INTERNATIONAL STANDARD

ISO 13209-3

First edition
2012-08-15

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

Part 3: Standard extensions and requirements

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

Partie 3: Exigences et spécifications des extensions du standard

Reference number
ISO 13209-3:2012(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web 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.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

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.

ISO 13209-3 was prepared by Technical Committee ISO/TC 22, *Road vehicles*, Subcommittee SC 3, *Electrical and electronic equipment*.

ISO 13209 consists of the following parts, under the general title *Road vehicles — Open Test sequence eXchange format (OTX)*:

— *Part 1: General information and use cases*

— *Part 2: Core data model specification and requirements*

— *Part 3: Standard extensions and requirements*

Introduction

Diagnostic test sequences are utilized whenever automotive components or functions with diagnostic abilities are being diagnosed, tested, reprogrammed or initialised 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 environments 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.

ISO 13209 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.

ISO 13209-2 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.

This part of ISO 13209 extends the Core by a set of additional features, using the extension mechanism rules described in ISO 13209-2. The extensions defined herein comprise features which allow diagnostic communication to a vehicle's diagnostic interface, flashing, executing diagnostic jobs, controlling measurement equipment, internationalisation, working with physical units, accessing the environment, communication via a human machine interface (HMI) and other utility extensions.

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

Part 3:
Standard extensions and requirements

1 Scope

This part of ISO 13209 defines the Open Test sequence eXchange (OTX) extension requirements and data model specifications.

The requirements are derived from the use cases described in ISO 13209-1. They are listed in Clause 4.

The data model specification aims at an exhaustive definition of all features of the OTX extensions which have been implemented to satisfy the requirements. This part of ISO 13209 establishes rules for the syntactical entities of each extension. Each of these syntactical entities is accompanied by semantic rules which determine how OTX documents containing extension features are to be 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.

2 Normative references

The following referenced documents are indispensable for the application 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/IEC 646:1991, *Information technology — ISO 7-bit coded character set for information interchange*

ISO 8601:2004, *Data elements and interchange formats — Information interchange — Representation of dates and times*

ISO/IEC 8859-1:1998, *Information technology — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1*

ISO/IEC 10646, *Information technology — Universal Multiple-Octet Coded Character Set (UCS)*

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

ISO/IEC 13209-2, *Road vehicles — Open Test sequence eXchange format (OTX) — Part 2: Core data model specification and requirements*

ISO/IEC 19501:2005, *Information technology — Open Distributed Processing — Unified Modeling Language (UML) Version 1.4.2*

ISO 14229 (all parts), *Road vehicles — Unified diagnostic services (UDS)*

ISO 22900 (all parts), *Road vehicles — Modular vehicle communication interface (MVCI)*

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

RFC 1866, *Hypertext Markup Language - 2.0*

SAE J1979, E/E *Diagnostic Test Modes*

W3C XPtr:2003*, W3C Recommendation: XPointer Framework (all parts)*

W3C XLink:2001, *W3C Recommendation: XML Linking Language (XLink) Version 1.0*

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

W3C XMLNS:2009, W3C *Recommendation: Namespaces in XML 1.0 (Third Edition)*

W3C XSD:2004, W3C *Recommendation: XML Schema (all parts)*

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, ISO 13209-2 and the following apply.

3.1.1
custom screen
screen with attributes and fields defined by a test sequence author

3.1.2
dialog
screen with predefined attributes and fields which can be set or read from an OTX sequence

3.1.3
ECOS measurement device
widely-used embedded system for testing electrical consumer's current and voltage curves

3.1.4
modal dialog
dialog which is blocking the flow execution until the user dismisses it

3.1.5
non-modal screen
asynchronous, non-blocking screen which is still displayed while the test sequence execution continues

3.1.6
tester
computer system attached to a vehicle via a Vehicle Communication Interface, running a diagnostic application

3.1.7
text ID
string reference to a thesaurus data base entry containing localized string translations

3.2 Abbreviated terms

API Application Programming Interface

DTC Diagnostic Trouble Code

ECOS	Electric Check-Out System
ECU	Electronic Control Unit
GUI	Graphical User Interface
HMI	Human Machine Interface
IFD	Interface Definition (OTX extension)
NOP	No Operation Performed
OEM	Original Equipment Manufacturer
OTX	Open Test sequence eXchange
PDU	Protocol Data Unit
UI	User Interface
UML	Unified Modeling Language
VCI	Vehicle Communication Interface
XML	Extensible Markup Language
XSD	XML Schema Definition

4 Requirements

4.1 Basic principles for requirements definition

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

a) OTX requirements specify the conditions that the OTX data model and format shall satisfy.

b) All stakeholders (System Suppliers, OEMs, Tool Suppliers), which offer diagnostic test procedures are expected to implement and follow the requirements of this standard.

The content of OTX documents and the quality of the information is the responsibility of the originator.

4.2 Requirement priorities

Each of the following requirements 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 requirement 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.

4.3 Requirement listing

Extensions_R01 – Read current date and time

Priority: SHALL

Rationale: It shall be possible to retrieve the current date and time.

Description: The current date and time shall be accessible in a way appropriate for calculating durations between two dates but also for generating a human readable form of a date.

Extensions_R02 – Support but not require ODX

Priority: SHALL

Rationale: For communication with vehicle ECUs, the usage of ODX shall be supported but not forced.

Description: Any vehicle communication related extension data model shall match to a useful subset of the functionality of ODX.

Extensions_R03 – Handle flash sessions

Priority: SHALL

Rationale: Functionality shall be provided to browse and select flash sessions.

Description: A extension for flashing shall provide the possibility to select by direction and name.

Extensions_R04 – Low level flash data access

Priority: SHALL

Rationale: Functionality shall be provided for browsing and selecting data from the flash environment (download container).

Description: The data shall be clustered in blocks and segments. Security functions, used by modern data formats like ODX Flash shall be supported.

Extensions _R05 – Flash data storage

Priority: SHALL

Rationale: Uploaded flash data shall be stored in local storage.

Description: For flash data upload, an OTX extension for flashing shall provide functionality to store in a selected format.

Extensions _R06 – Enable developer to use OTX in place of ODX Java Jobs

Priority: SHALL

Rationale: Functionality shall be provided to emulate ODX Java Jobs by OTX sequences.

Description: A job extension shall enable developers to run OTX sequences as ODX Java Jobs. SingelEcuJob, SecurityAccessJob and FlashJob shall be supported.

Extensions _R07 – Provide means for diagnostic communication with vehicle ECUs

Priority: SHALL

Rationale: Functionality shall be provided for diagnostic communication with a vehicle's ECU systems.

Description: There shall be an OTX extension which allows configuring and executing diagnostic services of vehicle ECUs. It shall be possible establish a communication channel to a particular ECU, to configure request parameters of a diagnostic service which is sent to the ECU and to analyze the response parameters of the ECU. The description of communication channels, diagnostic services and parameters shall happen in a human-readable and symbolic way; any existing diagnostic symbolic-to-binary mapping (e.g. ODX) shall be supported. The actual functionality for sending a diagnostic service and receiving shall be provided through an interface between test sequence and vehicle (e.g. MCD 3D API and MVCI).

Extensions _R08 – Provide means to browse diagnostic data

Priority: SHALL

Rationale: Functionality shall be provided to read information from the static diagnostic data base of a diagnostic application.

Description: An OTX extension shall be provided which allows reading static information from a diagnostic data base, like e.g. available communication channels, diagnostic services for a communication channel or parameters for a diagnostic service.

Extensions _R09 – Enable developer to handle events

Priority: SHALL

Rationale: Functionality shall be provided which allows for an OTX test sequence to react on a well-defined set of events.

Description: An OTX extension shall enable developers to configure a test sequence so that it can wait for certain events to happen (e.g. when a timer expires, a variable value changes or user input is received from the UI, etc.). There shall be a way to get further information about an event, e.g. what kind of event it is, and additional information about a particular event.

Extensions _R10 – Provide means for human machine interface functionality

Priority: SHALL

Rationale: Functionality shall be provided which allows OTX test sequences to communicate with a user in a bidirectional way.

Description: An OTX extension is required which allows sending and receiving information to and from a user interface (e.g. a GUI window with input controls). The extension shall not provide means for explicitly configuring the graphical layout of the information; instead it shall only provide a bidirectional interface for the communicated data itself.

Extensions _R11 – Enable developer to configure localized test sequences

Priority: SHALL

Rationale: A test sequence developer shall be supported in configuring OTX test sequences which are prepared for translation to different languages.

Description: An OTX extension is required which allows the developer to access a thesaurus data base via a text ID concept. The developer shall be supported by functionality which translates text IDs into the language

configured for the runtime system or to other languages (as far as known by the runtime system). The thesaurus data base itself shall not be part of the standard. A generic approach shall support different kinds of thesaurus data bases.

Extensions _R12 – Provide means for logging

Priority: SHALL

Rationale: It shall be possible to write log messages to a logging resource.

Description: An OTX extension is required which allows writing log messages to a logging resource; messages shall be filterable according to severity.

Extensions _R13 – Support measurement equipment

Priority: SHALL

Rationale: Measurement equipment in manufacturing and after sales workshops shall be accessible via appropriate functionality.

Description: An OTX extension is required which allows receiving measurement values from measurement equipment. There shall be an abstraction layer which allows using any kind of measurement equipment.

Extensions _R14 – Support physical units

Priority: SHALL

Rationale: Functionality is required which allows the handling of physical values with units

Description: An OTX extension is required which allows describing physical quantities. The extension shall facilitate common calculations done on such physical quantities, like e.g. the transformation of a physical value from one unit-system to another (e.g. representing a distance by kilometres or miles, etc.). It shall also allow basic mathematical operations on quantities without requiring the developer to explicitly care for the unit (e.g. it shall be possible to calculate 10 m + 2 km directly).

Extensions _R15 – Support for enhanced string operations

Priority: SHALL

Rationale: The OTX Core string operations shall be extended by additional commonly used string operations.

Description: An OTX extension is required which describes additional string operations which shall facilitate calculations on string values.

Extensions _R16 – Support of basic mathematical functions

Priority: SHALL

Rationale: The arithmetic operations of the OTX Core shall be extended by additional mathematical functions.

Description: An OTX extension is required which describes a set of additional mathematical functions which are needed in some diagnostic applications (like e.g. trigonometric and logarithmic functions, etc.).

5 Extension overview

5.1 General

This document represents the specification of the OTX standard extensions for data model version `"1.0.0"`.

5.2 Dependencies

Figure 1 shows a UML package diagram [ISO/IEC 19501] describing the full set of OTX extensions (together with the OTX Core) and the import dependencies in between them. OTX extensions use or extend types defined in the OTX Core. Therefore, all of the extensions are (directly or indirectly) based on the OTX Core data model, as specified by Part 2 of ISO 13209. Aside of the OTX Core, the OTX EventHandling, OTX DiagCom and OTX Quantities extensions also play a central role; types defined there are used or extended by other OTX extensions.

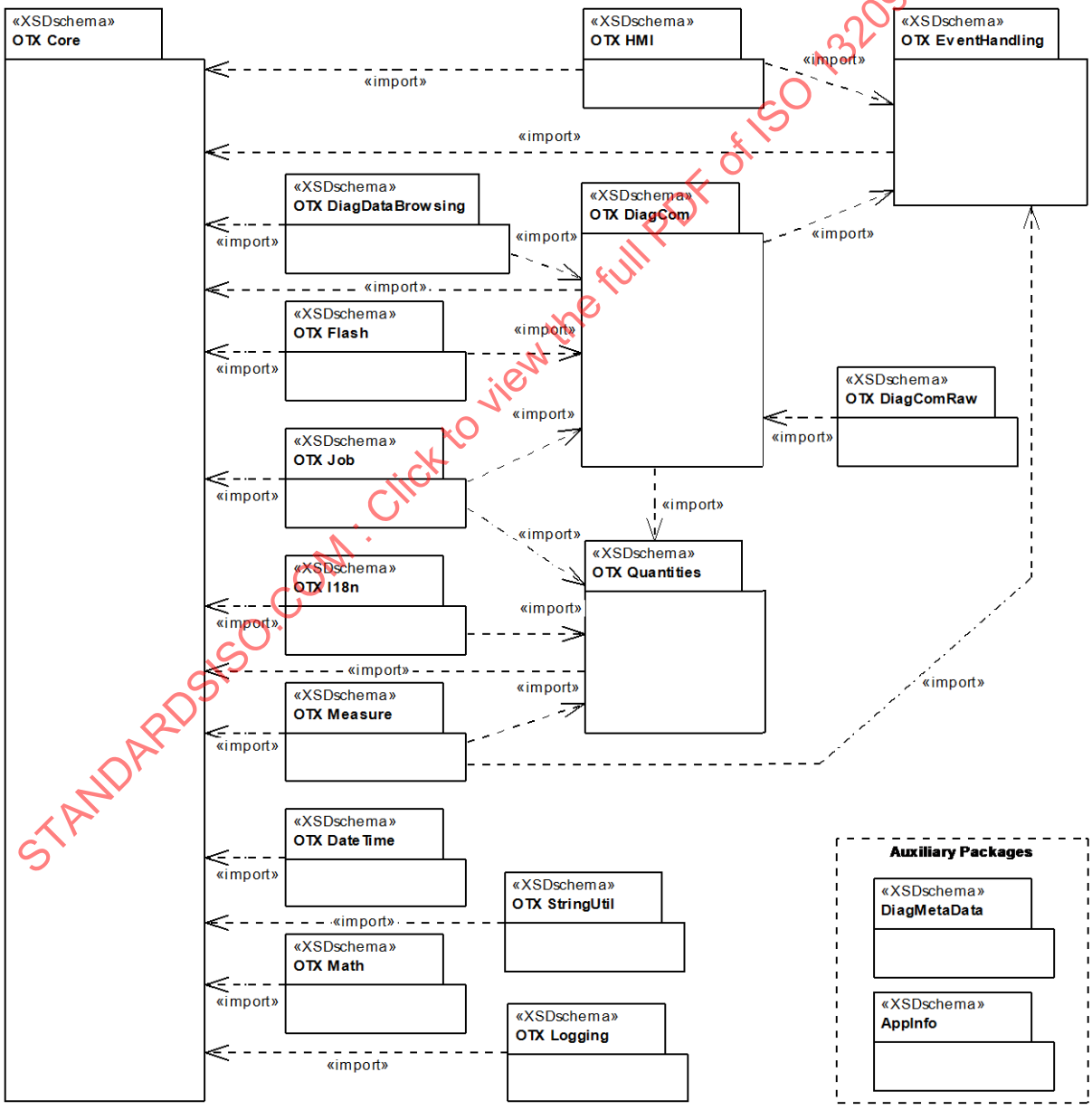

Figure 1 — Overview: OTX schema dependencies

IMPORTANT — The OTX Core is a prerequisite for any OTX application and shall always be fully supported. By contrast, OTX applications are NOT required to support all of the extensions specified in this standard. The set of supported extensions may vary depending on the field of application. However, an OTX application supporting an extension which imports other extensions shall support these, too. This guarantees that the set of supported extensions is consistent with regard to the dependencies.

Figure 1 also shows the auxiliary packages OTX DiagMetaData as well as OTX AppInfo. These packages are no OTX extensions; they support OTX authoring systems with additional data used only at authoring time. The information is not required at runtime of an OTX test sequence. For the DiagMetaData auxiliary, please see Annex C. For the AppInfo auxiliary, please refer to Part 2 of ISO 13209.

5.3 Basic characteristics of the OTX extensions

Table 1 provides an overview about all OTX extensions and their basic characteristics.

Table 1 — OTX extension characteristics

Extension (schema file)	**Summary**	**Dependencies**
DateTime (otxIFD_DateTime.xsd)	provides access to system time	Core
DiagCom (otxIFD_DiagCom.xsd)	connecting to ECUs, creating and executing diagnostic services, analysing communication data	EventHandling, Quantities, Core
DiagComRaw (otxIFD_DiagComRaw.xsd)	direct diagnostic communication on a non-symbolic (binary) level	DiagCom
DiagDataBrowsing (otxIFD_DiagDataBrosing.xsd)	browsing functionality for reading data from the diagnostic data base	DiagCom, Core
EventHandling (otxIFD_Event.xsd)	support for the OTX event handling mechanism.	Core
Flash (otxIFD_Flash.xsd)	functionality for downloading and uploading flash data to and from ECUs	DiagCom, Core
HMI (otxIFD_HMI.xsd)	functionality for communicating with the UI (user interface), through dialogs and screens	EventHandling, Core
i18n (otxIFD_I18n.xsd)	internationalisation features, multi-language support and translation mechanisms	Quantities, Core
Job (otxIFD_Job.xsd)	functionality for emulating ODX Java jobs by OTX test sequences	DiagCom, Quantities, Core
Logging (otxIFD_Logging.xsd)	support for (Log4J-style) logging	Core
Math (otxIFD_Math.xsd)	extended mathematical functions	Core
Measure (otxIFD_Measure.xsd)	executing measurement device services, measuring physical values, analyzing measurements	EventHandling Quantities, Core
Quantities (otxIFD_Quantities.xsd)	handling of quantity data, wrt. SI unit system, transformations between units, etc.	Core
StringUtil (otxIFD_StringUtil.xsd)	extended functionality for string handling	Core

Table 2 shows the XSD namespace associations of all OTX extensions [W3C XSD] [W3C XMLNS]. Each namespace has a prefix assigned to it. This applies also to the OTX Core namespace which has the **`otx:`** prefix (not shown in the table). In the remainder of this document, the prefixes defined here are used to mark types which belong to extensions other than the one which is currently described. In contrast, the types defined by the currently described extension are not prefixed.

Table 2 — OTX extension namespace associations

Extension	Namespace	Prefix
DateTime	`http://iso.org/OTX/1.0.0/DateTime`	`time:`
DiagCom	`http://iso.org/OTX/1.0.0/DiagCom`	`diag:`
DiagComRaw	`http://iso.org/OTX/1.0.0/DiagComRaw`	`raw:`
DiagDataBrowsing	`http://iso.org/OTX/1.0.0/DiagDataBrowsing`	`data:`
EventHandling	`http://iso.org/OTX/1.0.0/Event`	`event:`
Flash	`http://iso.org/OTX/1.0.0/Flash`	`flash:`
HMI	`http://iso.org/OTX/1.0.0/HMI`	`hmi:`
i18n	`http://iso.org/OTX/1.0.0/i18n`	`i18n:`
Job	`http://iso.org/OTX/1.0.0/Job`	`job:`
Logging	`http://iso.org/OTX/1.0.0/Logging`	`log:`
Math	`http://iso.org/OTX/1.0.0/Math`	`math:`
Measure	`http://iso.org/OTX/1.0.0/Measure`	`measure:`
Quantities	`http://iso.org/OTX/1.0.0/Quantities`	`quant:`
StringUtil	`http://iso.org/OTX/1.0.0/StringUtil`	`string:`

Please consider the example in Figure 2. It shows the **`IsDiagServiceEvent`** term from the OTX DiagCom extension. The term accepts a parameter which is defined in the OTX EventHandling extension, therefore the type of the element is marked with the **`event:`** prefix (**`event:EventValue`**). The same applies to the Boolean return type defined for the figure, which is defined in the OTX Core and is marked accordingly with the **`otx:`** prefix (**`otx:BooleanTerm`**). The type **`IsDiagServiceEvent`** itself is not prefixed since is a member of the currently described OTX DiagCom extension.

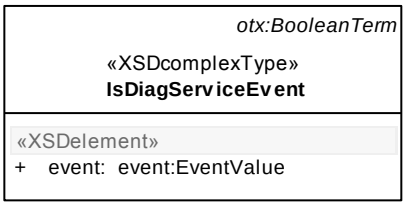

Figure 2 — Example: Usage of extension prefixes

6 OTX DateTime extension

6.1 Introduction

The purpose of the OTX DateTime extension is to retrieve information about the current date and time provided by the diagnostic application.

6.2 Terms

6.2.1 Overview

The terms in the OTX DateTime extension shall be used to retrieve information about the current system time.

6.2.2 Syntax

Figure 3 shows the syntax of all terms in the OTX DateTime extension.

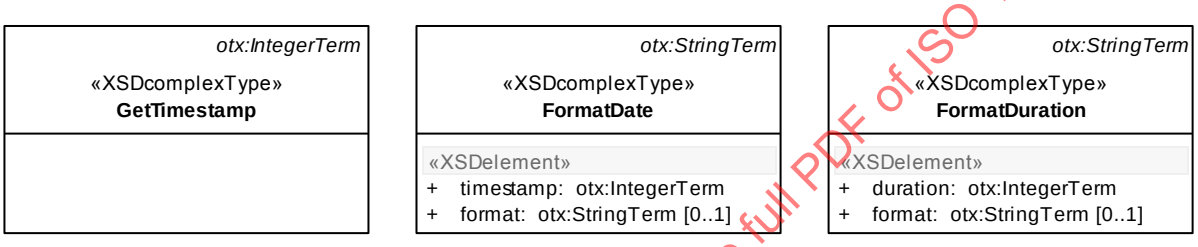

Figure 3 — Data model view: TimeDate terms

6.2.3 Semantics

6.2.3.1 GetTimeStamp

`GetTimestamp` shall return a timestamp, expressed in milliseconds elapsed since 1970-01-01 00:00:00 UTC.

The semantics of **`GetTimestamp`** shall be according to the **`java.util.Date.getTime()`** method as specified by the *Java™ 2 Platform Standard Ed. 6.*

`GetDate` is an **`otx:Integer`** term. It has no members.

6.2.3.2 FormatDate

The **`FormatDate`** term shall transform a timestamp (see **`GetTimestamp`** term above) into a date representation which shall be formatted as follows:

a) In case there is no custom format specified, the returned string shall be formatted according to the rules given by the ISO 8601 standard [ISO 8601:2004].

b) If a custom format is given (by the **`<format>`** element), the string shall be formatted according to the cusom format rules as specified below.

The custom format pattern can be configured by the OTX author; it controls the text presentation of the date. A format pattern consists of one or more predefined date and time format specifiers (see Table 3) as well as user-defined string sequences which have to be escaped by quotes (`'`). Non-numeric outputs (e.g. the name of a month) shall be translated automatically to the currently set locale (cf. Clause 12, OTX i18n extension).

Table 3 — Date format pattern specifiers

Specifier(s)	Meaning	Presentation	Example
`G`	Era	Text (localized)	`AD`
`yy, yyyy`	Year (two digits / four digits)	Number	`11, 2011`
`M, MM`	Month in year (without / with leading zero)	Number	`9, 09`
`MMM, MMMM`	Month in year (short form / long form)	Text (localized)	`Jan, January`
`d, dd`	Day in month (without / with leading zero)	Number	`3, 09`
`D`	Day in year	Number	`304`
`F`	Day of week of month	Number	`3`
`E, EEEE`	Day of week (short form / long form)	Text (localized)	`Wed, Wednesday`
`h, hh`	Hours, 1-12 count (without / with leading zero)	Number	`7, 07`
`H, HH`	Hours, 0-23 count (without / with leading zero)	Number	`7, 07`
`m, mm`	Minutes (without / with leading zero)	Number	`2, 02`
`s, ss`	Seconds (without / with leading zero)	Number	`4, 04`
`S, SS, SSS`	Milliseconds (without / with leading zeros)	Number	`357, 034, 002`
`w, W`	Week in year / Week in month	Number	`34, 3`
`a`	AM/PM designator	Text (localized)	`AM`
`z, zzzz`	Time zone (short form / long form)	Text (localized)	`CET, Central European Time`
`Z`	RFC 822 timezone (timeshift to GMT)	Text	`+0100`

The format pattern rules are analogous to the rules given for the class **`java.text.SimpleDateFormat`** as specified by the *Java™ 2 Platform Standard Ed. 6*. The overall semantics of **`FormatDate`** shall be according to the semantics of the method **`SimpleDateFormat.format(Date date)`**.

EXAMPLE 1 At a given date and time of 2011-03-10 11:23:56 in the Central European Time zone (CET) and with the current locale set to en-US, a pattern like e.g. **`"hh 'o''clock' a, zzzz"`** would produce the following formatted output: "**`11 o'clock AM, Central European Time`**."

If no custom format is given, the ISO 8601 conform date output shall be formatted equivalent to the custom pattern **`"yyyy-MM-dd'T'HH:mm:ss'.'SSSZ"`**, where **`"T"`** is the time designator and **`"."`** is a separator for the following millisecond portion. This pattern is language independent; the currently set locale does not influence the output.

EXAMPLE 2 At a given date and time of 2011-03-10 11:23:56 in the Central European Time zone (CET), the following standard-format output will be produced: **`"2011-03-10T11:23:56.123+0100"`**.

`FormatDate` is an **`otx:StringTerm`**. Its members have the following semantics:

— **`<timestamp> : otx:NumericTerm [1]`**

This element represents a date given as timestamp which shall be interpreted as amount of milliseconds elapsed since January 1, 1970 00:00:00 UTC. The corresponding date shall be formatted to a string output according to the rules given above. **`Float`** values shall be truncated.

— **`<pattern> : otx:StringTerm [0..1]`**

This optional element represents the custom format pattern which shall be applied in order to produce a custom date output string.

Throws:

— **otx:OutOfBoundsException**
If the timestamp value is negative.

6.2.3.3 FormatDuration

The **FormatDuration** term shall return a given millisecond duration in a string representation. Formatting shall be done in analogy to the **FormatDate** term, with the difference that the milliseconds passed to the term are to be interpreted as duration, not as date.

Since some of the format specifiers given in Table 3 are meaningless with respect to durations (e.g. time zone, week day name, era, etc.), only the specifiers defined in Table 4 should be used.

Table 4 — Duration format pattern specifiers

Specifier(s)	Meaning	Example
y	Years portion of duration	**11, 124**
M, MM	Months portion of duration, 0-11 count (without / with leading zero)	**2, 02**
d, dd	Days portion of duration, 0-365 count (without / with leading zero)	**3, 09**
H, HH	Hours portion of duration, 0-23 count (without / with leading zero)	**7, 07**
m, mm	Minutes portion of duration, 0-59 count (without / with leading zero)	**2, 02**
s, ss	Seconds portion of duration, 0-59 count (without / with leading zero)	**4, 04**
S, SS, SSS	Milliseconds of duration, 0-999 count (without / with leading zeros)	**357, 034, 002**

EXAMPLE 1 For a given duration of 203443 milliseconds (this is 3 minutes, 23 seconds and 443 milliseconds), a pattern like e.g. **"'This took about' m 'minutes and' s 'seconds.'"** would produce the following formatted output: "**This took about 3 minutes and 23 seconds.**"

If no custom format is given, the ISO 8601 conform date output shall be formatted equivalent to the custom pattern **"'P'yyyy-MM-dd'T'HH:mm:ss'.'SSS"**, where **"P"** is the duration designator and **"T"**is the time designator.

EXAMPLE 2 For a given duration of 203443 milliseconds (this is 3 minutes, 23 seconds and 443 milliseconds), the following standard-format output will be produced: **"P0000-00-00T00:03:23.443"**.

FormatDuration is an **otx:StringTerm**. Its members have the following semantics:

— **<duration> : otx:NumericTerm [1]**

This element represents a duration in milliseconds which shall be transformed to a string which is formatted according to the rules given above. **Float** values shall be truncated.

— **<pattern> : otx:StringTerm [0..1]**

This optional element represents the custom format pattern which shall be applied in order to produce a custom duration output string.

Throws:

— **otx:OutOfBoundsException**
If the duration value is negative.

7 OTX DiagCom extension

7.1 Introduction

The purpose of the OTX DiagCom extension is to provide the necessary OTX elements for performing diagnostic vehicle communication. Specifically, the following diagnostic use cases have been considered:

— Handling of ECU communication channels

— Execution of a diagnostic service

— Setting of service request parameters and evaluation of service response parameters

— Dealing with positive or various negative responses of a diagnostic service

— Handling of communication channel protocol parameters

— Performing variant identification of an ECU

— Functionally addressed diagnostic services: more than one ECU will respond to a request

— Repeated/cyclic execution of diagnostic services: a single request will result in multiple responses from the same ECU

— A potential combination of functional addressing and cyclic service execution: multiple ECUs responding multiple times to one request

— Complex data structures within the requests and responses of diagnostic services: structures of parameters, lists of parameters, lists containing structures of parameters

The considerations below introduce the problem domain that is addressed by the design of the DiagCom extension. Although it is unlikely that these features will all be supported by all runtime environments that execute OTX sequences, the OTX DiagCom extension has to provide the means to deal with these concepts, as it aims to be a universally usable way for defining vehicle diagnostics.

NOTE 1 It is an explicit design goal of the DiagCom extension to be usable with any diagnostic communication kernel. As a design guideline, an ODX/MVCI [ISO 22901] [ISO 22900] based system has been considered – as ODX/MVCI is solving the vehicle communication problem domain on a highly generic level, the design concepts that have been adopted for the DiagCom extension should be usable abstractions for any system that is implementing a solution to the vehicle communication problem domain.

NOTE 2 It is an explicit design goal of the DiagCom extension to only provide a runtime interface for diagnostic vehicle communication. The browsing of diagnostic data bases (e.g. the database parts of the ASAM MCD3 API) is not a design goal of the DiagCom extension. For such use cases, a separate OTX extension specifically providing data access functionality should be created.

7.2 General considerations

7.2.1 Communication channels

The prerequisite for performing any diagnostic communication is a communication channel between the diagnostic application and the electronic control unit(s) of a vehicle. In OTX this instance is called a **`ComChannel`**, designating a logical connection between the test sequence and the intended communication target. A **`ComChannel`** is not concerned with any details about the protocols, cabling, connectors or pinning required for communication with the desired endpoint; rather, these aspects are to be handled by the underlying vehicle communication layer. A **`ComChannel`** works on a symbolic level in that it is supposed to address ECUs through their name and be aware about an ECUs diagnostic capability through the specific

variant of an ECU that is present at runtime. As such, in OTX there is the concept of ECU variant identification on a **`ComChannel`**, and the capability of creating channels to point to specific ECU variants or retrieve the currently active ECU variant name of a **`ComChannel`**.

NOTE It is an explicit design goal of the OTX DiagCom extension to be useable with any diagnostic communication kernel. However, the concepts of **`ComChannel`**s and ECU variants are based on the MVCI definitions for Logical Links and ECU Variant Identification as they represent a generic, high-level approach to a widely applicable design problem.

7.2.2 Diagnostic services

Figure 4 gives a high-level overview of a diagnostic service's request and result data structure. The service contains one request. The request comprises one or more parameters. A diagnostic service can have an arbitrary number of results. In the example, one result is shown. A service result can contain an arbitrary number of ECU responses. A response contains one or more parameters. A parameter can either be a simple data type or a complex type containing lists or structures of parameters.

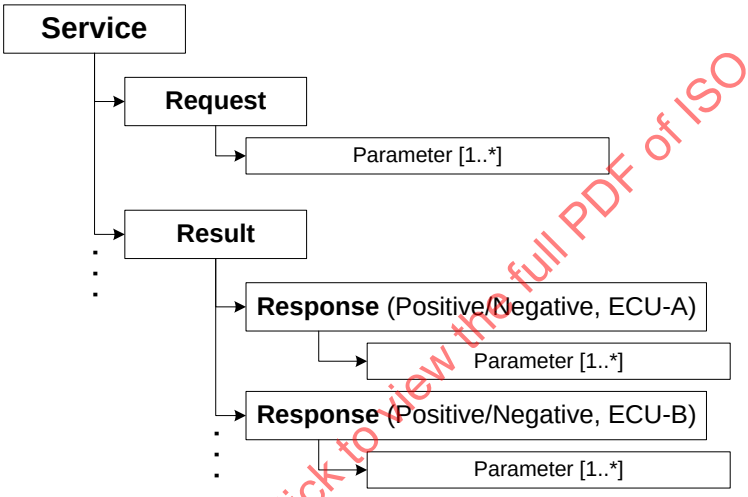

Figure 4 — Diagnostic service request - result structure

The request illustrated in Figure 5 consists of a simple parameter, a struct parameter containing two inferior parameters and a list parameter containing two items which in turn contain three simple data type parameters each. It is also shown how the different parameters can be accessed by terms and actions within the DiagCom extension, using the **`<stepByIndex>`** and **`<stepByName>`** method defined by the **`<path>`** element (please refer to the remainder of this clause, as well as Part 2 of ISO 13209 for more information). The way the parameters are accessed by path as shown in the example request also applies to ECU responses, which can comprise complex parameter structures as well.

To deal with repeated service execution patterns (please refer to 7.2.3), a diag service features the concept of a result queue. Every time a request is sent to an ECU, a new result element is added to the queue which contains the ECUs response(s) to that request. The OTX DiagCom extension provides three methods of interacting with the result queue:

a) the first result of the queue can be accessed by using the **`GetFirstResult`** term

b) all results currently in the queue can be retrieved as an OTX list by using the **`GetAllResults`** term

c) the **`GetAllResultsAndClear`** action retrieves all results in the queue and clears the queue

The lifecycle of the results in a diagnostic service's queue is delimited by service execution requests: a diagnostic service's queue is cleared each time **`ExecuteDiagService`** or **`StartRepetition`** is invoked on that **`DiagService`** object.

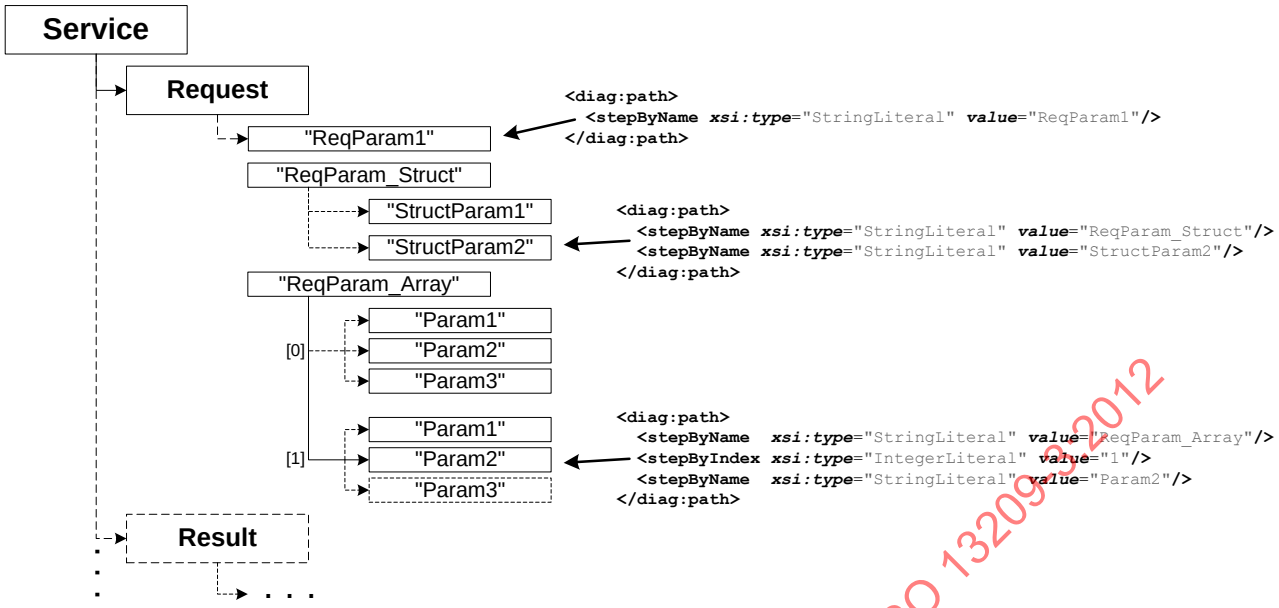

Figure 5 — Complex request structure with `<stepByName>` and `<stepByIndex>`

7.2.3 Diagnostic communication patterns

7.2.3.1 General

Besides diagnostic service requests and responses being of arbitrary structural complexity, the interaction model between a diagnostic application and the ECUs within a vehicle is also providing challenges to a diagnostic application: One diagnostic service request sent to a vehicle can result in multiple ECUs answering that request, or in multiple request and response frames in case of a diagnostic service being executed repeatedly by the communication backend. Further complications arise when an ECU's answer is sent in multiple parts. The conceivable permutations of physical and functional addressing, one-shot and repeated execution and single- and multi-part responses and the resulting result structures are illustrated below.

7.2.3.2 One-shot service, physical addressing, single-part response

Figure 6 shows a communication flow between a tester and one ECU.

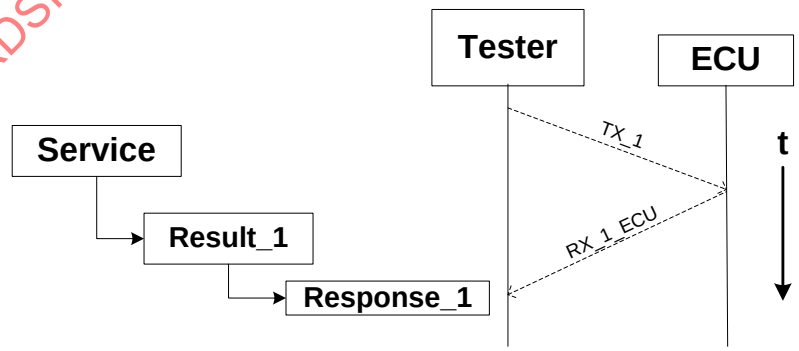

Figure 6 — One-shot service, physical addressing, single-part response

The ECU receives a physically addressed service request which results in the ECU sending a single-part response to the tester system. The request sent to the ECU leads to the creation of a corresponding result object (Result_1). The ECU's response is contained within that result object (Response_1).

7.2.3.3 One-shot service, physical addressing, multi-part responses

Figure 7 shows a communication flow between a tester and one ECU.

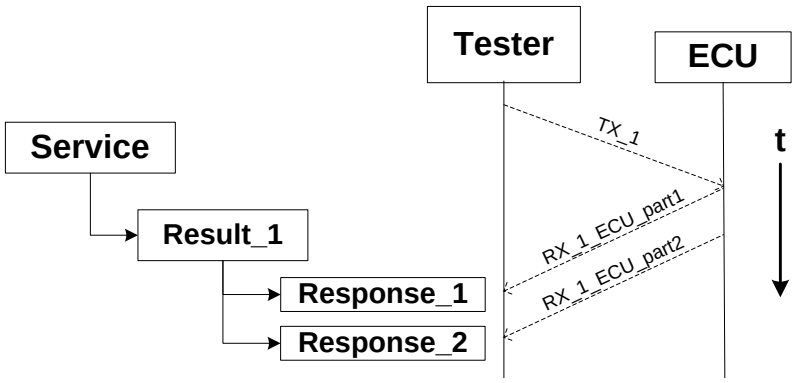

Figure 7 — One-shot service, physical addressing, multi-part responses

The ECU receives a physically addressed service request which results in the ECU sending a multi-part response to the tester system. The request sent to the ECU leads to the creation of a corresponding result object (Result_1). The ECU's responses are contained within that result object (Response_1 and Response_2).

7.2.3.4 One-shot service, functional addressing, single-part response

Figure 8 shows a communication flow between a diagnostic application and a set of ECUs.

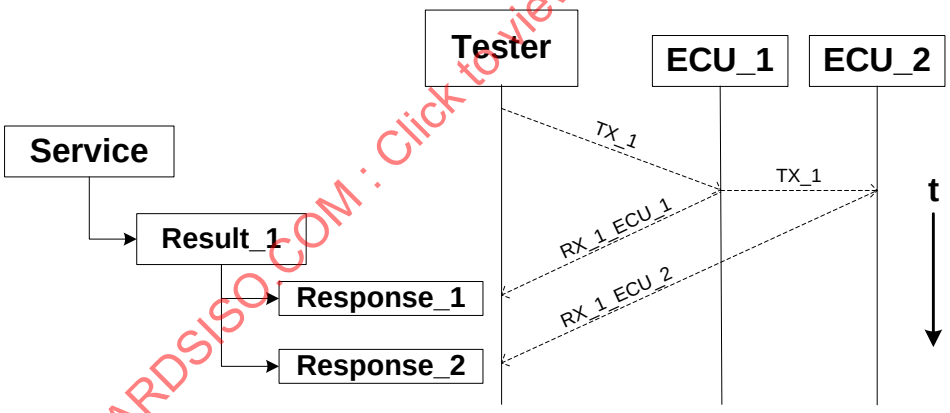

Figure 8 — One-shot service, functional addressing, single-part response

The ECUs receive a functionally addressed service request which results in the ECUs sending a single-part response to the application. The request sent to the ECUs leads to the creation of a corresponding result object (Result_1). The ECU's response is contained within that result object (Response_1 and Response_2). Please note that the OTX DiagCom term **`GetComChannelIdentifierFromResponse`** can be used to identify the ECU (**`ComChannel`**) that a response is associated with.

7.2.3.5 One-shot service, functional addressing, multi-part responses

Figure 9 shows a communication flow between a diagnostic application and a set of ECUs.

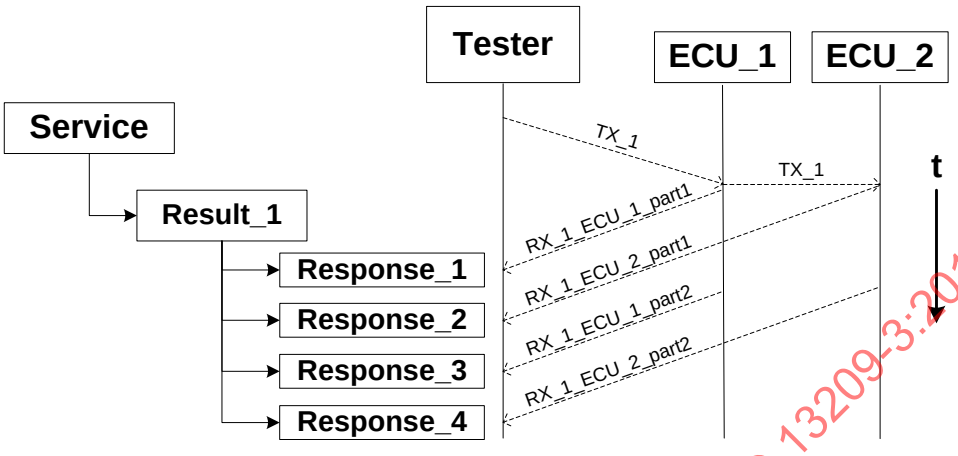

Figure 9 — One-shot service, functional addressing, multi-part responses

The ECUs receive a functionally addressed service request which results in the ECUs sending a multi-part response to the tester system. The request sent to the ECUs leads to the creation of a corresponding result object (Result_1). The ECU's responses are contained within that result object (Response_1 through Response_4).

7.2.3.6 Repeated service, physical addressing, single-part response

Figure 10 shows a communication flow between a diagnostic application and one ECU.

