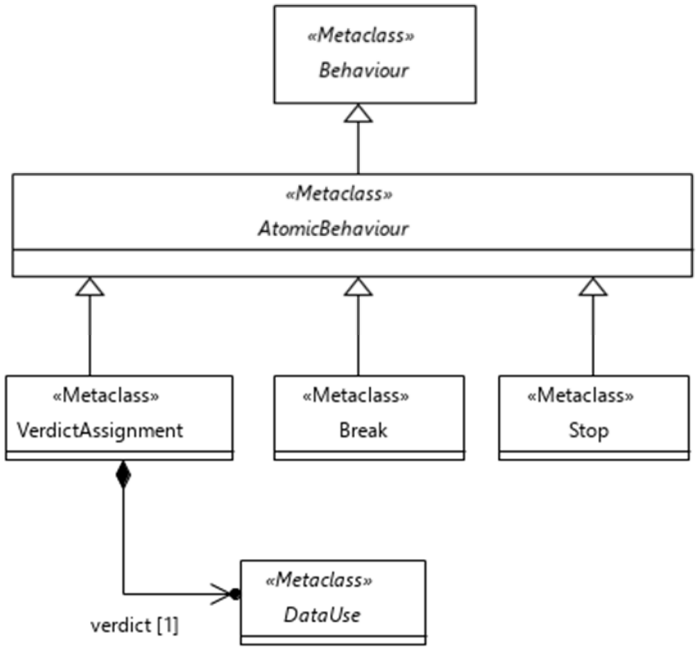

Figure 9.4: Global atomic behaviour concepts

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 7.2.5).

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' is used to conditionally terminate the execution of a 'Block'. A 'Break' shall be contained directly in a block of a 'ConditionalBehaviour' and it shall terminate the 'Block', in which the 'ConditionalBehaviour' is contained. There shall be no other behaviours following a 'Break' in the same 'Block'. Execution shall continue with the 'Behaviour' that follows the terminated 'CombinedBehaviour'.

In case of 'ParallelBehaviour', a 'Break' shall terminate only the execution of its own 'Block', but shall not affect the execution of the other parallel 'Block'(s).

The 'Break' shall apply to all components in the scope of the 'Block' that is to be terminated. A 'Break' does not affect the scope of the containing 'Block'.

Generalization

- AtomicBehaviour

Properties

There are no properties specified.

Constraints

- **Break in conditional behaviour only**
  A 'Break' shall be contained directly in the block of a 'ConditionalBehaviour'.
  inv: **ConditionalBreak**:
  self.container().container().oclIsKindOf(ConditionalBehaviour)

- **No behaviours after break**
  A 'Break' shall be the last behaviour in the containing 'Block'.
  inv: **BreakIsLast**:
  self.container().oclAsType(Block).behaviour->last() = self

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).

A 'Stop' does not affect the scope of the containing 'Block'.

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.

The scope of a 'VerdictAssignment' is the scope of the containing 'Block'. A 'VerdictAssignment' does not affect the scope of the containing 'Block'.

Generalization

- AtomicBehaviour

Properties

- verdict: DataUse [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.resolveDataType().name = 'Verdict'

- **No 'SpecialValueUse'**
  The 'verdict' shall not evaluate to an instance of a 'SpecialValueUse'.
  inv: **VerdictNoSpecialValueUse**:
  not self.verdict.oclIsKindOf(SpecialValueUse)

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'. An 'Assertion' may be optionally associated with a 'ComponentInstance' by means of the 'componentInstance' property inherited from 'ActionBehaviour'. This determines the context in which the 'condition' shall be evaluated. Any changes in the test verdict resulting from the evaluation of the 'Assertion' shall apply to the whole 'TestDescription'.

Generalization

- ActionBehaviour

Properties

- condition: DataUse [1]
  Refers to the test condition that is evaluated.

- otherwise: DataUse [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.resolveDataType().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.resolveDataType().name = 'Verdict'

- **No 'SpecialValueUse'**
  The 'otherwise' shall not evaluate to an instance of a 'SpecialValueUse'.
  inv: **AssertionNoSpecialValueUse**:
  self.otherwise.oclIsUndefined() or not self.otherwise.oclIsKindOf(SpecialValueUse)

