INTERNATIONAL STANDARD

ISO 13209-2

Second edition
2022-07

Road vehicles — Open Test sequence eXchange format (OTX) —

Part 2: Core data model specification and requirements

Véhicules routiers — Format public d'échange de séquence-tests (OTX) —

Partie 2: Exigences et spécifications du modèle de données central

Reference number
ISO 13209-2:2022(E)

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Page

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of ISO documents should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT), see www.iso.org/iso/foreword.html.

This document was prepared by Technical Committee ISO/TC 22, Road vehicles, Subcommittee SC 31, Data communication.

This second edition cancels and replaces the first edition (ISO 13209-2:2012), which has been technically revised.

The main changes are as follows:

— introduction of new extension interfaces, e.g. for procedure realisations, compound nodes, and **`NamedAndSpecified`**;

— made **`ForEachLoop`** more convenient;

— new terms added (e.g. **`RoundToNearest`**);

— introduction of **`MutexLock`**;

— added new checker rules;

— deprecated **`TerminateLanes`** node.

A list of all parts in the ISO 13209 series can be found on the ISO website.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

Introduction

Diagnostic test sequences are utilized whenever automotive components or functions with diagnostic abilities are being diagnosed, tested, reprogrammed or initialized by off-board test equipment. Test sequences define the succession of interactions between the user (i.e. workshop or assembly line staff), the diagnostic application (the test equipment) and the vehicle communication interface as well as any calculations and decisions that have to be carried out. Test sequences provide a means to define interactive, guided diagnostics or similar test logic.

Today, the automotive industry mainly relies on paper documentation and/or proprietary authoring environmments to document and to implement such test sequences for a specific test application. An author who is setting up engineering, assembly line or service diagnostic test applications needs to implement the required test sequences manually, supported by non-uniform test sequence documentation, most likely using different authoring applications and formats for each specific test application. This redundant effort can be greatly reduced if processes and tools support the OTX concept.

The ISO 13209 series proposes an open and standardized format for the human- and machine-readable description of diagnostic test sequences. The format supports the requirements of transferring diagnostic test sequence logic uniformly between electronic system suppliers, vehicle manufacturers and service dealerships/repair shops.

This document represents the requirements and technical specification for the fundament of the OTX format, namely the “OTX core”. The core describes the basic structure underlying every OTX document. This comprises detailed data model definitions of all required control structures by which test sequence logic is described, but also definitions of the outer, enveloping document structure in which test sequence logic is embedded. To achieve extensibility the core also contains well-defined extension points that allow a separate definition of additional OTX features—without the need to change the core data model.

ISO 13209-3[2] extends the core by a set of additional features, using of the core extension mechanism (which may also be applied for proprietary extensions).

This document is the most generic and stand-alone part of the ISO 13209 series. In principle, it is also applicable in other areas for any sequential logic description, even outside the automotive domain. Automotive-specific features are, therefore, contained solely in ISO 13209-3[2].

Road vehicles — Open Test sequence eXchange format (OTX) —

Part 2: Core data model specification and requirements

1 Scope

This document defines the OTX core requirements and data model specifications.

The requirements are derived from the use cases described in ISO 13209-1. They are listed in the requirements section.

The data model specification aims at an exhaustive definition of all OTX core features implemented to satisfy the core requirements. Since OTX is designed for describing test sequences, which themselves represent a kind of program, the core data model follows the basic concepts common to most programming languages.

Thus, this document establishes rules for syntactical entities like parameterised procedures, constant and variable declarations, data types, basic arithmetic, logic and string operations, flow control statements like loop, branch or return, simple statements like assignment or procedure call as well as exception handling mechanisms. Each of these syntactical entities is accompanied by semantic rules which determine how OTX documents are interpreted. The syntax rules are provided by UML class diagrams and XML schemas, whereas the semantics are given by UML activity diagrams and prose definitions.

With respect to documentation use cases, special attention is paid to defining a specification/realisation concept (which allows for "hybrid" test sequences: human readable test sequences that are at the same time machine-readable) and so-called floating comments (which can refer to more than one node of the sequence).

The core data model does not define any statements, expressions or data types that are dependent on a specific area of application.

For the convenience of the user, the ISO 13209-2 OTX XML schema definition file (XSD) is published alongside this document.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO 13209-1, *Road vehicles — Open Test sequence eXchange format (OTX) — Part 1: General information and use cases*

ISO 22901 (all parts), *Road vehicles — Open diagnostic data exchange (ODX)*

IEEE 754:2019, *IEEE Standard for Floating-Point Arithmetic*

RFC 2045, *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*

RFC 2046, *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*

W3C XSD (all parts):2012, *W3C Recommendation: W3C XML Schema Definition Language (XSD) 1.1*

W3C XML:2008, *W3C Recommendation: Extensible Markup Language (XML) 1.0*

W3C XLink:2010, *W3C Recommendation: XML Linking Language (XLink) Version 1.1*

3 Terms, definitions and abbreviated terms

3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO 13209-1 and the following apply.

ISO and IEC maintain terminology databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at https://www.electropedia.org/

3.1.1
attribute
<UML> property of a UML class

3.1.2
attribute
<XSD/XML> named property of an XSD complex type or an XML element

3.1.3
constant
identifier of a non-writable memory location

3.1.4
context
environmental circumstances which influence *test sequence* (3.1.11) execution

Note 1 to entry: OTX test sequences can be configured to behave differently according to different context situations. Contextual information depends on factors such as the particular vehicle that is currently attached to the test application (e.g. the current vehicle's model type, the engine type), on the test application settings (e.g. a setting controlling whether the test sequence shall run in debug mode) or on other factors such as whether the test sequence is running in a manufacturing or a service workshop environment, etc.

3.1.5
expression
syntactical construct which describes a specific computation with a set of arguments and a single return value

3.1.6
identification routine
method or software by which a diagnostic application identifies contextual information

3.1.7
procedure signature
description of the interface of an OTX procedure

3.1.8
reference
value which refers to data in memory

3.1.9
session
instance of *test sequence* (3.1.11) execution

3.1.10
term
value described by and computed from an *expression* (3.1.5)

3.1.11
test sequence
test procedure (3.1.12) defining a full test

Note 1 to entry: A test sequence is also a procedure, but not all procedures are test sequences. In an OTX document, the procedure representing a test sequence shall be named "main". By using procedures, a test sequence may be split into several procedure modules. An adequately assembled set of frequently needed procedures may serve as a library which provides procedures that can be called from any other (client) procedure or test sequence.

3.1.12
test procedure
procedure
stand-alone, parameterisable flow of OTX actions that can be called from other OTX procedures

3.1.13
validity
named Boolean *expression* (3.1.5) used for activating/deactivating parts of the OTX *test sequences* (3.1.11) according to the current *context* (3.1.4) situation

Note 1 to entry: Parts of OTX test sequences which are marked with a validity name shall be executed only if the associated Boolean expression is true according to the current context situation.

3.1.14
variable
identifier of a writable memory location

Note 1 to entry: The term "variable" is used as a collective term for document scope variables, local variables, non-constant parameters and also items in non-constant lists or maps or other compound data structures. In OTX, these can be addressed by giving the identifier of the variable or parameter, optionally accompanied by a path into compound data structures which allows the inner parts of variables or parameters to be addressed.

3.2 Abbreviated terms

API	Application Programming Interface
IFD	Interface Definition (OTX extension)
JRE	Java Runtime Environment
NOP	No Operation Performed
OEM	Original Equipment Manufacturer
OTX	Open Test sequence eXchange
UML	Unified Modelling Language
XML	Extensible Markup Language
XSD	XML Schema Definition

4 Requirements and recommendations

4.1 General

Since OTX is merely a static data format and not a software application, it shall be kept in mind that all of the following entries (requirements or recommendations) are related to static format features and **not** to the behaviour of any OTX-based software product. All such products are indirectly affected by the requirements or recommendations given in this document considering that they shall be able to write, read or execute valid OTX documents according to the rules given in this document. Aside from that, requirements towards any such product are not in the scope of this document.

4.2 Basic principles for requirements and recommendations definition

Basic principles have been established as a guideline to define the OTX requirements or recommendations:

— OTX requirements or recommendations specify the conditions that the OTX data model and format shall satisfy,

— all stakeholders (system suppliers, OEMs, tool suppliers), which offer diagnostic test procedures are expected to implement and follow the requirements of this document,

— the content of OTX documents and the quality of the information is the responsibility of the originator,

— the runtime system defines an OTX home directory. It is the entry directory for all relative file and directory access (see Annex F).

4.3 Clustering of requirements and recommendations

Table 1 provides an overview of the main categories of OTX requirements or recommendations. Each category may have one or more entries.

Table 1 — Main requirements clustering

#	Main title of requirement cluster	Brief description
1	General format and language requirements	Requirements or recommendations regarding the general aspects like the chosen programming paradigm, file format (XML), etc.
2	Test sequence development process support	Requirements or recommendations about different stages in the test procedure authoring process, outlining human-readable (documentation) versus machine-readable (execution) test procedures
3	Language feature details	Requirements or recommendations concerning details like declarations, data types, expressions, statements, etc.
4	Boundaries	Features that should not be part of OTX

4.4 Entries priorities

Each of the following requirements and recommendations carries a priority-attribute which can be set to SHALL or SHOULD.

— SHALL:
The requirement represents stakeholder-defined characteristics, the absence of which will result in a deficiency that cannot be compensated by other means.

— SHOULD:
If the recommendation-defined characteristic is not or not fully implemented in the data model, it

does not result in a deficiency, because other features in the data model can be used to circumvent this.

4.5 General format and language aspects

Core_R01 – Machine readable format

Priority: SHALL

Rationale: The focus of OTX is on the exchange of data between tools in the vehicle diagnostic process. To leverage highest efficiency, the tools shall be able to operate automatically on OTX files (e.g. for importing and exporting of OTX-relevant data).

Description: OTX format shall be machine-readable to allow a tool to open an existing document for editing, checking, displaying or executing.

Core_R02 – Platform independence

Priority: SHALL

Rationale: If OTX would bind to specific hardware, operating system or application, its potential usages are diminished and applicability of the ISO 13209 series is decreased.

Description: OTX shall not be dependent on any specific hardware or software platform. OTX shall not be bound to any particular hardware, operating system or application.

Core_R03 – Well-defined syntax and semantics

Priority: SHALL

Rationale: OTX shall be a machine-readable data format. This implies an unambiguously defined syntax and semantics.

Description: All OTX elements shall be defined clearly (syntax + semantics). For the syntax definition, XML Schema shall be used. For the behavioural/semantics specification, a prose description shall exist.

Core_R04 – Universal language

Priority: SHALL

Rationale: Diagnostic applications can be seen as domain-specific computer programs. These require complex computations and no limits are known or foreseen today that allow OTX to be restricted with respect to Turing-completeness.

Description: OTX shall have the ability to solve any computable problem (Turing-completeness).

NOTE Legacy sequences can (theoretically) be transformed to OTX and back, if the legacy sequence format and OTX are Turing-complete.

Core_R05 – Minimal language

Priority: SHOULD

Rationale: Fulfilment of this recommendation reduces the implementation effort necessary to integrate OTX into tools and is thus a very relevant market-driving factor for OTX.

Description: OTX should be defined with the minimal set of language elements necessary to reach Turing-completeness.

OTX is required to be not designed for comfort of expressing computational programs (as are programming languages like Java, C++ or Delphi), but rather for effectiveness of transporting diagnostic application knowledge unambiguously between different tools/parties in the diagnostic process.

Core_R06 – Structured programming approach

Priority: SHALL

Rationale: Structured programming can be seen as a subset or sub discipline of procedural programming, one of the major programming paradigms. It removes reliance on the GOTO statement for controlling the flow of a program. Using GOTO statements in programming often leads to a complex, tangled and unreadable control structure, which is clearly not desired in OTX.

Description: OTX shall follow the structured programming approach. Only flow control statements branch, loop return, continue, break and throw may implicitly induce jumps. The behaviour of these jumps shall be well-defined in the prose semantic documentation of each of these statements. An explicit GOTO statement which allows to jump anywhere in the procedure **shall not** be supported.

Core_R07 – Imperative structure

Priority: SHALL

Rationale: Test procedures are usually considered as a procedure of commands that need to be executed one after one by a runtime system. Since the imperative programming paradigm matches exactly for this concept, it is well suited for OTX.

Description: OTX shall only support program structures that can be translated by a compiler into imperative programming languages.

Core_R08 – Extensibility

Priority: SHALL

Rationale: The scope of diagnostic applications in the diagnostic process is wide. Engineering, production and after sales applications interface numerous and diverse devices, server applications and modules, which cannot be completely addressed with the first release of the standard and which evolve over time.

Description: OTX shall be extendable to integrate means to access new technology employed within the diagnostic process. It shall be possible to integrate interfaces of various base technologies into OTX.

4.6 Test sequence development process support

Core_R09 – Embed non-machine readable content

Priority: SHALL

Rationale: Use cases will occur where diagnostic applications shall be expressed in OTX but, for example the interfaces to all used devices are not available (e.g. how to communicate to a nut runner). In this case it would be preferable to express the diagnostic application in OTX and express the non-standardized device access in prose or in pseudo code. An OTX-compliant tool could then import such a file and mark the parts of the diagnostic application that need to be replaced with executable content by a diagnostics engineer.

Description: OTX shall provide means to express parts of a diagnostic application in a non-machine readable format. This non-machine readable content shall be clearly marked so that processes operating on OTX files can identify it.

Core_R10 – High level test procedure

Priority: SHALL

Rationale: In a step-wise test procedure design process, it can become necessary to specify procedures in prose-form only. Skeletal control structures might already be part if this high-level description, but the details of implementation might not be known at design time (loop conditions, exact service names, etc.).

Description: It shall be possible to describe test procedures at a high level.

Core_R11 – Exchange high level test procedure

Priority: SHALL

Rationale: A test procedure specified in prose-form only shall nevertheless pose a valid OTX document, even though it is not executable.

Description: It shall be possible to exchange a high-level test plan using a plain text description.

Core_R12 – Exchange a fully functional test procedure

Priority: SHALL

Rationale: A test procedure containing no prose-form, but only implementation details shall nevertheless pose a valid OTX document, even though it is not easily human-readable.

Description: It shall be possible to mix high-level description and implementation details on the same procedure.

Core_R13 – Exchange an intermediate stage test procedure

Priority: SHALL

Rationale: A test procedure containing a mix of prose and fully implemented parts shall nevertheless pose a valid OTX document.

Description: It shall be possible to mix high-level description and implementation details on the same procedure.

Core_R14 – Floating comments

Priority: SHALL

Rationale: Situations will occur where comments are needed that can be freely attached to parts of the flow of commands in a test procedure. Such comments shall not be locally bound or contained within single statements; they shall be defined aside from the flow and only point to parts of it. Comments are purely informational nodes that shall not be relevant for execution of a test procedure.

Description: It shall be possible add floating comments to a test procedure that can refer to one or more statements its flow or its sub-flows at any block depth.

4.7 Language feature details

4.7.1 Declarations

Core_R15 – Declarations

Priority: SHALL

Description: OTX shall support the declaration of constants and variables as well as test procedure parameters. A declaration shall contain a name, a data type, an optional initialization value and an optional description.

Core_R16 – Initialisation

Priority: SHALL

Rationale: It shall be possible to set the initial value for an identifier to a value other than the default.

Description: OTX shall support the optional initialization of declared identifiers.

Core_R17 – Constant declarations

Priority: SHALL

Rationale: There will be cases when an OTX author wants to guarantee that the value of an identifier in the test procedure cannot be changed. Therefore, the author needs to have a means to mark an identifier as a constant. The value of a constant is not allowed to change during the lifetime of the constant.

Description: OTX shall support the declaration of constants. Constants shall be set with the declaration (initialization is mandatory).

Core_R18 – Variable declaration

Priority: SHALL

Rationale: In order to reflect the fact that an identifier can change its value during procedure execution, a means for marking identifiers as variables is needed. The value of a variable is allowed to change during procedure execution; new values can be assigned to it.

Description: OTX shall support the declaration of variables.

Core_R19 – Input parameter declarations

Priority: SHALL

Rationale: When a test procedure is called, information will need to be passed to the called procedure.

Description: OTX shall support declaration of any number of input parameters that are passed to the test procedure from the caller.

Core_R20 – Output parameter declarations

Priority: SHALL

Rationale: When a test procedure is called, information will need to be retrieved from the called procedure, after it has executed.

Description: OTX shall support declaration of any number of output parameters. Output parameters shall be assignable to variables in the calling test procedure.

Core_R21 – Two-way parameter declarations

Priority: SHOULD

Rationale: There are situations when a variable reference shall be passed to a called test sequence, so that any modifications will be visible to the caller also.

Description: OTX should support declaration of any number of two-way parameters. Any declared two-way parameter is passed from the caller to the test procedure. Any changes of the value of a two-way parameter can be assigned to variables of the caller test sequence.

NOTE This is the combination of input and output parameters.

4.7.2 Data types

Core_R22 – Strong typing

Priority: SHALL

Rationale: In order to have the possibility to translate into various other languages, OTX shall be strong typed. This means that the binding of a variable to a data type persists during the variables whole lifetime.

Description: OTX shall be a strong typed, static checked language.

Core_R23 – Data types

Priority: SHALL

Rationale: To maintain a state during test-procedure execution, information needs to be stored in memory. Since the type and the structure of the stored information needs to be known for a typed language (be it a string, an integer or a map, etc.) declared parameters, it shall be possible to mark variables and constants with the corresponding data type.

Description: OTX shall support a well-defined set of data types.

Core_R24 – Extension mechanism for data types

Priority: SHALL

Rationale: There will be situations when the predefined set of data types will not be sufficient. Therefore, OTX shall be extensible by new data types.

Description: It shall be possible to extend the set of data types by new data types. This shall happen by a well-defined extension mechanism.

Core_R25 – Raw memory data type

Priority: SHALL

Description: OTX shall support a raw memory data type.

Core_R26 – Integer data type

Priority: SHALL

Description: OTX shall support an integer data type.

Core_R27 – Floating-point data type

Priority: SHALL

Description: OTX shall support a floating-point data type.

Core_R28 – String data type

Priority: SHALL

Description: OTX shall support a string data type.

Core_R29 – Boolean data type

Priority: SHALL

Description: OTX shall support a Boolean data type.

Core_R30 – Container data type

Priority: SHALL

Description: OTX shall support at least one container data type. It shall be possible to access, dynamically add and remove elements in the container. The elements in the container shall all be of the same data type. Recursive declaration shall be possible (e.g. container of containers of integers).

4.7.3 Expressions

Core_R31 – Expressions

Priority: SHALL

Description: OTX shall support expressions. At runtime, it shall be possible to evaluate such an expression in a well-defined way. After evaluation, it has exactly one return value of a defined data type and shall not have any side-effects. This means, no data from the test procedure is allowed to be changed by evaluation of the expression but the variable where the evaluated value will be assigned to.

NOTE In general, OTX expressions correspond to the term "function" in the mathematical sense.

Core_R32 – Extension mechanism for expressions

Priority: SHALL

Description: It shall be possible to extend the set of expressions by new expression types. This shall happen by a well-defined extension mechanism.

Core_R33 – Literal expressions

Priority: SHALL

Description: OTX shall support literal expressions defined for each simple data type and for collection types (lists, maps, etc.). The literal shall represent the value of the literal expression directly.

Core_R34 – Dereferencing expressions

Priority: SHALL

Description: OTX shall support dereferencing expressions. They shall allow reading data referenced by parameter-, variable- or constant-names.

Core_R35 – Combined expressions: functions

Priority: SHALL

Rationale: OTX shall allow creation of higher-level expressions (functions) that contain other expressions as arguments. The evaluation shall happen recursively.

Description: OTX shall support a well-defined set of functions.

IMPORTANT — Most programming languages describe very frequently used mostly mathematical functions by operators (+ for add(a,b), * for multiply(a,b), etc.) There shall be no special treatment of operators in OTX; they shall rather be described like functions. It is the task of OTX tools to show them as operators, if needed.

Core_R36 – Basic function set

Priority: SHALL

Description: OTX shall support the basic functions:

— mathematical: addition, subtraction, multiplication, division, negation, modulo, power,

— bitwise: conjunction, disjunction, negation,

— relations: equality, greater than, less than,

— logical: conjunction, disjunction, negation,

— string: concatenation.

Core_R37 – Statements

Priority: SHALL

Rationale: Statements are needed to represent single commands or higher-level control structures in a procedure. In order to achieve Turing-completeness, a set of basic statements with different semantics shall be part of OTX. Statements are made up out of distinct parts, for example, in a loop statement, parts are a condition which is a logic expression, and a sub-sequence of commands. A procedure call statement consists of other parts, namely the called procedures name and a list of parameters.

Description: OTX shall support statements.

Core_R38– Extension mechanism for statements

Priority: SHALL

Description: It shall be possible to extend the set of statements by new statement types. This shall happen by a well-defined extension mechanism.

Core_R39 – Blocks of statements

Priority: SHALL

Rationale: Situations will often occur where a sequence of statements needs to be grouped together. The sequence of grouped statements builds a block. There may also be blocks in blocks recursively. Blocks are needed for defining the body of a loop, branch case, etc., but also for simply providing a better overview in a test procedure.

Description: It shall be possible to define blocks of statements, recursively.

Core_R40 – Block statement

Priority: SHALL

Description: It shall be possible that blocks of statements can be used as statements themselves.

Core_R41 – Assignment statement

Priority: SHALL

Description: OTX shall support an assignment statement for assigning expression values to variables.

Core_R42 – Call procedure statement

Priority: SHALL

Description: OTX shall support a call statement for calling other OTX test procedures with a list of arguments that correspond to the test procedure parameter list. The call shall be synchronous, i.e. the caller shall wait until the called test procedure returns.

Core_R43 – Branch statement

Priority: SHALL

Description: OTX shall support a branch statement, which allows reacting to different conditions.

Core_R44 – Parallelisation statement

Priority: SHALL

Description: OTX shall support a parallelisation statement, which allows executing two or more blocks at the same time. The sub-sequences shall be embedded within the parallelisation statement.

Core_R45 – Loop statement

Priority: SHALL

Description: OTX shall support a loop statement, which allows executing a block repetitively, as long as a defined condition is met.

Core_R46 – Continue statement

Priority: SHALL

Description: OTX shall support a continue statement. If used within a loop sub-sequence, this shall stop block execution immediately and induce the next iteration. Outside of a loop, it shall have no meaning.

Core_R47 – Break statement

Priority: SHALL

Description: OTX shall support a break statement. If used within a loop block, this shall force the sub-sequence execution to stop immediately. The outer block shall then be continued at the statement right after the broken loop. Outside of a loop, it shall have no meaning.

Core_R48 – Return statement

Priority: SHALL

Description: OTX shall support a return statement. This shall force the execution of the running test procedure to stop immediately and pass control to the caller.

Core_R49 – Exception handler statement

Priority: SHALL

Rationale: When an exception occurs during test procedure execution, it is needed to treat such an exception so that execution may still be continued in a controlled manner. It should be possible to define exception-monitored blocks in the procedure, and also to define blocks dedicated to handle a particular exception type.

Description: OTX shall support a statement for exception handling. The statement shall allow reacting to different exceptions in different ways.

Core_R50 – Throw statement

Priority: SHALL

Rationale: Under certain circumstances it is necessary to throw exceptions intentionally.

Description: There shall be a throw statement that allows throwing an exception of a particular type.

Core_R51 – Validity information for statements

Priority: SHALL

Rationale: Enable/disable statements according to validity information.

Description: There shall be a well-defined means to add validity information to any statement in OTX procedures. The validity information defines under which predefined context conditions the statement is valid. Invalid statements are to be skipped at runtime.

4.8 Boundaries

Core_B01 – No graphical procedure layout information

Priority: SHALL

Description: It shall be possible to represent a procedure graphically without the need for supporting geometric information in the exchange format.

NOTE Meta data can be used to transport such information. It is up to the tool where the information is stored, if needed.

Core_B02 – Performance

Priority: SHALL

Rationale: OTX is solely a data format.

Description: No requirements on OTX performance exist.

Core_B03 – No versioning and configuration management

Priority: SHALL

Rationale: The versioning and configuration of OTX files is a process-dependent aspect of using OTX. OTX shall not define or prescribe a process.

Description: OTX shall not comprise data model aspects or techniques that relate to document versioning or configuration management of OTX files.

Core_B04 – Licensing

Priority: SHALL

Rationale: OTX is solely a data format.

Description: OTX shall not be based on any technology that limits its application and distribution or distribution of files that are OTX-compliant by legal restrictions or licensing cost.

Core_B05 – No explicit memory management

Priority: SHALL

Description: OTX shall not give the possibility to handle memory management, control garbage collectors or other runtime system tasks which are out of the diagnostic scope.

Core_B06 – No exception handling for environment fail

Priority: SHALL

Description: OTX does not need to define exception handling for environment failure. If the runtime environment has problems, test sequence execution shall stop. Only named exceptions can be used at the different actions and terms.

Core_B07 – No global variables

Priority: SHALL

Description: OTX shall not allow variables with global scope.

Core_B08 – No string translation utilities

Priority: SHALL

Description: No string translation utilities are required in the core language. The language is translation-agnostic.

Core_B09 – No unit localization utilities

Priority: SHALL

Description: No unit localization utilities are available in the core language. The core language is unit-agnostic.

5 Introduction to modelling in UML and XSD

5.1 General aspects

The Unified Modelling Language (UML, see Reference [3]) is used to define the OTX data model formally and unambiguously. It enhances readability by graphical data model diagrams. The ability to create different views onto distinct aspects and parts of the overall OTX data model is a great advantage compared to modelling directly by using the XML Schema Definition Language (XSD, see W3C XSD (all parts):2012). In combination with state-of-the-art UML-to-XSD generator technologies, using UML is a powerful modelling method.

A short introduction to UML is given in this clause. For the sake of brevity, only those aspects of UML that are needed for the understanding of the OTX data model are described at this point. This means that, from the large set of available UML diagrams, only the use of class diagrams for OTX modelling and the activity diagrams for specifying OTX behavioural aspects will be exemplified. Special attention will be paid to the use of specific XSD stereotypes which are needed for preparing the UML model precisely towards XSD generation.

5.2 Class diagrams

5.2.1 General

By using UML class diagrams, a set of classes and the structural interdependencies in between those classes can be described graphically. There are two major means of description in class diagrams: classes with properties representing blueprints for instances of that class, and associations depicting distinct relations in between the classes of the model, like, e.g. inheritance or aggregation relations.

5.2.2 Class

The central UML modelling element used for the OTX data model is the class. A class represents a set of similar objects. Generally, a class can be instantiated many times. Every instance of a class is called an object.

A class can contain any number of named, typed “attributes” (defining the properties of these objects) and “methods”[1)] (defining the actions an object can perform). A class may also have a so-called “stereotype”-accretion that indicates the special usage of the class for a particular problem domain.

CAUTION — In UML nomenclature, any kind of class-properties are called “attributes”, whereas in XSD there is the distinction between “attribute” and “element” properties. In order to avoid confusions, the term “attribute” is henceforward to be understood as class attribute in the context of UML diagrams, while in the XSD context it is to be perceived as an XSD type attribute.

Classes can also be “abstract”, this means they cannot be instantiated. In the OTX data model, abstract classes are used to transport a common set of properties to child classes (see 5.2.3 about inheritance relationships).

Figure 1 shows the representation of a class and its attributes in UML notation. A class is symbolized by a rectangle having up to three fields.

— The top field contains the name of the class, e.g. **`Contact`**. For abstract classes, the class name is italic. The stereotype **`«XSDcomplexType»`** denotes that this class shall be mapped to an XSD complex type in the corresponding XML schema (stereotypes for the problem domain XSD).

1) For the static OTX data model, methods are irrelevant and are not used.

— The second field contains the attributes of the class, e.g. the string labelled **`contactId`**, another string labelled **`name`** and an integer labelled **`age`**. The type descriptor is denoted directly after the colon behind the label. The stereotypes **`«XSDelement»`** and **`«XSDattribute»`** specify that **`name`** and **`age`** map to elements, whereas **`contactId`** maps to an attribute (in the XSD domain). Each attribute in the class also carries cardinality information in square brackets, e.g. `[0..1]`, `[1..*]`, `[2]`, etc. The cardinality defines how many instances of the attribute may appear in an instance of the class. If there is no cardinality shown, a default cardinality of `[1]` applies. Furthermore, a default value for the attribute may be specified.

— There is no third field here – in other contexts it is used for methods which are irrelevant for the OTX data model.

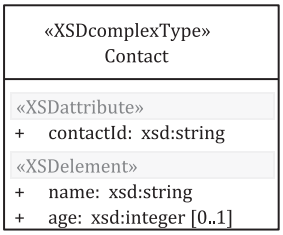

Figure 1 — UML representation of a class

5.2.3 Inheritance relationships

By using inheritance relationships, classes can inherit attributes from other classes.

In Figure 2, a new class **`BusinessContact`** is derived from the class **`Contact`**. This means that implicitly the class **`BusinessContact`** has all the same attributes as **`Contact`** plus those that are defined specifically for the new class **`BusinessContact`**, e.g. the string attribute **`company`**. **`Contact`** is called the parent or super-class; **`BusinessContact`** is called the child or sub-class of the inheritance relationship. Because the sub-class adds more detail to the super-class and is thus more specific, inheritance relationships are often called “specializations”.

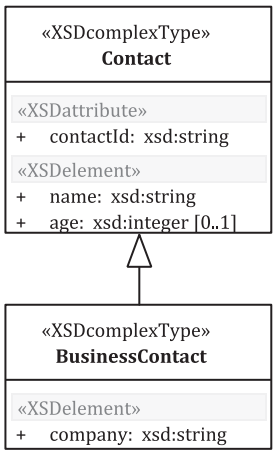

Figure 2 — UML representation of inheritance relationship

Inheritance relationships can be used to build inheritance trees of arbitrary depth. A class in such a tree inherits all attributes from those classes in the transitive closure of all ancestors (parents, grandparents, etc.) in the inheritance tree.

NOTE Concerning the mapping to XSD, it is important to point out that XSD only supports single inheritance. As a consequence, UML models containing classes with more than one direct super-class cannot be mapped correctly. Apart from that, mapping to XSD is straightforward.

UML diagrams usually show certain aspects or fractions of the overall data model. Therefore, diagrams exist where a super-class of another class is not shown. To still reflect the fact that a class is a child of the hidden super-class, a special notation form has been chosen in this document, as presented in Figure 3: an additional property is shown to the upper right of the class `BusinessContact`, denoting its super-class named `Contact`.

Figure 3 — Alternative UML representation of inheritance relationship

5.2.4 Aggregation relationships

Besides the inheritance relationship, a pair of classes may also have an aggregation[2] relationship. Aggregation relationships are used if an object of one class is contained in an object of another class.

An aggregation relationship is drawn as a line with an unfilled diamond at the end of the containing class. In the OTX data model, the relationship end at the contained class always carries a so-called role name. A role can be used to distinguish two objects of the same class when describing its use in the context of the association. The role end also carries a cardinality information, e.g. "`0..1`", "`1..*`", "`2`", etc. The cardinality defines how many instances of the associated class may be contained in an instance of the class.

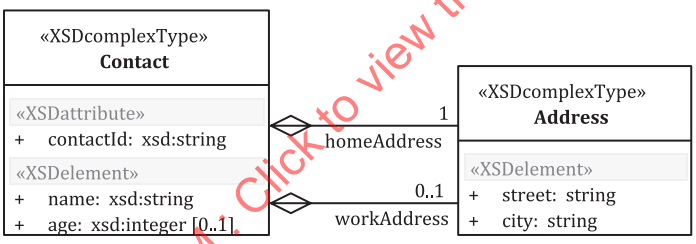

Figure 4 — UML representation of aggregation relationship

Figure 4 gives an example of two classes with aggregation relationships defined. A `Contact` may have two addresses, that is, two objects of type `Address`: one of the addresses plays the role called `homeAddress`, the other plays `workAddress`. The cardinalities are 1 respectively "`0..1`". In other words, this has the following semantics: "A contact has exactly one home address and an optional work address".

5.3 Mapping to the XML Schema Definition language (XSD)

5.3.1 General

The OTX target data model format is an XML Schema Definition (XSD). In this subclause, a very short idea of how the UML model is mapped onto XSD language features is given. It does not claim to be complete, but is only a subsidiary that simplifies the comprehension of the OTX examples given throughout this document.

2) The special case of aggregation, namely *composition*, does not apply in the scope of XSD data model implementation. Composition means that an object may only be contained in exactly one other object. Since in instance XML documents, element snippets can be moved to different places in the document or even in between different XML documents, there is no reason for this restriction. Therefore, only aggregation relationships are used in the OTX data model.

The general rules for mapping are given in the following subclause, they are accompanied by a UML example together with its corresponding XSD output and a valid XML document according to the XSD.

5.3.2 Mapping rules

The following mapping rules apply.

— A class is mapped onto an XSD complex type with XSD sequence content as a default content type [the order of elements in an XSD sequence is significant (!)]. In cases where the stereotype of the class is not «`XSDcomplexType`», analogous rules are applied and explained in the place needed.

— Attributes of a class are mapped onto attributes of the XSD complex type, if the «`XSDattribute`» stereotype is defined for the UML attribute. A cardinality of `[1]` will set the XSD attribute property `use = "required"`; a cardinality of `[0..1]` will result in `use = "optional"`. If there is no cardinality given, the default, `use = "optional"`, applies.

— Attributes of a class are mapped onto sub elements of the XSD complex type, if the «`XSDelement`» stereotype is defined. The cardinality of the attribute is mapped to the cardinality properties of the sub element (XSD element properties `minOccurs`, `maxOccurs`). As an example, a cardinality of `[1..*]` will result in `minOccurs = "1"` and `maxOccurs = "unbounded"`. If there is no cardinality given, the default, `minOccurs = "1"` and `maxOccurs = "1"`, applies.

— Classes connected to another class via an aggregation relationship are mapped onto sub elements in the corresponding XSD complex type of the containing class by the following rules: The role names at the aggregation (contained end) are mapped onto the sub element `name` property. The name of a contained class is the `type` property of the corresponding sub element. The cardinality of an aggregation (contained end) is mapped to the cardinality of the sub element (XSD element properties `minOccurs`, `maxOccurs`).

— An inheritance relationship between a super-class `Super` and a child class `Child` is mapped onto the XSD extension `base` property of the XSD complex type `Child`, e.g. `base = "Super"`. Multiple inheritance (more than one super-class) cannot be mapped correctly to XSD, because XSD allows single inheritance only.

— An abstract marked class is mapped to an XSD complex type with the property `abstract = "true"`.

Throughout the document some other stereotypes are used that have not been explained within this section. These include the following:

— «`XSDsequence`», «`XSDchoice`», «`XSDall`», these are stereotypes for classes representing an XSD sequence, choice or all content-model separately or outside of an XSD complex type;

— «`XSDsimpleType`» to make a class represent a simple content type, e.g. classes derived from string, integer;

— «`XSDtopLevelElement`», this makes a class represent a document root element (only used once in the OTX data model for the `<otx>` element definition itself).

5.3.3 Full mapping example

5.3.3.1 UML model example

Consider the example UML data model “Directory” depicted in Figure 5. The example model is designed for describing the structure of XML documents representing a contact-directory.

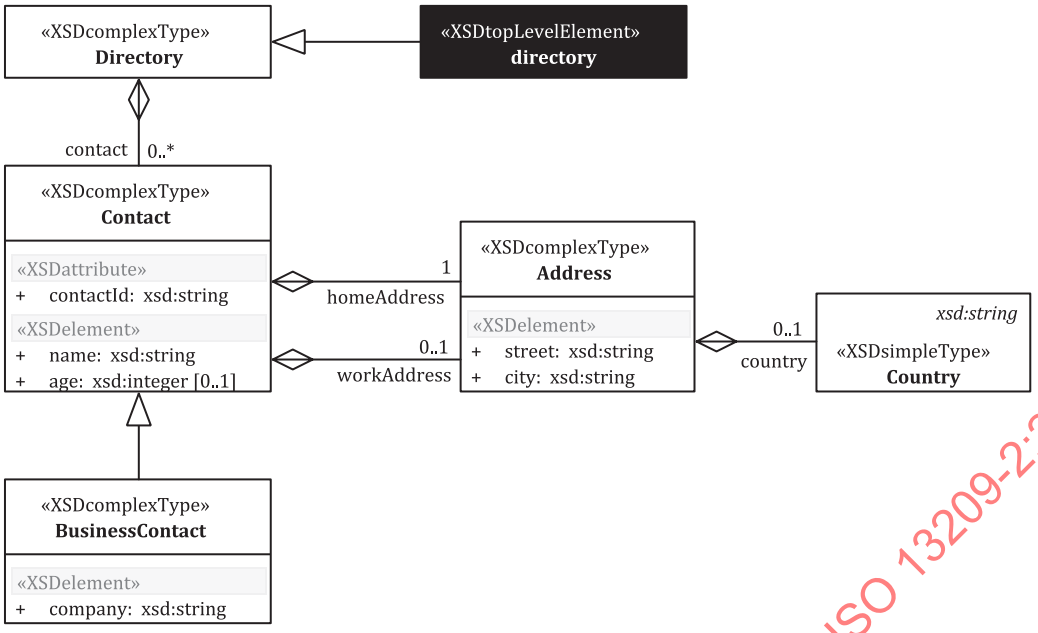

Figure 5 — Mapping example: "Directory" UML model

5.3.3.2 UML to XML translation

The translation of the UML to XML happens like the following:

— The root of every such document shall be the top-level element `<directory>`. Since the corresponding `directory` class is derived from `Directory`, an unbounded list of `<contact>` elements is allowed below `< directory >` .

— The `<contact>` elements are of `Contact` type, so each of them has an attribute `contactId` and the elements `<name>`, `<age>` (optional), `<homeAddress>` and `<workAddress>` (optional).

— Simple type elements like `<name>` (of `xsd:string` type) or `<age>` (of `xsd:integer` type) are described implicitly within the aggregating `Contact` class.

— Features that are complex themselves, like the `<homeAddress>` (of `Address` type), are described explicitly by an own `Address` class, which is connected to `Contact` by an aggregation relation[3)].

— The elements `<homeAddress>` and `<workAddress>` have simple string sub elements `<street>` and `<city>` and also `<country>`.

— In UML, `<country>` is modelled in a special way: it is a simple string and could have been modelled as such, but here the alternative display of inheritance relationships shall be exemplified (see Figure 3).

5.3.3.3 XSD result

According to the mapping rules given above, an XSD model can be derived from the UML model, as shown in the XSD source code below. With the resulting XSD, any XML document claiming to be a contact-directory can be validated for directory schema compliance.

Sample of directory

3) This modelling approach is just a guideline, it is not mandatory – de facto, this could have been modelled alternatively by adding «`XSDelement`» attributes "homeAddress" and "workAddress" directly to the class "Contact". Both approaches would lead to the same XSD/XML result (compare 5.2.2). However, modelling containment relations explicitly with drawn aggregation relations creates a better overview – especially in larger UML diagrams.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="http://example.org/directory"
  xmlns="http://example.org/directory"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">

  <xsd:element name="directory" type="Directory" />

  <xsd:complexType name="Directory">
    <xsd:sequence>
      <xsd:element name="contact" type="Contact" minOccurs="0" maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="Contact">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string" minOccurs="1" maxOccurs="1" />
      <xsd:element name="age" type="xsd:integer" minOccurs="0" maxOccurs="1" />
      <xsd:element name="homeAddress" type="Address" minOccurs="1" maxOccurs="1" />
      <xsd:element name="workAddress" type="Address" minOccurs="0" maxOccurs="1" />
    </xsd:sequence>
    <xsd:attribute name="contactId" use="required" type="xsd:string" />
  </xsd:complexType>

  <xsd:complexType name="Address">
    <xsd:sequence>
      <xsd:element name="street" type="xsd:string" minOccurs="1" maxOccurs="1" />
      <xsd:element name="city" type="xsd:string" minOccurs="1" maxOccurs="1" />
      <xsd:element name="country" type="Country" minOccurs="0" maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:simpleType name="Country">
    <xsd:restriction base="xsd:string" />
  </xsd:simpleType>

  <xsd:complexType name="BusinessContact">
    <xsd:complexContent>
      <xsd:extension base="Contact">
        <xsd:sequence>
          <xsd:element name="company" type="xsd:string" minOccurs="1" maxOccurs="1" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:schema>
```

Note that the elements of the complex type **Directory**, **Contact**, **BusinessContact** and **Address** shown above are contained in **<xsd:sequence>** tags. In XSD, there are two additional complex content types, namely **<xsd:choice>** and **<xsd:all>**. In the UML diagrams, the to-be-generated content type is not shown; the default used by the utilized schema generator is **<xsd:sequence>**. Only in cases where the other content types apply, it will be pointed out especially in the specification.

5.3.3.4 XML instance document

An XML instance document according to the XSD is given below.

Sample showing some directory entries

```
<?xml version="1.0" encoding="UTF-8"?>
<directory
  xmlns="http://example.org/directory"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://example.org/directory directory.xsd">

  <contact contactId="contact1">
    <name>Antje Vanderdam</name>
    <age>27</age>
    <homeAddress>
      <street>Waterkant 23</street>
      <city>Rotterdam</city>
      <country>Netherlands</country>
    </homeAddress>
  </contact>