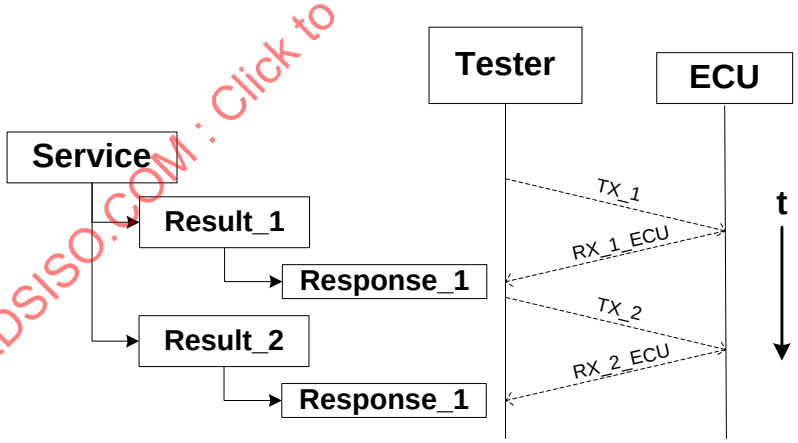

Figure 10 — Repeated service, physical addressing, single-part response

The ECU receives a repeated physically addressed service request which results in the ECU sending a single-part response to the diagnostic application for each request. The requests sent to the ECU lead to the creation of corresponding result objects (Result_1 and Result_2). The ECU's responses to the repeated requests are contained within the result object corresponding to the execution cycle that provoked the response (Result_1 for the first cycle, Result_2 for the second cycle, and so on).

7.2.3.7 Repeated service, functional addressing, single-part response

Figure 11 shows a communication flow between a diagnostic application and a set of ECUs.

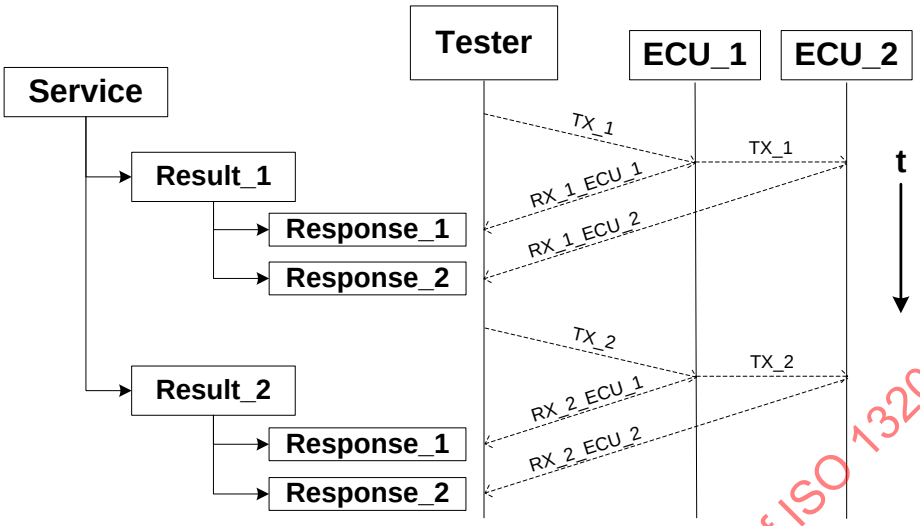

Figure 11 — Repeated service, functional addressing, single-part response

The ECUs receive a repeated functionally addressed service request which results in the ECUs sending a single-part response to the diagnostic application for each request. The requests sent to the ECUs lead to the creation of corresponding result objects (Result_1 and Result_2). The ECU's responses to the repeated requests are contained within the result object corresponding to the execution cycle that provoked the response (Result_1 for the first cycle, Result_2 for the second cycle, and so on).

7.2.3.8 Repeated service, physical addressing, multi-part responses

Figure 12 shows a communication flow between a diagnostic application and one ECU.

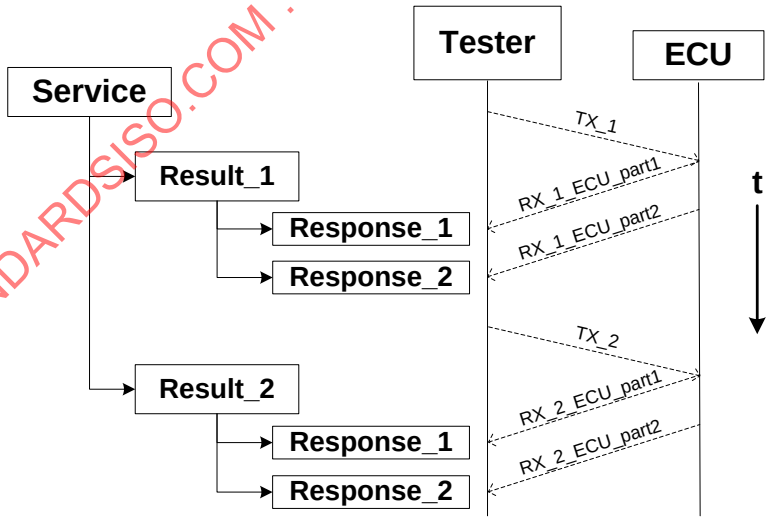

Figure 12 — Repeated service, physical addressing, multi-part responses

The ECU receives a repeated physically addressed service request which results in the ECU sending a multi-part response to the diagnostic application for each request. The requests sent to the ECU lead to the creation of corresponding result objects (Result_1 and Result_2). The ECU's responses to the repeated requests are contained within the result object corresponding to the execution cycle that provoked the response (Result_1 for the first cycle, Result_2 for the second cycle, and so on).

7.2.3.9 Repeated service, functional addressing, multi-part responses

Figure 13 shows a communication flow between a diagnostic application and a set of ECUs.

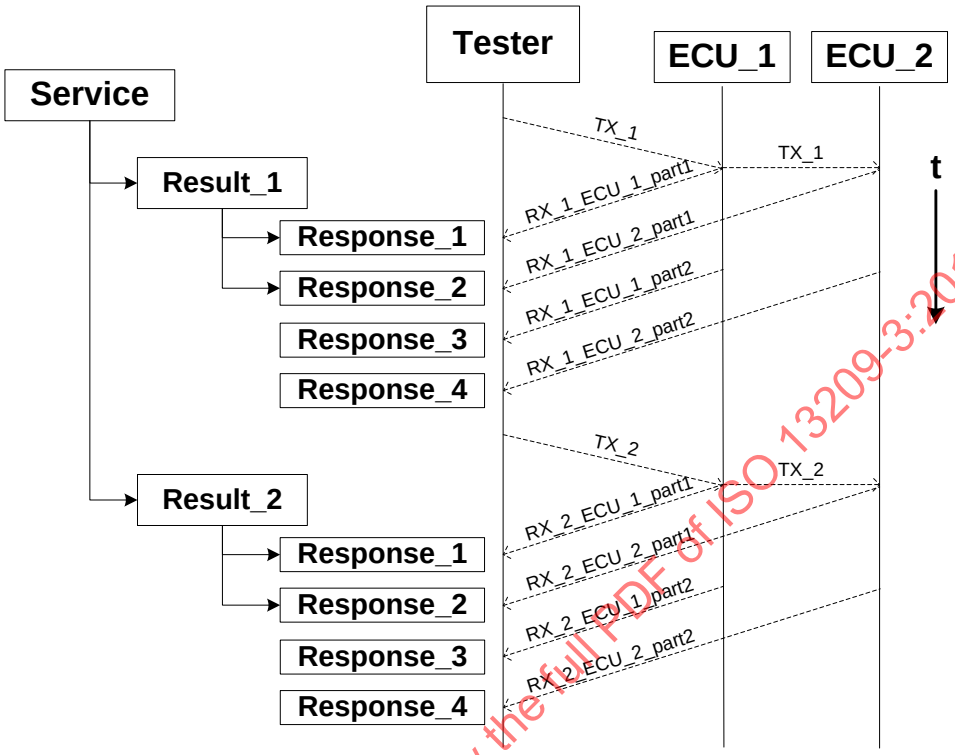

Figure 13 — Repeated service, functional addressing, multi-part responses

The ECUs receive a repeated functionally addressed service request which results in the ECUs sending a multi-part response to the diagnostic application for each request. The requests sent to the ECUs lead to the creation of corresponding result objects (Result_1 and Result_2). The ECU's responses to the repeated requests are contained within the result object corresponding to the execution cycle that provoked the response (Result_1 for the first cycle, Result_2 for the second cycle, and so on).

7.2.3.10 Other patterns

Please note that the OTX DiagCom extension does not explicitly support the feature of cyclically executing diagnostic services, i.e. services where one request to an ECU leads to the ECU cyclically sending responses to the tester system without corresponding requests. If such behaviour has to be mapped by an OTX DiagCom runtime system, the basic rule is that an OTX result object always corresponds to one request sent out by the tester system.

Figure 14 illustrates the theoretical case of a group of ECUs cyclically sending multi-part responses to a diagnostic application. In OTX, the initial request send by the application will cause a corresponding result object to be created, which subsumes any responses that were subsequently received. Please note that OTX does not support any convenience functionality for the stopping of cyclic diagnostic services. An OTX author that needs an ECU to stop it's sending of cyclic responses has to manually select and execute the appropriate diagnostic service for telling the ECU to stop.

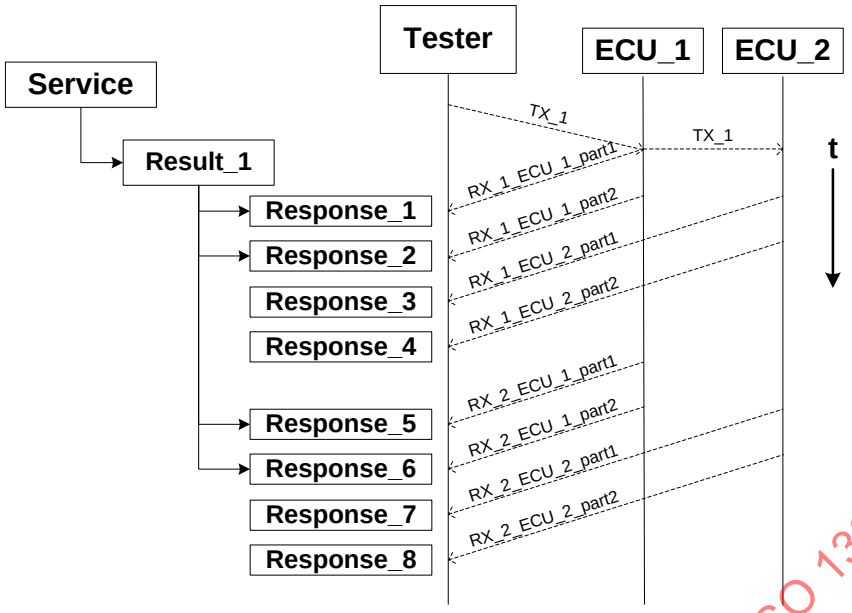

Figure 14 — One-shot service, functional addressing, cyclical multi-part responses

7.2.4 Special-purpose diagnostic data types

As OTX does not support the explicit definition of structured data types, it needs to be mentioned how the DiagCom extension treats ubiquitious diagnostic datatypes like DTCs or freeze frame data. Looking at a DTC, generally speaking it is nothing else than a structured data type with a set of structure parameters defined by the SAE J1979 standard [SAE J1979], others that are OEM specific. As such, a DTC in OTX is treated like any other structured parameter: when a parameter that represents a DTC is retrieved from a diagnostic service's response, the DTC's fields can be accesed through using the DiagCom term **`GetParameterByPath`** on the DTC parameter, passing the name of the required sub-parameter in the **`<path>`** element.

For instance, if in a diagnostic system, a DTC's PID value is named "TroubleCode" to access that information the OTX sequence would look as shown in the OTX sample below.

EXAMPLE Sample of OTX-file "DtcExample.otx"

```
<action id="a1">
  <specification>Get trouble code parameter from DTC</specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="diag:ParameterVariable" name="TroubleCodeParameter"/>
    <term xsi:type="diag:GetParameterByPath">
      <diag:parameterContainer xsi:type="diag:ParameterValue" valueOf="dtc"/>
      <diag:path>
        <stepByName xsi:type="StringLiteral" value="TroubleCode"/>
      </diag:path>
    </term>
  </realisation>
</action>

<action id="a2">
  <specification>Get trouble code quantity from parameter</specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="IntegerVariable" name="TroubleCodeValue"/>
    <term xsi:type="diag:GetParameterValueAsInteger">
      <diag:parameter xsi:type="diag:ParameterValue" valueOf="dtc"/>
    </term>
  </realisation>
</action>
```

7.3 Data types

7.3.1 Overview

All datatypes introduced by the OTX DiagCom extension are derived from the OTX Core **`ComplexType`** which means they define complex data types as defined by Part 2 of ISO 13209. The elements described here are handles to the corresponding objects of the underlying communication system.

7.3.2 Syntax

The syntax of all OTX DiagCom exception type declarations is shown in Figure 15.

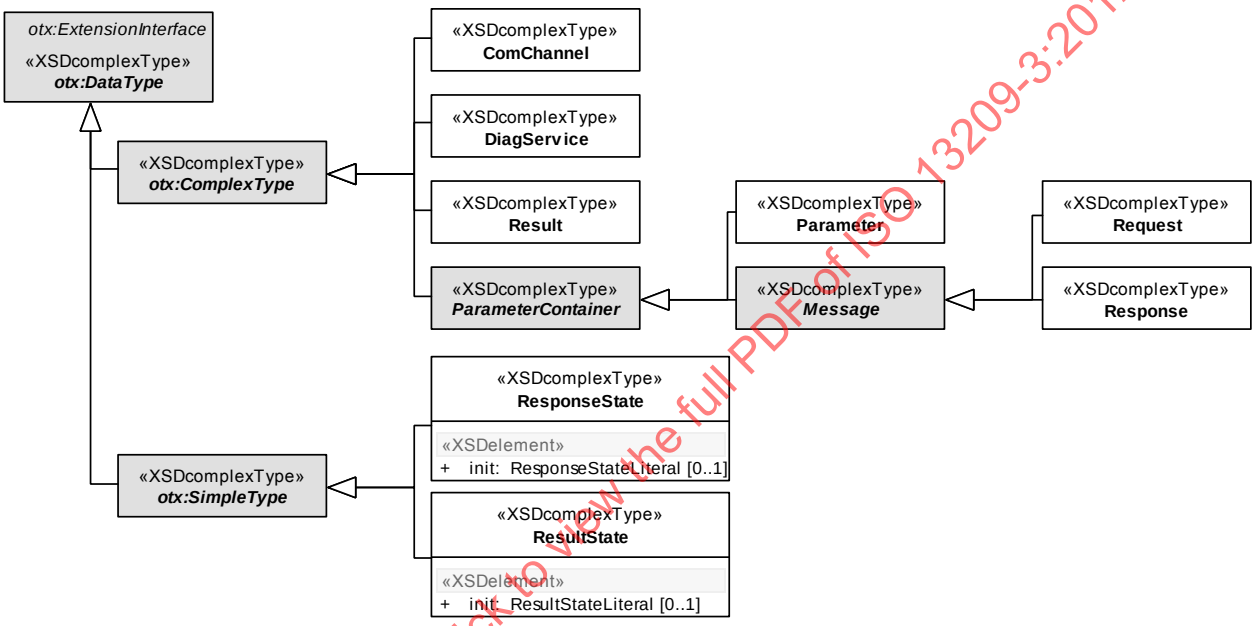

Figure 15 — Data model view: DiagCom data types

7.3.3 Semantics

7.3.3.1 General

The OTX DiagCom data types have no initialisation parts (with the exception of the enumeration types **`ResponseState`** and **`ResultState`**); therefore, these can not be declared constant.

7.3.3.2 ComChannel

A **`ComChannel`** is a handle to a communication channel. It represents the concept of linking to one specific communication endpoint, e.g. an ECU module (physical addressing) or a set of ECU modules (functional addressing).

NOTE In case of a MVCI/ODX based system a **`ComChannel`** handle points to a **`MCDDLogicalLink`** object.

7.3.3.3 DiagService

A **`DiagService`** is a handle to an object representing a diagnostic service, e.g. a service for reading error codes. A **`DiagService`** handle can be used for performing a diagnostic service execution using the **`ExecuteDiagService`** action (see 7.6.4.3.1).

NOTE In case of a MVCI/ODX based system a **`DiagService`** handle represents a **`MCDDiagComPrimitive`** object.

7.3.3.4 Result

A **Result** is a handle to the result of a diagnostic service object. See Figure 4 for an explanation of the structure of **Request**, **Result**, **Response** and **Parameter** instances of a diagnostic service.

NOTE In case of a MVCI/ODX based system a **Result** handle represents a **MCDResult** object.

7.3.3.5 ParameterContainer

The **ParameterContainer** is an abstract data type which subsumes all data types that contain parameters, i.e. **Parameter** and **Message** handles.

7.3.3.6 Parameter

A **Parameter** is a handle to a parameter object of a diagnostic service request or response. It can represent a simple or a complex type parameter, i.e. a **Parameter** handle might point to a simple **Integer** or **String** parameter, or it might correspond to a parameter structure or a list of parameters, depending on the definitions of the underlying communication system. See Figure 4 for an explanation of the structure of **Request**, **Result**, **Response** and **Parameter** instances of a diagnostic service.

NOTE In case of a MVCI/ODX based system a **Parameter** handle represents a **MCDParameter** object (or its specializations **MCDRequestParameter** and **MCDResponseParameter**, respectively).

7.3.3.7 Message

The **Message** element is an abstract data type that encapsulates actual ECU messages.

7.3.3.8 Response

A **Response** is a handle to a response object of a diagnostic service's result. See Figure 4 for an explanation of the structure of **Request**, **Result**, **Response** and **Parameter** instances of a diagnostic service.

NOTE In case of a MVCI/ODX based system a **Response** handle represents a **MCDResponse** object.

7.3.3.9 Request

A **Request** is a handle to a request of a diagnostic service. See Figure 4 for an explanation of the structure of **Request**, **Result**, **Response** and **Parameter** instances of a diagnostic service.

NOTE In case of a MVCI/ODX based system a **Request** handle represents a **MCDRequest** object.

7.3.3.10 ResultState

ResultState is an enumeration type describing the state of a **Result**.

The list of allowed enumeration values is defined as follows:

— **ALL_FAILED**: all ECUs in a functional group (listening to the same functional address) failed to answer, in case of physical addressing: the one requested ECU failed to answer

— **ALL_INVALID**: all ECUs in a functional group (listening to the same functional address) returned an invalid answer, in case of physical addressing: the one requested ECU returned an invalid response

— **ALL_NEGATIVE**: all ECUs in a functional group (listening to the same functional address) returned a negative response, in case of physical addressing: the one requested ECU returned a negative response

— **ALL_POSITIVE**: all ECUs in a functional group (listening to the same functional address) returned a positive response, in case of physical addressing: the one requested ECU returned a positive response

— **FAILED**: some of the ECUs in a functional group (listening to the same functional address) failed to answer

— **INVALID**: some of the ECUs in a functional group (listening to the same functional address) returned an invalid response

— **NEGATIVE**: some of the ECUs in a functional group (listening to the same functional address) returned a negative response

— **POSITIVE**: some of the ECUs in a functional group (listening to the same functional address) returned a positive response

IMPORTANT — `ResultState` values may occur as operands of comparisons (cf. Part 2 of ISO 13209, relational operations). For this case, the following order relation shall apply:
`ALL_FAILED < ALL_INVALID < ALL_NEGATIVE < ALL_POSITIVE < FAILED < INVALID < NEGATIVE < POSITIVE`.

IMPORTANT — When applying `otx:ToString` on a `ResultState` value, the resulting string shall be the name of the enumeration value, e.g. `otx:ToString(POSITIVE)="POSITIVE"`. Furthermore, applying `otx:ToInteger` shall return the index of the value in the `ResultStates` enumeration (smallest index is `0`). The behaviour is undefined for other conversion terms (cf. Part 2 of ISO 13209).

`ResultState` is an **`otx:SimpleType`**. Its members have the following semantics:

— **`<init> : ResultStateLiteral [0..1]`**

This optional element stands for the hard-coded initialisation value of the identifier at declaration time.

— **`value : ResultStates={ALL_FAILED|ALL_INVALID|ALL_NEGATIVE|ALL_POSITIVE|FAILED|INVALID|NEGATIVE|POSITIVE} [1]`**

This attribute shall contain one of the values defined in the **`ResultStates`** enumeration.

IMPORTANT — If the `ResultState` declaration is not explicitly initialized (omitted `<init>` element), the default value shall be `ALL_FAILED`.

7.3.3.11 ResponseState

`ResponseState` is an enumeration type describing the state of a **`Response`**.

The list of allowed enumeration values is defined as follows:

— **FAILED**: the ECU failed to answer

— **INVALID**: the ECUs returned an invalid response

— **NEGATIVE**: the ECUs returned a negative response

— **POSITIVE**: the ECUs returned a positive response

IMPORTANT — `ResponseState` values may occur as operands of comparisons (cf. Part 2 of ISO 13209, relational operations). For this case, the following order relation shall apply:
`FAILED < INVALID < NEGATIVE < POSITIVE`.

IMPORTANT — When applying `otx:ToString` on a `ResponseState` value, the resulting string shall be the name of the enumeration value, e.g. `otx:ToString(POSITIVE)="POSITIVE"`. Furthermore, applying `otx:ToInteger` shall return the index of the value in the `ResponseStates` enumeration (smallest index is `0`). The behaviour is undefined for other conversion terms (cf. Part 2 of ISO 13209).

ResponseState is an **otx:SimpleType**. Its members have the following semantics:

— **<init> : ResponseStateLiteral [0..1]**

This optional element stands for the hard-coded initialisation value of the identifier at declaration time.

— **value : ResponseStates={FAILED|INVALID|NEGATIVE|POSITIVE} [1]**

This attribute shall contain one of the values defined in the **ResponseStates** enumeration.

IMPORTANT — If the ResponseState declaration is not explicitly initialized (omitted <init> element), the default value shall be FAILED.

7.4 Exceptions

7.4.1 Overview

All elements referenced in this clause are derived from the OTX Core **Exception** type as defined by Part 2 of ISO 13209. They represent the full set of exceptions added by the OTX DiagCom extension.

7.4.2 Syntax

The syntax of all OTX DiagCom exception type declarations is shown in Figure 16.

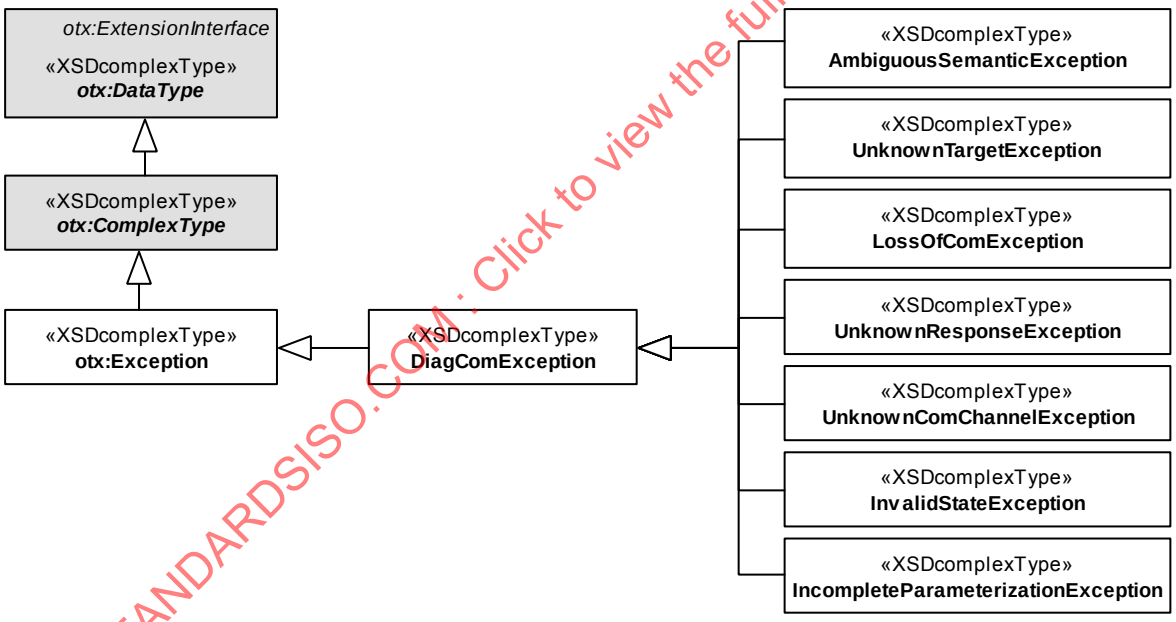

Figure 16 — Data model view: DiagCom exceptions

7.4.3 Semantics

7.4.3.1 General

Since all OTX DiagCom exception types are implicit exceptions without initialisation parts, they can not be declared constant.

7.4.3.2 DiagComException

The **DiagComException** is the super class for all exceptions in the DiagCom extension. A **DiagComException** shall be used in case the more specific exception types described in the remainder of this section do not apply to the problem at hand.

7.4.3.3 AmbiguousSemanticException

The **AmbiguousSemanticException** is thrown if there is more than one object with the same semantic attribute matching a DiagCom activity. This exception can be thrown by the following actions/terms:

— **CreateDiagServiceBySemantic**

— **GetParameterBySemantic**

— **SetParameterValueBySemantic**

7.4.3.4 UnknownTargetException

The **UnknownTargetException** is thrown if a DiagCom action or term references an object in the underlying communication system that is not available or not defined.

7.4.3.5 LossOfComException

The **LossOfComException** is thrown if there is a communication breakdown during a service execution, e.g. in case the cable to the vehicle gets unplugged.

7.4.3.6 UnknownResponseException

This exception is thrown in case execution of a diagnostic service returned a response that was not mapped by the **ExecuteDiagService** action (see 7.6.4.3.1).

7.4.3.7 UnknownComChannelException

This exception is thrown in case a **Response** handle can not be linked to a communication channel when using the **GetComChannelNameFromResponse** term (see 7.7.2.3.4).

7.4.3.8 InvalidStateException

This exception is thrown in case the **StartRepeatedExecution** action used on a **DiagService** that is already executing repeatedly, in case the **StopRepeatedExecution** action used on a **DiagService** that is not currently executing repeatedly or in case the **SetRepetitionTime** action is used on a **DiagService** that is currently executing repeatedly.

7.4.3.9 IncompleteParameterizationException

This exception is thrown in case a **DiagService** was executed where not all request parameters have been set that didn't have a default value.

7.5 Variable access

7.5.1 Overview

As specified in Part 2 of ISO 13209, OTX extensions shall define a variable access type for each datatype they define (exception types inclusively). All variable access types are derived from the OTX Core **`Variable`** extension interface. The following specifies all variable access types defined for the DiagCom extension.

7.5.2 Syntax

Figure 17 shows the syntax of the DiagCom extension's variable access types.

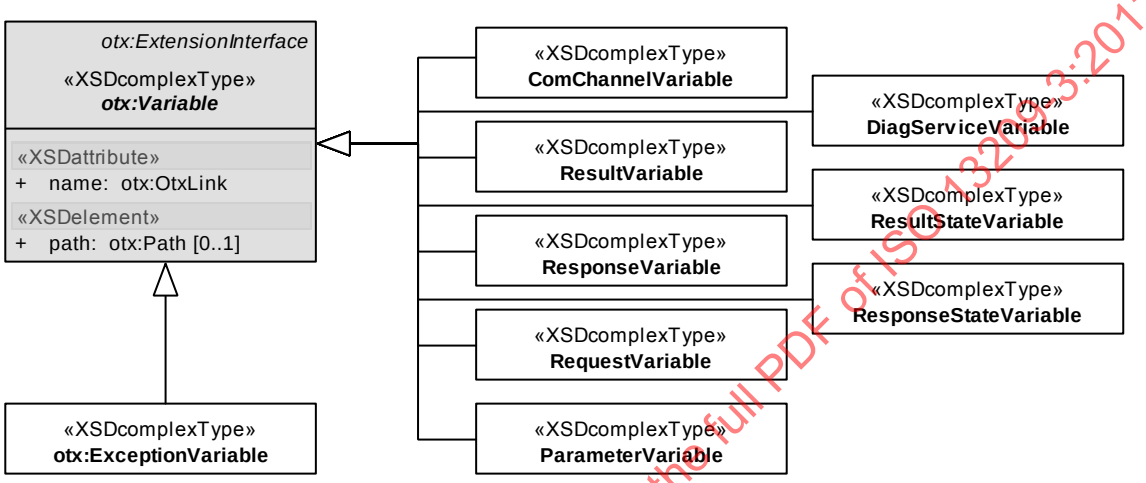

Figure 17 — Data model view: DiagCom variable access types

7.5.3 Semantics

The general semantics for all variable access types shall apply. Please refer to Part 2 of ISO 13209 for details.

7.6 Actions

7.6.1 Overview

All of the elements shown in Figure 18 extend the `otx:ActionRealisation` extension interface as defined by Part 2 of ISO 13209.

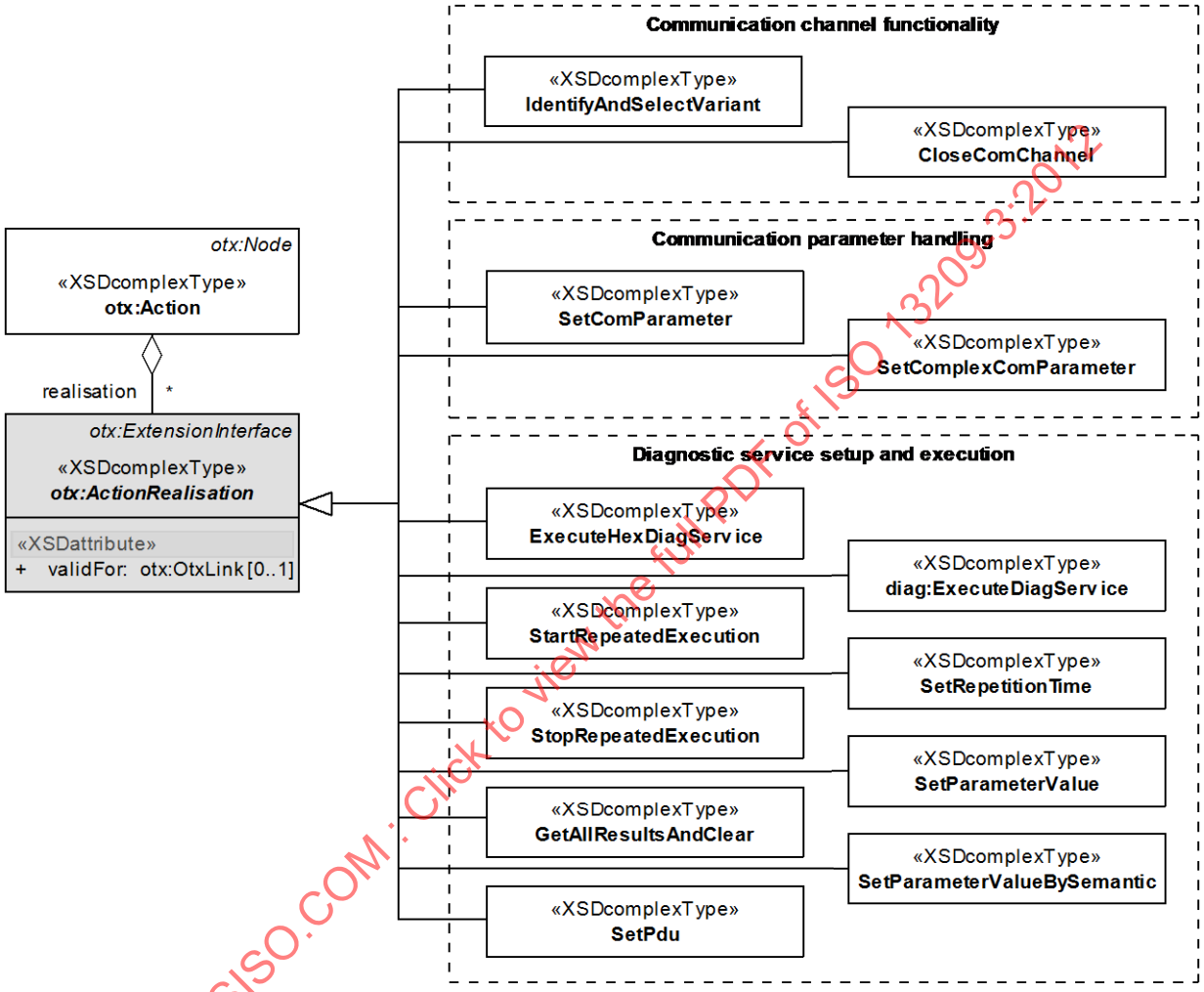

Figure 18 — Data model view: DiagCom actions overview

7.6.2 ComChannel related actions

7.6.2.1 Description

All actions described in this clause effect changes on a `ComChannel` handle.

7.6.2.2 Syntax

Figure 19 shows the syntax of all `ComChannel` related `ActionRealisation` types of the DiagCom extension.

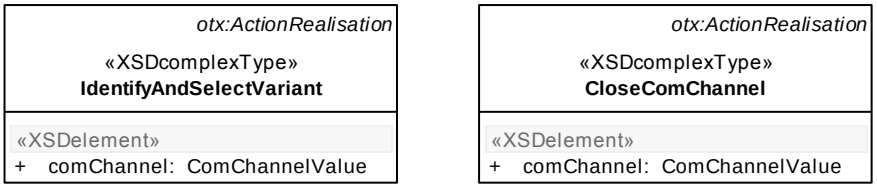

Figure 19 — Data model view: `ComChannel` related actions

7.6.2.3 Semantics

7.6.2.3.1 IdentifyAndSelectVariant

The **`IdentifyAndSelectVariant`** action shall be used to tell the communication backend to identify the ECU variant that is present at runtime at a specific communication channel. In case an ECU variant can be identified, the communication channel is switched to point to that specific variant.

This standard can not make assumptions about whether the vehicle communication component used by an OTX runtime supports the concept of ECU variant identification or about the behaviour of the communication component in case it does. The relevant parts of the DiagCom extension are based on the following assumptions:

— A communication channel to an ECU is associated with a data set describing diagnostic behaviour of a specific variant of that ECU.

— The vehicle communication component is able to explicitly perform an ECU variant identification operation on a communication channel to an ECU.

— The required logic and data for performing the variant identification is intrinsic to the vehicle communication component, i.e. there is no additional external information required for the communication component to perform the ECU variant identification.

— After an ECU variant has been identified, the vehicle communication component is able to explicitly associate the communication channel to that ECU with the specific data set for that ECU variant, effectively switching the communication channel from the old variant data set to a new one.

The **`IdentifyAndSelectVariant`** action tells the runtime system to perform the variant identification operation on the provided communication channel and then switch the data set associated with that channel to the one fitting the newly identified variant (if any). Please refer also to the **`GetComChannel`** term (see 7.7.2.3.3) which tells the runtime system to create a new communication channel, immediately perform the variant identification operation on the new communication channel and then switch the data set associated with that channel to the one fitting the newly identified variant (if any).

NOTE In case an ODX/MVCI system is used, the exact semantics of variant identification and selection are specified by the ISO ODX and MVCI standards.

The members of the **`IdentifyAndSelectVariant`** action the following semantics:

— **`<comChannel> : ComChannelValue [1]`**

This element comprises the communication channel which shall be used for identifying the actual variant of the ECU the communication channel is connected to.

Throws:

— **`LossOfComException`**
If communication to the ECU was interrupted during the variant identification process.

7.6.2.3.2 CloseComChannel

The action **`CloseComChannel`** tells the OTX runtime system that the communication channel to an ECU can be closed and associated resources can be freed. Please note that the use of the **`CloseComChannel`** action by an OTX sequence only indicates that the channel is not needed any more – it is up to the implementation of a specific runtime system whether it actually frees all resources and closes the channel at this point. If a diagnostic sequence uses a **`ComChannel`** handle after it's been freed by a **`CloseComChannel`** action, the runtime system shall throw an **`otx:InvalidReferenceException`**.

The members of the **`CloseComChannel`** action have the following semantics:

— **`<comChannel>: ComChannelVariable [1]`**

This element comprises communication channel which shall be closed.

7.6.3 ComParameter related actions

7.6.3.1 Description

All actions described in this clause change communication parameter settings of a **`ComChannel`** handle. For example, CAN timeouts or baudrate settings usually are modelled as communication parameters.

7.6.3.2 Syntax

Figure 20 shows the syntax of all parameter handling related **`ActionRealisation`** types of the DiagCom extension.

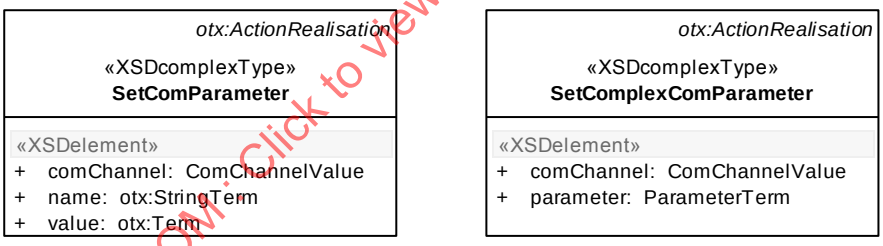

Figure 20 — Data model view: Communication parameter handling

7.6.3.3 Semantics

7.6.3.3.1 SetComParameter

The **`SetComParameter`** action shall be used to change the value of a communication parameter used by the communication backend. For example, bus timeouts or baud rates can be set using the **`SetComParameter`** node.

NOTE In case an ODX/MVCI system is used for vehicle communication, the communication parameter names and data types that can be set are defined by the D-PDU API/ODX communication parameter specification.

The members of the **`SetComParameter`** action have the following semantics:

— **`<comChannel> : ComChannelValue [1]`**

This element comprises the communication channel where the communication parameter shall be modified.

— **<name> : otx:StringTerm [1]**

This element specifies the name of the communication parameter which shall be changed.

— **<value> : otx:Term [1]**

This element specifies the new communication parameter value that shall be set.

Throws:

— **UnknownTargetException**
If no communication parameter with the specified name exists.

— **otx:TypeMismatchException**
If the specified quantity type does not match the data type of the communication parameter to be set.

7.6.3.3.2 SetComplexComParameter

The **SetComplexComParameter** action is an enhanced variant of **SetComParameter**. The difference between these actions is that in this case complex data types can be used.

NOTE For instance, in an ODX/MVCI based system, complex communication parameter data types are used to define response ID lists for the functional addressing use case.

The members of the **SetComplexComParameter** action have the following semantics:

— **<comChannel> : ComChannelValue [1]**

This element comprises the communication channel where the communication parameter shall be modified.

— **<parameter> : ParameterTerm [1]**

This element comprises the parameter structure which shall be set.

Throws:

— **otx:TypeMismatchException**
If the specified **<parameter>** element does not match the communication parameter to be set.

7.6.4 DiagService related actions

7.6.4.1 Description

Actions described in this clause are used for setting up and performing actual ECU communication.

7.6.4.2 Syntax

Figure 21 shows the syntax of all **`ActionRealisation`** types of the DiagCom extension which relate to diagnostic service configuration and execution.

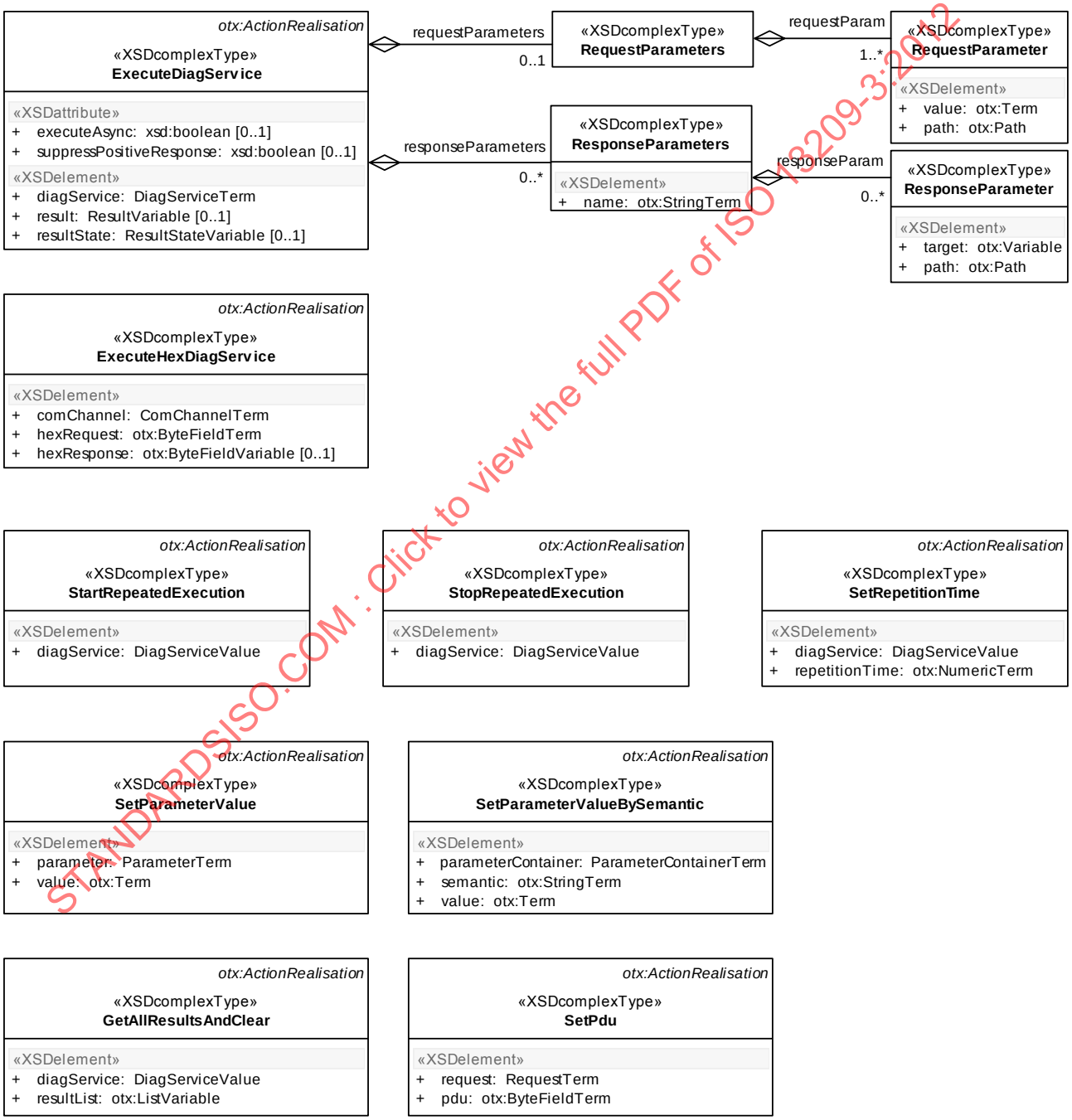

Figure 21 — Data model view: `DiagService` related actions

7.6.4.3 Semantics

7.6.4.3.1 ExecuteDiagService

The **`ExecuteDiagService`** action shall be used for performing diagnostic vehicle communication. An **`ExecuteDiagService`** node in an OTX sequence indicates to the runtime system that at this point, a service request shall be transmitted to one or more ECUs, and that any associated responses might have to be provided to the OTX sequence. To be able to do this, the **`ExecuteDiagService`** action requires two sets of information:

a) The **`DiagService`** to use

b) The definition for mapping OTX values to the service's request parameters as well as the values of the service's response parameters back to OTX variables.

The writing/reading of values to/from service parameters can be done in two ways, depending on whether a service's parameter structure is known at OTX authoring time or will have to be dynamically evaluated at run time.

— **Inline mapping:** In case a service's parameter structure is static (known at authoring time), the **`ExecuteDiagService`** action can be used to define request and response parameter mappings *inline* through its **`<RequestParameters>`** and **`<ResponseParameters>`** members. A detailed explanation will be given in the remainder of this clause. Please note that the inline mapping approach is only meant to be used in cases where there is one response from one ECU to a diagnostic service. In case more than one ECU respond to a service request and/or ECUs respond more than once, the inline mapping will only relate to the first response within the first result.