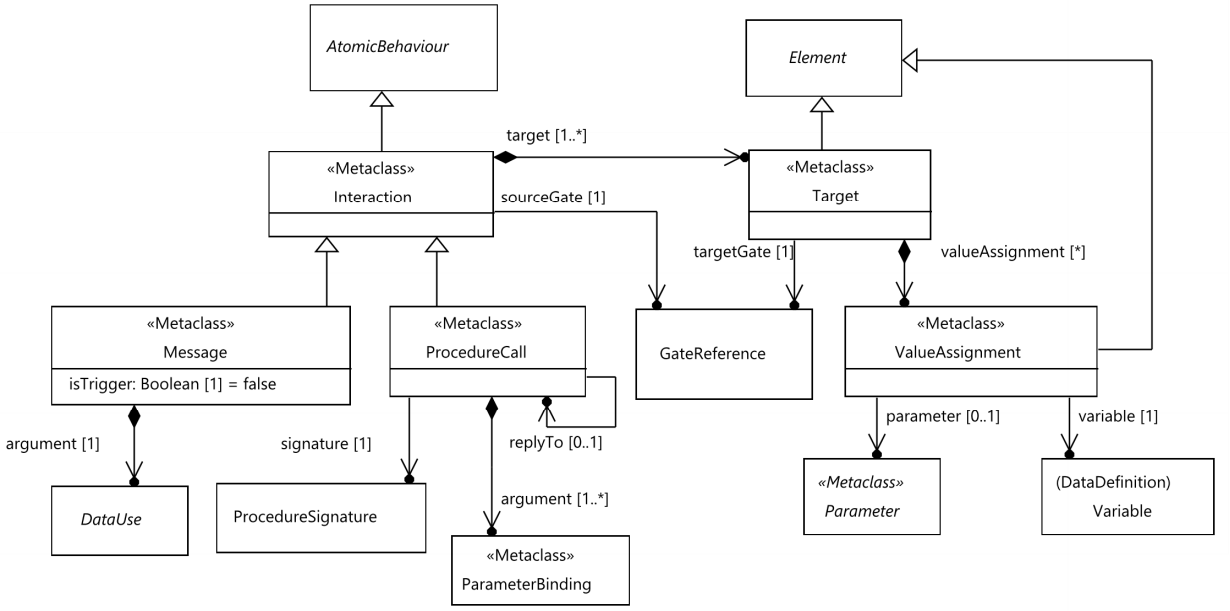

Figure 9.5: Interaction behaviour

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. In case of local ordering, the scope of an 'Interaction' consists of the 'componentInstance's referred to by 'sourceGate' and 'target'.

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. If the source of an 'Interaction' is also a tester then it is not tester-input event.

'Interaction' arguments specify the expected data values being exchanged. Successful execution of an 'Interaction' implies that these expected data values occur at runtime among the participating components and the implicit test verdict 'pass' shall be set. If the expected values do 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 values occurs, the execution is not successful.

NOTE 1: The time period to wait for the specified interaction to occur is defined outside the scope of the present document.

The 'DataUse' specifications, which the arguments refer to, may contain 'Variable's of 'ComponentInstance's participating in this 'Interaction'. Use of a 'Variable' in an argument specification implies the use of its value.

Placeholders, such as 'AnyValue' or 'AnyValueOrOmit', may be used if the concrete value is not known or is irrelevant (see clauses 6.3.7 and 6.3.8).

NOTE 2: How the <undefined> value within the 'DataUse' specification of 'argument' is resolved is outside of the scope of the present document.

The mechanism for specifying arguments is defined in subclasses of 'Interaction'.

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

NOTE 3: If the 'Variable' refers to a 'StructuredDataType', the non-optional 'Member's of this data type can only be assigned values that are different from 'OmitValue'; see clause 6.3.2.