  <contact contactId="contact2" xsi:type="BusinessContact">
    <name>Peter Piper</name>
    <homeAddress>
      <street>Broad St. 34</street>
      <city>Springfield</city>
      <country>USA</country>
    </homeAddress>
    <workAddress>
      <street>Schema Ave 42</street>
      <city>Validationtown</city>
    </workAddress>
    <company>OTX-Works</company>
  </contact>
</directory>
```

Special attention shall be paid to the **xsi:type** attribute. This is needed in places where the underlying complex type for an element can be one out of the set of child types. The type chosen is stated by this attribute. Compare this to the example XML document above, where a **Contact** is cast to the child class **BusinessContact**.

The OTX data model relies on the **xsi:type** attribute wherever prospective extensions to the data model are to be expected, whose names are not known at the time of core data model creation.

6 OTX principles

6.1 General

The OTX represents a high-level domain-specific language which is especially designed for graphical notation and editing.

The syntax of the language is defined by the UML data model and the corresponding XML schema. The semantics are defined by this document. The general principles followed by the OTX are described here.

6.2 XML format

OTX documents shall be represented as XML as described in the W3C XML recommendation (W3C XML: 2008). Using XML in combination with XSD enables OTX to benefit from the whole strength of XML, with its great number of off-the-shelf solutions, its world-wide acceptance and adoption. XML is the file format of choice also because:

— XML is designed to represent hierarchically structured information, which applies especially for test sequences;

— the basic XML rules for well-formedness eliminate the need for defining an own set of such basic grammar rules specific for OTX;

— checking XML documents for well-formedness is a well-defined task for which a great many number of solutions exist off-the-shelf; thus, there is no need to implement well-formedness-checkers specifically for OTX;

— by modelling the detail correlations of the OTX grammar by using XSD, validity checking can be done by off-the-shelf solutions;

— the interpretation of OTX documents in authoring-, checker- and runtime-applications is assisted by off-the-shelf XML parser frameworks; many of them are schema-driven. This prevents implementations of specific low-level OTX parsers.

IMPORTANT — To ensure the exchangeability an OTX document shall be valid, against the OTX xsd!.

6.3 Imperative and structured programming paradigm

OTX follows the paradigm of imperative, structured programming:

— **imperative**: OTX procedures define sequences of commands for the computer to perform;

— s**tructured**: there are no explicit jumps allowed in the OTX. Instead, branches, loops, exception handlers, return, continue, break and throw statements as well as procedure calls define jumps implicitly and in a **controlled manner**. Therefore, the OTX also defines a block structure, as typically found in all languages that follow the structured programming paradigm.

6.4 Graphical authoring of OTX sequences

In contrast to other programming languages where programs are usually edited in text-based source code editors, OTX aims at sequence editing using graphical authoring tools. This keeps away the cumbersome and error-prone line-by-line source code editing work from the sequence author, a fact that allows XML to be the preferred file format for OTX—even though XML is often denounced to be too verbose. The complexity of managing the OTX XML code shall happen in the background, hidden from the sequence author. On the other hand, the OTX format supports features needed by graphical authoring tools, which are broadly disregarded by source-code edited languages, e.g. explicit specification and realisation compartments in statements, or floating comments that can link to more than one statement, just to mention a few.

6.5 Specification/realisation concept

The OTX format offers explicit support for a three-stage development of test sequences.

— **Specification stage:** a test sequence may be described at specification level only. This is helpful for early stages in the test sequence development process, when not all of the details for creating an executable test sequence are known. The specification stage allows the development of test sequences at a prose level, which is human readable, but not fully executable—i.e. the overall sequence logic may already be there, but single steps in the sequence are filled by prose only, so a runtime interpreter cannot make use out of it. However, starting from this stage, the test sequence can be continuously broken down by the following stages.

— **Intermediate stage:** OTX also allows intermediate stage test sequences, which occur while transforming a specification stage test sequence towards a realisation stage sequence. Such an intermediate stage test sequence may contain parts that are already fully executable, whereas other parts are still at specification stage. The sequence author uses the human readable information from the specification stage parts and implements those parts following the specification instructions by adding realisation counterparts. At any step in this process, the sequence is validly saveable and exchangeable.

— **Realisation stage:** At this stage, there are no more specification-only parts in the sequence. The sequence is then fully executable. Note that even at this stage, specification parts of the sequence may still be contained, but not without their corresponding realisation. With other words, this coexistence represents an “executable specification”.

6.6 Modular OTX extension concept and OTX-based runtime architecture

The OTX data model is structured into a language core and various functional extensions. It is important to note that the OTX core, as well as the extensions defined by this document, does only

provide syntactical definitions for the elements of test sequences and a functional description of the expected behaviour. The implementation of the runtime behaviour itself is not part of the OTX standard. This is necessary to allow for the needed freedom when implementing OTX runtime components to fit specific use cases. For example, it would be impractical for the OTX standard to define the specific implementation of an HMI library, as the requirements and backend frameworks for HMI functionality will differ for each test environment.

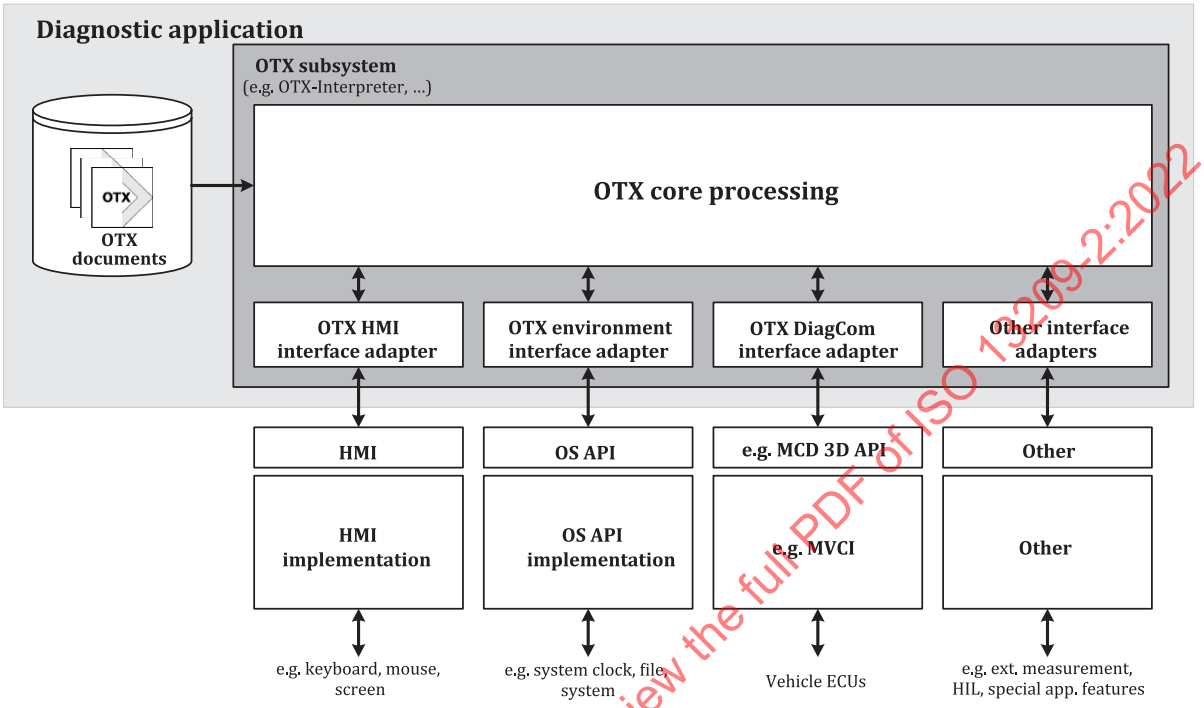

Figure 6 — OTX-based runtime architecture

Figure 6 comprises an overview of the standard extensions specified in ISO 13209-3[2]. The following specification of the OTX core data model touches the extensions only by defining extension interfaces which serve as hooks for the respective extensions. Otherwise, the core is fully stand-alone.

This separation of OTX structure (intent) and runtime implementation (execution) will allow for OTX to be used as a test sequence exchange format that can be deployed throughout an entire organization, or even across company boundaries. Figure 6 illustrates the architecture of an OTX runtime diagnostic application. The OTX subsystem that is part of the diagnostic application is interpreting OTX sequences which make use of OTX core data model constructs, as well as of various extension interfaces (HMI, environment-specifics, vehicle communications and other proprietary functionalities).

The actual runtime system implementation knows how to map, e.g. an HMI extension action within an OTX sequence to the GUI framework that is used by the diagnostic application. For example, an after-sales application might map an OTX 'ChoiceDialog' construct to a Java Swing dialog entity that allows interaction with the workshop mechanic. The same OTX sequence could also be executed on a manufacturing station, where the diagnostic application implementation would map the 'ChoiceDialog' construct to a state machine connected to manufacturing line measurement equipment instead of a GUI interface.

6.7 Context concept

Diagnostic test sequences require a technique for getting access to a diversity of contextual information. Such information can be, e.g. vehicle related, diagnostic application related, user related or application area related. A few instances of contextual information are mentioned here.

— Vehicle related: data about the currently diagnosed vehicle's model, vendor, identification number, engine-type or other identification data concerning the physical and electronical vehicle configuration.

— Diagnostic application related: application version, name, used vehicle communication interface type or version or also diverse settings of the application.

— User related: login name, idle time, user access rights, etc.

— Application area related: information about the location where the test sequences are applied, to make the difference, e.g. between manufacturing, engineering or workshop appliance.

In general, contextual information may be required by heterogeneous subsystems of a diagnostic application. Concerning OTX, only the provision of contextual information to the OTX subsystem is considered.

OTX does not prescribe a set of context items which shall be supplied by every conforming application. Also, OTX does not make any assumptions or rules about the method by which a diagnostic application should identify particular context items (such methods are henceforth called identification routines). Instead, OTX only specifies how context-dependent OTX documents shall declare the particular context items which they require. It is the task of the diagnostic application to connect each context declaration of an OTX document to the specific identification routines yielding the respective context item.

Figure 7 shows a conceptional example of a mapping between context declarations defined by an OTX document and identification routines of a diagnostic application. The implementation of the mapping itself is not specified by this document. Diagnostic applications are free to implement an appropriate mapping mechanism. Furthermore, the identification routines can be implemented in any programming language appropriate for the respective diagnostic applications. Therefore, it is also possible to use OTX again for implementing identification routines. Diagnostic applications should also provide a mechanism which allows for OTX authors to integrate context dependent OTX documents into the application by adjusting the mapping and adding new identification routines.

From the perspective of an OTX test sequence, contextual information is treated like static information. Contextual information cannot simply be set by an OTX test sequence—if this would be possible, a test sequence would be able to set, for example, the engine-type information of the currently diagnosed vehicle to “diesel” despite the fact that the concrete vehicle is gasoline driven. Consequently, OTX test sequences shall only be able to read contextual information by indirectly utilizing the identification routines provided by the diagnostic application.

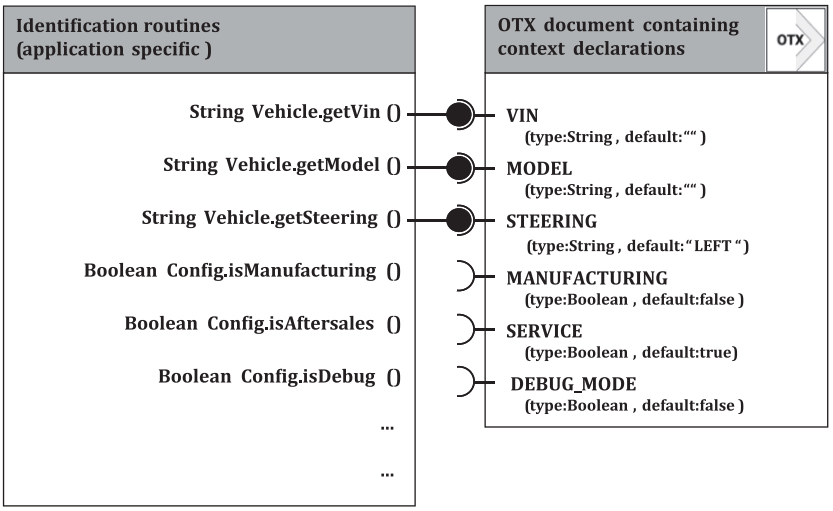

Figure 7 — Mapping context declarations to identification routines

The OTX context concept has several advantages.

— Transparency: working with context data is almost similar to working with global constants for OTX authors.

— Compatibility: integrating OTX into existing diagnostic applications does not require a change of the original context identification/management strategy of the application or the diagnostic session management, etc.

— Exchangeability: since OTX documents declare each context item they require explicitly and in a well-defined place, using an OTX sequence originating from a foreign diagnostic application only requires an adjustment of the mapping between context declarations and identification routines (and possibly the implementation of new identification routines). In this way a context dependent OTX document can be "docked" to heterogeneous diagnostic applications.

— Cooperation: by agreeing upon an OTX-file or a set of OTX-files declaring commonly used context data, two or more parties can improve their cooperation concerning OTX test sequence development.

— Simulation: various context situations can be simulated by, e.g. feeding simulated data to identification routines.

— Aside from the above topics, the context concept is crucial for the validities concept which is described in 6.8.

6.8 Validities concept

Based on the context concept described above, OTX offers the validities concept which allows for configuring test sequences for varying runtime contexts. The behaviour of such a configured test sequence changes according to the context it is running in. An OTX author can use this to configure and prepare generic purpose test sequences for different scenarios. At runtime, the test sequence adapts its behaviour according to the configuration.

To achieve this ability, an OTX author first needs to define validities: a validity encapsulates a Boolean term. The OTX author may use the declared validities for marking parts of his test sequence to be valid only if a connected validity is true at runtime. An invalid part will not be executed at runtime—like this, the author can activate and deactivate parts of the sequence in a context-driven manner.

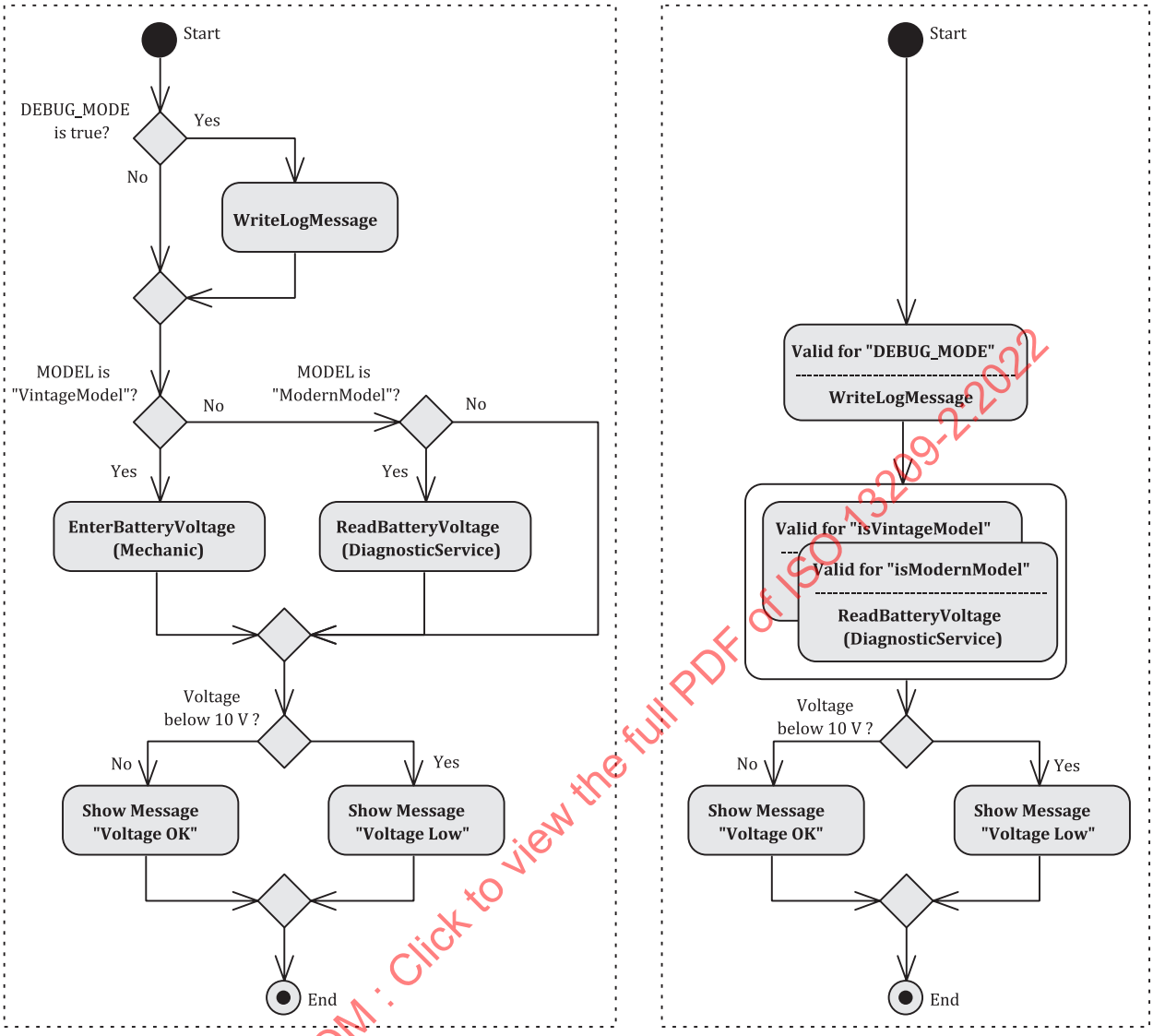

Figure 8 — Validities concept, example of use

Consider the example test sequence to the left of Figure 8. This sequence does not use validities. It is designed to do the following.

a) If the diagnostic application's runtime system is in debug mode, it shall write a log message.

b) The battery voltage of a vehicle shall be measured. According to the type of vehicle under test (a vintage model, a modern model or neither of these, e.g. a bicycle), this shall happen in different ways:

   1) vintage model: the voltage shall be measured and entered into the system manually.

   2) modern model: the voltage can be queried automatically (by using a diagnostic service).

   3) it is some other type of vehicle with no battery: nothing can be measured.

c) If the measured voltage is too low, a message “voltage low” shall be shown, otherwise “voltage OK”.

The test sequence to the right of Figure 8 does (superficially) the same, but it uses validities in every place where context-dependent decisions are taken.

a) The step “WriteLogMessage” is marked to be valid for “DEBUG_MODE” only. It will be inactive if the Boolean value of the context item “DEBUG_MODE” is `false`.

b) The step for measuring the voltage comes in two different flavours: one is valid only for "isVintageModel", the other only for "isModernModel". If none is valid, no action is specified (there is some other type of vehicle). The validities "isVintageModel" and "isModernModel" are validity terms analysing the current value of the "MODEL" context item.

c) The voltage value is dynamic: it is neither predictable at authoring time nor depending on the context. It is only known at runtime. It is not a configuration. That is why validities are not used here.

The context items "MODEL" and "DEBUG_MODE", and the validity terms "isModernModel" and "isVintageModel" can be defined, for example, in a central document and be reused by all OTX authors who want to configure their test sequences towards the same context situations.

The advantages of this approach are the following.

— A distinction is drawn between decisions based on static context data (e.g. enumerations of expected context values known at authoring time) and dynamic data (which is computed at runtime).

— Validity information controls the flow implicitly regarding context, not explicitly by normal dynamic branch conditions. This fact is also reflected in the more compact way of representation, the "pure" test sequence logic becomes more apparent.

— Sets of commonly used validities can be stored separately, in a central place. This prevents a lot of redundancy, since authors can reuse preconfigured validities for their test sequences. This also improves maintainability.

— Filtering: OTX authoring system may allow the author to configure certain context situations in a simulated environment. This produces very compact views of test sequences for a given context (mask steps that are not valid).

— Filtered test sequences may be extracted and/or exported. This results in test sequences which are tailored for a specific context.

— Enables the signature concept which is described later in this document.

Figure 9 shows the filtering of test sequences according to simulated contexts. In the left context, debug mode is on, the model is a vintage car. In the right context, debug mode is off, the model is a modern car.

The example is of course naïve, it serves only for comprehension. In real world test sequences, the validity concept can be used for various purposes, for example, to configure sequences for different vehicle type series, ECU variants, vehicle configuration codes, different operating systems, different application areas like in a repair workshop or an assembly line, etc.

Especially concerning graphical OTX authoring environments, the validity concept supports the compact and at the same time flexible representation of test sequences.

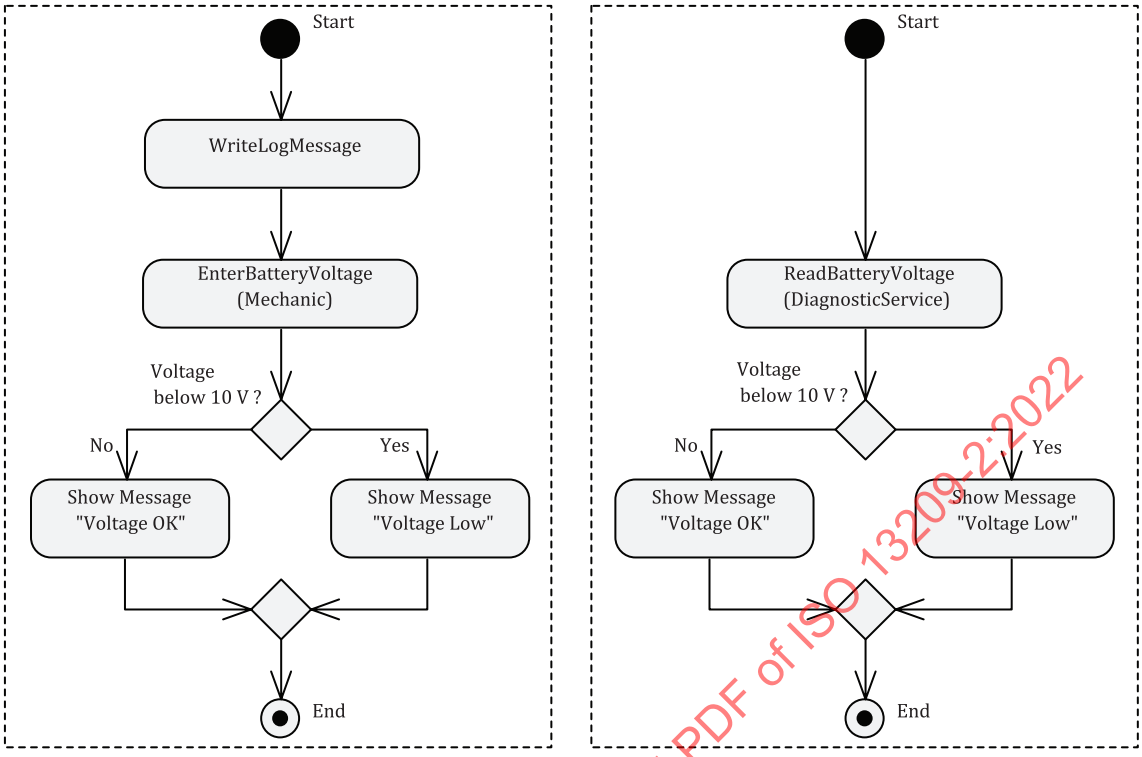

Figure 9 — Validities concept, filtering

6.9 Signature concept

The OTX signature concept is designed to support dynamic linking.

The concept allows for defining prototype procedures called signatures. A signature is like a procedure but without programme content (implementation); it consists only of a name, a set of parameter declarations and a prose specification. Procedures that implement a signature contain the same set of parameters like the signature and they have to implement a programme that accomplishes the task that was specified in prose in the signature. Hence, a signature defines an interface which implementing procedures have to obey. If this precondition is met, implementing procedures can be called indirectly via a signature – the caller only needs to know the parameters and the specification of the signature, the caller does not need to know implementation details which are hidden in the implementing procedures (in simple words: "I want you to accomplish this, but I don't care **how** you accomplish it").

Signatures are especially suited for cases where a task has to be carried out differently depending on the context, but superficially it is still the same task. The variant in which the task is carried out is chosen dynamically at runtime when the context becomes known. At authoring time, this is left open. This allows writing generic test sequences that do not need to be changed as long as used interfaces (the called signatures) do not change.

The signature concept is closely linked to the validities concept (see above). Consider the example in Figure 10. The generic procedure "Voltage_Test" needs to read the current battery voltage of a vehicle. In the example, there are two kinds of vehicles: a vintage model where the voltage needs to be measured manually by a mechanic and a modern model where the voltage can be read automatically by electronic test equipment. Since the author of the generic procedure does not know how to measure the voltage in the different cases, she/he leaves the work up to colleagues who implement the signature "ReadBatteryVoltage" that she/he provides. The output are two procedures, "Manual_ReadVoltage" which works for the vintage model, and "Auto_ReadVoltage" which works for the modern model. Both procedures obey to the signature, they do the same job (returning the voltage) by different means.

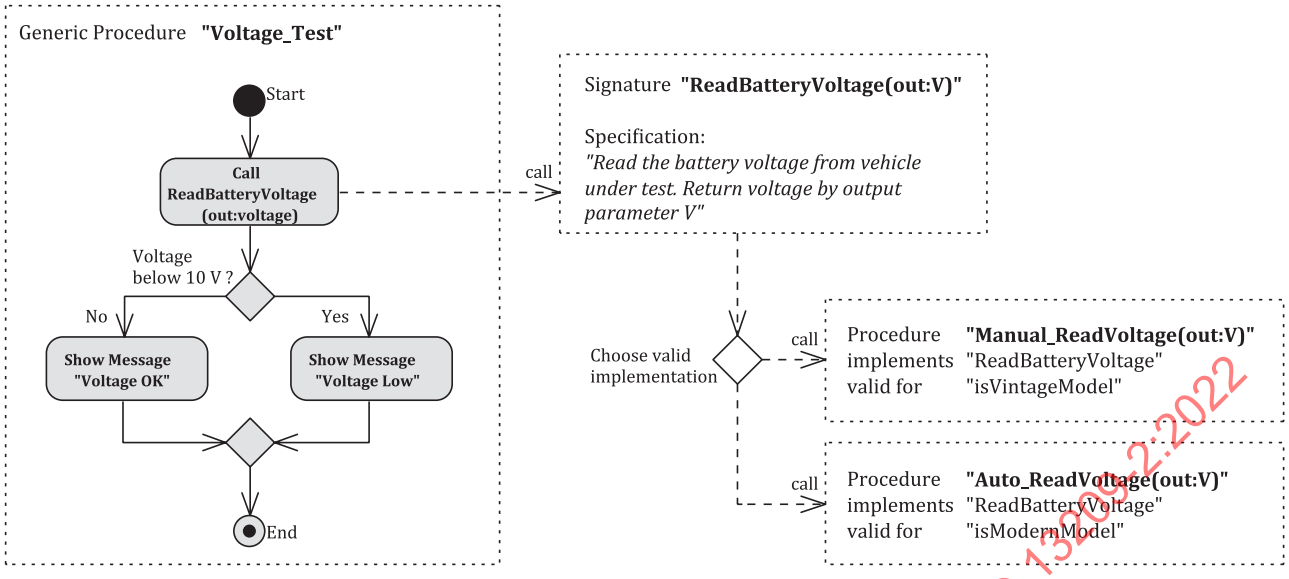

Figure 10 — Signatures concept

At runtime, the procedure call is redirected dynamically to the implementing procedure which is valid with respect to the current context (validities “isVintageModel” or “isModernModel”).

The advantage of this approach is that the generic procedure is stable; it does not have to be changed even if there is a new context situation to be considered (e.g. a new high voltage battery which requires a third variant of voltage reading). This improves maintainability and long-time availability of generic test sequences. Also, this supports cases where test sequence modules are developed by more than one party; the signature concept is a way to define clear and formal interfaces between these distributed parties.

Since the signature concept is dynamic (the effective procedure that will be carried out is in general not known until runtime), special arrangements in the OTX runtime application are required. The indirection via the signature to the called procedure has to be resolved. For this task a mapping is needed which allows looking up a valid procedure implementing a given signature. Details on this topic are specified in 7.14.5 concerning the `ProcedureCall` action.

7 OTX core data model specification

7.1 General

The following represents the full data model specification of the OTX core, data model version `"1.0.0"`.

For each OTX feature, syntax and semantic definitions are provided. The syntax definition will specify the exact XML structure of the feature, whereas the semantics definition specifies how the feature shall be interpreted (by OTX runtime systems or graphical OTX authoring tools).

The specification aims at describing the data model in a hierarchical way, following the document structure from top to bottom, starting at the `<otx>` root element. Commonly used features that appear in more than one place in the structure are described in 7.16.

The OTX core data model specification is accompanied by additional information given in the annexes.

— Annex A specifies the OTX data types and shall be followed.

— Annex B specifies the runtime behaviour concerning scope and memory allocation of OTX constants and variables and shall be followed.

— Annex C contains a comprehensive listing of all checker rules and shall be followed. The rules are needed because some constraints existing on OTX documents cannot be ensured by XSD validation alone. These constraints need to be checked by additional checker applications.

— Annex D describes the extenson mechansim of the OTX core data model and shall be followed. By the creation of OTX extensions new features can be added to the format. Nevertheless, there are well-defined extension rules which shall be adhered by every correct OTX extension.

— Annex E describes the special schema annotations for exceptions and shall be followed.

— Annex F contains the definition of the OTX home directory and is recommended.

7.2 High-level overview of the OTX core data model

Figure 11 shows a high-level overview of the OTX core data model. It only contains a subset of types, attributes and relationships of the overall model; nonetheless it reflects the idea of the essential structure of OTX documents. Details are given in later clauses; this is a quick walk-through.

OTX documents contain header information like the document name, metadata or imports and other global scope information like global declarations and validities followed by a list of so-called OTX procedures and a list of procedure signatures (for brevity, not all of these properties are shown in the diagram).

Validities are named Boolean terms which can be used for configuring context-dependant procedure behaviour.

A procedure itself contains a procedure name, a specification and a realisation section needed for the concept described in 6.5. The realisation of the procedure contains parameter and local variable declaration blocks as well as a flow of nodes representing the procedure logic. It may also contain a list of comments that may be linked to various places in the flow.

In a procedure parameters declaration block, in-, out- and inout-parameters can be declared (not shown in the diagram). Local variables and constants are declared in the declarations block (not shown in the diagram). Each declaration (parameter, constant or variable) has a data type chosen from a list of OTX data types. Semantically, the declared identifiers are visible to all nodes in the procedure flow.

The heart of every procedure is represented by its flow element. It contains a list of nodes that shall be executed sequentially. There are different kinds of nodes: simple nodes representing single commands to be carried out (`Action`, `Return`, `Continue`, `Break`, `Throw` and `TerminateLanes`) and compound nodes that may themselves embed flows (`Group`, `Loop`, `Branch`, `Parallel`, `MutexGroup` and `Handler`). OTX flows are comparable to the block concept as used by other programming languages.

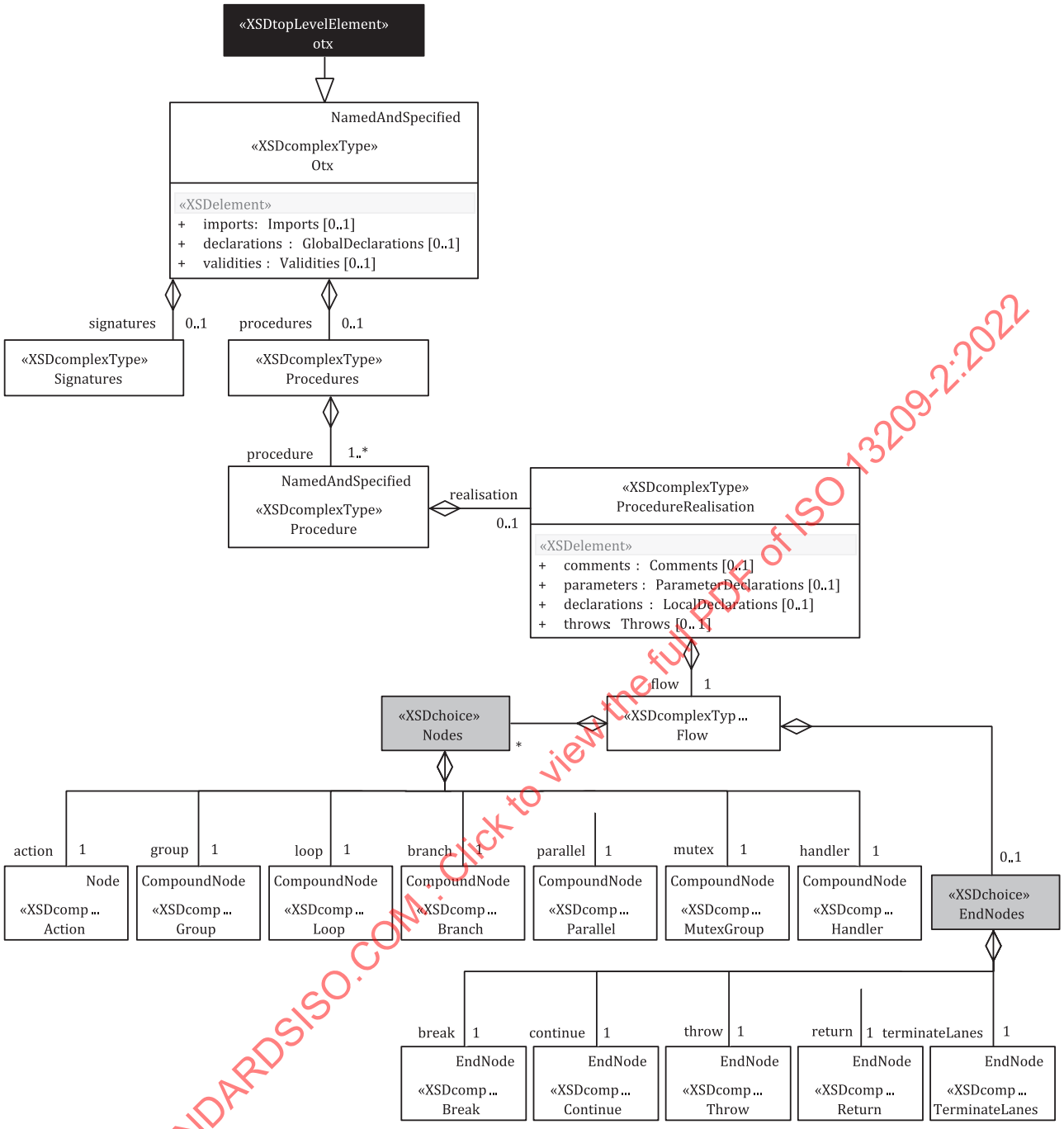

Figure 11 — High-level overview of the OTX core data model

7.3 Document root

7.3.1 Description

This element stands for the document root of OTX documents. It serves as the “entry point” for any OTX parsing application and constitutes a kind of envelope around the test sequence logic itself.

It mainly comprises header information for packaging, naming and versioning of an OTX document and about the links to other OTX documents (import information). It is also the parent for all global entities defined in a document, namely global constants, document variables, contexts, validities, signatures and procedures.

7.3.2 Syntax

Every OTX document starts with the document root element **<otx>** which is derived from the complex type **Otx**. The syntax is shown in Figure 12.

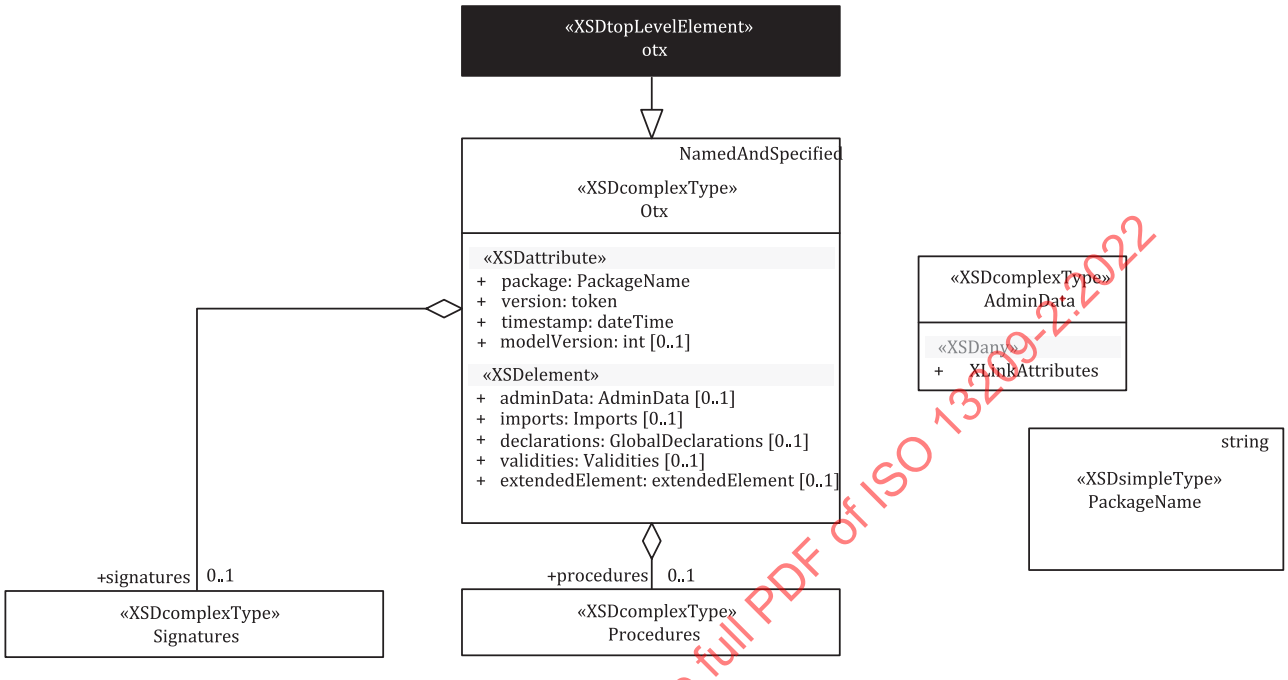

Figure 12 — Data model view: OTXdocument root

7.3.3 Semantics

The **Otx** type is derived from **NamedAndSpecified** (see 7.16.4). It contains attributes concerning the overall document in general:

— **id: OtxId [1]** (derived from **NamedAndSpecified**, see 7.16.4)

This represents the **<otx>** element's id. It shall be unique among all other ids in a document. Please refer to 7.16.4 for details concerning ids in OTX documents.

— **name: OtxName [1]** (derived from **NamedAndSpecified**, see 7.16.4)

This attribute shall contain the name of the OTX document. The name shall match the file name, which simplifies finding an OTX document by name. Among all the OTX documents defined within the same package, the document name shall be unique in order to avoid ambiguities. This is crucial for locating OTX documents, e.g. when calling a procedure defined in another OTX document (see 7.9 and 7.14.5).

Associated checker rules:

— Core_Chk001 – document name matches file name (see C.2.1);

— Core_Chk002 – package-wide uniqueness of document names (see C.2.2).

— **package: PackageName [1]**

This attribute shall represent the package which the OTX document belongs to.

The **PackageName** type is a pattern-restriction of the **xsd:string** simple type. The value space of the attribute is restricted by a regular expression which enforces a dotted notation of package names, e.g. **"com.myCompany.myOtxDocuments"** (see 7.16.2 for the **PackageName** type).

Recommendations:

— OTX documents can be stored in a file system or in a database. If stored in a file system, certain constraints on the organization of files and folders allow a simple implementation to find OTX documents easily. The same applies to the uniqueness of file names, which is ensured automatically by the file system if the OTX-files reside within a folder dedicated to their common package. The package name (in dotted notation) can simply be translated into a directory path or vice versa, whereas the OTX document name can be translated directly to the file name. This practice is strongly recommended. It is an analogy to the file system organization of packages and classes recommended by Reference [1], subclause 7.2.

— For the transport of larger OTX projects consisting out of numerous interlinked OTX-files and packages, the usage of the PDX archive format specified by the ISO 22901 series is strongly recommended. In the case of an MVCI[4]/ODX (see the ISO 22901 series) based system, this allows transporting OTX data along with the ODX data required by the OTX sequences.

— The file name extension should be *.ptx. The following values are predefined for CATEGORY:

  — OTX-MAIN for the main OTX document which is the entry point for the runtime execution. The element is optional. No more than one OTX document in the PDX can have the OTX-MAIN category. The referenced OTX document shall have a main procedure.

  — OTX-DATA for all other OTX documents.

— The file extension for OTX documents should be *.otx .

— The attribute F-XSD-VERSION shall contain the OTX version instead of ODX-version.

— The index xml can have an optional version number of the PDX file directly after the prolog in the format of a processing instruction. The format of the version is unspecified.

  Sample: < ?PTX version = ”1.2.3.4”? >

— Alternatively (if the latter recommendation is not applicable), the Java JAR file format is recommended. It represents a simple and popular archiving method. It is, like PDX, based on the ZIP archive format. Therefore, both approaches PDX and JAR could even be combined in one project file.

IMPORTANT — OTX documents are unambiguously identified with the combination of package-name and their own name. This allows for having equal document names in different OTX packages and avoiding ambiguity.

IMPORTANT — Global entities (global constants, procedures, validities, etc.) contained in the OTX document are unambiguously identified with the combination of package- and document-name, and their name. This allows for having equal global names in different OTX files and avoiding ambiguity.

— **`xmlns[:prefix]: anyURI [0..1]`** (standard XML attribute)

The special purpose attribute **`xmlns`** is specified in W3C XMLNS:2009. It defines the XML schema namespace to which an XML document complies to. The information is used by XSD validation for testing document compliance to corresponding schema(s).

With regards to OTX, the attribute shall associate the XSD namespace “**`http://iso.org/OTX/1.0.0`**” to the **`<otx>`** document root element, which can then be tested by validation for compliance to the OTX XSD.

NOTE The OTX data model version (**`1.0.0`**) to which this specification adheres is included in the OTX namespace URI “**`http://iso.org/OTX/1.0.0`**”. For future versions of the standard the version number in the URI will be adapted accordingly. With this technique, the correct XSD version to be used for validation of an OTX document can be identified unambiguously.

If any OTX extension features are used in the document, the XSD namespaces of the respective extension schema shall be associated by using the `xmlns` attribute as well (extension namespaces are defined by the schemas specified by ISO 13209-3 [2]).

If the `xmime:contentType` attribute is used in `<metaData>` elements of an OTX document, the namespace "`http://www.w3.org/2005/05/xmlmime`" shall be associated by `xmlns` also.

For making the the `xsi:type` and the `xsi:schemaLocation` attributes available in OTX documents, the "XML Schema instance" namespace "`http://www.w3.org/2001/XMLSchema-instance`" shall be associated W3C XSD (all parts):2012.

Overview of common XML namespace associations used for OTX:

— `xmlns = "http://iso.org/OTX/1.0.0"`

— `xmlns:diag = "http://iso.org/OTX/1.0.0/DiagCom"`

— `xmlns:hmi = "http://iso.org/OTX/1.0.0/HMI"`

— `xmlns:myext = "(custom OTX extension namespace)"`

— `xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"`

— `xmlns:xmime = "http://www.w3.org/2005/05/xmlmime"`

IMPORTANT — The prefixes used as abbreviations for the lengthy namespace qualifiers can be chosen freely. Each prefix provided shall be unique in the scope of the document (this is enforced automatically by XSD validation).

— `version: xsd:token [1]`

This attribute shall contain the version of the OTX document (for supporting versioning systems).

NOTE 2 OTX does not prescribe any rules for versioning. It just defines a location where simple versioning information can be put, if needed. For more complex versioning tasks, the metadata elements can be used.

— `timestamp: xsd:dateTime [1]`

This attribute shall contain the document creation timestamp. Further information regarding document history is also described by the `<adminData>` element, see below.

— `modelVersion: xsd:int [0..1]`

This optional attribute shall contain the model version of the OTX document. The only valid value is 2022. It means that the model version is ISO 13209-2:2022. For backwards compatibility the attribute can be omitted. In this case the model version is 2012 or ISO 13209-2:2012.

IMPORTANT — New versions of the core data model shall be backward compatible. Thus, every OTX document that was valid in earlier OTX versions (ISO 13209-2:2012) must stay valid in the upcoming OTX version, without any need for manual or automatic migration.

— `<metaData> : MetaData [0..1]` (derived from `NamedAndSpecified`, see 7.16.4)

This optional element shall contain meta information concerning the OTX document as a whole. For a description of `MetaData` refer to 7.16.5.

— `<specification> : xsd:string [0..1]` (derived from `NamedAndSpecified`, see 7.16.4)

The static string content of this optional element shall be used to describe the purpose of the OTX document in simple prose.

— **<adminData>: AdminData [0..1]**

This optional element shall be used for linking to document history information via XLink attributes, as specified by W3C XLink:2010. The history information shall be structured according to the **ADMINDATA** element as specified in the ISO 22901 series. The linked information should be located in the top-level **<metaData>** element of the OTX document (see above), but it may also be stored externally.

— **<imports>: Imports [0..1]**

Contains a list of **<import>** elements for importing external OTX documents (see 7.4).

— **<declarations>: GlobalDeclarations [0..1]**

Element representing the global declaration block. This is the place where global constants shall be defined. Except for the global scope, the semantics are equivalent to locally declared procedure constants. The type is not further specified here, see 7.5 for details.

— **<validities>: Validities [0..1]**

A list of global scope validity terms shall be defined here (see 7.6).

— **<signatures> : Signatures [0..1]**

This optional element shall contain a list of **<signature>** elements. The complex type **Signature** is specified in 7.7.

— **<procedures> : Procedures [0..1]**

This optional element shall contain a list of **<procedure>** elements. The complex type **Procedure** is specified in 7.9.

— **<extendedElement> : ExtensibleElement [0..*]**

Declares a general element which can be extented by new general functionality defined in new OTX extensions using the standardised extension mechanism, example: test cases for unit tests.

IMPORTANT — The extension mechanism described here only defines the syntactic rules which extension implementers shall follow in order to conform to ISO 13209-2. It shall be ensured that no OTX requirements are violated.

7.3.4 Example

The example below contains a skeletal OTX document. The example does not use the **<adminData>**, **<imports>**, **<metadata>**, **<validities>**, **<procedures>** or **<signatures>** elements, but it contains a **<specification>** element. Examples for the missing elements are given in following sub clauses.

Sample for a DocumentRootExample

```
<?xml version="1.0" encoding="UTF-8"?>
<otx xmlns="http://iso.org/OTX/1.0.0" id="1"
  name="DocumentRootExample"
  package="org.iso.otx.examples"
  version="1.0"
  timestamp="2010-03-18T14:40:10"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://iso.org/OTX/1.0.0 Core/otx.xsd" >

  <specification>Example for showing the OTX document root structure</specification>
</otx>
```

NOTE The **xsi:schemaLocation** attribute is only a local, in-document hint for XML parsers (validators) which maps each used XML namespace to an absolute or relative path (URL) where the corresponding XSD files are located. XML parsers can use alternative techniques for resolving corresponding XSD files, e.g. an XML catalogue. The **xsi:schemaLocation** attribute is optional and will therefore not appear in all further OTX examples.

Following the recommendation for OTX packages given above, the full file system path to the example file is `"[OTX Base Directory]/org/iso/otx/examples/DocumentRootExample.otx"`. Here, the chosen format for the `package` value allows a simple translation from logical namespace to physical location of the OTX document. By contrast, in a database the namespace and the document name can be used as primary keys to access the document.

7.4 Imports

7.4.1 Description

The top-level `<imports>` section contains a list of `<import>` elements. They are used by an OTX document to import global names defined in other OTX documents. Imported names are all visible procedure-, signature- and validity-identifiers as well as all visible identifiers declared in the global declaration block. They can be referenced from the importing document by using `OtxLink` type attributes, as will be specified later.

7.4.2 Syntax

Figure 13 shows the syntax of the `Imports` type.

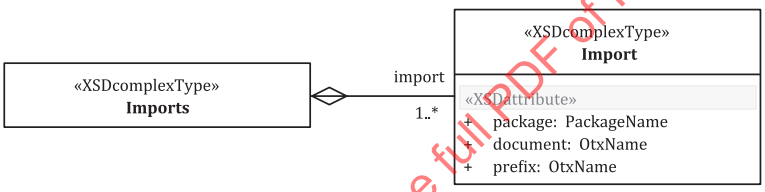

Figure 13 — Data model view: `Imports`

7.4.3 Semantics

The `<import>` element has the following semantics:

— `package: PackageName [1]`

Contains the package which the imported document belongs to (see 7.16.2 for the `PackageName` type).

— `document: OtxName [1]`

Contains the document name of the document which shall be imported (see 7.16.3 for the `OtxName` type).

— `prefix: OtxName [1]`

Defines a prefix which serves as an alias for the imported document. It shall be used in all places where an imported name from the imported document is accessed. For accessing, the counterpart of `OtxName` called `OtxLink` is used (see 7.16.3 for the `OtxName` and `OtxLink` types).

The prefix name can be chosen freely insofar as it complies with the syntax enforced by `OtxName`. Another restriction is that a prefix shall be unique among all other import prefixes defined in the same document (this is enforced by an `xsd:key` restriction in the OTX schema).

Associated checker rules:

— Core_Chk003 – no dead import links (see C.2.3);

— Core_Chk004 – no unused imports (see C.2.4);

— Core_Chk006 – match of imported document's data model version (sse C.2.6).

7.4.4 Example

The example below shows an OTX document importing two other OTX documents. The first imported document named **"Signatures"** of package **"org.iso.otx.examples"** gets the prefix **"sig"**. The other document named **"Validities"** of the same package gets the prefix **"val"**. The use of the prefixes is shown in the procedure **"test"**, which refers to a signature **"sig:mySignature"** and to a validity **"val:myValidity"**. Both identifiers **"mySignature"** and **"myValidity"** shall be declared in the document identified by the respective prefixes (for the sake of brevity, imported documents are not shown here).

Sample of imports

```
<?xml version="1.0" encoding="UTF-8"?>
<otx xmlns="http://iso.org/OTX/1.0.0" id="2"
  name="ImportsExample"
  package="org.iso.otx.examples"
  version="1.0"
  timestamp="2010-11-11T14:40:10" >

  <imports>
    <import package="org.iso.otx.examples" document="Signatures" prefix="sig"/>
    <import package="org.iso.otx.examples" document="Validities" prefix="val"/>
  </imports>