— **Dynamic response:** In case a service's parameter structure is dynamic at runtime (not known at authoring time), it is possible to use terms defined by the DiagCom extension to evaluate request and response parameter structures by explicit OTX statements. This way, it is possible to e.g. loop through a service response that contains a list of structures. An example for a diagnostic service where the response parameter structure varies in such a way at runtime is the read DTC service as defined by the UDS protocol [ISO 14229]. As it returns the number of DTCs that are present within the ECU at the time of the service execution, it can not be known at authoring time exactly how many or which DTCs will be contained in the service's response at run time.

Manual evaluation of results is also needed in case a diagnostic service produces a complex result structure. This can happen in two cases, the first one being a diagnostic service where one service execution results in multiple, cyclic responses from an ECU. In this case, the diagnostic service will have multiple results associated with it – one for each of the ECUs responses along the timeline. The individual results have to be accessed and evaluated through the terms defined in 7.6.4.4. The second case is when one diagnostic request results in multiple responses from different ECUs, i.e. when using functional addressing. In this case, there is one result associated with the diagnostic service, which in turn contains multiple responses – one for each ECU that responded to the functional request. The individual responses have to be accessed and evaluated using the terms defined in 7.6.4.4. Please note that in theory, both cases can also be combined – it is possible to imagine a repeatedly sent diagnostic service which uses functional addressing, producing multiple results including multiple responses each. Please refer to 7.6.4.4 for more details.

The following rules apply for inline mapping of parameters in the **`ExecuteDiagService`** element:

— Allowed OTX data types for parameter mapping: **`RequestParameter`** and **`ResponseParameter`** mappings are only allowed to reference OTX simple types, OTX **`ByteField`**, **`List`** and **`Map`** types as well as OTX **`Quantity`** types.

— Response mapping and exception behaviour of **`ExecuteDiagService`**: Generally, the **`ExecuteDiagService`** action can contain multiple sequences of response parameters - one for each response that is of interest to the OTX sequence. If at runtime an ECU response is encountered that is not represented by a **`<responseParameters>`** element in the **`ExecuteDiagService`** node, this shall

result in an **UnknownResponseException**. To be able to determine the nature of the problem, in this case it is possible to retrieve the **DiagService** object that was executed from the **UnknownResponseException** using the term **GetDiagServiceFromException** (see 7.7.3.3.7), and then to analyse it's response structure by looking at the PDU of the response (see term **GetPdu** in 7.7.4.3.4) or by the usual traversal methods using the appropriate DiagCom terms.

— Defining which service responses are of interest to the diagnostic sequence: as has been elaborated in the previous paragraph, an **ExecuteDiagService** node can freely define which of the responses of a service it is interested in by providing a **<responseParameters>** element with the appropriate response name. Please note that a **<responseParameters>** element is allowed to be empty, it is not required that any mappings of actual response parameters are defined. This allows a diagnostic sequence author fine-grained control over the behaviour of the **ExecuteDiagService** action: whether or not the **ExecuteDiagService** action is supposed to throw an exception in case an unexpected/unwanted response is encountered is simply a matter of providing a potenitally empty response parameter mapping for the response in question. For example if a diagnostic sequence author is only interested in a positive response and considers the occurence of a negative response to be an error, he should only provide a mapping for the positive response's name. If the occurence of the negative response is not considered an error and shall not cause an **UnknownResponseException**, he can simply provide a mapping for the negative response as well, which can be empty if he doesn't care about the acutal response parameters. The princple applies for any combination of responses (positive, negative, global negative, etc.) – simply by providing an appropriate **<responseParameters>** element, the author can control whether the occurence of the response shall be treated as an **UnknownResponseException** (no mapping) or not (mapping present).

NOTE In case the OTX DiagCom extension functionality is used with an ODX/MVCI system, the name of the **<responseParameters>** element is the **SHORT-NAME** of the response (positive, local negative, global negative) in the corresponding ODX data. In case of the DiagCom extension being used with a different communications backend that has no concept of multiple responses for a service/no names for responses, a similar internal naming convention could be used to map OTX response parameters to positive or negative responses.

The **ExecuteDiagService** action can also be configured to tell the communication backend that the addressed ECU shall suppress the sending of a positive response, in case that concept is supported by the communication backend and by the specific diagnostic service that is to be executed. This shall happen if the value of the attribute **suppressPositiveResponse** is set to **true**. If the attribute is omitted from **ExecuteDiagService**, the default value shall be **false**.

The members of **ExecuteDiagService** action have the following semantics:

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

This option tells the communication backend to make this diagnostic service execution non-blocking. This means that if **executeAsync** is set to **true**, the OTX execution flow will immediately move on to the next **Action**, without waiting for the result of the **ExecuteDiagService** action. As a consequence, any response parameter mappings defined by this **ExecuteDiagService** action are ignored: as the diagnostic service execution has not necessarily finished with the execution of the **ExecuteDiagService** node, the OTX variables that are statically mapped to contain the service's responses cannot contain a value at this point. The use of the **executeAsync** capability always requires the OTX sequence to perform dynamic response interpretation. An OTX sequence can make use of the **DiagServiceEventSource** term (refer to 7.7.9.3.1) to be notified when a new result for an asynchronously executed diagnostic service has arrived.

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

This option tells the ECU(s) addressed by the diagnostic service to suppress sending of a positive response. This feature has to be supported by the underlying communication system, diagnostic protocol and specific diagnostic service (compare to **suppressPosRspMsgIndicationBit** of the UDS protocol [ISO 14229]).

— **<diagService> : DiagServiceTerm [1]**

The element specifies the service which shall be executed. Syntax and semantics of expression **DiagServiceTerm** are specified in 7.7.3.

— **<requestParameters> : RequestParameters [0..1]**

In this part OTX values are mapped to service request parameters.

— **<requestParam> : RequestParameter [1..*]**

This element shall be used to assign one specific OTX value to one specific request parameter of the diagnostic service.

— **<value> : otx:Term [1]**

This element specifies the value which shall be assigned to a service's request parameter. At runtime, the value is yieled by evaluation of the term given by the **<value>** member element. It is only allowed to map from OTX simple data types, OTX bytefields, lists and maps as well as OTX quantities as defined in the OTX Quantities extension. The specific data type to be used in a mapping depends on the type expected by the diagnostic service's request parameter.

— **<path> : otx:Path [1]**

This element is described in the OTX Core language specification. Here it shall be used to locate the request parameter to which the value shall be assigned to.

The usage of the following elements depends on the elements necessary stepping through the parameter hierarchy. If more steps like **<stepByIndex>** / **<stepByName>** are necessary these elements get combined.

— **<stepByIndex> : otx:NumericTerm [1]**

The element shall be used locate a parameter inside a list of request parameters. For example, in case a diagnostic service request contains a list of three parameter structures, the **<stepByIndex>** element can be set to **0** to indicate a mapping to the first of these three list entries. **Float** values shall be truncated.

— **<stepByName> : otx:StringTerm [1]**

The element shall be used to locate a named parameter of a request. For example, in case a diagnostic service request contains three parameters **"RequestParameterA"**, **"RequestParameterB"** and **"RequestParameterC"**, the **<stepByName>** element can be set to **"RequestParameterB"** to indicate a mapping to the second of these request parameters.

— **<responseParameters> : ResponseParameters [0..*]**

In this part service response parameters are assigned to OTX variables.

— **<name> : otx:StringTerm [1]**

This element shall contain the name of the response that shall be used for this mapping definition.

NOTE In case an ODX/MVCI based communication backend is used, this element shall contain the **SHORT-NAME** of the **RESPONSE** element that shall be mapped. In case a non-ODX based system is used, this element should contain an equivalent response identifier to denote a positive, negative etc. response.

— **<responseParam> : ResponseParameter [0..*]**

This element shall be used to assign one specific response parameter value of a diagnostic service to a specific OTX variable.

— **<target> : otx:Variable [1]**

This element specifies the OTX variable the response value shall be assigned to. It is only allowed to assign to OTX simple data types, OTX bytefields, lists and maps and OTX quantities.

— **<path> : otx:Path [1]**

This element is described in the OTX core language definition. Here it shall be used to define which response parameter shall be mapped to an OTX variable.

The member elements **<stepByIndex>** and **<stepByName>** are not further specified here since they have identical semantics as specified for the **<requestParameters>** mapping explained above.

— **<result> : otx:ResultVariable [0..1]**

After execution of the diagnostic service, the first result shall be assigned to the variable given by this optional element.

In order to get further results (e.g. in case of cyclic execution), the **GetAllResults** term shall be used (cf. 7.7.5.3.4).

— **<resultState> : otx:ResultStateVariable [0..1]**

After execution of the diagnostic service, the state of its first result (i.e. whether the ECU(s) answered at all, correctly, positively or negatively) shall be assigned to the variable given by this optional element. Allowed result state values are specified by the **ResultState** data type as defined in 7.3.3.10.

In order to get the result state of further results (e.g. in case of cyclic execution), the **GetResultState** term shall be used (cf. 7.7.5.3.8).

Throws:

— **IncompleteParameterizationException**
One or more request parameters of the diag service have not been set and don't have a default value.

— **LossOfComException**
If communication to the ECU was interrupted during diagnostic service execution.

— **UnknownTargetException**
If no request or response parameter with the specified name in any of the parameter mappings exist.

— **UnknownResponseException**
If execution of the diagnostic service returned a response that was not mapped by any **<responseParameters>** element.

— **otx:OutOfBoundsException**
If a conversion can not be made because the value of an OTX variable exceeds the limits of the target data type of a parameter of the vehicle communication component.

— **otx:TypeMismatchException**
If an invalid OTX data type is mapped to a request parameter or a response parameter is mapped to an invalid OTX data type. For instance it is thrown if a **String** variable gets mapped onto a request parameter that is of type **Integer**.

Associated checker rules:

— DiagCom_Chk001 – No Path in ExecuteDiagService response parameter arguments

An example for the **ExecuteDiagService** action is given in 7.6.4.4.

7.6.4.3.2 ExecuteHexDiagService

The **ExecuteHexDiagSevice** action allows the sending of diagnostic services by directly entering the request byte stream, bypassing the symbolic level that is utilized by the normal **ExecuteDiagService** action. By using this action, ECUs can be directly addressed with hex requests defined by the OTX sequence author. Possible use cases for this functionality are errors in the diagnostic database which have to be bypassed to achieve a temporary workaround. The response to an **ExecuteHexDiagService** is provided as a **ByteField** containing the raw, uninterpreted ECU response message. Please note that the **ExecuteHexDiagSevice** action is only meant to be used in cases where there is one response from one ECU to a diagnostic service. In case more than one ECU respond to a service request and/or ECUs respond more than once, the **<hexResponse>** assignment will only contain the first **Response** of the first **Result**.

A PDU as understood by the DiagCom extension comprises the complete payload of a message including the service identifier and any other request parameters. It does not include header or checksum bytes from underlying protocol layers.

The members of the **ExecuteHexDiagService** action have the following semantics:

— **<comChannel> : ComChannelTerm [1]**

This element shall comprise the handle of the communication channel which shall be used for communication with the ECU.

— **<hexRequest> : otx:ByteFieldTerm [1]**

This element shall contain the service request as a set of raw bytes.

— **<hexResponse> : otx:ByteFieldVariable [0..1]**

This element specifies the OTX **ByteField** variable to which the raw response bytes of the service shall be assigned.

Throws:

— **LossOfComException**
If communication to the ECU was interrupted during diagnostic service execution.

7.6.4.3.3 StartRepeatedExecution

This action causes a **DiagService** to be executed repeatedly by the underlying communication backend. The repetition time shall be set through the **SetRepetitionTime** action and queried by the **GetRepetitionTime** term. The **StartRepeatedExecution** action will return immediately, the results of the **DiagService** created by the repeeated service execution can be queried through the **GetFirstResult** or **GetAllResults** terms or the **GetAllResultsAndClear** action. Each new result (each execution cycle) will cause a **DiagServiceEvent** to be raised by the **DiagService** object. To stop a repeated service execution, the **StopRepeatedExecution** action is to be used.

The members of the **StartRepeatedExecution** action have the following semantics:

— **<diagService> : DiagServiceValue [1]**

The element specifies the service which shall be executed repeatedly.

Throws:

— **InvalidStateException**
The diag service is already being executed repeatedly.

— **IncompleteParameterizationException**
One or more request parameters of the diag service have not been set and don't have a default value.

7.6.4.3.4 StopRepeatedExecution

This action causes the repeated execution of a **DiagService** to be stopped. The results of the **DiagService** created by the repeated service execution can be queried through the **GetFirstResult** / or **GetAllResults** terms or the **GetAllResultsAndClear** action. To start a repeated service execution, the **StartRepeatedExecution** action is to be used.

The members of the **StopRepeatedExecution** action have the following semantics:

— **<diagService> : DiagServiceValue [1]**

The element specifies the service which shall not be executed repeatedly any more.

Throws:

— **InvalidStateException**
The diag service is currently not being executed repeatedly.

7.6.4.3.5 SetRepetitionTime

This action sets the repetition cycle time of a diagnostic service. The repetition time is always provided in millisecond (ms) granularity. It is not allowed to set the repetition time of a service while it is being executed repeatedly. To start or stop a repeated service execution, the StartRepeatedExecution and StopRepeatedExecution actions are to be used ().

The members of the **SetRepetitionTime** action have the following semantics:

— **<diagService> : DiagServiceValue [1]**

The element specifies the service where the repetition time should be set.

— **<repetitionTime> : otx:NumericTerm [1]**

This element specifies the repetition cycle time in milliseconds (ms). **Float** values shall be truncated.

Throws:

— **InvalidStateException**
The diag service is currently being executed repeatedly.

— **otx:OutOfBoundsException**
The repetition time value is negative.

7.6.4.3.6 GetAllResultsAndClear

This action retrieves all available result entries from a diagnostic service and then clears the diagnostic communication system's result buffer. The results are provided as a list of **Result** elements. In comparison to the term **GetAllResults** defined in 7.7.5.3.4, **GetAllResultsAndClear** is modelled as an **ActionRealisation** because it changes the **DiagService** object it is invoked on by clearing its result buffer.

This action is designed based on the assumption that a diagnostic communication component as used by an OTX runtime has the capability to buffer results it receives from ECUs. Especially for dealing with system behaviour as illustrated in Figure 6 (one ECU returning multiple results for one diagnostic request) a communication system requires a buffering concept for ECU results. The following assumptions are made in the context of the DiagCom extension regarding the result buffer:

— The result buffer is owned and managed by the vehicle communication component and is outside the scope of an OTX runtime.

— Every **DiagService** object has an associated result buffer which contains any results that were received as a reaction to an **ExecuteDiagService** action.

— This result buffer is of finite size, i.e. a loop buffer that will wrap around after a number of results have been received by the vehicle communication component.

— The DiagCom term **GetFirstResult** (see 7.7.5.3.3) only fetches the first (in time) result out of the communication component's result buffer, but does not modify that buffer.

— The DiagCom term **GetAllResults** (see 7.7.5.3.4) fetches all results present at the time of the call out of the communication component's result buffer, but does not modify that buffer. The list of results that is returned to OTX will be in ascending order from first (oldest) to last (most recent) result.

— The DiagCom action **GetAllResultsAndClear** (see 7.6.4.3.6) fetches all results present at the time of the call out of the communication component's result buffer and tells the communication component to clear the buffer afterwards. The list of results that is returned to OTX will be in ascending order from first (oldest) to last (most recent) result.

The members of the **GetAllResultsAndClear** action have the following semantics:

— **<diagService> : DiagServiceValue [1]**

This element specifies the diagnostic service to retrieve results from. Syntax and semantics of expression **DiagServiceVariable** are specified in 7.5.

— **<resultList> : otx:ListVariable [1]**

This element specifies the **List** to which the list of **Result** items shall be assigned.

7.6.4.3.7 SetParameterValue

This action sets a specific value to a **Parameter** element. The value to be set is to be provided as an OTX simple type, an OTX lists or maps or an OTX **Quantity** as defined in 17.

The members of the **SetParameterValue** action have the following semantics:

— **<parameter> : ParameterTerm [1]**

This element specifies the parameter which will be set. Syntax and semantics of the **ParameterTerm** type are specified in 7.7.6.3.9.

— **<value> : otx:Term [1]**

This element specifies the value that shall be set to the **Parameter**. Allowed input types are OTX simple types, OTX bytefields, lists and maps and OTX quantities.

Throws:

— **otx:OutOfBoundsException**
If the conversion can not be made because the value of an OTX variable exceeds the limits of the target data type of a parameter of the vehicle communication component.

— **otx:TypeMismatchException**
If the data type of the OTX value to be set does not match the parameter vehicle communication component. For instance it is thrown if a **String** variable gets mapped onto a parameter that is of type **Integer**.

7.6.4.3.8 SetParameterValueBySemantic

This action sets a value to a **<parameter>** element with a specific semantic. This action is used in case the backend communication system provides the means to associate semantic metadata with parameters of diagnostic services. The value to be set is to be provided as an OTX simple type, an OTX bytefield, list or map or an OTX **Quantity** as defined in 17.

NOTE 1 The ability to assign a semantic value to a diagnostic service or service parameter allows applications working with diagnostic data to access functionality in a manner more abstract than directly pointing to specific names of services/parameters. For example, the diagnostic service to be used for DTC reading could be required to carry the semantic value **"DEFAULT-FAULTREAD"** company wide or even industry wide, no matter what the actual name of the service in a specific data set is. Through the use of semantic attributes, certain elements of a diagnostic data set can become universally identifiable, even though the names of these elements have to conform to user-specific conventions and therefore differ between or even within companies.

NOTE 2 When using an ODX/MVCI based system it is mandatory to assign specific semantic attribute values to the parameters used by diagnostic services for implementing DDLID (Dynamically Defined Local Identifier) functionality, like the DDLID-POS semantic attribute that's used for indicating the parameter that defines the position of a value in a dynamically created response.

The members of the **SetParameterValueBySemantic** action have the following semantics:

— **<parameterContainer> : ParameterContainerTerm [1]**

The object that contains the parameter that shall be changed. Syntax and semantics of expression **ParameterContainerTerm** are specified in 7.7.6.3.9.

— **<semantic> : otx:StringTerm [1]**

This element specifies the semantic of the parameter that shall be modified.

— **<value> : otx:Term [1]**

This element specifies the value that shall be set to the **Parameter**. Allowed input types are OTX simple types, OTX bytefields, lists and maps and OTX quantities.

Throws:

— **AmbiguousSemanticException**
If there are none or more than one parameters present in the **ParameterContainerTerm** with the semantic value specified by the **<semantic>** element.

— **otx:OutOfBoundsException**
If the conversion can not be made because the value of an OTX variable exceeds the limits of the target data type of a parameter of the vehicle communication component.

— **otx:TypeMismatchException**
If the data type of the OTX value to be set does not match the parameter vehicle communication component. For instance it is thrown if a **String** variable is mapped onto an **Integer**-type parameter.

7.6.4.3.9 SetPdu

This action is used to directly set a specific **ByteField** to a **Request** instance, without using the symbolic level provided by the parameter mapping mechanism of the **ExecuteDiagService** action or the related DiagCom terms. In addition to **SetPdu**, there exists a term **GetPdu** which is used to retrieve the raw byte representation from a **Response** instance (see 7.7.4.3.4). **SetPdu** is modelled as an **ActionRealisation** because it modifies the object it is invoked on.

A PDU as understood by the OTX DiagCom extension comprises the complete payload of a message including the service identifier and any other request parameters. It does not include header or checksum bytes from underlying protocol layers.

The members of the **SetPdu** action have the following semantics:

— **<request> : RequestTerm [1]**

This element specifies the **Request** to which the value given by **<pdu>** shall be assigned. Syntax and semantics of expression **RequestTerm** are specified in 7.7.3.3.8.

— **<pdu> : otx:ByteFieldTerm [1]**

The **ByteField** which shall be written to the **Request**.

7.6.4.4 Example

The example below illustrates the inline mapping usage of the **ExecuteDiagService** action node using the prefix **"ODX_"** to indicate identifiers that link to the ODX/MVCI communication component. The example executes a diagnostic service called **"ODX_DiagServiceName"** on a **ComChannel** defined by the variable **"ComChannelHandle"**. The request parameter with the name "**ODX_RequestParameterShortName"** is set to the string value **"ExampleParameterValue"**, and the response parameter named **"ODX_ResponseParameterShortName"** of the response named **"ODX_PositiveResponseName"** is mapped to the OTX variable called **"outputParamHandle"**.

Figure 22 shows the usage of the **<stepByName>** element. The first XML example defines a reference to parameter **"ReqParam2"**. In the second example a reference over two levels to **"StructParam2"** is shown.

Figure 23 shows the usage of the **<stepByIndex>** element. The XML example shows a reference to a parameter over three levels, inside a list of parameters structures. The first and the third path steps are **<stepByName>** references. The second is a **<stepByIndex>** reference to indicate the desired list entry.

More comprehensive examples for both inline and dynamic response mappings are provided in Annex E.

EXAMPLE Sample of OTX-file "ExecuteDiagServiceExample.otx"

```
<action id="a1">
  <specification>Execute a diagservice and map a response to an OTX variable</specification>
  <realisation xsi:type="diag:ExecuteDiagService">
    <diag:diagService xsi:type="diag:CreateDiagServiceByName">
      <diag:comChannel xsi:type="diag:ComChannelValue" valueOf="comChannelHandle"/>
      <diag:name xsi:type="StringLiteral" value="ODX_DiagServiceName"/>
    </diag:diagService>
    <diag:requestParameters>
      <diag:requestParam>
        <diag:value xsi:type="StringLiteral" value="ExampleParameterValue"/>
        <diag:path>
          <stepByName xsi:type="StringLiteral" value="ODX_RequestParameterShortName"/>
        </diag:path>
      </diag:requestParam>
    </diag:requestParameters>
    <diag:responseParameters>
      <diag:name xsi:type="StringLiteral" value="ODX_PositiveResponseName"/>
      <diag:responseParam>
        <diag:target xsi:type="diag:ParameterVariable" name="outputParamHandle"/>
        <diag:path>
          <stepByName xsi:type="StringLiteral" value="ODX_ResponseParameterShortName"/>
        </diag:path>
      </diag:responseParam>
    </diag:responseParameters>
  </realisation>
</action>

<action id="a2">
  <specification>Deselect the communication channel</specification>
  <realisation xsi:type="diag:CloseComChannel">
    <diag:comChannel xsi:type="diag:ComChannelVariable" name="comChannelHandle"/>
  </realisation>
</action>
```

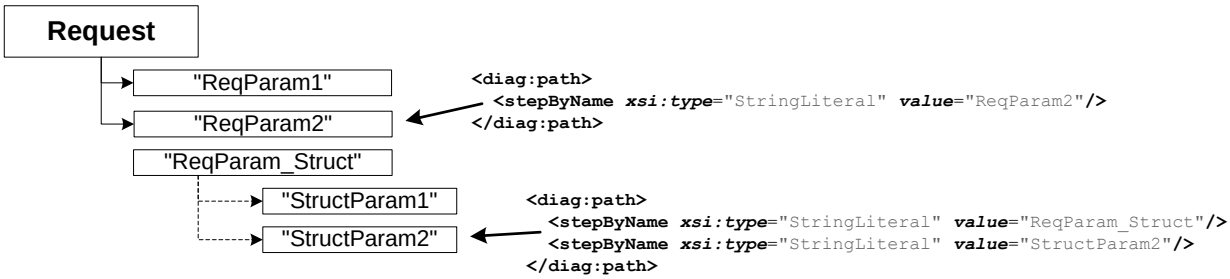

Figure 22 — Referencing parameters via <stepByName>

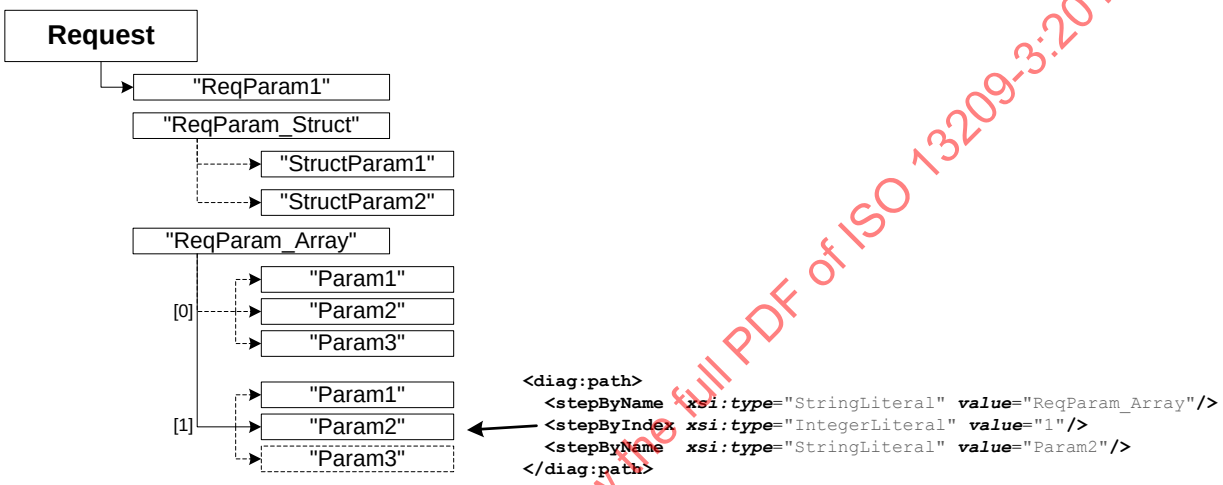

Figure 23 — Referencing parameters via <stepByName> and <stepByIndex>

7.7 Terms

7.7.1 Overview

All of the DiagCom terms shown in Figure 24 extend the **`otx:Term`** extension interface as defined by Part 2 of ISO 13209. Information about the specific super class of a term is provided in the individual term description clauses below.

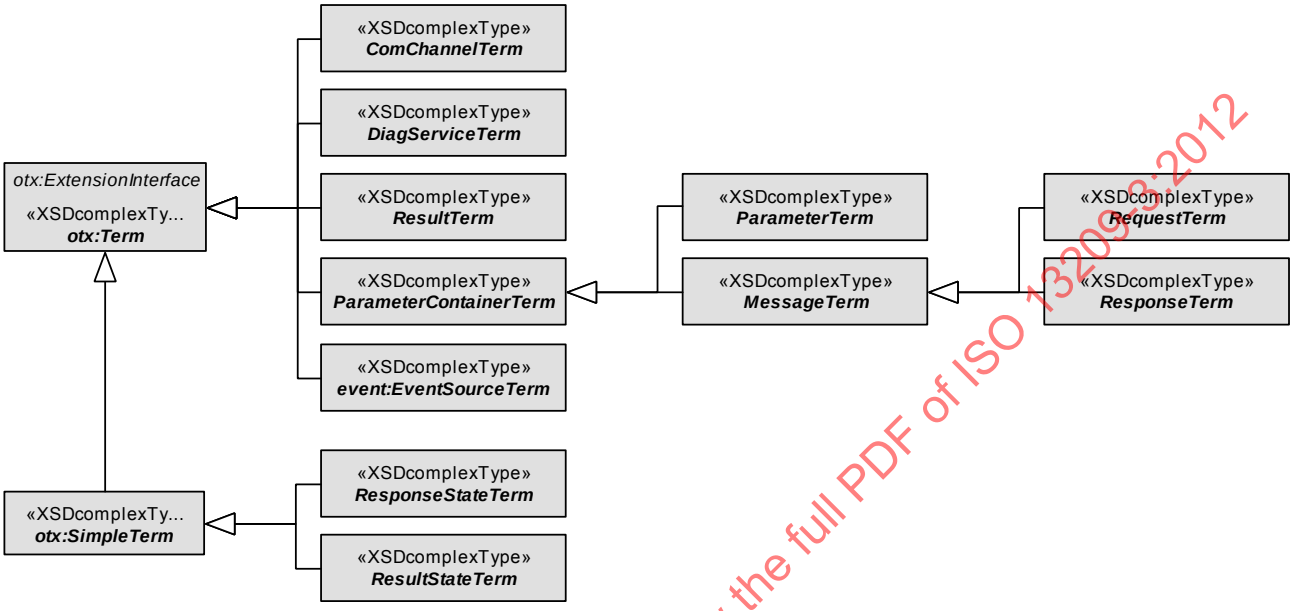

Figure 24 — Data model view: Abstract DiagCom term hierarchy

The abstract types **`ComChannelTerm`**, **`DiagServiceTerm`** and **`ResultTerm`** are the base types for all DiagCom terms returning a **`ComChannel`**, **`DiagService`** or **`Result`** object, respectively.

`ParameterContainerTerms` return handles to any kind of object that can contain parameters. It is an abstract type which is the super class of the **`ParameterTerm`** (**`Parameter`** objects can contain sub-parameters in case of complex parameter structures) and the **`MessageTerm`** which subsumes diagnostic service requests and responses (**`Request`** and **`Response`** objects) which also contain parameters.

Since there are DiagCom terms which return **`event:EventSource`** objects, the Event extension term **`event:EventSourceTerm`** is also listed here. See Clause 8 for details about the Event extension.

Furthermore, the **`otx:SimpleTerm`** types **`ResultStateTerm`** and **`ResponseStateTerm`** are the base types for all DiagCom terms returning a **`ResultState`** or **`ResponseState`** enumeration value.

7.7.2 ComChannel related terms

7.7.2.1 Description

All terms specified in the following clauses relate to the handling of **`ComChannel`** objects.

7.7.2.2 Syntax

Figure 25 shows the syntax of all **`ComChannel`** related terms of the DiagCom extension.

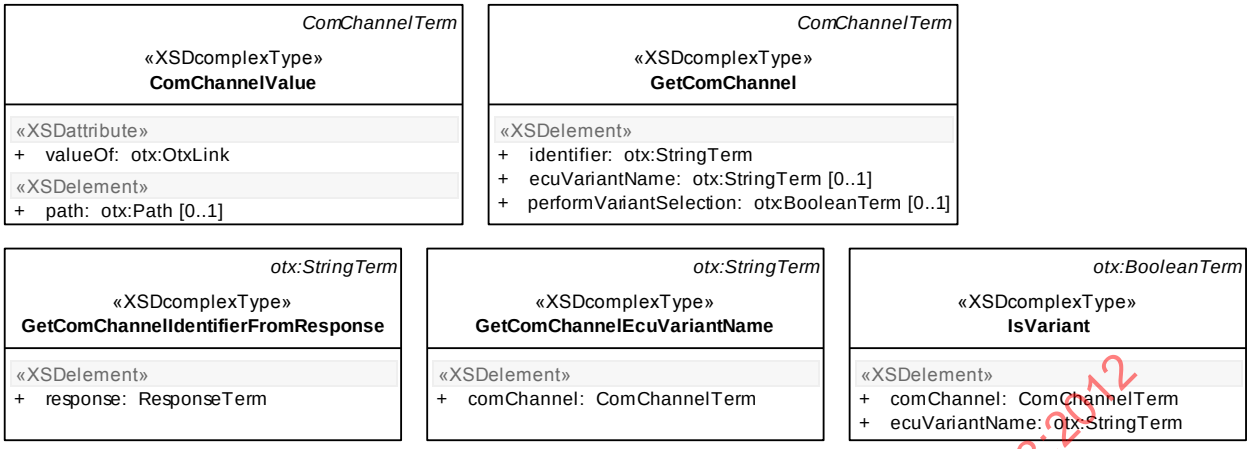

Figure 25 — Data model view: `ComChannel related` terms

7.7.2.3 Semantics

7.7.2.3.1 ComChannelTerm

The abstract type **`ComChannelTerm`** is an **`otx:Term`**. It serves as a base for all concrete terms which return a **`ComChannel`**. It has no special members.

7.7.2.3.2 ComChannelValue

This term returns the **`ComChannel`** stored in a **`ComChannel`** variable. For more information on value-terms and the syntax and semantics of the **`valueOf`** attribute and **`<path>`** element, please refer to Part 2 of ISO 13209.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

Throws:

— **`otx:OutOfBoundsException`**
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).

— **`otx:InvalidReferenceException`**
If the variable value is not valid (no value was assigned to the variable before).

7.7.2.3.3 GetComChannel

This term shall create a communication channel to an ECU. It depends on the implementation of the OTX runtime system when the channel is actually created by the communications layer. There are three possible scenarios:

— The channel is created at the time this term is executed.

— The channel already exists; no additional action is carried out by the execution of this term.

— The channel is created when it is first needed for actual diagnostic communication.

No matter which approach is chosen, the term **`GetComChannel`** shall always return a handle to the same **`ComChannel`** for a given ECU. It is possible to manually control the lifecycle of a **`ComChannel`** object by closing a **`ComChannel`** handle using the **`CloseComChannel`** action (refer to section 7.6.2.3.2). This is up to

the author of a diagnostic sequence, an OTX runtime system is expected to clean up open **ComChannel** handles at the end of a diagnostic session.

The OTX runtime shall perform an ECU variant selection after opening of the channel if the term given by the optional element **<performVariantSelection>** yields **true**. This implies that when the next action on a communication channel is performed, the runtime system has identified the variant of the ECU actually present at runtime and configured the **ComChannel** accordingly. In case both an **<ecuVariantName>** is provided and **<performVariantSelection>** yields **true**, the channel is created to point at the desired ECU variant and variant selection is performed on the link afterwards. The variant identification functionality also exists as a separate action, see 7.6.2.3.1.

GetComChannel is a **ComChannelTerm**. Its members have the following semantics:

— **<identifier> : otx:StringTerm [1]**

This element represents a string identifying the communication channel which shall be created.

NOTE In case a MVCI/ODX system is used the identifier specifies the **SHORT-NAME** of the **MCDLogicalLink** to be used for communication.

— **<ecuVariantName> : otx:StringTerm [0..1]**

This optional element allows an OTX sequence to explicitly specify a particular ECU variant that the **ComChannel** shall be associated with. It is provided in addition to the identifier attribute based on the assumption that the **ComChannel** identifier specifies a connection to a base variant of an ECU, the precise variant of which then can be implicitly or explicitly identified by the diagnostic application (compare the **<performVariantSelection>** element in this clause and the **IdentifyAndSelectVariant** action in 7.6.2.3.1). The **<ecuVariantName>** element can be used to directly create a **ComChannel** to a specific ECU variant without needing to perform the ECU variant identification step.

NOTE In case a MVCI/ODX system is used the **<ecuVariantName>** element specifies the **SHORT-NAME** of the **MCDDbEcuVariant** to be associated with the logical link.

— **<performVariantSelection> : otx:BooleanTerm [0..1]**

This optional element can be used by the OTX author for controlling whether an implicit variant selection shall be done. If **<performVariantSelection>** yields **true** at runtime, the variant selection is done automatically after the **ComChannel** is created. If the element is not set, the default value **false** applies.

This standard can not make assumptions about whether the vehicle communication component used by an OTX runtime supports the concept of ECU variant identification or about the behaviour of the communication component in case it does. The relevant parts of the OTX DiagCom standard are based on the following assumptions:

— A communication channel to an ECU is associated with a data set describing diagnostic behaviour of a specific variant of that ECU.

— The vehicle communication component is able to explicitly perform an ECU variant identification operation on a communication channel to an ECU.

— The required logic and data for performing the variant identification is intrinsic to the vehicle communication component, i.e. there is no additional external information required for the communication component to perform the ECU variant identification.

— After an ECU variant has been identified, the vehicle communication component is able to explicitly associate the communication channel to that ECU with the specific data set for that ECU variant, effectively switching the communication channel from the old variant data set to a new one.

— The **IdentifyAndSelectVariant** action (see 7.6.2.3.1) tells the runtime system to perform the variant identification operation on the provided communication channel and then switch the data set associated with that channel to the one fitting the newly identified variant (if any).

— The **GetComChannel** term (see7.7.2.3.3) tells the runtime system to create a new communication channel, immediately perform the variant identification operation on the new communication channel and then switch the data set associated with that channel to the one fitting the newly identified variant (if any).

NOTE In case an ODX/MVCI system is used, the exact semantics of Variant Identification and Selection are specified by the ISO ODX and MVCI standards.

Throws:

— **UnknownTargetException**
If the **ComChannel** identifier provided by the **<identifer>** element doesn't exist or is invalid.

— **LossOfComException**
If communication to the ECU was interrupted during the variant identification process.

7.7.2.3.4 GetComChannelIdentifierFromResponse

This term accepts a response and returns the communication channel associated with the ECU that sent the response. This term is especially useful for results containing responses from different ECUs (functional addressing, refer to the example in Figure 7).

GetComChannelIdentifierFromResponse is an **otx:StringTerm**. Its members have the following semantics:

— **<response> : ResponseTerm [1]**

This element specifies the response of which the originating ECU shall be returned.

Throws:

— **UnknownComChannelException**
If no **ComChannel** can be found that is associated with the **Response** referenced by the **<response>** element.

7.7.2.3.5 GetComChannelEcuVariantName

The **GetComChannelEcuVariantName** term accepts a handle of a communication channel and returns the name of the ECU variant associated with that channel. For instance, this term can be used to determine the identified ECU variant after having used the **IdentifyAndSelectVariant** action (please refer to 7.6.2.3.1).

NOTE In case a MVCI/ODX system is used the term shall return the **SHORT-NAME** of the **MCDDbEcuVariant** associated with the logical link represented by the **ComChannel**.

GetComChannelIdentifier is an **otx:StringTerm**. Its members have the following semantics:

— **<comChannel> : ComChannelTerm [1]**

The **ComChannelTerm** yields the handle of the communication channel of which the identifier shall be returned.

7.7.2.3.6 IsVariant

The **IsVariant** term is used to compare the name of the ECU variant associated with the communication channel with the given variant name. It accepts a communication channel handle as well as the name of the ECU variant in question. The result is **true** or **false** depending on whether the ECU variant name equals the **ComChannel** identifier or not.

IsVariant is an **otx:BooleanTerm**. Its members have the following semantics:

— **<comChannel> : ComChannelTerm [1]**

The **ComChannelTerm** represents the communication channel which shall be evaluated.

— **<ecuVariantName> : otx:StringTerm [1]**

The **StringTerm** specifies the ECU variant name to be compared with the ECU variant associated with the communication channel.

NOTE In case a MVCI/ODX system is used the variant attribute specifies the **SHORT-NAME** of the **MCDDbEcuVariant** to be queried.

7.7.3 DiagService related terms

7.7.3.1 Description

All terms specified in the following clauses relate to the handling of **DiagService** objects.

7.7.3.2 Syntax

Figure 26 shows the syntax of all **DiagService** related terms of the DiagCom extension.

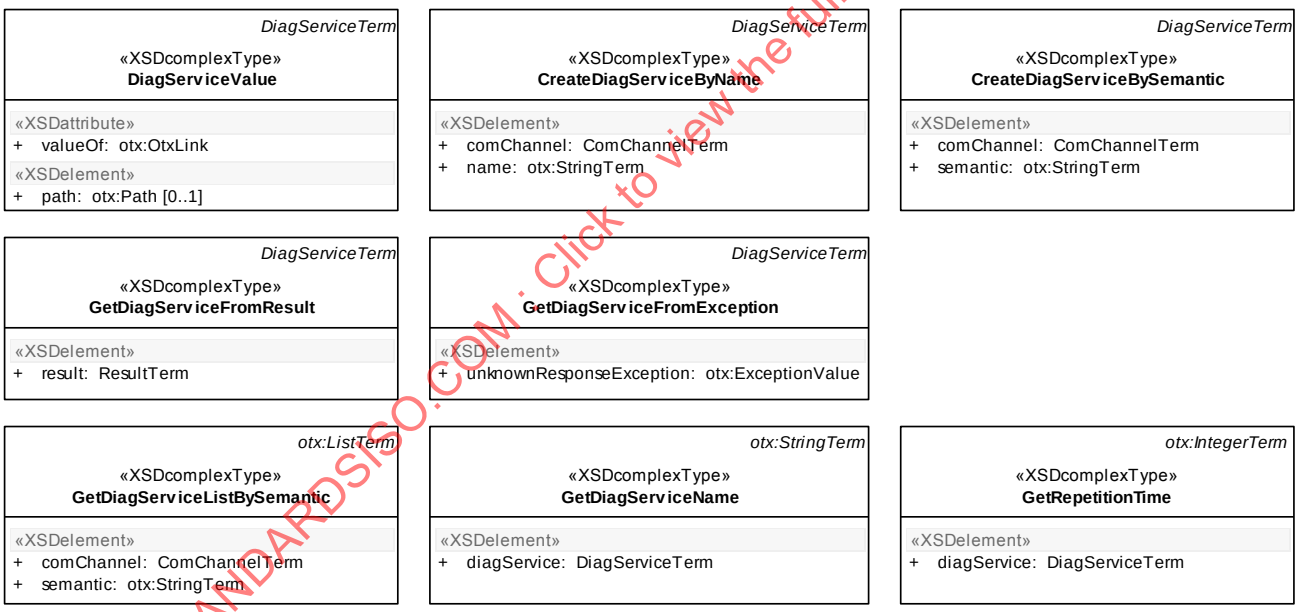

Figure 26 — Data model view: DiagService related terms

7.7.3.3 Semantics

7.7.3.3.1 DiagServiceTerm

The abstract type **DiagServiceTerm** is an **otx:Term**. It serves as a base for all concrete terms which return a **DiagService**. It has no special members.

7.7.3.3.2 DiagServiceValue

This term returns the **DiagService** stored in a **DiagService** variable. For more information on value-terms and the syntax and semantics of the **valueOf** attribute and **<path>** element, please refer to Part 2 of ISO 13209.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

Throws:

— **otx:OutOfBoundsException**
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).

— **otx:InvalidReferenceException**
If the variable value is not valid (no value was assigned to the variable before).

7.7.3.3.3 CreateDiagServiceByName

The **CreateDiagServiceByName** term creates a handle to a diagnostic service that can subsequently be used for parameterizing or executing that service. The diagnostic service to be created is identified by its name. The **CreateDiagServiceByName** term accepts a **ComChannelTerm** and the name of the desired diagnostic service as an **otx:StringTerm**. As a result the term returns a **DiagService** handle.

NOTE In case a MVCI/ODX system is used, the name passed into the **CreateDiagServiceByName** term shall be the **SHORT-NAME** of the associated **MCDDiagComPrimitive** object.

CreateDiagServiceByName is a **DiagServiceTerm**. Its members have the following semantics:

— **<comChannel> : ComChannelTerm [1]**

The **ComChannelTerm** to which the to-be-created diagnostic service belongs to and will be executed on when the **ExecuteDiagService** action is used.

— **<name> : otx:StringTerm [1]**

Name of the to-be-created diagnostic service.

Throws:

— **UnknownTargetException**
If no **DiagService** with the name provided by the **<name>** element exists.

7.7.3.3.4 CreateDiagServiceBySemantic

The **CreateDiagServiceBySemantic** term creates a handle to a diagnostic service that can subsequently be used for configuring or executing that service. The diagnostic service to be created is identified by its semantic attribute. The term accepts a **ComChannelTerm** and the semantic value as an **otx:StringTerm**. As a result the term returns a **DiagService** handle. Please note that using this term can result in an **AmbiguousSemanticException** in case more than one diagnostic service with the desired semantic attribute value exists within this communication channel.