Generalization

- AtomicBehaviour

Properties

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

- target: Target [1..*] {unique}
  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.getParentTestDescription().testConfiguration.connection->exists(c |
        not c.endPoint->reject(ep |
            (ep.component = self.sourceGate.component and ep.gate = self.sourceGate.component) or
            (ep.component = t.targetGate.component and ep.gate = t.targetGate.gate)
        )->isEmpty()))
```

9.4.7 Message

Semantics

A 'Message' represents a one-way interaction. 'Message' is directed, i.e. the information being exchanged is sent by a component via the 'sourceGate' and received by one or many components via the 'target's (point-to-point and point-to-multipoint communication, see clause 8.2.8).

In case of local ordering, the scope for 'DataUse's in 'argument' is limited to tester components. In case of multipoint 'Message's the scope is additionally limited to the component of the 'sourceGate'. This implies that in case of local ordering, scope dependent data, such as 'Variable's, shall be associated with one of those components.

If a 'Message' is a trigger 'Message' (the 'isTrigger' property is set), the execution of the 'Message' 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 'Message'.

Generalization

- Interaction

Properties

- argument: DataUse [1]
  The contained 'DataUse' that is taken as the argument (data value) of this message.

- 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.

Constraints

- **'DataType' resolvable**
  If the 'argument' is 'DataInstanceUse', either the 'dataType' property or the 'dataInstance' property shall be provided. If the 'argument' is a 'DataElementUse', the 'dataElement' property shall be provided.
  inv: **DataTypeResolvable**:
  not self.argument.oclIsUndefined()

- **Type of message 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: **MessageArgumentAndGateType**:
  (self.argument.oclIsKindOf(AnyValue)
  and self.argument. resolveDataType().oclIsUndefined())
  or (self.sourceGate.gate.type.allDataTypes()-> exists(t | self.argument.resolveDataType().conformsTo(t))
  and self.target->forAll(t | t.targetGate.gate.type.allDataTypes()->exists(t |
  self.argument.resolveDataType().conformsTo(t))))

- **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 'Message' via the provided 'GateReference's.
  inv: **MessageArgumentVariableUse**:
  (not self.argument.oclIsKindOf(VariableUse)
  or (self.sourceGate.component = self.argument.oclAsType(VariableUse).componentInstance
  or self.target->exists(t |
  t.targetGate.component = self.argument.oclAsType(VariableUse).componentInstance)))

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

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

- **Conforming data type for 'argument' and 'variable'**
  If a 'Variable' is specified for a 'Target', the 'DataType' of 'DataUse' specification of the 'argument' shall conform to the 'DataType's of referenced 'Variable's of all 'Target's.
  inv: **MessageArgumentAndVariableType**:
  self.target->forAll(t | t.valueAssignment->size() = 0
  or not self.argument.resolveDataType().oclIsUndefined()
  and t.valueAssignment->forAll(v | self.argument.resolveDataType().conformsTo(v.variable.dataType)))

9.4.8 ProcedureCall

Semantics

A procedure call is a two-way interaction and consists of a call and a reply. The call is directed from the calling to called component and the reply is directed from called to calling component. For the calling component, a procedure call shall be synchronous: there shall be no other behaviours between call and reply. The called component may execute other behaviours between call and reply. The reply shall not be a starting event of 'ExceptionalBehaviour'.

A 'ProcedureCall' element represents one part of a procedure call. It is either a call behaviour or a reply behaviour. Each call behaviour shall have at least one reply behaviour and the latter shall specify related call behaviour in 'replyTo'. That is, a procedure call shall always consist of at least two 'ProcedureCall's. A 'ProcedureCall' without 'replyTo' shall be a call behaviour.

Any number of alternative reply behaviours may be specified for a single call behaviour. In that case all the reply 'ProcedureCall's shall be initial behaviours of 'Block's in an 'AlternativeBehaviour'.

NOTE: It is possible to define test behaviour in a way that at runtime a reply is never received. The requirement for specifying a reply to every call applies only to the TDL description of the behaviour.

A procedure call shall always have exactly two participants: the calling and the called component. There shall not be point-to-multipoint or intra-component procedure calls. The calling component shall be specified in 'sourceGate' of the call behaviour and in 'target' of the reply behaviour.

The arguments of a 'ProcedureCall' shall match the procedure signature defined by the signature attribute. The argument (data value) for a parameter shall represent either the data sent by the calling or the data sent by the called component depending on the 'ParameterKind' of the associated 'parameter'. Arguments for the 'In' parameters shall be specified for the 'ProcedureCall' that represents the call behaviour and arguments for the 'Out' and 'Exception' parameters for the reply 'ProcedureCall'.