  <procedures>
    <procedure name="test" implements="sig:mySignature" validFor="val:myValidity" id="2-1">
      <specification>A test procedure.</specification>
    </procedure>
  </procedures>
</otx>
```

7.5 Global declarations

7.5.1 Description

The global declaration block is the place where global identifiers for constants, document scope variables and context variables shall be declared. For each type of global declaration special rules apply as specified in the following. Globally declared identifiers shall be visible for every procedure and validity term defined in the same document, but they can also be imported by external documents for cross-document usage (see **<imports>** element, 7.4).

The counterparts of the global declaration block are the local parameter declaration block used by signatures and procedures (see 7.11) and the local declaration block used by procedures (see 7.12).

Concerning scope and memory allocation, special rules shall apply for global declarations. Please refer to Annex B for further specifications.

7.5.2 Syntax

The global **<declarations>** block is enclosed top-level by the **<otx>** element (see 7.3.2). Figure 14 shows the syntax of the **GlobalDeclarations** type.

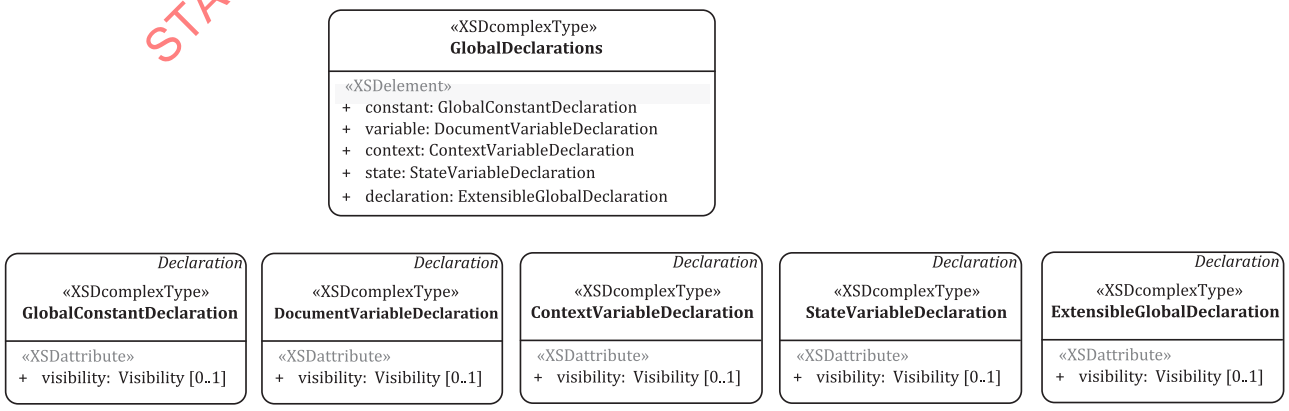

Figure 14 — Data model view: `GlobalDeclarations`

7.5.3 Semantics

All of the three global declaration types have properties inherited from their common base type `Declaration` and `NamedAndSpecified`. For the semantics of inherited properties see 7.16.7. Only the specific semantic properties of each type of `Declaration` are described here. `GlobalDeclarations` allows for declaring an arbitrary number of global constants, document variables and context variables (by utilizing `<xsd:choice>` with choice cardinality `[1..*]`):

— `name: OtxName [1]` (derived from `NamedAndSpecified`, see 7.16.4)

This attribute represents the name of the global declaration. It shall be unique throughout all global scope identifiers defined in the same OTX document. This constraint is verifiable through XML Schema validation (by the `<xsd:key>` constraint `"UniqueGlobalIdentifierNames"` specified in the OTX schema).

— `<constant>: GlobalConstantDeclaration`

Declares a global constant identifier. The value of a constant is fixed at declaration time – it is not allowed to be changed.

— `visibility: Visibility = {PRIVATE|PACKAGE|PUBLIC} [0..1]`

The attribute is a visibility modifier. The visibility levels are described by the `Visibility` enumeration (see 7.16.8), which allows the following attribute values:

— `"PRIVATE"`: the constant shall be used only from procedures or validities defined in the **same document**. This means that the constant shall be invisible for other OTX documents. This is the default.

— `"PACKAGE"`: the constant shall be used only from procedures or validities defined in the **same package** like the local OTX document. This means that the constant shall be invisible for OTX documents belonging to other packages.

— `"PUBLIC"`: the constant can be used from **anywhere**.

Associated checker rules:

— Core_Chk051 – immutability of constants, input parameters and context variables (see C.2.51);

— Core_Chk009 – mandatory constant initialization (see C.2.9).

— `<variable>: DocumentVariableDeclaration`

Declares a document variable which is only visible to procedures and validity terms defined within the same document.

— `visibility: Visibility = {PRIVATE|PACKAGE|PUBLIC} [0..1]`

The attribute is a visibility modifier. The visibility levels are described by the `Visibility` enumeration (see 7.16.8). For document variables, the attribute value is **fixed** to `"PRIVATE"`. This ensures that document variables are not visible to other OTX documents.

— `<context>: ContextVariableDeclaration`

Declares a context variable. See 6.7 for further information about the OTX context concept.

— `visibility: Visibility = {PRIVATE|PACKAGE|PUBLIC} [0..1]`

The attribute is a visibility modifier. The visibility levels are described by the `Visibility` enumeration (see 7.16.8), which allows the following attribute values:

— `"PRIVATE"`: the context variable shall be used only from procedures or validities defined in the **same document**. This means that the context variable shall be invisible for other OTX documents. This is the default.

— `"PACKAGE"`: the context variable shall be used only from procedures or validities defined in the **same package** as the local OTX document. This means that the context variable shall be invisible for OTX documents belonging to other packages.

— `"PUBLIC"`: the context variable can be used from **anywhere**.

Associated checker rules:

— Core_Chk051 – immutability of constants, input parameters and context variables (see C.2.51).

Context variables play a special role in the OTX data model. They represent a connection to contextual information which is defined by the surroundings, e.g. information about the currently diagnosed vehicle, the name of the workshop where the diagnostic application is running in, the user name of the currently logged in user. Unlike normal variables, the value of context variables cannot be changed directly by OTX actions. In OTX procedures, context variables shall—despite the name—be treated as constants. Context variables can only be changed by identification routines of the test application, for example, when another vehicle gets connected to the diagnostic application, an identification routine determines, e.g. the model or the vehicle identification number and makes this information available to the OTX runtime. OTX procedures can then read that information by using the context variable which stands for the respective context information.

OTX authors may define default values for context variables (this shall be done by using the `<init>` element, see 7.16.7). For cases when the value of a context variable cannot be determined, the default value applies instead. If no explicit default value is given, the implicit default value defined for the assigned data type applies. For data types for which no implicit value is specified, the context variable is uninitialized—reading such a variable will produce an exception at runtime, as specified in 7.15.3.

OTX documents declaring context variables should be connected to corresponding identification routines which return the current value of the contextual information in scope. Therefore, diagnostic applications require a mapping between context variables and according identification routines. The mapping may be kept in application specific places, but it may also be contained in metadata compartments of an OTX document (see 7.16.5).

Recommendations:

Concerning identification routines for gathering context information, there are no requirements or assumptions made in this document. The implementation of identification routines is application specific. Only in the case that an identification routine is implemented in the form of an OTX procedure, the following approach is recommended.

For a context variable to identification routine mapping, the `<metaData>` element should be used, containing two special purpose `<data>` elements:

— `< data key = "[Application Id].OtxPackage" > [PackageName]</data>`

  The `[ApplicationId]` should be a string identifying the specific diagnostic application for which the mapping applies. The string should be in dotted notation, providing as much uniqueness as possible, e.g. `"com.myCompany.myApplication"`. The `[PackageName]` identifies the package that contains the OTX procedure which serves as the identification routine.

— `< data key = ”[Application Id].OtxDocument” > [DocumentName]</data>`

  The `[DocumentName]` identifies the document that contains the OTX procedure which serves as identification routine.

— `< data key = "[Application Id].OtxProcedure" > [ProcedureName]</data>`

  The `[ProcedureName]` should be the name of the OTX procedure providing the identification routine.

— `< data key = "[Application Id].OtxOutParameter" > [ParameterName]</data>`

The `[ParameterName]` should be the name of the OTX procedure's out parameter which contains the context information.

By following this recommendation, the exchangeability of OTX documents can be ameliorated: a diagnostic application using an OTX document created by another application may add its own mapping information to its specific identification routines without having to overwrite the mapping of the other application.

— `<state>: StateVariableDeclaration`

Declares a state variable.

Test sequences require a technique to provide different status information to the environment, e.g. progress of execution or currently selected ECU. This is the intention of state variables. State variables are the counterpart of context variables, see context concept (see 6.7). They provide the mechanism to transport status information from inside a procedure to the environment, while context variables are the mechanism to transport environment information to the procedure.

IMPORTANT — State variables can only be set and cannot read inside OTX, i.e. the test logic shall not be based on the value of a state variable.

IMPORTANT — OTX documents declaring state variables should be connected to corresponding setting routines which can set the current value of the status in scope. Therefore, test applications require a mapping between state variables and according setting routines. The mapping may be kept in application specific places, but it may also be contained in metadata compartments of an OTX document (see 7.16.5). The recommendations for metadata compartments described in 7.5.3 shall be applied also for state variables.

OTX authors may define default values for state variables. If an initial value is defined the setting routine should be called immediately after declaration of the state variable. For data types for which no implicit value is specified, the state variable is un-initialized, the setting routine will not be called. If no initial value is defined and the variable value is never written inside a procedure, the setting routine will never be called. Inside a procedure the setting routine will only be called if the value of that variable is changed.

Changes of a state variable shall be immediately reported to the runtime environment.

Associated checker rules:

— Core_Chk051 – immutability of constants, input parameters and context variables (see C.2.51);

— Core_Chk065 – write only state variables (see C.2.65).

— `visibility: Visibility = {PRIVATE|PACKAGE|PUBLIC} [0..1]`

The attribute is a visibility modifier. The visibility levels are described by the `Visibility` enumeration (see 7.16.8), which allows the following attribute values:

— `"PRIVATE"`: the context variable shall be used only from procedures or validities defined in the **same document**. This means that the context variable shall be invisible for other OTX documents. This is the default.

— `"PACKAGE"`: the context variable shall be used only from procedures or validities defined in the **same package** as the local OTX document. This means that the context variable shall be invisible for OTX documents belonging to other packages.

— `"PUBLIC"`: the context variable can be used from **anywhere**.

— `<declararation>: ExtensibleGlobalDeclaration`

Declares a global declaration which can be extented by additional realisations defined in new OTX extensions using the standardised extension mechanism.

— `visibility: Visibility = {PRIVATE|PACKAGE|PUBLIC} [0..1]`

The attribute is a visibility modifier. The visibility levels are described by the `Visibility` enumeration (see 7.16.8), which allows the following attribute values:

— `"PRIVATE"`: the context variable shall be used only from procedures or validities defined in the **same document**. This means that the context variable shall be invisible for other OTX documents. This is the default.

— `"PACKAGE"`: the context variable shall be used only from procedures or validities defined in the **same package** as the local OTX document. This means that the context variable shall be invisible for OTX documents belonging to other packages.

— `"PUBLIC"`: the context variable can be used from **anywhere.**

7.5.4 Example

The example below defines a global constant, a document variable as well as two context variables:

— A global constant float named `"PI"`, initialized by float value "`3.14159265`". Its visibility is `"PUBLIC"` which means that the constant can be accessed by any other importing OTX document.

— A Boolean document variable named "`dv`". The `visibility` attribute is not given explicitly in the document; nonetheless the variable is implicitly `"PRIVATE"`. This setting is fixed for all document variables by the OTX schema.

— A context variable `"VIN"`, which is connected to an identification routine. The metadata assigned to the context variable contains a mapping to the identification routine; it is an OTX procedure named `"getVehicleIdent"` belonging to the package `"identification"`. The out-parameter `"vin"` yields the context value (the vehicle identification number of the currently diagnosed vehicle).

— A context variable `"HAS_SUNROOF"`, there is no mapping to an identification routine given in the metadata. If there is no external mapping also, the context variable is not connected to the "real world" – in this case the default value `false` will apply.

Sample of GlobalDeclarations

```
<?xml version="1.0" encoding="UTF-8"?>
<otx xmlns="http://iso.org/OTX/1.0.0" id="3"
  name="GlobalDeclarationsExample"
  package="org.iso.otx.examples"
  version="1.0"
  timestamp="2009-10-20T14:40:10"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <declarations>
    <constant name="PI" visibility="PUBLIC" id="3-d1">
      <specification>This defines a global constant</specification>
      <realisation>
        <dataType xsi:type="Float"><init value="3.14159265"/></dataType>
      </realisation>
    </constant>

    <variable name="dv" id="3-d2">
      <specification>A document-wide visible variable</specification>
      <realisation>
        <dataType xsi:type="Boolean"/>
      </realisation>
    </variable>

    <context name="VIN" visibility="PUBLIC" id="3-d3">
      <specification>Holds the VIN of the currently diagnosed vehicle</specification>
      <metaData>
        <data key="com.myCompany.myApp.OtxPackage">identification</data>
        <data key="com.myCompany.myApp.OtxProcedure">getVehicleIdent</data>
        <data key="com.myCompany.myApp.OtxOutParameter">vin</data>
      </metaData>
      <realisation>
        <dataType xsi:type="String"/>
      </realisation>
    </context>

    <context name="HAS_SUNROOF" visibility="PUBLIC" id="3-d4">
      <specification>True if diagnosed vehicle has a sunroof</specification>
      <realisation>
        <dataType xsi:type="Boolean"><init value="false"/></dataType>
      </realisation>
    </context>
  </declarations>
</otx>
```

7.6 Validity terms

7.6.1 Description

OTX documents may contain a list of validity terms. This is required for the validities concept, see 6.8. A validity term is a uniquely named, reusable Boolean term which is defined at global level. Validity terms are primarily used to configure test sequence behaviour according to runtime context. In contrast to Boolean context variables which can be used as validity, too, validity terms allow describing compound Boolean expressions which may depend on more than just one Boolean value.

Validities can be addressed cross-document; this allows defining a set of popular validities in a central OTX document which can be reused by other OTX documents via **`OtxLink`** association.

`Action` nodes, **`Group`** nodes and **`Procedures`** can be connected to validity terms by the **`validFor`** attribute (see 7.14, 7.13.4.2 and 7.9.). At runtime, the Boolean term value of the referenced validity term is evaluated; it determines whether an **`ActionRealisation`**, **`GroupRealisation`** or a **`Procedure`** is valid or not.

7.6.2 Syntax

Figure 15 shows the syntax of the **`Validities`** type.

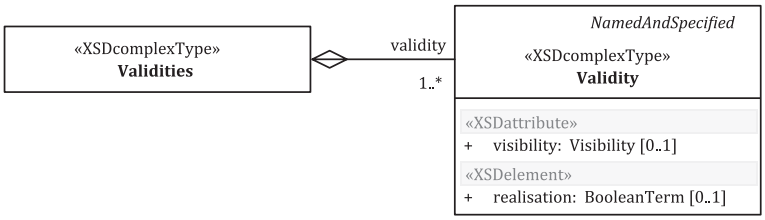

Figure 15 — Data model view: `Validities`

7.6.3 Semantics

The properties of the **`Validity`** type have the following semantics:

— **`id: OtxId [1]`** (derived from **`NamedAndSpecified`**, see 7.16.4)

This represents the **`<validity>`** element's id. It shall be unique among all other ids in a document. Please refer to 7.16.4 for details concerning ids in OTX documents.

— **`name: OtxName [1]`** (derived from **`NamedAndSpecified`**, see 7.16.4)

This attribute represents the name of the validity term. It shall be unique throughout all global scope identifiers defined in the same OTX document. This constraint is verifiable through XML Schema validation (by the **`<xsd:key>`** constraint **`"UniqueGlobalIdentifierNames"`** specified in the OTX schema).

Actions, groups and procedures can be connected to validity terms by referring to the name (through the **`validFor`** attribute containing an **`OtxLink`**).

— **`visibility: Visibility = {PRIVATE|PACKAGE|PUBLIC} [0..1]`**

The attribute is a visibility modifier. The visibility levels are described by the **`Visibility`** enumeration (see 7.16.8), which allows the following attribute values:

— **`"PRIVATE"`**: the validity term shall be accessible only by the **same document**. This means that the validity term shall be invisible for other OTX documents. This is the default.

— **`"PACKAGE"`**: the validity term shall be accessible only by OTX documents of the **same package** like the local document. This means that the validity term shall be invisible for documents belonging to other packages.

— **`"PUBLIC"`**: the validity term shall be accessible by **any** other OTX document.

— **`<metaData>: MetaData [0..1]`** (derived from **`NamedAndSpecified`**, see 7.16.4)

This optional element is used for adding metadata to a validity term. For details on the **`MetaData`** type, refer to 7.16.5.

— **`<specification> : xsd:string [0..1]`** (derived from **`NamedAndSpecified`**, see 7.16.4)

The static string content of this optional element shall be used to specify the purpose of the validity term for human readers. The complement of **`<specification>`** is the element **`<realisation>`** (see below).

Associated checker rules:

— Core_Chk007 – have specification if no realisation exists (see C.2.7).

— **`<realisation> : BooleanTerm [0..1]`**

The **`BooleanTerm`** given by the **`<realisation>`** element represents the validity condition: the validity term holds if and only if the term evaluation returns **`true`** with respect to the runtime context.

Validity term truth values may be evaluated on first use. Later uses of a validity term do not require re-evaluation as long as no relevant context change occurred. In between context changes, validity term values may be treated as runtime constants. Only in the case when a context changes, concerned validity terms should be re-evaluated on next use.

7.6.4 Example

The example below defines two validities: they can be used to identify in which environment the runtime is located, either in a repair workshop or at an assembly line. Of course, relevant context information shall be available for the OTX runtime system, namely via the **LOCATION** context variable.

Sample of ValidityTerm

```
<?xml version="1.0" encoding="UTF-8"?>
<otx xmlns="http://iso.org/OTX/1.0.0" id="4"
  name="ValidityTermExample"
  package="org.iso.otx.examples"
  version="1.0"
  timestamp="2009-10-20T14:40:10"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <imports>
    <import document="contexts" package="org.iso.otx.examples" prefix="context"/>
  </imports>

  <validities>
    <validity name="Workshop" id="4-v1">
      <specification>Valid if executed in a workshop environment</specification>
      <realisation xsi:type="IsEqual">
        <term xsi:type="StringValue" valueOf="context:LOCATION" />
        <term xsi:type="StringLiteral" value="DealershipWorkshop" />
      </realisation>
    </validity>

    <validity name="Assembly" id="4-v2">
      <specification>Valid if executed at an assembly line</specification>
      <realisation xsi:type="IsEqual">
        <term xsi:type="StringValue" valueOf="context:LOCATION"/>
        <term xsi:type="StringLiteral" value="AssemblyLine" />
      </realisation>
    </validity>
  </validities>
</otx>
```

7.7 Signatures

7.7.1 Description

OTX signatures support the design by contract concept: generally speaking, a signature[4] represents an interface description to another software module whose inner implementation details do not need to be known by the user of that module. As long as the user and the implementing software module(s) obey to the signature specification, the interoperability is guaranteed: the user has to fulfill all the prerequisites described by the signature (e.g. by providing correct **ProcedureCall** arguments defined by a **ProcedureSignature**, see 7.8) and the software module implementing the signature is obligated to complete the function specified by the signature if the prerequisites are fulfilled.

OTX allows different types of signatures. In the OTX core data model there only one type of signature called **ProcedureSignature**. OTX extensions may specify additional signatures for extension-specific purposes, so for example the **ScreenSignature** of the HMI extension as specified in ISO 13209-3[2].

7.7.2 Syntax

Figure 16 shows the syntax of the **Signatures** type.

4) This is also called *interface*, *prototype* or *header* in other programming languages.

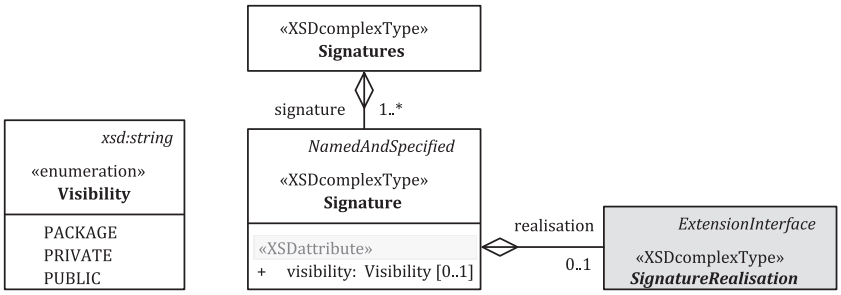

Figure 16 — Data model view: `Signatures`

7.7.3 Semantics

The properties of the **`Signature`** type have the following semantics:

— **`id: OtxId [1]`** (derived from **`NamedAndSpecified`**, see 7.16.4)

This represents the **`<signature>`** element's id. It shall be unique among all other ids in a document. Please refer to 7.16.4 for details concerning ids in OTX documents.

— **`name: OtxName [1]`** (derived from **`NamedAndSpecified`**, see 7.16.4)

This attribute represents the name of the signature. It shall be unique throughout all global identifiers defined in the same OTX document. This constraint is verifiable through XML Schema validation (by the **`<xsd:key>`** constraint **`"UniqueProcedureAndSignatureNames"`** specified in the OTX schema).

— **`visibility: Visibility = {PRIVATE|PACKAGE|PUBLIC} [0..1]`**

The attribute is a visibility modifier. The visibility levels are described by the **`Visibility`** enumeration, which allows the following attribute values:

— **`"PRIVATE"`**: the signature is visible only within the **same document**. This means that using the signature from outside of the document is not allowed.

— **`"PACKAGE"`**: the signature is visible only within the **same package**. Using the signature from outside of the package is not allowed. This is the default.

— **`"PUBLIC"`**: the signature can be used from **anywhere**.

— **`<metaData>: MetaData [0..1]`** (derived from **`NamedAndSpecified`**, see 7.16.4)

This optional element is used for adding metadata to a signature. For details on the **`MetaData`** type, refer to 7.16.5.

— **`<specification> : xsd:string [0..1]`** (derived from **`NamedAndSpecified`**, see 7.16.4)

The static string content of this optional element shall be used to specify the purpose of the signature for the human reader. The complement of **`<specification>`** is the element **`<realisation>`** (see below).

Associated checker rules:

— Core_Chk007 – have specification if no realisation exists (see C.2.7).

— **`<realisation> : SignatureRealisation [0..1]`**

`SignatureRealisation` is an abstract type. This means that for the **`<realisation>`** element, a concrete child of **`SignatureRealisation`** shall be chosen by using the **`xsi:type`** attribute. In the OTX core, there is only one child available, namely **`ProcedureSignature`** (see 7.8). OTX extensions may

derive further child types, e.g. the **ScreenSignature** in the HMI extension (see ISO 13209-3 [2]). For details about derived **SignatureRealisation** types refer to the respective specifications.

7.8 Procedure signatures

7.8.1 Description

The **ProcedureSignature** type is required for the signature concept (see 6.9).

Where procedures declare a set of parameters, a set of local variables and constants and the procedure flow containing the program logic, procedure signatures only declare a set of parameters. Procedures which implement a specific procedure signature need to have the same set of parameters (see 7.9 on procedures). Procedure signatures represent the "empty shell" of such procedures.

Signature-implementing procedures can be called indirectly via a procedure signature. This is described in the **ProcedureCall** action, see 7.14.5.

7.8.2 Syntax

Figure 17 shows the syntax of the **ProcedureSignature** type.

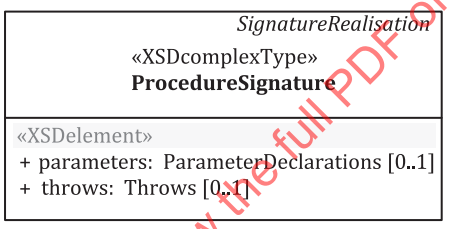

Figure 17 — Data model view: ProcedureSignatures

7.8.3 Semantics

ProcedureSignature is a **SignatureRealisation** (7.7). Only the specific semantic properties of the **ProcedureSignature** type are described here.

The properties of **ProcedureSignature** have the following semantics:

— **<parameters>: Parameters [0..1]**

Parameters of the signature shall be declared in this block (see 7.11).

— **<throws>: Throws [0..1]**

Exceptions shall be declared in this block. Each exception listed here may potentially be thrown by signature-implementing procedures.

— **<exception>: Exception [1..*]**

Represents the exceptions. For each exception, the type shall be chosen by the **xsi:type** attribute.

IMPORTANT — A signature without <realisation> represents a NOP (No Operation) at runtime.

7.8.4 Example

The first example below shows a couple of signatures in an OTX File. While the latter two signatures are only in specification stage, the first one, **GetIgnitionState**, is in realisation stage. The procedure signature describes procedures which return the state of the ignition of a vehicle. Since this is a

signature, there is no implementation; it is up to the signature-implementing procedures to define how exactly the ignition state shall be computed.

The second example below shows two signature-implementing procedures. The first gets the ignition via a diagnostic service sent to the vehicle (valid for a workshop environment) whereas the other gets it by measuring at the OBD connector (valid for an assembly line environment). Please refer also to the example given in 7.14.5.4 (`ProcedureCall`).

Sample of Signatures

```
<?xml version="1.0" encoding="UTF-8"?>
<otx xmlns="http://iso.org/OTX/1.0.0" id="5"
  name="SignatureExample"
  package="org.iso.otx.examples"
  version="1.0"
  timestamp="2009-10-20T14:40:10"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <signatures>
    <signature name="getIgnitionState" id="5-s1">
      <specification>Returns the status of the ignition.</specification>
      <realisation xsi:type="ProcedureSignature">
        <parameters>
          <outParam name="state" id="5-d1">
            <specification>Ignition state result. true:ON | false:OFF</specification>
            <realisation>
              <dataType xsi:type="Boolean" />
            </realisation>
          </outParam>
        </parameters>
      </realisation>
    </signature>

    <signature name="checkHeadlights" id="5-s2">
      <specification>Tests the head lights for proper functioning.</specification>
    </signature>

    <signature name="retrieveVIN" id="5-s3">
      <specification>Returns the vehicle Identification number.</specification>
    </signature>
  </signatures>
</otx>
```

Sample of ImplementingProcedures

```
<?xml version="1.0" encoding="UTF-8"?>
<otx xmlns="http://iso.org/OTX/1.0.0" id="6"
  name="ImplementingProcedureExample"
  package="org.iso.otx.examples"
  version="1.0"
  timestamp="2009-10-20T14:40:10"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <imports>
    <import prefix="sig" package="org.iso.otx.examples" document="SignatureExample" />
    <import prefix="val" package="org.iso.otx.examples" document="Validities" />
  </imports>

  <procedures>
    <procedure name="getIgsByDiagService" implements="sig:getIgnitionState" validFor="val:Workshop" id="6-p1">
      <specification>Gets the state by using a diagnostic service</specification>
      <realisation>
        <parameters>
          <outParam name="state" id="6-d1">
            <specification>Ignition state result. true:ON | false:OFF</specification>
            <realisation>
              <dataType xsi:type="Boolean" />
            </realisation>
          </outParam>
        </parameters>
        <flow>
          <!-- The implementation goes here -->
        </flow>
      </realisation>
    </procedure>

    <procedure name="getIgsByObdConnector" implements="sig:getIgnitionState" validFor="val:Assembly" id="6-p2">
      <specification>Gets the state by measuring via OBD connector</specification>
    </procedure>
  </procedures>
</otx>
```

7.9 Procedures

7.9.1 Description

A **<procedure>** element represents an executable unit in an OTX document. It can be used as the entry point for OTX interpreters when a test sequence shall be executed, but it can also be the target of a procedure call from another OTX procedure (see 7.14.5).

7.9.2 Syntax

Figure 18 shows the syntax of the **Procedure** type.

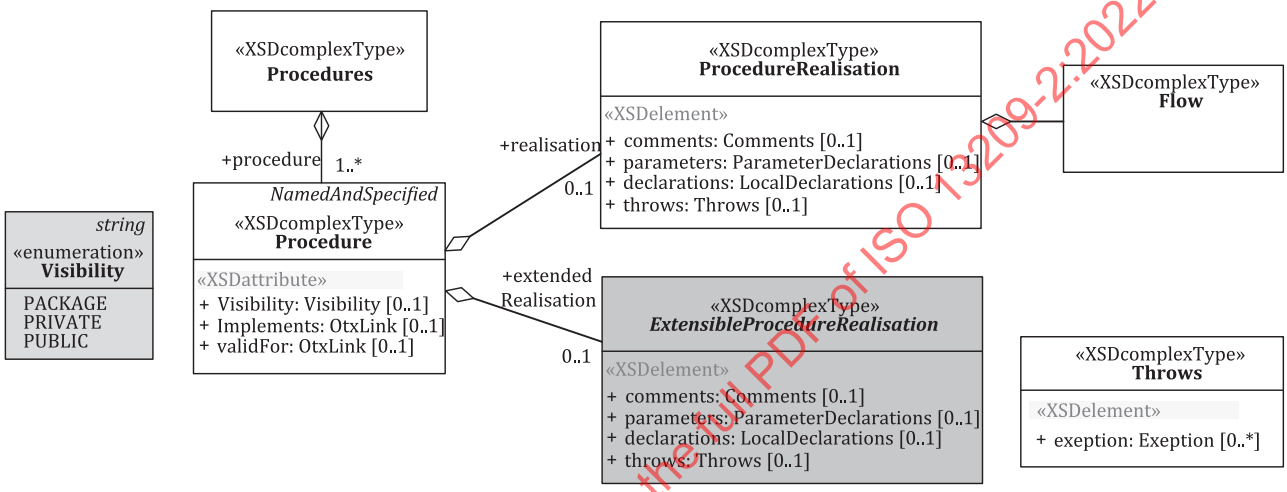

Figure 18 — Data model view: `Procedures`

7.9.3 Semantics

The properties of the **Procedure** type have the following semantics:

— **id: OtxId [1]** (derived from **NamedAndSpecified**, see 7.16.4)

This represents the **<procedure>** element's id. It shall be unique among all other ids in a document. Please refer to 7.16.4 for details concerning ids in OTX documents.

— **name: OtxName [1]** (derived from **NamedAndSpecified**, see 7.16.4)

This attribute represents the name of the procedure. It shall be unique throughout global identifiers defined in the same OTX document. This constraint is verifiable through XML Schema validation (by **<xsd:key>** constraints specified in the OTX schema).

IMPORTANT — Special semantic for main-procedures: If the `name` attribute of a procedure has the value `"main"`, then this procedure shall be treated as a top-level procedure which represents the entry point for test sequence execution in an OTX application. Main-procedures shall always be public. They can be called by other OTX test sequences also like normal procedures.

Associated checker rules:

— Core_Chk008 – public main procedure (see C.2.8).

— **visibility: Visibility = {PRIVATE|PACKAGE|PUBLIC} [0..1]**

The attribute is a visibility modifier. The visibility levels are described by the **Visibility** enumeration, which allows the following attribute values:

- `"PRIVATE"`: the procedure shall be visible only within the **same document**. This means that calling the procedure from outside of the document is not allowed. This is the default.
- `"PACKAGE"`: the procedure shall be visible only within the **same package**. Calling the procedure from outside of the package is not allowed.
- `"PUBLIC"`: the procedure shall be visible from **any document in any package**.

— `implements: OtxLink [0..1]`

This optional attribute supports the signature concept (see 6.9). It shall contain the qualified name of the signature that a procedure implements.

Associated checker rules:

- Core_Chk053 – no dangling OtxLink associations (see C.2.53);
- Core_Chk026 – no duplicate validities for procedures implementing the same signature (see C.2.26);
- Core_Chk027 – procedure parameters match signature parameters (see C.2.27).

— `validFor: OtxLink [0..1]`

This optional procedure attribute supports the signature concept which is based on the validities concept (see 6.9 and 6.8). It can be connected to a `Boolean` context variable, a global `Boolean` constant or to a validity term (by `OtxLink`). At runtime, the truth value of the associated context variable, constant or the validity term determines whether a procedure shall be executed (validity value is `true`) or not (validity value is `false`). This allows context-based disabling/enabling of procedures. Refer to the `ProcedureCall` action described in 7.14.5.

Associated checker rules:

- Core_Chk053 – no dangling OtxLink associations (see C.2.53);
- Core_Chk013 – correct referencing of validities (see C.2.13).

— `<metaData>: MetaData [0..1]` (derived from `NamedAndSpecified`, see 7.16.4)

This optional element is used for adding metadata to a procedure (refer to the `MetaData` type, 7.16.5).

— `<specification> : xsd:string [0..1]` (derived from `NamedAndSpecified`, see 7.16.4)

The static string content of this optional element shall be used to specify the purpose of the procedure for the human reader. The complement of `<specification>` is the element `<realisation>` (see below).

Associated checker rules:

- Core_Chk007 – have specification if no realisation exists (see C.2.7).

— `<realisation> : ProcedureRealisation [0..1]`

Specifies the procedure implementation parts. When a procedure enters realisation stage in the test sequence development process (after specification stage), this element shall be instantiated. Its type, `ProcedureRealisation`, represents a wrapper for all elements needed for the procedure implementation:

— **<comments>: Comments [0..1]**

This optional element is a container for an arbitrary-length list of floating comments. They shall be used for commenting parts of the procedure flow implementation for the human reader. For a description of the **Comment** type, refer to 7.10.

— **<parameters>: Parameters [0..1]**

Parameters of the procedure shall be declared in this block (see 7.11).

— **<declarations>: Declarations [0..1]**

Local variables and constants of the procedure shall be declared in this block (see 7.12).

— **<throws>: Throws [0..1]**

Exceptions shall be declared here. Each exception listed may potentially be thrown by the procedure under certain circumstances.

— **<exception>: Exception [1..*]**

Represents the exceptions. For each exception, the type is chosen by the **xsi:type** attribute.

— **<flow>: Flow [1]**

This block contains the procedure logic (see 7.16.9). The logic is represented by a sequence of nodes, each of which has type-specific semantics (see 7.13).

— **<extendedRealisation> : ExtensibleProcedureRealisation [0..1]**

Declares a procedure which can be extended by additional realisations defined in new OTX extensions using the standardised extension mechanism, example: flow chart procedure. This procedure can be called by the ProcedureCall.

Its type, **ExtensibleProcedureRealisation**, represents a wrapper for all elements needed for the procedure implementation:

— **<comments>: Comments [0..1]**

This optional element is a container for an arbitrary-length list of floating comments. They shall be used for commenting parts of the procedure flow implementation for the human reader. For a description of the **Comment** type, refer to 7.10.

— **<parameters>: Parameters [0..1]**

Parameters of the procedure shall be declared in this block (see 7.11).

— **<declarations>: Declarations [0..1]**

Local variables and constants of the procedure shall be declared in this block (see 7.12).

— **<throws>: Throws [0..1]**

Exceptions shall be declared here. Each exception listed may potentially be thrown by the procedure under certain circumstances.

— **<exception>: Exception [1..*]**

Represents the exceptions. For each exception, the type is chosen by the **xsi:type** attribute.

IMPORTANT — A procedure without <realisation> or <extendedRealisation> represents a NOP (No Operation) at runtime. The same applies procedure signatures without <realisation>.

7.9.4 Example

The example below shows two procedures defined in the same OTX document. The first is an entry point test sequence (`name = "main"`). The other is a procedure that can be used only by procedure calls from within the same package. It implements the signature "`sig:writeMsg`" which is defined in the document "`Signatures`" in the package "`org.iso.otx.examples`". It can be executed only if the validity value of "`val:DebugMode`" is `true`. The validity term is defined in the document "`Validities`" in the same package. Both procedures neither declare any parameters, variables or constants, nor contain a program flow. These sub-elements are declared in the following subclauses.

Sample of Procedures

```
<?xml version="1.0" encoding="UTF-8"?>
<otx xmlns="http://iso.org/OTX/1.0.0" id="7"
  name="ProcedureExample"
  package="org.iso.otx.examples"
  version="1.0"
  timestamp="2009-10-20T14:40:10"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <imports>
    <import prefix="sig" package="org.iso.otx.examples" document="Signatures" />
    <import prefix="val" package="org.iso.otx.examples" document="Validities" />
  </imports>

  <procedures>
    <procedure name="main" visibility="PUBLIC" id="7-p1">
      <specification>This is an empty top-level procedure, a test sequence</specification>
      <realisation>
        <flow/>
      <realisation>
    </procedure>
    <procedure name="writeDebugMsg" implements="sig:writeMsg"
      validFor="val:DebugMode" visibility="PACKAGE" id="7-p2">
      <specification>A empty procedure implementing a signature with validity information.</specification>
      <realisation>
        <flow/>
      <realisation>
    </procedure>
  </procedures>
</otx>
```

7.10 Floating comments

7.10.1 Description

Procedures allow an optional `<comments>` element. It contains any number of `<comment>` elements which can be linked to any number of nodes inside the procedure flow. Comments can link to sub-structures inside of nodes and to other comment nodes also, as specified below. This feature supports graphical OTX editors by allowing any graphically presentable OTX feature to be linked to comments. They are thought to help test sequence authors to provide additional, human readable information during the test sequence development process, especially for the specification and intermediate stage in test sequence development.

A single comment carries a text message for the human reader, and has a list of links pointing to the elements in the procedure that the comment refers to.

7.10.2 Syntax

Figure 19 shows the structure of the complex type `Comment`.

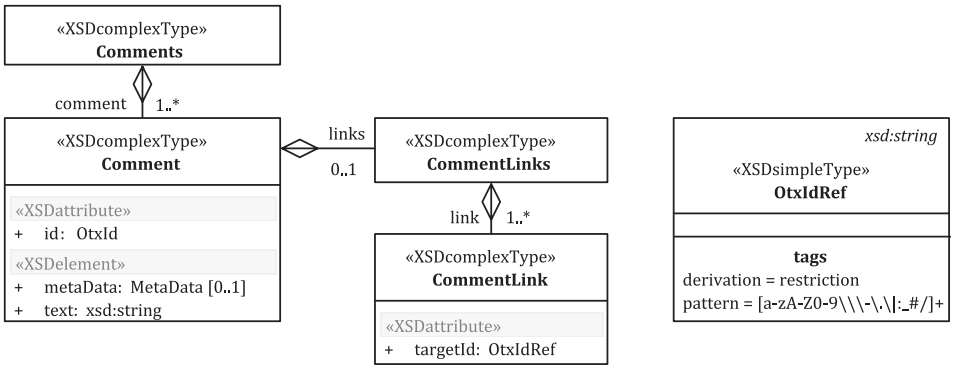

Figure 19 — Data model view: `Comment`

7.10.3 Semantics

The properties of the **`Comment`** type have the following semantics:

— **`id: OtxId [1]`**

This represents a **`<comment>`** element's id. It shall be unique among all other ids in a document. Please refer to 7.16.4 for details concerning ids in OTX documents.

— **`<metaData>: MetaData [0..1]`**

Used for adding metadata to a comment. For details on the **`MetaData`** type, refer to 7.16.5.

— **`<text>: xsd:string [1]`**

The comment text that shall be displayed to the human reader.

— **`<links>: Links [0..1]`**

Contains an arbitrary-length list of **`<link>`** elements.

— **`<link>: CommentLink [1..*]`**

Links the comment to a commented entity by its **`targetId`** attribute:

— **`targetId: OtxIdRef [1]`**

The link between a comment and the commented entity is established with the **`targetId`** attribute of the comment link. The attribute value shall match the **`id`** of a commented entity. Otherwise, a comment link might point to nowhere which is undesirable. XML validation enforces that for each **`targetId`** value, there is an entity in the procedure with the corresponding **`id`** value (using **`<xsd:keyref>`** constraints specified in the OTX schema).

The **`OtxIdRef`** type is the counterpart of the **`OtxId`** type. In analogy to **`OtxId`**, it is a pattern-restriction of the **`xsd:string`** simple type. The value space of the attribute is restricted by the regular expression "**`[a-zA-Z0-9\\\-\.\|:_#/]+`**", which allows the basic letters, numbers as well as a set of delimiters.

NOTE Entities to which a comment's **`<link>`** can point to are the flow nodes, branch case conditions, loop conditions as well as comments themselves.

7.10.4 Example

The example below shows two comments linked to different entities in the procedure. The first comment points at the action node in the procedure flow and at the group in the loop node. The second comment points at the loop configuration and to another comment.

Sample of Comments

```
<?xml version="1.0" encoding="UTF-8"?>
<otx xmlns="http://iso.org/OTX/1.0.0" id="8"
  name="CommentExample"
  package="org.iso.otx.examples"
  version="1.0"
  timestamp="2009-10-20T14:40:10"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
  <procedures>
    <procedure name="main" visibility="PUBLIC" id="8-p1">
      <specification>Demonstration of a commented procedure</specification>
      <realisation>
        <comments>
          <comment id="cmt1">
            <text>This is a comment for two nodes in the flow</text>
            <links>
              <link targetId="action1" />
              <link targetId="group1" />
            </links>
          </comment>
          <comment id="cmt2">
            <text>This comment is linked to another comment and a loop configuration</text>
            <links>
              <link targetId="loop1config" />
              <link targetId="cmt1" />
            </links>
          </comment>
        </comments>
        <flow>
          <action id="action1" />
          <loop id="loop1" name="emptyLoop">
            <realisation>
              <configuration id="loop1config" />
              <flow>
                <group id="group1" />
                <action id="action2" />
              </flow>
            </realisation>
          </loop>
        </flow>
      </realisation>
    </procedure>
  </procedures>
</otx>
```

7.11 Parameter declarations

7.11.1 Description

The parameter declaration block allows for declaring in-, inout- and out-parameters of signatures and procedures. This information is especially important for the **`ProcedureCall`** action, as specified by 7.14.5.

7.11.2 Syntax

The declaration block **`<parameters>`** is located in the **`<realisation>`** element of procedures or signatures (see 7.7 and 7.9). Figure 20 shows the syntax of the **`ParameterDeclarations`** type.

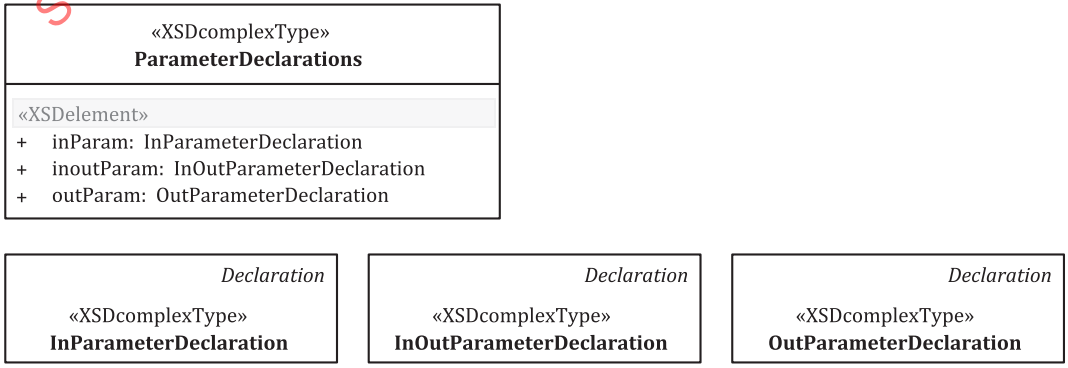

Figure 20 — Data model view: `ParameterDeclarations`

7.11.3 Semantics

All parameter declaration types have common properties inherited from their common base types **`Declaration`** and **`NamedAndSpecified`**. For the semantics of inherited properties see 7.16.7. Only the specific semantic properties of each type of **`Declaration`** are described here.

The syntactical order of parameter declarations is semantically irrelevant. Parameter identifiers have procedure wide scope. **`ParameterDeclarations`** allows for declaring an arbitrary number of in-, out- and inout-parameters (by utilizing **`<xsd:choice> [1..*]`**):

— **`<inParam>: InParameter`**

Declares an input parameter of the signature or procedure. When calling a signature or procedure, an input argument may be omitted **if and only if** there is an initial value defined for the parameter. This initial value shall apply in place of the missing argument. Input parameters shall be treated as constants – the value is not allowed to change throughout procedure execution.

Associated checker rules:

— Core_Chk051 – immutability of constants, input parameters and context variables (see C.2.51);

— Core_Chk030 – input- and in&output-argument omission (see C.2.30).

— **`<inoutParam>: InOutParameter`**

Declares an input/output parameter of the procedure. Arguments for in&output parameters shall be passed by reference. This means that any change to the parameter value is visible to the caller also. When calling a signature or procedure, an input/output argument may be omitted **if and only if** there is an initial value defined for the parameter.

Associated checker rules:

— Core_Chk030 – input- and in&output-argument omission (see C.2.30).

— **`<outParam>: OutParameter`**

Declares an output parameter of the procedure. Output parameters shall be returned to the caller. The caller may omit output parameters freely (e.g. in the case when there is no interest in one of the returned values). If the procedure never changes (e.g. assignment of a new value or modification of an existing value, including deep changes on complex values) a declared output parameter and there is no explicit initial value defined for it, the output argument variable stays unchanged. If the procedure throws an exception the output parameter stays unchanged. Otherwise the following rule shall apply: if the parameter has an explicit initial value, or the procedure writes it, the value will be assigned to the corresponding output argument variable (this rule is also applicable where a procedure is terminated as a result of **`TerminateLanes`**).

Signature or Procedure parameter declarations carry a **`name`** attribute (derived from **`NamedAndSpecified`**, see 7.16.4). Local parameter declaration names shall be unique among all local declaration and local parameter declaration names. This constraint is verifiable through XML schema validation (by **`<xsd:key>`** constraints specified in the OTX schema).

Furthermore, parameter declarations with the same name as a global declaration shall be allowed. At this, OTX follows the concept of shadowing: if an identifier is used which is declared both locally and globally, the local identifier shadows the global identifier. This means that the global identifier is invisible and not useable by any node in the local procedure. Since this may lead to unwanted situations, identifier shadowing is discouraged. To avoid it, distinct names should be used for local and global declarations. Another solution is making an OTX document import itself; this provides a prefix which can be used for unambiguously referring to global declarations.

Associated checker rules:

— Core_Chk052 – identifier shadowing (see C.2.52).

7.11.4 Example

The example given below shows a procedure **"ListItemMeanValue"** with two parameter declarations:

— a **List** of **Integer** type input parameter named **"listOfInt"**,

— a **Float** type output parameter named **"mean"**.

Sample of ParameterDeclarations

```
<?xml version="1.0" encoding="UTF-8"?>
<otx xmlns="http://iso.org/OTX/1.0.0" id="9" version="1.0" timestamp="2009-10-20T14:40:10"
     name="ParameterDeclarationsExample" package="org.iso.otx.examples"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <procedures>
    <procedure name="ListItemMeanValue" visibility="PUBLIC" id="9-p1">
      <specification>
        Demonstration of parameter declarations. The procedure calculates the arithmetic average of the
        input list's items ("list" parameter) and returns the average value in output parameter "mean"
      </specification>
      <realisation>
        <parameters>
          <inParam name="listOfInt" id="9-d1">
            <specification>Input parameter: List of Integers</specification>
            <realisation>
              <dataType xsi:type="List" >
                <itemType xsi:type="Integer"/>
              </dataType>
            </realisation>
          </inParam>
          <outParam name="mean" id="9-d2">
            <specification>Output parameter: Mean value (Float)</specification>
            <realisation>
              <dataType xsi:type="Float"/>
            </realisation>
          </outParam>
        </parameters>
        <flow/>
      </realisation>
    </procedure>
  </procedures>
</otx>
```

7.12 Local declarations

7.12.1 Description

The local declaration block allows for declaring constants and variables which are visible only for the declaring procedure.

7.12.2 Syntax

The local declaration block **<declarations>** is enclosed in the **<procedure>** element (see 7.9). Figure 21 shows the syntax of the **LocalDeclarations** type.

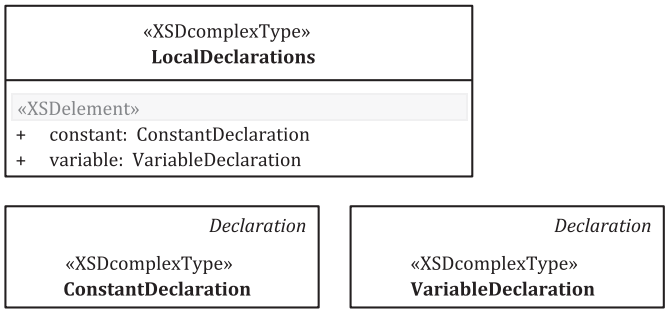

Figure 21 — Data model view: LocalDeclarations

7.12.3 Semantics

All local constant and variable declarations have common properties inherited from their common base type **`Declaration`** and **`NamedAndSpecified`**. See 7.16.7 for semantics of inherited properties. Only the specific semantic properties of each type of **`Declaration`** are described here.

The syntactical order of variable and constant declarations is semantically irrelevant. Constant and variable identifiers have procedure wide scope. **`LocalDeclarations`** allows declaring an arbitrary number of local constants and variables (by utilizing **`<xsd:choice> [1..*]`**):

— **`<constant>: Constant`**

  Declares a constant identifier for the procedure. The value of a constant is fixed at declaration time, it is not allowed to change throughout procedure execution.

  Associated checker rules:

  — Core_Chk009 – mandatory constant initialization (see C.2.9).

— **`<variable>: Variable`**

  Declares a variable identifier for the procedure. The value of the variable is allowed to change throughout procedure execution. If no initial value is given, the default initial value defined for its data type applies (refer to 7.16.7.4).

Local constants and variables carry a **`name`** attribute (derived from **`NamedAndSpecified`**, see 7.16.4). Local constant and variable declaration names shall be unique among all local declaration and local parameter declaration names. This constraint is verifiable through XML schema validation (by **`<xsd:key>`** constraints specified in the OTX schema).

Furthermore, local declarations with the same name as a global declaration shall be allowed. At this, OTX follows the concept of shadowing: if an identifier is used which is declared both locally and globally in the same OTX document, the local identifier shadows the global identifier. This means that the global identifier is invisible and not useable by any node in the local procedure. Since this may lead to unwanted situations, identifier shadowing is discouraged. To avoid it, distinct names should be used for local and global declarations. Another solution is making an OTX document import itself; this provides a prefix which can be used for unambiguously referring to global declarations.

Associated checker rules:

— Core_Chk052 – identifier shadowing (see C.2.52).

7.12.4 Example

The example below shows the declaration of

— a local constant float named **`"PI"`**, initialized by float value “**`3.14159265`**”,

— a local variable named “**`v`**” which has no data type assigned to it yet.

Sample of LocalDeclarations

```
<?xml version="1.0" encoding="UTF-8"?>
<otx xmlns="http://iso.org/OTX/1.0.0" id="10"
  name="LocalDeclarationsExample"
  package="org.iso.otx.examples"
  version="1.0"
  timestamp="2009-10-20T14:40:10"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <procedures>
    <procedure name="main" visibility="PUBLIC" id="10-p1">
      <specification>Demonstration of variable and constant declarations</specification>
      <realisation>
        <declarations>

          <constant name="PI" id="10-d1">
            <specification>A local constant float value</specification>
            <realisation>
              <dataType xsi:type="Float">
                <init value="3.14159265" />
              </dataType>
            </realisation>
          </constant>

          <variable name="v" id="10-d2">
            <specification>A local variable of yet unknown type</specification>
          </variable>