NOTE 1 The ability to assign a semantic value to a diagnostic service or service parameter allows applications working with diagnostic data to access functionality in a manner more abstract than directly pointing to specific names of services/parameters. For example, the diagnostic service to be used for DTC reading could be required to carry the semantic value **"DEFAULT-FAULTREAD"** company wide or even industry wide, no matter what the actual name of the service in a specific data set is. Through the use of semantic attributes, certain elements of a diagnostic data set can become universally identifiable, even though the names of these elements have to conform to user-specific conventions and therefore differ between or even within companies.

NOTE 2 The semantic attribute concept is defined by the ODX/MVCI standards, for example the diagnostic service used for clearing an ECU's fault memory has the semantic attribute **"FAULTCLEAR"**.

CreateDiagServiceBySemantic is a **DiagServiceTerm**. Its members have the following semantics:

— **<comChannel> : ComChannelTerm [1]**

The **ComChannelTerm** to which the to-be-created diagnostic service belongs to and will be executed on when the **ExecuteDiagService** action is used.

— **<semantic> : otx:StringTerm [1]**

The semantic value of the to-be-created diagnostic service.

Throws:

— **AmbiguousSemanticException**
In case there are none or more than one **DiagService** present at the **ComChannel** with the semantic value specified by the **<semantic>** element.

7.7.3.3.5 GetDiagServiceListBySemantic

The term **GetDiagServiceListBySemantic** returns a complete list of all **DiagService** handles which have the same semantic. This is required in case more than one service with the same semantic attribute value exists within the data set associated with the **ComChannel**.

NOTE The ability to assign a semantic value to a diagnostic service or service parameter allows applications working with diagnostic data to access functionality in a manner more abstract than directly pointing to specific names of services/parameters. For example, the diagnostic service to be used for DTC reading could be required to carry the semantic value **"DEFAULT-FAULTREAD"** company wide or even industry wide, no matter what the actual name of the service in a specific data set is. Through the use of semantic attributes, certain elements of a diagnostic data set can become universally identifiable, even though the names of these elements have to conform to user-specific conventions and therefore differ between or even within companies.

GetDiagServiceListBySemantic is a **DiagServiceTerm**. Its members have the following semantics:

— **<comChannel> : ComChannelTerm [1]**

The **ComChannelTerm** that shall be queried for all the services with the given semantic.

— **<semantic> : otx:StringTerm [1]**

The semantic value of the **DiagServices** to be returned.

7.7.3.3.6 GetDiagServiceFromResult

The **GetDiagServiceNameFromResult** term accepts a **ResultTerm** and will return the handle of the **DiagService** the **Result** belongs to.

GetDiagServiceNameFromResult is an **otx:ListTerm**. Its members have the following semantics:

— **<result> : ResultTerm [1]**

Specifies the **Result** for which the containing **DiagService** name shall be retrieved.

Throws:

— **otx:InvalidReferenceException**
If the **DiagService** to which the **Result** belongs to can not be determined.

7.7.3.3.7 GetDiagServiceFromException

The **GetDiagServiceFromException** term accepts an **ExceptionReference** and shall return the handle of the **DiagService** that caused the exception to be thrown. It shall only be used together with exceptions of type **UnknownResponseException** that shall be thrown by the **ExecuteDiagService** action in case the static response mapping does not map a response that has been returned from the vehicle. In that case, it allows the OTX sequence to analyse the result that caused the exception by making it accessible through the **DiagService** object.

GetDiagServiceFromException is a **DiagServiceTerm**. Its members have the following semantics:

— **<unknownResponseException> : otx:ExceptionValue [1]**

Specifies the **Exception** for which the **DiagService** shall be retrieved that caused the exception when executed. It is only allowed to reference exceptions of type **UnknownResponseException**.

Throws:

— **UnknownTargetException**
If the **DiagService** belonging to the exception can not be determined.

— **otx:TypeMismatchException**
If the specified exception is not of type **UnknownResponseException**.

7.7.3.3.8 GetDiagServiceName

The **GetDiagServiceName** term accepts a **DiagService** handle and returns the name of the **DiagService** as a string.

NOTE In case a MVCI/ODX system is used, this term will return the **SHORT-NAME** of the **MCDDiagComPrimitive** object represented by the **DiagService** handle.

GetDiagServiceName is an **otx:StringTerm**. Its members have the following semantics:

— **<diagService> : DiagServiceTerm [1]**

The **DiagService** of which the name shall be returned.

7.7.3.3.9 GetRepetitionTime

The **GetRepetitionTime** term accepts a **DiagService** and returns the currently set repetition cycle time of that diag service in milliseconds (ms).

GetRepetitionTime is an **otx:IntegerTerm**. Its members have the following semantics:

— **<diagService> : DiagServiceTerm [1]**

The **DiagService** of which the current repetition cycle time shall be returned.

7.7.4 Request related terms

7.7.4.1 Description

All terms specified in the following clauses relate to the handling of **`Request`** objects.

7.7.4.2 Syntax

Figure 27 shows the syntax of all **`Request`** related terms of the DiagCom extension.

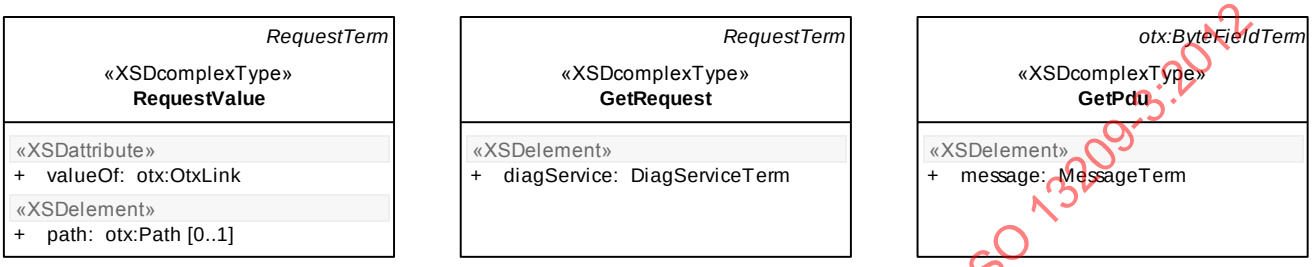

Figure 27 — Data model view: `Request` related terms

7.7.4.3 Semantics

7.7.4.3.1 RequestTerm

The abstract type **`RequestTerm`** is a **`MessageTerm`**. It serves as a base for all concrete terms which return a **`Request`**. It has no special members.

7.7.4.3.2 RequestValue

This term returns the **`Request`** stored in a **`Request`** variable. For more information on value-terms and the syntax and semantics of the **`valueOf`** attribute and **`<path>`** element, please refer to Part 2 of ISO 13209.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

Throws:

— **`otx:OutOfBoundsException`**
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).

— **`otx:InvalidReferenceException`**
If the variable value is not valid (no value was assigned to the variable before).

7.7.4.3.3 GetRequest

The **`GetRequest`** term shall return the **`Request`** belonging to a diagnostic service. It accepts a diagnostic service handle.

`GetRequest` is a **`RequestTerm`**. Its members have the following semantics:

— **`<diagService> : DiagServiceTerm [1]`**

The term shall yield a handle to the **`DiagService`** that the **`Request`** belongs to.

7.7.4.3.4 GetPdu

The **GetPdu** term shall return the raw byte stream data represented by a **Request** or a **Response** as seen on the physical layer. The **GetPdu** term is derived from **ByteFieldTerm**. A possible use case for retrieving raw communication data could be to implement bus tracing functionality. The corresponding opposite operation to the **GetPdu** term is provided by the **SetPdu** action (see 7.6.4.3.8).

A PDU as understood by the DiagCom extension comprises the complete payload of a message including the service identifier and any other request parameters. It does not include header or checksum bytes from underlying protocol layers.

GetPdu is an **otx:ByteFieldTerm**. Its members have the following semantics:

— **<message> : MessageTerm [1]**

The **Message** (e.g. **Request** or **Response**) which is to be returned in **ByteField** form.

7.7.5 Result related terms

7.7.5.1 Description

All terms specified in the following clauses relate to the handling of **Result** objects.

The result related terms are designed based on the assumption that a diagnostic communication component as used by an OTX runtime has the capability to buffer results it receives from ECUs. Especially for dealing with system behaviour as illustrated in Figure 6 (one ECU returning multiple results for one diagnostic request) a communication system requires a buffering concept for ECU results. The following assumptions are made in the OTX DiagCom context regarding the result buffer:

— The result buffer is owned and managed by the vehicle communication component and is outside the scope of an OTX runtime.

— Every **DiagService** object has an associated result buffer which contains any results that were received as a reaction to an **ExecuteDiagService** action.

— This result buffer is of finite size, i.e. a loop buffer that will wrap around after a number of results have been received by the vehicle communication component.

— The DiagCom term **GetFirstResult** (see 7.7.5.3.3) only fetches the first (in time) result out of the communication component's result buffer, but does not modify that buffer.

— The DiagCom term **GetAllResults** (see 7.7.5.3.4) fetches all results present at the time of the call out of the communication component's result buffer, but does not modify that buffer. The list of results that is returned to OTX will be in ascending order from first (oldest) to last (most recent) result.

— The DiagCom action **GetAllResultsAndClear** (see 7.6.4.3.6) fetches all results present at the time of the call out of the communication component's result buffer and tells the communication component to clear the buffer afterwards. The list of results that is returned to OTX will be in ascending order from first (oldest) to last (most recent) result.

7.7.5.2 Syntax

Figure 28 shows the syntax of all **Result** related terms of the DiagCom extension.

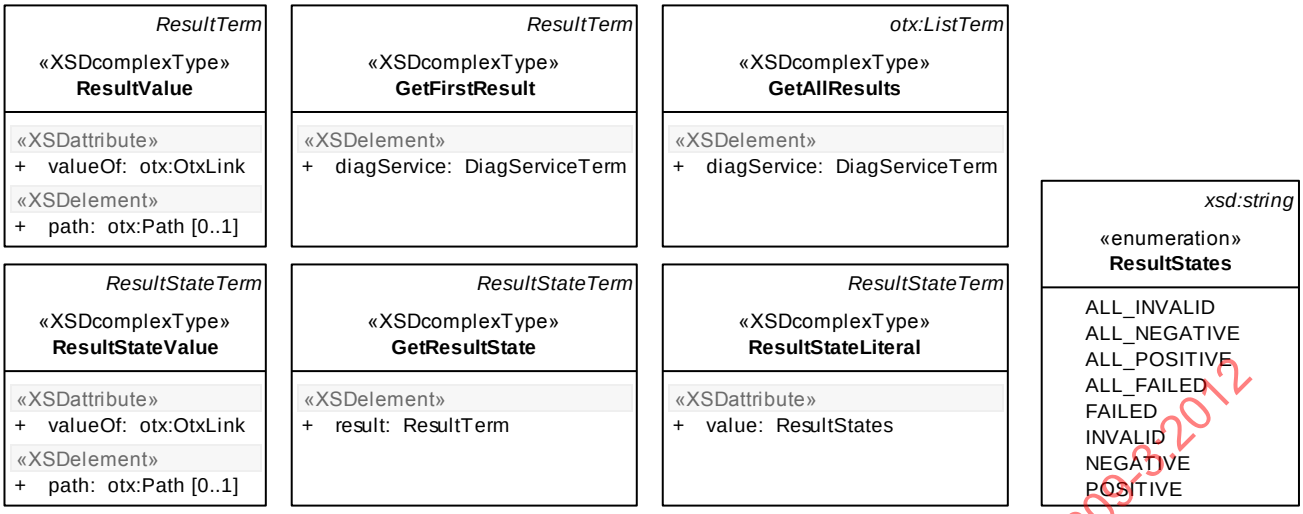

Figure 28 — Data model view: `Result` related terms

7.7.5.3 Semantics

7.7.5.3.1 ResultTerm

The abstract type **`ResultTerm`** is an **`otx:Term`**. It serves as a base for all concrete terms which return a **`Result`**. It has no special members.

7.7.5.3.2 ResultValue

This term returns the **`Result`** stored in a **`Result`** variable. For more information on value-terms and the syntax and semantics of the **`valueOf`** attribute and **`<path>`** element, please refer to Part 2 of ISO 13209.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

Throws:

— **`otx:OutOfBoundsException`**
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).

— **`otx:InvalidReferenceException`**
If the variable value is not valid (no value was assigned to the variable before).

7.7.5.3.3 GetFirstResult

The **`GetFirstResult`** term returns the first result of a service execution, irrespective of whether there exists more than one result. The term accepts a **`DiagServiceTerm`** argument and returns a **`Result`** object.

`GetFirstResult` is a **`ResultTerm`**. Its members have the following semantics:

— **`<diagService> : DiagServiceReference [1]`**

Represents the **`DiagService`** object of which the first **`Result`** shall be returned.

Throws:

— **`otx:OutOfBoundsException`**
If there exists no **`Result`** object in the **`DiagService`** object.

7.7.5.3.4 GetAllResults

The **GetAllResults** returns all available results of a diagnostic service as a **ListTerm**. The list contains **Result** objects. In comparison to the action **GetAllResultsAndClear** referenced in DiagCom actions specified in 7.6, this term only reads **Result** entries and does not delete the buffer containing the results. Possible use case is the monitoring of results without changing the state of the **DiagService**. **GetAllResults** is derived from **ListTerm**.

GetAllResults is an **otx:ListTerm**. Its members have the following semantics:

— **<diagService> : DiagServiceTerm [1]**

Represents the **DiagService** of which the **Results** shall be returned.

7.7.5.3.5 ResultStateTerm

The abstract type **ResultStateTerm** is an **otx:SimpleTerm**. It serves as a base for all concrete terms which return a **ResultState** value (see 7.3.3.10). It has no special members.

7.7.5.3.6 ResultStateValue

This term returns the **ResultState** stored in a **ResultState** variable. For more information on value-terms and the syntax and semantics of the **valueOf** attribute and **<path>** element, please refer to Part 2 of ISO 13209.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

Throws:

— **otx:OutOfBoundsException**
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).

7.7.5.3.7 ResultStateLiteral

This term shall return a **ResultState** value (see 7.3.3.10) from a hard-coded literal.

ResultStateLiteral is a **ResultStateTerm**. Its members have the following semantics:

— **value : ResultStates={ALL_FAILED|ALL_INVALID|ALL_NEGATIVE|ALL_POSITIVE|FAILED|INVALID|NEGATIVE|POSITIVE} [1]**

This attribute shall contain one of the values defined in the **ResultStates** enumeration.

7.7.5.3.8 GetResultState

This term shall retrieve the state of a **Result** (i.e. whether the ECU(s) answered at all, correctly, positively or negatively). Allowed result state values are specified by the **ResultState** data type as defined in 7.3.3.10. This also corresponds to the **<resultState>** element of the **ExecuteDiagService** action, see 7.6.4.3.2.

GetResultState is a **ResultStateTerm**. Its members have the following semantics:

— **<result> : ResultTerm [1]**

The **Result** whose state shall be returned.

7.7.6 Response related terms

7.7.6.1 Description

All terms specified in the following clauses relate to the handling of **`Response`** objects.

7.7.6.2 Syntax

Figure 29 shows the syntax of all **`Response`** related terms of the DiagCom extension.

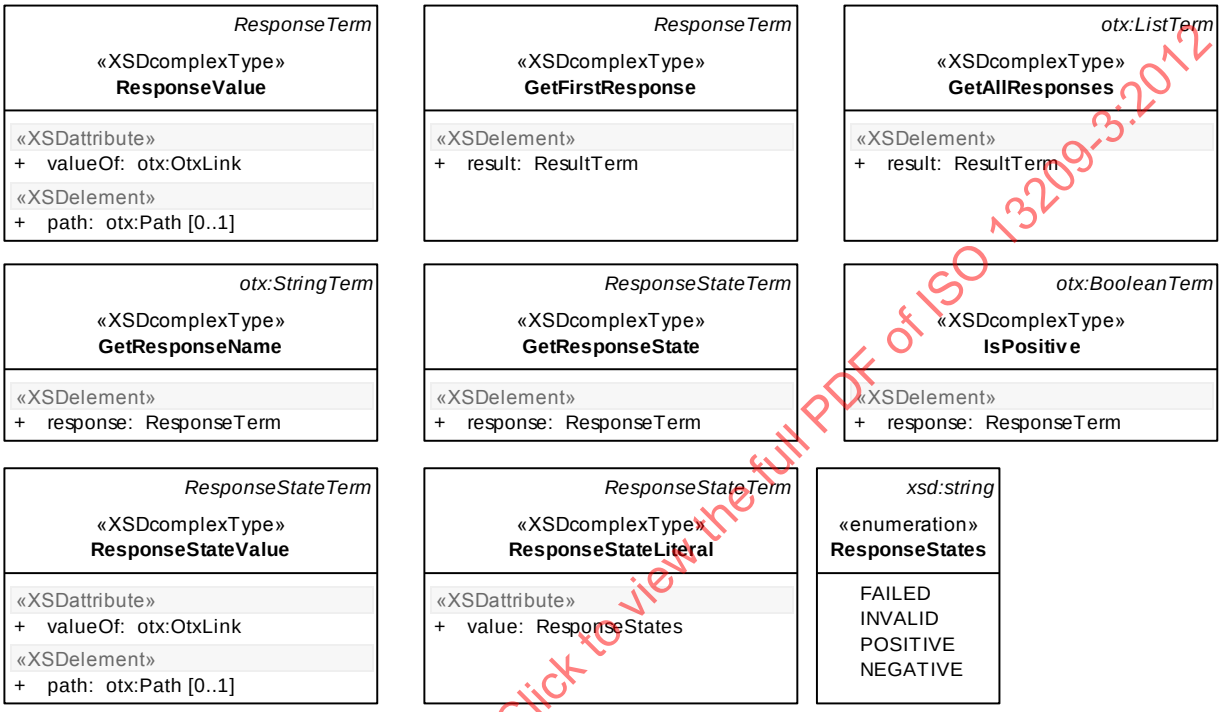

Figure 29 — Data model view: `Response` related terms

7.7.6.3 Semantics

7.7.6.3.1 ResponseTerm

The abstract type **`ResponseTerm`** is a **`MessageTerm`**. It serves as a base for all concrete terms which return a **`Response`**. It has no special members.

7.7.6.3.2 ResponseValue

This term returns the **`Response`** stored in a **`Response`** variable. For more information on value-terms and the syntax and semantics of the **`valueOf`** attribute and **`<path>`** element, please refer to Part 2 of ISO 13209.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

Throws:

— **`otx:OutOfBoundsException`**
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).

— **otx:InvalidReferenceException**
If the variable value is not valid (no value was assigned to the variable before).

7.7.6.3.3 GetFirstResponse

The **GetFirstResponse** term is used to retrieve the first **Response** of a **Result** handle. In case there is more than one **Response** available in a **Result**, only the first **Response** will be returned.

GetFirstResponse is a **ResponseTerm**. Its members have the following semantics:

— **<result> : ResultTerm [1]**

The **Result** whose first response shall be returned.

7.7.6.3.4 GetAllResponses

The **GetAllResponses** term returns a list of all responses that are available for that **Result**. It accepts a **ResultTerm**. For example in case of a functionally addressed diagnostic service, this term can be used to retrieve all ECU responses that were received in response to the functional service execution. Normally there will only be one response per diagnostic service (standard physical addressing), in which case the term **GetFirstResponse** shall be used.

GetAllResponses is an **otx:ListTerm**. Its members have the following semantics:

— **<result> : ResultTerm [1]**

The **Result** whose responses shall be returned.

7.7.6.3.5 GetResponseName

This term shall retrieve the name of a **Response**. For example it can be used to determine whether a **Response** is positive or negative by comparing the response name with preset response names valid for the vehicle communication component.

NOTE In case a MVCI/ODX system is used the **GetResponseName** term returns the **SHORT-NAME** of the associated **MCDResponse** object.

GetResponseName is an **otx:StringTerm**. Its members have the following semantics:

— **<response> : ResponseTerm [1]**

The **Response** whose name shall be returned.

7.7.6.3.6 ResponseStateTerm

The abstract type **ResponseStateTerm** is an **otx:SimpleTerm**. It serves as a base for all concrete terms which return a **ResponseState** value (see 7.3.3.11). It has no special members.

7.7.6.3.7 ResponseStateValue

This term returns the **ResponseState** stored in a **ResponseState** variable. For more information on value-terms and the syntax and semantics of the **valueOf** attribute and **<path>** element, please refer to Part 2 of ISO 13209.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

Throws:

— **otx:OutOfBoundsException**
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).

7.7.6.3.8 ResponseStateLiteral

This term shall return a **ResponseState** value (see 7.3.3.11) from a hard-coded literal.

ResponseStateLiteral is a **ResponseStateTerm**. Its members have the following semantics:

— **value : ResponseStates={FAILED|INVALID|NEGATIVE|POSITIVE} [1]**

This attribute shall contain one of the values defined in the **ResponseStates** enumeration.

7.7.6.3.9 GetResponseState

This term shall retrieve the state of a **Response**. Allowed response state values are specified by the **ResponseState** data type as defined in 7.3.3.11.

GetResponseState is a **ResponseStateTerm**. Its members have the following semantics:

— **<result> : ResponseTerm [1]**

The **Response** whose state shall be returned.

7.7.6.3.10 IsPositive

The **IsPositive** term shall check whether a response is positive. It accepts a **ResponseTerm**. For details on response states, please refer to the **ResponseState** data type (see 7.3.3.11).

IsPositive is an **otx:BooleanTerm**. Its members have the following semantics:

— **<response> : ResponseTerm [1]**

The **Response** which shall be checked for being positive.

7.7.7 Parameter related terms

7.7.7.1 Description

All terms specified in the following clauses relate to the handling of **Parameter** objects.

7.7.7.2 Syntax

Figure 30 shows the syntax of all **Parameter** related terms of the DiagCom extension.

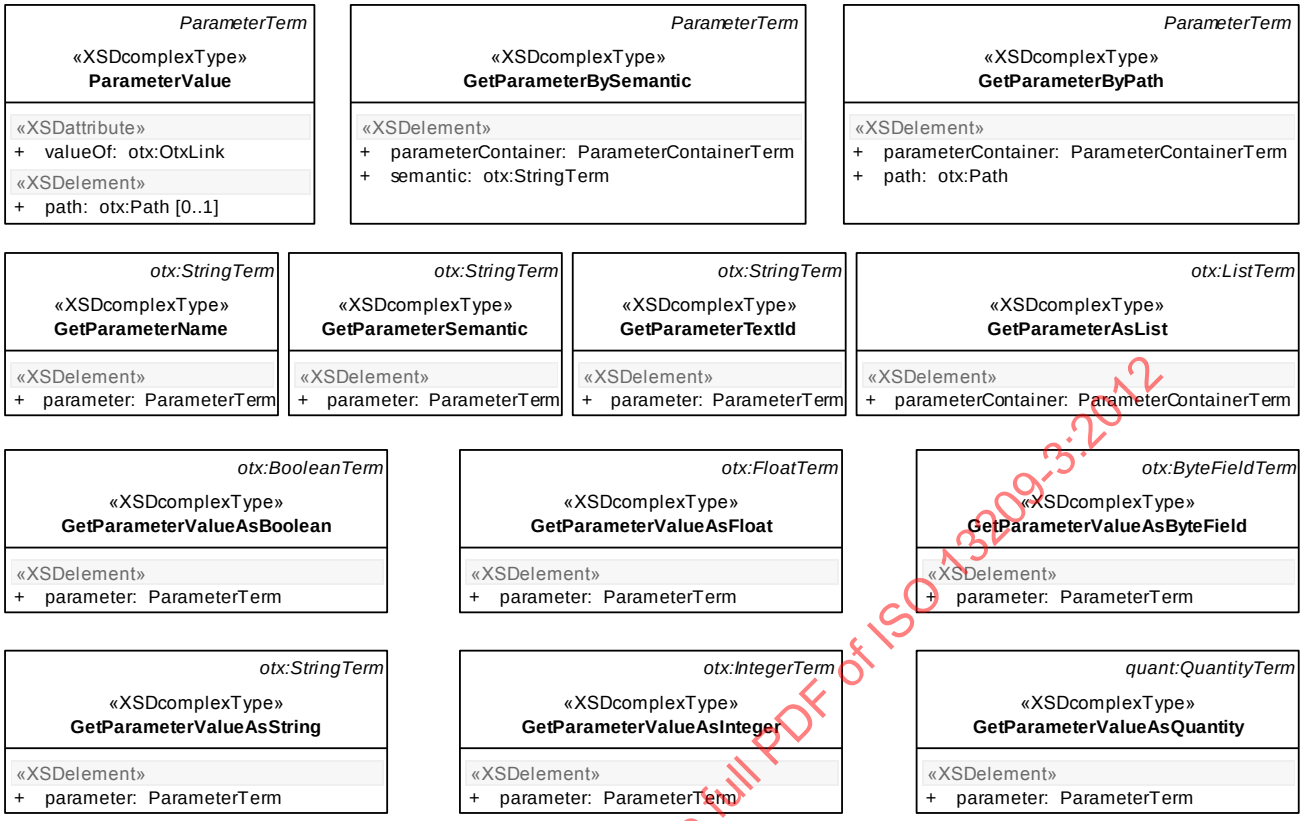

Figure 30 — Data model view: `Parameter` related terms

7.7.7.3 Semantics

7.7.7.3.1 ParameterTerm

The abstract type **`ParameterTerm`** is a **`ParameterContainerTerm`**. It serves as a base for all concrete terms which return a **`Parameter`**. It has no special members.

7.7.7.3.2 ParameterValue

This term returns the **`Parameter`** stored in a **`Parameter`** variable. For more information on value-terms and the syntax and semantics of the **`valueOf`** attribute and **`<path>`** element, please refer to Part 2 of ISO 13209.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

Throws:

— **`otx:OutOfBoundsException`**
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).

— **`otx:InvalidReferenceException`**
If the variable value is not valid (no value was assigned to the variable before).

7.7.7.3.3 GetParameterBySemantic

The **`GetParameterBySemantic`** term accepts a **`ParameterContainerTerm`** and the semantic value of the parameter to be retrieved. It can return simple type or complex type parameters, depending on the

parameter structure of the diagnostic service definition of the underlying communication system and the specific parameter that is being retrieved.

NOTE 1 The ability to assign a semantic value to a diagnostic service or service parameter allows applications working with diagnostic data to access functionality in a manner more abstract than directly pointing to specific names of services/parameters. For example, the diagnostic service to be used for DTC reading could be required to carry the semantic value **"DEFAULT-FAULTREAD"** company wide or even industry wide, no matter what the actual name of the service in a specific data set is. Through the use of semantic attributes, certain elements of a diagnostic data set can become universally identifiable, even though the names of these elements have to conform to user-specific conventions and therefore differ between or even within companies.

NOTE 2 In case a MVCI/ODX system is used the semantic value is equivalent to the semantic attribute of the corresponding **MCDParameter** object.

GetParameterBySemantic is a **ParameterTerm**. Its members have the following semantics:

— **<parameterContainer> : ParameterContainerTerm [1]**

The container from which the **Parameter** shall be retrieved.

— **<name> : otx:StringTerm [1]**

The semantic attribute of the **Parameter** which shall be returned.

Throws:

— **AmbiguousSemanticException**
If there are none or more than one parameters present in the **ParameterContainerTerm** with the semantic value specified by the **<semantic>** element.

7.7.7.3.4 GetParameterByPath

The **GetParameterByPath** term accepts a **ParameterContainerTerm** and a **Path** to the parameter to be retrieved. It returns the parameter that is pointed to within the parameter container by the **Path** definition. It can return simple type or complex type parameters, depending on the parameter structure of the diagnostic service definition of the underlying communication system and the specific parameter that is being retrieved. This term operates on the assumption that parameter names are unique within one hierarchy level of the parameter structure. An example for retrieving a **Parameter** by **otx:Path** is shown in Figure 22.

GetParameterByPath is a **ParameterTerm**. Its members have the following semantics:

— **<parameterContainer> : ParameterContainerTerm [1]**

The container from which the **Parameter** shall be retrieved.

— **<path> : otx:Path [1]**

The path element specifies the path to the desired parameter.

Throws:

— **UnknownTargetException**
If the **Parameter** object referenced by the **<path>** element doesn't exist or is invalid.

7.7.7.3.5 GetParameterName

The **GetParameterName** term accepts a **ParameterTerm** and returns the name of the parameter.

NOTE In case a MVCI/ODX system is used it returns the **SHORT-NAME** of the corresponding **MCDParameter** object.

GetParameterName is an **otx:StringTerm**. Its members have the following semantics:

— **<parameter> : ParameterTerm [1]**

The **Parameter** whose name shall be returned.

7.7.7.3.6 GetParameterSemantic

The **GetParameterSemantic** term accepts a **ParameterTerm** and returns the semantic attribute value of the **Parameter**.

NOTE In case a MVCI/ODX system is used it returns the semantic attribute of the corresponding **MCDParameter** object.

GetParameterSemantic is an **otx:StringTerm**. Its members have the following semantics:

— **<parameter> : ParameterTerm [1]**

The **ParameterTerm** whose semantic attribute shall be returned.

7.7.7.3.7 GetParameterTextId

The **GetParameterTextId** term accepts a **ParameterTerm** and returns the text id of the **Parameter**.

The actual functionality of this term and format of returned information depends on the communication backend that is used by the OTX runtime and is not defined by this standard.

NOTE In case a MVCI/ODX system is used it returns the **LongNameId** attribute of the corresponding **MCDDbObject** object. In case the parameter represents a DTC, the **DiagTroubleCodeTextID** of the **MCDDbDiagTroubleCode** is returned.

GetParameterTextId is an **otx:StringTerm**. Its members have the following semantics:

— **<parameter> : ParameterTerm [1]**

The **Parameter** whose text id attribute shall be returned.

7.7.7.3.8 GetParameterAsList

The **GetParameterAsList** term accepts a **ParameterContainerTerm** and returns an **otx:List** of **Parameter** handles corresponding to the contents of the passed in parameter container object. This term is used in case a parameter of a diagnostic service contains a set of identically typed parameters, i.e. an array or a list of parameters. Please refer to Figure 5 which shows an example of a complex list-type parameter.

GetParameterAsList is an **otx:ListTerm**. Its members have the following semantics:

— **<parameterContainer> : ParameterContainerTerm [1]**

The **ParameterContainer** whose value shall be returned as an **otx:List**.

Throws:

— **otx:TypeMismatchException**
If the specified **ParameterContainer** is not of a list or array type.

7.7.7.3.9 GetParameterValueAsBoolean

The **GetParameterValueAsBoolean** term accepts a **ParameterTerm** and returns the actual value of the parameter as a Boolean quantity.

GetParameterValueAsBoolean is an **otx:BooleanTerm**. Its members have the following semantics:

— **<parameter> : ParameterTerm [1]**

The **Parameter** whose value shall be returned as a Boolean.

Throws:

— **otx:TypeMismatchException**
If the specified **Parameter** is not of boolean type.

7.7.7.3.10 GetParameterValueAsString

The **GetParameterValueAsString** term accepts a **ParameterTerm** and returns the actual value of the parameter as a string.

GetParameterValueAsString is an **otx:StringTerm**. Its members have the following semantics:

— **<parameter> : ParameterTerm [1]**

The **Parameter** whose value shall be returned as a string.

Throws:

— **otx:TypeMismatchException**
If the specified **Parameter** is not of string type.

7.7.7.3.11 GetParameterValueAsInteger

The **GetParameterValueAsInteger** term accepts a **ParameterTerm** and returns the actual value of the parameter as an integer.

GetParameterValueAsInteger is an **otx:IntegerTerm**. Its members have the following semantics:

— **<parameter> : ParameterTerm [1]**

The **Parameter** whose value shall be returned as an integer.

Throws:

— **otx:TypeMismatchException**
If the specified **Parameter** is not of integer type.

7.7.7.3.12 GetParameterValueAsFloat

The **GetParameterValueAsFloat** term accepts a **ParameterTerm** and returns the actual value of the parameter as a float.

GetParameterValueAsFloat is an **otx:FloatTerm**. Its members have the following semantics:

— **<parameter> : ParameterTerm [1]**

The **Parameter** whose value shall be returned as an integer.

Throws:

— **otx:TypeMismatchException**
If the specified **Parameter** is not of float type.

7.7.7.3.13 GetParameterValueAsByteField

The **GetParameterValueAsByteField** term accepts a **ParameterTerm** and returns the actual value of the parameter as a bytefield.

GetParameterValueAsByteField is an **otx:ByteFieldTerm** with the following member semantics:

— **<parameter> : ParameterTerm [1]**

The **Parameter** whose value shall be returned as a bytefield.

Throws:

— **otx:TypeMismatchException**
If the specified **Parameter** is not of bytefield type.

7.7.7.3.14 GetParameterValueAsQuantity

The **GetParameterValueAsQuantity** term accepts a **ParameterTerm** and returns the actual value of the parameter as a quantity.

GetParameterValueAsQuantity is a **quant:QuantityTerm**. Its members have the following semantics:

— **<parameter> : ParameterTerm [1]**

The **Parameter** whose value shall be returned as a quantity.

Throws:

— **otx:TypeMismatchException**
If the specified **Parameter** is not of quantity type.

7.7.8 ComParam related terms

7.7.8.1 Description

All terms specified in the following clauses relate to the handling of communication parameters.

7.7.8.2 Syntax

Figure 31 shows the syntax of all ComParam related terms of the DiagCom extension.

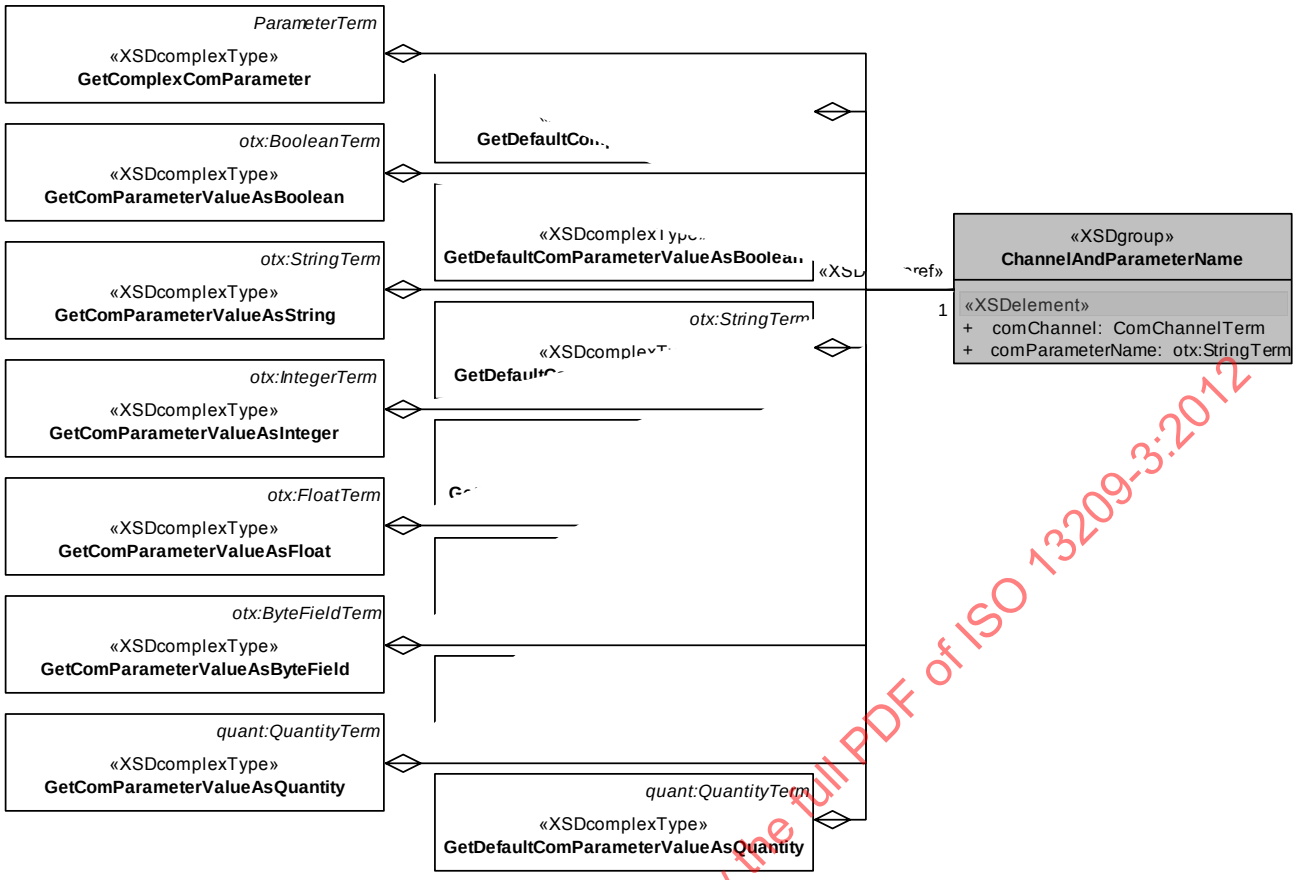

Figure 31 — Data model view: ComParam related terms

7.7.8.3 Semantics

7.7.8.3.1 ChannelAndParameterName group

The following properties are part of all of the following terms and are therefore defined as a separate group.

The members of the **`ChannelAndParameterName`** group have the following semantics:

— **`<comChannel> : ComChannelTerm [1]`**

The **`ComChannelTerm`** specifies the **`ComChannel`** which shall be queried.

— **`<comParameterName> : otx:StringTerm [1]`**

The **`otx:StringTerm`** specifies the name of a communication parameter.

Throws:

— **`UnknownTargetException`**
If there exists no communication parameter with the name provided by **`<comParameterName>`**.

— **`otx:TypeMismatchException`**
If the specified parameter is not of the correct type

7.7.8.3.2 GetDefaultComplexComParameter

The **GetDefaultComplexComParameter** term comprises the **ChannelAndParameterName** attribute group and shall return the default value of a complex communication parameter (e.g. list and struct parameter types).

GetDefaultComplexComParameter is a **ParameterTerm**. Its members are described by the group **ChannelAndParameterName**, as specified above.

7.7.8.3.3 GetComplexComParameter

The **GetComplexComParameter** term comprises the **ChannelAndParameterName** attribute group and shall return the current value of a complex communication parameter (e.g. list and struct parameter types). If the communication parameter has not been previously modified by the **SetComplexComParameter** action (defined in 7.6.3.3.2), the default parameter value shall be returned.

GetComplexComParameter is a **ParameterTerm**. Its members are described by the group **ChannelAndParameterName**, as specified above.

7.7.8.3.4 GetComParameterValueAsBoolean

The **GetComParameterAsBoolean** term comprises the **ChannelAndParameterName** attribute group and shall return the current value of a Boolean communication parameter. If the communication parameter has not been previously modified by the **SetComParameter** action (defined in 7.6.3.3.1), the default parameter value shall be returned.

GetComParameterAsBooleanQuantity is an **otx:BooleanTerm**. Its members are described by the group **ChannelAndParameterName**, as specified above.

7.7.8.3.5 GetComParameterValueAsString

The **GetComParameterAsString** term comprises the **ChannelAndParameterName** attribute group and shall return the current value of a string type communication parameter. If the communication parameter has not been previously modified by the **SetComParameter** action (defined in 7.6.3.3.1), the default parameter value shall be returned.

GetComParameterAsString is an **otx:StringTerm.** Its members are described by the group **ChannelAndParameterName**, as specified above.

7.7.8.3.6 GetComParameterValueAsInteger

The **GetComParameterAsInteger** term comprises the **ChannelAndParameterName** attribute group and shall return the current value of an integer type communication parameter. If the communication parameter has not been previously modified by the **SetComParameter** action (defined in 7.6.3.3.1), the default parameter value shall be returned.

GetComParameterAsInteger is an **otx:IntegerTerm.** Its members are described by the group **ChannelAndParameterName**, as specified above.

7.7.8.3.7 GetComParameterValueAsFloat

The **GetComParameterAsFloat** term comprises the **ChannelAndParameterName** attribute group and shall return the current value of a float type communication parameter. If the communication parameter has not been previously modified by the **SetComParameter** action (defined in 7.6.3.3.1), the default parameter value shall be returned.

GetComParameterAsFloat is an **otx:FloatTerm.** Its members are described by the group **ChannelAndParameterName**, as specified above.

7.7.8.3.8 GetComParameterValueAsByteField

The **GetComParameterAsByteField** term comprises the **ChannelAndParameterName** attribute group and shall return the current value of a bytefield type communication parameter. If the communication parameter has not been previously modified by the **SetComParameter** action (defined in 7.6.3.3.1), the default parameter value shall be returned.

GetComParameterAsByteField is an **otx:ByteFieldTerm**. Its members are described by the group **ChannelAndParameterName**, as specified above.

7.7.8.3.9 GetComParameterValueAsQuantity

The **GetComParameterAsQuantity** term comprises the **ChannelAndParameterName** attribute group and shall return the current value of a quantity type communication parameter. If the communication parameter has not been previously modified by the **SetComParameter** action (defined in 7.6.3.3.1), the default parameter value shall be returned.

GetComParameterAsQuantity is a **quant:QuantityTerm**. Its members are described by the group **ChannelAndParameterName**, as specified above.

7.7.8.3.10 GetDefaultComParameterValueAsBoolean

The **GetDefaultComParameterAsBoolean** term comprises the **ChannelAndParameterName** attribute group and shall return the default value of a Boolean type communication parameter.

GetDefaultComParameterAsBoolean is an **otx:BooleanTerm.** Its members are described by the group **ChannelAndParameterName**, as specified above.

7.7.8.3.11 GetDefaultComParameterValueAsString

The **GetDefaultComParameterAsString** term comprises the **ChannelAndParameterName** attribute group and shall return the default value of a string type communication parameter.

GetDefaultComParameterAsString is an **otx:StringTerm**. Its members are described by the group **ChannelAndParameterName**, as specified above.

7.7.8.3.12 GetDefaultComParameterValueAsInteger

The **GetDefaultComParameterAsInteger** term comprises the **ChannelAndParameterName** attribute group and shall return the default value of an integer type communication parameter.

GetDefaultComParameterAsInteger is an **otx:IntegerTerm**. Its members are described by the group **ChannelAndParameterName**, as specified above.

7.7.8.3.13 GetDefaultComParameterValueAsFloat

The **GetDefaultComParameterAsFloat** term comprises the **ChannelAndParameterName** attribute group and shall return the default value of a float type communication parameter.

GetDefaultComParameterAsFloat is an **otx:FloatTerm**. Its members are described by the group **ChannelAndParameterName**, as specified above.

7.7.8.3.14 GetDefaultComParameterValueAsByteField

The **GetDefaultComParameterAsByteField** term comprises the **ChannelAndParameterName** attribute group and shall return the default value of a bytefield type communication parameter.

GetDefaultComParameterAsByteFieldQuantity is an **otx:ByteFieldTerm**. Its members are described by the group **ChannelAndParameterName**, as specified above.

7.7.8.3.15 GetDefaultComParameterValueAsQuantity

The **GetDefaultComParameterAsQuantity** term comprises the **ChannelAndParameterName** attribute group and shall return the default value of a communication parameter.

GetDefaultComParameterAsQuantity is an **quant:QuantityTerm**. Its members are described by the group **ChannelAndParameterName**, as specified above.

7.7.9 Event related terms

7.7.9.1 Description

All terms specified in the following clauses relate to event handling. For further details about the OTX EventHandling extension please refer to Clause 8.

7.7.9.2 Syntax

Figure 32 shows the syntax of all event related terms of the DiagCom extension.