'Out' and 'Exception' parameters shall not be mixed in a 'ProcedureCall'. If both kinds are expected then at least two 'ProcedureCall's shall be specified.

Generalization

- Interaction

Properties

- signature: ProcedureSignature [1]
  Signature of the called procedure.

- argument: ParameterBinding [1..*]
  Arguments of the called procedure.

- replyTo: ProcedureCall [0..1]
  The calling part of the procedure call that this 'ProcedureCall' is a reply to.

Constraints

- **Only point-to-point procedure calls**
  The 'target' of 'ProcedureCall' shall contain exactly one 'Target'.
  inv: **ProcedureCallTargetCount**:
  self.target->size() = 1

- **Each call has a reply**
  For every 'ProcedureCall' with empty 'replyTo' there shall be one or more 'ProcedureCall's that have this 'ProcedureCall' as 'replyTo'.
  inv: **ProcedureCallHasReply**:
  ProcedureCall.allInstances()->exists(pc | pc.replyTo = self)

- **Call and reply within the same 'TestDescription'**
  The 'ProcedureCall' referenced in the 'replyTo' shall be within the same 'TestDescription' as this 'ProcedureCall'.
  inv: **ProcedureCallAndReply:**
  self.replyTo.oclIsUndefined()
  or self.replyTo.getParentTestDescription() = self.getParentTestDescription()

- **Call and reply between same components**
  The 'sourceGate' and 'target' of a 'ProcedureCall' with 'replyTo' shall match the 'target' and 'sourceGate' of the 'ProcedureCall' in the 'replyTo'. That is, corresponding 'GateReference's shall be the equal.
  inv: **ProcedureCallReplyGates**:
  ProcedureCall.allInstances()->select(pc | pc.replyTo = self)->forAll(
  reply |
  reply.target->forAll(t | t.targetGate.component = self.sourceGate.component)
  and reply.target->forAll(t | t.targetGate.gate = self.sourceGate.gate)
  and self.target->forAll(t | t.targetGate.component = reply.sourceGate.component)
  and self.target->forAll(t | t.targetGate.gate = reply.sourceGate.gate))

- **Synchronous procedure calls**
  A 'ProcedureCall' with empty 'replyTo' shall not be followed by any behaviour in which the component specified in the 'sourceGate' is participating, other than a 'ProcedureCall' that specifies this 'ProcedureCall' as 'replyTo' or an 'AlternativeBehaviour' that contains such a 'ProcedureCall' in the beginning of a 'block'.
  inv: **ProcedureCallSynchronousCalling**
  let source = self.sourceGate.component,
  affectingBehaviours = self.container().oclAsType(Block).behaviour
  ->reject(b | b.oclIsKindOf(ActionBehaviour)
  and b.oclAsType(ActionBehaviour).componentInstance <> source)
  ->reject(b | b.oclIsKindOf(Interaction)
  and b.oclAsType(Interaction).sourceGate.component <> source
  and b.oclAsType(Interaction).target->forAll(t | t.targetGate.component <> source))
  ->reject(b| b.oclIsKindOf(TestDescriptionReference)
  and (not b.oclAsType(TestDescriptionReference).componentInstanceBinding->isEmpty()
  and not b.oclAsType(TestDescriptionReference).componentInstanceBinding
  .actualComponent->includes(self))),
  following = affectingBehaviours ->at(affectingBehaviours->indexOf(self) + 1)
  in (following.oclIsKindOf(ProcedureCall) and following.oclAsType(ProcedureCall).replyTo = self)
  or (following.oclIsKindOf(AlternativeBehaviour)
  and following.oclAsType(AlternativeBehaviour).block->exists(
  b | b.behaviour->first().oclIsKindOf(ProcedureCall)
  and b.behaviour->first().oclAsType(ProcedureCall).replyTo = self))

- **Type of procedure call**
  The 'ProcedureSignature' referred to in the 'procedure' shall be 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 'ProcedureCall'.
  inv: **ProcedureCallSignatureInGateTypes**
  self.sourceGate.gate.type.allDataTypes()->includes(self.signature)
  and self.target->forAll(targetGate.gate.type.allDataTypes()->includes(self.signature))

- **No mixing of parameters**
  All 'ParameterBinding's specified in the 'argument' shall refer to 'ProcedureParameter's of the same 'ParameterKind'.
  inv: **ProcedureParameterKind**
  self.argument->collect(pb | pb.parameter.oclAsType(ProcedureParameter).kind)
  ->asSet()->size() <= 1

- **Matching procedure arguments**
  For a 'ProcedureCall' with empty 'replyTo' there shall be one 'ParameterBinding' instance in the 'argument' for each 'ProcedureParameter' with kind 'In' in the associated 'ProcedureSignature'. For a 'ProcedureCall' with 'replyTo' there shall be one 'ParameterBinding' instance in the 'argument' for each 'ProcedureParameter' with kind 'Out' or 'Exception' in the associated 'ProcedureSignature'.
  inv: **ProcedureCallArguments**
  (self.replyTo.oclIsUndefined() and self.signature.parameter->select(p | p.kind = ParameterKind::In)
  ->forAll(p | self.argument.parameter->includes(p)))
  or (not self.replyTo.oclIsUndefined() and self.signature.parameter->reject(p | p.kind = ParameterKind::In)
  ->forAll(p | self.argument.parameter->includes(p)))

- **Use of variables in the 'argument' specification**