        </declarations>
        <flow />
      </realisation>
    </procedure>
  </procedures>
</otx>
```

7.13 Nodes

7.13.1 Overview

OTX nodes represent the single steps in the program flow of a procedure. They appear in **Flow**-type elements, see 7.16.9.

Nodes can be atomic or compound.

— Atomic nodes[5] are single steps containing no further embedded flows. At runtime, atomic nodes should be executed as one monolithic unit. The following node types are atomic: **Return**, **Continue**, **Break**, **TerminateLanes**, **Throw**, **ExtensibleEndNode**. In general, **Action** nodes are also atomic, but there are exemptions depending on the **ActionRealisation** type (e.g. the **ProcedureCall** action which is not atomic).

— Compound nodes[6] are steps that embed one or more flow(s) recursively (**Node** types **Group**, **Loop**, **Branch**, **Parallel**, **MutexGroup**, **ExtensibleCompoundNode**, and **Handler**). To execute these compound nodes, their inner flow(s) shall be interpreted step by step as well.

In order to reflect the above distinction in the data model, the inheritance hierarchy of OTX nodes shown in Figure 22 has been implemented: there is an abstract base type **Node** which contains all features common to all nodes, refer to the **Node** specification in 7.13.2 for details. All compound nodes are derived from the abstract type **CompoundNode**. All other nodes are in general atomic nodes (implicitly). Furthermore, there is an abstract sub-classification **EndNode** for nodes that can only be used at the end of a node sequence, refer to the specification of all **EndNode** types in 7.13.4.10. The special role of the **Action** node is specified in 7.13.3.

5) OTX atomic nodes can be seen as the equivalent to one-line statements in line-based programming languages.

6) OTX compound nodes are the equivalent to control statements that stretch over more than one line in text-based programming languages. Those statements usually enclose an inner block of statements which is often introduced by "{" and closed by "}" brackets.

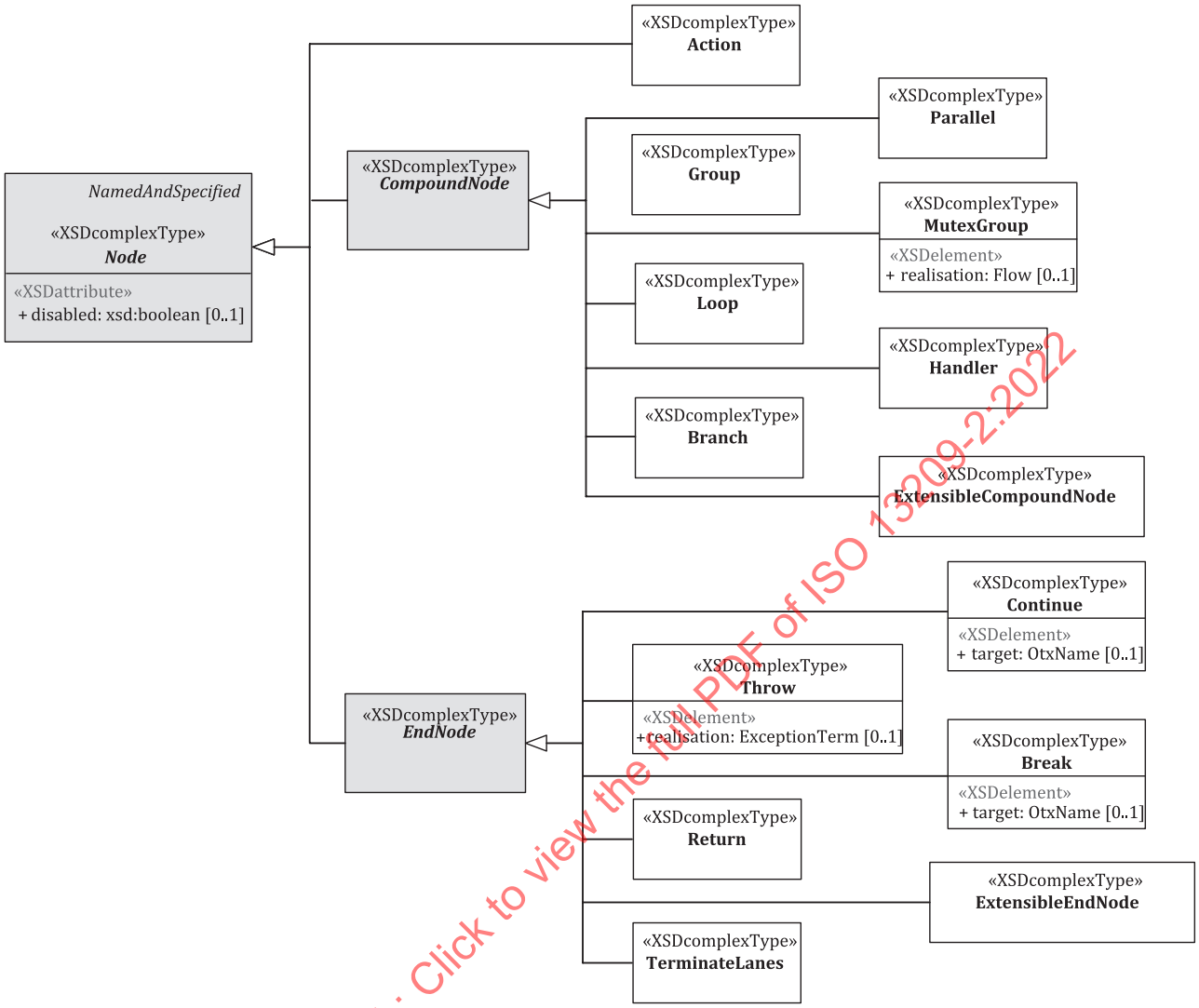

Figure 22 — Data model view: Node inheritance hierarchy

7.13.2 Node

7.13.2.1 Description

This type serves as the abstract base for all **`Node`** types. The properties of the type are therefore part for all derived nodes.

7.13.2.2 Syntax

The syntax of the abstract type **`Node`** is shown in Figure 22.

7.13.2.3 Semantics

`Node` is based on **`NamedAndSpecified`**. The properties of the **`Node`** type have the following semantics:

— **`id: OtxId [1]`** (derived from **`NamedAndSpecified`**, see 7.16.4)

This represents a node's id. It shall be unique among all other ids in a document. Please refer to 7.16.4 for details concerning ids in OTX documents.

— **name: OtxName [0..1]** (derived from **NamedAndSpecified**, see 7.16.4)

The attribute represents the human readable counterpart to the **id** attribute. It shall serve authors to identify nodes easily by name on the surface of a graphical OTX editor. There are no constraints concerning uniqueness of the name attribute values; nonetheless uniqueness is recommended, but not enforced.

Associated checker rules:

— Core_Chk010 – unique node names (see C.2.10).

— **disabled: xsd:boolean = {false|true} [0..1]**

By using this attribute a node can be switched-on and -off: a node with **disabled = "true"** shall be ignored at runtime; otherwise it shall be executed normally[7]. If the attribute is not set, the node is enabled (default is **disabled = "false"**).

NOTE OTX XSD does not allow nodes after an **EndNode** regardless if it is disabled or enabled (see 7.16.9).

Associated checker rules:

— Core_Chk011 – no disabled nodes (see C.2.11).

— **<metaData>: MetaData [0..1]** (derived from **NamedAndSpecified**, see 7.16.4)

This optional element is used for adding metadata to a node. For details on the **MetaData** type, refer to 7.16.5.

— **<specification> : xsd:string [0..1]** (derived from **NamedAndSpecified**, see 7.16.4)

The simple **xsd:string** content of this optional element shall be used to specify the purpose of the node for the human reader. The complement of **<specification>** is the element **<realisation>** which is not specified for the **Node** type, because it is specific for each distinct node type (see below).

Associated checker rules:

— Core_Chk007 – have specification if no realisation exists (see C.2.7)-

7.13.3 Action node

7.13.3.1 Description

IMPORTANT — Action nodes play a central role concerning the extensibility of the OTX data model. The extension mechanism is described in Annex D.

An **Action** node is a simple node in a flow (see "single-line statement"). The runtime behaviour of an **Action** node can be customized for different contexts by applying the validities concept (see 6.8). A context-specific behaviour can be configured by choosing one out of a list of available **ActionRealisations** defined by the OTX core or any OTX extension. The comprehensive specification of all core **ActionRealisations** can be found in 7.14. **Action** nodes are in general to be treated as atomic, but there are exemptions depending on the **ActionRealisation** type. Exemptions are especially emphasized in the specification for the particular type of **ActionRealisation**.

7.13.3.2 Syntax

Figure 23 shows the syntax of the **Action** node type.

7) This is comparable to commenting out lines or blocks of source code in text-based programming languages.

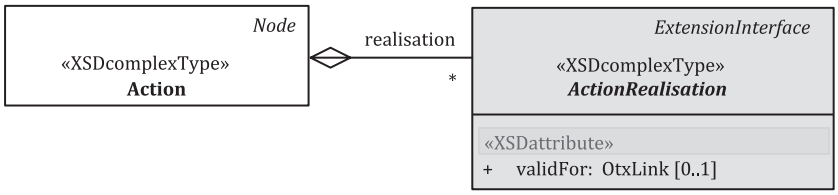

Figure 23 — Data model view: `Action`

7.13.3.3 Semantics

`Action` is a **`Node`** which is **`NamedAndSpecified`**. The semantics of all derived properties are described in 7.13.2. Only the specific semantic properties of **`Action`** are described here.

The properties of **`Action`** have the following semantics:

— **`realisation: ActionRealisation [0..*]`**

This is the counterpart of the **`<specification>`** element for an **`Action`** node.

IMPORTANT — As long as no `<realisation>` elements are set, an `Action` node has no runtime semantics which means that it shall be skipped during execution.

In an action node, multiple **`<realisation>`** elements are allowed. At runtime, the validity of each **`<realisation>`** shall be evaluated one by one (in the order of appearance in the OTX document). The first valid element found shall be executed.

The precise behaviour of the finally executed **`<realisation>`** element is determined by the value of the **`xsi:type`** attribute: Any complex type extending **`ActionRealisation`** can be inserted here, e.g. the types **`ProcedureCall`**, **`Assignment`** or **`ListInsertItems`**, just to name a few.

The behaviour of all OTX core **`ActionRealisations`** is described in 7.14.

— **`validFor: OtxLink [0..1]`**

An **`ActionRealisation`** may be connected to a **`Boolean`** context variable, a global **`Boolean`** constant or a validity term (by **`OtxLink`**). Analysing the **`validFor`** attribute of a **`<realisation>`** element in an **`Action`** node determines whether the **`<realisation>`** is valid or not. Here, the following rules apply:

— if there is no **`validFor`** attribute, the **`<realisation>`** is valid;

— if there is a **`validFor`** attribute, the **`Boolean`** context variable, constant or the validity term referenced by the **`OtxLink`** shall be evaluated. The **`<realisation>`** is valid if and only if the result is **`true`**.

Associated checker rules:

— Core_Chk012 – no unreachable realisations in Action and Group nodes (see C.2.12);

— Core_Chk053 – no dangling OtxLink associations (see C.2.53);

— Core_Chk013 – correct referencing of validities (see C.2.13).

Figure 24 shows a behavioural description of the **`Action`** node. The examples in 6.8 show the benefits of the static validity concept which sets itself apart from the dynamic branch node, even if the behaviour shown in Figure 24 resembles the branch node behaviour to a great extent.

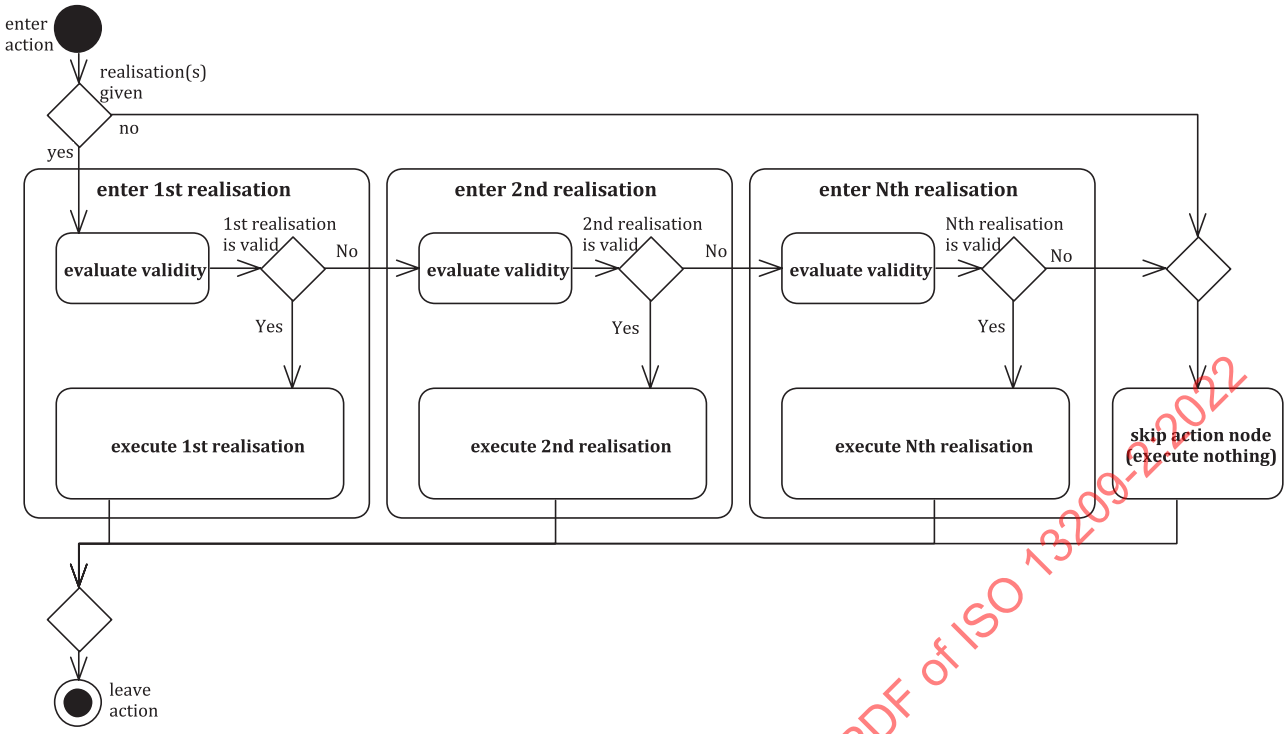

Figure 24 — Behavioural description: `Action`

7.13.3.4 Example

The example below shows context variables and validity terms which are defined in a central document named “Validities.otx” and another document named “ActionExample.otx” using the validity terms in an **`Action`** node with multiple **`<realisation>`** elements. With this setup, the one-time defined context variables and validity terms can be reused by any other OTX document that imports this document.

“ActionExample.otx” uses action realisations which are defined by the OTX extension schemas HMI, DiagCom and Logging (see ISO 13209-3[2]). They are introduced to the root of the document by the namespace definition attributes **`xmlns:hmi`**, **`xmlns:diag`**, **`xmlns:log`** together with the respective fully qualified namespace strings (see 7.3).

Sample of Validities

```
<?xml version="1.0" encoding="UTF-8"?>
<otx xmlns="http://iso.org/OTX/1.0.0" id="11"
  name="Validities"
  package="org.iso.otx.examples"
  version="1.0"
  timestamp="2009-10-20T14:40:10"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <specification>
    Central validity term and context variable definition document,
    use validity terms and context variables defined here.
  </specification>

  <declarations>
    <context name="DEBUG_MODE" id="11-d1">
      <specification>
        Reflects the current debug setting: true: debugging is on. false: debugging is off.
      </specification>
      <realisation>
        <dataType xsi:type="Boolean">
          <init value="false" />
        </dataType>
      </realisation>
    </context>
    <context name="MODEL" id="11-d2">
      <specification>Reflects the currently diagnosed vehicle's model</specification>
      <realisation>
        <dataType xsi:type="String" />
      </realisation>
    </context>
  </declarations>

  <validities>
    <validity name="isModernModel" id="11-v1">
      <specification>Valid if context variable MODEL=="ModernCar"</specification>
      <realisation xsi:type="IsEqual">
        <term xsi:type="StringValue" valueOf="MODEL" />
        <term xsi:type="StringLiteral" value="ModernCar" />
      </realisation>
    </validity>
    <validity name="isVintageModel" id="11-v2">
      <specification>Valid if context variable MODEL=="VintageCar"</specification>
      <realisation xsi:type="IsEqual">
        <term xsi:type="StringValue" valueOf="MODEL" />
        <term xsi:type="StringLiteral" value="VintageCar" />
      </realisation>
    </validity>
  </validities>
</otx>
```

To get access to the declarations in "Validities.otx", "ActionExample.otx" needs to import it by using an **<import>** element. All visible global scope identifiers of the imported document (here, these are context variables and validity terms) are then accessible by using the freely chosen prefix **val**.

The first **Action** node in "ActionExample.otx" writes a log message if the Boolean context variable **"DEBUG_MODE"** is **true**. The **<realisation>** uses the **WriteLog** action realisation from the OTX Environment extension.

The second **Action** node reads the battery voltage. Whether the voltage is read by manual input or automatically by a diagnostic service depends on the current vehicle-related context.

— If the context variable **MODEL** equals **"VintageModel"**, then the validity term **"isVintageModel"** holds and the first **<realisation>** of the **Action** node is executed. The **<realisation>** uses the **InputDialog** action realisation from the OTX HMI extension which allows the mechanic to input a measured voltage value (details for **InputDialog** are not given here).

— If the context variable **MODEL** equals **"ModernModel"**, then the validity term **"isModernModel"** holds and the second **<realisation>** of the **Action** node is executed. The **<realisation>** uses the **ExecuteDiagnosticService** action realisation from the OTX DiagCom extension to read the voltage automatically from the vehicle.

— If none of the validity terms hold, no action is performed.

Sample of Actions

```
<?xml version="1.0" encoding="UTF-8"?>
<otx xmlns="http://iso.org/OTX/1.0.0" id="12"
  name="ActionExample"
  package="org.iso.otx.examples"
  version="1.0"
  timestamp="2009-10-20T14:40:10"
  xmlns:diag="http://iso.org/OTX/1.0.0/DiagCom"
  xmlns:hmi="http://iso.org/OTX/1.0.0/HMI"
  xmlns:log="http://iso.org/OTX/1.0.0/Logging"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <imports>
    <import prefix="val" package="org.iso.otx.examples" document="Validities"/>
  </imports>

  <procedures>
    <procedure name="main" visibility="PUBLIC" id="11-p1">
      <specification>Check battery voltage of vehicle</specification>
      <realisation>
        <declarations>
          <variable name="voltage" id="11-d1">
            <specification>Variable for the measured voltage</specification>
            <realisation>
              <dataType xsi:type="Float"/>
            </realisation>
          </variable>
        </declarations>

        <flow>
          <action id="a1">
            <specification>Write LogMessage (debug only)</specification>
            <realisation validFor="val:DEBUG_MODE" xsi:type="log:WriteLog">
              <!-- ActionRealisation details -->
            </realisation>
          </action>

          <action id="a2">
            <specification>Read battery voltage</specification>

            <!-- For the vintage car: -->
            <!-- This will ask the mechanic to enter the measured voltage manually -->
            <realisation validFor="val:isVintageCar" xsi:type="hmi:InputDialog">
              <!-- ActionRealisation details -->
            </realisation>

            <!-- For the modern car: -->
            <!-- This will read the voltage by using a diagnostic service -->
            <realisation validFor="val:isModernCar" xsi:type="diag:ExecuteDiagService">
              <!-- ActionRealisation details -->
            </realisation>
          </action>

          <branch id="b1">
            <specification>Show OK/NOK message depending on voltage level read</specification>
            <realisation>
              <!-- branch according to measured voltage value -->
            </realisation>
          </branch>
        </flow>
      </realisation>
    </procedure>
  </procedures>
</otx>
```

7.13.4 Compound nodes

7.13.4.1 Overview

Nodes derived from the abstract **CompoundNode** type contain one or more nested **Flow**-type elements. The set of all such nodes represents the control structures of OTX. The following subclauses specify each of the **CompoundNode** subtypes shown in Figure 22.

NOTE The abstract and empty **CompoundNode** type itself needs no further specification because it serves for logical data model structuring only.

7.13.4.2 Group node

7.13.4.2.1 Description

The simplest of all compound nodes is the **Group** node. It can be used to wrap a sequence of nodes together. Nodes contained in a **Group** form a higher-level unit that can be treated as one logical block by authors. This makes sense especially for nodes that belong together logically; grouping allows the author to demonstrate this togetherness and to provide clarity through modular sequence design.

Like the **Action** node (see 7.13.3), the runtime behaviour of a **Group** node can be customized for different contexts by applying the validities concept (see 6.8). Context-specific behaviour can be configured by implementing group flows for each anticipated context situation.

NOTE In a graphical OTX authoring environment, groups can be used for folding and unfolding parts of a procedure in order to create views with different levels of detail; it can be used for refactoring tasks as well, e.g. making a loop out of a group or extracting a procedure based on a group.

7.13.4.2.2 Syntax

Figure 25 shows the syntax of the **Group** node type.

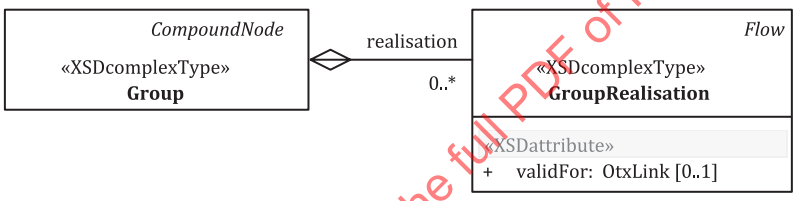

Figure 25 — Data model view: `Group`

7.13.4.2.3 Semantics

Group is a **CompoundNode** which is a **Node** which is **NamedAndSpecified**. The semantics of all derived properties are described in 7.13.2. Only the specific properties of the **Group** type are described here, with the following semantics:

— **<realisation>: GroupRealisation [0..*]**

This optional element is the counterpart to the **<specification>** element described for the **Node** type. Since **GroupRealisation** is derived from the **Flow** type (see 7.16.9), the **<realisation>** element is a container for a sequence of nodes which shall be executed one by one in the order of appearance.

— **validFor: OtxLink [0..1]**

A **GroupRealisation** can be connected to a **Boolean** context variable or a validity term (by **OtxLink**). Analysing the **validFor** attribute of a **<realisation>** element in a **Group** node determines whether the **<realisation>** is valid or not:

— If there is no **validFor** attribute, the **<realisation>** is valid.

— If there is a **validFor** attribute, the **Boolean** context variable, global **Boolean** constant or the validity term referenced by the **OtxLink** shall be evaluated. The **<realisation>** is valid if and only if the result is **true**.

Associated checker rules:

— Core_Chk012 – no unreachable realisations in Action and Group nodes (see C.2.12);

— Core_Chk053 – no dangling OtxLink associations (see C.2.53);

— Core_Chk013 – correct referencing of validities (see C.2.13).

The context-specific behaviour of Group nodes with multiple **<realisation>** elements is analogous to the **Action** node behaviour which is shown in Figure 24.

7.13.4.2.4 Example

The example below shows a procedure-flow level group (**id = "group1"**) with three nodes in it: an **Action** node (**id = "action1"**), an embedded **Group** node (**id = "group2"**) and a **Return** node (**id = "ret1"**). The embedded group itself contains a single **Action** node (**id = "action2"**). In this example, **Group** nodes are not context-dependent (no **validFor** attribute).

Sample of Groups

```
<?xml version="1.0" encoding="UTF-8"?>
<otx xmlns="http://iso.org/OTX/1.0.0" id="13"
  name="GroupExample"
  package="org.iso.otx.examples"
  version="1.0"
  timestamp="2009-10-20T14:40:10"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <procedures>
    <procedure name="main" visibility="PUBLIC" id="13-p1">
      <specification>Demonstration of nested flows</specification>
      <realisation>
        <flow>

          <group id="group1">
            <specification>A group containing a group</specification>
            <realisation>
              <action id="action1" />
              <group id="group2">
                <specification>Inner group</specification>
                <realisation>
                  <action id="action2" />
                </realisation>
              </group>
              <return id="ret1" />
            </realisation>
          </group>

          <action id="action3" />
        </flow>
      </realisation>
    </procedure>
  </procedures>
</otx>
```

The example also demonstrates the use of the **Flow** type which occurs with two different names here: the **<flow>** element of the procedure and the **<realisation>** elements of the groups.

7.13.4.3 Loop node

7.13.4.3.1 Description

For repetitive execution of a sequence of nodes, the **Loop** node shall be utilized. OTX supports several loop configurations which are commonly called (do-)while-, for- and for-each-loop.

7.13.4.3.2 Syntax

Figure 26 shows the syntax of the **Loop** node. The syntax of the significant sub-element **<configuration>** (which allows choosing and configuring different loop types) is depicted in Figure 27.

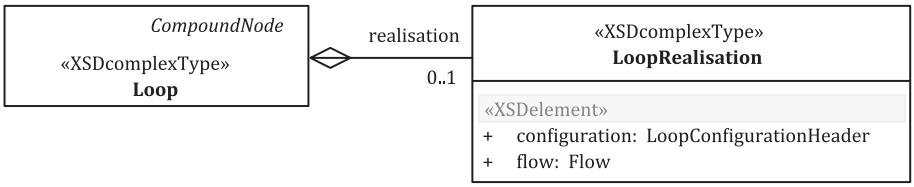

Figure 26 — Data model view: `Loop`

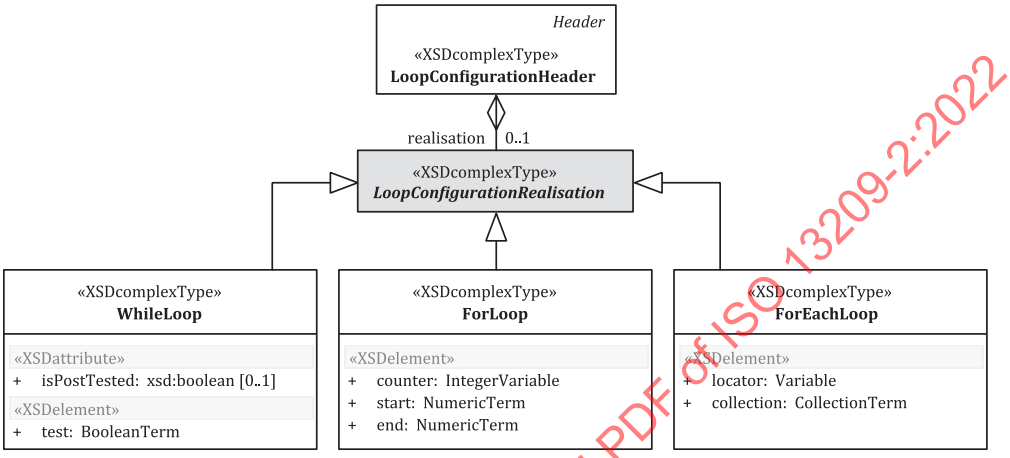

Figure 27 — Data model view: `LoopConfigurationHeader`

7.13.4.3.3 Semantics

`Loop` is a **`CompoundNode`** which is a **`Node`** which is **`NamedAndSpecified`**. The semantics of all derived properties are described in 7.13.2. Only the specific properties of the **`Loop`** type are described here.

IMPORTANT — `Continue` and `Break` nodes have a special meaning for loops, as specified in 7.13.5.3 (`Continue`) and 7.13.5.4 (`Break`). Since these nodes specify their target by using the targeted loop's name, the `name` attribute is mandatory for loops (unlike the base type `NamedAndSpecified` where it is optional). This is enforced by `<xsd:key>` constraints specified in the OTX schema.

`Loop` nodes shall be configured in order to define the desired type of loop, (do-)while-, for- and for-each-loop. For configuring the **`LoopConfigurationHeader`** type is used, which is described separately in the following sections.

The top-level properties of the **`Loop`** type have the following semantics:

— **`<realisation>: LoopRealisation [0..1]`**

This optional element is the counterpart to the **`<specification>`** element described for the **`Node`** type. In the **`<realisation>`** parts of a loop, a **`<configuration>`** and a **`<flow>`** are indispensable:

— **`<configuration>: LoopConfigurationHeader [1]`**

Represents the loop header which defines the loop configuration type (plus the derived properties of the **`Header`** base type, see 7.13.4.9).

— **`<realisation>: LoopConfigurationRealisation [1]`**

This element determines the loop configuration type of the **`Loop`** node. Since **`LoopConfigurationRealisation`** is an abstract base type for the different loop configuration types, the **`xsi:type`** attribute shall be used to choose the type of interest. Semantics of the individual loop types are described below.

— `<flow>: Flow [1]`

This element is a container for a sequence of nodes which shall be executed one by one in the order of appearance. For details about the `Flow` type, see 7.16.9.

Semantics of the `WhileLoop` configuration

The (do-)while-loop is a generic loop with a test that controls repetition of the included flow. The test is a Boolean term. As long as the test condition holds, the loop flow is executed repeatedly. In OTX, it can be configured whether the test shall be checked before (while-loop) or after the first flow execution (do-while-loop).

The `WhileLoop` configuration is derived from `LoopConfigurationRealisation` and has the following semantic properties:

— `isPostTested: xsd:boolean = {false|true} [0..1]`

This optional attribute controls whether the condition for the next loop shall be tested prior to the next loop iteration start (the default) or after finishing the loop flow. Note that flows in post tested loops will always be executed at least once.

— `<test>: BooleanTerm [0..1]`

This contains the Boolean term that shall be evaluated prior to executing the loop flow. Syntax and semantics of terms like `BooleanTerm` are specified in 7.15.

Figure 28 illustrates the runtime behaviour of pre-tested while- and post-tested do-while- loops.

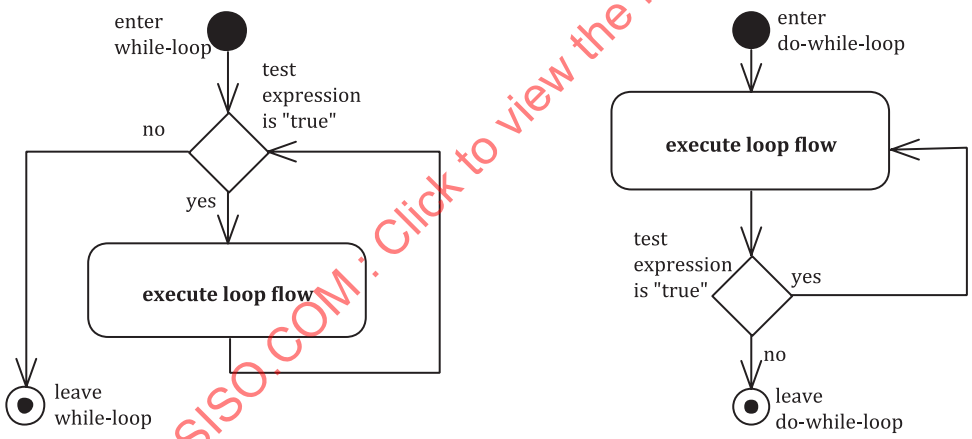

Figure 28 — Behavioural description: while- and do-while-loop

Semantics of the `ForLoop` configuration

The OTX for-loop allows configuring a start and an end value (integer terms). When the loop starts, the start value is assigned to the counter (an integer). Prior to each iteration, the counter value is compared to the end value. If the counter value exceeds the end value, the loop exits, otherwise the iteration is carried out. After each iteration—also after the final one—the counter is incremented by one. The counter value can be accessed and be used for any kind of computation in the loop flow.

IMPORTANT — Since a counter is visible procedure-wide in OTX, it is accessible on the outside of the loop as well.

IMPORTANT — When entering a for-loop, the old counter value will be overwritten implicitly by the start value. After finishing the final iteration normally (not by break), the counter value will be incremented one more time. This value is the exit value visible to the follower nodes of the loop node.

NOTE 1 In contrast to the for-each-loop behaviour which throws a `ConcurrentModificationException` if its collection is changed during loop execution, it is valid to change a for-loop's counter value while the for-loop is active.

The `ForLoop` configuration is derived from `LoopConfigurationRealisation` and has the following semantic properties:

— `<counter>: IntegerVariable [1]`

This element identifies an `Integer` variable that shall be used as the counter of the for-loop.

— `<start>: NumericTerm [1]`

The value of the `NumericTerm` given by `<start>` shall be evaluated once when entering the for-loop. It represents the start value which is assigned to the counter (see above). `Float` values shall be truncated.

— `<end>: NumericTerm [1]`

The value of the `NumericTerm` given by `<end>` shall be evaluated once when entering the for-loop. It represents the end value which is compared to the counter value prior to each iteration. `Float` values shall be truncated.

See the activity diagrams in Figure 29 for the runtime behaviour of for-loops.

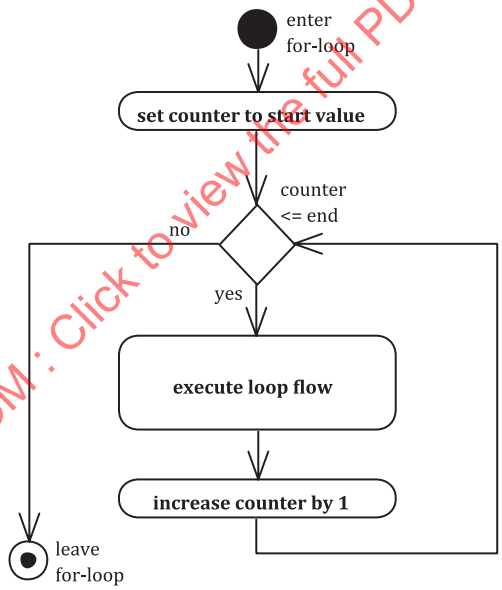

Figure 29 — Behavioural description: for-loop

IMPORTANT — The counter variable shall be increased by 1 at the end of each loop iteration even if terminated by a `Continue` node. Only in the case that a loop iteration is terminated by a `Break` node, the counter variable shall not be increased before leaving the loop.

Semantics of the `ForEachLoop` configuration

The OTX for-each-loop is configured by a collection (a list or a map) and a so-called locator and value variables. The collection is expressed by a list or map expression which shall be evaluated initially. The flow of the loop shall be executed repeatedly for each item found in the collection. The current item of an iteration can be accessed by using the locator variable which contains the index or key of the list or map item, respectively. Furthermore, the current item can be accessed by using the value variable, which contains the the list item or map value. Like this, the current item can be used for any kind of computation in the loop flow.

IMPORTANT — If the collection is a `List`, there is a defined order for the iterations: the loop starts with the first `List` item (index 0), followed the second item and so on. By contrast, there is no particular order defined for `Map`.

IMPORTANT — Since the locator and value variables are visible procedure-wide, they are accessible on the outside of the loop as well.

IMPORTANT — When entering a for-each-loop, the old value of the locator variable shall be overwritten implicitly by the index/key of the first item of the list/map. Similarly, the old value of the value variable shall be overwritten implicitly by the first item/value. After leaving the loop, the variable values of the final iteration will be visible to the follower nodes of the loop node.

IMPORTANT — Actions derived from `ListModifier` and `MapModifier` are collection modifiers. Since they change a collection's length (`List` case) or size (`Map` case), applying such modifications on a for-each-loop's collection during loop execution may cause serious inconsistencies (e.g. when a collection item is accessed which does not exist anymore). Therefore, OTX runtimes shall throw `ConcurrentModificationException` when such modifications occur.

IMPORTANT — By contrast, changing the value of items of a for-each-loop's collection does not represent a concurrent modification and shall not cause a `ConcurrentModificationException`.

NOTE 2 When catching a **`ConcurrentModificationException`**, it is possible that the data in a for-each-loop's collection is already corrupt (e.g. when a **`List`** item has been removed). In this case, it is not possible for exception handling to restore the original collection.

The **`ForEachLoop`** configuration is derived from **`LoopConfigurationRealisation`** and has the following semantic properties:

— **`<locator>: Variable [0..1]`**

  This element points to an **`Integer`** or a **`String`** that shall be used as the locator variable of the for-each-loop.

— **`<value>: Variable [0..1]`**

  This element points to a variable that shall hold the list items or map values.

— **`<collection>: CollectionTerm [1]`**

  The term value expressed by **`<collection>`** represents the List or Map over which the for-each-loop shall iterate.

Associated checker rules:

— Core_Chk014 – correct locator variable type in for-each-loop (see C.2.14);

— Core_Chk056 – no modification of collection inside foreach-loops (see C.2.56);

— Core_Chk063 – locator or value in for-each-loop (see C.2.63);

— Core_Chk064 – correct value variable type in for-each-loop (see C.2.64).

Throws:

— **`ConcurrentModificationException`**

  It is thrown in case that the collection is modified while the loop is active. Modifications can be triggered by actions inside of the loop or from the outside, e.g. by actions in a parallel lane.

See the activity diagram in Figure 30 for the runtime behaviour of for-each-loops.

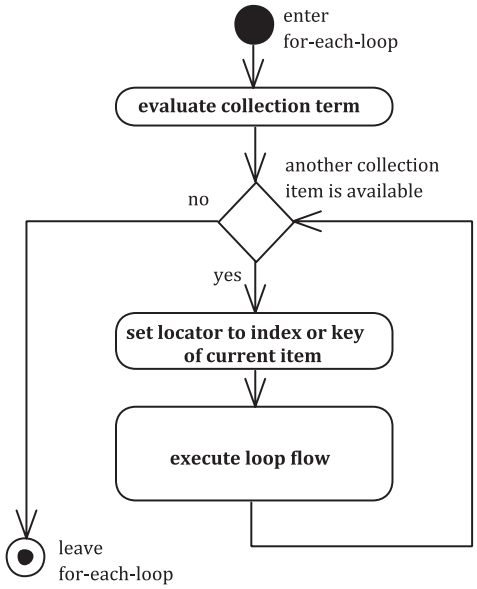

Figure 30 — Behavioural description: for-each-loop

7.13.4.3.4 Example

The examples below demonstrate a while-, for- and for-each-loop. Example 1 shows an endless while loop (the loop condition is always `true`). Example 2 shows a for-loop which counts from 1 to 10. In each iteration, a dialog is shown containing the current loop iteration number. Example 3 shows a for-each-loop which iterates over a list of values. The value of the current iterations list item is shown in a dialog.

Sample of while-loop

```
<loop id="loop1" name="myWhileLoop">
  <specification>An endless while loop</specification>
  <realisation>
    <configuration id="loop1config">
      <specification>Always true loop condition</specification>
      <realisation xsi:type="WhileLoop">
        <test xsi:type="BooleanLiteral" value="true" />
      </realisation>
    </configuration>
    <flow>
      <action id="a1" />
      <action id="a2" />
    </flow>
  </realisation>
</loop>
```

Sample for-loop

```
<loop id="loop2" name="myForLoop">
  <specification>Loop repeats an action 10 times</specification>
  <realisation>
    <configuration id="loop2config">
      <specification>Start at 1, end at 10, counter variable is i</specification>
      <realisation xsi:type="ForLoop">
        <counter xsi:type="IntegerVariable" name="i" />
        <start xsi:type="IntegerLiteral" value="1" />
        <end xsi:type="IntegerLiteral" value="10" />
      </realisation>
    </configuration>
    <flow>
      <action id="a3">
        <specification>Shows the iteration number i</specification>
        <realisation xsi:type="hmi:ConfirmDialog" messageType="INFO">
          <hmi:message xsi:type="ToString" >
            <term xsi:type="IntegerValue" valueOf="i"/>
          </hmi:message>
        </realisation>
      </action>
    </flow>
  </realisation>
</loop>
```

Sample for-each-loop

```
<loop id="loop3" name="myForEachLoop">
  <specification>Loop iterates over all items of list L and shows them</specification>
  <realisation>
    <configuration id="loop3config">
      <specification>Collection is L, locator variable is index</specification>
      <realisation xsi:type="ForEachLoop">
        <locator xsi:type="IntegerVariable" name="index" />
        <collection xsi:type="ListValue" valueOf="L" />
      </realisation>
    </configuration>
    <flow>
      <action id="a4">
        <specification>Show the value of current List item by using the iterator</specification>
        <realisation xsi:type="hmi:ConfirmDialog" messageType="INFO">
          <hmi:message xsi:type="StringValue"
            valueOf="L">
            <path>
              <stepByIndex xsi:type="IntegerValue" valueOf="index" />
            </path>
          </hmi:message>
        </realisation>
      </action>
    </flow>
  </realisation>
</loop>
```

7.13.4.4 Branch node

7.13.4.4.1 Description

For conditional execution of flows, **Branch** nodes are used. With a **Branch** node, a series of one or more condition/flow pairs can be defined. The conditions are evaluated one after the other, in the order of appearance in the document. The flow of the first **true** condition found will be executed. If no condition holds, an optional unconditional flow will be executed, if given. The **Branch** construct is commonly referred to as "if..elseif..else"-statement.

7.13.4.4.2 Syntax

Figure 31 shows the syntax of the **Branch** node type.

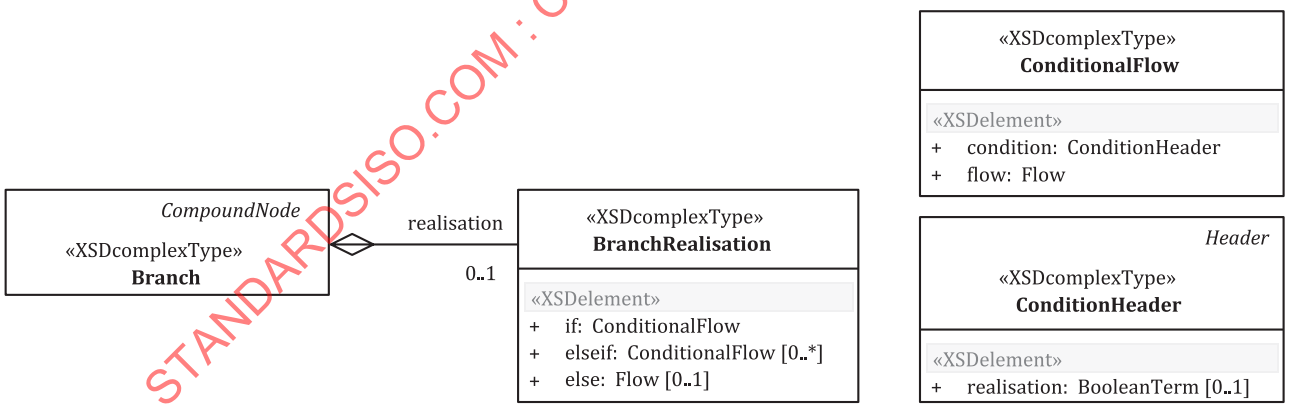

Figure 31 — Data model view: `Branch`

7.13.4.4.3 Semantics

Branch is a **CompoundNode** which is a **Node** which is **NamedAndSpecified**. The semantics of all derived properties are described in 7.13.2. Only the specific semantic properties of **Branch** are described here.

— **<realisation>: BranchRealisation [0..1]**

  Contains one **<if>** element followed by an optional list of **<elseif>** elements and an optional **<else>** element:

— **<if>: ConditionalFlow [1]**

This is the only mandatory part of the **Branch** node. At runtime, its **<condition>** shall always be the first to be tested, prior to any **<elseif>** conditions. If the condition holds, its **<flow>** shall be executed. Otherwise, any trailing **<elseif>** or **<else>** elements shall be processed.

— **<condition>: ConditionHeader [1]**

The **<realisation>** of the element is a **BooleanTerm** which represents the test to be carried out prior to flow execution. For additional properties from the **Header** base type, see 7.13.4.9.

NOTE In graphical representation, the condition header can be shown by a "diamond" shape.

— **<flow>: Flow [1]**

The flow to be carried out when the condition holds.

— **<elseif>: ConditionalFlow [0..*]**

If there is more than one conditional flow to be defined, a list of **<elseif>** elements shall be used.

At runtime, the **<condition>** of each **<elseif>** is tested one by one, after the **<if>** part has been processed. The **<flow>** of the first **<elseif>** with a **true** condition shall be executed. Further semantics are identical to the **<if>** element (see above).

— **<else>: Flow [0..1]**

If none of the **<if>** or **<elseif>** elements could be executed, the **<else>** flow shall apply. In cases where no **<else>** exists, the whole **<branch>** node is skipped.

Figure 32 shows an activity diagram describing the runtime behaviour of a **Branch**.

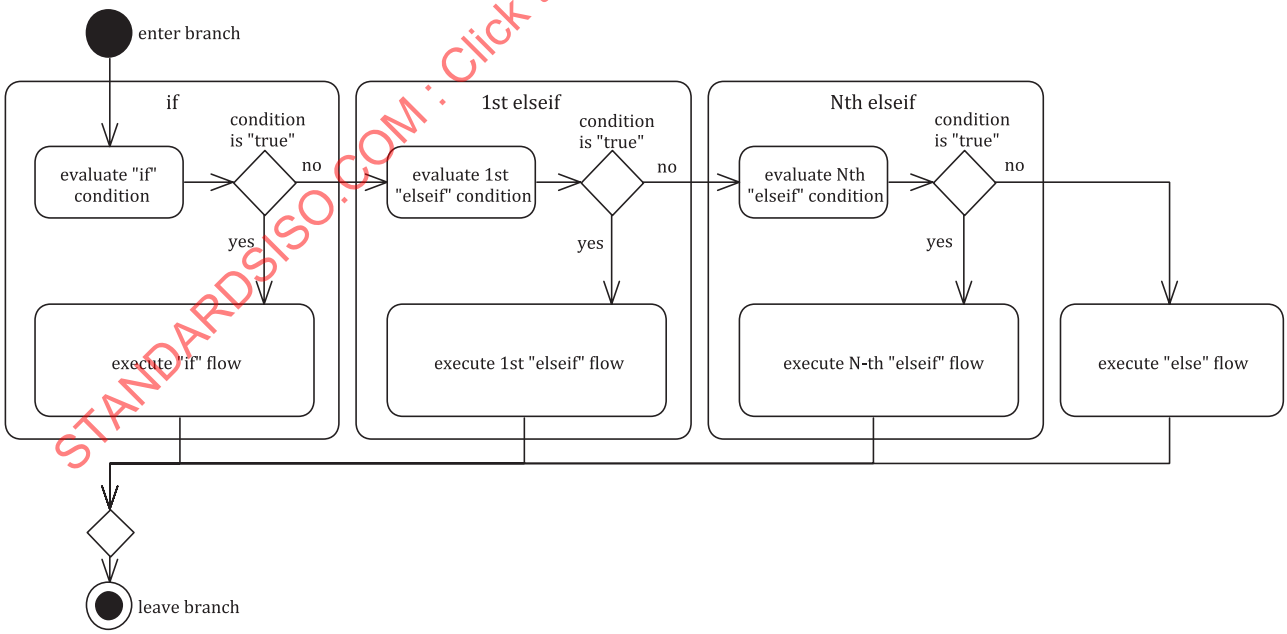

Figure 32 — Behavioural description: Branch

7.13.4.4.4 Example

The example below shows a simple **Branch** node.

Sample of Branches

```
<branch id="branch1">
  <specification>if-elseif-else branch</specification>
  <realisation>
    <if>
      <condition id="ifcond">
        <specification>Unreachable flow (condition always false)</specification>
        <realisation xsi:type="BooleanLiteral" value="false" />
      </condition>
      <flow>
        <action id="a1" />
      </flow>
    </if>
    <elseif>
      <condition id="elseif1cond">
        <specification>unspecified elseif condition</specification>
      </condition>
      <flow>
        <action id="a2" />
      </flow>
    </elseif>
    <else>
      <action id="a3" />
    </else>
  </realisation>
</branch>
```

7.13.4.5 Parallel node

7.13.4.5.1 Description

A parallel node consists of two or more flows (lanes) that shall be executed simultaneously and synchronously. Optionally nested parallel lanes can be supported.

7.13.4.5.2 Syntax

Figure 33 shows the syntax of the **`Parallel`** node type.

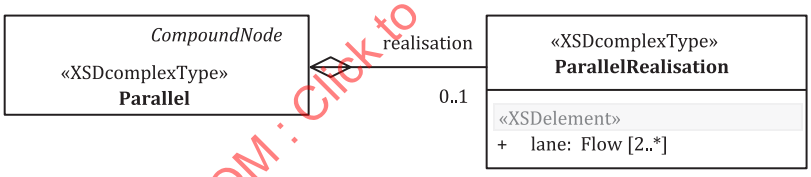

Figure 33 — Data model view: `Parallel`

7.13.4.5.3 Semantics

`Parallel` is a **`CompoundNode`** which is a **`Node`** which is **`NamedAndSpecified`**. The semantics of all derived properties are described in 7.13.2. Only the specific semantic properties of **`Parallel`** are described here.

— **`<realisation>: ParallelRealisation [0..1]`**

Contains at least two **`<lane>`** elements. All given lanes shall be executed in parallel.

— **`<lane>: Flow [1..*]`**

Every lane is a flow of nodes. See 7.16.9 for a detailed description of the **`Flow`** type.

Associated checker rules:

— Core_Chk015 – correct nesting of Parallel nodes (see C.2.15).

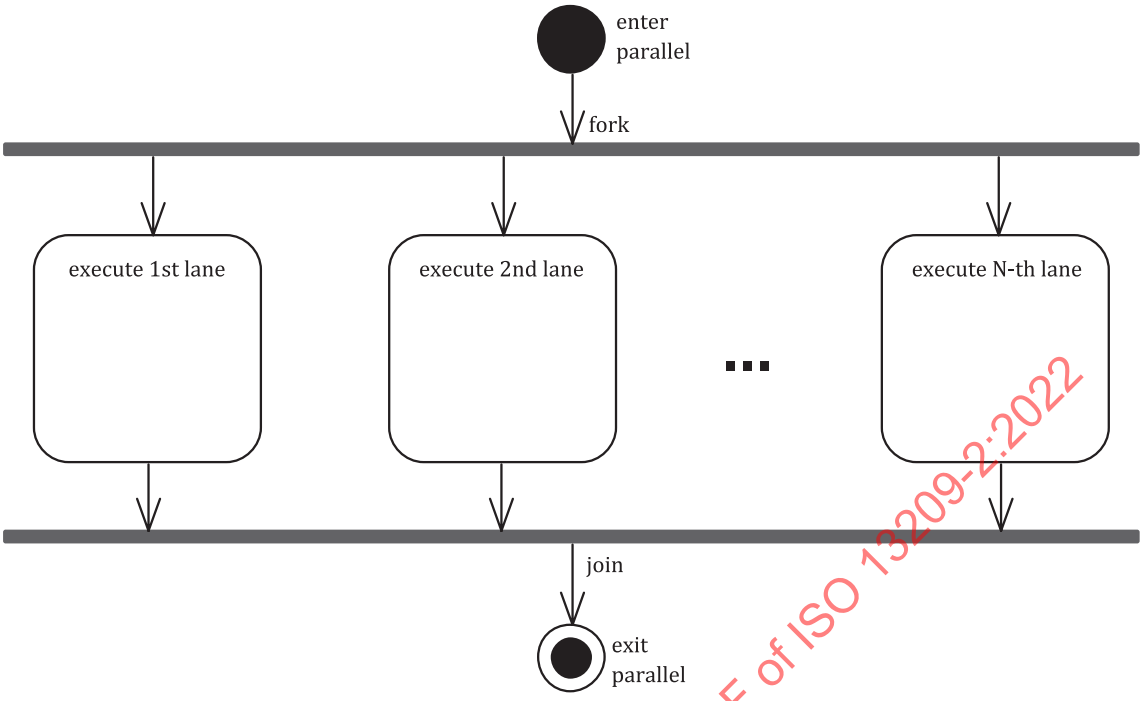

Figure 34 — Behavioural description: `Parallel`

Figure 34 shows an activity diagram describing the runtime behaviour of a **`Parallel`** node. The lanes after the fork shall be executed simultaneously. Since the **`Parallel`** node's normal runtime behaviour shall be synchronized, a **`Parallel`** node shall not complete until all of its lanes have completed.

CAUTION — The relative timing of node execution in different lanes is not determined – race conditions may occur. For that reason one lane should not rely on the execution of any other lane in the `Parallel` node. In order to gain thread safety, authors should use `MutexGroup` nodes (see 7.13.4.6).

There are several conditions which induce special completion rules for the **`Parallel`** node. The rules shall be applied when a **`Return`** node, a **`TerminateLanes`** node or an exception occurs in the lanes.

— **TerminateLanes node:**

This special end node induces a special behaviour: if a **`TerminateLanes`** node is executed in one of the lanes, all other lanes are signalled to complete prematurely.

IMPORTANT — On receiving a premature completion signal, all lanes shall complete immediately after completion of the last atomic node, `MutexGroup` or finally block (if any). Atomic nodes are `Action` nodes with an atomic `ActionRealisation` as well as the end nodes `Return`, `Break`, `Continue`, `TerminateLanes` and `Throw`. Since there is no completion timeout defined for atomic nodes, finally blocks or `MutexGroup` nodes, lane completion is delayed no matter how long the node takes to complete. This avoids producing undefined states and implies possibilities of deadlocks. An OTX author is responsible for avoiding deadlock situations.

IMPORTANT — Terminating a called `Procedure` using `TerminateLanes` is not recommended because the behaviour may not be predictable. It is possible that the author of the called `Procedure` did not expect it to be terminated from the outside at arbitrary positions. Thus, it is recommended practice to pass a flag into the called `Procedure`, so that the called `Procedure` can terminate itself in a controlled manner.

— **`Return` node:**

  If a **`Return`** node is executed in one of the lanes, the lane shall complete, but the jump induced by the **`Return`** node shall be delayed until all other lanes complete. After this, the **`Procedure`** completes.

— Throw of an unhandled exception:

  If a node in one of the lanes throws an exception which is **not** handled in that lane, the lane shall complete, but the throw is delayed until all other lanes have completed. After this, the exception is thrown upwards. If more than one exception was delayed, the **first** thrown exception shall be thrown upwards and the others shall be discarded.

Concerning concurrency issues which might occur on multicore systems, the following rules apply:

— When there are two or more delayed exceptions which occurred **simultaneously** (same timestamp), it is **not specified** which exception will be thrown upwards. The behaviour depends on the specific OTX runtime implementation.

— When there is a delayed exception and a delayed return which occurred simultaneously (same timestamp), the behaviour is **unspecified**, it depends on the specific OTX runtime implementation whether an exception is thrown upwards or the **`Procedure`** completes.

7.13.4.5.4 Example

The example below shows a simple parallel node with three lanes. The lanes contain branches, loops and actions to be carried out in parallel. It also contains a **`TerminateLanes`** node (other nodes in the lanes are not further specified for this example).

Sample of Parallel nodes