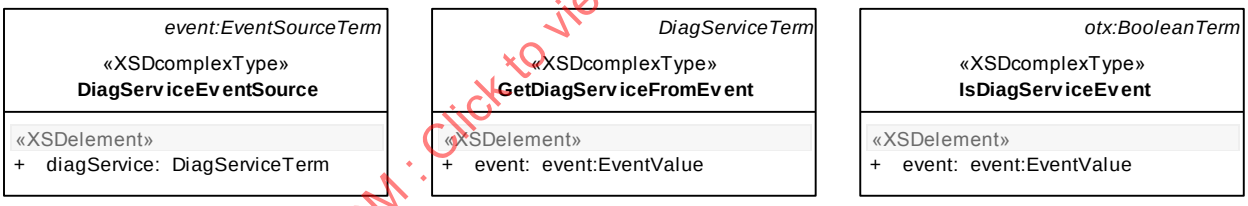

Figure 32 — Data model view: Event related terms

7.7.9.3 Semantics

7.7.9.3.1 DiagServiceEventSource

The **DiagServiceEventSource** term accepts a **DiagService** object that is to be made an event source. This term enables an OTX sequence to use a **DiagService** as a source for events in the context of the OTX EventHandling extension (please refer to Clause 8). A **DiagService** shall trigger an event every time a new **Result** has arrived (please compare Figure 6). The **DiagServiceEventSource** term is the complementary functionality to the asynchronous execution feature of the **ExecuteDiagService** action: when **ExecuteDiagService** is used with **<executeAsync>** set to true, the only way to be notified of available results for the executed diagnostic service is to use it as an event source through the **DiagServiceEventSource** term (the type of event can then be retrieved by using the **IsDiagServiceEvent** term as specified below.

DiagServiceEventSource is an **event:EventSource**. Its members have the following semantics:

— **<diagService> : DiagServiceTerm [1]**

  Represents the **DiagService** that shall be connected to the event source.

7.7.9.3.2 IsDiagServiceEvent

The **IsDiagServiceEvent** term accepts an **EventValue** term yielding an **Event** object that has been raised by the OTX runtime, as a result of declaring a **DiagService** object as an event source by using the term **DiagServiceEventSource**. The term shall return **true** if and only if the **Event** originates from a **DiagServiceEventSource** term.

IsDiagServiceEvent is an **otx:BooleanTerm**. Its members have the following semantics:

— **<event> : event:EventValue [1]**

Represents the **Event** whose type shall be tested.

7.7.9.3.3 GetDiagServiceFromEvent

The **GetDiagServiceFromEvent** term accepts an **EventValue** term yielding an **Event** object that has been raised by the OTX runtime, as a result of declaring a **DiagService** object as an event source by using the term **DiagServiceEventSource**. It returns a handle to the **DiagService** object that caused the event (i.e. because a new ECU **Result** has been received after the **DiagService** has been executed, please refer to 7.6.4.3.1 and 7.7.9.3.1). By using this term, an OTX sequence can wait for an **Event** raised by a **DiagService** receiving a new **Result** and then evaluate the **ResultResponse** structure of that **DiagService**.

GetDiagServiceFromEvent is a **DiagServiceTerm**. Its members have the following semantics:

— **<event> : event:EventValue [1]**

Represents the event that was raised by the **DiagService** that shall be retrieved.

Throws:

— **otx:TypeMismatchException**
If the specified event has not been raised by a **DiagServiceEventSource**.

8 OTX DiagDataBrowsing extension

8.1 Introduction

The OTX DiagDataBrowsing extension provides a set of terms for reading static information associated with communication channels, diagnostic services and request- or response-parameters. The data is static insofar that it originates from a diagnostic vehicle information database; this is unlike dynamic data which is e.g. read from an ECU.

The extension is designed for supporting cases where diagnostic information is required by a test sequence but the information is not known at authoring time and therefore needs to be retrieved at runtime, for instance if a list of available communication channels is required, if different variants of a communication channel need to be queried or if details of the diagnostic services of a communication channel need to be retrieved at runtime.

The terms provided in this extension are based on the assumption that the diagnostic data associated to the specific to-be-diagnosed vehicle (model) is provided implicitly by the runtime system. The identification and retrieval of the data is the task of the initialisation process of diagnostic application; it is not intended to provide the ability to specify the diagnostic data to load by means of this extension.

NOTE 1 For an ODX-MVCI based system, the information provided by the OTX DiagDataBrowsing terms is dependent on the pre-loaded ODX data and especially on the selected vehicle information table (VIT)."

The OTX DiagDataBrowsing extension is based on the OTX DiagCom extension, as specified in Clause 7. It uses the **`diag:ComChannel`**, **`diag:DiagService`** and **`diag:Parameter`** objects from which diverse static information can be queried.

NOTE 2 In case an ODX/MVCI system is used, the targeted data is contained in the **`VEHICLE-INFORMATION`** section of the ODX data which can be queried via the ASAM MCD-3D-API.

8.2 Data types

8.2.1 Overview

The OTX DiagDataBrowsing extension introduces a single data type named **`ComChannelCategory`**, as described in the following.

8.2.2 Syntax

The syntax of the **`ComChannelCategory`** datatype declaration of the OTX DiagDataBrowsing extension is shown in Figure 33.

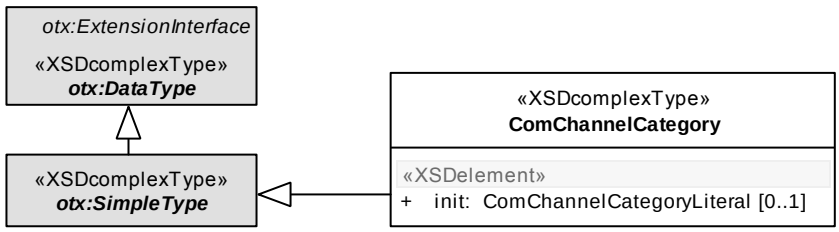

Figure 33 — Data model view: DiagDataBrowsing data types

8.2.3 Semantics

8.2.3.1 General

The **ComChannelCategory** enumeration type in the OTX DiagDataBrowsing extension is derived from **otx:SimpleType**.

8.2.3.2 ComChannelCategory

ComChannelCategory is an enumeration type describing the category of a **ComChannel**.

The list of allowed enumeration values is defined as follows:

- **BASE_VARIANT**: A **ComChannel** of this category references a base variant that is the common denominator of a set of ECU variants.
- **FUNCTIONAL_GROUP**: A **ComChannel** of this category references a functional group of ECUs, i.e. a set of ECUs that share the same functional address.
- **PROTOCOL**: A **ComChannel** of this category references a protocol-level communication link, i.e. it contains a set of diagnostic services that are common to all ECUs implementing a specific procotol.

Communication channel categories are used by **GetComChannelList** for filtering available communication channels by category (cf. 8.4.3.1). Since filtering by the fourth category – **ECU_VARIANT** – would in many cases produce a large and rather unmanageable list of ECU variants, this category is intentionally not part of the **ComChannelCategory** enumeration. Instead, the term **GetEcuVariantList** shall be used for getting only those ECU variants associated to a single ECU base variant at a time (cf. 8.4.3.2).

IMPORTANT — ComChannelCategory values may occur as operands of comparisons (cf. Part 2 of ISO 13209, relational operations). For this case, the following order relation shall apply:
BASE_VARIANT < FUNCTIONAL_GROUP < PROTOCOL.

IMPORTANT — When applying otx:ToString on a ComChannelCategory value, the resulting string shall be the name of the enumeration value, e.g. otx:ToString(PROTOCOL)="PROTOCOL". Furthermore, applying otx:ToInteger shall return the index of the value in the enumeration ComChannelCategories (smallest index is 0). The behaviour is undefined for other conversion terms (cf. Part 2 of ISO 13209).

ComChannelCategory is an **otx:SimpleType**. Its members have the following semantics:

- **<init> : ComChannelCategoryLiteral [0..1]**

  This optional element stands for the hard-coded initialisation value of the identifier at declaration time.

  - **value : ComChannelCategories={BASE_VARIANT|FUNCTIONAL_GROUP|PROTOCOL} [1]**

    This attribute shall contain one of the values defined in the **ComChannelCategories** enumeration.

IMPORTANT — If the ComChannelCategory declaration is not explicitly initialized (omitted <init> element), the default value shall be BASE_VARIANT.

8.3 Variable access

8.3.1 Overview

As specified in Part 2 of ISO 13209, OTX extensions shall define a variable access type for each datatype they define (exception types inclusively). All variable access types are derived from the OTX Core **`Variable`** extension interface. The following specifies all variable access types defined for the DiagDataBrowsing extension.

8.3.2 Syntax

Figure 34 shows the syntax of the DiagDataBrowsing extension's variable access types.

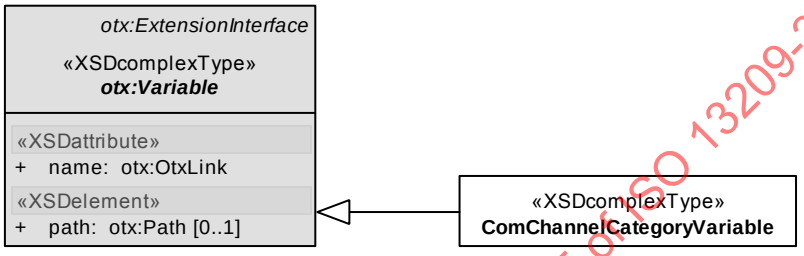

Figure 34 — Data model view: DiagDataBrowsing variable access types

8.3.3 Semantics

The general semantics for all variable access types shall apply. Please refer to Part 2 of ISO 13209 for further details.

8.4 Terms

8.4.1 Overview

The terms in the OTX DiagDataBrowsing extension shall be used to retrieve static information from the diagnostic vehicle information database, at runtime.

8.4.2 Syntax

Figure 35 shows the syntax of all terms in the OTX DiagDataBrowsing extension.

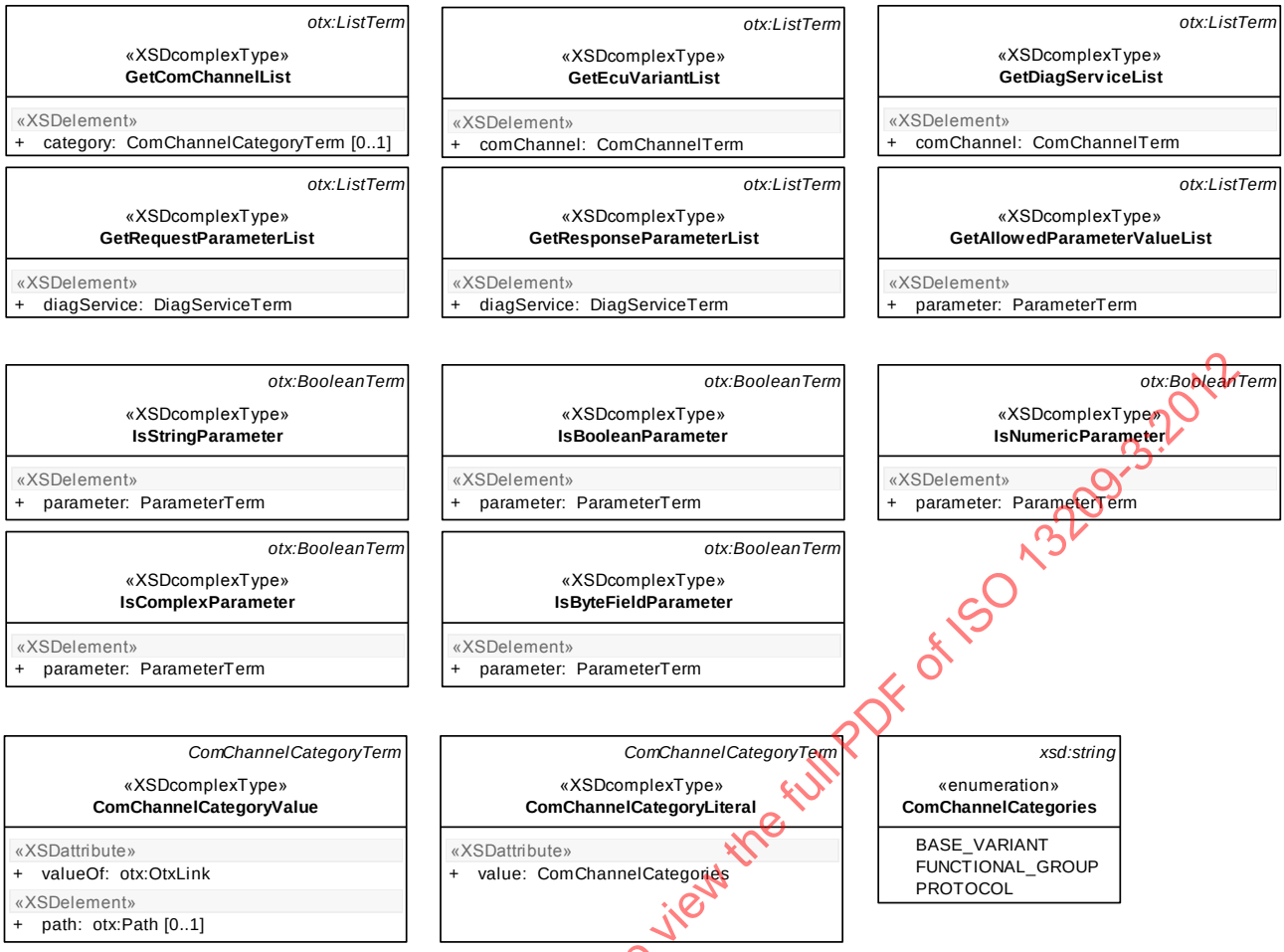

Figure 35 — Data model view: DiagDataBrowsing terms

8.4.3 Semantics

8.4.3.1 GetComChannelList

`GetComChannelList` shall return a list of strings containing the identifiers of all communication channels described in the diagnostic vehicle information data base.

If the optional attribute **`category`** is set, only those communication channel identifiers shall be returned which belong to the given category.

NOTE In the case of a MVCI/ODX based system, the equivalent of a communication channel identifier shall be the **`SHORT-NAME`** of a **`LOGICAL-LINK`**.

`GetComChannelList` is an **`otx:ListTerm`**. Its members have the following semantics:

— **`<category> : ComChannelCategoryTerm [0..1]`**

This optional element specifies the category according to which the com channels shall be filtered.

8.4.3.2 GetEcuVariantList

`GetEcuVariantList` shall return a list of strings which represents the names of all ECU variants for a given communication channel (cf. 7.7.2, communication channel related terms of the OTX DiagCom extension). The channel shall either point to a base variant or an ECU variant – in both cases, the names of the ECU variants of the base variant shall be returned. If a base variant has no associated ECU variants, an empty list shall be

returned. Furthermore, in case the communication channel points to a protocol or functional group, an exception shall be thrown.

NOTE In the case of a MVCI/ODX based system, the equivalent of a variant name shall be the **SHORT-NAME** of an **ECU-VARIANT**.

GetEcuVariantList is an **otx:ListTerm**. Its members have the following semantics:

— **<comChannel> : diag:ComChannelTerm [1]**

This element represents the communication channel which provides the data.

Throws:

— **otx:TypeMismatchException**
If the communication channel belongs to the category **PROTOCOL** or **FUNCTIONAL GROUP** (cf. 8.2.3.2).

8.4.3.3 GetDiagServiceList

GetDiagServiceList shall return a list of strings containing the names of all diagnostic services available for a given communication channel (cf. 7.7.2, communication channel related terms of OTX DiagCom).

NOTE In the case of a MVCI/ODX based system, the equivalent of a diagnostic service name shall be the **SHORT-NAME** of a **DIAG-COMM**.

GetDiagServiceList is an **otx:ListTerm**. Its members have the following semantics:

— **<comChannel> : diag:ComChannelTerm [1]**

This element represents the communication channel whose diagnostic services shall be listed.

8.4.3.4 GetRequestParameterList

GetRequestParameterList shall return a list of strings containing the names of all request parameters of a given diagnostic service (cf. 7.7.3, diagnostic service related terms of the OTX DiagCom extension).

NOTE In the case of a MVCI/ODX based system, the returned list shall contain the **SHORT-NAMEs** of all **PARAM** objects (enclosed in a **PARAMS** object of the **REQUEST**). In case a request parameter is a complex parameter (e.g. a **STRUCT**) there shall be no deep recursion into that parameter.

GetRequestParameterList is an **otx:ListTerm**. Its members have the following semantics:

— **<diagService> : diag:DiagServiceTerm [1]**

This element represents the diagnostic service whose request parameters shall be listed.

8.4.3.5 GetResponseParameterList

GetResponseParameterList shall return a list of strings containing the names of all response parameters of a given diagnostic service (cf. 7.7.3, diagnostic service related terms of the OTX DiagCom extension).

NOTE In the case of a MVCI/ODX based system, the returned list shall contain the **SHORT-NAMEs** of all **PARAM** objects (enclosed in a **PARAMS** object of the first **POS-RESPONSE**). In case a response parameter is a complex parameter (e.g. a **STRUCT**) there is NO deep recursion into that parameter.

GetResponseParameterList is an **otx:ListTerm**. Its members have the following semantics:

— **<diagService> : diag:DiagServiceTerm [1]**

This element represents the diagnostic service whose response parameters shall be listed.

8.4.3.6 GetAllowedParameterValueList

GetAllowedParameterValueList shall return a list of strings containing the allowed values for a parameter. If there is no enumeration of allowed values associated to the parameter, the empty list shall be returned.

NOTE In the case of a MVCI/ODX based system, this applies only to parameters which have a **TEXTTABLE** as **COMPU-METHOD** or to parameters which are of type **TABLE-KEY**. For those parameters the list contains all valid entries of a **TEXTTABLE** or all entries which are valid for the **TABLE-KEY**. For other parameters the returned list is empty.

GetParameterValueList is an **otx:ListTerm**. Its members have the following semantics:

— **<parameter> : diag:ParameterTerm [1]**

The element addresses the name of the request or response parameter.

8.4.3.7 IsStringParameter

IsStringParameter shall return true if and only if the given parameter represents a string value according to its definition in the diagnostic data base.

IsStringParameter is an **otx:BooleanTerm**. Its members have the following semantics:

— **<parameter> : diag:ParameterTerm [1]**

The element addresses the name of the request or response parameter to be type-tested.

8.4.3.8 IsBooleanParameter

IsBooleanParameter shall return true if and only if the given parameter represents a Boolean value according to its definition in the diagnostic data base.

IsBooleanParameter is an **otx:BooleanTerm**. Its members have the following semantics:

— **<parameter> : diag:ParameterTerm [1]**

The element addresses the name of the request or response parameter to be type-tested.

8.4.3.9 IsNumericParameter

IsNumericParameter shall return true if and only if the given parameter represents a numeric value according to its definition in the diagnostic data base.

IsNumericParameter is an **otx:BooleanTerm**. Its members have the following semantics:

— **<parameter> : diag:ParameterTerm [1]**

The element addresses the name of the request or response parameter to be type-tested.

8.4.3.10 IsByteFieldParameter

IsByteFieldParameter shall return true if and only if the given parameter represents a bytefield value according to its definition in the diagnostic data base.

IsByteFieldParameter is an **otx:BooleanTerm**. Its members have the following semantics:

— **<parameter> : diag:ParameterTerm [1]**

The element addresses the name of the request or response parameter to be type-tested.

8.4.3.11 IsComplexParameter

IsComplexParameter shall return true if and only if the given parameter neither represents a string, Boolean, numeric nor bytefield value according to its definition in the diagnostic data base.

IsComplexParameter is an **otx:BooleanTerm**. Its members have the following semantics:

— **<parameter> : diag:ParameterTerm [1]**

The element addresses the name of the request or response parameter to be type-tested.

8.4.3.12 ComChannelCategoryTerm

The abstract type **ComChannelCategoryTerm** is an **otx:SimpleTerm**. It serves as a base for all concrete terms which return a **ComChannelCategory** enumeration value (see 8.2.3.2). It has no special members.

8.4.3.13 ComChannelCategoryValue

This term returns the **ComChannelCategory** stored in a **ComChannelCategory** variable. For more information on value-terms and the syntax and semantics of the **valueOf** attribute and **<path>** element, please refer to Part 2 of ISO 13209.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

Throws:

— **otx:OutOfBoundsException**
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).

8.4.3.14 ComChannelCategoryLiteral

This term shall return a **ComChannelCategory** enumeration value (see 8.2.3.2) from a hard-coded literal.

ComChannelCategoryLiteral is a **ComChannelCategoryTerm**. Its members have the following semantics:

— **value : ComChannelCategories={BASE_VARIANT|FUNCTIONAL_GROUP|PROTOCOL} [1]**

This attribute shall contain one of the values defined in the **ComChannelCategories** enumeration.

9 OTX EventHandling extension

9.1 Introduction

At some point during execution, an OTX sequence needs to interact with the outside world. OTX sequences can cause things to happen in various ways, for example by calling actions from the HMI and DiagCom extensions. Responses can also come back into OTX through these actions (for example a blocking call to a **`hmi:ConfirmDialog`**), but in addition to these blocking mechanisms, OTX provides an event concept for finer-grained control of input events.

During the execution of an OTX procedure events may occur as a result of activities outside the procedure (for example a user screen click or a timer expires) or inside (for example a variable changes state as a result of an assignment in a parallel thread). OTX has no mechanisms (such as call-backs or listeners) to handle these events asynchronously. The OTX EventHandling extension is designed to be fully synchronous – it uses a procedural mechanism to wait for events to occur. A procedure with complex event requirements may process events sequentially in a loop until some exit criteria is encountered.

The primary elements of the OTX EventHandling extension are:

— **Event source**. An event source is something that creates events as a result of some occurrence – for example a screen press or a timer expiring. In OTX, event sources are created by terms that extend the abstract term **`EventSourceTerm`**. An event source starts queuing events right after being created. Event sources may contain multiple events in their event queue which can be removed from the queue (eldest first) by using the **`WaitForEvent`** action.

— **Event**. An event encapsulates all the information about what has occurred. Events are created and populated by event sources and can be stored in **`Event`**-type variables. Various terms exist to examine and extract content from events. There is no programmatic way to create events.

— **WaitForEvent**. The EventHandling extension defines a single action that blocks a thread of execution until an event occurs. This action is the synchronisation point between the event sources and the OTX execution thread.

9.2 Data types

9.2.1 Overview

The OTX EventHandling extension introduces two data types named **`Event`** and **`EventSource`**, as described in the following.

9.2.2 Syntax

The syntax of all OTX EventHandling data type declarations is shown in Figure 36.

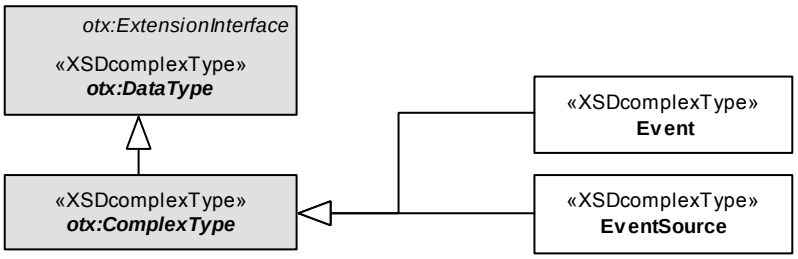

Figure 36 — Data model view: EventHandling data types

9.2.3 Semantics

9.2.3.1 General

Since the OTX Event data types have no initialisation parts, they can not be declared constant.

9.2.3.2 Event

Variables of type **`Event`** can be declared to hold events generated by event sources. **`Event`** variables cannot be initialised; therefore it is not permitted to declare an **`Event`** constant.

The **`Event`** data type encapsulates the information about a single event. There are no terms or actions to create events explicitly, programmatically – they are only created implicitly by event sources, once the awaited event occurs.

Once an event has been obtained from an event source (by using a **`WaitForEvent`** action) it can be examined using terms of the EventHandling extension (or other extensions with event handling), so for instance terms which tell which type of event source an event originates from.

Since **`Event`** has no initialisation parts, an **`Event`** can not be declared constant.

IMPORTANT — Other OTX extensions may define additional event source terms by extending the `EventSourceTerm` type. E.g. the OTX HMI extension defines the `hmi:ScreenClosedEventSource` term which listens for the closed-event when the user closes the screen.

9.2.3.3 EventSource

Variables of type **`EventSource`** are handles to event sources created by any **`EventSourceTerm`**.

Once an **`EventSource`** has been created, its internal event queue shall start registering events which correspond to the **`EventSourceTerm`** subtype chosen for creating. Queueing shall be done in a separate thread of the runtime system.

For instance, in the case of an event source which was created by a **`MonitorChangeEventSource`** term and assigned to an **`EventSource`** variable, the event source's internal queue starts registering each change event of the monitored value immediately.

Registered events may be read out and removed one by one from an event source's queue by repeatedly calling the **`WaitForEvent`** action on that event source. See 9.4.3.1 for details on the **`WaitForEvent`** action.

Event source queueing can be stopped *explicitly* by using the **`CloseEventSource`** action, as specified in 9.4.3.2. Event sources which are created on-the-fly within a **`WaitForEvent`** action shall be closed *implicitly* as soon as the action exits.

Since **`EventSource`** has no initialisation parts, an **`EventSource`** can not be declared constant.

9.3 Variable access

9.3.1 Overview

As specified in Part 2 of ISO 13209, OTX extensions shall define a variable access type for each datatype they define. All variable access types are derived from the OTX Core **`otx:Variable`** extension interface. The following specifies all variable access types defined for the OTX EventHandling extension.

9.3.2 Syntax

Figure 46 shows the syntax of the EventHandling extension's variable access types.

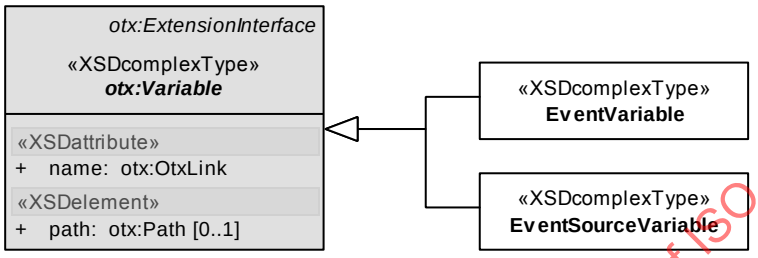

Figure 37 — Data model view: EventHandling variable access types

9.3.3 Semantics

The general semantics for all variable access types apply. Refer to Part 2 of ISO 13209 for further details.

9.4 Actions

9.4.1 Overview

The OTX EventHandling extension introduces the actions named **`WaitForEvent`** and **`CloseEventSource`**, as described in the following.

9.4.2 Syntax

Figure 38 shows the syntax of all actions in the OTX EventHandling extension.

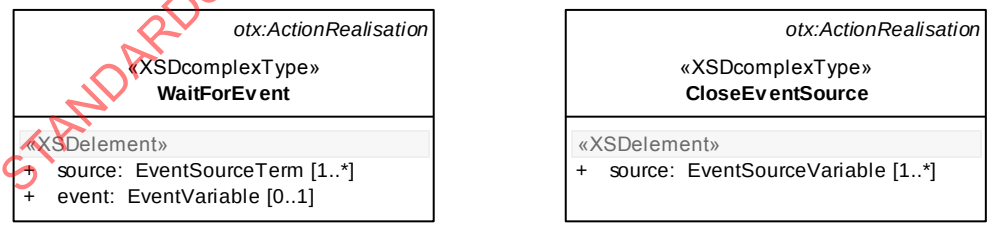

Figure 38 — Data model view: EventHandling actions

9.4.3 Semantics

9.4.3.1 WaitForEvent

The **`WaitForEvent`** action shall block the thread of execution until it receives an event from one of its event sources. As soon as an event becomes available in one of the sources' event queues, **`WaitForEvent`** shall remove that event from the event source's queue and exit; the thread of execution continues to the next node.

If an event variable was specified, the event that caused **WaitForEvent** to exit is assigned to the variable.

Special semantics apply for the following cases:

a) Situations may occur when event sources already contain one or multiple events in their event queue before being used by a **WaitForEvent** action. In that case, **WaitForEvent** shall use the *eldest* event available in any of its event sources' queues (and assign it to an event variable, if specified). If there is more than one eldest event – this may happen for events that occurred at the same time – the event of the event source which is listed first in the action shall be used (first in XML document order).

b) When **WaitForEvent** exits, those event sources which were created on-the-fly within the action shall be closed (the ones that are not assigned to an **EventSource** variable).

In order to determine later which of the event sources has fired the event, the terms described in 9.5.4 should be used.

The members of the **WaitForEvent** action have the following semantics:

— **<source> : EventSourceTerm [1..*]**

This represents one or more event sources that the action shall wait for. The wait shall be terminated by the first source to fire an event.

— **<event> : EventVariable [0..1]**

This optional element represents an **Event**-type variable which shall receive the event that terminates this wait.

9.4.3.2 CloseEventSource

The **CloseEventSource** action shall close and dispose given event sources. Closed event sources will no more queue any events.

Once closed, an event source can not be reopened. Using a closed event source e.g. in a **WaitForEvent** action is an error and will cause an **otx:InvalidReferenceException** (through the **EventSourceValue** term as specified in 9.5.2.3.2).

In case that **CloseEventSource** is applied to an event source which is already closed, the action shall perform nothing (NOP).

CAUTION — In parallel execution, situations may occur where an event source gets closed by a CloseEventSource action while being used in a WaitForEvent action (in another parallel lane). If the WaitForEvent action has no other event sources registered, this will cause a deadlock situation. OTX authors should avoid such situations by careful test sequence design and the usage of the MutexGroup node, as specified by Part 2 of ISO 13209.

The members of the **CloseEventSource** action have the following semantics:

— **<source> : EventSourceVariable [1..*]**

This represents one or more variables which contain the event sources that shall be closed.

9.4.4 Example

The example below shows the use of the **WaitForEvent** action with a **MonitorChangeEventSource** on a variable **x** and a **TimerExpiredEventSource** of 10 seconds.

The **MonitorChangeEventSource** starts queueing change events of variable **x** prior to being used in the **WaitForEvent** action, right after being created in the **Assignment** action. In constrast the **TimerExpiredEventSource** is created on-the-fly inside of the **WaitForEvent**.

If **x** does not change its value, the wait will exit after 10 seconds. In any case – once one of the event sources fires the event, it is assigned to the event variable **myEvent** which might be used later for analysis.

After the wait, the **MonitorChangeEventSource** is closed by an explicit **CloseEventSource** action. By contrast, the **TimerExpiredEventSource** is closed implicitly as soon as the wait exits.

EXAMPLE Sample of OTX-file "EventHandlingExample.otx"

```
<action id="a1">
  <specification>Create a MonitorChangeEventSource listening to variable x</specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="event:EventSourceVariable" name="xMonitor"/>
    <term xsi:type="event:MonitorChangeEventSource">
      <event:variable xsi:type="IntegerVariable" name="x"/>
    </term>
  </realisation>
</action>

<action id="a2">
  <specification>Wait for a change of x's value, stop waiting after 10 seconds</specification>
  <realisation xsi:type="event:WaitForEvent">
    <event:source xsi:type="event:EventSourceValue" valueOf="xMonitor"/>
    <event:source xsi:type="event:TimerExpiredEventSource">
      <event:timeout value="10000" xsi:type="IntegerLiteral"/>
    </event:source>
    <event:event name="myEvent"/>
  </realisation>
</action>

<action id="a3">
  <specification>Close xMonitor event source</specification>
  <realisation xsi:type="event:CloseEventSource">
    <event:source name="xMonitor"/>
  </realisation>
</action>
```

9.5 Terms

9.5.1 Overview

The Terms of the OTX EventHandling extension are grouped into three different categories:

— **Event terms.** Event terms return events. The OTX EventHandling extension defines exactly one event term named **EventValue**.

— **Event source terms**. Event source terms can be used within **WaitForEvent** actions. This extension defines several event sources, but additional event sources may be defined in other OTX extensions. In particular the OTX HMI extension defines the **hmi:ScreenClosedEventSource** term as a source of GUI events.

— **Event property terms**. The terms in this category are used to examine events that are produced by event sources. They all operate on an event that is accessed using an **EventTerm** and return one of the values stored in the event for further processing.

The term categories described above are shown in Figure 39.

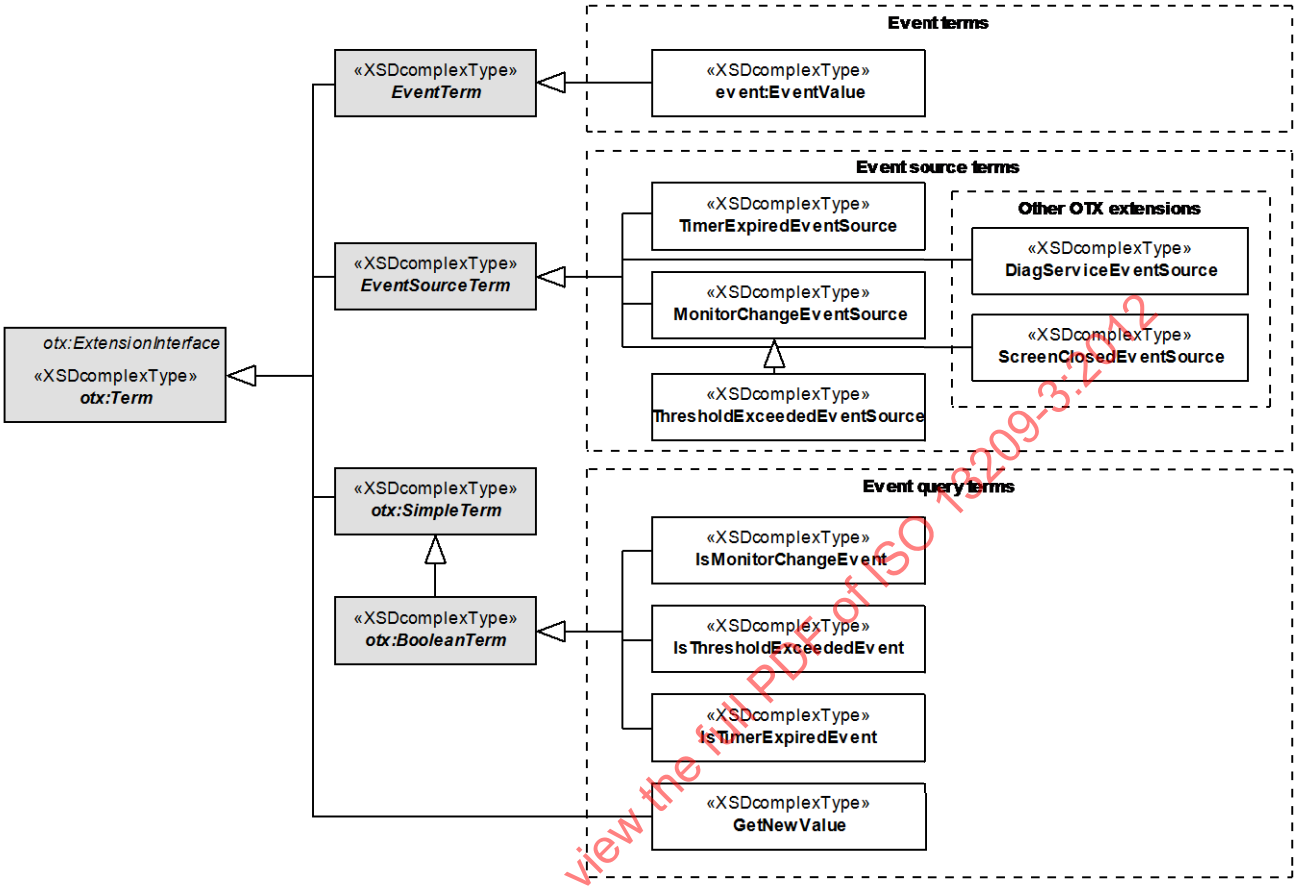

Figure 39 — Data model view: EventHandling term categories

9.5.2 Event terms

9.5.2.1 Description

Terms in this category return events.

9.5.2.2 Syntax

Figure 40 shows the syntax of the event terms.

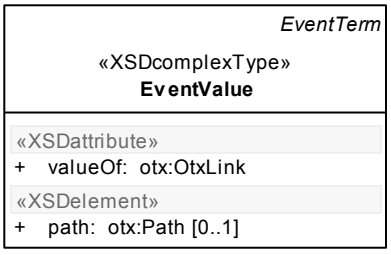

Figure 40 — Data model view: Event terms

9.5.2.3 Semantics

9.5.2.3.1 EventTerm

The abstract type **`EventTerm`** is an **`otx:Term`**. It serves as a base for all concrete terms which return an **`Event`**. It has no special members.

9.5.2.3.2 EventValue

This term returns the **`Event`** stored in an **`Event`** variable. For more information on value-terms and the syntax and semantics of the **`valueOf`** attribute and **`<path>`** element, please refer to Part 2 of ISO 13209.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

Throws:

— **`otx:OutOfBoundsException`**
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).

— **`otx:InvalidReferenceException`**
If the variable value is not valid (no value was assigned to the variable before).

9.5.3 Event source terms

9.5.3.1 Description

Terms in this category represent event sources. In a **`WaitForEvent`** action, any of the event source terms defined here or in other OTX extensions may be used. The **`WaitForEvent`** action waits so long until one of the embedded event source term fires an event.

NOTE It is an intended design goal of the OTX EventHandling extension that there is no explicit **`EventSource`** data type defined. Therefore it is not possible to declare **`EventSource`** variables. Event source terms are useable only within **`WaitForEvent`** actions.

9.5.3.2 Syntax

Figure 41 shows the syntax of the event source terms.

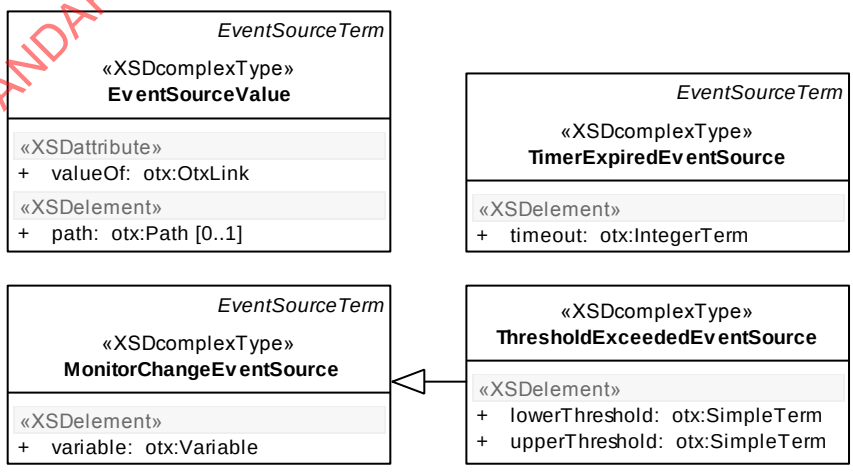

Figure 41 — Data model view: Event source terms

9.5.3.3 Semantics

9.5.3.3.1 EventSourceTerm

The abstract type **`EventSourceTerm`** is an **`otx:Term`**. It serves as a base for all concrete terms which return an **`EventSource`**. It has no special members.

9.5.3.3.2 EventSourceValue

This term returns the **`Event`** stored in an **`Event`** variable. For more information on value-terms and the syntax and semantics of the **`valueOf`** attribute and **`<path>`** element, please refer to Part 2 of ISO 13209.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

Throws:

— **`otx:OutOfBoundsException`**
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).

— **`otx:InvalidReferenceException`**
If the variable value is not valid (no value was assigned to the variable before).

9.5.3.3.3 MonitorChangeEventSource

This term creates an event source that shall monitor a variable's value and fire an event when it changes. The fired event shall maintain a snapshot of the new value of the monitored variable, which may be read out later (see **`GetNewValue`** term). Event queueing shall start immediately once the event source is created.

IMPORTANT — Change-monitoring shall be *shallow*. This means that changes inside of complex values shall NOT be recognized, like e.g. a change of an item in a `List` or `Map`, or the removal of items from a `List` or `Map`, etc. Regarding complex data types the only recognized change is when the variable changes its value, e.g. when another `List` is assigned to the variable.

IMPORTANT — The case when a value is assigned to a formerly uninitialized variable shall also be recognized as a change event and shall NOT pose an error.

`MonitorChangeEventSource` is an **`EventSourceTerm`**. Its members have the following semantics:

— **`<variable> : otx:Variable [1]`**

Represents the variable that shall be monitored. If the variable value changes, the event shall be fired, causing an embedding **`WaitForEventAction`** to exit.

Associated checker rules:

— Event_Chk002 – No Path in MonitorChange related terms

9.5.3.3.4 ThresholdExceededEventSource

This term creates an event source that shall monitor the value of a variable and fire an event when the value goes outside a specified range. If the value is outside of the specified range right from the start, the event shall be fired, too. The fired event shall maintain a snapshot of the new value that exceeded the threshold, which may be read out later (see **`GetNewValue`** term).

Event queueing shall start immediately once the event source is created.

This event source term shall only be applied for data types on which an *order relation* is defined. These are the **`SimpleType`** data types as specified in Part 2 of ISO 13209.

IMPORTANT — A `ThresholdExceededEventSource` which is applied to an uninitialized variable shall also count as threshold exceeded event and does NOT pose an error.

`ThresholdExceededEventSource` is a **`MonitorChangeEventSource.`** Its members have the following semantics:

— **`<variable> : otx:Variable [1]`** (derived from **`MonitorChangeEventSource`**)

Represents the variable that shall be monitored. If the variable value goes outside of the specified range (see below) or is already outside from the beginning, the event shall be fired, causing an embedding **`WaitForEventAction`** to exit.

— **`<lowerThreshold>: otx:SimpleTerm [1]`**

Represents a value to compare against. If the value of the monitored variable becomes less that this value, the event shall be fired.

— **`<upperThreshold>: otx:SimpleTerm [1]`**

Represents a value to compare against. If the value of the monitored variable becomes greater than this value, the event shall be fired.

Associated checker rules:

— Event_Chk002 – No Path in MonitorChange related terms

— Event_Chk001 – Correct data types of ThresholdExceededEventSource arguments

9.5.3.3.5 TimerExpiredEventSource

This term shall create an event source that produces an event when a specified time expires. If the specified time expires, the timer expiry event is produced and put into the event source's queue. Event queueing shall start immediately once the event source is created.

`TimerExpiredEventSource` is an **`EventSourceTerm`**. Its members have the following semantics:

— **`<timeout>: otx:NumericTerm [1]`**

This element specifies an **`Integer`** value that is interpreted as a time in milli-seconds to wait. Once the given number of milli-seconds has passed, the event shall be fired, causing an embedding **`WaitForEventAction`** to exit. **`Float`** values shall be truncated.

Throws:

— **`otx:OutOfBoundsException`**
if the timeout value is negative.

9.5.4 Event property terms

9.5.4.1 Description

Terms in this category return diverse information on event properties.

9.5.4.2 Syntax

Figure 42 shows the syntax of the event property terms.