  The use of 'Variable's in the 'DataUse' specifications in 'ParameterBinding's shall be restricted to 'Variable's of 'ComponentInstance's that participate in this 'ProcedureCall' via the provided 'GateReference's.
  inv: **ProcedureCallVariableUse**
  self.argument
  ->closure(pb | pb.dataUse.argument)->union(self.argument)->collect(pb | pb.dataUse)
  ->select(du | du.oclIsKindOf(VariableUse))
  ->forAll(du | self.getParticipatingComponents()->includes(
  du.oclAsType(VariableUse).componentInstance))

- **Reply not starting event of exceptional behaviour**
  A 'ProcedureCall' that specifies replyTo shall not be the first behaviour of a block in an 'ExceptionalBehaviour'.
  inv: **ProcedureCallReplyNotInExceptional**
  self.replyTo.oclIsUndefined() or not self.container().container().oclIsKindOf(ExceptionalBehaviour)

9.4.9 Target

Semantics

A 'Target' holds the 'GateReference' that acts as target for the 'Interaction', which in turn contains this 'Target', and optional 'ValueAssignment's that store the received data values from the containing 'Interaction' to 'Variable's.

Generalization

- Element

Properties

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

- valueAssignment: ValueAssignment [0..*]
  Contained set of argument assignment specifications.

Constraints

- **Variable and target gate of the same component instance**
  The 'Variable's referenced by 'valueAssignment' shall exist in the same 'ComponentType' as the 'GateInstance' that is referred to by the 'GateReference' of the 'targetGate'.
  inv: **TargetComponent**:
  self.valueAssignment->isEmpty()
  or self.targetGate.component.type.allVariables()->includesAll(self.valueAssignment.variable)

- **Variable of a tester component only**
  If a 'ValueAssignment' is specified, the 'ComponentInstance' referenced by 'targetGate' shall be in the role 'Tester'.
  inv: **TargetVariableComponentRole**:
  self.valueAssignment->isEmpty () or self.targetGate.component.role = ComponentInstanceRole::Tester

9.4.10 ValueAssignment

Semantics

A 'ValueAssignment' is specified in the context of a 'target' of an 'Interaction'. It associates a 'Variable' of the 'ComponentInstance' specified for the target with an argument of the 'Interaction'. If the interaction is a 'ProcedureCall' then the 'ValueAssignment' shall specify the 'Parameter' whose corresponding runtime value is assigned to the 'variable'. The 'parameter' shall not be specified for 'Message's.

Generalization

- Element

Properties

- parameter: Parameter [0..1]
  Refers to the 'Parameter' of a 'ProcedureSignature' that specifies which argument shall be assigned to the 'variable'.

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

Constraints

- **Conforming data type for 'parameter' and 'variable'**
  If the 'parameter' is specified then its type shall conform to the type of the 'variable'.
  inv: **AssignedParamterType**:
  self.parameter.oclIsUndefined() or self.parameter.dataType.conformsTo(self.variable.dataType)

- **Parameter of associated procedure signature**
  If the 'parameter' is specified then it shall be contained in the 'ProcedureSignature' that is referred in the 'signature' of the 'ProcedureCall' containing this 'ValueAssignment'.
  inv: **AssignedProcedureParameter**:
  self.parameter.oclIsUndefined()
  or (self.container().container().oclIsKindOf(ProcedureCall) and
  self.container().container().oclAsType(ProcedureCall).signature.parameter->includes(self.parameter))

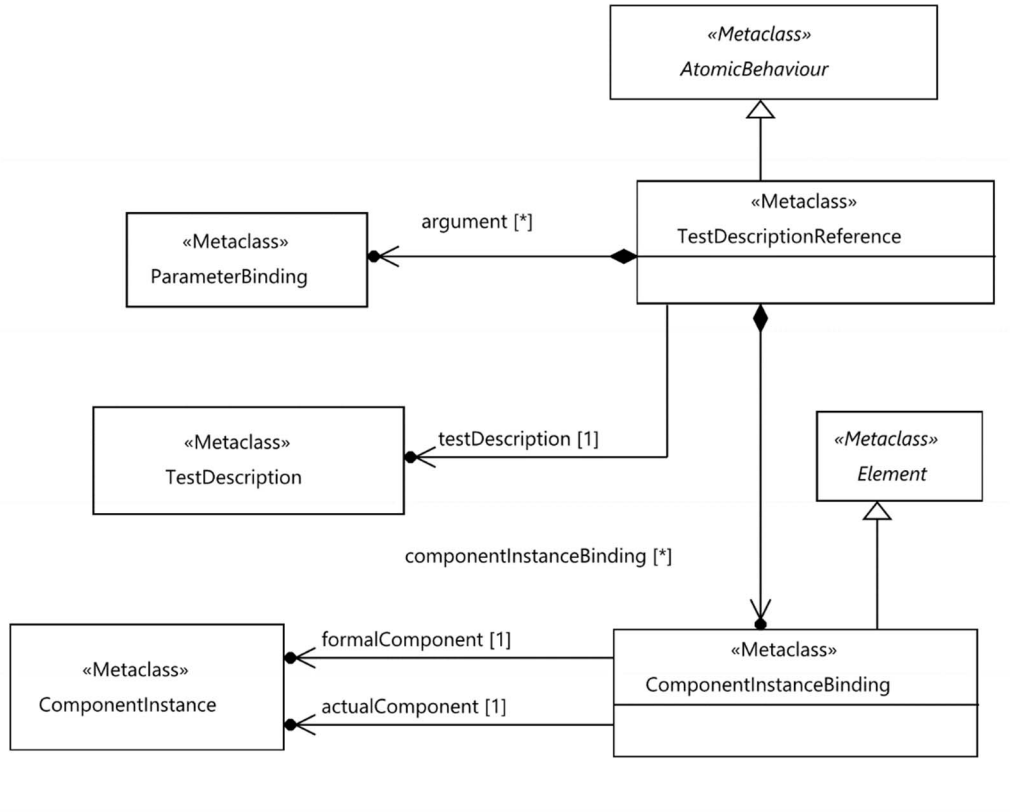

Figure 9.6: Test description reference

9.4.11 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. In case of locally ordered 'TestDescription', the execution and completion of invoked behaviour shall be independent for each 'ComponentInstance'.

A 'TestDescriptionReference' has a possibly empty list of arguments 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. In case of different 'TestConfiguration's, '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 'TestConfiguration's are identical. Explicit substitution is defined using the 'ComponentInstanceBinding'.

A 'TestConfiguration' *TConf2* of the referenced (invoked) 'TestDescription' is considered compatible with the 'TestConfiguration' *TConf1* of the referencing (invoking) 'TestDescription', under the provided 'ComponentInstanceBinding's between the 'ComponentInstance's in *TConf1* and *TConf2*, if:

- There are valid 'ComponentInstanceBinding's for all 'ComponentInstance's in *TConf2* to 'ComponentInstance's in *TConf1*.
- There are compatible 'Connection's between the bound 'ComponentInstance's in *TConf1* for all 'Connection's between the bound 'ComponentInstance's in *TConf2*. A 'Connection' in *TConf1* is compatible to a 'Connection' in *TConf2* if the same 'GateInstance's are used in the definition of the 'GateReference's of the 'Connection's.

NOTE 1: The compatibility between 'TestConfiguration's 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.

In case of local ordering, the scope of a 'TestDescriptionReference' consists of the 'ComponentInstance's referred to by the 'actualComponent' properties of the 'componentInstanceBinding' values or the 'ComponentInstance's of the 'TestConfiguration' when no bindings are specified.

Generalization

- AtomicBehaviour

Properties