```
<parallel id="parallel1">
  <specification>Demonstration of three parallel lanes</specification>
  <realisation>
    <lane>
      <action id="a1"/>
      <branch id="b1">
        <realisation>
          <if>
            <condition id="b1c1"/>
            <flow>
              <terminateLanes id="t1"/>
            </flow>
          </if>
        </realisation>
      </branch>
    </lane>
    <lane>
      <action id="a2"/>
      <loop id="l1" name="MyLoop"/>
    </lane>
    <lane>
      <action id="a3"/>
      <action id="a4"/>
      <action id="a5"/>
    </lane>
  </realisation>
</parallel>
```

7.13.4.6 MutexGroup node

7.13.4.6.1 Description

The **`MutexGroup`** node is designed for resolving concurrency issues which might occur within the scope of **`Parallel`** nodes, as specified in 7.13.4.5. Syntactically, the **`MutexGroup`** node has similarities to the **`Group`** node because it can be used to wrap a sequence of nodes together, but beyond that it has special semantics concerning parallel execution, as specified below.

7.13.4.6.2 Syntax

Figure 35 shows the syntax of the **`MutexGroup`** node type.

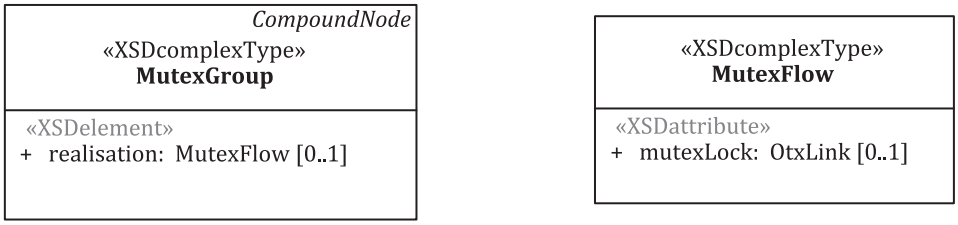

Figure 35 — Data model view: `MutexGroup`

7.13.4.6.3 Semantics

`MutexGroup` is a **`CompoundNode`** which is a **`Node`** which is **`NamedAndSpecified`**. The semantics of all derived properties are described in 7.13.2. Only the specific properties of the **`MutexGroup`** type are described here, with the following semantics:

— **`<realisation>: MutexFlow [0..1]`**

This optional element is the counterpart to the **`<specification>`** element described for the **`Node`** type. It is a **`Flow`** of nodes (see 7.16.9).

— **`<mutexLock>: OtxLink [0..1]`**

This optional attribute can be connected to a declaration of type **`MutexLock`** (by **`OtxLink`**). If the attribute is set, entry to the flow shall be controlled by that **`MutexLock`**. If the attribute is omitted, entry to the flow shall be controlled by a one runtime-specific, global lock.

In multi-threaded environments, more than one process can try to access the same variable(s) or resource(s). This can cause serious concurrency issues and shall be dealt with properly. Consider concurrency situation shown to the left of Figure 36.

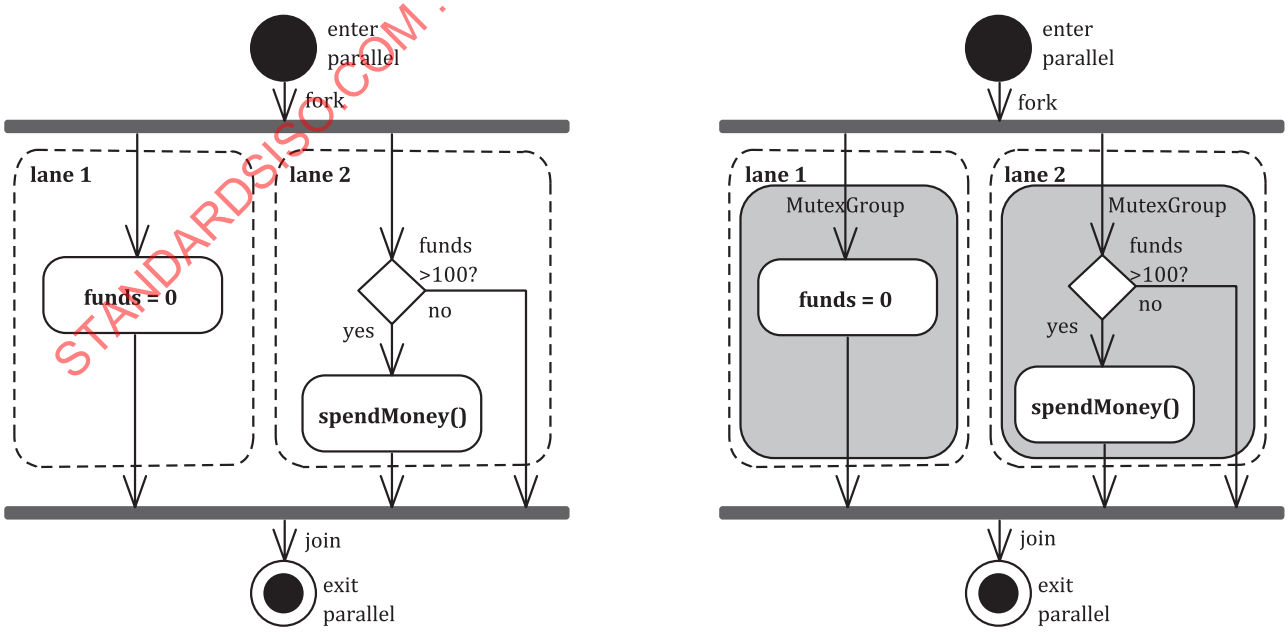

Figure 36 — Behavioural description: `MutexGroup`

In the above example, both lanes try to access the variable `"funds"`. The condition in lane 2 aims to make sure money is only spent if there are enough funds available. In lane 1, `"funds"` changes its value to 0. In parallel execution this change might happen **after** the condition `"funds > 100?"` is checked in lane 2 but **before** `"spendMoney()"` is executed. In this case the result is unexpected. At the time the procedure is called, the value is not greater than 100 anymore even though this value was checked. The condition for the procedure call is not valid and the procedure should not be called.

In general, parts of code where multiple threads try to access the same variable(s) or resource(s) are called critical sections. By only allowing one thread at a time to enter a critical section, access is effectively reduced to one thread at a time – other threads have to wait for their turn.

The OTX `MutexGroup` node is shown to the right of Figure 36. It allows wrapping critical sections so that access happens in a controlled manner (as it is mutually exclusive). Of all `MutexFlows` which refer to the same `MutexLock`, only one shall be executed at a time. Similarly, of all `MutexFlows` which do not refer to a `MutexLock`, only one shall be executed at a time.

`MutexLocks` shall be reentrant: a single parallel lane may acquire the `MutexLock` more than once. Likewise, the global lock, which is used for `MutexGroups` without the `mutexLock` attribute, shall be reentrant. Therefore, a `MutexGroup` which is nested inside another `MutexGroup` within the same parallel lane shall not cause a deadlock.

The node closely resembles the `synchronized` block as specified by Reference [9], subclause 14.19.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see C.2.53);

— Core_Chk066 – correct referencing of MutexLocks (see C.2.66).

7.13.4.6.4 Example

The example below shows a specification-level OTX snippet which demonstrates the concurrency situation discussed above (see Figure 36).

Sample of MutexGroup

```
<parallel id="p1">
  <specification>Controlled concurrency</specification>
  <realisation>

    <lane>  <!-- left lane -->
      <mutex id="m2">
        <specification>Critical section</specification>
        <realisation>
          <action id="a1">
            <specification>funds = 0</specification>
            <!-- Implementation goes here -->
          </action>
        </realisation>
      </mutex>
    </lane>

    <lane> <!-- right lane -->
      <mutex id="m2">
        <specification>Critical section</specification>
        <realisation>
          <branch id="b1">
            <specification>Spend money if funds are greater 100</specification>
            <realisation>
              <if>
                <condition id="b1c1">
                  <specification>Condition: funds greater 100?</specification>
                  <!-- Implementation goes here -->
                </condition>
                <flow>
                  <action id="call1">
                    <specification>Call procedure 'spendMoney' </specification>
                    <!-- Implementation goes here -->
                  </action>
                </flow>
              </if>
            </realisation>
          </branch>
        </realisation>
      </mutex>
    </lane>
  </realisation>
</parallel>
```

7.13.4.7 Handler node

7.13.4.7.1 Description

The **Handler** node is designed for monitoring sections of a procedure for unexpected behaviour (exceptions). If an exception was thrown in the monitored section, it can be treated by so-called catch blocks.

7.13.4.7.2 Syntax

Figure 37 shows the syntax of the complex type **Handler**.

The **<realisation>** element of **Handler** has a special design: in addition to the mandatory **<try>** flow, it forces authors to implement:

— either a single but mandatory **<finally>** flow or

— a list of at least one **<catch>** flow followed by an optional **<finally>** flow.

Like this, it is guaranteed that there is at the least a **<finally>** when no **<catch>** is given, but if there are **<catch>** flows, **<finally>** becomes optional.

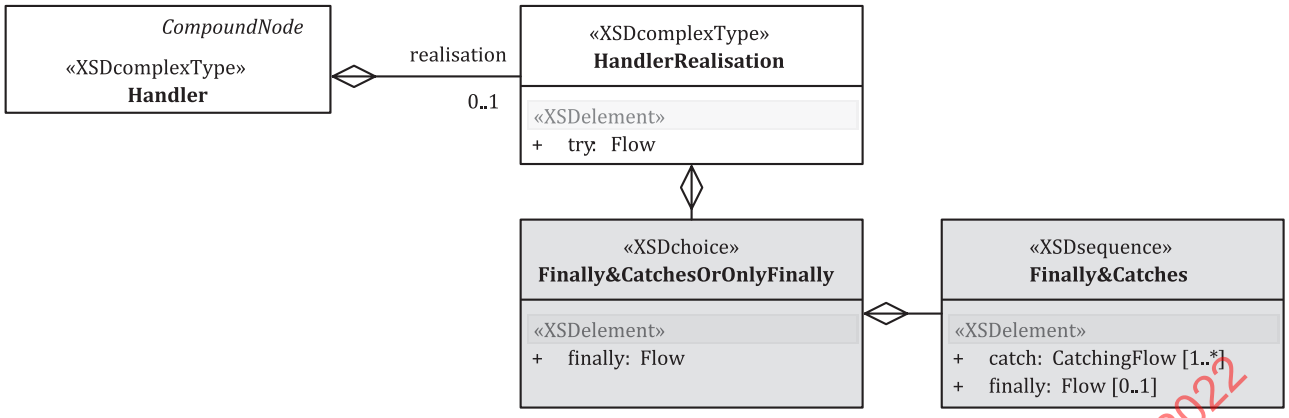

Figure 37 — Data model view: `Handler`

7.13.4.7.3 Semantics

`Handler` is a **`CompoundNode`** which is itself a **`Node`** which is **`NamedAndSpecified`**. The semantics of all derived properties are described in 7.13.2. Only specific semantic properties of **`Handler`** are described here.

A **`<handler>`** node contains a so-called **`<try>`** flow accompanied by a list of **`<catch>`** flows for exception treatment and/or a **`<finally>`** flow. The **`<handler>`** node monitors the **`<try>`** flow for exceptions.

When an exception is raised in the monitored **`<try>`** flow, it completes abruptly. If there is a **`<catch>`** flow for treating the kind of exception that was raised, control is passed to that **`<catch>`** flow. After **`<catch>`** execution, node execution will resume right after the **`<handler>`** node (if the catch execution completes normally). Otherwise, if there is no **`<catch>`** for the exception raised, the exception is passed upwards to an outer **`<handler>`** that shall treat the exception, and so on. If the exception is not treated by outer handlers either, the exception is passed to the runtime system itself (out of OTX scope). If there is a **`<finally>`** flow, it will **always** be executed at the end, disregarding if there was any exception or not or if a catch flow has thrown also or not.

IMPORTANT — The full runtime behaviour of the `<handler>` node is even more complex than described above. Since the semantics are equivalent to the semantics of the try statement in the Java language, the specification given by Reference [9], subclause 14.20 is indispensable for full comprehension of the `<handler>` node.

`Handler` has the following properties:

— **`<realisation>: HandlerRealisation [0..1]`**

This optional element is the machine-readable counterpart to the **`<specification>`** element described for the **`Node`** type. It is a container for a **`<try>`** flow, **`<catch>`** flows and a **`<finally>`** flow:

— **`<try>: Flow [1]`**

Represents the flow that is monitored by the **`Handler`**. Potentially thrown exceptions may be treated by **`<catch>`** flows. Any occurring exception in this flow will cause it to complete abruptly.

— **`<catch>: CatchingFlow [1..*]`**

When an exception is thrown in the **`<try>`** block, the **`<catch>`** flows of a **`Handler`** are analysed one by one in order of appearance in the OTX document. The first **`<catch>`** with a matching exception type will be executed. If there is no matching **`<catch>`**, control is passed to the next outer handler (after executing the **`<finally>`** flow, if given). Refer to 7.13.4.8 for a detail description of **`CatchingFlow`**.

Associated checker rules:

— Core_Chk016 – no redundant exception catches (see C.2.16).

— **<finally>: Flow [0..1]**

If existing, this flow will always be executed at the end of the **<handler>**. It will also be executed, if there is no **<catch>** treating the type of exception, or if the **<catch>** flow also completed abruptly.

7.13.4.7.4 Example

The example below shows the **Handler** node. In its **<try>** flow, an exception is thrown explicitly which is caught by a **<catch>** flow. Also refer to 7.13.5.5 for an overview of the **Throw** node type.

Sample of Handler

```
<?xml version="1.0" encoding="UTF-8"?>
<otx xmlns="http://iso.org/OTX/1.0.0" id="18"
  name="HandlerExample"
  package="org.iso.otx.examples"
  version="1.0"
  timestamp="2009-10-20T14:40:10"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:hmi="http://iso.org/OTX/1.0.0/HMI">

  <procedures>
    <procedure name="main" visibility="PUBLIC" id="18-p1">
      <realisation>
        <declarations>
          <variable name="userExc" id="18-d1">
            <realisation>
              <dataType xsi:type="UserException" />
            </realisation>
          </variable>
        </declarations>

        <throws>
          <exception xsi:type="UserException" />
        </throws>

        <flow>
          <handler id="handler1">
            <specification>Handler with a finally and a catch</specification>
            <realisation>
              <try>
                <throw id="throw1">
                  <specification>Throws an exception configured by the author</specification>
                  <realisation xsi:type="UserExceptionCreate">
                    <qualifier xsi:type="StringLiteral" value="MyManualException" />
                    <text xsi:type="StringLiteral"
                      value="This is an exception thrown without reason!" />
                  </realisation>
                </throw>
              </try>
              <catch>
                <exception id="exc1">
                  <specification>Catch UserExceptions</specification>
                  <realisation>
                    <type xsi:type="UserException" />
                    <handle name="userExc" />
                  </realisation>
                </exception>
                <flow>
                  <action id="a3">
                    <specification>Show this exception text</specification>
                    <realisation xsi:type="hmi:MessageDialog">
                      <hmi:message xsi:type="GetExceptionText">
                        <exception xsi:type="ExceptionVariable" name="userExc" />
                      </hmi:message>
                      <hmi:messageType xsi:type="IntegerLiteral"
                        value="1" />
                    </realisation>
                  </action>
                </flow>
              </catch>
              <finally>
                <action id="finalAction" />
              </finally>
            </realisation>
          </handler>
        </flow>
      </realisation>
    </procedure>
  </procedures>
</otx>
```


7.13.4.8 CatchingFlow

7.13.4.8.1 Description

The **CatchingFlow** type bundles information about a to-be-caught **Exception** to a **Flow** that handles it. The flow shall only be executed if there was an exception of the right type thrown in the enclosing exception **Handler**, see above.

7.13.4.8.2 Syntax

Figure 38 shows the syntax of the **CatchingFlow** type.

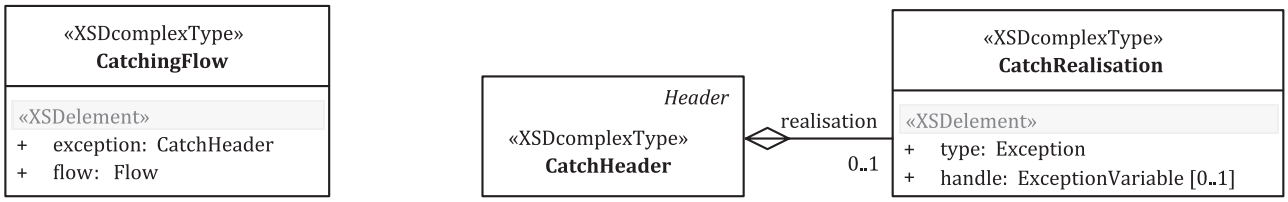

Figure 38 — Data model view: `CatchingFlow`

7.13.4.8.3 Semantics

`CatchingFlow` has the following semantic properties:

— **`<exception>: CatchHeader [1]`**

Represents the catch header which aggregates information about the exception **`<type>`** and the **`<handle>`** to which a caught exception shall be assigned to (plus the derived properties of the **`Header`** base type, see 7.13.4.9).

— **`<type>: Exception [1]`**

Allows choosing one out of the list of all **`Exception`** types (by using the **`xsi:type`** attribute). The base type **`Exception`** may be chosen here as well: In this case, any kind of exception is caught by the **`CatchingFlow`**.

The **`<flow>`** of the **`CatchingFlow`** shall only be executed if the type of the thrown exception matches (or is a subtype of) the type given here.

If an **`<init>`** sub-element ocurrs here, it shall be ignored (see **`UserException`**, 7.16.7.4.8).

Refer to 7.16.7.4.8 for a definition of core **`Exception`** types.

Associated checker rules:

— Core_Chk057 – no use of init in catch header exception type definition (see C.2.57).

— **`<handle>: ExceptionVariable [0..1]`**

If the associated **`<flow>`** element needs to access the exception object itself, the optional **`<handle>`** element allows assigning the object to an **`Exception`** type variable.

Associated checker rules:

— Core_Chk019 – type-safe exception catches (see C.2.19);

— Core_Chk053 – no dangling OtxLink associations (see C.2.53).

— **`<flow>: Flow [1]`**

The flow shall be carried out when the exception stated by the **`<exception>`** element has been caught (see 7.16.9).

7.13.4.8.4 Example

Refer to the example given in 7.13.4.7.4.

7.13.4.9 Header type

7.13.4.9.1 Description

The compound nodes **`Loop`**, **`Branch`** and **`Handler`** use the **`Header`** type as a container for heading information like loop conditions, the if- or else-if-case conditions in a branch or the exception indication in catch blocks of a handler. In graphical representations of OTX-sequences, this information can be displayed in distinct graphical shapes, e.g. a "diamond" for displaying conditions.

The abstract **`Header`** type describes the common features used by all of its subtypes, namely **`LoopConfigurationHeader`**, **`ConditionHeader`** and **`CatchHeader`**.

7.13.4.9.2 Syntax

Figure 39 shows the syntax of the **`Header`** type.

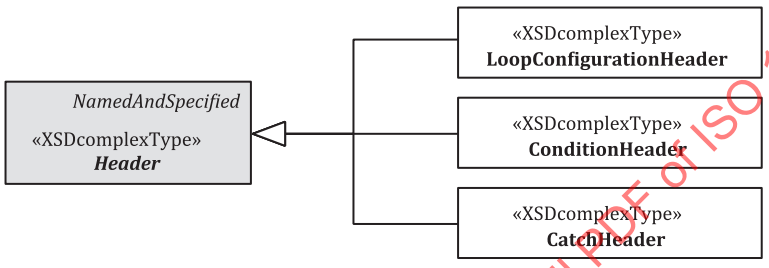

Figure 39 — Data model view: `Header`

7.13.4.9.3 Semantics

With respect to graphical representation, headers play a similar role as nodes: they can be displayed by graphical shapes, they can be specified or/and realized, floating comments may point to them and they may carry a name. The only difference to the abstract **`Node`** type is the missing **`disabled`** attribute, which does not make sense for headers.

The properties of the abstract **`Header`** type have the following semantics:

— **`id: OtxId [1]`** (derived from **`NamedAndSpecified`**, see 7.16.4)

  Semantics are identical to the **`Node`** semantics, see 7.13.2.

— **`name: OtxName [0..1]`** (derived from **`NamedAndSpecified`**, see 7.16.4)

  Semantics are identical to the **`Node`** semantics, see 7.13.2.

— **`<metaData>: MetaData [0..1]`** (derived from **`NamedAndSpecified`**, see 7.16.4)

  Semantics are identical to the **`Node`** semantics, see 7.13.2.

— **`<specification> : xsd:string [0..1]`** (derived from **`NamedAndSpecified`**, see 7.16.4)

  Semantics are identical to the **`Node`** semantics, see 7.13.2.

7.13.4.9.4 Example

Below is an example of two different uses of the **`Header`** type: As a specification stage loop condition (the **`<configuration>`** element is of **`Header`** type) and as a branches if-case condition (the **`<condition>`** element is of **`Header`** type).

Sample of Header

```
<?xml version="1.0" encoding="UTF-8"?>
<otx xmlns="http://iso.org/OTX/1.0.0" id="19"
  name="HeaderExample"
  package="org.iso.otx.examples"
  version="1.0"
  timestamp="2009-10-20T14:40:10"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <procedures>
    <procedure name="main" visibility="PUBLIC" id="19-p1">
      <specification>Demonstration of a loop</specification>
      <realisation>
        <flow>

          <loop id="loop1" name="myLoop">
            <realisation>
              <configuration id="loop1cond1">
                <specification>unspecified loop condition</specification>
              </configuration>
              <flow />
            </realisation>
          </loop>

          <branch id="b1">
            <realisation>
              <if>
                <condition id="b1cond1">
                  <specification>This is always true</specification>
                  <realisation xsi:type="BooleanLiteral"
                    value="true" />
                </condition>
                <flow />
              </if>
              <else>
                <action id="a1" />
              </else>
            </realisation>
          </branch>

        </flow>
      </realisation>
    </procedure>
  </procedures>
</otx>
```

7.13.4.10 ExtensibleCompundNode

7.13.4.10.1 Description

Declares a compound node which can be extented by additional realisations defined in new OTX extensions using the standardised extension mechanism, example: switch-case statement.

7.13.4.10.2 Syntax

Figure 40 shows the syntax of the **`ExtensibleCompoundNode`** type.

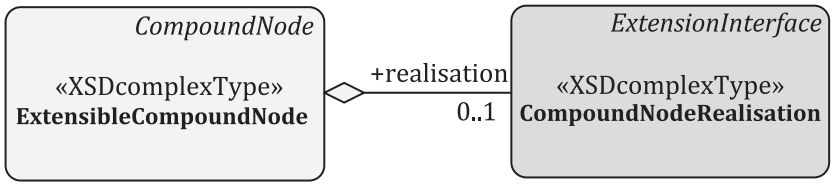

Figure 40 — Data model view: `ExtensibleCompoundNode`

7.13.4.10.3 Semantics

The properties of **`ExtensibleCompoundNode`** have the following semantics:

— **`realisation: CompoundNodeRealisation [0..*]`**

  This optional element is the counterpart to the **`<specification>`** element described for the **`Node`** type. It declares a compound node, which can be extended by new realisations.

IMPORTANT — The extension mechanism described here only defines the syntactic rules to which extension implementers shall obey in order to conform to this document. It shall be ensured that no OTX requirements are violated.

7.13.5 End nodes

7.13.5.1 Overview

Reconsider 7.16.9 concerning the behaviour of flows. After flow processing has executed the last node of a flow, control is passed to the next-in-row node in the outer flow. If there is no more outer flow—this is the case when the end of the procedure level flow is reached—the procedure execution ends.

This is the standard behaviour for implicit flow ends. By contrast, the explicit **`EndNode`** types described in the following clauses enforce a different end-of-flow behaviour.

7.13.5.2 Return node

7.13.5.2.1 Description

A **`Return`** node shall complete the execution of a procedure immediately. Control shall be passed back to the caller. **`Return`** nodes are allowed at any nesting depth. **`Return`** is a controlled jump with a well-defined jump target, namely the end of the procedure. Compare this to the structured programming paradigm in 6.3.

7.13.5.2.2 Syntax

Figure 41 shows the syntax of the **`Return`** node type.

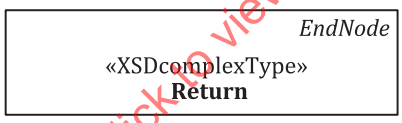

Figure 41 — Data model view: `Return`

7.13.5.2.3 Semantics

`Return` is an **`EndNode`** which is a **`Node`** which is **`NamedAndSpecified`**. The semantics of all derived properties are described in 7.13.2. Only the specific semantic properties of **`Return`** are described here.

IMPORTANT — For the `Parallel` node, special semantics apply when `Return` is executed in parallel lanes. For details, see 7.13.4.5 which is indispensable for the comprehension of the correct behaviour of `Return` in `Parallel`.

The activity diagram in Figure 42 clarifies the semantics. It shows a procedure flow which consists out of a **`Branch`** and another node labelled “last node in procedure”. The if-flow in the branch contains a node “node” followed by a **`Return`** node. There is no default case.

— Without the explicit **`Return`** node, “last node in procedure” would always be executed after branch execution, since control would pass to the next node in the outer procedure level flow, independent of the truth value of the if-condition.

— With the explicit **`Return`** node, procedure execution ends immediately and “last node in procedure” is bypassed if the condition is **`true`**.

A corresponding example OTX document is given below.

This ability of the **`Return`** node is helpful in situations where the procedure execution shall end prematurely because certain circumstances are met.

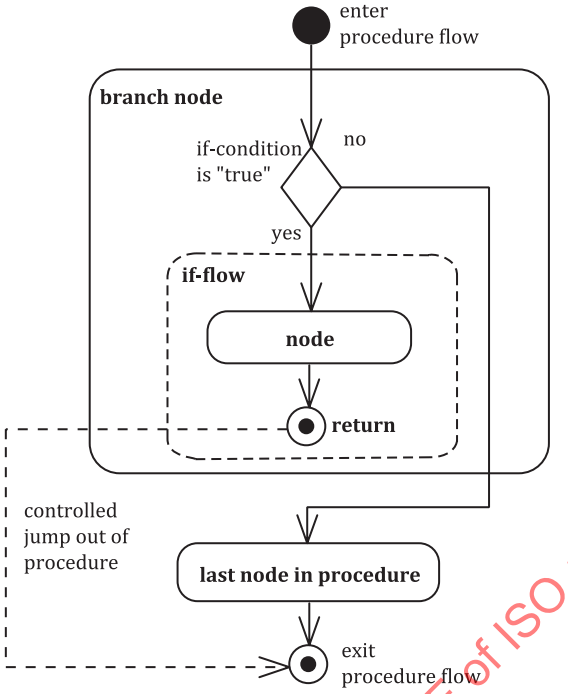

Figure 42 — Behavioural description: `Return`

7.13.5.2.4 Example

Sample of Return

```
<?xml version="1.0" encoding="UTF-8"?>
<otx xmlns="http://iso.org/OTX/1.0.0" id="20"
  name="ReturnExample"
  package="org.iso.otx.examples"
  version="1.0"
  timestamp="2009-10-20T14:40:10">

  <procedures>
    <procedure name="main" visibility="PUBLIC" id="20-p1">
      <realisation>
        <flow>
          <branch id="branch1">
            <specification>Branch with a return</specification>
            <realisation>
              <if>
                <condition id="cond1">
                  <specification>Cause for early procedure completion</specification>
                </condition>
                <flow>
                  <action id="a1" />
                  <return id="return1">
                    <specification>Bypasses node "a2"</specification>
                  </return>
                </flow>
              </if>
            </realisation>
          </branch>
          <action id="a2">
            <specification>Last node in procedure</specification>
          </action>
        </flow>
      </realisation>
    </procedure>
  </procedures>
</otx>
```

7.13.5.3 Continue node

7.13.5.3.1 Description

A **Continue** node completes execution of a **Loop** flow immediately and initiates the next iteration, after loop condition evaluation. **Continue** shall be used only within **Loop** nodes, at any nesting depth. **Continue** is a controlled jump with a well-defined jump target, namely the first node of the **Loop** flow. Compare this to the structured programming paradigm explained in 6.3.

7.13.5.3.2 Syntax

Figure 43 shows the syntax of the **Continue** node type.

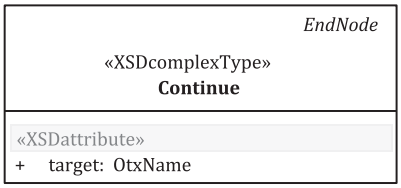

Figure 43 — Data model view: Continue

7.13.5.3.3 Semantics

Continue is an **EndNode** which is a **Node** which is **NamedAndSpecified**. The semantics of all derived properties are described in 7.13.2. Only the specific semantic properties of **Continue** are described here.

Figure 44 shows the runtime behaviour of loops with nested **Continue** nodes. For the sake of brevity, the condition test has been left out in the figure. The condition shall always be tested before the first node in the loop flow will be executed.

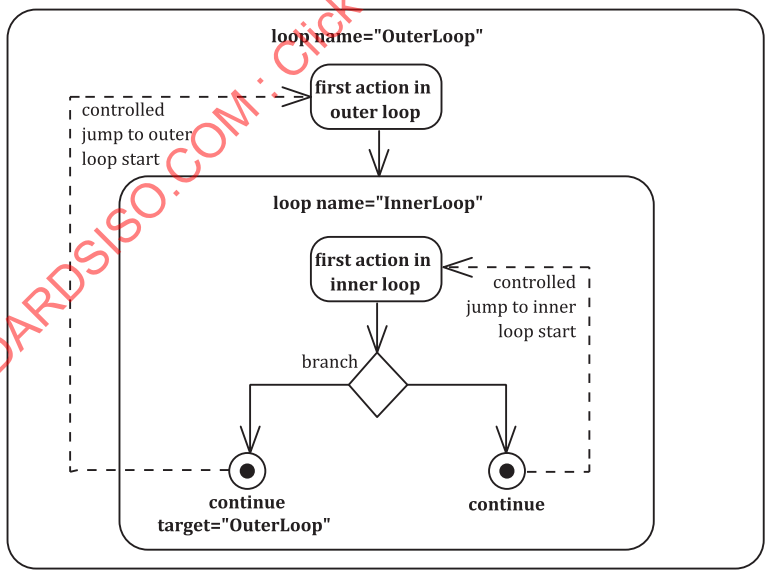

Figure 44 — Behavioural description: Continue

The properties of **Continue** have the following semantics:

— **target: OtxName [0..1]**

This attribute specifies which of the outer loops shall be continued. If the attribute is omitted, the default target shall be the innermost loop. Otherwise, the value shall match the **name** attribute of

the targeted loop in the procedure. This is ensured by XML validation. However, XML validation cannot ensure that the **`Continue`** node is in fact nested in the targeted loop. There is a checker rule defined for ensuring this requirement.

Associated checker rules:

— Core_Chk020 – correct nesting of Continue node (see C.2.20).

7.13.5.4 Break node

7.13.5.4.1 Description

A **`Break`** node forces **`Loop`** execution to complete immediately. Control is passed to the next node after the "broken" **`Loop`** node. **`Break`** shall be used only within **`Loop`** nodes, at any nesting depth.

`Break` is a controlled jump with a well-defined jump target, namely the next node after the **`Loop`**. Compare this to the structured programming paradigm explained in 6.3.

7.13.5.4.2 Syntax

Figure 45 shows the syntax of the **`Break`** node type.

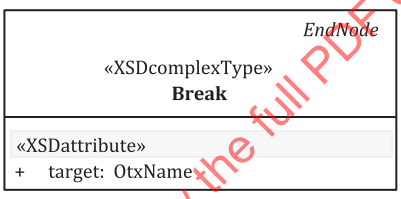

Figure 45 — Data model view: `Break`

7.13.5.4.3 Semantics

`Break` is an **`EndNode`** which is a **`Node`** which is **`NamedAndSpecified`**. The semantics of all derived properties are described in 7.13.2. Only the specific semantic properties of **`Break`** are described here.

The properties of **`Break`** have the following semantics:

— **`target: OtxName [0..1]`**

This attribute names the target **`Loop`** node which is forced to complete immediately. If the attribute is omitted, the default target shall be the innermost loop. Otherwise, the value shall match the **`name`** attribute of the targeted loop in the procedure. This is ensured by XML validation. However, XML validation cannot ensure that the **`Break`** node is in fact nested in the targeted loop. There is a checker rule defined for ensuring this requirement.

Associated checker rules:

— Core_Chk021 – correct nesting of Break node (see C.2.21).

Figure 46 shows the runtime behaviour of loops with nested **`Break`** nodes. For the sake of brevity, the condition test has been left out in the figure. The condition shall always be tested before the first node in the loop flow will be executed.

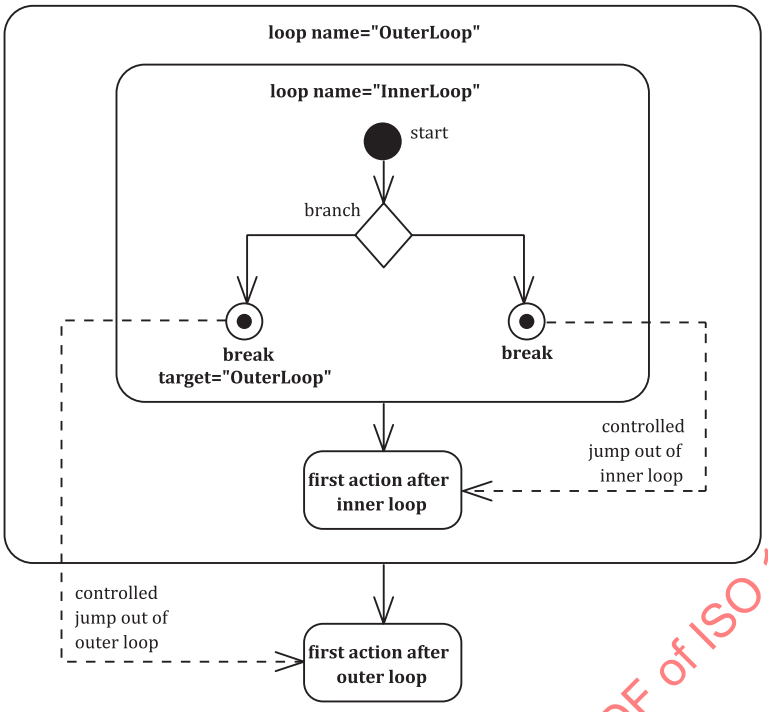

Figure 46 — Behavioural description: `Break`

7.13.5.5 Throw node

7.13.5.5.1 Description

There are different reasons for exceptions to occur.

Exceptions can be thrown implicitly, e.g. by the execution of an **`Action`** node (see 7.13.3) or during evaluation of a **`Term`** (see 7.15). For any of such **`Action`** nodes and **`Term`** types, the potentially thrown exceptions are defined in this document.

For example, the **`Divide`** term throws an **`ArithmeticException`** under well-defined conditions. Here, it is not the author who throws the exception explicitly, but it is the evaluation algorithm for **`Divide`** which is executed by the runtime system which throws the exception implicitly. The only task left for the author is to define a reaction onto the potentially expected exception by using a **`<catch>`** in a **`Handler`** node (see 7.13.4.7).

By contrast, the **`Throw`** node is a means to throw exceptions explicitly. The author can define exactly when an exception shall be thrown and he/she may configure the exception by providing textual information describing the user defined exception.

7.13.5.5.2 Syntax

Figure 47 shows the syntax of the **`Throw`** node type.

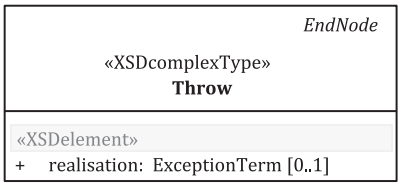

Figure 47 — Data model view: `Throw`

7.13.5.5.3 Semantics

Throw is derived from **EndNode** which is a **Node** which is **NamedAndSpecified**. The semantics of all derived properties are described in 7.13.2. Only the specific semantic properties of **Throw** are described here.

The properties of **Throw** have the following semantics:

— **realisation: ExceptionTerm [0..1]**

Specifies an **ExceptionTerm** that returns the exception object that shall be thrown. This can be either the value of an exception variable of any **Exception** type, or a newly created **UserException**.

Associated checker rules:

— Core_Chk017 – checked exceptions (1) (see C.2.17);

— Core_Chk018 – checked exceptions (2) (see C.2.18).

IMPORTANT — Special semantics apply when exceptions are thrown in Parallel lanes. For details see 7.13.4.5 which is indispensable for the comprehension of the correct behaviour of Throw in Parallel.

7.13.5.5.4 Example

An example is provided in the **Handler** specification, see 7.13.4.7.4.

7.13.5.6 TerminateLanes node

7.13.5.6.1 Description

The **TerminateLanes** end node is exclusively designed for deployment in **Parallel** node lanes. It allows aborting the execution of all lanes in a **Parallel** node without having to wait for them to complete normally (for details about the **Parallel** node, see 7.13.4.5).

A **TerminateLanes** Node shall only terminate lanes inside the enclosing **Parallel** node, its nested **Parallel** nodes and called **Procedures**. It shall not terminate parent parallel lanes of the enclosing **Parallel** node.

7.13.5.6.2 Syntax

Figure 48 shows the syntax of the **TerminateLanes** node type.

Figure 48 — Data model view: TerminateLanes

7.13.5.6.3 Semantics

TerminateLanes is derived from **EndNode** which is a **Node** which is **NamedAndSpecified**. The semantics of all derived properties are described in 7.13.2. Only the specific semantic properties of **TerminateLanes** are described here.

When a **TerminateLanes** node is executed in one of the lanes of a **Parallel** node, all other lanes are signalled to complete prematurely.

IMPORTANT — Additional semantics for TerminateLanes are specified in 7.13.4.5. These are indispensable for the comprehension of the correct behaviour of TerminateLanes in Parallel.

IMPORTANT — On receiving a premature completion signal, all lanes shall complete immediately after completion of the last atomic node or finally block (if any). Atomic nodes are `Action` nodes with an atomic `ActionRealisation` as well as the end nodes `Return`, `Break`, `Continue`, `TerminateLanes` and `Throw`. There is no completion timeout defined for atomic nodes, a lane shall wait no matter how long the node takes to complete. This avoids producing undefined states.

DEPRECATED Do not use TerminateLanes, because it can lead to unexpected behaviour if a called procedure is terminated by its caller. TerminateLanes can be replaced by code that modifies some variable to indicate that the parallel lanes should terminate. The parallel lanes should check this variable regularly and terminate in an orderly fashion.

Associated checker rules:

— Core_Chk022 – correct nesting of TerminateLanes node (see C.2.22);

— Core_Chk060 – usage of deprecated elements (see C.2.60).

7.13.5.6.4 Example

An example is provided in the `Parallel` specification, see 7.13.4.5.

7.13.5.7 ExtensibleEndNode

7.13.5.7.1 Description

Declares an end node which can be extented by additional realisations defined in new OTX extensions using the standardised extension mechanism, example: goto.

7.13.5.7.2 Syntax

Figure 49 shows the syntax of the `ExtensibleEndNode` type.

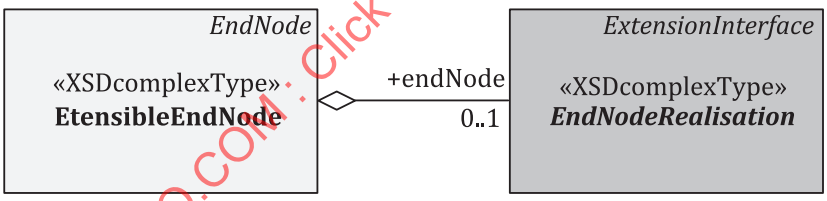

Figure 49 — Data model view: `ExtensibleEndNode`

7.13.5.7.3 Semantics

The properties of `ExtensibleCompundNode` have the following semantics:

— `realisation: EndNodeRealisation [0..*]`

This optional element is the counterpart to the `<specification>` element described for the `Node` type. It declares an end node, which can be extended by new realisations.

IMPORTANT — The extension mechanism described here only defines the syntactic rules to which extension implementers shall obey in order to conform to this document. It shall be ensured that no OTX requirements are violated.

7.14 Actions

7.14.1 Overview

The following describes all **`ActionRealisation`** extensions defined in the OTX core. Those are designed for setting the behaviour of **`Action`** nodes, as specified in 7.13.3.

7.14.2 Syntax

The syntax of all **`ActionRealisation`** types is shown in the overview of Figure 50.

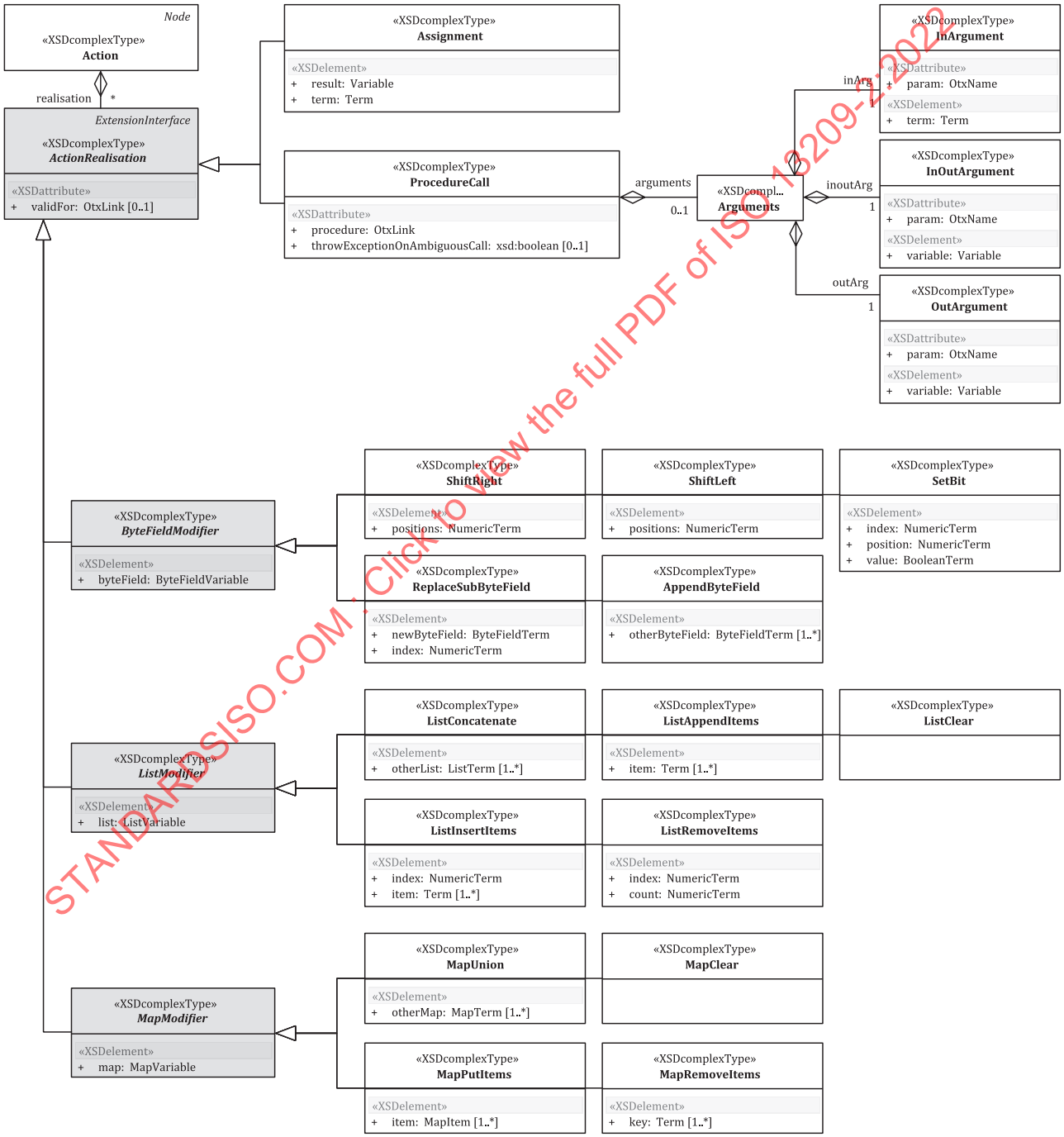

Figure 50 — OTX core `ActionRealisation` extensions overview

7.14.3 General considerations

There are several basic considerations concerning all of the action realisation types.

IMPORTANT — The evaluation order of arguments of the actions is not always specified. Therefore, if exceptions occur in more than one argument, the behaviour may differ for different OTX runtime systems.

IMPORTANT — Actions modify the value of one or several variables. In order to guarantee that variable values are always in a defined state, all arguments of an action and all preconditions shall be checked before modifying a variable value.

IMPORTANT — List, Map, Bytefield variables are always initialized. If the List declaration is not explicitly initialized (omitted `<init>` element), an empty list shall be created and assigned to the List identifier.

7.14.4 Assignment

7.14.4.1 Description

The **`Assignment`** action assigns a value to a variable.

7.14.4.2 Syntax

The syntax of the **`ActionRealisation`** action type is shown in Figure 50.

7.14.4.3 Semantics

`Assignment` is an **`ActionRealisation`**. See 7.13.3 for details on the **`Action`** node type.

The properties of **`Assignment`** have the following semantics:

— **`<result>: Variable [1]`**

This describes the variable to which the resulting value of term evaluation shall be assigned (see 7.16.6 for the **`Variable`** type).

— **`<term>: Term [1]`**

This describes the value which shall be evaluated out of the given term at runtime. The evaluation semantics of all OTX core terms are described in 7.15.

Associated checker rules:

— Core_Chk023 – type-safe assignments (see C.2.23).

7.14.5 ProcedureCall

7.14.5.1 Description

This action calls another OTX procedure (directly or indirectly via a signature). If there are input parameters declared by the callee, the **`ProcedureCall`** action allows describing a list of corresponding arguments which shall be passed to the callee at runtime. OTX procedures do not have one single dedicated return value, instead they can define a set of output parameters which can (but do not have to) be used by the caller.

Procedure calls can happen in-document or cross-document; the latter means that a procedure defined in one document can call another signature or procedure defined in another, separate document. In the latter case, the visibility information of the called signature or procedure shall be taken into account (see 7.7 and 7.9). Procedures or signatures which are not visible to the caller shall not be callable.

7.14.5.2 Syntax

The syntax of the `ProcedureCall` action type is shown in Figure 50.

7.14.5.3 Semantics

`ProcedureCall` is an `ActionRealisation`. See 7.13.3 for details on the `Action` node type.

IMPORTANT — `ProcedureCall` represents an exemption concerning the atomicity of `Action` nodes. It is not atomic. If a `ProcedureCall` is executed in a `Parallel` lane and a `TerminateLanes` node is executed in another lane, procedure execution shall be completed prematurely. The same rules as for prematurely completed lanes apply, see 7.13.4.5 for details.

Since `ProcedureCall` is closely linked to the specification of the `Procedure` and `Signature` types and their parameter declarations, compare the following to 7.9 and 7.11.

The properties of `ProcedureCall` have the following semantics:

— `procedure: OtxLink [1]`

Contains the fully qualified name of the called procedure or signature. For cross-document calls, a prefix is needed which indicates the external document. The prefix shall be defined by a corresponding `<import>` element in the `<imports>` section of the OTX document (see 7.4). For in-document calls, no prefix is needed. See 7.16.3 for details of the `OtxLink` type.

IMPORTANT — The target of a `ProcedureCall` shall be visible to the caller. The information given by the `visibility` attribute of the called signature or procedure shall to be taken into account.

IMPORTANT — If the target of a `ProcedureCall` is a procedure or signature without `<realisation>`, the call shall be a NOP. The same applies for the call of a procedure which is not valid with respect to the context. Given arguments of the `ProcedureCall` action shall stay unchanged.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see C.2.53);

— Core_Chk028 – No Operation (NOP) ProcedureCall (see C.2.28);

— Core_Chk024 – correct target for ProcedureCall (see C.2.24);

— Core_Chk025 – procedure signature has at least one implementing procedure (see C.2.25);

— Core_Chk017 – checked exceptions (1) (see C.2.17);

— Core_Chk018 – checked exceptions (2) (see C.2.18).

— `throwExceptionOnAmbiguousCall: xsd:boolean = {false|true} [0..1]`

For indirect calls (i.e. the call target is a signature), this attribute determines the behaviour when call ambiguity occurs: if two or more of the implementing procedures are valid, an `AmbiguousCallException` shall be thrown when `throwExceptionOnAmbiguousCall = "true"` (the default). Otherwise, the first valid procedure in alphabetic order shall be executed (see Figure 51 and the following).

— `<arguments>: Arguments [0..1]`

This simple container element represents the list of arguments for a procedure call. The content-type of `Arguments` is `<xsd:choice> [1..*]` which allows an arbitrary-length list of `<inArg>`, `<inoutArg>` and `<outArg>` elements.

Associated checker rules:

— Core_Chk029 – correct ProcedureCall arguments (see C.2.29).

Semantics of the argument types:

— **<inArg>: InArgument**

This describes an input argument. In the callee, input arguments shall be treated like constants; they cannot be written (like for constants, a static checker rule interdicts this). An input argument may be omitted **if and only if** there is an explicit initial value defined for the corresponding parameter. This initial value applies in place of the missing argument. The parameter for the argument is identified by name; the value that shall be passed into that parameter is described by a term:

— **param: OtxName [1]**

This is the name of the parameter which receives the argument value.

— **<term>: Term [1]**

This is the term describing the argument value.

NOTE 1 In graphical presentation of the procedure call, the initial value of an in-argument can be shown in an accentuated way to the author. Like this, the author gets feedback about the omitted argument (maybe she/he did not omit it on purpose).

— **<inoutArg>: InOutArgument**

This describes an input/output argument. Any changes made to the corresponding parameter in the callee shall be visible in the associated argument variable of the caller. An input/output argument may be omitted **if and only if** there is an explicit initial value defined for the corresponding parameter. In the called procedure, this initial value applies for the parameter in place of the missing argument value.

The callee parameter for the callers argument is identified by name, the callers argument is a variable:

— **param: OtxName [1]**

This is the name of the callee parameter which receives the argument value.

— **<variable>: Variable [1]** (see 7.16.6)

This represents the variable which shall be used for the input/output parameter.

IMPORTANT — In parallel execution (`Parallel` node), any changes made to an input/output argument or parameter will be visible in the caller and the callee at the same time. If the callee changes the value of the parameter, the new value will be available on the outside, in the parallel lanes using the argument variable. Vice versa, if the argument variable value is changed in one of the parallel lanes, the change will be visible in the corresponding parameter of the callee.

NOTE 2 In graphical presentation of the procedure call, the initial value of an inout-argument can be shown in an accentuated way to the author. Like this, the author gets feedback about the omitted argument (maybe he/she did not omit it on purpose).

— **<outArg>: OutArgument**

This describes where the returned value of an output parameter shall be assigned to. The caller may omit output arguments **freely** (e.g. in the case when there is no interest in one of the returned data).

The callee parameter for the callers argument is identified by name, the callers argument is a variable:

— `param: OtxName [1]`

This is the name of the parameter whose returned value is of interest.

— `<variable>: Variable [1]` (see 7.16.6)

This describes the variable to which the returned parameter value shall be written.

IMPORTANT — The writing-back of the returned parameter value into the argument variable shall not happen until termination of the called procedure.

Associated checker rules:

— Core_Chk030 – input- and in&output-argument omission (see C.2.30).

Throws:

— `InvalidReferenceException`

It is thrown if one of the inout-arguments is invalid (e.g. an uninitialized variable).

— `AmbiguousCallException`

It is thrown in case of indirect call via signature and `throwExceptionOnAmbiguousCall = "true"`: if two or more implementing procedures are valid at the same time (call target cannot be determined unambiguously).

The OTX procedure call is closely linked to the signature concept (see 6.9). Therefore, the runtime semantics differ depending if a procedure was directly called or if there is an indirection via a signature (see Figure 51).

The two major cases to be considered are the following.

— In the direct case, the called procedure will be executed after input argument evaluation and passing, **if it is valid with respect to the context**. If the procedure is not valid, nothing is executed (NOP).

— In the indirect case (the call target is a signature) a procedure implementing the signature needs to be identified. Since this is a dynamic process (the linking is done at runtime, not at authoring time), an OTX runtime system is required to have a signature-to-procedure-mapping: For a given signature name, the mapping shall contain a list of all procedures implementing that signature. When a signature is called, the runtime system shall infer the procedure from the mapping. The runtime system shall only identify procedures which are visible to the document defining the **signature** (see `visibility` attribute for signatures and procedures, 7.7 and 7.9) and which are valid with respect to the context (see validities concept, 6.8). If there is more than one applicable procedure available for the signature and the attribute `throwExceptionOnAmbiguousCall` is `true` an exception shall be thrown. Otherwise, **the first** applicable procedure (in ascending alphabetic order) shall be executed. The ordering shall be done after package-, document- and procedure-name. If there is no implementing procedure, nothing will be executed; nonetheless, inout- and out-arguments shall be set to init-values of the corresponding parameters, if init-values are provided in the signature. Otherwise, inout- and out-arguments stay unchanged.

IMPORTANT — It is recommended that OTX runtime applications should log cases when ambiguous procedure calls occur. This supports later debugging.

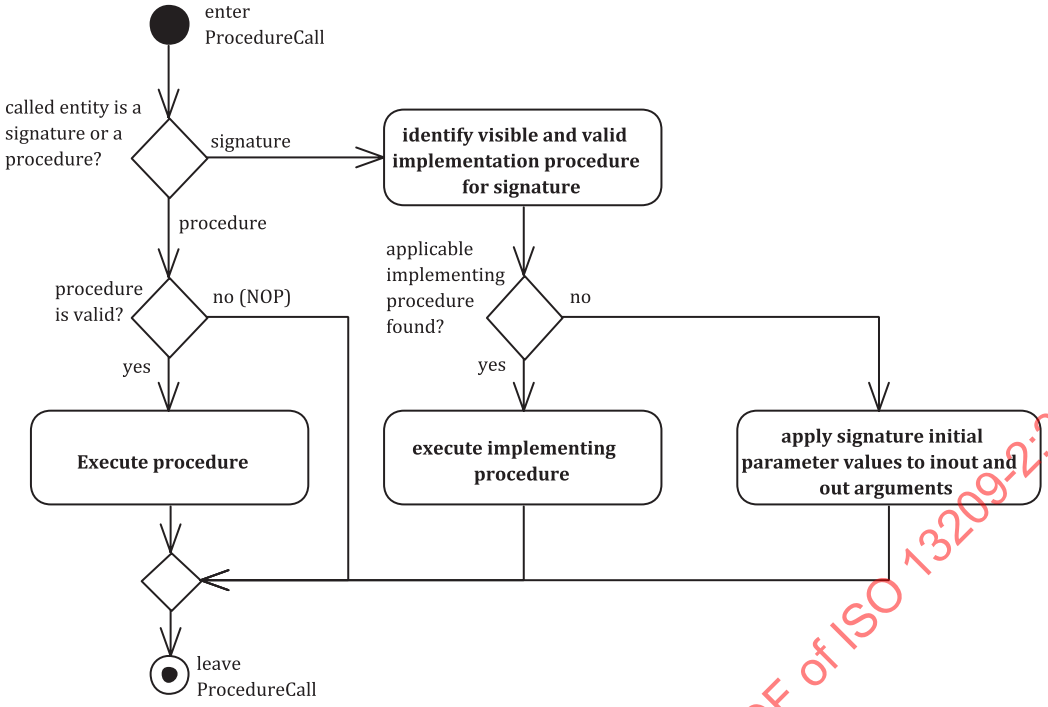

Figure 51 — ProcedureCall Activity

7.14.5.4 Example

Examples 1 and 2 below show two `ProcedureCall` action nodes.

— A call to a procedure `computeDelta` which is described in the same document. Note the matching parameter names in the call and in the procedure definition. Two input arguments, a literal integer `5` and an integer variable `x` are passed to parameters `a` and `b`. The output parameter `delta` is returned into variable `x` of the caller.

— A call to a procedure via a signature named `modifyValue`. The external document containing the signature is identified by the prefix `sig`. The names defined in the document are imported to the local document by an `<import>` element. Implementation procedures for the signature operate on a single input/output argument `x` which is passed to the `value` parameter (no implementation procedures given here).

Sample of ProcedureCall