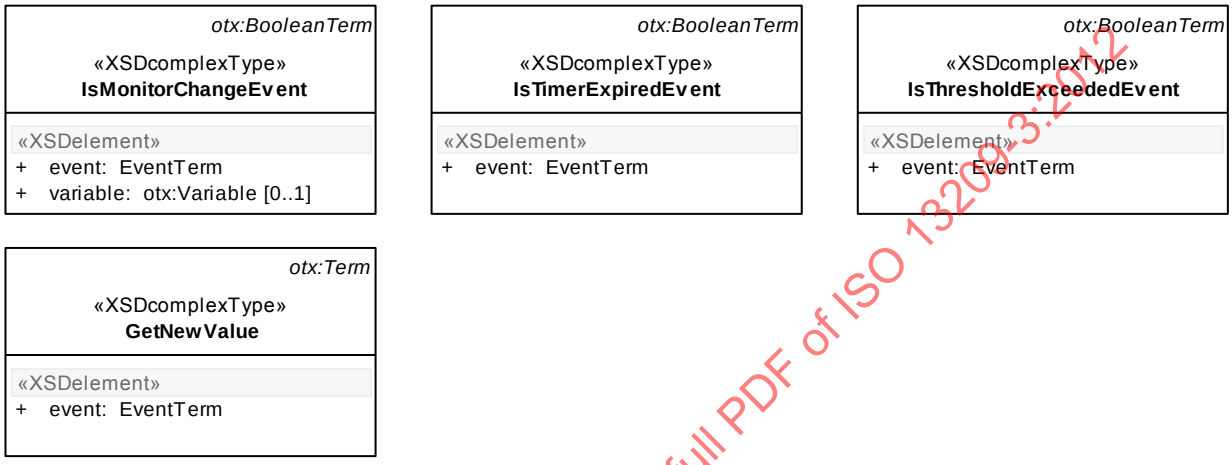

Figure 42 — Data model view: Event property terms

9.5.4.3 Semantics

9.5.4.3.1 IsMonitorChangeEvent

The **`IsMonitorChangeEvent`** term accepts an **`EventTerm`** yielding an **`Event`** object that has been raised by the OTX runtime system, as a result of either using a **`MonitorChangeEventSource`** or a **`ThresholdExceededEventSource`** in a **`WaitForEvent`** action. The term shall return **`true`** if and only if the **`Event`** originates from such a kind of event source. In case an optional **`Variable`** is specified, the term shall return **`true`** if and only if the **`Event`** was fired because that particular **`Variable`** changed. If the given **`Variable`** was not the reason for the event, **`false`** shall be returned.

`IsMonitorChangeEvent` is an **`otx:BooleanTerm`**. Its members have the following semantics:

— **`<event> : EventTerm [1]`**

Represents the **`Event`** whose type shall be tested.

— **`<variable> : otx:Variable [0..1]`**

Optionally specifies the variable which shall be tested for being the reason for the event.

Associated checker rules:

— Event_Chk002 – No Path in MonitorChange related terms

9.5.4.3.2 IsTresholdExceededEvent

The **`IsThresholdExceededEvent`** term accepts an **`EventTerm`** yielding an **`Event`** object that has been raised by the OTX runtime, as a result of using a **`ThresholdExceededEventSource`** in a **`WaitForEvent`** action. The term shall return **`true`** if and only if the **`Event`** originates from a **`ThresholdExceededEventSource`**.

IsThresholdExceededEvent is an **otx:BooleanTerm**. Its members have the following semantics:

— **<event> : EventTerm [1]**

Represents the **Event** whose type shall be tested.

9.5.4.3.3 IsTimerExpiredEvent

The **IsTimerExpiredEvent** term accepts an **EventTerm** term yielding an **Event** object that has been raised by the OTX runtime, as a result of using a **TimerExpiredEventSource** in a **WaitForEvent** action. The term shall return **true** if and only if the **Event** originates from a **TimerExpiredEventSource**.

IsTimerExpiredEvent is an **otx:BooleanTerm**. Its members have the following semantics:

— **<event> : EventTerm [1]**

Represents the **Event** whose type shall be tested.

9.5.4.3.4 GetNewValue

GetNewValue shall only be applied to events which were fired by a **MonitorChangeEventSource** or one of its descendants. The term shall return the value which was stored in the given **Event**; that value represents a snapshot of the monitored variable's new value at the time when the event was fired. The term is useful to find out which new value a variable had after it changed.

IMPORTANT — Since it depends on the datatype of the variable which was monitored by MonitorChangeEventSource, the return type of GetNewValue is in general not known at authoring time. Therefore, type-safety of this term cannot be checked statically. Runtime exceptions may occur when results of this term are used in the wrong place, e.g. when using otx:ToInteger on a value which cannot be converted to integer.

GetNewValue is an **otx:Term**. Its members have the following semantics:

— **<event> : EventTerm [1]**

Represents the monitor change event from which the new value of the formerly monitored variable at the time of value change shall be returned.

Throws:

— **otx:TypeMismatchException**
If the specified event has not been raised by a **MonitorChangeEventSource** or one of its descendants.

10 OTX Flash extension

10.1 Introduction

The OTX Flash extension provides access to data types, terms and actions for reading data from a flash session context and creating flash jobs.

IMPORTANT — It is an explicit design goal of the OTX Flash extension that it supports the flash data acquisition side in the flash process *only*. There are no actions defined herein which carry out the actual ECU flashing; this functionality is provided already by the OTX DiagCom extension as specified in Clause 6.

The OTX Flash extension is designed for flash-data acquisition and flash job creation; downloading to an ECU shall happen by executing a flash job via **`ExecuteDiagService`** as defined by the OTX DiagCom extension.

The Flash extension assumes that several flash sessions can exist for a communication channel. A flash session contains several flash blocks and a flash block several flash segments. The segments contain an arbitrary number of data bytes. Since data can be compressed, size information is supplied. Additionally security information is attached to blocks and the session.

The Flash extension is designed to support use cases from the flash process domain, e.g. choose a flash session and handle low level functions which are needed inside a flash job to access flash data and its additional information.

NOTE 1 It is an explicit design goal of the OTX Flash extension to be usable with any diagnostic communication kernel. As a design guideline, an ODX/MVCI based system has been considered – as ODX/MVCI is solving the vehicle communication problem domain on a highly generic level, the design concepts that have been adopted for this extension should be usable abstractions for any system that is implementing a solution to the vehicle communication problem domain.

NOTE 2 In an ODX/MVCI based system, the session context is an ODX ECU-MEM container. Therefore the examples regarding the usage of the terms and actions of the Flash extension describe ODX scenarios. Nevertheless it is possible to use a subset of the nodes to describe download via proprietary protocols and raw data sources like binary.

Figure 43 shows the data structure model of the OTX Flash extension.

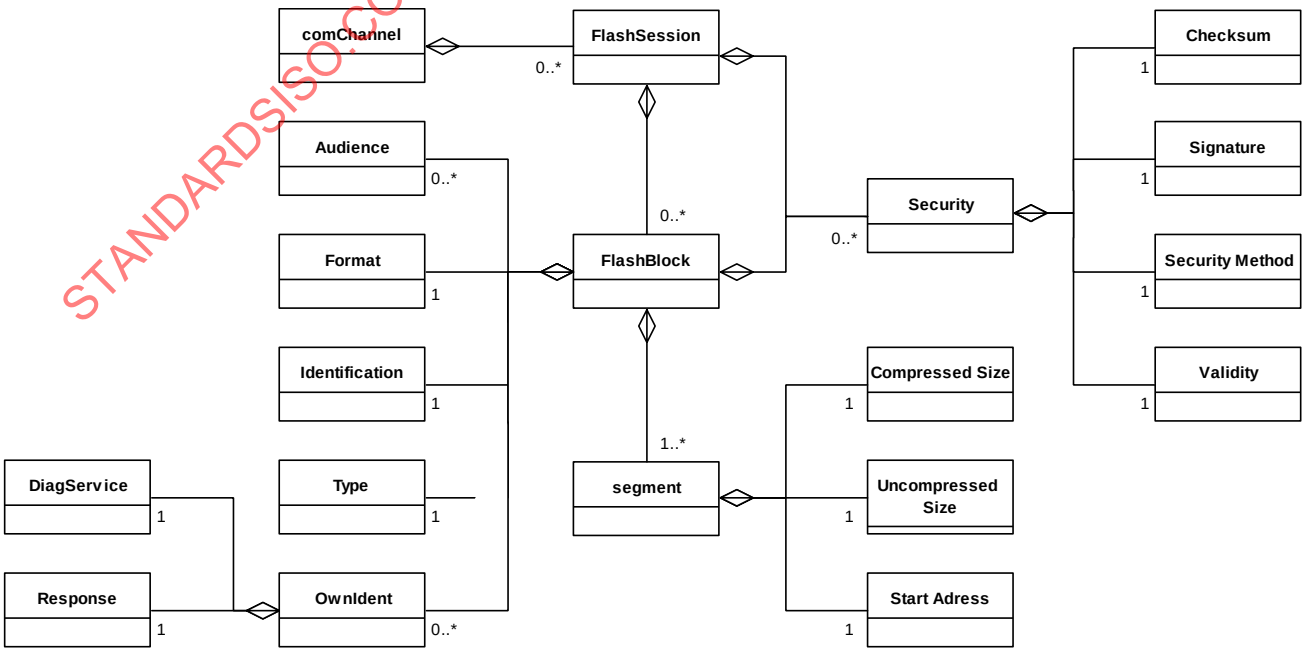

Figure 43 — Data structure model of the OTX Flash extension

10.2 Data types

10.2.1 Overview

The OTX Flash extension introduces the data types named **`FlashJob`** and **`FlashSession`**, as well as the enumeration types **`FlashFileFormat`** and **`Audience`**.

10.2.2 Syntax

The syntax of all OTX Flash data type declarations is shown in Figure 44.

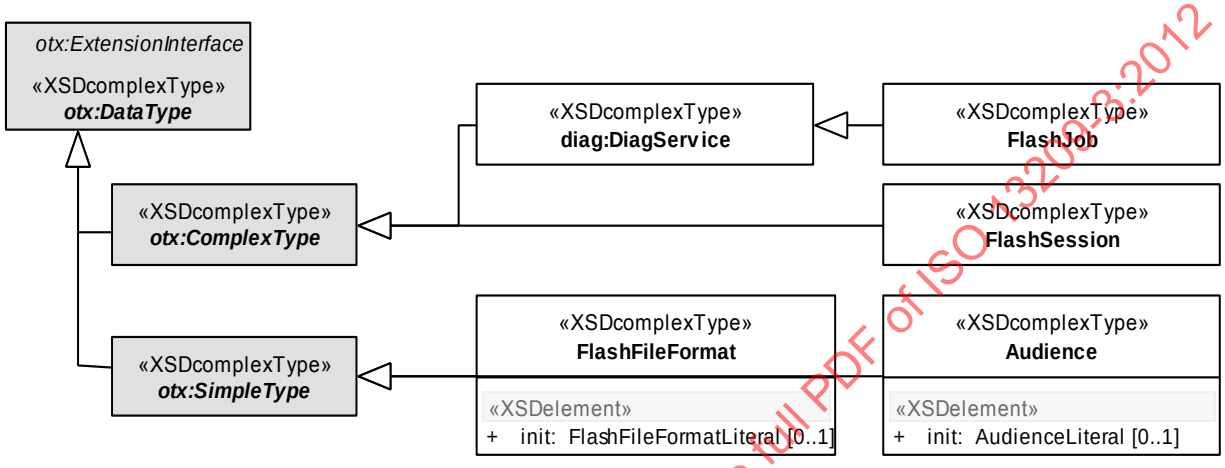

Figure 44 — Data model view: Flash data types

10.2.3 Semantics

10.2.3.1 General

The data types in the OTX Flash extension are based on **`otx:ComplexType`** and on **`otx:SimpleType`**.

10.2.3.2 FlashJob

The **`FlashJob`** data type represents a diagnostic service that is used for performing the ECU reprogramming process. Based on the concepts of the ODX/MVCI standard, a **`FlashJob`** can be parameterized with a specific flash session which contains the data to be programmed into the ECU. This is the interface-level difference between a **`FlashJob`** and a **`diag:DiagService`**. To parameterize a **`FlashJob`** with a flash session, please refer to the **`SetFlashSession`** action (see 10.5.3.3).

10.2.3.3 FlashSession

The **`FlashSession`** data type serves as storage for information regarding the context of a diagnostic session and download information [ISO 22901].

Since **`FlashSession`** has no initialisation parts, a **`FlashSession`** can not be declared constant.

10.2.3.4 FlashFileFormat

`FlashFileFormat` is an enumeration type describing the format of a flash file. It is used by the action **`StoreUploadData`** (see 10.5.3.2).

OTX runtimes should at least support a basic set of flash file formats, which is defined by the following list of allowed enumeration values:

— **BINARY**: Raw binary data

— **INTEL**: Intel hex file

— **SREC**: Motorola S-Record file

IMPORTANT — `FlashFileFormatTerm` values may occur as operands of comparisons (cf. Part 2 of ISO 13209, relational operations). For this case, the following order relation shall apply:
`BINARY < INTEL < SREC`.

IMPORTANT — When applying `otx:ToString` on a `FlashFileFormat` value, the resulting string shall be the name of the enumeration value, e.g. `otx:ToString(BINARY)="BINARY"`. Furthermore, applying `otx:ToInteger` shall return the index of the value in the `FlashFileFormat` enumeration (smallest index is `0`). The behaviour is undefined for other conversion terms (cf. Part 2 of ISO 13209).

`FlashFileFormat` is an **`otx:SimpleType`**. Its members have the following semantics:

— **`<init> : FlashFileFormatLiteral [0..1]`**

This optional element stands for the hard-coded initialisation value of the identifier at declaration time.

— **`value : FlashFileFormats={BINARY|SREC|INTEL} [1]`**

This attribute shall contain one of the values defined in the **`FlashFileFormats`** enumeration.

IMPORTANT — If the `FlashFileFormat` declaration is not explicitly initialized (omitted `<init>` element), the default value shall be `BINARY`.

10.2.3.5 Audience

`Audience` is an enumeration type which is used by the term **`GetListOfValidFlashSessions`** (for filtering flash sessions according to audience property) as well as by the term **`BlockIsValidForAudience`** (see 10.6.3.3.3 and 10.6.4.3.7).

The list of allowed enumeration values is defined as follows:

— **`"SUPPLIER"`**

— **`"DEVELOPMENT"`**

— **`"MANUFACTURING"`**

— **`"AFTERSALES"`**

— **`"AFTERMARKET"`**

IMPORTANT — `AudienceTerm` values may occur as operands of comparisons (cf. Part 2 of ISO 13209, relational operations). For this case, the following order relation shall apply:
`SUPPLIER < DEVELOPMENT < MANUFACTURING < AFTERSALES < AFTERMARKET`.

IMPORTANT — When applying `otx:ToString` on an `Audience` value, the resulting string shall be the name of the enumeration value, e.g. `otx:ToString(SUPPLIER)="SUPPLIER"`. Furthermore, applying `otx:ToInteger` shall return the index of the value in the `Audiences` enumeration (smallest index is `0`). The behaviour is undefined for other conversion terms (cf. Part 2 of ISO 13209).

Audience is an **otx:SimpleType**. Its members have the following semantics:

— **<init> : AudienceLiteral [0..1]**

This optional element stands for the hard-coded initialisation value of the identifier at declaration time.

— **value : Audiences={SUPPLIER|DEVELOPMENT|MANUFACTURING|AFTERSALES|AFTERMARKET} [1]**

This attribute shall contain one of the values defined in the **Audiences** enumeration.

IMPORTANT — If the Audience declaration is not explicitly initialized (omitted <init> element), the default value shall be SUPPLIER.

10.3 Exceptions

10.3.1 Overview

All elements referenced in this Clause are derived from the OTX Core **Exception** type as defined by Part 2 of ISO 13209. They represent the full set of exceptions added by the OTX Flash extension.

10.3.2 Syntax

The syntax of all OTX Flash exception type declarations is shown in Figure 45.

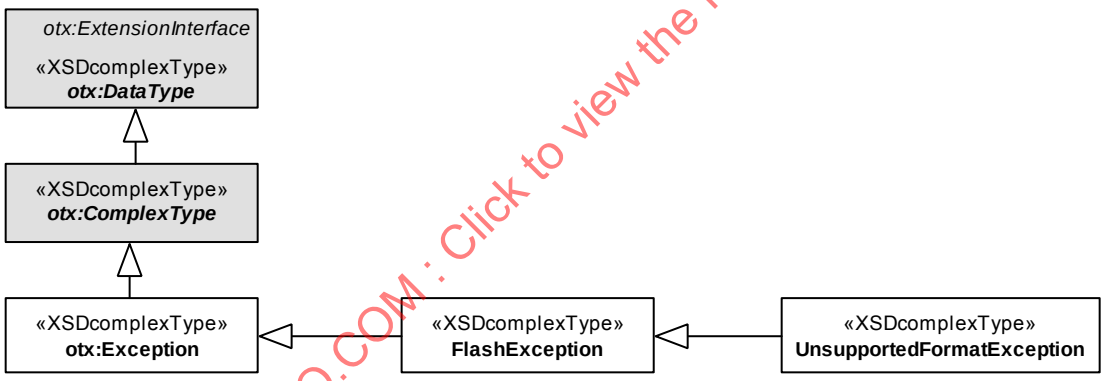

Figure 45 — Data model view: Flash exceptions

10.3.3 Semantics

10.3.3.1 General

Since all OTX Flash exception types are implicit exceptions whithout initialisation parts, they can not be declared constant.

10.3.3.2 FlashException

The **FlashException** is the super class for all exceptions in the Flash extension. A **FlashException** shall be used in case the more specific exception types described in the remainder of this section do not apply to the problem at hand.

10.3.3.3 UnsupportedFormatException

The **UnsupportedFormatException** shall be thrown if the flash file format used by a **StoreUploadData** action is not supported by the runtime system.

10.4 Variable access

10.4.1 Overview

As specified in Part 2 of ISO 13209, OTX extensions shall define a variable access type for each datatype they define. All variable access types are derived from the OTX Core **`otx:Variable`** extension interface. The following specifies all variable access types defined for the Flash extension.

10.4.2 Syntax

Figure 46 shows the syntax of the Flash extension's variable access types.

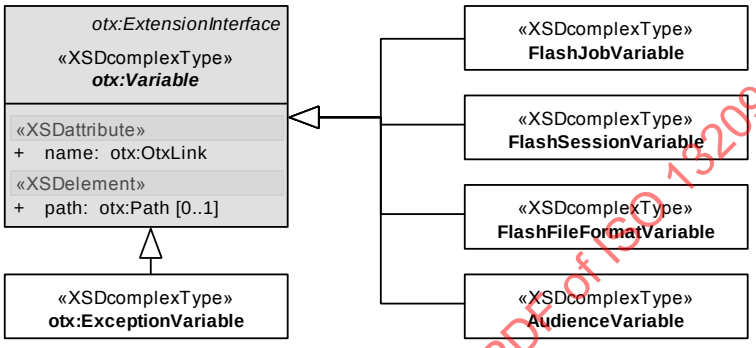

Figure 46 — Data model view: Flash variable access types

10.4.3 Semantics

The general semantics for all variable access types shall apply. Please refer to Part 2 of ISO 13209 for details.

10.5 Actions

10.5.1 Overview

There are three action types defined for the OTX Flash extension: **`GetDownloadData`**, **`StoreUploadData`** as well as **`SetFlashSession`**. The types extend the **`ActionRealisation`** extension interface as defined by Part 2 of ISO 13209.

10.5.2 Syntax

Figure 47 shows the syntax of the actions **`GetDownloadData`** and **`StoreUploadData`**.

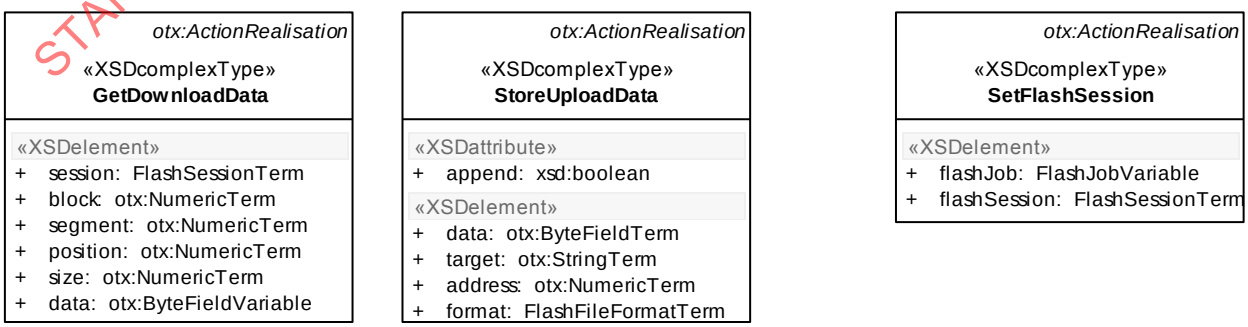

Figure 47 — Data model view: Flash actions

10.5.3 Semantics

10.5.3.1 GetDownloadData

GetDownloadData shall fill a given **otx:ByteField** variable with data from the **FlashSession** context.

The members of **GetDownloadData** have the following semantics:

— **<session> : FlashSessionTerm [1]**

Represents the **FlashSession** which provides the data that shall be addressed.

— **<block> : otx:NumericTerm [1]**

By this element, a block in the **FlashSession** context shall be addressed. The value shall be in the range of the existing blocks. **Float** values shall be truncated.

— ** : otx:NumericTerm [1]**

This element shall address a segment in the **FlashSession** context. The value shall be in the range of the existing segments in the block. **Float** values shall be truncated.

— **<size> : otx:NumericTerm [1]**

This element defines how much bytes of memory shall be read from the context. It shall be a positive value. **Float** values shall be truncated.

— **<position> : otx:NumericTerm [1]**

This element defines the first position which shall be read by the action. Position shall be greater than or equal to zero and not greater than the size of the segment minus one. **Float** values shall be truncated.

— **<data> : otx:ByteFieldVariable [1]**

This element represents the variable into which the read data shall be stored. It shall be of the type **otx:ByteFieldVariable**. The size of the **ByteField** after execution of the action should be the number of bytes read from the context. If the context does not contain the amount of data which is requested with the size parameter, then the resulting **ByteField** is shorter. If the position parameter overlaps the segment size, the resulting **ByteField** will be empty.

Throws:

— **otx:OutOfBoundsException**
If the block, segment or position number does not exist in the download data or if size is zero or negative.

10.5.3.2 StoreUploadData

A **StoreUploadData** action tells an OTX runtime to store data in a data-storage.

The members of **StoreUploadData** have the following semantics:

— **append : xsd:boolean [1]**

The truth-value set for this attribute defines whether data shall be appended to existing data (**true**) or not (**false**). If not, the storage shall be cleaned before write access.

— **<data> : otx:ByteFieldTerm [1]**

This element represents the data which shall be stored.

— **<target> : otx:StringTerm [1]**

The element shall provide a data storage. If the target is an URI that describes a file, the data is stored in that file.

— **<address> : otx:NumericTerm [1]**

This element shall be used to define the base address of the to-be-stored data. **Float** values shall be truncated.

— **<format> : FlashFileFormatTerm [1]**

This element defines the format of the flash data file. The basic set of formats which should be supported by any runtime system specified by the **FlashFileFormat** data type (see 10.2.3.4). For other proprietary formats, proprietary extensions may be used.

Throws:

— **otx:InvalidReferenceException**
If the data storage resource given by the **<target>** element is not available or not accessible.

— **UnsupportedFormatException**
If the runtime system does not support the flash data file format.

10.5.3.3 SetFlashSession

This action shall set the flash session to be programmed when the **FlashJob** is executed. Only one session can be set at a time. If this action is used multiple times, the later call shall overwrite the session set by a previous call.

The members of **SetFlashSession** have the following semantics:

— **<flashJob> : FlashJobVariable [1]**

Represents the **FlashJob** where the session shall be set.

— **<flashSession> : FlashSessionTerm [1]**

Represents the **FlashSession** to be programmed by the **FlashJob**.

10.5.4 Example

The example below shows a **GetDownloadData** action working on mySession, block 1, segment 1, position 0 and a size request of 64 bytes. The data is assigned to the **ByteField**-variable **"myData"**.

The second part of the example shows a **StoreUploadData** action with appends the data contained in a **ByteField**-variable named **"data"** to an INTEL-format storage-file at **"file://file.hex"**.

EXAMPLE Sample of OTX-file "FlashActionsExample.otx"

```
<action id="a1">
  <specification>
    Get 64 bytes of data from mySession, block 1, segment1, position 0 and put it in myData
  </specification>
  <realisation xsi:type="flash:GetDownloadData">
    <flash:session xsi:type="flash:FlashSessionValue" valueOf="mySession"/>
    <flash:block xsi:type="IntegerLiteral" value="1"/>
    <flash:segment xsi:type="IntegerLiteral" value="1"/>
    <flash:position xsi:type="IntegerLiteral" value="0"/>
    <flash:size xsi:type="IntegerLiteral" value="64"/>
    <flash:data xsi:type="ByteFieldVariable" name="myData"/>
  </realisation>
</action>
<action id="a2">
  <specification>Store the upload data in file file.hex</specification>
```

```
    <realisation xsi:type="flash:StoreUploadData" append="true">
      <flash:data xsi:type="ByteFieldValue" valueOf="data"/>
      <flash:target xsi:type="StringLiteral" value="file://file.hex"/>
      <flash:address xsi:type="IntegerLiteral" value="1024"/>
      <flash:format xsi:type="flash:FlashFileFormatLiteral" value="INTEL"/>
    </realisation>
  </action>
```

10.6 Terms

10.6.1 Overview

The terms of the OTX Flash extension are sorted into several categories, depending on whether they are mainly flash job-, session-, block-, segment-, security- or own ident related. Additionally, there are auxiliary enumeration-type term categories for describing flash file format types and audiences.

IMPORTANT — For all terms described in the following, it is assumed that the blocks in a flash session's data will be numbered starting from 0 (first block). The same applies to segment- and own ident-numbering.

Figure 48 shows an overview of the OTX Flash extension term categories.

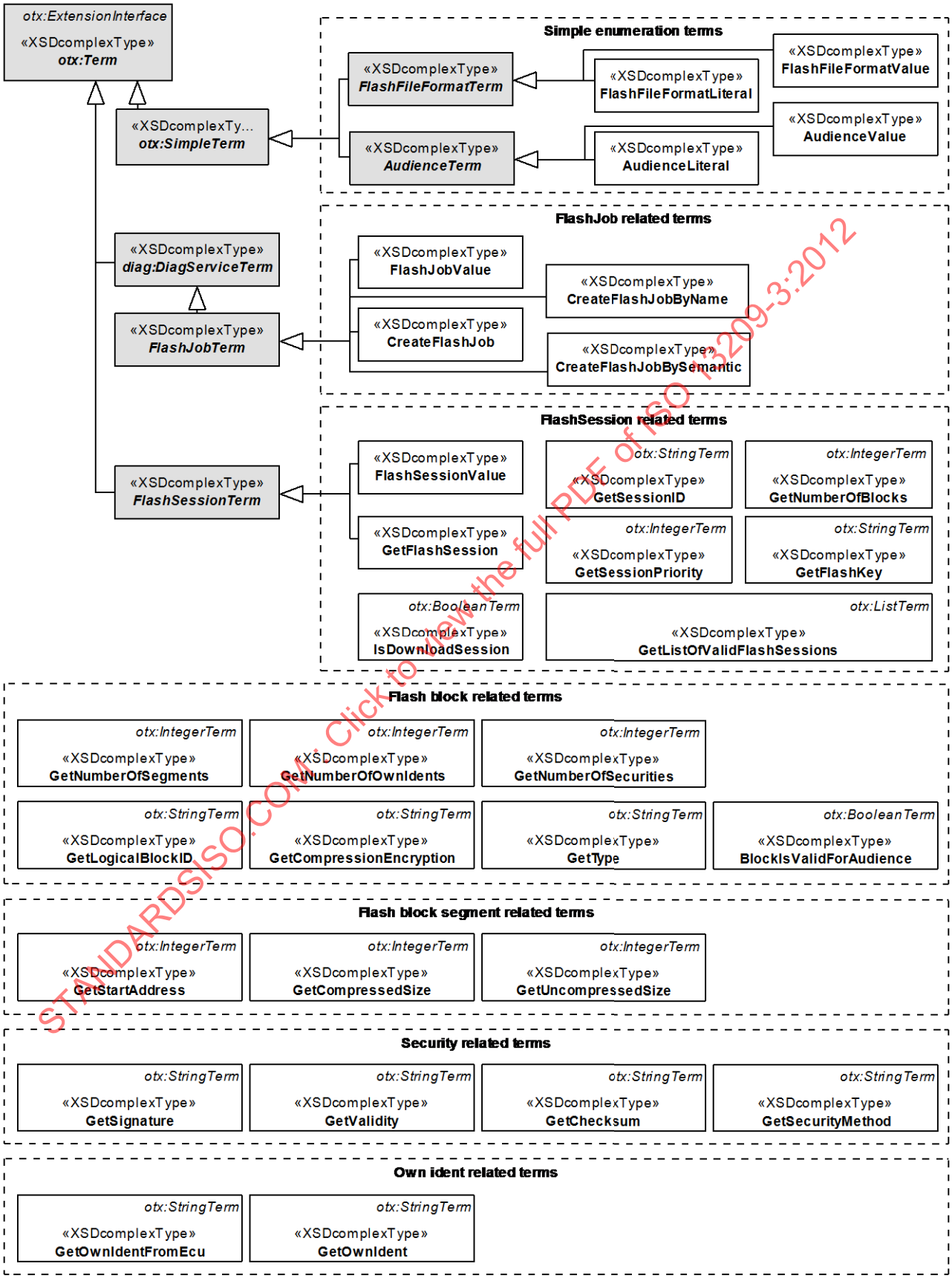

Figure 48 — Data model view: Flash term categories

10.6.2 Flash job related terms

10.6.2.1 Description

The following describes the flash job related terms of the OTX Flash extension.

10.6.2.2 Syntax

Figure 49 shows the syntax of the flash job related terms.

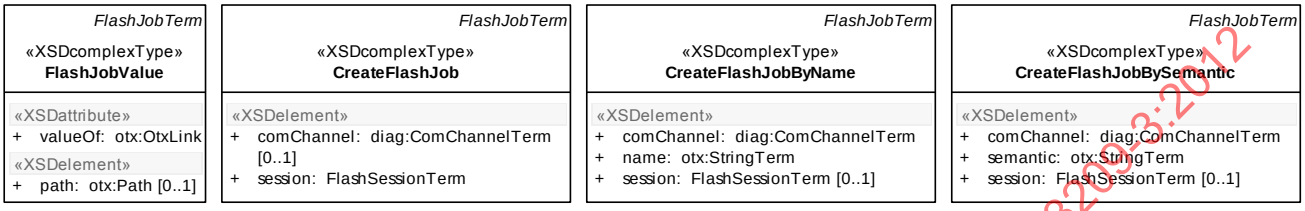

Figure 49 — Data model view: Flash job related terms

10.6.2.3 Semantics

10.6.2.3.1 FlashJobTerm

The abstract type **`FlashJobTerm`** is a **`diag:DiagServiceTerm`**. It serves as a base for all concrete terms which return a **`FlashJob`**. It has no special members.

10.6.2.3.2 FlashJobValue

This term returns the **`FlashJob`** stored in a **`FlashJob`** variable. For more information on value-terms and the syntax and semantics of the **`valueOf`** attribute and **`<path>`** element, please refer to Part 2 of ISO 13209.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

Throws:

— **`otx:OutOfBoundsException`**
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).

— **`otx:InvalidReferenceException`**
If the variable value is not valid (no value was assigned to the variable before).

10.6.2.3.3 CreateFlashJob

This term shall create a new **`FlashJob`** for the specified **`FlashSession`**. The **`FlashJob`** can subsequently be used for initiating an ECU reprogramming session.

`CreateFlashJob` is a **`FlashJobTerm`**. Its members have the following semantics:

— **`<comChannel> : diag:ComChannelTerm [0..1]`**

This optionally specifies the **`diag:ComChannel`** object to which the to-be-created **`FlashJob`** belongs to and will be executed on when the **`diag:ExecuteDiagService`** action is used (cf. 7.6.4.3.1).

— **`<session> : FlashSessionTerm [1]`**

This element represents the **`FlashSession`** to be programmed by the **`FlashJob`**.

10.6.2.3.4 CreateFlashJobByName

This term shall create a new **FlashJob** for the specified **ComChannel**. The **FlashJob** can subsequently be used for initiating an ECU reprogramming session. Optionally a **FlashSession** can be specified which will be used by the **FlashJob** for reprogramming (alternatively the **SetFlashSession** action can be used to assign a different **FlashSession** to an already existing **FlashJob** object).

CreateFlashJobByName is a **FlashJobTerm**. Its members have the following semantics:

— **<comChannel> : diag:ComChannelTerm [1]**

This specifies the **diag:ComChannel** object to which the to-be-created **FlashJob** belongs to and will be executed on when the **diag:ExecuteDiagService** action is used (cf. 7.6.4.3.1).

— **<name> : otx:StringTerm [1]**

Represents the name of the to-be-created **FlashJob**.

— **<session> : FlashSessionTerm [0..1]**

This optional element represents the **FlashSession** to be programmed by the **FlashJob**.

Throws:

— **UnknownTargetException**
If no **FlashJob** with the name provided by the **<name>** element exists.

10.6.2.3.5 CreateFlashJobBySemantic

This term shall create a new **FlashJob** for the specified **ComChannel** with the semantic attribute provided as an argument. The **FlashJob** can subsequently be used for initiating an ECU reprogramming session. Optionally a **FlashSession** can be specified which will be used by the **FlashJob** for reprogramming (alternatively the **SetFlashSession** action can be used to assign a different **FlashSession** to an already existing **FlashJob** object).

CreateFlashJobBySemantic is a **FlashJobTerm**. Its members have the following semantics:

— **<comChannel> : diag:ComChannelTerm [1]**

This specifies the **diag:ComChannel** object to which the to-be-created **FlashJob** belongs to and will be executed on when the **diag:ExecuteDiagService** action is used (cf. 7.6.4.3.1).

— **<semantic> : otx:StringTerm [1]**

This represents the semantic attribute of the to-be-created **FlashJob**.

— **<session> : FlashSessionTerm [0..1]**

This optional element represents the **FlashSession** to be programmed by the **FlashJob**.

Throws:

— **AmbiguousSemanticException**
In case there are none or more than one **FlashJob** present at the **ComChannel** with the semantic value specified by the **<semantic>** element.

10.6.3 Flash session related terms

10.6.3.1 Description

The following describes the flash session related terms of the OTX Flash extension.

10.6.3.2 Syntax

Figure 50 shows the syntax of the flash session related terms.

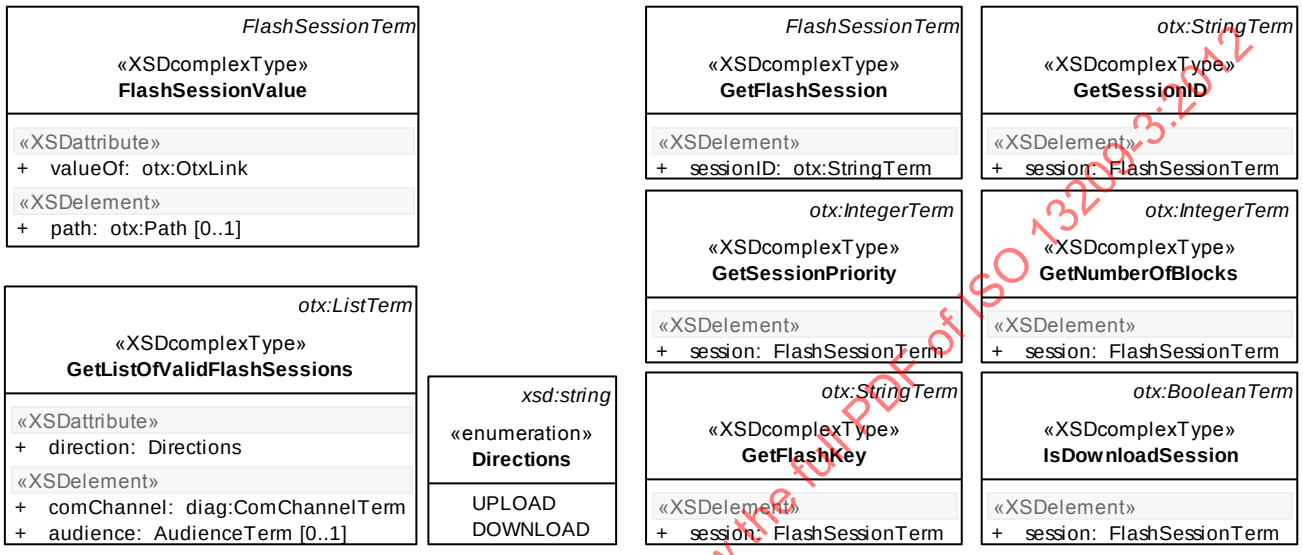

Figure 50 — Data model view: Flash session related terms

10.6.3.3 Semantics

10.6.3.3.1 FlashSessionTerm

The abstract type **`FlashSessionTerm`** is an **`otx:Term`**. It serves as a base for all concrete terms which return a **`FlashSession`**. It has no special members.

10.6.3.3.2 FlashSessionValue

This term returns the **`FlashSession`** stored in a **`FlashSession`** variable. For more information on value-terms and the syntax and semantics of the **`valueOf`** attribute and **`<path>`** element, please refer to Part 2 of ISO 13209.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

Throws:

— **`otx:OutOfBoundsException`**
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).

— **`otx:InvalidReferenceException`**
If the variable value is not valid (no value was assigned to the variable before).

10.6.3.3.3 GetListOfValidFlashSessions

The **GetListOfValidFlashSessions** term shall return an **otx:List** of **otx:String** items which identify the **FlashSessions** that are valid. The validity of a **FlashSession** shall be defined by rules which exist in the respective technological environment. For instance, in an ODX environment the **ExpectedIdents** shall be checked. In other environments the rules may differ.

IMPORTANT — GetListOfValidFlashSessions shall return the flash sessions in the order of their session priority. The highest-ranking FlashSession shall be the first item in the resulting List whereas the lowest-ranking shall be the last. For equally-ranked FlashSessions the order is unspecified.

NOTE In an ODX/MVCI based system, the session priority is a non-negative integer value assigned to a flash session, where a value of **0** represents the highest possible priority. For flash sessions without an explicit priority setting a default priority of **100** applies.

GetListOfValidFlashSessions is an **otx:ListTerm**. Its members have the following semantics:

— **direction : Directions={UPLOAD|DOWNLOAD} [1]**

This attribute defines which kind of **FlashSessions** shall be returned.

— **<comChannel> : diag:ComChannelTerm [1]**

This element defines a communication channel which is associated to the flash sessions. Please refer to Clause 6 (OTX DiagCom extension) for details on the **diag:ComChannelTerm** type.

— **<audience> : AudienceTerm [0..1]**

This optional element defines a filter on a special audience. Only flash sessions with the given audience shall be returned. If the attribute is omitted, no audience filtering shall be done. Please refer to 10.2.3.5 for information about the **Audience** enumeration.

10.6.3.3.4 GetFlashSession

The **GetFlashSession** term shall return a **FlashSession** handle which is identified by a session ID.

GetFlashSession is a **FlashSessionTerm**. Its members have the following semantics:

— **<sessionID> : otx:StringTerm [1]**

This element shall represent a unique identifier in the environment which is used for identifying a flash session.

Throws:

— **UnsupportedFormatException**
If the runtime system does not support the flash data file format.

10.6.3.3.5 GetSessionID

The **GetSessionID** term shall return the identifier of a flash session. The identifier is a string value.

NOTE In ODX/MVCI based systems, the returned ID string should correspond to the **SHORT-NAME** of the session.

GetSessionID is an **otx:StringTerm**. Its members have the following semantics:

— **<session> : FlashSessionTerm [1]**

This element shall represent the **FlashSession** to be used.

10.6.3.3.6 GetFlashKey

The **GetFlashKey** term shall return the key of a flash session. The key is a string value.

NOTE In ODX/MVCI based systems, the returned key should correspond to the **PARTNUMBER** of the session (**SESSION-DESC**).

GetFlashKey is an **otx:StringTerm**. Its members have the following semantics:

— **<session> : FlashSessionTerm [1]**

This element shall represent the **FlashSession** to be used.

10.6.3.3.7 GetSessionPriority

The **GetSessionPriority** term shall return the priority setting for a flash session. The resulting priority shall be represented by a non-negative integer value where **0** represents the highest possible priority. If no priority information is available for a flash session, a default value of **100** shall be returned.

NOTE In ODX/MVCI based systems, the flash session priority information is given by a non-negative integer value, where a value of **0** shall represent the highest possible priority. For proprietary systems using a different priority concept, it should nevertheless be possible to define a mapping between proprietary priorities and the priority values required by this standard.

GetSessionPriority is an **otx:IntegerTerm**. Its members have the following semantics:

— **<session> : FlashSessionTerm [1]**

This element shall represent the **FlashSession** to be used.

10.6.3.3.8 GetNumberOfBlocks

The **GetNumberOfBlocks** term shall return the number of blocks in a **FlashSession**. If no blocks exist, the return value shall be zero, otherwise it shall be a positive number.

GetNumberOfBlocks is an **otx:IntegerTerm**. Its members have the following semantics:

— **<session> : FlashSessionTerm [1]**

This element represents the **FlashSession** from which the number of blocks shall be returned.

10.6.3.3.9 IsDownloadSession

The **IsDownloadSession** term shall return **true** if and only if the flash session's direction is **DOWNLOAD**. If the session's direction is **UPLOAD**, **false** shall be returned.

IsDownloadSession is an **otx:BooleanTerm**. Its members have the following semantics:

— **<session> : FlashSessionTerm [1]**

This element shall represent the **FlashSession** from which the direction shall be determined.

10.6.3.4 Example

The example below shows the flash session related terms, embedded in assignment actions.

EXAMPLE Sample of OTX-file "FlashSessionRelatedTermsExample.otx"

```
<action id="a1">
  <specification>Get all download session for the after sales department</specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="ListVariable" name="AllSessions"/>
    <term xsi:type="flash:GetListOfValidFlashSessions" direction="DOWNLOAD">
      <flash:comChannel xsi:type="diag:ComChannelValue" valueOf="cc"/>
      <flash:audience xsi:type="flash:AudienceLiteral" value="AFTERSALES"/>
    </term>
  </realisation>
</action>

<action id="a2">
  <specification>Get the first session from a list</specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="flash:FlashSessionVariable" name="mySession"/>
    <term xsi:type="flash:GetFlashSession">
      <flash:sessionID xsi:type="StringValue" valueOf="AllSessions">
        <path>
          <stepByIndex xsi:type="IntegerLiteral" value="0"/>
        </path>
      </flash:sessionID>
    </term>
  </realisation>
</action>

<action id="a3">
  <specification>Get the session ID and write it into String variable</specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="StringVariable" name="SessionString"/>
    <term xsi:type="flash:GetSessionID">
      <flash:session xsi:type="flash:FlashSessionValue" valueOf="mySession"/>
    </term>
  </realisation>
</action>

<action id="a4">
  <specification>Get the number of blocks in session</specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="IntegerVariable" name="blocks"/>
    <term xsi:type="flash:GetNumberOfBlocks">
      <flash:session xsi:type="flash:FlashSessionValue" valueOf="mySession"/>
    </term>
  </realisation>
</action>
```

10.6.4 Flash block related terms

10.6.4.1 Description

The following describes all terms of the OTX Flash extensions by which diverse information on flash blocks can be retrieved.

10.6.4.2 Syntax

Figure 51 shows the syntax of all flash block related terms.