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

- argument: ParameterBinding [0..*] {ordered}
  Refers to an ordered set of arguments 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

- **All test description parameters bound**
  For each 'FormalParameter' defined in 'formalParameter of' the referenced 'TestDescription' there shall be a 'ParameterBinding' in 'argument' that refers to that 'FormalParameter' in 'parameter'.
  inv: **AllTestDescriptionParametersBound**:
  self.testDescription.formalParameter->forAll(p | self.argument->exists(a | a.parameter = p))

- **No use of variables in arguments**
  In locally ordered 'TestDescription's, 'DataUse' expressions used to describe arguments shall not contain variables directly or indirectly in 'TestDescriptionReference's.
  inv: **NoVariablesInLocallyOrderedTestDescriptionReference**:
  self.argument. dataUse->forAll(du |
  not du.oclIsKindOf(VariableUse)
  and du.argument->forAll(a | not a.dataUse.oclIsKindOf(VariableUse))
  and du.argument->closure(a | a.dataUse.argument)->forAll(a |
  not a.dataUse.oclIsKindOf(VariableUse)))

- **No use of time labels in arguments**
  In locally ordered 'TestDescription's, 'DataUse' expressions used to describe arguments shall not contain time labels directly or indirectly in 'TestDescriptionReference's.
  inv: **NoTimeLabelsInLocallyOrderedTestDescriptionReference**:
  self.argument.dataUse->forAll(du |
  not du.oclIsKindOf(TimeLabelUse)
  and du.argument->forAll(a | not a.dataUse.oclIsKindOf(TimeLabelUse))
  and du.argument->closure(a | a.dataUse.argument)->forAll(a |
  not a.dataUse.oclIsKindOf(TimeLabelUse)))

- **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 'TestConfiguration' of the referenced (invoked) 'TestDescription' shall be compatible with the 'TestConfiguration' of the referencing (invoking) 'TestDescription' under the provided 'ComponentInstanceBinding's between the 'ComponentInstance's of the 'TestConfiguration's of referenced and referencing 'TestDescription's.
  inv: **CompatibleConfiguration**:
  self.componentInstanceBinding->isEmpty()
  or self.testDescription.testConfiguration.compatibleWith(
  self.getParentTestDescription().testConfiguration, self.componentInstanceBinding)

- **No combining of local and total ordering**
  The referenced 'TestDescription' shall have the same ordering assumption as the referencing 'TestDescription'.
  inv: **LocalAndTotalOrdering**:
  self.getParentTestDescription().isLocallyOrdered = self.testDescription.isLocallyOrdered

9.4.12 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 9.4.11.

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

- **Conforming component types**
  The 'ComponentType' of the actual 'ComponentInstance' shall conform to 'ComponentType' of the formal 'ComponentInstance.
  inv: **BindingComponentTypes**:
  self.actualComponent.type.conformsTo(self.formalComponent.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

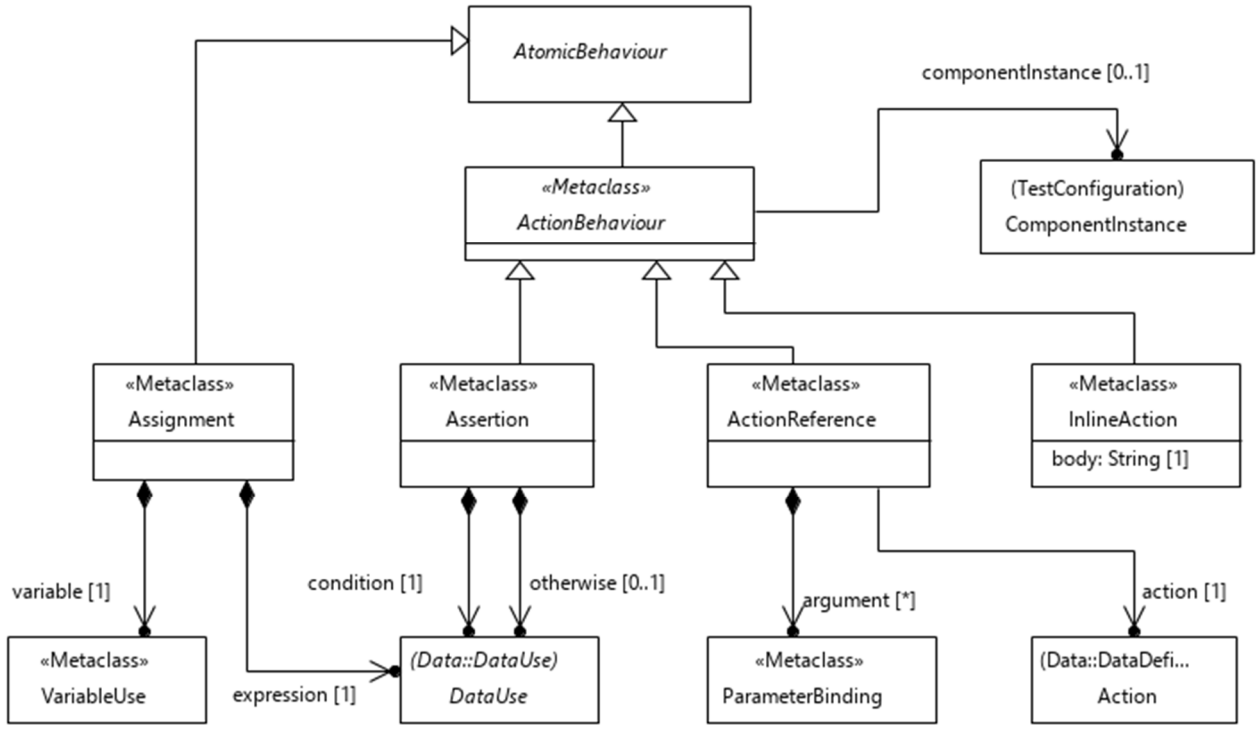

Figure 9.7: Action behaviour concepts

9.4.13 ActionBehaviour

Semantics

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

It may refer to a 'Tester' 'ComponentInstance' that specifies the scope in which the 'ActionBehaviour' is executed. If no reference to a 'ComponentInstance' is given, the scope of the 'ActionBehaviour' is the scope of the containing 'Block' and it does not affect the scope of the 'Block'.

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 = ComponentInstanceRole::Tester

- **Known 'componentInstance' with locally-ordered behaviour**
  The 'ComponentInstance' that an 'ActionBehaviour' refers to shall be specified if the 'ActionBehaviour' is used within a locally-ordered 'TestDescription'.
  inv: **ActionBehaviourComponentInstance**:
  not self.componentInstance.oclIsUndefined() or not self.getParentTestDescription().isLocallyOrdered

9.4.14 ActionReference

Semantics

An 'ActionReference' invokes an 'Action'. It may carry a list of 'ParameterBinding' specifications to denote arguments of this 'Action'.

Generalization

- ActionBehaviour

Properties

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

- argument: ParameterBinding [0..*] {ordered, unique}
  Refers to an ordered set of arguments passed to the referenced action.

Constraints

- **All action parameters bound**
  For each 'FormalParameter' defined in 'formalParameter of' the referenced 'Action' there shall be a 'ParameterBinding' in 'argument' that refers to that 'FormalParameter' in 'parameter'.
  inv: **AllActionParametersBound**:
  self.action.formalParameter->forAll(p | self.argument.parameter->includes(p))

- **No 'Function's in 'ActionReference'**
  The referenced 'Action' shall not be a 'Function'.
  inv: **ActionReferenceFunction**:
  not self.action.oclIsTypeOf(Function)

9.4.15 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.

9.4.16 Assignment

Semantics

An 'Assignment' denotes the assignment of a value that is expressed as a 'DataUse' specification to a 'Variable' or a 'Member' of a 'Variable' (by means of the 'reduction' property) within a 'ComponentInstance'.