```
<?xml version="1.0" encoding="UTF-8"?>
<otx xmlns="http://iso.org/OTX/1.0.0" name="ProcedureCallExample" package="org.iso.otx.examples"
  version="1.0" timestamp="2009-10-20T14:40:10" id="20"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <imports>
    <import prefix="sig" package="org.iso.otx.examples" document="Signatures" />
  </imports>
  <procedures>
    <procedure name="main" visibility="PUBLIC" id="20-p1">
      <specification>Demonstration of variable, constant and parameter declarations</specification>
      <realisation>
        <declarations>
          <variable name="x" id="20-d1" />
        </declarations>
        <flow>
          <action id="call1">
            <specification>Calls procedure located in same document</specification>
            <realisation xsi:type="ProcedureCall" procedure="computeDelta">
              <arguments>
                <inArg param="a">
                  <term xsi:type="IntegerLiteral" value="5" />
                </inArg>
                <inArg param="b">
                  <term xsi:type="IntegerValue" valueOf="x" />
                </inArg>
                <outArg param="delta">
                  <variable xsi:type="IntegerVariable" name="d" />
                </outArg>
              </arguments>
            </realisation>
          </action>
          <action id="call2">
            <specification>Calls procedure by an external signature</specification>
            <realisation xsi:type="ProcedureCall" procedure="sig:modifyValue">
              <arguments>
                <inoutArg param="value">
                  <variable xsi:type="IntegerVariable" name="x" />
                </inoutArg>
              </arguments>
            </realisation>
          </action>
        </flow>
      </realisation>
    </procedure>
    <procedure name="computeDelta" id="20-p2">
      <specification>Computes the difference between two values</specification>
      <realisation>
        <parameters>
          <inParam name="a" id="20-d2" />
          <inParam name="b" id="20-d3" />
          <outParam name="delta" id="20-d4" />
        </parameters>
        <flow />
      </realisation>
    </procedure>
  </procedures>
</otx>
```

Sample of Signatures

```
<?xml version="1.0" encoding="UTF-8"?>
<otx xmlns="http://iso.org/OTX/1.0.0" name="SignatureExample" package="org.iso.otx.examples"
  version="1.0" timestamp="2009-10-20T14:40:10" id="21 "
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <signatures>
    <signature name="modifyValue" id="21-s1">
      <specification>Adds a number to parameter value</specification>
      <realisation xsi:type="ProcedureSignature">
        <parameters>
          <inoutParam name="value" id="21-d1">
            <realisation>
              <dataType xsi:type="Integer" />
            </realisation>
          </inoutParam>
        </parameters>
      </realisation>
    </signature>
  </signatures>
</otx>
```

7.14.6 ByteFieldModifiers

7.14.6.1 Overview

`ByteFieldModifier` actions are designed for modifying bytefields. They shall only operate on `ByteField` variables.

7.14.6.2 ShiftRight

7.14.6.2.1 Description

Shifts a `ByteField` right. Bits at the most significant positions will be stuffed by zero. Bits at the least significant positions will be discarded. The size of the ByteField will not change.

7.14.6.2.2 Syntax

The syntax of the `ShiftRight` action type is shown in Figure 50.

7.14.6.2.3 Semantics

`ShiftRight` is a `ByteFieldModifier` which is an `ActionRealisation`. See 7.13.3 for details on the `Action` node type.

The properties of `ShiftRight` have the following semantics:

— `<byteField>: ByteFieldVariable [1]` (derived from `ByteFieldModifier`)

This identifies a `ByteField` variable which shall be shifted right.

— `<positions>: NumericTerm [1]`

It is the number of positions to shift. `Float` values shall be truncated.

Throws:

— `OutOfBoundsException`

It is thrown if the numer of positions to shift is a negative number.

IMPORTANT — The `ShiftRight` operation shall have no effect on an empty `ByteField`.

7.14.6.3 ShiftLeft

7.14.6.3.1 Description

Shifts a `ByteField` left. Bits at the least significant positions will be stuffed by zero. Bits at the most significant positions will be discarded. The size of the ByteField will not change.

7.14.6.3.2 Syntax

The syntax of the `ShiftLeft` action type is shown in Figure 50.

7.14.6.3.3 Semantics

`ShiftLeft` is a `ByteFieldModifier` which is an `ActionRealisation`. See 7.13.3 for details on the `Action` node type.

The properties of **ShiftLeft** have the following semantics:

— **<byteField>: ByteFieldVariable [1]** (derived from **ByteFieldModifier**)

This identifies a **ByteField** variable which shall be shifted left.

— **<positions>: NumericTerm [1]**

It is the number of positions to shift. **Float** values shall be truncated.

Throws:

— **OutOfBoundsException**

It is thrown if the numer of positions to shift is a negative number.

IMPORTANT — The ShiftLeft operation shall have no effect on an empty ByteField.

7.14.6.4 SetBit

7.14.6.4.1 Description

Sets a bit in a **ByteField**.

7.14.6.4.2 Syntax

The syntax of the **SetBit** action type is shown in Figure 50.

7.14.6.4.3 Semantics

SetBit is a **ByteFieldModifier** which is an **ActionRealisation**. See 7.13.3 for details on the **Action** node type.

The properties of **SetBit** have the following semantics:

— **<byteField>: ByteFieldVariable [1]** (derived from **ByteFieldModifier**)

This identifies a **ByteField** variable in which the bit shall be set.

— **<index>: NumericTerm [1]**

This is a term identifying the byte of the **ByteField** in which in the bit shall be set. **Float** values shall be truncated.

— **<position>: NumericTerm [1]**

This is a term representing the bit position in the byte chosen by **<index>**. **Float** values shall be truncated.

— **<value>: BooleanTerm [1]**

If the Boolean term given by **<value>** returns **true** the bit shall be set to 1, otherwise to 0.

Throws:

— **OutOfBoundsException**

It is thrown if the index is not within the range **[0,n-1]**, where **n** is the size of the **ByteField**.

— **OutOfBoundsException**

It is thrown if the bit position not within the range **[0,7]** of allowed bit positions.

7.14.6.5 ReplaceSubByteField

7.14.6.5.1 Description

Replaces parts of a `ByteField` at a given position by overwriting it with another `ByteField`.

7.14.6.5.2 Syntax

The syntax of the `ReplaceSubByteField` action type is shown in Figure 50.

7.14.6.5.3 Semantics

`ReplaceSubByteField` is a `ByteFieldModifier` which is an `ActionRealisation`. See 7.13.3 for details on the `Action` node type.

The properties of `ReplaceSubByteField` have the following semantics:

— `<byteField>: ByteFieldVariable [1]` (derived from `ByteFieldModifier`)

This identifies a `ByteField` variable in which part shall be replaced.

— `<newByteField>: ByteFieldTerm [1]`

This represents the new `ByteField` which shall overwrite the original `ByteField` starting at `<index>`.

— `<index>: NumericTerm [1]`

This is a numeric term representing the byte index in the original `ByteField` where the replacement shall start. `Float` values shall be truncated.

IMPORTANT — If the replacement exceeds the byte size of the original `ByteField` (size of original `ByteField` < index+size of new `ByteField`), then the original bytefield will be extended to the minimum size necessary to hold the excess bytes.

Throws:

— `OutOfBoundsException`

It is thrown if the index is not within the range `[0,n-1]`, where `n` is the size of the `ByteField`.

7.14.6.6 AppendByteField

7.14.6.6.1 Description

Appends `ByteField` values to the end of another `ByteField`.

7.14.6.6.2 Syntax

The syntax of the `AppendByteField` action type is shown in Figure 50.

7.14.6.6.3 Semantics

`AppendByteField` is a `ByteFieldModifier` which is an `ActionRealisation`. See 7.13.3 for details on the `Action` node type.

The properties of `AppendByteField` have the following semantics:

— `<byteField>: ByteFieldVariable [1]` (derived from `ByteFieldModifier`)

This identifies a `ByteField` variable to which the other `ByteField` values shall be appended.

— **<otherByteField>: ByteFieldTerm [1..*]**

This represents the **ByteField** values which shall be appended.

7.14.7 ListModifiers

7.14.7.1 Overview

ListModifier actions are designed for modifying lists. They shall only operate on **List** variables.

7.14.7.2 ListConcatenate

7.14.7.2.1 Description

This action concatenates lists. The modified list shall contain all original items in the front, plus shallow copies of the items of the other lists in the end. The lists shall be concatenated one after the other (in the order of appearance in the OTX document). The items shall be ordered like they were ordered in the original lists.

7.14.7.2.2 Syntax

The syntax of the **ListConcatenate** action type is shown in Figure 50.

7.14.7.2.3 Semantics

ListConcatenate is a **ListModifier** which is an **ActionRealisation**. See 7.13.3 for details on the **Action** node type.

The properties of **ListConcatenate** have the following semantics:

— **<list>: ListVariable [1]** (derived from **ListModifier**)

This identifies a **List** to which **shallow** copies of the items of the other lists shall be appended.

— **<otherList>: ListTerm [1..*]**

Shallow copies of the items in the lists given by **<otherList>** elements shall be appended to the end of **<list>**. The lists shall be concatenated one after the other (in the order of appearance in the OTX document).

Associated checker rules:

— Core_Chk031– type-safe ListConcatenate (see C.2.31).

7.14.7.3 ListAppendItems

7.14.7.3.1 Description

Appends one or more items to the end of a list.

7.14.7.3.2 Syntax

The syntax of the **ListAppendItems** action type is shown in Figure 50.

7.14.7.3.3 Semantics

ListAppendItems is a **ListModifier** which is an **ActionRealisation**. See 7.13.3 for details on the **Action** node type.

The properties of `ListAppendItems` have the following semantics:

— `<list>: ListVariable [1]` (derived from `ListModifier`)

This identifies a `List` variable to which the items shall be appended.

— `<item>: Term [1..*]`

Each `<item>` element given here shall be appended to the end of the `<list>`. Items are represented as terms which shall be evaluated first before the resulting value will be appended. Items shall be appended in the order of appearance in the OTX document.

Associated checker rules:

— Core_Chk032 – type-safe ListAppendItems (see C.2.32).

7.14.7.4 ListInsertItems

7.14.7.4.1 Description

Inserts one ore more items into a list at a defined position.

7.14.7.4.2 Syntax

The syntax of the `ListInstertItems` action type is shown in Figure 50.

7.14.7.4.3 Semantics

`ListInsertItems` is a `ListModifier` which is an `ActionRealisation`. See 7.13.3 for details on the `Action` node type.

The properties of `ListInsertItems` have the following semantics:

— `<list>: ListVariable [1]` (derived from `ListModifier`)

This identifies a `List` variable into which the items shall be inserted.

— `<index>: NumericTerm [1]`

This describes the position in the list where the items shall be inserted. The first item in a list shall have the index value 0. The items shall be inserted in front of the item that was originally located at the given index. `Float` values shall be truncated.

— `<item>: Term [1..*]`

Each `<item>` element given here shall be inserted to the `<list>`. Items are represented as terms which shall be evaluated first before the resulting value will be inserted. Items shall be inserted in the order of appearance in the OTX document.

Associated checker rules:

— Core_Chk033 – type-safe ListInsertItems (see C.2.33).

Throws:

— `OutOfBoundsException`

It is thrown if the index is not in the list (index < 0 or index > length-1 of list).

7.14.7.5 ListRemoveItems

7.14.7.5.1 Description

Removes one or more items from a list at a defined position.

7.14.7.5.2 Syntax

The syntax of the `ListRemoveItems` action type is shown in Figure 50.

7.14.7.5.3 Semantics

`ListRemoveItems` is a `ListModifier` which is an `ActionRealisation`. If index + count exceed the list length, the action shall remove as many elements as possible and shall not throw an exception. See 7.13.3 for details on the `Action` node type. The properties of `ListRemoveItems` have the following semantics:

— `<list>: ListVariable [1]` (derived from `ListModifier`)

This identifies a `List` variable from which items shall be removed.

— `<index>: NumericTerm [1]`

This describes the position where item removal starts. The first item in a list shall have the index value 0. `Float` values shall be truncated.

— `<count>: NumericTerm [1]`

This describes the number of items that shall be removed starting from the index. `Float` values shall be truncated.

Throws:

— `OutOfBoundsException`

It is thrown if the index is not in the list (index < 0 or index > length-1 of list) or count is negative (count < 0).

7.14.7.6 ListClear

7.14.7.6.1 Description

Removes all items from a `List`.

7.14.7.6.2 Syntax

The syntax of the `ListClear` action type is shown in Figure 50.

7.14.7.6.3 Semantics

`ListClear` is a `ListModifier` which is an `ActionRealisation`. See 7.13.3 for details on the `Action` node type. The properties of `ListClear` have the following semantics:

— `<list>: ListVariable [1]` (derived from `ListModifier`)

This identifies the `List` which shall be cleared.

7.14.8 MapModifiers

7.14.8.1 Overview

`MapModifier` actions are designed for modifying maps. They shall only operate on `Map` variables.

7.14.8.2 MapUnion

7.14.8.2.1 Description

Unites (merges) maps. The modified map shall contain all original items, plus shallow copies of the items of the other maps.

7.14.8.2.2 Syntax

The syntax of the `MapUnion` action type is shown in Figure 50.

7.14.8.2.3 Semantics

`MapUnion` is a `MapModifier` which is an `ActionRealisation`. See 7.13.3 for details on the `Action` node type.

The properties of `MapUnion` have the following semantics:

— `<map>: MapVariable [1]` (derived from `MapModifier`)

This identifies a `Map` to which the other maps shall be merged.

— `<otherMap>: MapTerm [1..*]`

This describes the maps which shall be merged to the `<map>`. To do so, all items of the maps given by the `<otherMap>` elements shall be shallow copied to `<map>`.

IMPORTANT — If the modified map already contains the key of a put item, then the corresponding value is overwritten.

Associated checker rules:

— Core_Chk023 – type-safe assignments (see C.2.23);

— Core_Chk042 – type-safe MapUnion (see C.2.42).

7.14.8.3 MapPutItems

7.14.8.3.1 Description

Puts one ore more map items into a map.

7.14.8.3.2 Syntax

The syntax of the `MapPutItems` action type is shown in Figure 50.

7.14.8.3.3 Semantics

`MapPutItems` is a `MapModifier` which is an `ActionRealisation`. See 7.13.3 for details on the `Action` node type.

The properties of `MapPutItems` have the following semantics:

— `<map>: MapVariable [1] (derived from MapModifier)`

This identifies a `Map` variable that shall receive the new items.

— `<item>: MapItem [1..*]`

Each `<item>` element given here shall be put into the `<map>`. The map items are represented as key/value pairs:

— `<key>: SimpleTerm [1]`

This describes the map item key by which the to-be-put item can be addressed later.

— `<value>: Term [1]`

This describes the map item value.

IMPORTANT — If the key of a put item already exists in the map, the new item value shall overwrite the original item value. This behaviour is similar to the `Assignment` action (`M1["key"]: = "value"`).

Associated checker rules:

— Core_Chk038 – type-safe MapPutItems (see C.2.38).

7.14.8.4 MapRemoveItems

7.14.8.4.1 Description

Removes a set of items from a map.

7.14.8.4.2 Syntax

The syntax of the `MapRemoveItems` action type is shown in Figure 50.

7.14.8.4.3 Semantics

`MapRemoveItems` is a `MapModifier` which is an `ActionRealisation`. See 7.13.3 for details on the `Action` node type.

The properties of `MapRemoveItems` have the following semantics:

— `<map>: MapVariable [1]` (derived from `MapModifier`)

This identifies a `Map` variable from which items shall be removed.

— `<key>: Term [1..*]`

Each `<key>` element describes a key of a map item that shall be removed.

Associated checker rules:

— Core_Chk039 – type-safe MapRemoveItems (see C.2.39).

Throws:

— `OutOfBoundsException`

It is thrown if a key does not exist in the map.

7.14.8.5 MapClear

7.14.8.5.1 Description

Removes all items from a `Map`.

7.14.8.5.2 Syntax

The syntax of the `MapClear` action type is shown in Figure 50.

7.14.8.5.3 Semantics

`MapClear` is a `MapModifier` which is an `ActionRealisation`. See 7.13.3 for details on the `Action` node type. The properties of `MapClear` have the following semantics:

— `<map>: MapVariable [1]` (derived from `MapModifier`)

This identifies the `Map` which shall be cleared.

7.15 Terms

7.15.1 Overview

OTX terms represent syntactic expressions which can be evaluated in order to yield a value. The resulting value of a term can be a simple value (e.g. an integer in the OTX `Integer` case) or a reference to complex data (e.g. a reference to a list in the OTX `List` case). Terms are required in various places in the data model, when a value needs to be computed which is then, for example, assigned to a variable, used as an input parameter for a procedure call, or used as the condition truth value in a branch.

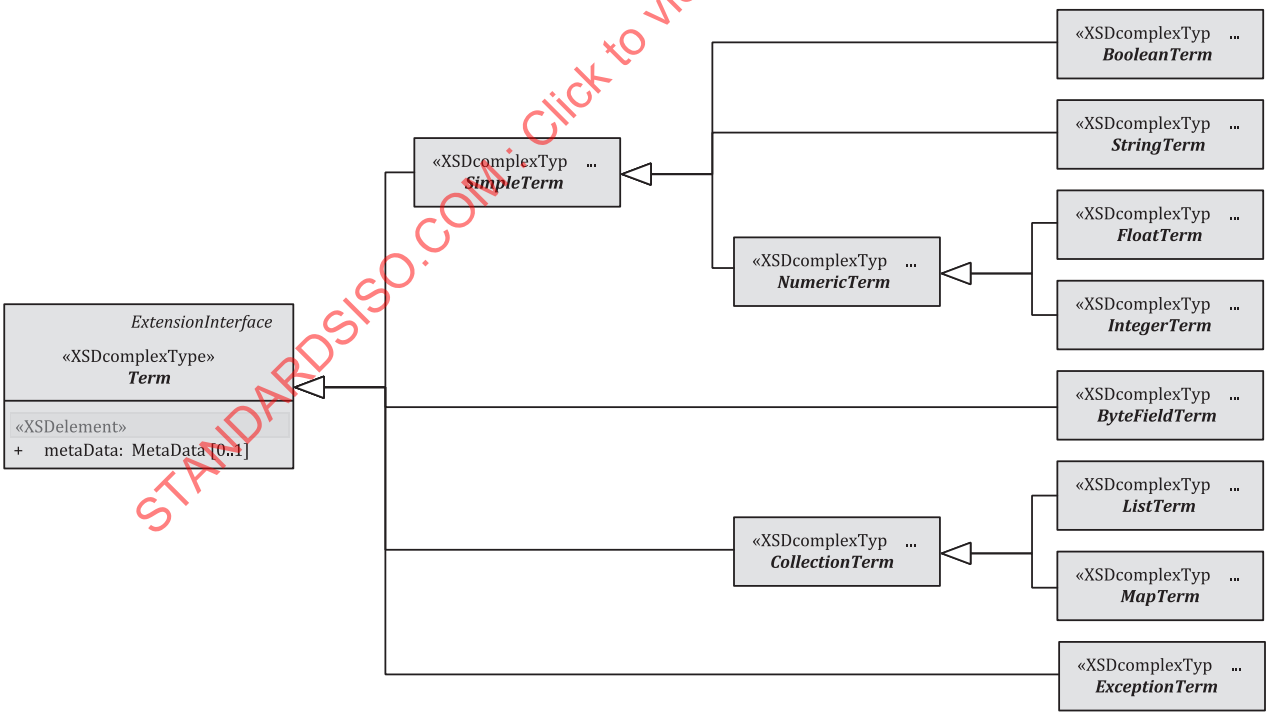

Figure 52 — Data model view: Term hierarchy

As shown in Figure 52, every OTX term is categorized according to its return type. The OTX syntax prescribes a term hierarchy; the leaf term categories (term to the right side of the hierarchy figure) correspond directly to the OTX data types, e.g. every `StringTerm` will return a `String` value.

The higher-level term categories bundle other term categories, e.g. all terms returning an **`Integer`** (**`IntegerTerm`**) and terms returning a **`Float`** (**`FloatTerm`**) are in the category **`NumericTerm`**. The higher-level categories are used in places where more then one data type is allowed, for example, in a for-each-loop, the collection over which the loop iterates may be a **`List`** or a **`Map`**, so the **`CollectionTerm`** is used which bundles both **`MapTerm`** and **`ListTerm`**.

There is also the high-level term **`SimpleTerm`** which serves to separate terms returning simple values from terms returning references.

NOTE In OTX applications, simple OTX data types can also be implemented as boxed types if necessary.

There are also concrete terms derived directly from a higher level category; this is required for terms where the returned data type is determined according to the term argument data types, e.g. the **`Add`** term which is a **`NumericTerm`** because it returns an **`Integer`** value if all summand arguments are **`Integer`**, but it returns a **`Float`** value if all summand arguments are **`Float`**.

All category term types shown in the figure are abstract types. The following subclauses specify all the concrete terms which are derived from the abstract terms, e.g. the **`NumericTerm`** **`Add`**, the **`ByteFieldTerm`** **`ShiftLeft`**.

IMPORTANT — OTX terms never have side-effects. This means that the arguments of a term should not be changed by term evaluation.

IMPORTANT — The abstract OTX core terms are extensible. This means that OTX extensions may derive new terms from any of the abstract term categories defined in the OTX core (see Annex D).

Due to the way terms are modelled in OTX, they are appearing in parsed form in OTX documents. This means that OTX terms do not have to be parsed anymore for computing the expression tree. As a consequence, OTX does not have to define any operator precedence rules to disambiguate terms (this is only needed in languages with expression strings in combination with infix-notation).

Since all terms are derived from the **`Term`** type, they may optionally contain a **`<metaData>`** element (as specified in 7.16.5).

7.15.2 Literal terms

7.15.2.1 Description

The simplest types of terms are the literal terms. They are a notation for representing a fixed value in the source code of an OTX document.

All OTX core data types (without the Exception type) provide a literal notation term. In the case of simple data types, a literal term returns the fixed value for simple data types at runtime. For complex data types, a reference to the **`List`** or **`Map`** object constructed out of the literal is returned.

7.15.2.2 Syntax

Figure 53 shows the syntax of the literal term types.

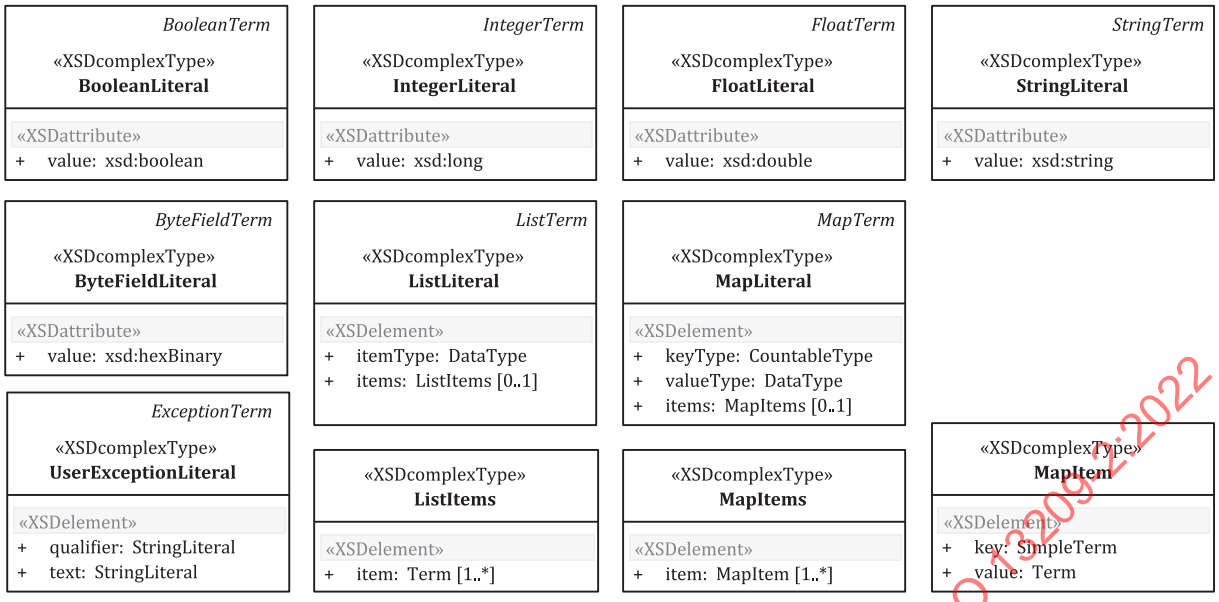

Figure 53 — Data model view: Literal terms

7.15.2.3 Semantics

7.15.2.3.1 Boolean Literal

`BooleanLiteral` is a `BooleanTerm`. It returns the `Boolean` value of the literal.

Its properties have the following semantics:

— `value: xsd:boolean [1]`

This attribute contains a fixed Boolean value. Refer to W3C XSD (all parts):2012 for an overview of allowed values for `xsd:boolean`.

7.15.2.3.2 IntegerLiteral

`IntegerLiteral` is an `IntegerTerm`. It returns the `Integer` value of the literal.

Its properties have the following semantics:

— `value: xsd:long [1]`

This attribute contains a fixed integer value. Refer to W3C XSD (all parts):2012 for an overview of allowed values for `xsd:long`.

7.15.2.3.3 FloatLiteral

`FloatLiteral` is a `FloatTerm`. It shall return the `Float` value of the literal.

Its properties have the following semantics:

— `value: xsd:double [1]`

This attribute contains a fixed double value. Refer to W3C XSD (all parts):2012 for an overview of allowed values for `xsd:double`.

7.15.2.3.4 StringLiteral

StringLiteral is a **StringTerm**. It shall return the **String** value of the literal.

Its properties have the following semantics:

— **value: xsd:string [1]**

This attribute shall contain the fixed string value. Refer to W3C XSD (all parts):2012 for an overview of allowed values for **xsd:string**.

7.15.2.3.5 ByteFieldLiteral

ByteFieldLiteral is a **ByteFieldTerm**. It shall return the **ByteField** value of the literal.

Its properties have the following semantics:

— **value: xsd:hexBinary [1]**

This attribute shall contain a fixed hexadecimal value or an empty **ByteField**. Refer to W3C XSD (all parts):2012 for an overview of allowed values for **xsd:hexBinary**.

7.15.2.3.6 ListLiteral

ListLiteral is a **ListTerm**. It shall return a new **List** created out of the literal list items.

By using the **ListLiteral** term, a list of **<item>** elements can be described which shall represent a literal notation of each list item. It is possible to describe nested item literal structures, e.g. a map of lists of integers. All literal items in a list shall be of the same datatype.

The **ListLiteral** properties have the following semantics:

— **<itemType>: DataType [1]**

This specifies the data type of all items in the list, in a flat or recursive way (list of strings, list of floats, list of lists of integers). For this definition, the **DataType** complex type itself is reused recursively. All items in the **<items>** element shall be of the type stated here.

Associated checker rules:

— Core_Chk034 – no use of init in list item type definition (see C.2.34).

— **<items>: ListItems [0..1]**

This is the wrapper element for the literal **List** items. If **<items>** is omitted, this shall represent the empty list.

— **<item>: Term [1..*]**

Each **<item>** shall represent an item in the list literal. The item literals need to be of the exact (deep) type stated in the **<itemType>** element (see above).

Associated checker rules:

— Core_Chk036 – ListLiteral and ListCreate item types follow item type definition (see C.2.36);

— Core_Chk037 – ListLiteral items are literal terms (see C.2.37).

7.15.2.3.7 MapLiteral

MapLiteral is a **MapTerm**. It shall return a new **Map** created out of the given literal **Map** items.

A **MapLiteral** term contains a list of **<item>** elements which shall represent a literal notation of each key/value pair of the map. It is possible to describe nested item literal structures, e.g. a map of lists of integers.

The **MapLiteral** properties have the following semantics:

— **<keyType>: CountableType [1]**

This defines the data type for the keys of all key/value pairs in the map. The key type shall be a simple, countable type. All keys in the **<items>** element shall be of the type stated here.

Associated checker rules:

— Core_Chk044 – no use of init in map key type definition (see C.2.44).

— **<valueType>: DataType [1]**

This defines the data type for all values of the key/value pairs in the map, flat or recursive (map of strings, map of list of floats, map of maps of integers). For this definition, the **DataType** complex type itself is reused recursively. All values in the **<items>** element shall be of the type stated here.

Associated checker rules:

— Core_Chk045 – no use of init in map value type definition (see C.2.45).

— **<items>: MapItems [0..1]**

This is the wrapper element for the literal **Map** items. If **<items>** is omitted, this shall represent the empty map.

— **<item>: MapItem [1..*]**

This represents key/value pair items in the map literal.

— **<key>: SimpleTerm [1]**

This is the key literal of the map item literal.

— **<value>: Term [1]**

This is the value literal of the map item literal.

Associated checker rules:

— Core_Chk047 – MapLiteral and MapCreate key&value types follow key&value type definition (see C.2.47);

— Core_Chk048 – MapLiteral items are literal terms (see C.2.48);

— Core_Chk043 – unique keys in MapLiteral (see C.2.43).

7.15.2.3.8 UserExceptionLiteral

UserExceptionLiteral is an **ExceptionTerm**. The term shall be used for creating a customized exception (a **UserException**). The most prominent uses of this term are the initialization in **UserException** declarations as well as the explicit **Throw** node which allows for the author to customize and throw such an exception.

The properties of **UserExceptionLiteral** have the following semantics:

— **<qualifier>: StringLiteral [1]**

This string literal allows for the author to provide a short qualifier for the exception initialization. A qualifier shall be used to categorize an exception.

— **<text>: StringLiteral [1]**

This string literal allows for the author to provide a text which characterizes the exception's reason.

7.15.2.4 Example

The two examples below show literal terms for all types. The terms are embedded in an **Assignment** action node here, they can appear in other places like loop conditions or procedure call arguments also.

The first example shows the simple data type literal terms, the second example shows the complex type literal terms.

Sample of simple type literal terms

```
<action id="a1">
  <realisation xsi:type="Assignment">
    <result xsi:type="BooleanVariable" name="b" />
    <term xsi:type="BooleanLiteral" value="true" />
  </realisation>
</action>

<action id="a2">
  <realisation xsi:type="Assignment">
    <result xsi:type="IntegerVariable" name="i" />
    <term xsi:type="IntegerLiteral" value="42" />
  </realisation>
</action>

<action id="a3">
  <realisation xsi:type="Assignment">
    <result xsi:type="FloatVariable" name="f" />
    <term xsi:type="FloatLiteral" value="42.0815" />
  </realisation>
</action>

<action id="a4">
  <realisation xsi:type="Assignment">
    <result xsi:type="StringVariable" name="s" />
    <term xsi:type="StringLiteral" value="Hello World!!!" />
  </realisation>
</action>
```

Sample of complex type literal terms

```
<action id="a5">
  <realisation xsi:type="Assignment">
    <result xsi:type="ByteFieldVariable" name="bf" />
    <term xsi:type="ByteFieldLiteral" value="002A" />
  </realisation>
</action>

<action id="a6">
  <realisation xsi:type="Assignment">
    <result xsi:type="ListVariable" name="myListOfInts" />
    <term xsi:type="ListLiteral">
      <itemType xsi:type="Integer"/>
      <items>
        <item xsi:type="IntegerLiteral" value="4" />
        <item xsi:type="IntegerLiteral" value="2" />
        <item xsi:type="IntegerLiteral" value="1" />
      </items>
    </term>
  </realisation>
</action>

<action id="a7">
  <realisation xsi:type="Assignment">
    <result xsi:type="MapVariable" name="myMapOfIntegers" />
    <term xsi:type="MapLiteral">
      <keyType xsi:type="String"/>
      <valueType xsi:type="Integer"/>
      <items>
        <item>
          <key xsi:type="StringLiteral" value="stdNumber"/>
          <value xsi:type="IntegerLiteral" value="13209" />
        </item>
        <item>
          <key xsi:type="StringLiteral" value="stdDocumentParts"/>
          <value xsi:type="IntegerLiteral" value="3" />
        </item>
      </items>
    </term>
  </realisation>
</action>
```

7.15.3 Dereferencing terms

7.15.3.1 Description

Dereferencing terms are used for reading the value of variables or constants or values which are contained inside of a complex structure like a list or map.

Since the syntax and semantics of all dereferencing terms are similar for all data types (only the data types of the returned data differ), a general description applying to all types is provided hereby.

7.15.3.2 Syntax

Figure 54 shows the syntax of the dereferencing term types.

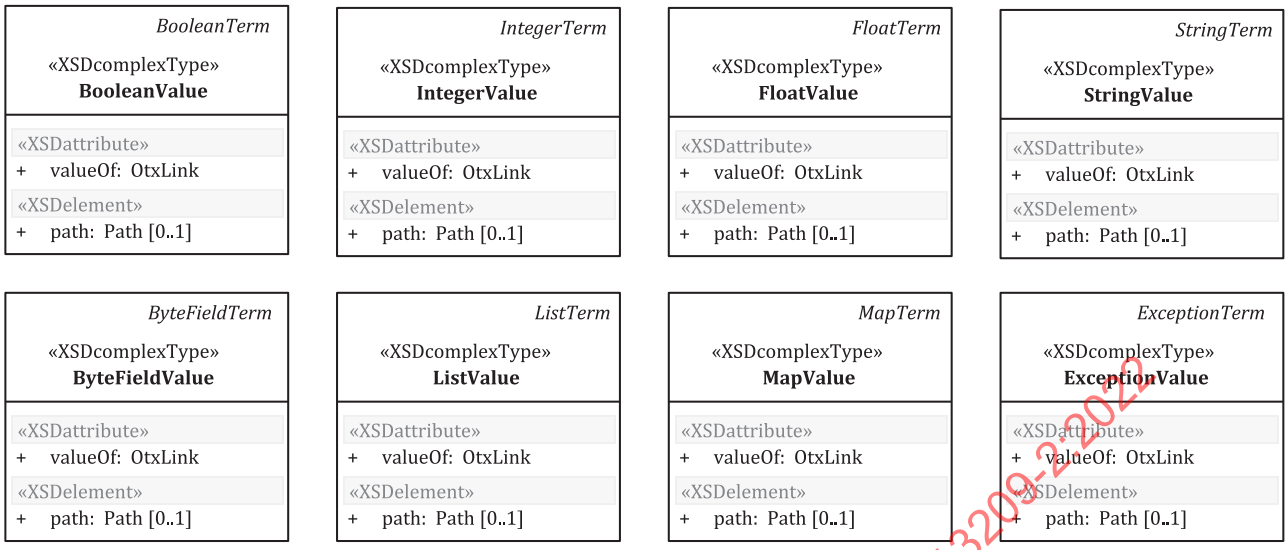

Figure 54 — Data model view: Dereferencing terms

7.15.3.3 Semantics

The properties of all addressing terms have the following semantics:

— **`valueOf: OtxLink [1]`**

  This contains the name of the variable, constant or parameter where the value is stored. If the value of interest resides deeper within a complex data structure like a **`List`** or **`Map`**, the **`<path>`** element shall be utilized to address the data inside of the structure.

— **`<path>: Path [0..1]`**

  The element addresses parts of complex structures like **`List`** or **`Map`**. It is built out of a series of index- and name-steps which allow navigate into the structure:

  — **`<stepByIndex>: NumericTerm [1]`**

    This step of a **`<path>`** shall be used for locations addressed by index. Items in a **`List`** and items in a **`Map`** with **`Integer`** keys shall be addressed in this way.

  — **`<stepByName>: StringTerm [1]`**

    This step of a **`<path>`** shall be used for addressing locations by name. Items in a **`Map`** with **`String`** keys shall be addressed in this way.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see C.2.53);

— Core_Chk050 – type-safe variable and constant usage (see C.2.50).

Throws:

— **`OutOfBoundsException`**

  It is thrown only if a **`<path>`** is set: the **`<path>`** points to a location which does not exist (like a list index exceeding list length, or a map key which is not part of the map).

— **InvalidReferenceException**

It is thrown for **ExceptionValue** only: if the variable value is not valid (this can happen since there is no implicit initialization for the **Exception** types).

NOTE In a graphical OTX editor, variable value terms can be displayed in the common notation form used by all major programming languages. For example, for a **BooleanValue** term for a variable b, the display in a graphical OTX editor simply shows b; a term with a **<path>** pointing to the item at index i in a list L of **Booleans**, L[i] In a graphical OTX editor be displayed, etc.

7.15.3.4 Example

Sample of dereferencing terms