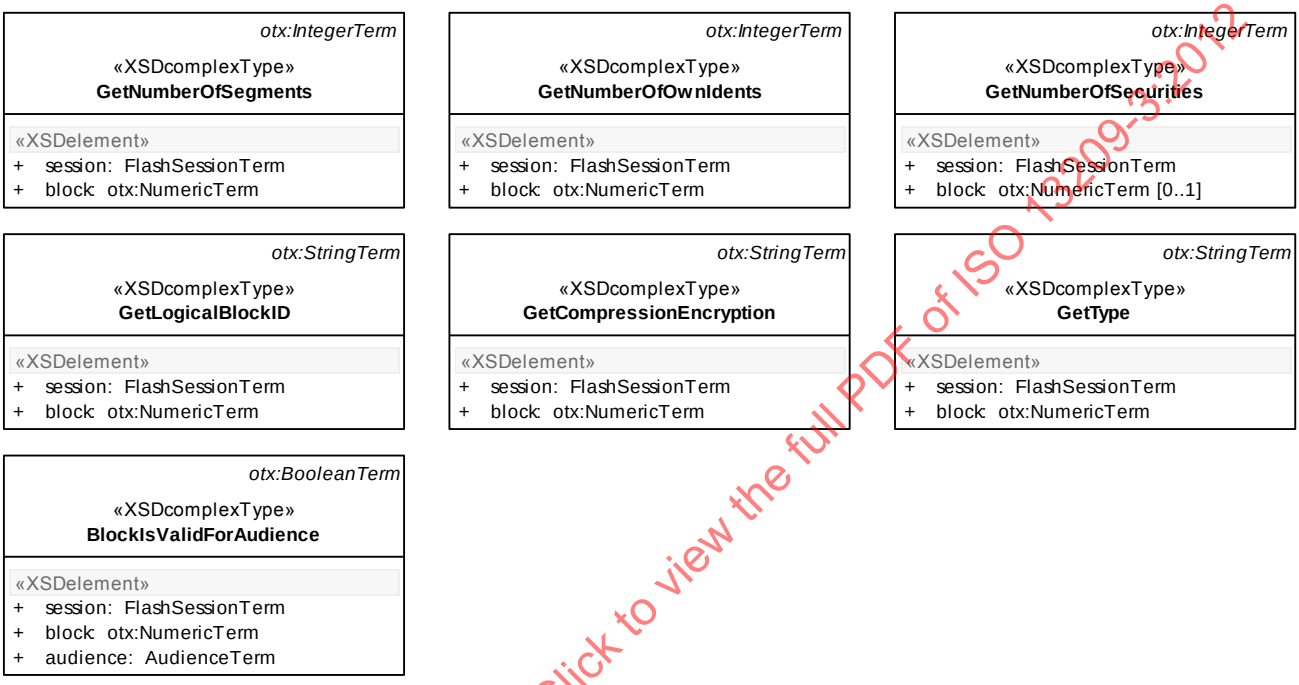

Figure 51 — Data model view: Flash block related terms

10.6.4.3 Semantics

10.6.4.3.1 GetNumberOfSegments

The **`GetNumberOfSegments`** term shall return the number of data segments in a block. If no segments exist, the return value shall be zero, otherwise it shall be a positive number.

`GetNumberOfSegments` is an **`otx:IntegerTerm`**. Its members have the following semantics:

— **`<session> : FlashSessionTerm [1]`**

This element represents the **`FlashSession`** in which the block of interest resides.

— **`<block> : otx:NumericTerm [1]`**

This element provides the number of the block from which the number of data segments shall be retrieved. **`Float`** values shall be truncated.

Throws:

— **`otx:OutOfBoundsException`**
If there was no block found with the requested number.

10.6.4.3.2 GetNumberOfOwnIdents

The **GetNumberOfOwnIdents** term shall return the number of required and to-be-fulfilled identifications of a block.

GetNumberOfOwnIdents is an **otx:IntegerTerm**. Its members have the following semantics:

— **<session> : FlashSessionTerm [1]**

This element represents the **FlashSession** in which the block of interest resides.

— **<block> : otx:NumericTerm [1]**

This element provides the number of the block from which the number of identifications shall be retrieved. **Float** values shall be truncated.

Throws:

— **otx:OutOfBoundsException**
If there was no block found with the requested number.

10.6.4.3.3 GetNumberOfSecurities

The **GetNumberOfSecurities** term shall return the number of security information of a block or session.

GetNumberOfSecurities is an **otx:IntegerTerm**. Its members have the following semantics:

— **<session> : FlashSessionTerm [1]**

This element represents the **FlashSession** of interest.

— **<block> : otx:NumericTerm [0..1]**

This optional element defines the block from which the number of security information shall be retrieved. If the **<block>** element is omitted, the term returns the number of securities defined for the flash session. **Float** values shall be truncated.

Throws:

— **otx:OutOfBoundsException**
If there was no block found with the requested number.

10.6.4.3.4 GetLogicalBlockID

The **GetLogicalBlockID** term shall return the unique string identification of a block.

NOTE In ODX/MVCI based systems, the returned ID string should correspond to the SHORT-NAME of the block.

GetLogicalBlockID is an **otx:StringTerm**. Its members have the following semantics:

— **<session> : FlashSessionTerm [1]**

This element represents the **FlashSession** in which the block resides.

— **<block> : otx:NumericTerm [1]**

This element represents the block number. **Float** values shall be truncated.

Throws:

— **otx:OutOfBoundsException**
If there was no block found with the requested number.

10.6.4.3.5 GetCompressionEncryption

The **GetCompressionEncryption** term shall return the compression and encryption information of a block (e.g. AES encryption, LZSS compression, etc.).

GetCompressionEncryption is an **otx:StringTerm**. Its members have the following semantics:

— **<session> : FlashSessionTerm [1]**

This element represents the **FlashSession** in which the block resides.

— **<block> : otx:NumericTerm [1]**

This element represents the block number. **Float** values shall be truncated.

Throws:

— **otx:OutOfBoundsException**
If there was no block found with the requested number.

10.6.4.3.6 GetType

The **GetType** term shall return the type of a block. The type information indicates whether a block is used for data or for program code.

GetType is an **otx:StringTerm**. Its members have the following semantics:

— **<session> : FlashSessionTerm [1]**

This element represents the **FlashSession** in which the block resides.

— **<block> : otx:NumericTerm [1]**

This element represents the block number. **Float** values shall be truncated.

Throws:

— **otx:OutOfBoundsException**
If there was no block found with the requested number.

10.6.4.3.7 BlockIsValidForAudience

The **BlockIsValidForAudience** term shall return **true** if and only if a block is valid for a given audience.

BlockIsValidForAudience is an **otx:BooleanTerm**. Its members have the following semantics:

— **<session> : FlashSessionTerm [1]**

This element represents the **FlashSession** in which the block resides.

— **<block> : otx:NumericTerm [1]**

This element represents the block number. **Float** values shall be truncated.

— **<audience> : AudienceTerm [1]**

This attribute defines which audience shall be used for the check. Please refer to 10.2.3.5 for information about the **Audience** enumeration.

Throws:

— **otx:OutOfBoundsException**
If there was no block found with the requested number.

10.6.4.4 Example

The example below shows the flash block related terms, embedded in assignment actions.

EXAMPLE Sample of OTX-file "FlashBlockRelatedTermsExample.otx"

```
<action id="a1">
  <specification>Get the number of segments in the first block</specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="IntegerVariable" name="segments"/>
    <term xsi:type="flash:GetNumberOfSegments">
      <flash:session xsi:type="flash:FlashSessionValue" valueOf="mySession"/>
      <flash:block xsi:type="IntegerLiteral" value="0"/>
    </term>
  </realisation>
</action>

<action id="a2">
  <specification>Get the number of own idents of block 0</specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="IntegerVariable" name="ownIdents"/>
    <term xsi:type="flash:GetNumberOfOwnIdents">
      <flash:session xsi:type="flash:FlashSessionValue" valueOf="mySession"/>
      <flash:block xsi:type="IntegerLiteral" value="0"/>
    </term>
  </realisation>
</action>

<action id="a3">
  <specification>Get the number of securities of the session</specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="IntegerVariable" name="securities"/>
    <term xsi:type="flash:GetNumberOfSecurities">
      <flash:session xsi:type="flash:FlashSessionValue" valueOf="mySession"/>
      <!-- omitted block signals session securities -->
    </term>
  </realisation>
</action>

<action id="a4">
  <specification>Get identification of a block</specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="IntegerVariable" name="identification"/>
    <term xsi:type="flash:GetLogicalBlockID">
      <flash:session xsi:type="flash:FlashSessionValue" valueOf="mySession"/>
      <flash:block xsi:type="IntegerLiteral" value="0"/>
    </term>
  </realisation>
</action>

<action id="a5">
  <specification>Get the compression and encryption method of the block 0</specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="StringVariable" name="format"/>
    <term xsi:type="flash:GetCompressionEncryption">
      <flash:session xsi:type="flash:FlashSessionValue" valueOf="mySession"/>
      <flash:block xsi:type="IntegerLiteral" value="0"/>
    </term>
  </realisation>
</action>

<action id="a6">
  <specification>Get Type of Block 0</specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="StringVariable" name="type"/>
    <term xsi:type="flash:GetType">
      <flash:session xsi:type="flash:FlashSessionValue" valueOf="mySession"/>
      <flash:block xsi:type="IntegerLiteral" value="0"/>
    </term>
  </realisation>
</action>
```

```
<action id="a7">
  <specification>checks if block 0 is valid for audience "AFTERSALES"</specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="BooleanVariable" name="isValid"/>
    <term xsi:type="flash:BlockIsValidForAudience">
      <flash:session xsi:type="flash:FlashSessionValue" valueOf="mySession"/>
      <flash:block xsi:type="IntegerLiteral" value="0"/>
      <flash:audience xsi:type="flash:AudienceLiteral" value="AFTERSALES"/>
    </term>
  </realisation>
</action>
```

10.6.5 Flash block segment related terms

10.6.5.1 Description

The following describes terms for retrieving information on flash block segments.

10.6.5.2 Syntax

Figure 52 shows the syntax of all flash block segment related terms.

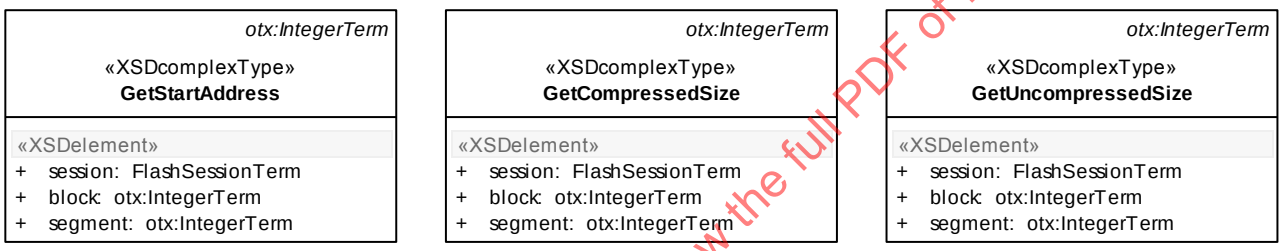

Figure 52 — Data model view: Flash block segment related terms

10.6.5.3 Semantics

10.6.5.3.1 GetStartAddress

The **`GetStartAddress`** term shall return the start address of a segment.

`GetStartAddress` is an **`otx:IntegerTerm`**. Its members have the following semantics:

— **`<session> : FlashSessionTerm [1]`**

This element represents the **`FlashSession`** in which the block containing the segment resides.

— **`<block> : otx:NumericTerm [1]`**

This element represents the block in which the segment resides. **`Float`** values shall be truncated.

— **`<segment> : otx:NumericTerm [1]`**

This element provides the segment number. **`Float`** values shall be truncated.

Throws:

— **`otx:OutOfBoundsException`**
If there was no block or segment found with the requested number.

10.6.5.3.2 GetCompressedSize

The **GetCompressedSize** shall return the number of bytes constituting the compressed data contained by a segment.

GetCompressedSize is an **otx:IntegerTerm**. Its members have the following semantics:

— **<session> : FlashSessionTerm [1]**

This element represents the **FlashSession** in which the block containing the segment resides.

— **<block> : otx:NumericTerm [1]**

This element represents the block in which the segment resides. **Float** values shall be truncated.

— ** : otx:NumericTerm [1]**

This element provides the segment number. **Float** values shall be truncated.

Throws:

— **otx:OutOfBoundsException**
If there was no block or segment found with the requested number.

10.6.5.3.3 GetUncompressedSize

The **GetUncompressedSize** shall return the number of bytes constituting the uncompressed data contained by a segment.

GetUncompressedSize is an **otx:IntegerTerm**. Its members have the following semantics:

— **<session> : FlashSessionTerm [1]**

This element represents the **FlashSession** in which the block containing the segment resides.

— **<block> : otx:NumericTerm [1]**

This element represents the block in which the segment resides. **Float** values shall be truncated.

— ** : otx:NumericTerm [1]**

This element provides the segment number. **Float** values shall be truncated.

Throws:

— **otx:OutOfBoundsException**
If there was no block or segment found with the requested number.

10.6.5.4 Example

The example below shows the flash block segment related terms, embedded in assignment actions.

EXAMPLE Sample of OTX-file "FlashSegmentRelatedTermsExample.otx"

```
<action id="a1">
  <specification>Get start address of segment</specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="IntegerVariable" name="startAddress"/>
    <term xsi:type="flash:GetStartAddress">
      <flash:session xsi:type="flash:FlashSessionValue" valueOf="mySession"/>
      <flash:block xsi:type="IntegerLiteral" value="0"/>
      <flash:segment xsi:type="IntegerLiteral" value="0"/>
    </term>
  </realisation>
</action>

<action id="a2">
  <specification>Get the compressed size of the Block</specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="IntegerVariable" name="compressedSize"/>
    <term xsi:type="flash:GetCompressedSize">
      <flash:session xsi:type="flash:FlashSessionValue" valueOf="mySession"/>
      <flash:block xsi:type="IntegerLiteral" value="0"/>
      <flash:segment xsi:type="IntegerLiteral" value="0"/>
    </term>
  </realisation>
</action>
```

10.6.6 Security related terms

10.6.6.1 Description

The following describes the security related terms of the OTX Flash extension.

10.6.6.2 Syntax

Figure 53 shows the syntax of the security related terms.

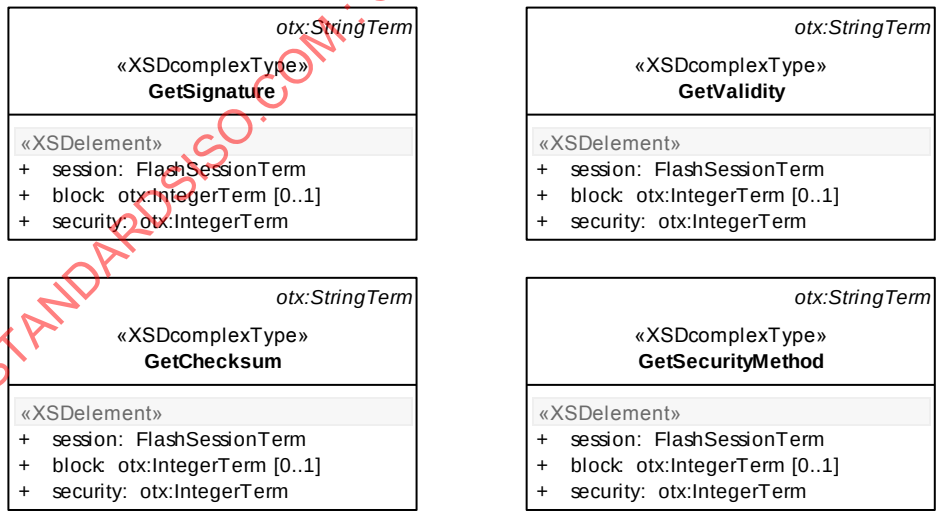

Figure 53 — Data model view: Security related terms

10.6.6.3 Semantics

10.6.6.3.1 GetSignature

The **GetSignature** shall return the signature information of a block or a session.

GetSignature is an **otx:StringTerm**. Its members have the following semantics:

— **<session> : FlashSessionTerm [1]**

This element represents the **FlashSession** in which the block resides.

— **<block> : otx:NumericTerm [0..1]**

This element represents the number of the block whose signature shall be returned. If the **<block>** element is omitted, the signature of the flash session shall be returned instead. **Float** values shall be truncated.

— **<security> : otx:NumericTerm [1]**

This element defines the number of the security on which the term execution is based. **Float** values shall be truncated.

Throws:

— **otx:OutOfBoundsException**
If there was no block found with the requested number or the security is not defined.

10.6.6.3.2 GetValidity

The **GetValidity** term shall return the validity information of a block or a session.

GetValidity is an **otx:StringTerm**. Its members have the following semantics:

— **<session> : FlashSessionTerm [1]**

This element represents the **FlashSession** in which the block resides.

— **<block> : otx:NumericTerm [0..1]**

This element represents the block number. If the **<block>** element is omitted, the security information of the flash session shall be returned instead. **Float** values shall be truncated.

— **<security> : otx:NumericTerm [1]**

This element defines the number of the security on which the term execution is based. **Float** values shall be truncated.

Throws:

— **otx:OutOfBoundsException**
If there was no block found with the requested number or the security is not defined.

10.6.6.3.3 GetChecksum

The **GetChecksum** term shall return the checksum information of a block or a session.

GetChecksum is an **otx:StringTerm**. Its members have the following semantics:

— **<session> : FlashSessionTerm [1]**

This element represents the **FlashSession** in which the block resides.

— **<block> : otx:NumericTerm [0..1]**

This element represents the number of the block whose checksum shall be returned. If the **<block>** element is omitted, the checksum of the flash session shall be returned instead. **Float** values shall be truncated.

— **<security> : otx:NumericTerm [1]**

This element defines the number of the security on which the term execution is based. **Float** values shall be truncated.

Throws:

— **otx:OutOfBoundsException**
If there was no block found with the requested number or the security is not defined.

10.6.6.3.4 GetSecurityMethod

The **GetSecurityMethod** shall return the security method information of a block or a session.

GetSecurityMethod is an **otx:StringTerm**. Its members have the following semantics:

— **<session> : FlashSessionTerm [1]**

This element represents the **FlashSession** in which the block resides.

— **<block> : NumericTerm [0..1]**

This element represents number of the block whose security method shall be returned. If the **<block>** element is omitted, the security method of the flash session shall be returned instead. **Float** values shall be truncated.

— **<security> : NumericTerm [1]**

This element defines the number of the security on which the term execution is based. **Float** values shall be truncated.

Throws:

— **otx:OutOfBoundsException**
If there was no block found with the requested number or the security is not defined.

10.6.6.4 Example

The example below shows the security related terms, embedded in assignment actions.

EXAMPLE Sample of OTX-file "FlashSecurityRelatedTermsExample.otx"

```
<action id="a1">
  <specification>Get signature 0 of block 0</specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="StringVariable" name="signature"/>
    <term xsi:type="flash:GetSignature">
      <flash:session xsi:type="flash:FlashSessionValue" valueOf="mySession"/>
      <flash:block xsi:type="IntegerLiteral" value="0"/>
      <flash:security xsi:type="IntegerLiteral" value="0"/>
    </term>
  </realisation>
</action>

<action id="a2">
  <specification>Get validity 0 of block 0</specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="StringVariable" name="validity"/>
    <term xsi:type="flash:GetValidity">
      <flash:session xsi:type="flash:FlashSessionValue" valueOf="mySession"/>
      <flash:block xsi:type="IntegerLiteral" value="0"/>
      <flash:security xsi:type="IntegerLiteral" value="0"/>
    </term>
  </realisation>
</action>

<action id="a3">
  <specification>Get checksum 0 of block 0</specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="StringVariable" name="checksum"/>
    <term xsi:type="flash:GetChecksum">
      <flash:session xsi:type="flash:FlashSessionValue" valueOf="mySession"/>
      <flash:block xsi:type="IntegerLiteral" value="0"/>
      <flash:security xsi:type="IntegerLiteral" value="0"/>
    </term>
  </realisation>
</action>

<action id="a4">
  <specification>Get security method 0 of block 0</specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="StringVariable" name="method"/>
    <term xsi:type="flash:GetSecurityMethod">
      <flash:session xsi:type="flash:FlashSessionValue" valueOf="mySession"/>
      <flash:block xsi:type="IntegerLiteral" value="0"/>
      <flash:security xsi:type="IntegerLiteral" value="0"/>
    </term>
  </realisation>
</action>
```

10.6.7 Own ident related terms

10.6.7.1 Description

The following describes the own ident related terms of the OTX Flash extension.

10.6.7.2 Syntax

Figure 54 shows the syntax of the own ident related terms.

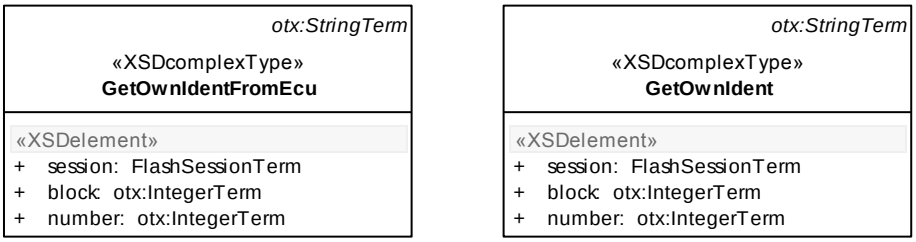

Figure 54 — Data model view: Own ident related terms

10.6.7.3 Semantics

10.6.7.3.1 GetOwnIdentFromEcu

The **`GetOwnIdentFromEcu`** term shall return an identification string which shall be read from an ECU. The environment shall store the information to access this string.

`GetOwnIdentFromEcu` is an **`otx:StringTerm`**. Its members have the following semantics:

— **`<session> : FlashSessionTerm [1]`**

This element represents the **`FlashSession`** in which the block resides.

— **`<block> : otx:NumericTerm [1]`**

This element represents the block number. **`Float`** values shall be truncated.

— **`<number> : otx:NumericTerm [1]`**

This element represents the own identification number. **`Float`** values shall be truncated.

Throws:

— **`otx:OutOfBoundsException`**
If there was no block or own ident number found with the requested number.

10.6.7.3.2 GetOwnIdent

The **`GetOwnIdent`** term shall return an identification string which is read from the download data.

`GetOwnIdent` is an **`otx:StringTerm`**. Its members have the following semantics:

— **`<session> : FlashSessionTerm [1]`**

This element represents the **`FlashSession`** in which the block resides.

— **`<block> : otx:NumericTerm [1]`**

This element represents the block number. **`Float`** values shall be truncated.

— **`<number> : otx:NumericTerm [1]`**

This element represents the own identification number. **`Float`** values shall be truncated.

Throws:

— **otx:OutOfBoundsException**
If there was no block or own ident number found with the requested number.

10.6.7.4 Example

The example below shows the own ident related terms, embedded in assignment actions.

EXAMPLE Sample of OTX-file "FlashOwnIdentRelatedTermsExample.otx"

```
<action id="a1">
  <specification>Get the own ident 0 of block 0</specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="BooleanVariable" name="OwnIdentEcu"/>
    <term xsi:type="flash:GetOwnIdentFromEcu">
      <flash:session xsi:type="flash:FlashSessionValue" valueOf="mySession"/>
      <flash:block xsi:type="IntegerLiteral" value="0"/>
      <flash:number xsi:type="IntegerLiteral" value="0"/>
    </term>
  </realisation>
</action>

<action id="a2">
  <specification>Get the identification 0 of block 0</specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="StringVariable" name="OwnIdent"/>
    <term xsi:type="flash:GetOwnIdent">
      <flash:session xsi:type="flash:FlashSessionValue" valueOf="mySession"/>
      <flash:block xsi:type="IntegerLiteral" value="0"/>
      <flash:number xsi:type="IntegerLiteral" value="0"/>
    </term>
  </realisation>
</action>
```

10.6.8 Enumeration related terms

10.6.8.1 Description

The following describes the terms related to the enumerations **FlashFileFormat** and **Audience**, as specified in 10.2.

10.6.8.2 Syntax

Figure 55 shows the syntax of the enumeration related terms.

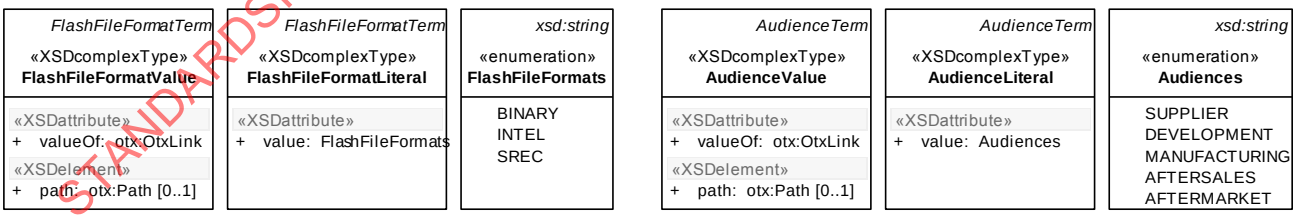

Figure 55 — Data model view: Enumeration related terms

10.6.8.3 Semantics

10.6.8.3.1 FlashFileFormatTerm

The abstract type **FlashFileFormatTerm** is an **otx:SimpleTerm**. It serves as a base for all concrete terms which return a **FlashFileFormat**. It has no special members.

10.6.8.3.2 FlashFileFormatValue

This term returns the **FlashFileFormat** stored in a **FlashFileFormat** variable. For more information on value-terms and the syntax and semantics of the **valueOf** attribute and **<path>** element, please refer to Part 2 of ISO 13209.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

Throws:

— **otx:OutOfBoundsException**
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).

10.6.8.3.3 FlashFileFormatLiteral

This term shall return a **FlashFileFormat** value (see 10.2.3.4) from a hard-coded literal.

FlashFileFormatLiteral is a **FlashFileFormatTerm**. Its members have the following semantics:

— **value : FlashFileFormats={BINARY|SREC|INTEL} [1]**

This attribute shall contain one of the values defined in the **FlashFileFormats** enumeration.

10.6.8.3.4 AudienceTerm

The abstract type **AudienceTerm** is an **otx:SimpleTerm**. It serves as a base for all concrete terms which return an **Audience**. It has no special members.

10.6.8.3.5 AudienceValue

This term returns the **Audience** stored in an **Audience** variable. For more information on value-terms and the syntax and semantics of the **valueOf** attribute and **<path>** element, please refer to Part 2 of ISO 13209.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

Throws:

— **otx:OutOfBoundsException**
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).

10.6.8.3.6 AudienceLiteral

This term shall return an **Audience** value (see 10.2.3.4) from a hard-coded literal.

AudienceLiteral is an **AudienceTerm**. Its members have the following semantics:

— **value :**
Audiences={SUPPLIER|DEVELOPMENT|MANUFACTURING|AFTERSALES|AFTERMARKET} [1]

This attribute shall contain one of the values defined in the **Audiences** enumeration.

11 OTX HMI extension

11.1 Introduction

11.1.1 General

The Human Machine Interface (HMI) extension provides access to data types, terms and actions for interacting with the user through the display of graphical screens, as well as through additional input and output devices such as keyboards etc.

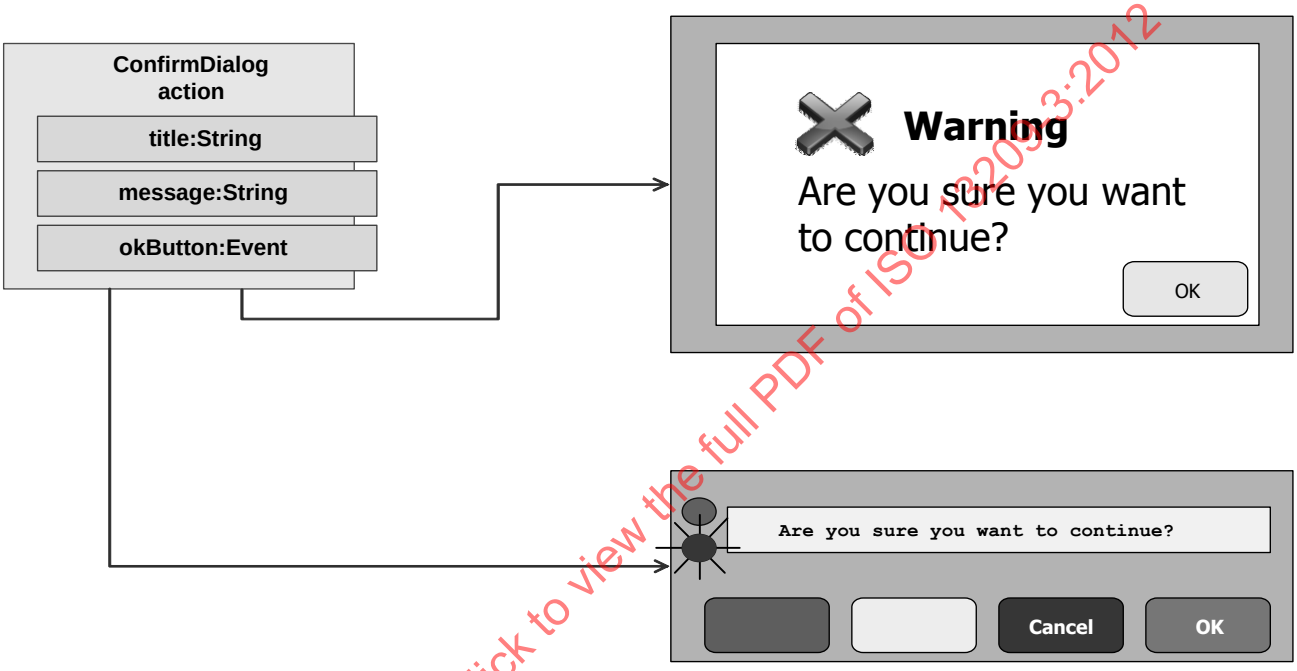

Figure 56 — Different hardware configurations

Due to the multiple possible variations on runtime systems, and the fact that some of the target runtime systems do not even have a display (see Figure 56), one of the design goals of the HMI extension was to abstract the details regarding the layout of the screens on the system, concentrating instead on the communication aspects between the test sequence and the user interface. To achieve this goal, there are two ways to operate screens: A set of basic dialogs that all systems should provide and customizable screens that allow extra flexibility.

11.1.2 Dialogs

The basic dialogs are used to cover the most elemental use cases, such as showing a warning to the user or asking for simple user input. Dialogs are always modal: It is assumed that the runtime system will pause execution of the test flow when reaching one of these dialog actions and will provide a way for the user to dismiss the dialog (normally with an "ok" or "close" button).

Dialogs do not assume any special graphical functionality and shall be supported by all test application systems. It is possible to implement them in systems without graphical display by using LEDs and reading static buttons on the device. In this case message information would be ignored.

11.1.3 Custom screens

Custom screens define an interface to a screen that is externally created. The layout and functionality of the screen itself is not defined in the OTX file and is only referenced by name, as shown in Figure 57. The call is similar to a standard procedure call, and it only defines ways to pass parameters in and out from the screen.

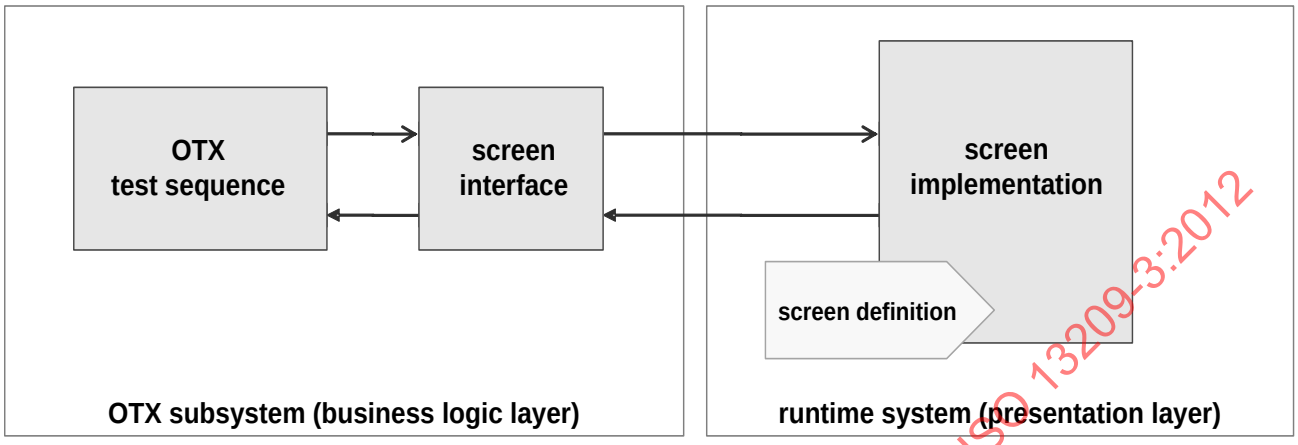

Figure 57 — Separation of concerns

Custom screens are non-modal. The execution of the test sequence continues after the screen is displayed in the runtime system. For this effect, there are actions and terms that help control the flow of the screen: A screen event source term by which execution can be stopped until a screen event has been received and an action to close the screen.

Screen implementation is up to the client: The client can either have a graphical user interface (UI), a console based application or a button layout on physical hardware. The screen interface provides a level of abstraction that decouples the description of the screen from the test sequence.

A screen is connected directly to the model of the test sequence. All input values to the screen are references to variables and all out parameteres are assigned to variables.

The update of the screen must be performed automatically by the runtime system. When one of the referred variables is updated, the runtime system must update the display on the screen automatically. It is assumed that the update will happen asynchronously in a UI thread and that the execution of the main sequence will not be interrupted.

The screen can communicate events back to the system by using screen event objects. These events can indicate if any of a screen's output parameters has changed, or if the user has performed any other operation on the screen such as closing, minimizing or dismissing. The usage of the event mechanism allows building applications with complex user interaction, without transmitting specific look and feel from the target applications. To monitor changes in the screen parameters, it is possible to use the terms defined in the OTX EventHandling extension (see Clause 8, **`MonitorChangeEventSource`** term).

Custom screens should be handled by a separate thread by the runtime system. As such, when handling events from the screen (i.e. when waiting for user interaction) it is often advisable to create a parallel lane in the OTX sequence dedicated to listen for these events with a **`WaitForEvent`** action in case that additional processing tasks need to be performed.

11.1.4 Custom screen usage example

Figure 58 provides a typical usage of the custom screens, represented using an UML activity diagram.

The use case is the following:

— Present a screen that displays a list of values measured with an "exit" button

— Read values from an electronic control unit periodically and refresh the screen

— When the user decides to exit the application, then stop reading values

To achieve this, two different ways are shown in the example:

The sequence shown to the left represents a solution for simple cases where fine-grained event evaluation is not necessary. After opening the screen, there is a loop for reading out ECU values which stops as soon as the screen was closed (cf. `ScreenIsOpen` term, as specified in 11.6.3.3).

The sequence to the right shows a solution which opens up the possibility to fine-grained handling of different kinds of events which may happen on the screen. After opening the screen, there are two parallel lanes. In one lane, the test sequence continiously reads new values from the ECU as long as a "finish" flag is false. In the second lane, a `WaitForEvent` action is used to react on the events fired by the screen. Once a screen closed event is received, then the sequence terminates (cf. `IsScreenClosedEvent` term in 11.6.3.4). Other event types from the screen might be processed in the event loop also, which is not exemplified here.

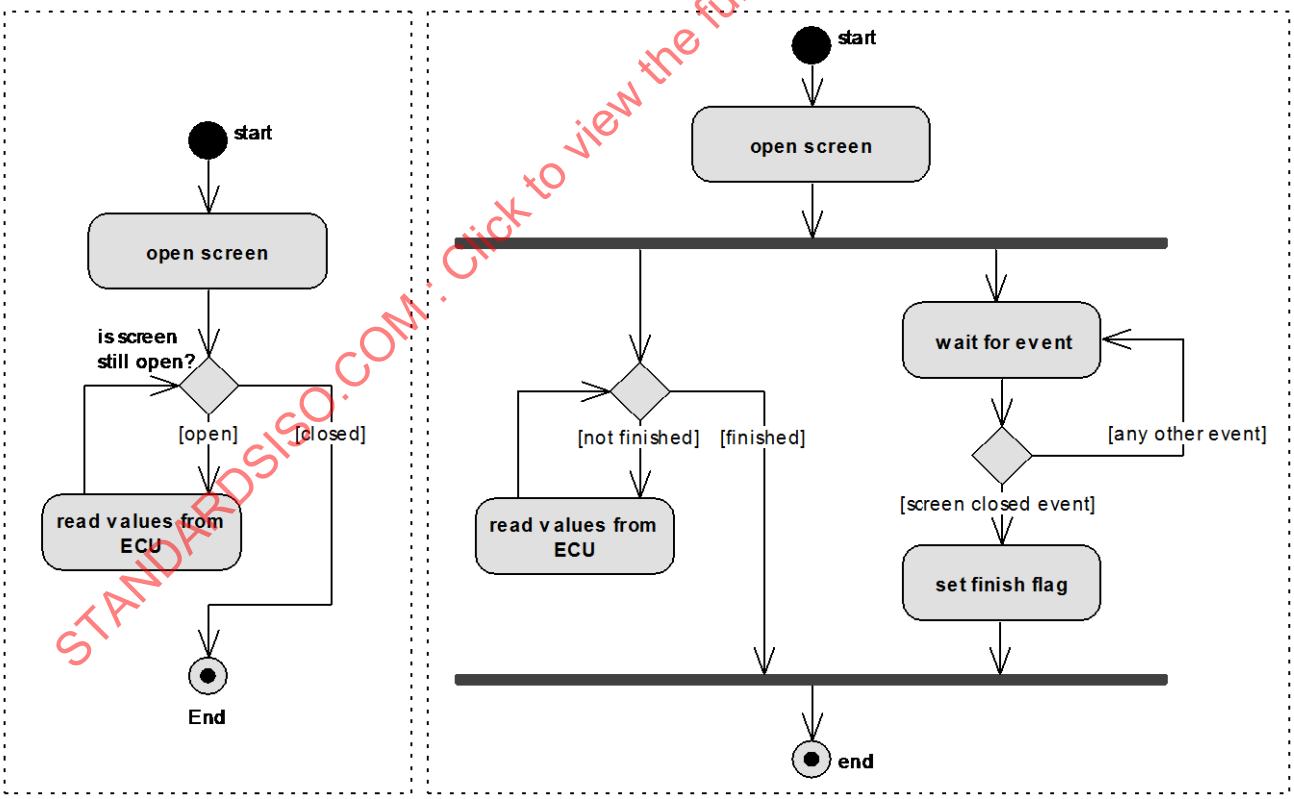

Figure 58 — Custom screen usage example

In both solutions, the "exit"-button is not controlled by the OTX sequence – the test applications presentation layer is responsible for closing the screen as soon as the "exit" button is pressed (note that there is also an explicit `CloseScreen` action, cf. 11.5.3.3.2). Furthermore, the update of the ECU values on the screen is automatic, due to the fact that screens can be connected directly to variables of the OTX sequence. In the given example, reads are interrupted cleanly, as once the last read is complete the sequence will finish.

11.2 Data types

11.2.1 Overview

The OTX HMI extension introduces the **`Screen`** data type required for the custom screen handling as well as the **`MessageType`** and **`ConfirmationType`** enumeration types used for dialogs.

11.2.2 Syntax

The syntax of the datatype declarations of the OTX HMI extension is shown in Figure 59.

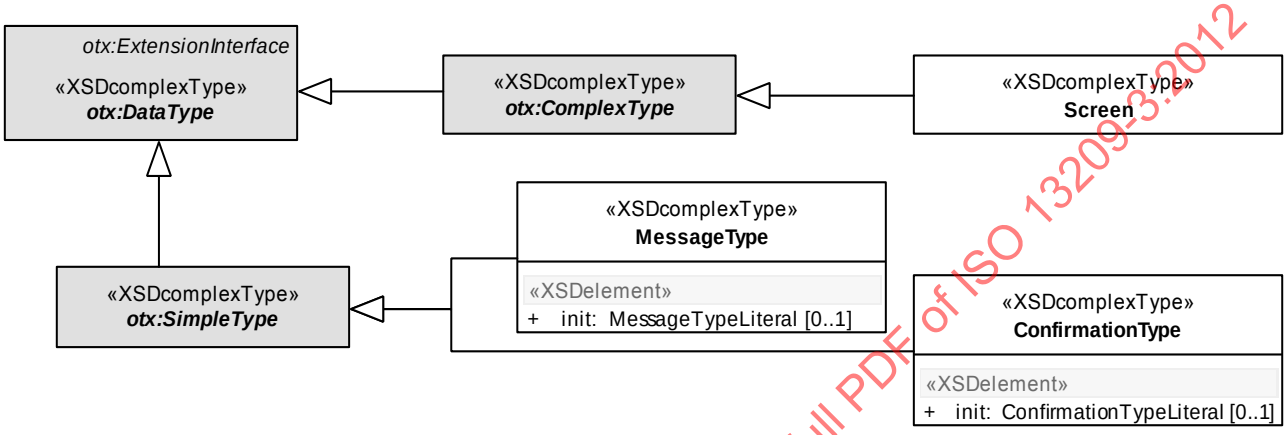

Figure 59 — Data model view: HMI data types

11.2.3 Semantics

11.2.3.1 General

The data types in the OTX HMI extension are based on **`otx:ComplexType`** and on **`otx:SimpleType`**.

11.2.3.2 Screen

The **`Screen`** data type is a handle to a complex screen resource on the runtime system. **`Screen`** handles represent an interface through which an OTX sequence can display data and receive user input. The current status of a screen can be checked by using the accessor terms associated to the **`Screen`** data type.

Since screens are also sources of screen closed events, they can be used as an argument of the term **`ScreenCloseEventSource`**, as specified 11.6.3.3 (cf. Clause 8, OTX EventHandling extension).

Since **`Screen`** has no initialisation parts, a **`Screen`** can not be declared constant.

NOTE It is an explicit design goal of the OTX HMI extension not to make assumptions regarding the layout, positioning or visualization style of a screen in a specific test application. These presentation layer details are left to the runtime systems.

11.2.3.3 MessageType

`MessageType` is an enumeration type describing the characteristics of a message shown in a **`ConfirmDialog`**. The type of message also controls which buttons are available in a **`ConfirmDialog`** (see 11.5.2.3.2).

The list of allowed enumeration values is defined as follows:

— **INFO**: Displayed message is just for information

— **WARNING**: Displayed message is a warning

— **ERROR**: Displayed message describes an error

— **YESNO_QUESTION:** Displayed message represents a question answerable by "yes" or "no"

— **YESNOCANCEL_QUESTION**: Displayed message is a question which does not require a response

IMPORTANT — `MessageType` values may occur as operands of comparisons (cf. Part 2 of ISO 13209, relational operations). For this case, the following order relation shall apply:
`INFO < WARNING < ERROR < YESNO_QUESTION < YESNOCANCEL_QUESTION`.

IMPORTANT — When applying `otx:ToString` on a `MessageType` value, the resulting string shall be the name of the enumeration value, e.g. `otx:ToString(INFO)="INFO"`. Furthermore, applying `otx:ToInteger` shall return the index of the value in the `MessageTypes` enumeration (smallest index is `0`). The behaviour is undefined for other conversion terms (cf. Part 2 of ISO 13209).