The 'ComponentInstance' of the 'VariableUse' which is assigned the data value resulting from the evaluation of the 'expression' specifies the scope in which the 'Assignment' is executed. It does not affect the scope of the containing 'Block'.

Generalization

- AtomicBehaviour

Properties

- variable: VariableUse [1]
  Refers to the 'Variable' or the 'Member' of a 'Variable' that is assigned the data value resulting from the evaluation of the 'expression'.

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

Constraints

- **Conforming data type**
  The provided 'DataUse' expression shall conform to the 'DataType' of the referenced 'Variable'.
  inv: **AssignmentDataType**:
  self.expression.resolveDataType().conformsTo(self.variable.variable.dataType)

- **Empty 'argument' set for 'variable'**
  The 'argument' and 'reduction' sets shall be empty.
  inv: **AssignmentVariableArgument**:
  self.variable.argument->isEmpty()

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 Integer

The predefined 'SimpleDataType' 'Integer' denotes the common integer data type with countable integral numeric values (instances of 'SimpleDataInstance').

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

10.2.3 String

The predefined 'SimpleDataType' 'String' denotes the common string data type with values containing sequences of alpha-numeric characters (instances of 'SimpleDataInstance').

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

10.2.4 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.

10.3 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 10.2.4). 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 10.2.4). 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 10.2.4). 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 'PredefinedFunction' 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.

If the parameter and return types are extended by other types, instances of the directly or indirectly extending types can be used as well. If instances of one type are combined with instances of a directly or indirectly extending type, the return type shall be the more generic type. If both parameters are of the same extending type, then the return type shall be of the same extending type as well.

No assumptions are made about how these functions are implemented in an underlying concrete type system. Unless specified otherwise, the arguments for the predefined functions shall be fully specified (in the case of 'StructuredDataInstance's) and exclude the use of 'SpecialValueUse' (either directly as arguments or nested within 'StructuredDataInstance's). If the arguments are not fully specified or include 'SpecialValueUse's, this shall result in an error during execution.

10.5.2 Functions of Return Type 'Boolean'

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

- _==_: **instanceOf**(DataUse), **instanceOf**(DataUse) → Boolean
  Denotes equality of the results from the evaluation of the 'DataUse's supplied as arguments. The 'DataUse's, shall refer to the same 'DataType'. Equality shall be determined based on content and not on identity.
- _!=_: **instanceOf**(DataUse), **instanceOf**(DataUse) → Boolean
  Denotes inequality of the results from the evaluation of the 'DataUse's supplied as arguments. The 'DataUse's, shall refer to the same 'DataType'. Inequality shall be determined based on content and not on identity.
- _and_: Boolean, Boolean → Boolean
  Denotes the standard logical AND operation.
- _or_: Boolean, Boolean → Boolean
  Denotes the standard logical OR operation.
- _xor_: Boolean, Boolean → Boolean
  Denotes the standard logical exclusive OR operation.
- not: Boolean → Boolean
  Denotes the standard logical NOT operation.
- _<_: Integer, Integer → Boolean
  Denotes the standard mathematical less-than operation.
- _>_: Integer, Integer → Boolean
  Denotes the standard mathematical greater-than operation.
- _<=_: Integer, Integer → Boolean
  Denotes the standard mathematical less-or-equal operation.
- _>=_: Integer, Integer → Boolean
  Denotes the standard mathematical greater-or-equal operation.

10.5.3 Functions of Return Type 'Integer'

The following functions of return type conforming to 'Integer' shall be predefined:

- _+_: Integer, Integer → Integer
  Denotes the standard arithmetic addition operation.
- _-_: Integer, Integer → Integer
  Denotes the standard arithmetic subtraction operation.
- _*_: Integer, Integer → Integer
  Denotes the standard arithmetic multiplication operation.
- _/_: Integer, Integer → Integer
  Denotes the standard arithmetic integer division operation.
- _mod_: Integer, Integer → Integer
  Denotes the standard arithmetic modulo operation.
- size: **instanceOf**(CollectionDataInstance) → Integer
  Returns the number of members in the 'CollectionDataInstance'.

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:

- value: **instanceOf**(Time) → Integer
  Returns the value of the 'Time' instance as an instance of 'Integer' for use with other functions.

10.6 Predefined Instances of the 'AnnotationType' Element

10.6.1 Master

The predefined 'AnnotationType' 'Master' is the 'key' of an 'Annotation' that may be attached to a 'TestDescription'. A 'TestDescription' containing this kind of 'Annotation' shall identify an entry point for test execution. The 'Master' 'AnnotationType' is provided to allow external tools to adapt to a specific execution order of a set of 'TestDescription's.