```
<branch id="branch1">
  <realisation>
    <if>
      <condition id="branch1condition">
        <specification>Is_myListOfBool[i]_true?</specification>
        <realisation xsi:type="BooleanValue" valueOf="myListOfBool">
          <path>
            <stepByIndex xsi:type="IntegerValue" valueOf="i" />
          </path>
        </realisation>
      </condition>
      <flow>
        <action id="a1" />
      </flow>
    </if>
  </realisation>
</branch>
```

The example above shows a **BooleanValue** term which is used in a **Branch** node condition. By the **valueOf** attribute and the **<path>** element, the **Boolean** value at index **i** is read from **myListofBool** (a list of Booleans). Note that the **<stepByIndex>** element in the **<path>** is a dereferencing term as well (**IntegerValue**).

7.15.4 Creation terms

7.15.4.1 Description

There are cases when a **List**, a **Map** or a **UserException** have to be created dynamically at runtime. Since literal terms cannot be used for that task (because the item literals are fix in the source code, not dynamic), OTX provides a set of creation terms.

7.15.4.2 Syntax

Figure 55 shows the syntax of the creation term types.

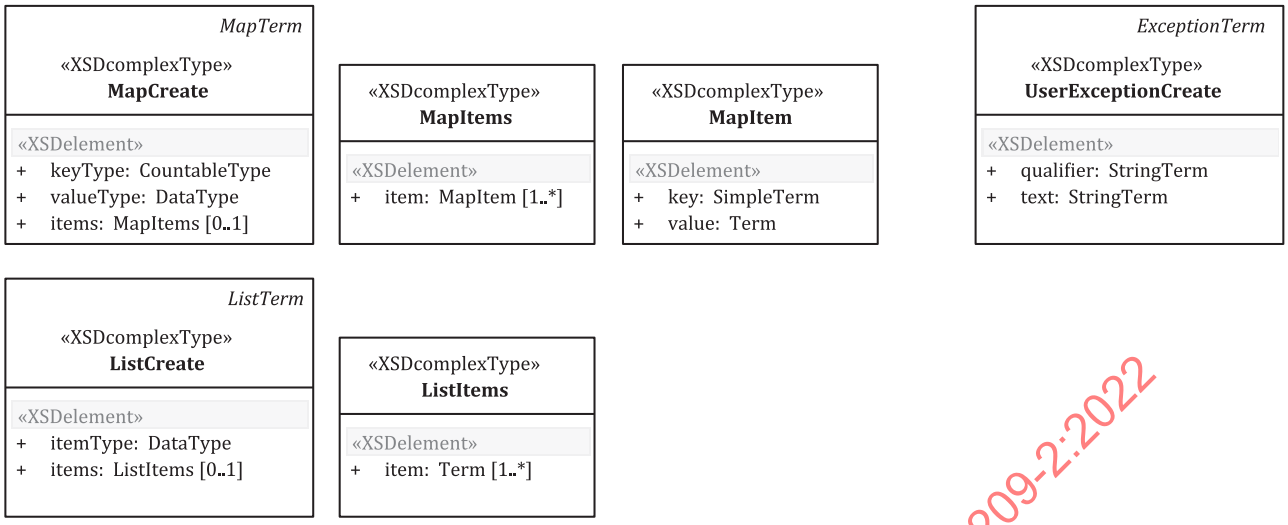

Figure 55 — Data model view: Creation terms

Note that the types `ListCreate` and `MapCreate` are of `<xsd:choice> [0..1]` content-type. For the `ListCreateTerm`, this means that authors can either use the `<otherList>` element or a list of `<item>` elements, but not both. Also, for the `MapCreateTerm`, either the `<otherMap>` element or a list of `<item>` elements can be used, but not both.

7.15.4.3 Semantics

7.15.4.3.1 ListCreate

`ListCreate` is a `ListTerm` that allows for creating a new list which is created by appending a given set of items to it (shallow copies). The `ListCreateTerm` returns the newly created list.

`ListCreate` is of `<xsd:choice> [0..1]` content-type. The choices of `ListCreate` have the following semantics:

— `<itemType>: DataType [1]`

Specifies the data type of all items in the list, in a flat or recursive way (list of strings, list of floats, list of lists of integers ...). For this definition, the `DataType` complex type itself is reused recursively. All items in the `<items>` element have to be of the type stated here.

Associated checker rules:

— Core_Chk034 – no use of init in list item type definition (see C.2.34).

— `<items>: ListItems [0..1]`

This is the wrapper element for the `List` items. If `<items>` is omitted, this shall represent the empty list.

— `<item>: Term [1..*]`

Each `<item>` shall represent an item in the newly created list. The items need to be of the exact (deep) type stated in the `<itemType>` element (see above).

Associated checker rules:

— Core_Chk036 – ListLiteral and ListCreate item types follow item type definition (see C.2.36).

7.15.4.3.2 MapCreate

MapCreate is a **MapTerm** that allows for creating a new map which is created by putting a given set of key/value items to it (shallow copies). The **MapCreateTerm** returns the newly created list.

MapCreate is of **<xsd:choice>** **[0..1]** content-type. The choices of **MapCreate** have the following semantics:

— **<keyType>: CountableType [1]**

This defines the data type for the keys of all key/value pairs in the map. The key type shall be a simple, countable type. All keys in the **<items>** element shall be of the type stated here.

Associated checker rules:

— Core_Chk044 – no use of init in map key type definition (see C.2.44);

— Core_Chk045 – no use of init in map value type definition (see C.2.45).

— **<valueType>: DataType [1]**

This defines the data type for all values of the key/value pairs of the newly created map, flat or recursive (map of strings, map of list of floats, map of maps of integers). For this definition, the **DataType** complex type itself is reused recursively. All values in the **<items>** element shall be of the type stated here.

Associated checker rules:

— Core_Chk045 – no use of init in map value type definition (see C.2.45).

— **<items>: MapItems [0..1]**

This is the wrapper element for the **Map** items. If **<items>** is omitted, this shall represent the empty map.

— **<item>: MapItem [1..*]**

This represents key/value pair items of the newly created map.

— **<key>: SimpleTerm [1]**

This is the key of the map item.

— **<value>: Term [1]**

This is the value of the map item.

Associated checker rules:

— Core_Chk047 – MapLiteral and MapCreate key&value types follow key&value type definition (see C.2.47).

Throws:

— **OutOfBoundsException**

It is thrown if two or more items in the literal have the same key.

7.15.4.3.3 UserExceptionCreate

UserExceptionCreate is an **ExceptionTerm** which is used for creating a customized exception (a **UserException**). The most prominent use of this term is the explicit **Throw** node which allows for the author to customize and throw such an exception. The term returns a reference to the newly created exception.

The properties of **UserExceptionCreate** have the following semantics:

— **<qualifier>: StringTerm [1]**

This allows for the author to provide a short string qualifier for the customized exception. A qualifier may be used to categorize an exception.

— **<text>: StringTerm [1]**

This allows for the author to provide a text which characterizes the reason for the exception.

7.15.4.4 Example

The example shows the creation of a **List**, a **Map** and a **UserException**.

The **List** is created out of to literal integers **10** and **20** and the value of **i** (known only at runtime). The **Map** is created out of two literal key/value items. The **UserException** is created in a **Throw** node, with a qualifier and a text defined by **StringLiteral** terms.

Sample of creation terms

```
<action id="a8">
  <specification>Creates a list out of three items 10, 20, i </specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="ListVariable" name="myNewList" />
    <term xsi:type="ListCreate">
      <itemType xsi:type="Integer" />
      <items>
        <item xsi:type="IntegerLiteral" value="10" />
        <item xsi:type="IntegerLiteral" value="20" />
        <item xsi:type="IntegerValue" valueOf="i" />
      </items>
    </term>
  </realisation>
</action>

<action id="a9">
  <specification>Creates a map with items "stdNumber":13209 and "stdDocumentParts":3 </specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="MapVariable" name="myNewMap" />
    <term xsi:type="MapCreate">
      <keyType xsi:type="String" />
      <valueType xsi:type="Integer" />
      <items>
        <item>
          <key xsi:type="StringLiteral" value="stdNumber" />
          <value xsi:type="IntegerLiteral" value="13209" />
        </item>
        <item>
          <key xsi:type="StringLiteral" value="stdDocumentParts" />
          <value xsi:type="IntegerLiteral" value="3" />
        </item>
      </items>
    </term>
  </realisation>
</action>

<throw id="t1">
  <specification>Throws a customized exception</specification>
  <realisation xsi:type="UserExceptionCreate">
    <qualifier xsi:type="StringLiteral" value="Dire Exception" />
    <text xsi:type="StringLiteral" value="Something bad happened!" />
  </realisation>
</throw>
```

7.15.5 Conversion terms

7.15.5.1 Description

This category of terms is required to convert values of one data type to another data type. Since conversion depends on the data types and the conversion direction, special rules apply for each conversion term type.

Since the syntax and general semantics of all conversion terms are similar for all data types, a general description applying to all types is provided first, followed by the special rules for each data type conversion.

NOTE Since no use cases for converting to **`List`**, **`Map`** or **`Exception`** types were identified, no conversion terms exist for these types.

7.15.5.2 Syntax

Figure 56 shows the syntax of the conversion term types.

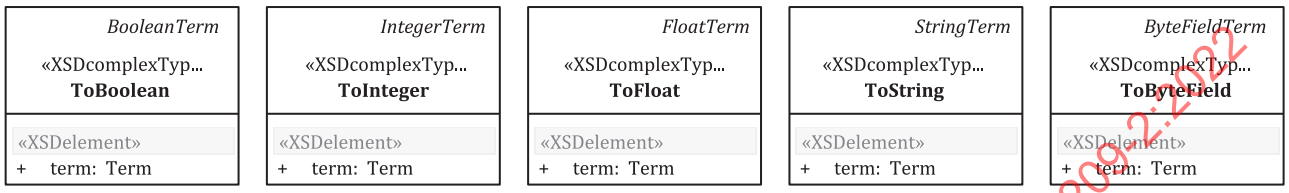

Figure 56 — Data model view: Conversion terms

7.15.5.3 Semantics

Every conversion term describes the value that shall be converted with a **`<term>`** element:

— **`<term>: Term [1]`**

This represents the value that shall be converted to a value of a particular data type.

IMPORTANT — The conversion terms are only specified herein for `Boolean`, `Integer`, `Float`, `String` and `ByteField` arguments. For all other core OTX data types, the behaviour is unspecified (except for the `ToString` conversion, see below). OTX applications may provide custom implementations of the conversion terms for other data types. If a conversion term is applied to data of a type for which no implementation exists, a runtime exception shall be thrown.

7.15.5.3.1 ToBoolean

This term shall return the **`Boolean`** counterpart of the argument term. The conversion is specified for the following data types (the behaviour for other OTX core types is unspecified).

— **`Integer`** Returns **`false`** if `0`, otherwise **`true`**

— **`Float`** Returns **`false`** if `0.0`, otherwise **`true`**.

— **`ByteField`** Returns **`false`** if empty, otherwise **`true`**.

— **`String`** Returns **`true`** if the string value is `"true"`, **ignoring case**, otherwise **`false`**.

— **`Boolean`** Returns **`the copy of the value`**.

7.15.5.3.2 ToInteger

This term shall return the **`Integer`** counterpart of the argument term. The conversion is specified for the following data types (the behaviour for other types is unspecified).

— **`Boolean`** Returns `1` if the value is **`true`**, otherwise `0`.

— **`Float`** Returns the integer part (the digits to the left of the decimal point – truncation).

— **ByteField** Returns the integer value of the **bytefield**. For interpretation of the **bytefield**, little endian byte order and **n**-bit two's complement shall be used, where **n** is the **bytefield's** size multiplied by 8 (i.e. The total number of bits in the original **bytefield**). For finer control of the byte order used for conversions, see **decodeinteger** term in 7.15.6.3.1.

— **String** Returns the integer value of the string. The string shall constitute an integer literal, as specified by Reference [9], subclause 3.10.1, otherwise an exception is thrown.

IMPORTANT — The reference implementation for this conversion is the Java method `java.lang.Long.valueOf()`. There shall be no deviation from this implementation.

— **Integer** Returns the copy of the value.

Throws:

— **OutOfBoundsException**

It is thrown in the **ByteField** case only: if the **ByteField** is empty or its size is greater than 8 bytes (in that case, the value space of **xsd:long** is exceeded, see Annex A). Conversions from **Float** INF, -INF or NAN shall result this exception, or if the value exceeds the value range of **Integer**.

— **TypeMismatchException**

It is thrown in the **String** case only: if the **String** does not constitute an integer value literal.

7.15.5.3.3 ToFloat

This term shall return the **Float** counterpart of the argument term. The conversion is specified for the following data types (the behaviour for other OTX core types is unspecified).

— **Boolean** Returns `1.0` if the value is **true**, otherwise `0.0`.

— **Integer** Returns a float copy of the integer value.

— **ByteField** Returns the float value of the **ByteField** (interpretation according to IEEE 754:2019). The result shall be stored in double precision, regardless of the argument being a single precision (32-bit) or a double precision (64-bit) **ByteField**. For interpretation of the ByteField, little endian byte order shall be used.

— **String** Returns the float value of the string. The string shall constitute a float literal, as specified by Reference [9], subclause 3.10.2, otherwise an exception is thrown.

IMPORTANT — The reference implementation for this conversion is the Java method `java.lang.Double.valueOf()`. There shall be no deviation from this implementation.

— **Float** Returns the copy of the value.

Throws:

— **TypeMismatchException**

It is thrown in the **String** case only: if the **String** does not constitute a float value literal.

— **OutOfBoundsException**

It is thrown in the **ByteField** case only: if the **ByteField**'s bit-size is neither 32 bit nor 64 bit (i.e. the value cannot be interpreted as single or double precision according to IEEE 754:2019).

7.15.5.3.4 ToByteField

This term shall return the `ByteField` counterpart of the argument term. The conversion is specified for the following data types (the behaviour for other types is unspecified).

— `Boolean` Returns `0x01` if the value is `true`, otherwise `0x00`.

— `Integer` Returns the smallest possible two's complement representation of the integer (using the smallest possible number of bytes). Concerning byte order in the resulting `ByteField`, little endian shall be used. With this, e.g. `127` converts to 1-byte 01111111_2 (`0x7F`), `-127` to 10000001_2 (`0x81`), whereas `6719` converts to 2-byte 00111111_2 00011010_2 (`0x3F 0x1A`), `-129` to 01111111_2 11111111_2 (`0x7F 0xFF`), and so on. Therefore, the size of the resulting `ByteField` depends on the input integer's magnitude. For explicit control over integer encoding, the `EncodeInteger` term specified in 7.15.6.3.2 should be used.

— `Float` Returns the 64-bit double precision encoded value according to IEEE 754:2019. Concerning byte order in the resulting `ByteField`, little endian shall be used.

— `String` Returns the UTF-8 encoded value.

— `ByteField` Returns the copy of the value.

7.15.5.3.5 ToString

This term shall return the `String` counterpart of the argument term. The conversion is specified for the following data types (the behaviour for other OTX core types is unspecified):

— `Boolean` Returns `"true"` if the value is `true`, otherwise `"false"`.

— `Integer` Returns the value in **decimal** string representation.

— `Float` Returns the value in **decimal** string representation. For an overview about string representations of floats see Reference [9], subclause 3.10.2.

IMPORTANT — The reference implementation for this conversion is the Java method `java.lang.Double.toString()`. There shall be no deviation from its specification.

— `String` Returns the copy of the value.

— `ByteField` Returns the UTF-8 interpretation.

Associated checker rules:

— Core_Chk062 – usage of string:Decode instead of ToString for ByteFields (see C.2.62).

All invalid bytes related to the current encoding shall translated to the so-called Unknown Character sign "□" (0xEFBFBD). In this case a re-conversion into a byte field will result in a different value, see sample below:

```
MyString = ToString(&66FC72); -> "f□r"
MyByteField = ToByteField(MyString); -> &66EFBFBD72
```

IMPORTANT — `ToString` shall not cause a runtime exception (see 7.15.5.3). For this, the `ToString` conversion shall be custom implemented for all other OTX data types in a way appropriate for the specific data type[8]. A `ToString` conversion applied, for example, onto an arbitrary `List` of `Integers` should yield a result similar to `"{4;12;13}"`. This allows for, e.g. writing debug messages containing serialized complex data during test sequence development.

8) The conversion term `ToString` represents an analogy to the Java method `java.lang.Object.toString()` which is designed to be overridden by new classes (see §4.3.2 of *The Java™ Language Specification* [9]).

Throws:

— `OutOfBoundsException`

It is thrown in the `ByteField` case only: if the value cannot be interpreted as string in UTF-8 format.

This exception is listed here for compatibility with the 2012 edition of the standard. It will never be thrown, as specified above.

7.15.5.4 Example

The example below shows a two-step conversion in an `Assignment` action: a float literal containing `Pi` is converted to an integer which is then converted to a string. The string is assigned to a variable `s` will be `"3"` in this example.

Sample of conversion terms

```
<action id="a10">
  <specification>Converts a float literal to integer, then to string</specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="StringVariable" name="s" />
    <term xsi:type="ToString">
      <term xsi:type="ToInteger">
        <term xsi:type="FloatLiteral" value="3.14159265358979323846264338327 95" />
      </term>
    </term>
  </realisation>
</action>
```

7.15.6 Integer conversion terms

7.15.6.1 Description

This category of terms allows converting between `Integer` and `ByteField` values. Compared to the generic conversion terms `ToInteger(ByteField)` and `ToByteField(Integer)` as described in 7.15.5, integer conversion terms allow controlling the `Bytefield` size, encoding type for negative integers and byte order which shall be used for this special kind of conversions.

7.15.6.2 Syntax

Figure 57 shows the syntax of the integer conversion term types.

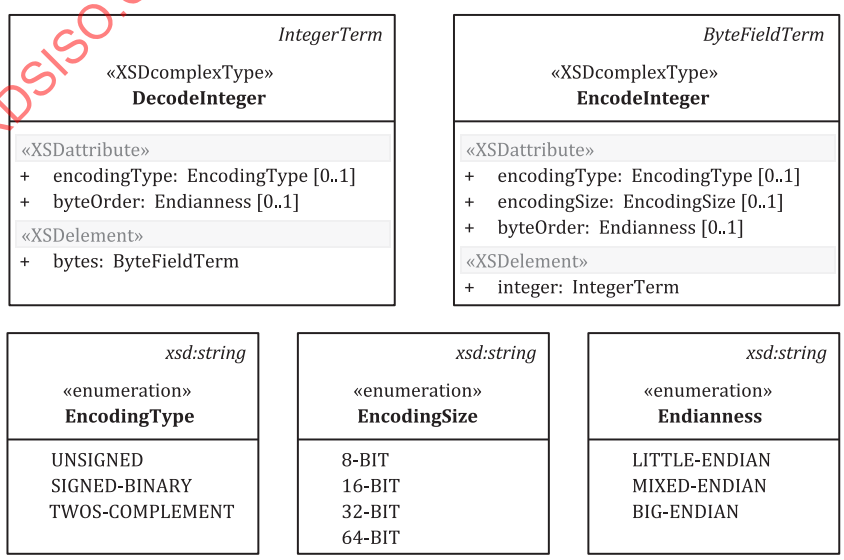

Figure 57 — Data model view: Integer conversion terms

7.15.6.3 Semantics

7.15.6.3.1 DecodeInteger

DecodeInteger is an **IntegerTerm**. It returns the decoded **Integer** value of the **ByteField** argument. The term provides control over the byte order which shall be used for decoding as well as the way how the **ByteField** shall be decoded regarding **n**-bit unsigned, **n**-bit signed binary or **n**-bit two's complement interpretation, where **n** is the **ByteField**'s size multiplied by 8 (in other words: the total number of bits in the original **ByteField**). Each **ByteField** size between 1 and 8 is possible.

— **encodingType: EncodingType = {UNSIGNED|SIGNED-BINARY|TWOS-COMPLEMENT} [1]**

This specifies how the bytes of the **ByteField** shall be interpreted (after respecting byte order):

— **UNSIGNED**: the **ByteField** shall be interpreted as an unsigned integer value. With this interpretation, a **ByteField** of, for example, 11111110_2, is decoded as **254**.

— **SIGNED-BINARY**: the most significant bit in the **ByteField** shall be interpreted as the sign and the rest as the binary encoded absolute integer value. With this interpretation, a **ByteField** of, for example, 11111110_2 is decoded as **-126**; a **BIG-ENDIAN ByteField 0x80007E** is also decoded as **-126**.

— **TWOS-COMPLEMENT** (default): the **ByteField** shall be interpreted according to the most widely used encoding called two's complement. With this interpretation, a **ByteField** of, for example, 11111110_2 is decoded as **-2**; a **BIG-ENDIAN ByteField 0xFFFF82** is decoded as **-126**.

— **byteOrder: Endianness = {LITTLE-ENDIAN|MIXED-ENDIAN|BIG-ENDIAN} [1]**

This specifies the byte order which shall be respected prior to decoding the **ByteField** value. The most widely used byte order **LITTLE-ENDIAN** is the default. The byte order of MIXED-ENDIAN is unspecified. It depends on the runtime system which of the following two orders are used:

— **MSB_FIRST_MSW_LAST ;**

— **MSB_LAST_MSW_FIRST .**

NOTE For explanation see Table 2.

— **<bytes>: ByteFieldTerm [1]**

This represents the **ByteField** value which shall be converted to **Integer**.

Throws:

— **OutOfBoundsException**

It is thrown if the **ByteField** is empty or the value space of **xsd:long** could be exceeded (see Annex A). This happens if the **ByteField**'s size is greater than 8 bytes, or when using **UNSIGNED** interpretation on an 8 byte **ByteField** in which the most significant bit is **1**. An **OutOfBoundsException** shall also occur if the byte order is **MIXED-ENDIAN** and the **ByteField** size is odd.

7.15.6.3.2 EncodeInteger

EncodeInteger is a **ByteFieldTerm**. It returns a **ByteField** which contains the encoded value of the original **Integer** argument. For the encoding, **n**-bit unsigned, **n**-bit signed-binary or **n**-bit two's complement representation of the integer can be chosen, where **n** shall be chosen out of **8**, **16**, **32** and **64** bit. The term also provides control over the byte order which shall be used for encoding.

— **encodingType: EncodingType = {UNSIGNED|SIGNED-BINARY|TWOS-COMPLEMENT} [0..1]**

This specifies how the **Integer** value shall be encoded (examples using **BIG-ENDIAN** byte order):

— **UNSIGNED**: the **Integer** shall be written to the **ByteField**, using all available bits (including the sign-bit). For negative integers, its absolute value shall be encoded. With this encoding, an **Integer** value of, for example, . **129** is encoded as 10000001_2, and an **Integer** value of, e.g. **-129** will also be encoded as 10000001_2.

— **SIGNED-BINARY**: with this encoding, the first bit in the resulting bytes shall mark the sign; the other bits encode the absolute value. According to this, for example, **129** is encoded as 00000000_2 10000001_2, whereas, e.g. **-129** will be encoded as 10000000_2 10000001_2. Two bytes are required in the example, since the sign-bit reduces the value space.

— **TWOS-COMPLEMENT** (default): the **Integer** shall be encoded according to the most widely used encoding called two's complement. With this encoding, an **Integer** value of, for example, **129** is encoded as 00000000_2 10000001_2, whereas a negative **Integer** value of, e.g. **-129** will be encoded as 11111111_2 01111111_2. Two bytes are required here, since the sign-bit reduces the value space.

— **encodingSize: EncodingSize = {8-BIT|16-BIT|32-BIT|64-BIT} [0..1]**

This specifies how many bits shall be used for the encoding. This shall be used to restrict the size of the resulting **ByteField** which can be useful if, e.g. only small integers are encoded, or if it is required that resulting **ByteFields** always have the same specified size. However, if an encoded integer requires more bits than specified by this attribute, an exception will be raised. Since the OTX **Integer** datatype is 64 bit, an encoding size of **64-BIT** is the default value.

— **byteOrder: Endianness = {LITTLE-ENDIAN|MIXED-ENDIAN|BIG-ENDIAN} [0..1]**

This specifies the byte order which shall be used when writing the encoded **Integer** value into the resulting **ByteField**. The most widely used byte order **LITTLE-ENDIAN** is the default. The byte order of MIXED-ENDIAN is unspecified. It depends on the runtime system which of the following two orders are used:

— **MSB_FIRST_MSW_LAST** ;

— **MSB_LAST_MSW_FIRST** .

Table 2 — Byte order (8 Byte values) - memory data deposition

Byte Order	**Increasing address – >**							
	n	**n+1**	**n + 2**	**n + 3**	**n+4**	**n+5**	**n + 6**	**n + 7**
MSB_FIRST_MSW_LAST	$Byte_1$	$Byte_0$	$Byte_3$	$Byte_2$	$Byte_5$	$Byte_4$	$Byte_7$	$Byte_6$
MSB_LAST_MSW_FIRST	$Byte_6$	$Byte_7$	$Byte_4$	$Byte_5$	$Byte_2$	$Byte_3$	$Byte_0$	$Byte_1$
MSB_FIRST	$Byte_7$	$Byte_6$	$Byte_5$	$Byte_4$	$Byte_3$	$Byte_2$	$Byte_1$	$Byte_0$
MSB_LAST	$Byte_0$	$Byte_1$	$Byte_2$	$Byte_3$	$Byte_4$	$Byte_5$	$Byte_6$	$Byte_7$
NOTE $Byte_n$ = = Most Significant Byte $Byte_0$ = = Least Significant Byte MSB_FIRST = = Motorola / BIG_ENDIAN MSB_LAST = = Intel / LITTLE_ENDIAN								

— **<integer>: IntegerTerm [1]**

This sepresents the **Integer** value which shall be converted to **ByteField**.

Throws:

— **OutOfBoundsException**

It is thrown if the number of bits required for encoding the **Integer** value exceeds the number of bits specified by the **encodingSize** attribute (overflow).

7.15.6.4 Example

— The example below shows two integer conversions: the `Integer` value `-42` is encoded into the `ByteField` named `"bytes"`, using `BIG-ENDIAN` byte order and `TWOS-COMPLEMENT` encoding. After execution of the `Assignment` action, `"bytes"` will have a value of 11111111_2 11010110_2. The second action converts the `ByteField` with value 10010101_2 (`0x95`) to `Integer` by using `LITTLE-ENDIAN`, and `UNSIGNED` interpretation. After execution, "`i`" will have a value of `149`.

Sample of integer conversion terms

```
<action id="a1">
    <specification>Converts a integer value to ByteField</specification>
    <realisation xsi:type="Assignment" >
        <result xsi:type="ByteFieldVariable" name="bytes" />
        <term xsi:type="EncodeInteger" byteOrder="BIG-ENDIAN" encodingSize="16-BIT">
          <integer xsi:type="IntegerLiteral" value="-42" />
        </term>
    </realisation>
</action>

<action id="a2">
    <specification>Converts a ByteField value to Integer</specification>
    <realisation xsi:type="Assignment" >
        <result xsi:type="IntegerVariable" name="i" />
        <term xsi:type="DecodeInteger" encodingType="UNSIGNED">
          <bytes xsi:type="ByteFieldLiteral" value="95" />
        </term>
    </realisation>
</action>
```

7.15.7 Logic operations

7.15.7.1 Description

The logic operation terms provide the most basic and widely used logical connectives for the Boolean algebra, namely conjunction (`LogicAnd`), disjunction (`LogicOr`), exclusive disjunction (`LogicXor`) and negation (`LogicNot`).

7.15.7.2 Syntax

Figure 58 shows the syntax of the logic operation term types.

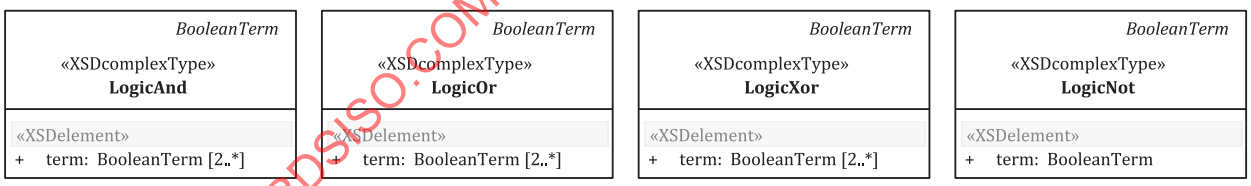

Figure 58 — Data model view: Logic operations

7.15.7.3 Semantics

7.15.7.3.1 LogicAnd

`LogicAnd` is a `BooleanTerm` which represents the Boolean conjunction. Returns `true` if and only if the values of all operands are `true`.

— `<term>: BooleanTerm [2..*]`

This represents the list of `BooleanTerm` operands of the conjunction. Terms shall be evaluated in the order of appearance in the OTX document.

IMPORTANT — Since OTX terms do not have side-effects, the evaluation of `LogicAnd` operand terms shall complete when the first `false` operand has been identified; `LogicAnd` returns `false`

then without evaluating the remaining operand terms. Note that one consequence of this is that exceptions which might have occurred in the remaining operands are bypassed.

IMPORTANT — In a graphical representation, LogicAnd shall be represented by the operator symbols ∧ or `&&` (or just `and`) in infix notation. This is the most common representation form used in mathematical notation and by many programming languages.

7.15.7.3.2 LogicOr

`LogicOr` is a `BooleanTerm` which represents the Boolean disjunction. Returns `true` if and only if at least one of the operands is `true`.

— `<term>: BooleanTerm [2..*]`

  This represents the list of `BooleanTerm` operands of the disjunction. Terms shall be evaluated in the order of appearance in the OTX document.

IMPORTANT — Since OTX terms do not have side-effects, the evaluation of LogicOr operand terms should complete when the first true operand has been identified; `LogicOr` returns true then without evaluating the remaining operand terms. This may improve execution speed.

IMPORTANT — Since OTX terms do not have side-effects, the evaluation of `LogicOr` operand terms shall complete when the first `true` operand has been identified; `LogicOr` returns `true` then without evaluating the remaining operand terms. Note that one consequence of this is that exceptions which might have occurred in the remaining operands are bypassed.

NOTE In graphical representation, `LogicOr` can be represented by the operator symbols ∨ or `||` (or just `or`) in infix notation.

7.15.7.3.3 LogicXor

`LogicXor` is a `BooleanTerm` which represents the Boolean exclusive disjunction. Returns `true` if one of the operands is `true`, but not all.

— `<term>: BooleanTerm [2..*]`

  This represents the `BooleanTerm` operands of the exclusive disjunction. Terms shall be evaluated in the order of appearance in the OTX document.

NOTE In graphical representation, `LogicXor` can be represented by the operator symbol ⊕(or just `xor`) in infix notation.

7.15.7.3.4 LogicNot

`LogicNot` is a `BooleanTerm` which represents the Boolean negation. Returns `true` if and only if the operand is `false`.

— `<term>: BooleanTerm [1]`

  This represents the `BooleanTerm` operand of the negation.

NOTE In graphical representation, `LogicNot` can be represented by the operator symbol ¬ or `!` (or just `not`).

7.15.7.4 Example

The example below shows a branch condition with represents the logical implication `a → b`, which is expressed by its logical equivalent `not(a or b)`.

Sample of logic operation terms

```
<branch id="branch2">
  <realisation>
    <if>
      <condition id="branch2condition">
        <specification>Does imlication "a->b" alias "not(a||b)" hold?</specification>
        <realisation xsi:type="LogicNot">
          <term xsi:type="LogicOr">
            <term xsi:type="BooleanValue" valueOf="a" />
            <term xsi:type="BooleanValue" valueOf="b" />
          </term>
        </realisation>
      </condition>
      <flow>
        <action id="a11" />
      </flow>
    </if>
  </realisation>
</branch>
```

7.15.8 Relational operations

7.15.8.1 Description

The relational operation terms provide the most basic and widely used comparators, namely equality (`IsEqual`), inequality (`IsNotEqual`), less-than (`IsLess`), greater-then (`IsGreater`), less-then-or-equal (`IsLessOrEqual`) and greater-then-or-equal (`IsGreaterOrEqual`).

While equality and inequality can be applied for all kinds of terms, the other comparators shall only be used for simple terms. Before such a simple term comparison can be performed, preparations shall be met depending on the data type of the operands.

— `Float` or `Integer` case: ordinary comparison is based on the numerical values of the operands. No preparatory steps to be taken.

— `String` case: relations are based on the lexicographical order (also known as alphabetical order) of the `String` operands. Ordering shall happen according to the Unicode character set[10].

— `Boolean` case: the convention `true` is greater than `false` applies. All other relations are based on this.

IMPORTANT — When an `Integer` value is compared to a `Float` value, automatic promotion of the `Integer` shall be performed. This means that the `Integer` is automatically cast to `Float` and then the values are compared.

7.15.8.2 Syntax

Figure 59 shows the syntax of the relational operation term types.

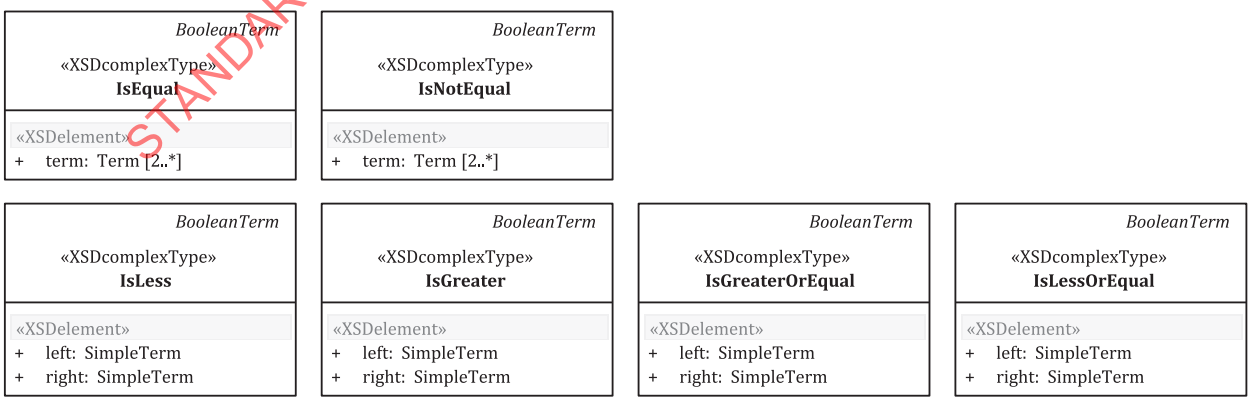

Figure 59 — Data model view: Relational operations

7.15.8.3 Semantics

7.15.8.3.1 IsEqual

IsEqual is a **BooleanTerm** which tests for equality of two or more operands. For **SimpleTerm** arguments, **true** is returned if and only if the values of all terms are equal. For terms returning a reference value (e.g. a reference to a **List** or **Map**), **true** is returned if and only if the references returned by all terms are equal.

— **<term>: Term [2..*]**

This represents the operands which shall be tested for equality. Terms shall be evaluated in the order of appearance in the OTX document.

IMPORTANT — Since OTX terms do not have side-effects, the evaluation of IsEqual operand terms shall complete when the first unequal to others operand has been identified; IsEqual returns false then without evaluating the remaining operand terms. Note that one consequence of this is that exceptions which might have occurred in the remaining operands are bypassed.

Associated checker rules:

— Core_Chk049 – uniform relation operand type (see C.2.49);

— Core_Chk061 – usage of util:Compare instead of IsEqual and IsNotEqual (see C.2.61).

NOTE In graphical representation, **IsEqual** can be represented by the operator symbol = or = = (or just **equals**) in infix notation.

7.15.8.3.2 IsNotEqual

IsNotEqual is a **BooleanTerm** which tests for inequality of two or more operands. For **SimpleTerm** arguments, **true** is returned if and only if not all of the values of the terms are equal. For terms returning a reference, **true** is returned if and only if not all of the references returned by the terms are equal.

— **<term>: Term [2..*]**

This represents the operands which shall be tested for equality. Terms shall be evaluated in the order of appearance in the OTX document.

IMPORTANT — Since OTX terms do not have side-effects, the evaluation of IsNotEqual operand terms shall complete when the first unequal to others operand has been identified; IsNotEqual returns true then without evaluating the remaining operand terms. Note that one consequence of this is that exceptions which might have occurred in the remaining operands are bypassed.

Associated checker rules:

— Core_Chk049 – uniform relation operand type (see C.2.49);

— Core_Chk061 – usage of util:Compare instead of IsEqual and IsNotEqual (see C.2.61).

NOTE In graphical representation, **IsNotEqual** can be represented by the operator symbol ! = or < > or ≠ in infix notation.

7.15.8.3.3 IsLess

IsLess is a **BooleanTerm**. Returns **true** if and only if the **<left>** value is less than the **<right>** value.

— **<left>: SimpleTerm [1]**

This represents the value that shall be less than the **<right>** value.

— **<right>: SimpleTerm [1]**

This represents the value that shall be greater than the **<left>** value.

Associated checker rules:

— Core_Chk049 – uniform relation operand type (see C.2.49).

NOTE In graphical representation, **IsLess** can be represented by the operator symbol **<** in infix notation.

7.15.8.3.4 IsGreater

IsGreater is a **BooleanTerm**. Returns **true** if and only if the **<left>** value is greater than the **<right>** value.

— **<left>: SimpleTerm [1]**

This represents the value that shall be greater than the **<right>** value.

— **<right>: SimpleTerm [1]**

This represents the value that shall be less than the **<left>** value.

Associated checker rules:

— Core_Chk049 – uniform relation operand type (see C.2.49).

NOTE In graphical representation, **IsGreater** can be represented by the operator symbol **>** in infix notation.

7.15.8.3.5 IsGreaterOrEqual

IsGreaterOrEqual is a **BooleanTerm**. Returns **true** if and only if the **<left>** value is greater or equal than the **<right>** value.

— **<left>: SimpleTerm [1]**

This represents the value that shall be greater or equal than the **<right>** value.

— **<right>: SimpleTerm [1]**

This represents the value that shall be less or equal than the **<left>** value.

Associated checker rules:

— Core_Chk049 – uniform relation operand type (see C.2.49).

NOTE In graphical representation, **IsGreaterOrEqual** can be represented by the operator symbol **≥** or **> =** in infix notation.

7.15.8.3.6 IsLessOrEqual

IsLessOrEqual is a **BooleanTerm**. Returns **true** if and only if the **<left>** value is less or equal than the **<right>** value.

— **<left>: SimpleTerm [1]**

This represents the value that shall be less or equal than the **<right>** value.

— **<right>: SimpleTerm [1]**

This represents the value that shall be greater or equal than the **<left>** value.

Associated checker rules:

— Core_Chk049 – uniform relation operand type (see C.2.49).

NOTE In graphical representation, **IsLessOrEqual** can be represented by the operator symbol ≤ or < = in infix notation.

7.15.8.4 Example

The **Branch** node in the example below contains a condition **(f < 10)&&(f = 10)** using **IsLess** and **IsEqual** relations.

Sample of relational operation terms

```
<branch id="branch3">
  <realisation>
    <if>
      <condition id="branch1condition">
        <specification>Is f less than 10 AND f equals 10 !?</specification>
        <realisation xsi:type="LogicAnd">
          <term xsi:type="IsLess">
            <left xsi:type="FloatValue" valueOf="f" />
            <right xsi:type="IntegerLiteral" value="10" />
          </term>
          <term xsi:type="IsEqual">
            <term xsi:type="FloatValue" valueOf="f" />
            <term xsi:type="IntegerLiteral" value="10" />
          </term>
        </realisation>
      </condition>
      <flow>
        <throw id="t1">
          <realisation xsi:type="UserExceptionCreate">
            <qualifier xsi:type="StringLiteral" value="Strange Problem" />
            <text xsi:type="StringLiteral" value="Something went terribly wrong!" />
          </realisation>
        </throw>
      </flow>
    </if>
  </realisation>
</branch>
```

7.15.9 Mathematical operations

7.15.9.1 Description

The mathematical operation terms provide the most basic mathematical operators, namely addition (**Add**), subtraction (**Subtract**), multiplication (**Multiply**), division (**Divide**), abs (**AbsoluteValue**), modulo (**Modulo**), negation (**Negate**) and rounding (**Round**).

7.15.9.2 Syntax

Figure 60 shows the syntax of the mathematical operation term types.

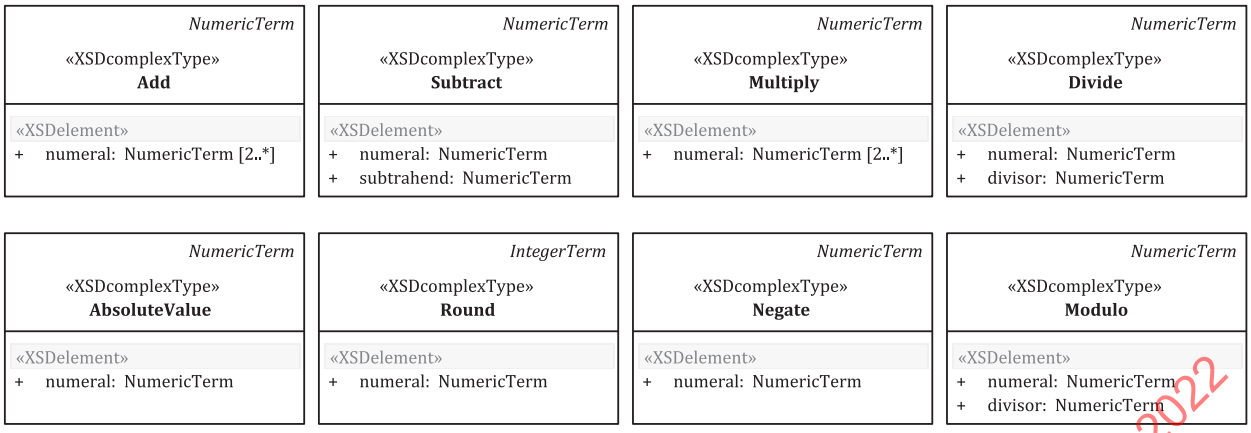

Figure 60 — Data model view: Mathematical operations

7.15.9.3 Semantics

7.15.9.3.1 General

The OTX **`Integer`** and **`Float`** data types have restricted value spaces, as specified in Annex A. Therefore, the mathematical operations described in the following may cause overflows or underflows. More specifically an overflow or underflow situation occurs when the mathematically correct result of an operation is outside of the value space of **`Integer`** or **`Float`**. OTX does not define exceptions for those situations. Overflows and underflows shall be handled in analogy to the specifications given by Reference [9], subclause 4.2.

7.15.9.3.2 Add

`Add` is a **`NumericTerm`** which returns the sum of all its operands.

— **`<numeral>: NumericTerm [2..*]`**

  This is the numeric operands (= summands) of the addition. Terms shall be evaluated in the order of appearance in the OTX document.

IMPORTANT — The actual return data type of the `Add` term depends on the data types of the operands: if one of the operands is `Float`, all `Integer` operands are automatically promoted to `Float` prior to the operation and the returned value is `Float`. Otherwise, the returned value is `Integer`.

NOTE In graphical representation, **`Add`** can be represented in infix notation by the operator symbol **`+`**.

7.15.9.3.3 Subtract

`Subtract` is a **`NumericTerm`** which returns the numerical difference between the operands.

— **`<numeral>: NumericTerm [1]`**

  This is a numeric term which represents the minuend value.

— **`<subtrahend>: NumericTerm [1]`**

  This is a numeric term which represents the subtrahend value.

IMPORTANT — The actual return data type of the `Subtract` term depends on the data types of the operands: If one of the operands is `Float`, all `Integer` operands are automatically promoted to `Float` prior to the operation and the returned value is `Float`. Otherwise, the returned value is `Integer`.

NOTE In graphical representation, **Subtract** can be represented in infix notation by the operator symbol –.

7.15.9.3.4 Multiply

Multiply is a **NumericTerm** which returns the product of the operands.

— **<numeral>: NumericTerm [2..*]**

This is the numeric term which represents the factor values. Terms shall be evaluated in the order of appearance in the OTX document.

IMPORTANT — The actual return data type of the Multiply term depends on the data types of the operands: If one of the operands is Float, all Integer operands are automatically promoted to Float prior to the operation and then the returned value is Float. Otherwise, the returned value is Integer.

NOTE In graphical representation, **Multiply** can be represented by the operator symbol ·, × or * in infix notation.

7.15.9.3.5 Divide

Divide is a **NumericTerm** which returns the quotient of two operands.

— **<numeral>: NumericTerm [1]**

This is a numeric term which represents the dividend value.

— **<divisor>: NumericTerm [1]**

This is a numeric term which represents the divisor value.

Throws:

— **ArithmeticException**

It is thrown if both dividend and divisor are **Integer** and the divisor value is zero.

IMPORTANT — The result of Divide is computed in two different ways, depending on the data types of the operands: if both dividend and divisor are Integer, then the fraction (quotient) is truncated and an Integer result is returned. If dividend or divisor are Float, then a Float fraction is returned.

NOTE In graphical representation, **Divide** can be represented in infix notation by the operator symbol /.

7.15.9.3.6 Modulo

Modulo is a **NumericTerm** which returns the remainder of a division. If the dividend value is positive, the returned remainder will be positive. If the dividend value is negative, the returned remainder will be negative.

— **<numeral>: NumericTerm [1]**

This is a numeric term which represents the dividend value.

— **<divisor>: NumericTerm [1]**

This is a numeric term which represents the divisor value.

Throws:

— **ArithmeticException**

It is thrown if both dividend and divisor are **Integer** and the divisor value is zero.

IMPORTANT — The result of `Modulo` is computed in two different ways, depending on the data types of the operands: if both dividend and divisor are `Integer`, then the returned remainder is `Integer`. If dividend or divisor are `Float`, then a `Float` remainder is returned. In the `Float` case, the Java approach shall be applied (see Reference [9], subclause 15.17.3 "Remainder Operator %") which truncates the quotient for the modulo computation. Note that this is not the IEEE 754 approach which rounds the quotient (see IEEE 754:2019). The Java approach shall also be followed if numeral and/or divisor is negative, or if one of them is INF, -INF, −0.0, or NaN.

NOTE In graphical representation, **`Modulo`** can be represented in infix notation by the operator symbol `%` (or just **`mod`**).

7.15.9.3.7 AbsoluteValue

`AbsoluteValue` is a **`NumericTerm`** which returns the value of the operand without regard to its sign.

— **`<numeral>: NumericTerm [1]`**

This is the numeric term whose absolute value shall be returned.

IMPORTANT — The actual return data type of the `AbsoluteValue` term depends on the data type of the operand: if the operand is `Float`, then the returned value is `Float`. Otherwise, the returned value is `Integer`.

NOTE In graphical representation, **`AbsoluteValue`** can be represented by enclosing the operator in `||` symbols, e.g. |a|.

7.15.9.3.8 Round

`Round` is an **`IntegerTerm`** which returns the rounded integer value of the operand.

— **`<numeral>: NumericTerm [1]`**

This is the numeric term that shall be rounded to the nearest integer towards plus infinity.

In case the range of the Integer type is exceeded through the rounding operation or NaN, -INF or INF, the behaviour is tool specific.

EXAMPLE

Round(−0.0) = = 0

Round(0.1) = = 1

Round(−1.9) = = −1

Round(1.234) = = 2

7.15.9.3.9 Negate

`Negate` is a **`NumericTerm`** which returns the value of the operand with inverted sign.

— **`<numeral>: NumericTerm [1]`**

This is the numeric term that shall be negated.

IMPORTANT — The actual return data type of the `Negate` term depends on the data type of the operand: If the operand is `Float`, then the returned value is `Float`. Otherwise, the returned value is `Integer`.

7.15.9.3.10 RoundToNearest

RoundToNearest is an **IntegerTerm** which returns the rounded integer value of the operand.

— **<numeral>: NumericTerm [1]**

This is the numeric term that shall be rounded to the nearest integer. If there are two nearest integers, the term shall be rounded to the nearest integer towards plus infinity for positive numbers and rounded to the nearest integer towards minus infinity for negative numbers.

In case the range of the Integer type is exceeded through the rounding operation or NaN, -INF or INF, the behaviour is tool specific.

EXAMPLE

Round(−0.0) = = 0

Round(0.5) = = 1

Round(−1.5) = = −2

Round(1.546) = = 2

7.15.9.4 Example

The example below shows mathematical terms in a **Branch** condition **E-m*c2 = = 0**.

Sample of mathematical operation terms