10.6.2 MappingName

The predefined 'AnnotationType' 'MappingName' is the 'key' of an 'Annotation' that may be attached to a 'DataResourceMapping' to indicate which source or target language or specification format the mapping applies to. The value of the 'Annotation' shall be a distinctive name of a tool or framework that is able to process the annotated mappings. Common names of languages or protocols shall only be used for mappings that are specified according to an ETSI deliverable.

10.6.3 Version

The predefined 'AnnotationType' 'Version' is the 'key' of an 'Annotation' that may be attached to a top level 'Package' to indicate the version number assigned to the contents of the package. The value of the 'Annotation' shall be a string consisting of 1 to 3 numbers separated by dots ('.').

NOTE: One common format for specifying versions is semantic versioning that is specified using following patter: MAJOR.MINOR.PATCH.

10.6.4 check

The predefined 'AnnotationType' 'check' is the 'key' of an 'Annotation' that may be attached to a 'DataUse' (especially 'LiteralValueUse' or 'AnyValueUse') to indicate a condition for the 'DataUse'. The specification and semantics for the condition is outside the scope of the present document.

10.6.5 where

The predefined 'AnnotationType' 'where' is the 'key' of an 'Annotation' that may be attached to a 'DataUse' (especially 'LiteralValueUse' or 'AnyValueUse') to indicate a condition for the 'DataUse'. The specification and semantics for the condition is outside the scope of the present document.

10.7 Predefined Instances of 'ConstraintType' Element

10.7.1 length

Applicable to 'CollectionDataType's, 'SimpleDataType's and 'Member's that are of a 'SimpleDataType' or 'CollectionDataType'. Specifies the exact length of the instances of a 'CollectionDataType' a 'SimpleDataType'. When applied to 'SimpleDataType' then the method of determining the length of an instance or a literal is outside the scope of the present document.

10.7.2 minLength

Applicable to 'CollectionDataType's, 'SimpleDataType's and 'Member's that are of a 'SimpleDataType' or 'CollectionDataType'. Specifies the minimum length of the instances of a 'CollectionDataType' a 'SimpleDataType'. When applied to 'SimpleDataType' then the method of determining the length of an instance or a literal is outside the scope of the present document.

10.7.3 maxLength

Applicable to 'CollectionDataType's, 'SimpleDataType's and 'Member's that are of a 'SimpleDataType' or 'CollectionDataType'. Specifies the maximum length of the instances of a 'CollectionDataType' a 'SimpleDataType'. When applied to 'SimpleDataType' then the method of determining the length of an instance or a literal is outside the scope of the present document.

10.7.4 range

Applicable to 'SimpleDataType's and 'Member's that are of a 'SimpleDataType'. Specifies minimum and maximum values that are allowed. The method of comparing the value of an instance to the specified minimum and maximum values is outside the scope of the present document.

10.7.5 format

Applicable to 'SimpleDataType's and 'Member's that are of 'SimpleDataType'. Specifies the allowed format of a value. The syntax of the format and the method of validating the value against that format is outside the scope of the present document.

10.7.6 union

Applicable to 'StructuredDataType's. Specifies that only one 'Member' shall be assigned in associated 'StructuredDataInstance's.

NOTE: The term union in TDL has the opposite meaning compared to the common understanding of the word.

10.7.7 uniontype

Applicable to 'StructuredDataType's. Specifies that the type inherits the 'Member's from only one of the 'StructuredDataType's that it extends.

NOTE: By default, a 'StructuredDataType' inherits all the 'Member's from all the 'StructuredDataType's that it extends.

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

The technical representation of the TDL meta-model is included as an electronic attachment, and is contained in archive es_20311901v010801p0.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): Legacy Examples of a TDL Concrete Textual Syntax

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 create TDL specifications. Though a concrete syntax is 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.

The standardized textual syntax for TDL as well as illustrative examples can be found in ETSI ES 203 119-8 [8]. The latest specification and examples of the legacy textual syntax previously included in the present document is available in the TDL Open Source Project (TOP) [i.4].

Annex C (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® formal/13-04-03: "UML Testing Profile (UTP) V1.2".
- ETSI ES 203 119-2 (V1.4.1): "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	September 2016	Publication
V1.4.1	May 2018	Publication
V1.5.1	August 2020	Publication
V1.6.1	May 2022	Publication
V1.7.1	December 2023	Publication
V1.8.1	May 2025	MAP process MV 20250707: 2025-05-08 to 2025-07-07
V1.8.1	July 2025	Publication