```
<branch id="branch4">
  <realisation>
    <if>
      <condition id="branch1condition">
        <specification>Is E-m*c^2==0?</specification>
        <realisation xsi:type="IsEqual">
          <term xsi:type="Subtract">
            <numeral xsi:type="FloatValue" valueOf="E" />
            <subtrahend xsi:type="Multiply">
              <numeral xsi:type="FloatValue" valueOf="m" />
              <numeral xsi:type="FloatValue" valueOf="constants:c" />
              <numeral xsi:type="FloatValue" valueOf="constants:c" />
            </subtrahend>
          </term>
          <term xsi:type="IntegerLiteral" value="0" />
        </realisation>
      </condition>
      <flow>
        <action id="a1">
          <specification>Einstein was right!!!</specification>
        </action>
      </flow>
    </if>
  </realisation>
</branch>
```

7.15.10 ByteField operations

7.15.10.1 Description

For operating on **ByteField** data, there is a set of special operations defined in the OTX core. These are the bitwise logical operations **BitwiseAnd**, **BitwiseOr**, **BitwiseXor** and **BitwiseNot** which are similar to the Boolean logic operations, and other **ByteField** related operations **ByteFieldGetSize** and **GetBit**.

7.15.10.2 Syntax

Figure 61 shows the syntax of the **ByteField** operation term types.

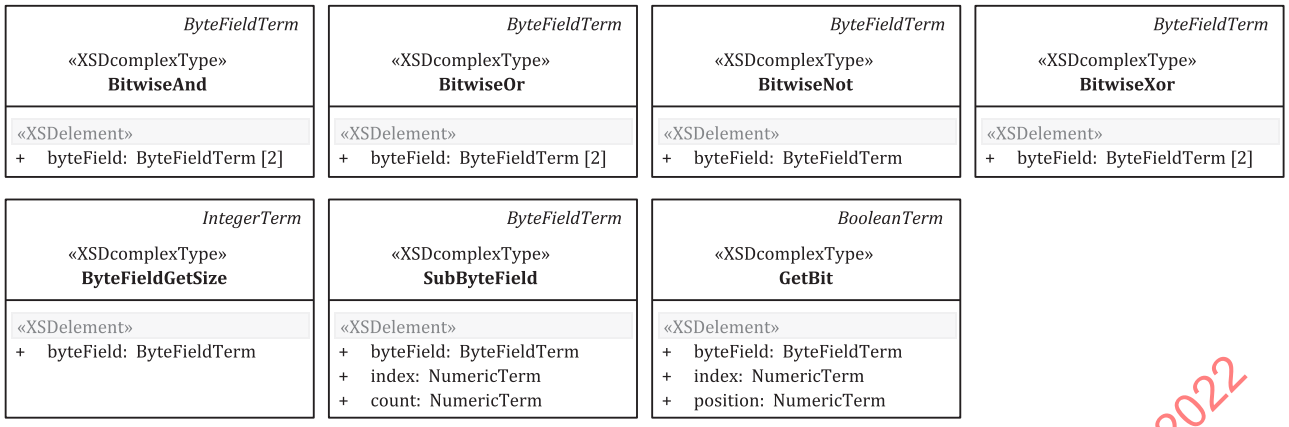

Figure 61 — Data model view: `ByteField` operations

7.15.10.3 Semantics

7.15.10.3.1 BitwiseAnd

`BitwiseAnd` is a **`ByteFieldTerm`** which operates on individual bits of **`ByteField`** values. It returns the bitwise conjunction of the two **`ByteField`** operands.

— **`<byteField>: ByteFieldTerm [2]`**

These are the two **`ByteField`** operands of the conjunction. Terms shall be evaluated in the order of appearance in the OTX document.

IMPORTANT — If an operand is shorter than the other, it is promoted to the length of the other by bit stuffing (0-bits) from the left prior to executing the bitwise operation. The length of the returned `ByteField` is the length of the longer one.

NOTE In graphical representation, **`BitwiseAnd`** can be represented in infix notation by the operator symbol **`&`**.

7.15.10.3.2 BitwiseOr

`BitwiseOr` is a **`ByteFieldTerm`** which operates on individual bits of **`ByteField`** values. It returns the bitwise disjunction of the two **`ByteField`** operands.

— **`<byteField>: ByteFieldTerm [2]`**

These are the two **`ByteField`** operands of the disjunction. Terms shall be evaluated in the order of appearance in the OTX document.

IMPORTANT — If an operand is shorter than the other, it is promoted to the length of the other by bit stuffing (0-bits) from the left prior to executing the bitwise operation. The length of the returned `ByteField` is the length of the longer one.

NOTE In graphical representation, **`BitwiseOr`** can be represented in infix notation by the operator symbol **`|`**.

7.15.10.3.3 BitwiseXor

BitwiseXor is a **ByteFieldTerm** which operates on individual bits of **ByteField** values. It returns the bitwise exclusive disjunction of the two **ByteField** operands.

— **<byteField>: ByteFieldTerm [2]**

These are the two **ByteField** operands of the exclusive disjunction. Terms shall be evaluated in the order of appearance in the OTX document.

IMPORTANT — If an operand is shorter than the other, it is promoted to the length of the other by bit stuffing (0-bits) from the left prior to executing the bitwise operation. The length of the returned ByteField is the length of the longer one.

NOTE In graphical representation, **BitwiseOr** can be represented in infix notation by the operator symbol **^**.

7.15.10.3.4 BitwiseNot

BitwiseNot is a **ByteFieldTerm** which operates on individual bits of a **ByteField** value. It returns the bitwise negation (the inverse) of the **ByteField** operand.

— **<byteField>: ByteFieldTerm [1]**

This is the **ByteField** operand that shall be inverted.

NOTE In graphical representation, **LogicNot** can be represented by the operator symbol **~**.

7.15.10.3.5 ByteFieldGetSize

ByteFieldGetSize is an **IntegerTerm** which returns the number of bytes that the **ByteField** operand contains.

— **<byteField>: ByteFieldTerm [1]**

This is the **ByteField** operand whose size shall be returned.

7.15.10.3.6 SubByteField

SubByteField is a **ByteFieldTerm** which reads a given number of bytes from a **ByteField** starting at a given byte index. The read bytes are returned as a **ByteField** value.

— **<byteField>: ByteFieldTerm [1]**

This is the **ByteField** from which the sub-bytefield shall be read.

— **<index>: NumericTerm [1]**

This is the byte index of the first byte of the sub-bytefield to be read. **Float** values shall be truncated.

— **<count>: NumericTerm [1]**

This is the total number of bytes to be read starting from **<index>**. If **<count>** equals **0**, an empty **Bytefield** shall be returned. **Float** values shall be truncated.

Throws:

— **OutOfBoundsException**

It is thrown if the index is not within the range **[0,n-1]**, where **n** is the size of the **ByteField**.

— **OutOfBoundsException**

It is thrown if index+count > size of the **ByteField** or count is negative.

7.15.10.3.7 GetBit

`GetBit` is a `BooleanTerm` which returns `true` if the bit at the given byte index and bit position is `1`, otherwise it returns `false`.

— `<byteField>: ByteFieldTerm [1]`

This is the `ByteField` in which the bit shall be looked up.

— `<index>: NumericTerm [1]`

This is the byte index of the byte in which the bit of interest resides. `Float` values shall be truncated.

— `<position>: NumericTerm [1]`

This is the bit position of the bit within the identified byte. `Float` values shall be truncated.

Throws:

— `OutOfBoundsException`

It is thrown if the index is not within the range `[0,n-1]`, where `n` is the size of the `ByteField`.

— `OutOfBoundsException`

It is thrown if the bit position not within the range `[0,7]` of allowed bit positions.

7.15.11 List-related terms

7.15.11.1 Description

`List` related terms operate on lists; OTX provides a term to identify the length of a list (`ListGetLength`) and a query for the existence of a value in a list (`ListContainsValue`).

7.15.11.2 Syntax

Figure 62 shows the syntax of the `List` related term types.

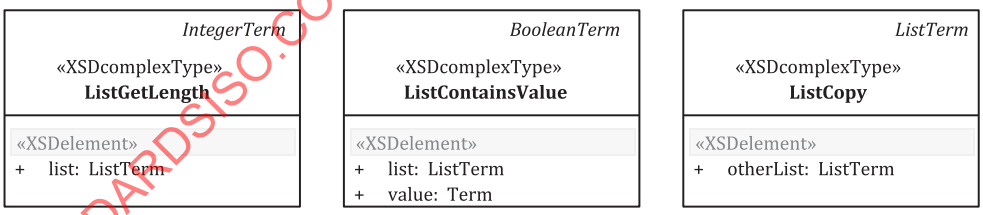

Figure 62 — Data model view: `List` related terms

7.15.11.3 Semantics

7.15.11.3.1 ListGetLength

`ListGetLength` is an `IntegerTerm` that returns the number of items contained in a `List`.

— `<list>: ListTerm [1]`

This is the `List` whose length shall be computed.

7.15.11.3.2 ListContainsValue

`ListContainsValue` is a **`BooleanTerm`** that returns **`true`** if and only if the given value is contained in the **`List`**.

— **`<list>: ListTerm [1]`**

This is the **`List`** which shall be checked for the **`<value>`** to exist.

— **`<value>: Term [1]`**

This is the value whose existence in the **`List`** shall be tested. If the **`<value>`** represents a reference, then it is tested whether the reference is part of the **`List`** or not.

Associated checker rules:

— Core_Chk059 – type-safe ListContainsValue (see C.2.59).

7.15.11.3.3 ListCopy

`ListCopy` is a **`ListTerm`** that returns a **`List`** which contains shallow copies of all items of another **`List`**.

— **`<otherList>: ListTerm [1]`**

This is the **`List`** which shall be copied.

7.15.12 Map-related terms

7.15.12.1 Description

Similar to the **`List`** related terms, OTX provides terms which operate on **`Map`** values. There is a term to identify the size of a map (**`MapGetSize`**), query terms for testing the existence of a key or a value in a map (**`MapContainsKey`** and **`MapContainsValue`**) and a term for extracting a list of keys out of a map (**`MapGetKeyList`**).

7.15.12.2 Syntax

Figure 63 shows the syntax of the **`Map`** related term types.

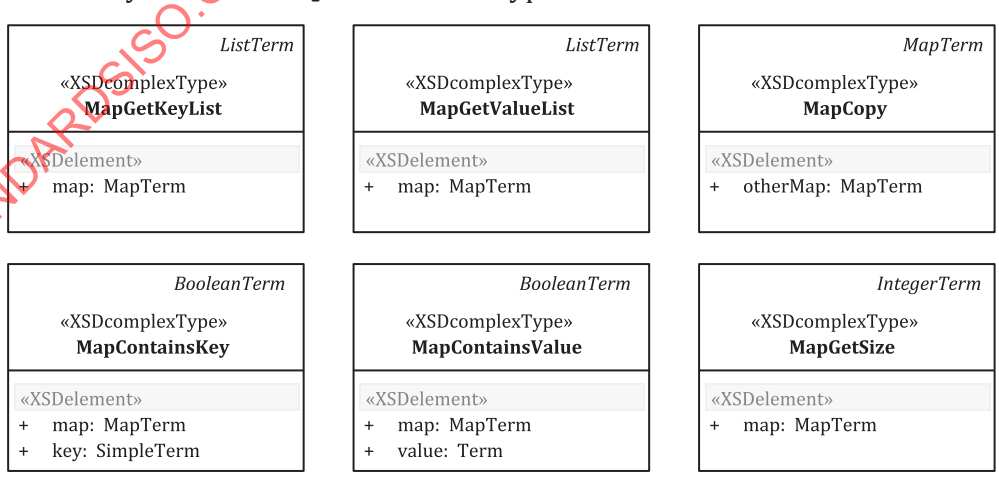

Figure 63 — Data model view: `Map` related terms

7.15.12.3 Semantics

7.15.12.3.1 MapContainsKey

MapContainsKey is a **BooleanTerm** that returns **true** if and only if the given key is contained in the **Map**.

— **<map>: MapTerm [1]**

This is the **Map** which shall be checked for the **<key>** to exist in one of the **Map** items.

— **<key>: SimpleTerm [1]**

This is the key whose existence in the **Map** shall be tested.

Associated checker rules:

— Core_Chk023 – type-safe assignments (see C.2.23);

— Core_Chk040 – type-safe MapContainsKey (see C.2.40).

7.15.12.3.2 MapContainsValue

MapContainsValue is a **BooleanTerm** that returns **true** if and only if the given value is contained in the **Map**.

— **<map>: MapTerm [1]**

This is the **Map** which shall be checked for the **<value>** to exist in one of the **Map** items.

— **<value>: Term [1]**

This is the value whose existence in the **Map** shall be tested. If the **<value>** represents a reference, then it is tested whether the reference is part of the **Map** or not.

Associated checker rules:

— Core_Chk041 – type-safe MapContainsValue (see C.2.41).

7.15.12.3.3 MapGetSize

MapGetSize is an **IntegerTerm** that returns the number of items contained in a **Map**.

— **<map>: MapTerm [1]**

This is the **Map** whose size shall be computed.

7.15.12.3.4 MapGetKeyList

MapGetKeyList is a **ListTerm** that returns a new **List** which shall contain the keys of a given **Map**. In the special case when the **Map** is empty, the returned **List** shall also be empty.

— **<map>: MapTerm [1]**

This is the **Map** whose keys shall be extracted into a **List**.

7.15.12.3.5 MapGetValueList

MapGetValueList is a **ListTerm** that returns a new **List** which shall contain shallow copies of the values of a given **Map**. In the special case when the **Map** is empty, the returned **List** shall also be empty.

— **<map>: MapTerm [1]**

This is the **Map** whose values shall be extracted into a **List**.

7.15.12.3.6 MapCopy

MapCopy is a **MapTerm** that returns a **Map** which contains shallow copies of all items of another **Map**.

— **<otherMap>: MapTerm [1]**

This is the **Map** which shall be copied.

7.15.13 Exception-related terms

7.15.13.1 Description

For treating exceptions in a catch block of a handler, the getter terms **GetExceptionOriginatorNode**, **GetExceptionQualifier**, **GetExceptionText** and **GetStackTrace** are provided.

7.15.13.2 Syntax

Figure 64 shows the syntax of the **Exception** related term types.

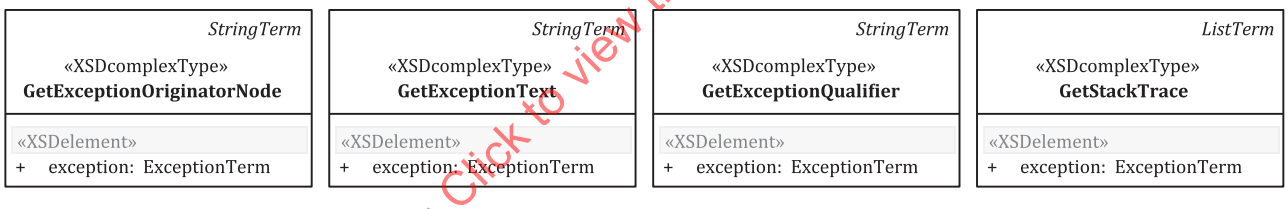

Figure 64 — Data model view: `Exception` related terms

7.15.13.3 Semantics

7.15.13.3.1 GetExceptionOriginatorNode

GetExceptionOriginatorNode is a **StringTerm** which returns the **id** of the node that caused the exception (see 7.13.2).

— **<exception>: ExceptionTerm [1]**

This represents the exception.

7.15.13.3.2 GetExceptionQualifier

GetExceptionQualifier is a **StringTerm** which returns the exception qualifier (see A.3.5).

— **<exception>: ExceptionTerm [1]**

This represents the exception.

IMPORTANT — Exception qualifiers are only defined for explicit exceptions (in OTX core, the only explicit exception is `UserException`). Therefore, `GetExceptionQualifier` shall return a string

representation of the exception's data type when it is applied on implicit exceptions. The format of the string is unspecified.

7.15.13.3.3 GetExceptionText

`GetExceptionText` is a `StringTerm` which returns the exception text (see A.3.5).

— `<exception>: ExceptionTerm [1]`

This represents the exception.

7.15.13.3.4 GetStackTrace

`GetStackTrace` is a `ListTerm` (`List` of `String`) which shall return the procedure call stack for a given exception. The call stack shall be determined when the exception is created, not when it is thrown. The zeroth element of the list shall represent the top of the stack, which is the last procedure invocation in the sequence. Typically, this is the point at which the exception was created and thrown. The returned list shall contain the fully qualified names of the invoked procedures, e.g. `{"myCompany.otx.myProcedure", "myCompany.otx.main"}`.

— `<exception>: ExceptionTerm [1]`

This represents the exception.

7.15.14 Validity concept related terms

7.15.14.1 Description

The only validity concept related term `IsValid` allows testing a validity for its truth value with respect to the current context.

7.15.14.2 Syntax

Figure 65 shows the syntax of the validity related term types.

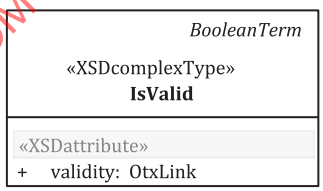

Figure 65 — Data model view: Validity concept related terms

7.15.14.3 Semantics

`IsValid` is a `BooleanTerm` which returns `true` if and only if the validity term referenced by `OtxLink` is `true` with respect to the current context (see 6.8). This term shall be used if two or more validity terms shall be pulled together by, e.g. `LogicAnd`, `LogicOr` or `LogicXor`. The term shall also be applicable to `Boolean` context variables or global `Boolean` constants (in this case, the `IsValid` term is semantically equivalent to the `BooleanValue` term).

— `validity: OtxLink [1]`

This is the validity term (or `Boolean` context variable) that shall be tested.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see C.2.53);

— Core_Chk013 – correct referencing of validities (see C.2.13).

7.15.14.4 Example

The example below shows the definition of a validity called **"DieseldrivenConvertible"** which is valid if and only if validities **"Convertible"** and **"Dieseldriven"** are both valid (the latter validities are not further specified for this example).

Sample of validity caoncept related terms

```
<?xml version="1.0" encoding="UTF-8"?>
<otx xmlns="http://iso.org/OTX/1.0.0" id="23"
  name="ValidityTermCombinationExample"
  package="org.iso.otx.examples"
  version="1.0"
  timestamp="2009-10-20T14:40:10"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <validities>
    <validity name="Convertible" id="23-v1">
      <specification>True if vehicle under test is a convertible</specification>
      <!-- Validity term goes here -->
    </validity>

    <validity name="Dieseldriven" id="23-v2">
      <specification>True if vehicle under test is a Diesel</specification>
      <!-- Validity term goes here -->
    </validity>

    <validity name="DieseldrivenConvertible" id="23-v3">
      <realisation xsi:type="LogicAnd">
        <term xsi:type="IsValid" validity="Convertible" />
        <term xsi:type="IsValid" validity="Dieseldriven" />
      </realisation>
    </validity>
  </validities>
</otx>
```

7.16 Universal types

7.16.1 Overview

In 7.16 are described auxiliary types, elements or attributes that are used by various types in the OTX data model. Since they cannot be associated to one specific feature of the model only, they are described in this comprehensive section.

7.16.2 PackageName

7.16.2.1 Description

The **PackageName** type prescribes the syntax for package naming. It is used by the **package** attribute of the **<otx>** root element (see 7.3) and the **<import>** element (see 7.4).

7.16.2.2 Syntax

Figure 66 shows the syntax of the **PackageName** type.

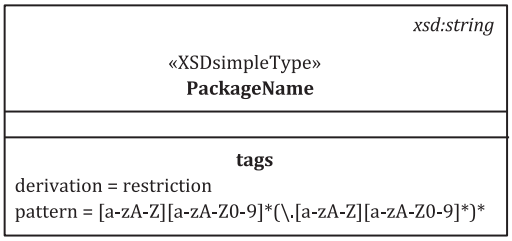

Figure 66 — Data model view: PackageName

7.16.2.3 Semantics

The simple type **PackageName** is derived from **xsd:string**. The type restricts the value space of **xsd:string** by the regular expression **"[a-zA-Z][a-zA-Z0-9]*(\.[a-zA-Z][a-zA-Z0-9]*)*"**.

As a consequence, a **PackageName** is segmented into one or more alphanumeric chunks which are separated from each other by a dot (**"."**). Each chunk shall start with a letter; all following characters shall be alphanumeric. This guarantees and enforces a uniform naming scheme for package names to which OTX authors shall obey.

Valid uses of this naming scheme allow identifiers like, for example, **"com.MyCompany.OtxSequences5"** or **"My5OtxSequences"**, whereas invalid uses are, e.g. **"com._MyCompany.OTX"**, **"mesSéquencesTest"**, **"9thPackage"** or just the empty string.

7.16.3 OtxName and OtxLink

7.16.3.1 Description

The two interrelated simple types **OtxName** and **OtxLink** are described here. In various places of the OTX data model, **OtxName** is used for naming entities like signatures, procedures or validities as well as global and local declarations. Its counterpart **OtxLink** is used for referring to these named entities (document-internally or cross-documents). Both **OtxName** and **OtxLink** represent restrictions on the syntax of such names and references, as described below.

7.16.3.2 Syntax

Figure 65 shows the syntax of the **OtxName** and **OtxLink** types.

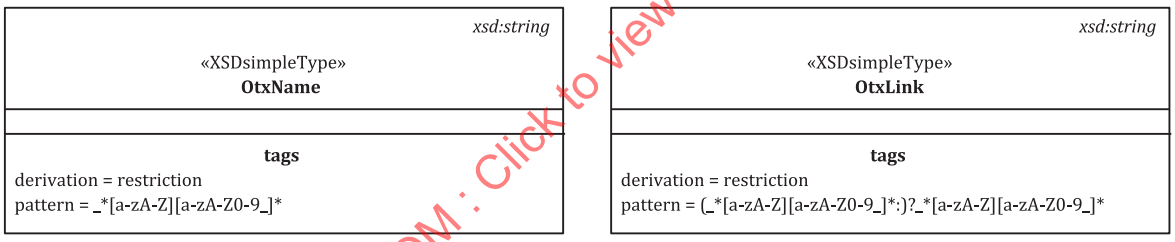

Figure 67 — Data model view: OtxName and OtxLink

7.16.3.3 Semantics

7.16.3.3.1 OtxName

The simple type **OtxName** is derived from **xsd:string**. The type restricts the value space of **xsd:string** by a regular expression pattern **"_*[a-zA-Z][a-zA-Z0-9_]*"**.

As a consequence, an **OtxName** shall start with a letter (optionally preceded by any number of underscore characters); all following characters shall be alphanumeric. This guarantees and enforces a uniform naming scheme to which OTX authors shall obey.

Valid uses of this naming scheme allow identifiers like, for example, **"MyProcedure"**, **"__validity10"**, **"PI"** or **"_y2"**, whereas invalid uses are, e.g. **"9x"**, **"___"** or just the empty string.

7.16.3.3.2 OtxLink

The simple type **OtxLink** is derived from **xsd:string**. The type restricts the value space of **xsd:string** by a regular expression pattern "**(_*[a-zA-Z][a-zA-Z0-9_]*:)?_*[a-zA-Z][a-zA-Z0-9_]***".

According to the pattern, the `OtxLink` syntax allows two ways of referring to a named entity, depending on whether the target entity is defined in the same document, or whether it is defined in an external document (which was imported to the local document as specified in 7.4). In the first case the syntax equals the `OtxName` syntax. In the latter case the syntax equals "`OtxName:OtxName`": here the first part shall be the name of the prefix assigned to the particular OTX document where the target entity is defined, the second part (after the colon) is the entity name itself.

Valid uses of this naming scheme allow document-internal references like, for example, `"MyProcedure"` or `"x"`, but also cross-document references to entities defined in external documents like, e.g. `"sig:mySignature5"` or `"constants:PI"`.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see C.2.53);

— Core_Chk005 – no use of undefined import prefixes (see C.2.5).

7.16.4 NamedAndSpecified

7.16.4.1 Description

The abstract type `NamedAndSpecified` is used by every type that shall be identifiable, nameable, carry a specification text and/or supply metadata. Types using this are, e.g. all `Node` types, variable, constant and parameter declarations, the `Procedure` and `Signature` types or the `Otx` root type itself.

7.16.4.2 Syntax

Figure 68 shows the syntax of the `NamedAndSpecified` type.

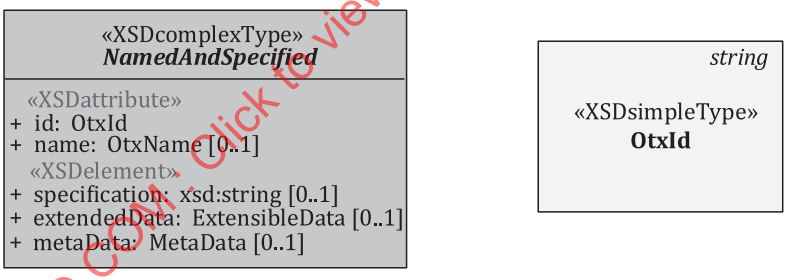

Figure 68 — Data model view: `NamedAndSpecified`

7.16.4.3 Semantics

The abstract type `NamedAndSpecified` has the following semantic properties:

— `id: OtxId [1]`

The attribute value represents the unique identifier of a `NamedAndSpecified` element. It shall be unique among all other ids in the same OTX document. This constraint is verifiable through XML schema validation (by `<xsd:key>` constraints specified in the OTX schema). The `OtxId` type is a pattern-restriction of the `xsd:string` simple type. The value space of the attribute is restricted by the regular expression "`[a-zA-Z0-9\\\-\.\|:_#/]+`", which allows the basic letters, numbers as well as a set of delimiters.

The `id` attribute is useful concerning versioning, differencing or concurrent editing of OTX documents. It facilitates traceability of changes that occurred in between different versions of an OTX document because an element can still be identified even after having been moved or changed thoroughly.

Concerning `id` value allocation, OTX authoring applications shall provide the following.

- When a new **NamedAndSpecified** element is created, the application shall generate and assign a unique id-value to the element's **id** attribute.
- After creation, applications shall not alter id values anymore. An id shall stay with its element as an inherent part as long as the element exists, it shall not be changed even if the element is moved or the contents of the element are changed.
- For each copy of an element, a new unique id value shall be generated and assigned to the copy.
- The allocation and handling of id values should happen automatically in the background. No user action should be required (manual creation of unique ids is an awkward task, beyond that this is not acceptable concerning useability). Id values should not be shown to the user and, most importantly, they shall not be editable by the user.

  NOTE Users can certainly bypass these rules by editing OTX-documents directly (e.g. with a simple text editor). Therefore, these rules only make sense as long as the user works within an OTX authoring application. This provided, optimal process stability concerning change traceability is ensured.

Associated checker rules:

- Core_Chk058 – unique OtxIds (see C.2.58).

Recommendation:

The standard requires ids to be unique only within the scope of their local OTX document. Even better process stability concerning change traceability can be gained when guaranteeing id uniqueness on larger scales, e.g. uniqueness among a full set of OTX documents, projects, company wide uniqueness or even universal uniqueness. To achieve this, it is beneficial to use Universally Unique Identifiers (UUID) for the **id** attribute, as specified by Reference [6].

- **name: OtxName [0..1]**

  This represents the OTX name of a **NamedAndSpecified** element.

  IMPORTANT — Since some NamedAndSpecified types require the name attribute while others do not, there are special rules defined for those types who require it. The rules are enforced by <xsd:key> restrictions in the OTX Schema.

- **<specification>: xsd:string [0..1]**

  This allows adding a descriptive specification to the **NamedAndSpecified** element.

- **<extendedData>: ExtensibleData [0..*]**

  This declares general data for **NamedAndSpecified** which can be extented by new general data defined in new OTX extensions using the standardised extension mechanism. For example, it can be used to specify specification relevant content in a better structured way.

  IMPORTANT — The extension mechanism described here only defines the syntactic rules to which extension implementers shall obey in order to conform to this document. It shall be ensured that no OTX requirements are violated.

- **<metaData>: MetaData [0..1]**

  If metadata are added to a **NamedAndSpecified** element, the **<metaData>** shall be used (see 7.16.5).

7.16.5 MetaData

7.16.5.1 Description

The complex type **MetaData** is used by several types in the OTX data model, e.g. by the **Otx**, **Procedure** and **Node** types, which contain a **<metaData>** element. The **MetaData** type allows tools to store additional, mainly tool-specific data.

IMPORTANT — By definition, any metadata in an OTX document shall be ignorable or even removable at any time without changing the original logic of the procedures contained.

7.16.5.2 Syntax

Figure 69 shows the syntax of the **MetaData** type.

There are no syntactical constraints on the metadata itself (however, there are semantic constraints). Syntactically, each **<data>** element in a **<metaData>** element can consist of:

— a simple string (since **Data** is a **mixed** type),

— an embedded XML document valid to an arbitrary XML schema (see **AnyContent** which is an **<xsd:any>** wildcard element in the OTX schema) or

— XLink attributes which allow pointing to metadata sources which may be contained in an external document (see W3C XLink:2010).

The validation setting **processContents = "lax"** is set for the **<xsd:any>** wildcard, which is used for arbitrary XML style metadata. This setting means that XML content will not be taken into account by validators unless the corresponding XML schema for the XML content is given (e.g. by the **xml:schemaLocation** attribute).

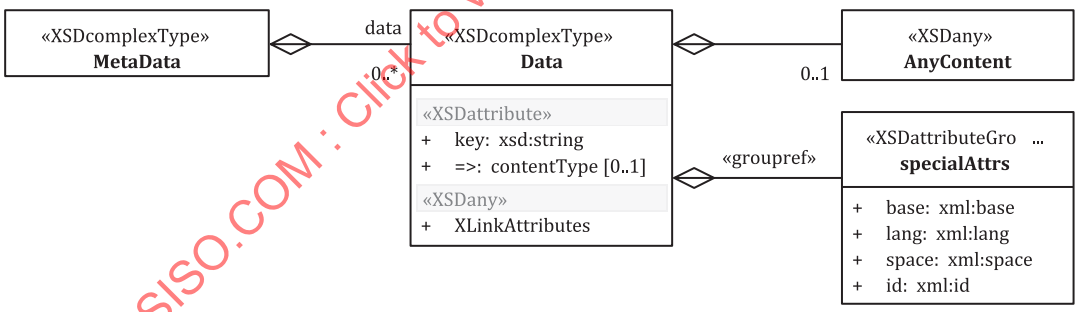

Figure 69 — Data model view: MetaData

7.16.5.3 Semantics

Since the contained metadata are potentially tool-specific and non-standardized, it is not guaranteed to be understood by every OTX conform application. This is why the data shall not be used by any OTX application in a way that the behaviour of the procedure logic is changed.

A **<metaData>** element is merely a container for an arbitrary-length list of **<data>** elements. The following describes the meaning of the corresponding **Data** type:

— **key: xsd:Name [1]**

This attribute contains a key used by applications to identify a specific metadata item. All **key** names shall be unique within the scope of the surrounding **<metadata>** element. This is ensured by XML schema validation (by **<xsd:key>** constraints specified in the OTX schema).

Recommendations:

There should be no overlapping of keys between different suppliers. For this reason, the company name or tool name should be used as prefix.

— **`xmime:contentType: xsd:string [0..1]`**

This optional attribute is used to describe the content type of the metadata. This information can be used by applications, e.g. to start a viewer which can handle this kind of content. The content type string shall be formatted according to the rules given in RFC 2045 and RFC 2046.

— **`xml:specialAttrs (xml:base, xml:lang, xml:space, xml:id [all optional])`**

This attribute group is referenced by the **`Data`** complex type; it is externally defined by the "xml": namespace http://www.w3.org/XML/1998/namespace, standardized by the W3C. For OTX metadata, these attributes can optionally be used to specify some further properties of the data, e.g. defining the language of the contents, how spaces should be treated. For detailed information about the use of the attributes, refer to the following W3C recommendations:

— W3C XMLNS:2009: General information about the **`xml:namespace`**;

— W3C XML:2008: Attributes **`xml:lang`** and **`xml:space`**;

— W3C XMLBASE:2009: Attribute **`xml:base`**.

— **XLinkAttributes**

The optional XLink attributes as specified by W3C XLink:2010 allow pointing to a URI of an external resource which contains the actual metadata. It is also possible to point to particular places inside of the external resource.

Recommendation:

If the XLink attributes are set, any other content between the **`<data>`** and **`</data>`** markup may be used as an alternative when the external resource cannot be accessed; otherwise it may be ignored.

— **Mixed content (text or any XML data)**

The content of **`<data>`** elements has no semantic defined by the OTX standard. Semantics are application specific and need to be defined separately. If non-standardized namespaces are used within the XML data, the corresponding XML schemas shall be transported alongside the XML document; otherwise, the OTX document is not valid.

7.16.5.4 Example

The example below shows different uses of **`<data>`** elements in the **`<metaData>`** section of the **`<otx>`** element.

Sample of MetaData

```
<?xml version="1.0" encoding="UTF-8"?>
<otx xmlns="http://iso.org/OTX/1.0.0" id="24"
  name="MetaDataExample"
  package="org.iso.otx.examples"
  version="1.0"
  timestamp="2009-10-20T14:40:10"
  xmlns:xmime="http://www.w3.org/2005/05/xmlmime"
  xmlns:xlink=" http://www.w3.org/1999/xlink"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <metaData>
    <data key="k1" xlink:href="../images/image1.jpeg" xmime:contentType="image/jpeg" />
    <data key="k2" xmime:contentType="application/xhtml+xml">
      <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
        <head><title>XHTML document</title></head>
        <body>
          <p>This is an example of meta-data, structured after XML rules</p>
        </body>
      </html>
    </data>
    <data key="k3" xml:lang="en-GB" xml:space="preserve">Some simple    meta information</data>
    <data key="k4"><![CDATA[More meta <data>]]></data>
  </metaData>
</otx>
```

The first use (`key = "k1"`) is an xlink to an external document, there is a hint for the content type of this document (`xmime:contentType = "image/jpeg"`). The second (`key = "k2"`) is an example for metadata which has XML form, in this case it is a XHTML document. Since there is no schema location given for the XHTML content in this example, validation will succeed even if the data were no valid (but well formed) XHTML. The third use (`key = "k3"`) shows metadata that is represented by static string content. The last use shows metadata contained in an unparsed CDATA section.

7.16.6 Variable access

7.16.6.1 Description

The variable access types represent the counterpart of the dereferencing terms (see 7.15.3) which are used to read the actual value stored in a declaration. By contrast, variable access types are used by OTX nodes and actions when the variable container itself is of primary interest, not the value stored in it. This is the case, for example, in an `Assignment` action: the calculated value given by the `<term>` will be assigned to the variable identified by `<result>` (see 7.14.4). Another example is the `Loop` (for-loop configuration) node with its `<counter>` element, which identifies the `Integer` variable which shall contain the current iteration number at runtime (see 7.13.4.3). In all of these cases, the current value of the variable is not relevant.

Since all `Variable` subtypes have uniform syntax and semantics, a general description applying to all types is provided hereby.

7.16.6.2 Syntax

Figure 70 shows the syntax of the `Variable` types.

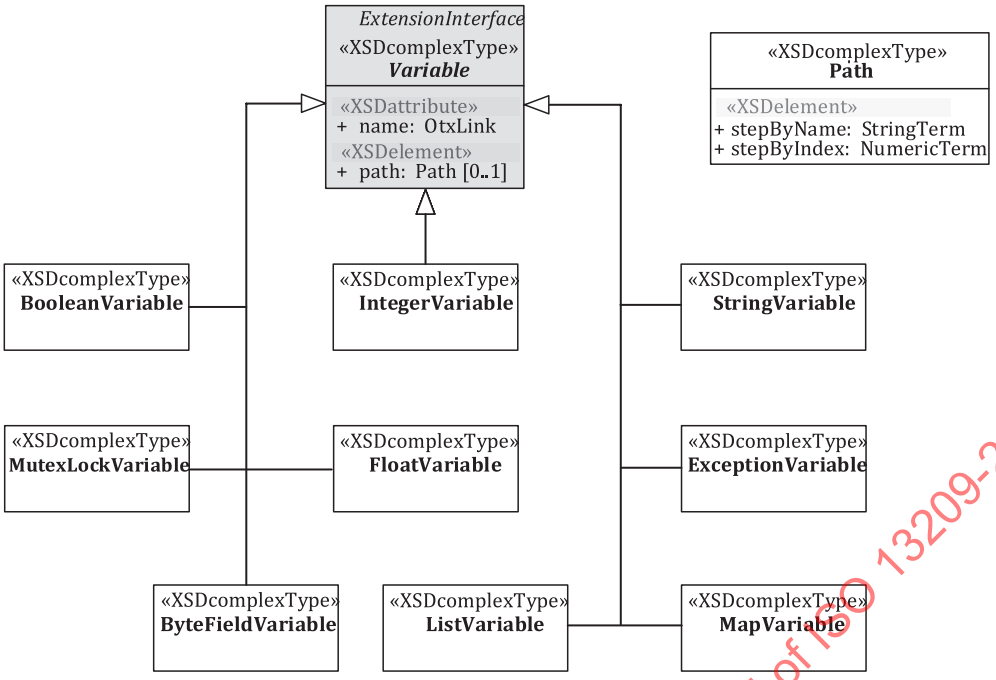

Figure 70 — Data model view: `Variable types`

7.16.6.3 Semantics

The properties of all **`Variable`** types have the following semantics:

— **`name: OtxLink [1]`**

This contains the **`OtxLink`** to the declaration which is of interest. For getting access to data which resides deeper within a complex data structure like a **`List`** or **`Map`**, the **`<path>`** element shall be utilized which points to the data inside of the structure.

— **`<path>: Path [0..1]`**

The element addresses parts of complex structures like **`List`** or **`Map`**. It is built out of a series of index- and name-steps which allow navigate into the structure (by utilizing **`<xsd:choice> [1..*]`**):

— **`<stepByIndex>: NumericTerm [1]`**

This step of a **`<path>`** shall be used for locations addressed by index. Items in a **`List`** and items in a **`Map`** with **`Integer`** keys shall be in this way. **`Float`** values shall be truncated.

— **`<stepByName>: StringTerm [1]`**

This step of a **`<path>`** shall be used for addressing locations by name. Items in a **`Map`** with **`String`** keys shall be addressed in this way.

The OTX data model does not allow any **`Variable`** subtype to refer to immutable data (constants).

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see C.2.53);

— Core_Chk050 – type-safe variable and constant usage (see C.2.50);

— Core_Chk051 – immutability of constants, input parameters and context variables (see C.2.51).

Throws:

— **OutOfBoundsException**

It is thrown only if there is a **<path>**: if the **<path>** points to a location which has not been allocated (like a list index exceeding list length, or a map key which is not part of the map).

In a graphical OTX editor, variable types can be displayed in the common notation form used by all major programming languages. For example, for a **BooleanVariable** type for a variable b, the display should simply show b; for a **BooleanVariable** with a **<path>** pointing to index i in a list L of **Booleans**, L[i] should be displayed.

7.16.6.4 Example

The example below shows an **IntegerVariable** term which is used as the in an **Assignment** action. The **name** attribute identifies the integer variable **i**. The value assigned to **i** is the value of another variable **j**, which is dereferenced by using the **IntegerValue** term (see 7.15.3).

Sample 1 of VariableAccess

```
<action id="a1">
  <specification>Assignment i:=j</specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="IntegerVariable" name="i" />
    <term xsi:type="IntegerValue" valueOf="j" />
  </realisation>
</action>
```

Consider the other example in the OTX snippet below. The **Assigment** action addresses the target of the result by using the **name** attribute with a **<path>**. The target resides in the complex structure **L** (a **List** of **Map**), more specifically at **L[1,"key1"]**. The **Assignment** also uses the **valueOf** attribute in the **IntegerValue** term for reading the constant named **c**.

Sample 2 of VariableAccess

```
<declarations>
  <variable name="L">
    <specification>A list of maps of {string:integer} items</specification>
  </variable>
  <constant name="c" />
</declarations>

<flow>
  <action id="a1">
    <specification>Assign the value of c to L[1,'key1']</specification>
    <realisation xsi:type="Assignment">
      <result xsi:type="IntegerVariable" name="L">
        <path>
          <stepByIndex xsi:type="IntegerLiteral" value="1" />
          <stepByName xsi:type="StringLiteral" value="key1" />
        </path>
      </result>
      <term xsi:type="IntegerValue" valueOf="c" />
    </realisation>
  </action>
</flow>
```

7.16.7 Declarations

7.16.7.1 Overview

Test sequences need a facility for storing data that represents the state of the program during execution. In OTX, data are stored in global constants, document variables, context variables, procedure parameters, local constants and local variables. These are declared in the global declaration block, the parameter declaration blocks or local declaration blocks, as shown in the overview given by Figure 71.

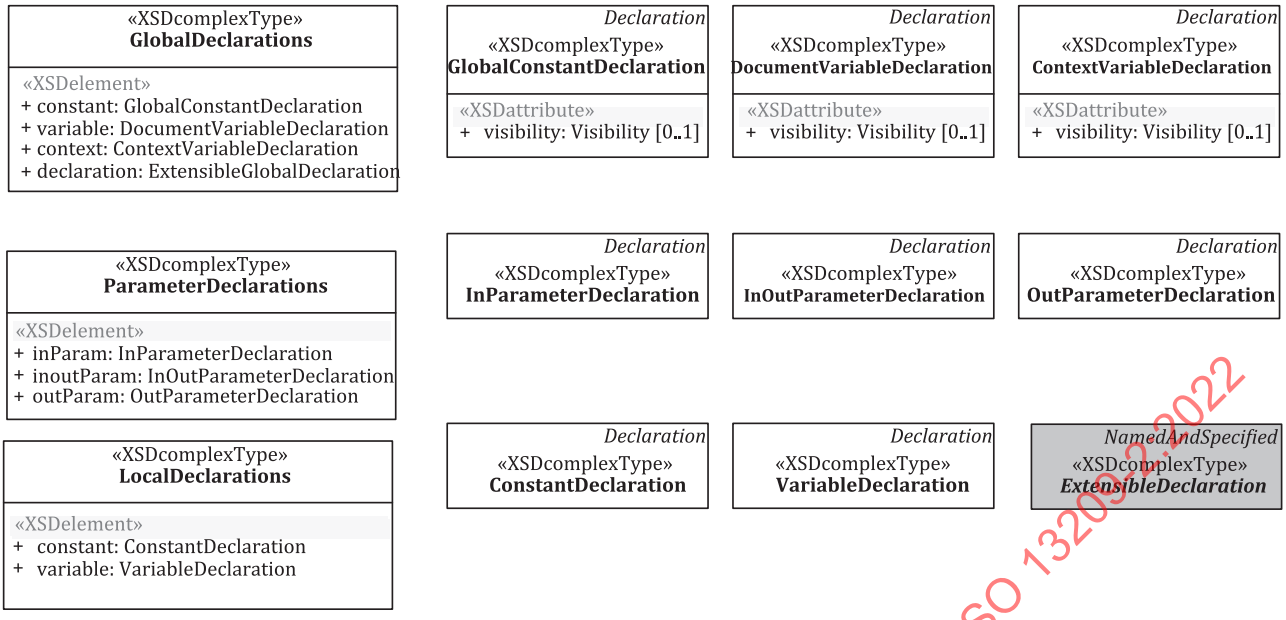

Figure 71 — Data model view: Declarations

NOTE The XSD complex types **`GlobalDeclarations`**, **`ParametersDeclarations`** and **`LocalDeclarations`** are of **`<xsd:choice> [1..*]`** content-type, which is not explicitly shown in Figure 71.

Since it shall be possible for nodes in the procedure flow to refer to the data, every declaration carries a globally/locally unique identifier. To every identifier, an OTX data type may be assigned. The declaration and the data type assignment are specified in the following.

7.16.7.2 Declaration

7.16.7.2.1 Description

The **`Declaration`** type is used for declarations in the global, local and parameter declaration blocks. The visibility of declared identifiers depends on the location of the declaration and of the visibility modifier (see 7.16.8).

7.16.7.2.2 Syntax

Generally spoken, every declaration is composed of a name (the identifier) and a data type. Compare to the **`Declaration`** type in Figure 72, which is the base for all parameter, constant and variable declaration types.

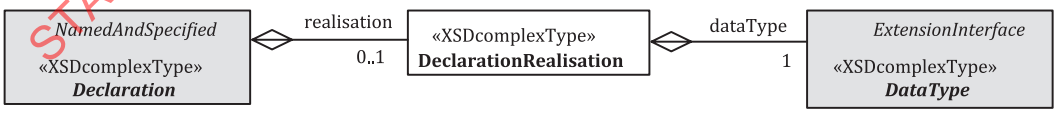

Figure 72 — Data model view: `Declaration`

7.16.7.2.3 Semantics

Declarations shall be processed before procedure execution. Memory needs to be reserved for the data, according to the data type. The properties of the abstract **`Declaration`** type have the following semantics:

— **`id: OtxId [1]`** (derived from **`NamedAndSpecified`**, see 7.16.4)

This represents a declaration's id. It shall be unique among all other ids in a document. Please refer to 7.16.4 for details concerning ids in OTX documents.

— **`name: OtxName [1]`** (derived from **`NamedAndSpecified`**, see 7.16.4)

The value of the attribute represents the human readable identifier for the data. For global declarations, the name shall be unique among all global identifiers of the same document. For local declarations, it shall be unique among all declaration identifiers in the procedure. These constraints are verifiable through XML schema validation (by **`<xsd:key>`** constraints specified in the OTX schema).

— **`<specification> : xsd:string [0..1]`** (derived from **`NamedAndSpecified`**, see 7.16.4)

The static string content of this element shall be used to specify the purpose of the declaration for the human reader. The complement of **`<specification>`** is the element **`<realisation>`** (see below).

Associated checker rules:

— Core_Chk007 – have specification if no realisation exists (see C.2.7).

— **`<realisation>: DeclarationRealisation [0..1]`**

For declarations at specification stage, having a **`name`** for the declaration is sufficient. For realisation stage, the declaration needs a data type assignment. The optional **`<realisation>`** element is used for exactly this task – it contains a **`<dataType>`** sub element:

— **`<dataType>: DataType [1]`**

This element represents the data type for the declared data. It can be chosen out of the list of available OTX data types; these are derived from the abstract **`DataType`** complex type. See 7.16.7.4 about data type assignment.

The runtime characteristics of all data types required by the OTX core are fully specified in Annex A.

7.16.7.3 ExtensibleDeclaration

7.16.7.3.1 Description

The Extensible**`Declaration`** type is used for declarations in the global extensible declaration block which can be extented by additional realisations defined in new OTX extensions using the standardised extension mechanism. The visibility of declared identifiers depends on the location of the declaration and of the visibility modifier (see 7.16.8).

7.16.7.3.2 Syntax

Generally spoken, every extensible declaration is composed of a name (the identifier) and a data type. Compare to the **`ExtensibleDeclaration`** type in Figure 73.

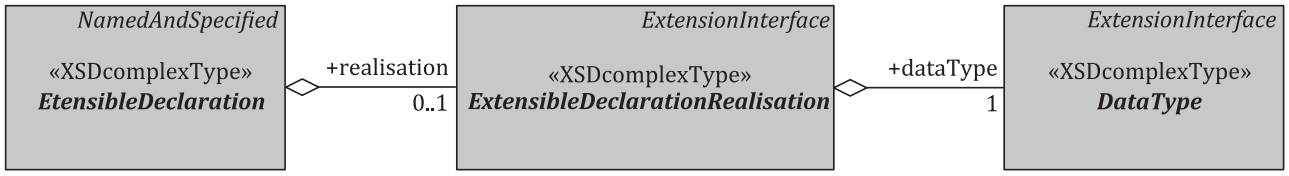

Figure 73 — Data model view: `ExtensibleDeclaration`

7.16.7.3.3 Semantics

Extensible declarations shall be processed before procedure execution. Memory needs to be reserved for the data, according to the data type. The properties of the abstract **`ExtensibleDeclaration`** type have the following semantics:

— **`id: OtxId [1]`** (derived from **`NamedAndSpecified`**, see 7.16.4)

This represents a declaration's id. It shall be unique among all other ids in a document. Please refer to 7.16.4 for details concerning ids in OTX documents.

— **`name: OtxName [1]`** (derived from **`NamedAndSpecified`**, see 7.16.4)

The value of the attribute represents the human readable identifier for the data. The name shall be unique among all global identifiers of the same document. These constraints are verifiable through XML schema validation (by **`<xsd:key>`** constraints specified in the OTX schema).

— **`<specification> : xsd:string [0..1]`** (derived from **`NamedAndSpecified`**, see 7.16.4)

The static string content of this element shall be used to specify the purpose of the declaration for the human reader. The complement of **`<specification>`** is the element **`<realisation>`** (see below).

Associated checker rules:

— Core_Chk007 – have specification if no realisation exists (see C.2.7).

— **`<realisation>: ExtensibleDeclarationRealisation [0..1]`**

For extensible declarations at specification stage, having a **`name`** for the declaration is sufficient. For realisation stage, the declaration needs a data type assignment. The optional **`<realisation>`** element is used for exactly this task – it contains a **`<dataType>`** sub element:

— **`<dataType>: DataType [1]`**

This element represents the data type for the declared data. It can be chosen out of the list of available OTX data types; these are derived from the abstract **`DataType`** complex type. See 7.16.7.4 about data type assignment.

The runtime characteristics of all data types required by the OTX core are fully specified in Annex A.

7.16.7.4 Data type assignment

For each OTX data type, there are different rules concerning the declaration and especially the initialization of identifier values.

An overview about the possible data types that can be assigned to an identifier at declaration time is given in Figure 74. This subclause defines the syntax of data type assignment. A definition of the OTX data types concerning internal data type structure and value space of each data type is provided by Annex A.