`MessageType` is an **`otx:SimpleType`**. Its members have the following semantics:

— **`<init> : MessageTypeLiteral [0..1]`**

This optional element stands for the hard-coded initialisation value of the identifier at declaration time.

— **`value : MessageTypes={INFO|WARNING|ERROR|YESNO_QUESTION|YESNOCANCEL_QUESTION} [1]`**

This attribute shall contain one of the values defined in the **`MessageTypes`** enumeration.

IMPORTANT — If the `MessageType` declaration is not explicitly initialized (omitted `<init>` element), the default value shall be `INFO`.

11.2.3.4 ConfirmationType

`ConfirmationType` is an enumeration type describing the button-choice of a user dismissing a **`ConfirmDialog`** (see 11.5.2.3.2). The information may later be used to find out which button was clicked for confirmation of the dialog.

The list of allowed enumeration values is defined as follows:

— **YES**: Confirmation by "Yes" button or "OK" button

— **NO**: Confirmation by "No" button

— **CANCEL**: Confirmation by "Cancel" button

IMPORTANT — `ConfirmationType` values may occur as operands of comparisons (cf. Part 2 of ISO 13209, relational operations). For this case, the following order relation shall apply:
`YES < NO < CANCEL`.

IMPORTANT — When applying `otx:ToString` on a `ConfirmationType` value, the resulting string shall be the name of the enumeration value, e.g. `otx:ToString(YES)="YES"`. Furthermore, applying `otx:ToInteger` shall return the index of the value in the `ConfirmationTypes` enumeration (smallest index is `0`). The behaviour is undefined for other conversion terms (cf. Part 2 of ISO 13209).

ConfirmationType is an **otx:SimpleType**. Its members have the following semantics:

— **<init> : ConfirmationTypeLiteral [0..1]**

This optional element stands for the hard-coded initialisation value of the identifier at declaration time.

— **value : ConfirmationType = {YES|NO|CANCEL} [1]**

This attribute shall contain one of the values defined in the **ConfirmationTypes** enumeration.

IMPORTANT — If the ConfirmationType declaration is not explicitly initialized (omitted <init> element), the default value shall be YES.

11.3 Exceptions

11.3.1 Overview

All exceptions specified in the following are derived from the **otx:Exception** type as defined by Part 2 of ISO 13209. They represent the full set of exceptions added by the OTX HMI extension.

11.3.2 Syntax

The syntax of all OTX HMI exception type declarations is shown in Figure 60.

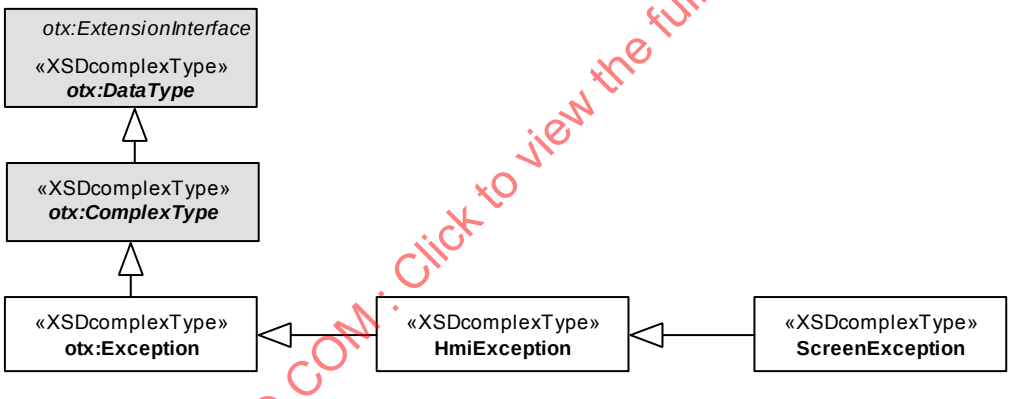

Figure 60 — Data model view: HMI exceptions

11.3.3 Semantics

11.3.3.1 General

Since all OTX HMI exception types are implicit exceptions whithout initialisation parts, they can not be declared constant.

11.3.3.2 HmiException

The **HmiComException** is the super class for all exceptions in the HMI extension. An **HmiException** shall be used in case the more specific exception types described in the remainder of this section do not apply to the problem at hand.

11.3.3.3 ScreenException

A **ScreenException** will be thrown by the runtime system in case that there are problems while processing custom screens.

Situations where a **`ScreenException`** will be thrown include e.g.

— non-existing screen definition in the runtime

— parameters of the called screen do not match to the signature of the screen

— errors while updating the screen

11.4 Variable access

11.4.1 Overview

As specified in Part 2 of ISO 13209, OTX extensions shall define a variable access type for each datatype they define (exception types inclusively). All variable access types are derived from the OTX Core **`Variable`** extension interface. The following specifies all variable access types defined for the HMI extension.

11.4.2 Syntax

Figure 61 shows the syntax of the HMI extension's variable access types.

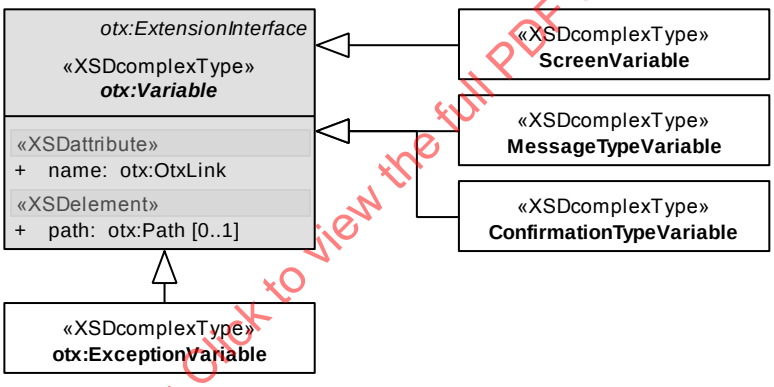

Figure 61 — Data model view: HMI variable access types

11.4.3 Semantics

The general semantics for all variable access types shall apply. Please refer to Part 2 of ISO 13209 for details.

11.5 Actions

11.5.1 Overview

All of the elements described in the following extend the **`otx:ActionRealisation`** extension interface as defined by Part 2 of ISO 13209.

As shown in Figure 62 there are two groups of actions: the dialog actions which serve for opening different kinds of modal dialogs as well as the custom screen actions for opening and closing custom screens.

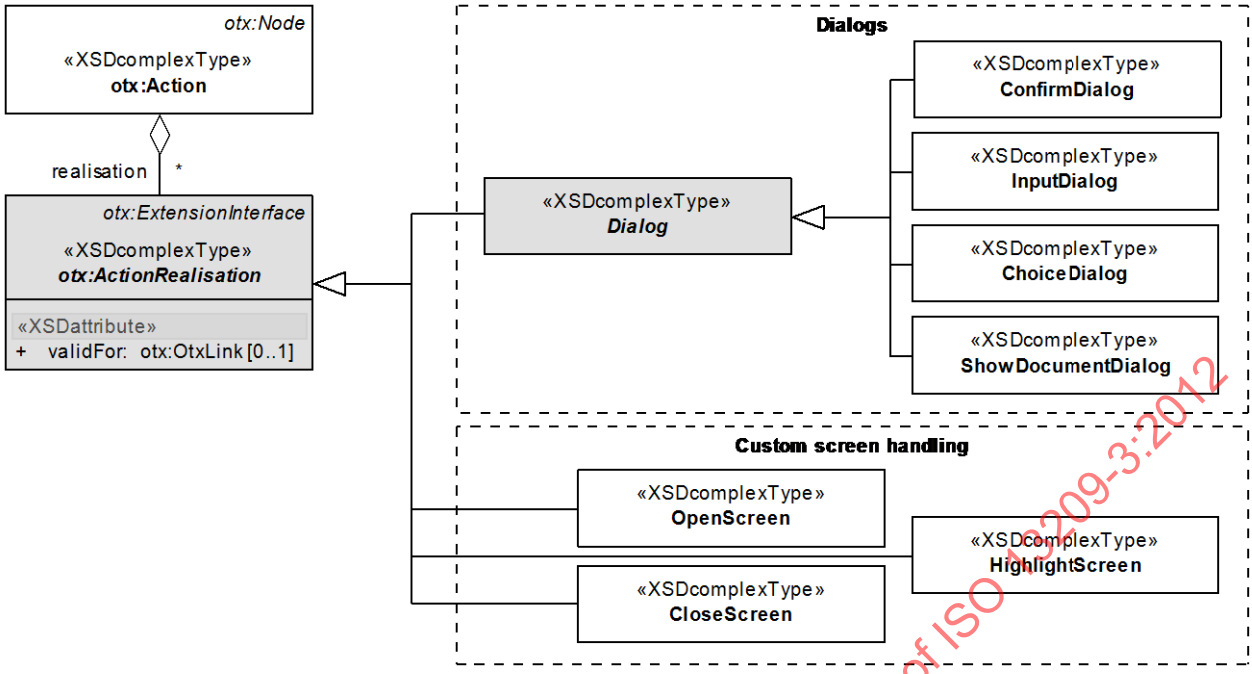

Figure 62 — Data model view: HMI actions overview

11.5.2 Dialog related actions

11.5.2.1 Description

The dialog related actions described in the following provide simple message dialogs, input dialogs, menu-like choice dialogs as well as displaying static documents. For a general description of dialogs, please refer to the introduction in 11.1.2.

11.5.2.2 Syntax

Figure 63 shows the syntax of all modal dialog actions of the HMI extension.

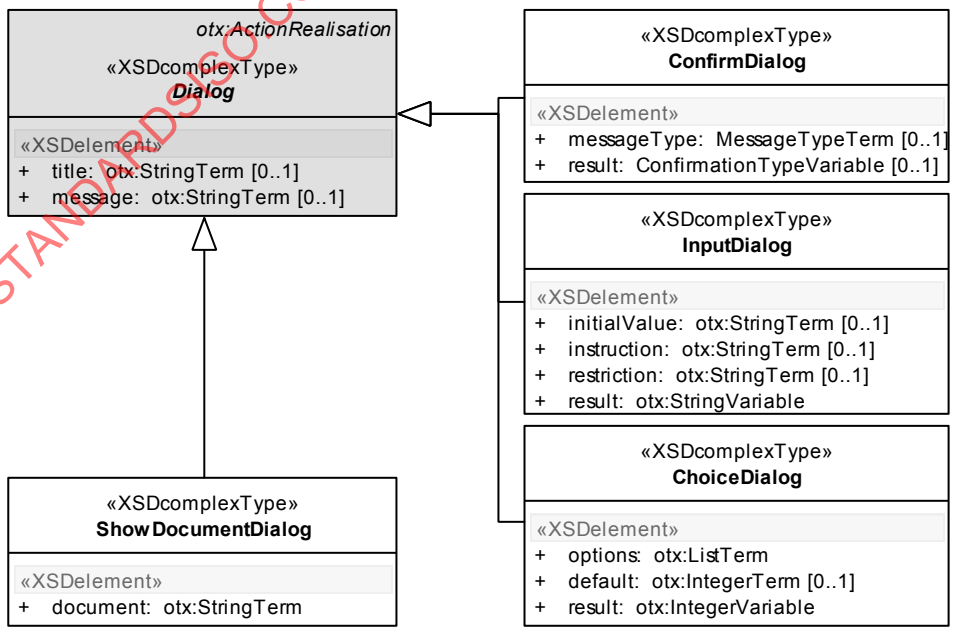

Figure 63 — Data model view: Dialog related actions

11.5.2.3 Semantics

11.5.2.3.1 Dialog

The abstract type **`Dialog`** is the base type for all the dialogs used as a part of the dialog usage pattern. It represents a modal display that blocks the execution of the test sequence until the user has provided an input. Dialogs are meant to be simple and well suited for basic interactions with the user, such as confirmations and single inputs.

The members of the **`Dialog`** action have the following semantics:

— **`<title> : otx:StringTerm [0..1]`**

If the runtime system contains the capability to show a dialog box with a title bar, the title string given by this element shall be shown. The title should be shown with more prominence than the message parameter (see below).

— **`<message> : otx:StringTerm [0..1]`**

If the runtime system contains the capability to show a message as part of the display, the message string given by this element shall be shown.

11.5.2.3.2 ConfirmDialog

The **`ConfirmDialog`** action shows a dialog asking for user confirmation. The choices of buttons and the decorations shown to the user can be configured by a parameter. Once one of the confirmation options is selected, the result field will contain the selected index of the options.

Confirmation dialogs are typically used to ask the user for acceptance before performing a procedure. Depending on the type of procedure to execute, it is possible to select different levels of severity.

Figure 64 shows a possible layout of a **`ConfirmDialog`** instance on a graphical user interface.

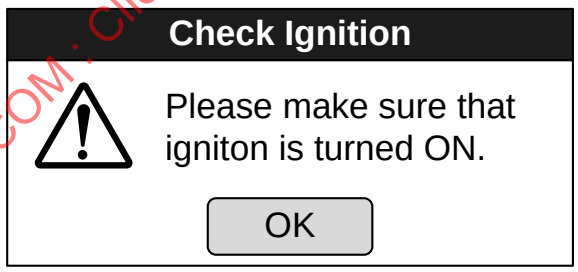

Figure 64 — Sample `ConfirmDialog` layout

`ConfirmDialog` is a **`Dialog`**. Its members have the following semantics:

— **`<messageType> : MessageTypeTerm [0..1]`**

This optional element defines the type of message and the buttons that shall be shown to the user to confirm the action. If the element is omitted, the default **`MessageType`** value **`INFO`** shall apply. Please refer to 11.2.3.3 for information about the **`MessageType`** enumeration.

The number of buttons displayed depends on the message type:

— **`INFO`**, **`WARNING`**, **`ERROR`**: Show "OK" button

— **`YESNO_QUESTION`**: Show "Yes" and "No" buttons

— **`YESNOCANCEL_QUESTION`**: Show "Yes", "No" and "Cancel" buttons

NOTE Since button labels usually get localized automatically according to test application locale settings, this standard does **not** force button labels to be "OK", "Yes", "No" or "Cancel". Any semantically equivalent labels are allowed.

— **<result> : ConfirmationTypeVariable [0..1]**

This element represents the variable where the selection from the user will be stored. The element can be omitted in cases when the result is nonrelevant (this especially applies to message types **INFO**, **WARNING** and **ERROR** which do only provide a single "OK" button).

Result values shall be one of:

— **YES**: "OK" or "Yes" button was pressed

— **NO**: "No" button was pressed

— **CANCEL**: "Cancel" button was pressed

11.5.2.3.3 InputDialog

The **InputDialog** action opens a dialog requesting string input from the user. If needed, an initial value can be passed to the dialog which shall be shown initially in the input field. Also an input restriction can be passed to the dialog; this shall be used by runtime systems to pre-validate inputs before they are passed back to the test sequence. Finally the entered value is assigned to a string variable for later use in the test sequence.

InputDialog can only handle simple strings. There are no facilities provided for number parsing etc. It is assumed that the OTX sequence will perform these actions upon receiving the value.

Figure 65 shows a possible layout of an **InputDialog** instance on a graphical user interface.

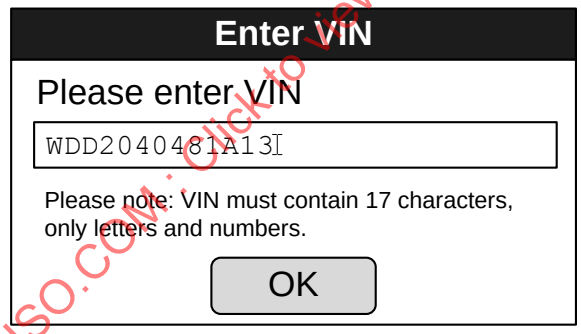

Figure 65 — Sample InputDialog layout

InputDialog is a **Dialog**. Its members have the following semantics:

— **<initialValue> : otx:StringTerm [0..1]**

This optional element represents the string value that shall be used to initialize the dialog's input field. Runtime systems should pre-populate the input field with this text, providing an option to the user to overwrite this value.

— **<instruction> : otx:StringTerm [0..1]**

The instruction is an additional message that can be shown on the input dialog to provide information regarding the expected value that should be introduced.

— **<restriction> : otx:StringTerm [0..1]**

This optional element represents a restriction onto the set of allowed input values. The restriction shall be formulated by a regular expression which shall be used by runtime systems to pre-validate the input data. The runtime system should not allow test sequence control to proceed until the input string matches the given regular expression. The regular expression shall be conforming to the *PERL 5 Regular Expression Description Version 12*.

— **<result> : otx:StringVariable [1]**

After the user dismisses the input dialog, the entered value shall be assigned to the string variable given by this element.

Throws:

— **otx:OutOfBoundsException**
If the restriction string is not conforming to the *PERL Regular Expression Description*.

11.5.2.3.4 ChoiceDialog

The **ChoiceDialog** shall present a list of options to the user. It shall be possible for the user to select one of the options and to dismiss the dialog (e.g. by double-clicking an option or by clicking an "OK" button, etc.). Once the dialog is dismissed, the chosen option's index shall be assigned to a result variable. It shall not be possible to dismiss the dialog unless a choice has been made.

The test sequence author may also preselect one of the options by using the dialog's optional default-selection property.

For the options, **ChoiceDialog** accepts a dynamic list of strings as argument. This is useful as the strings that will be shown can both be defined by the OTX author statically or they can be generated dynamically (e.g. by reading a list of values from an ECU).

The **ChoiceDialog** visual implementation is up to the runtime system. Suggested visualizations are:

— Combination of a combo box with an "OK" button

— A list component with an "OK" button

— A list component which allows dismissing the dialog by double-clicking an option

— A ring menu

Figure 66 shows a possible layout of a **ChoiceDialog** instance on a graphical user interface. The sample dialog contains three options which are rendered as a list. The user can select one of the options and then press the "OK" button commit her/ his choice and continue. The "OK" button should stay disabled as long as no choice has been made.

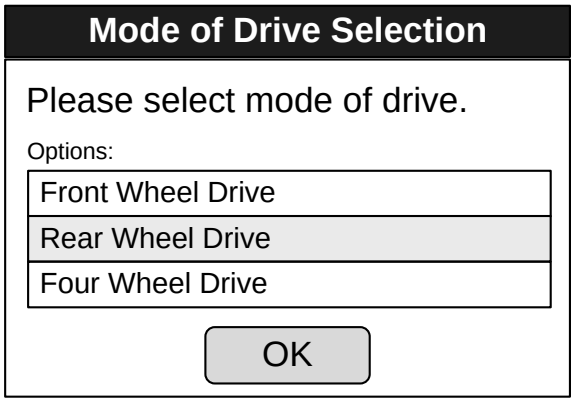

Figure 66 — Sample `ChoiceDialog` layout

`ChoiceDialog` is a **`Dialog`**. Its members have the following semantics:

— **`<options> : otx:ListTerm [1]`**

This element specifies a list of strings which contains the possible options that shall be displayed.

Associated checker rules:

— HMI_Chk001 – correct list type for ChoiceDialog options

— **`<default> : otx:IntegerTerm [0..1]`**

This optional element represents the index of the option which shall be preselected in the dialog.

— **`<result> : otx:IntegerVariable [1]`**

This element indicates the integer variable where the chosen option's index shall be assigned to:

— **`0`** The 1st option was selected.

— **`n-1`** The nth option was selected.

Throws:

— **`otx:OutOfBoundsException`**
If the list of options is empty (nothing to choose from) or if the preselection index is not within the range **`[0,n-1]`**, where **`n`** is the size of the list of options.

11.5.2.3.5 ShowDocumentDialog

The **`ShowDocumentDialog`** action will open a dialog that can display a document which is identified by e.g. a URI. The document can be any resource. The call will block until the user has confirmed reading the document.

Typical usage of this node is to show additional documentation to the users, such as repair guides and schematics, or lengthy security information that the user must read before proceeding with a potentially dangerous operation. If possible, tester applications should display the document in a maximized way, for best readability. Dismissing the dialog will close the document.

The set of supported document types is tester application specific. However, runtime systems should be able to display at least basic HTML 2.0 [RFC 1866]. Formatting, style and fonts can be stripped from display if the runtime system does not support advanced formatting capabilities (i.e. if only a single type, monospace font is used). Furthermore, popular image formats such as JPEG, GIF and PNG as well as document formats like plain text, rich text and PDF should be supported by tester applications also.

In case that a document type can not be opened and displayed by the test application itself, the runtime may delegate the opening of the document to the application which is registered for that document type on the operating system. If a tester application uses delegation, a dialog window shall pop up in the application, blocking execution and asking for confirmation from the user that the application can continue. Once the user confirms, the execution of the test sequence shall continue even if the external viewer is not closed.

Ultimately, if a document type is supported neither by the test application nor by any external viewer, then the test application shall show a suitable error message indicating that the document type is not supported and can therefore not be displayed.

ShowDocumentDialog is a **Dialog**. Its members have the following semantics:

— **<document> : otx:StringTerm [1]**

This element identifies the external document that should be shown.

Throws:

— **otx:InvalidReferenceException**
If the document resource given by the **<document>** element is not available or not accessible.

11.5.2.4 Example

The OTX fragment below shows uses of the **ConfirmDialog**, **InputDialog** and **ShowDocumentDialog**. Please compare this to Figure 64 and Figure 65 which show graphical equivalents of the dialog actions.

EXAMPLE Sample of OTX-file "HmiDialogsExample.otx"

```
<action id="a1">
  <specification>Ask user for confirmation that ignition is turned on.</specification>
  <realisation xsi:type="hmi:ConfirmDialog">
    <hmi:title xsi:type="StringLiteral" value="Check Ignition"/>
    <hmi:message xsi:type="StringLiteral" value="Please make sure that ignition is turned ON."/>
    <hmi:messageType xsi:type="hmi:MessageTypeLiteral" value="WARNING"/>
  </realisation>
</action>

<action id="a2">
  <specification>Get VIN from user and store in variable "Vin"</specification>
  <realisation xsi:type="hmi:InputDialog">
    <hmi:title xsi:type="StringLiteral" value="Enter VIN"/>
    <hmi:message xsi:type="StringLiteral" value="Please enter VIN"/>
    <hmi:instruction xsi:type="StringLiteral"
      value="Please note: VIN must contain 17 characters, only letters and numbers"/>
    <hmi:result name="Vin"/>
  </realisation>
</action>

<action id="a3">
  <specification>Show a wiring diagram to user.</specification>
  <realisation xsi:type="hmi:ShowDocumentDialog">
    <hmi:document xsi:type="StringLiteral" value="http://www.myCompany.com/WiringDiagram.svg"/>
  </realisation>
</action>
```


11.5.3 Custom screen related actions

11.5.3.1 Description

In contrast to the dialog actions described above, the actions below support the handling of custom screens, as described in the introduction in 11.1.3. In particular, there are the actions, **OpenScreen**, **CloseScreen** as well as **HighlightScreen** which allow controlling GUI output and input to and from the user and the opening and closing of screens. Custom screens are also related to so-called screen signatures which define the interface to application-specific screen definitions (cf. 11.7).

11.5.3.2 Syntax

Figure 67 shows the syntax of all custom screen related actions of the HMI extension.

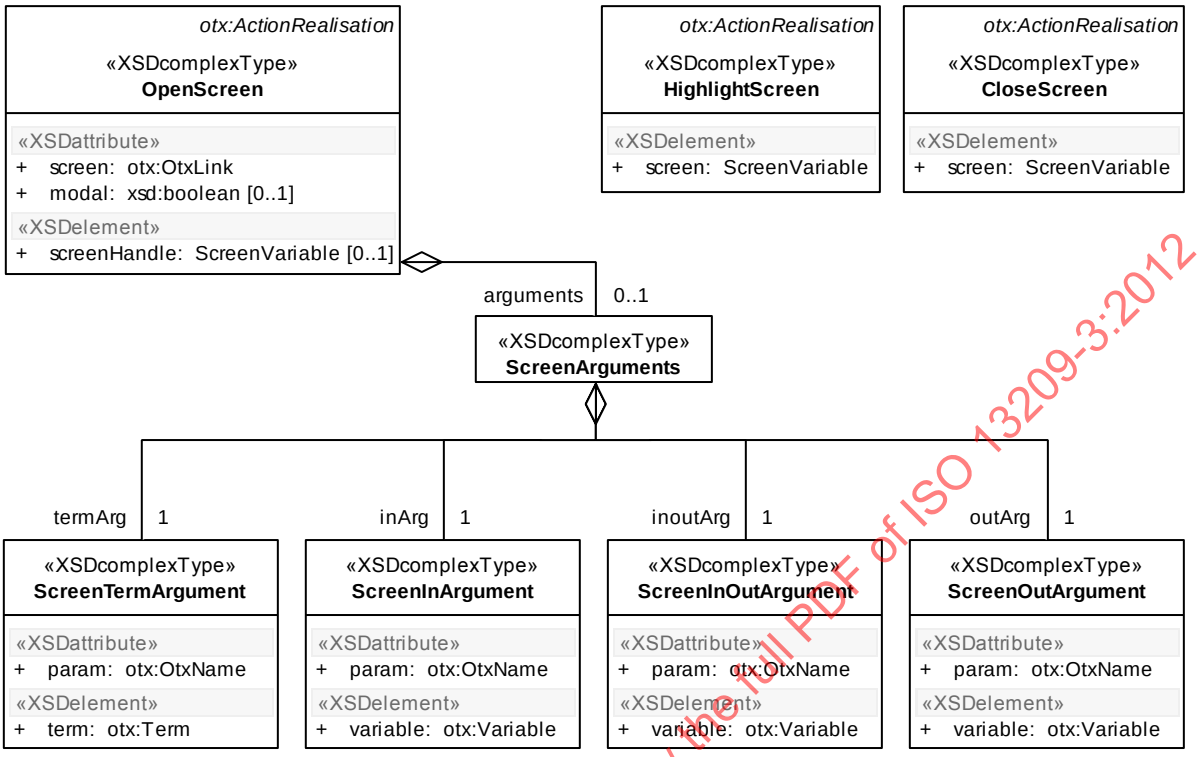

Figure 67 — Data model view: Custom screen related actions

11.5.3.3 Semantics

11.5.3.3.1 OpenScreen

The **`OpenScreen`** action creates and displays a custom screen at runtime. The screen shall be displayed immediately and it can accept user input. If other screens are already open when the new screen is opened, the test application shall ensure that the new screen is not hidden by other screens.

NOTE 1 In systems where screens are shown in separate GUI windows, the new screen should be sent on top of all other windows.

A screen shall remain opened until the user dismisses the screen via some UI control or a **`CloseScreen`** action is executed explicitly on its screen handle. Also, when there are unclosed screens at the moment when the procedure which opened the screens exits, only those screens on which at least one screen handle exists shall remain opened; all other screens shall be disposed of (cf. **`<screenHandle>`** element specified below).

When a screen is opened, the runtime system will internally locate a screen definition linked to the given screen signature name (see 11.7.3.2), or create a screen from scratch that allows displaying the given values to the user. If the screen can not be opened or the open action is not supported, a **`ScreenException`** shall be thrown.

NOTE 2 It is an explicit design goal of OTX not to describe the graphical layout of screens. Layout and look&feel of screens should be described by specific screen definitions used by a runtime system. Since these features are highly application-specific and do not represent semantically relevant information concerning the "pure" test sequence logic, screen definitions are not part of this standard.

The members of the **OpenScreen** action have the following semantics:

— **screen : otx:OtxLink [1]**

This attribute contains a name which points to a **ScreenSignature** which contains a parameter description for the screen that shall be opened (it is the interface description to a specific screen definition, see 11.7.3.2). The arguments of **OpenScreen** shall match the definitions in the corresponding signature. It is the task of the runtime system to provide a mapping from screen signatures to a runtime-specific screen definition (cf. Figure 56) which provides the actual screen layout.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

— HMI_Chk002 – correct target for OpenScreen

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

This option tells the runtime system to make this screen modal or non-modal. This means that if **modal** is **false** (the default), the OTX execution flow will immediately move on to the next **Action**, without waiting for the screen to close (see Figure 58 for usage examples of non-modal screens). Otherwise, if **modal** is **true**, the screen behaves like the dialog actions – it shall block the execution flow until the screen was closed (by a user action or a **CloseScreen** action e.g. in another parallel lane).

NOTE 3 Non-modal screens are well suited for dynamic cases where the test sequence needs to react on and process input from the screen, or needs to update values shown on the screen, where modal screens are better suited for the cases where static information is presented to the user.

— **<screenHandle> : ScreenVariable [0..1]**

This optional element represents the variable which shall be the handle for the opened screen. This can be later used to query the status of the screen, highlighting the screen or closing the screen explicitly (see **IsScreenOpen** term, **HighlightScreen** or **ScreenClose** actions below).

— **<arguments> : ScreenArguments [0..1]**

This simple container element represents a list of arguments for an open screen call. The content-type of **ScreenArguments** is **<xsd:choice> [1..*]** which allows an arbitrary-length list of different screen argument elements. The given arguments shall correspond to the parameters described in the screen signature (linked by the **screen** attribute). There are different argument types:

— **<termArg> : ScreenTermArgument**

This argument type allows setting a calculated value into the screen. The value passed is always an input-value to the screen. It shall be calculated exactly once upon opening the screen, no later recalculations of the term value shall happen.

This type of arguments can be used, when a value that is passed to a screen is not expected to change during the execution of the test sequence. Hence, it is not required that the runtime system keeps a "watch" on the value. Because it accepts a term, it becomes possible to calculate the value that should be set for a screen. For example, it might be desired to show a translated title that additionally contains a concatenated extra string.

A term argument may be omitted **if and only if** there's an explicit initial value defined for the corresponding parameter in the screen signature. In that situation, the initial value shall be used instead of the omitted argument.

The counterpart to a **<termArg>** shall be defined in the corresponding screen signature by a **<termParam>**.

— **param : otx:OtxName [1]**

This attribute indicates a unique parameter of the screen that shall receive the to-be-displayed value. The indicated parameter shall be defined in the corresponding screen signature. The screen definition should then contain a widget that will be fed with this value.

— **<term> : otx:Term [1]**

This element represents the term that shall be evaluated once and set as a value into the screen.

— **<inArg> : ScreenInArgument**

This argument type shall bind a variable to a parameter of the screen. Changes to the variable shall be automatically reflected on the screen.

An input argument may be omitted **if and only if** there's an explicit initial value defined for the corresponding parameter in the screen signature. In that situation, the initial value shall be used instead of the omitted argument.

The counterpart to an **<inArg>** shall be defined in the corresponding screen signature by an **<inParam>**.

— **param : otx:OtxName [1]**

This attribute indicates a unique input-parameter of the screen that shall be bound to a variable.

— **<variable> : otx:Variable [1]**

This element represents the to-be-bound variable whose value shall be monitored and whose current value shall be fed into the screen. The screen definition should then contain a widget that will be fed with this variable's current value.

— **<inOutArg> : ScreenInOutArgument**

This argument type shall bind a variable to a parameter of the screen, in a bidirectional fashion: Changes to the variable from the test sequence model shall be automatically reflected on the screen. Vice versa, changes triggered from the screen (e.g. by user actions) shall automatically change the value of the variable.

An input/output argument may be omitted **if and only if** there's an explicit initial value defined for the corresponding parameter in the screen signature. In that situation, the initial value shall be used instead of the omitted argument.

The counterpart to an **<inOutArg>** shall be defined in the corresponding screen signature by an **<inOutParam>**.

— **param : otx:OtxName [1]**

This attribute indicates a unique input/output parameter that shall be bound to a variable.

— **<variable> : otx:Variable [1]**

This element represents the to-be-bound variable whose value shall be monitored and whose current value shall be fed into the screen. The variable shall also reflect changes triggered from the screen, vice versa. The screen definition should then contain a widget that will be fed with this variable's current value and that also allows for the user to change the value.

— **<outArg> : ScreenOutArgument**

This argument type shall bind a variable to an output parameter of the screen. Changes on the screen shall trigger an update of the bound variable's value.

Output arguments may be omitted **freely** (e.g. in the case when there is no interest in one of the screen data).

The counterpart to the **<outArg>** shall be defined in the corresponding screen signature by a **<outParam>**.

— **param : otx:OtxName [1]**

This attribute indicates a unique output-parameter that shall be bound to a variable.

— **<variable> : otx:Variable [1]**

This element represents the to-be-bound variable which shall reflect the value set on the screen (e.g. entered by the user). The screen definition should then contain an input widget that allows for the user to change the value.

Associated checker rules:

— HMI_Chk003 – correct OpenScreen arguments

— HMI_Chk004 – OpenScreen term, input and input/output argument omission

— HMI_Chk005 – no Path in connected OpenScreen arguments

Throws:

— **ScreenException**
If the screen definition can not be found, if the assigned parameters are incorrect or if the runtime system can not support a screen open operation.

11.5.3.3.2 HighlightScreen

The **HighlightScreen** action shall highlight a given screen in a way appropriate for drawing the user's attention to the screen. This supports use cases where user attention is required, e.g. when a situation occurs which immediately requires user input on a particular screen, or when a screen displays important information which tells the user which actions to take to solve e.g. a critical situation, etc.

NOTE In systems where screens are shown in separate GUI windows, highlighting a screen should bring the screen on top of any other windows. For systems where screens are shown e.g. in one partitioned GUI window, highlighting may be done for instance by making the new screen's portion of the window blink for some time, or similar. Non-GUI systems may use e.g. warning LEDs to draw user attention.

The members of the **HighlightScreen** action have the following semantics:

— **<screen> : ScreenVariable [1]**

This element represents the screen handle of the screen that shall be highlighted.

11.5.3.3.3 CloseScreen

The **CloseScreen** action shall cause the runtime system to dismiss the screen and release all resources associated to the screen.

After the execution of the **CloseScreen** action, the screen shall not send any more events for processing to the OTX sequence and shall not allow any more user interaction to be performed.

Attempting to close the same screen element twice shall perform no operation and report no errors. It shall be for all effects a NOP.

The members of the **`CloseScreen`** action have the following semantics:

— **`<screen> : ScreenVariable [1]`**

This element represents the screen handle of the screen that shall be closed.

11.6 Terms

11.6.1 Overview

The terms of the OTX HMI extension are mainly related to custom screen handling and the events which may be fired by screens. Furthermore, there are simple enumeration type terms related to the **`ConfirmDialog`** action 11.5.2.3.2).

Figure 68 provides an overview about the different term categories.

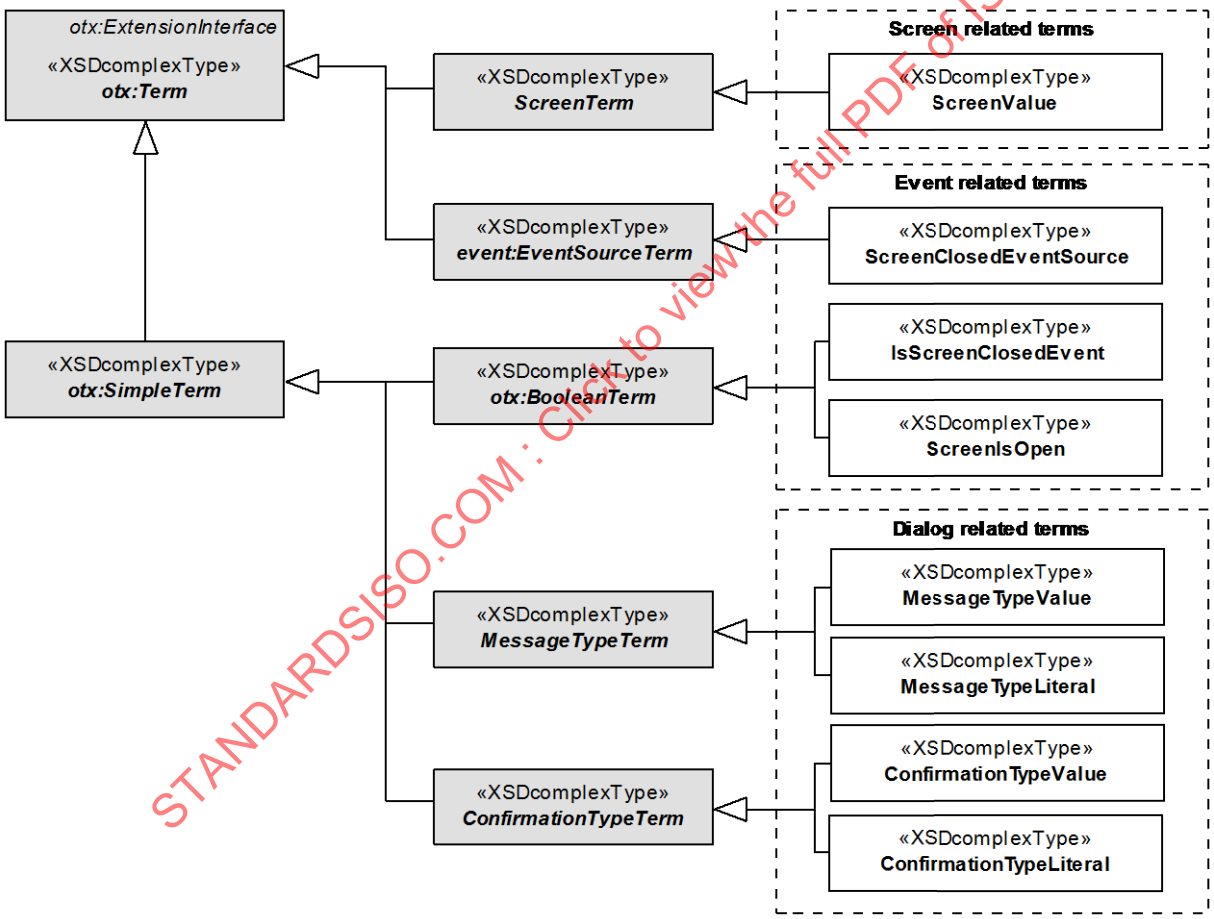

Figure 68 — Data model view: HMI term categories

11.6.2 Syntax

Figure 69 shows the syntax of all terms in the OTX HMI extension.

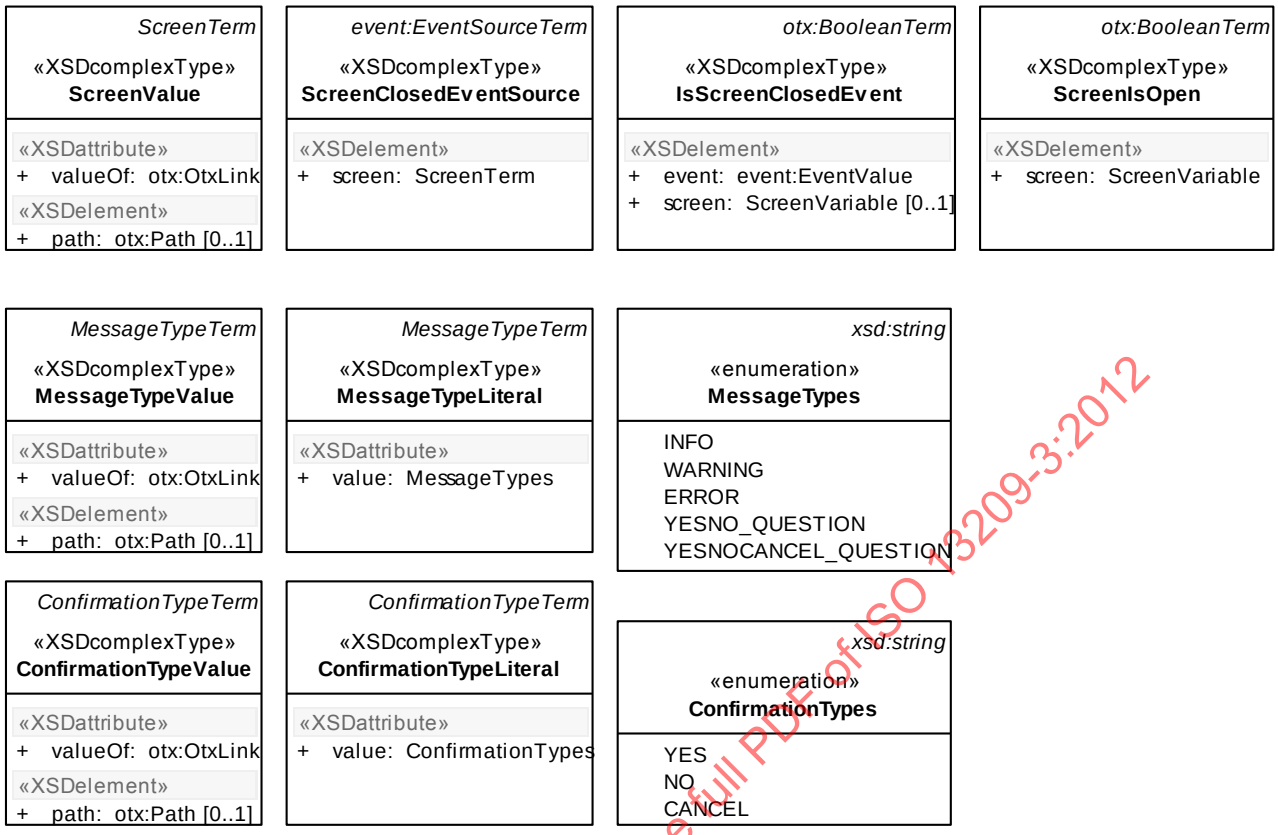

Figure 69 — Data model view: HMI terms

11.6.3 Semantics

11.6.3.1 ScreenTerm

`ScreenTerm` is an **`otx:Term`**. It serves as the abstract base type for all concrete terms which return a **`Screen`**. It has no further members.

11.6.3.2 ScreenValue

This term returns the **`Screen`** stored in a **`Screen`** variable. For more information on value-terms and the syntax and semantics of the **`valueOf`** attribute and **`<path>`** element, please refer to Part 2 of ISO 13209.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

Throws:

— **`otx:OutOfBoundsException`**
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).

— **`otx:InvalidReferenceException`**
If the variable value is not valid (no value was assigned to the variable before).

11.6.3.3 ScreenClosedEventSource

The **`ScreenClosedEventSource`** term accepts a **`Screen`** object that is to be made an event source. This term enables an OTX sequence to use a **`Screen`** as a source for events in the context of the OTX EventHandling extension (please refer to Clause 8). A **`Screen`** shall trigger an event every when the specified

screen is closed. This can be used within a **event:WaitForEventAction** to continue execution after a screen was closed.

NOTE Other events that may happen on a screen (like button presses, values entered into an input field etc.) can be identified by using the event source terms **MonitorChangeEventSource** or **ThresholdExceededEventSource**, as specified by the OTX EventHandling extension (see Clause 9). As screens are executed on an asynchronous thread, user interaction can be received at any time. Therefore the value monitoring event sources are especially useful with respect to non-modal custom screens (see Figure 58) in order to react on different user actions.

ScreenClosedEventSource is an **event:EventSource**. Its members have the following semantics:

— **<screen> : ScreenTerm [1]**

Represents the **Screen** that shall be connected to the event source.

11.6.3.4 IsScreenClosedEvent

The **IsScreenClosedEvent** term accepts an **EventValue** term yielding an **Event** object that has been raised by the OTX runtime, as a result of declaring a **Screen** object as an event source by using the term **ScreenClosedEventSource**. The term shall return **true** if and only if the **Event** originates from a **ScreenClosedEventSource** term. In case an optional **ScreenVariable** is specified, the term shall return **true** if and only if the **Event** was fired because that particular **Screen** was closed.

This term exists because closing a screen is a very common event and many times the execution flow must continue only when a screen is dismissed. To simplify writing test sequences, it is thus simpler to write a **WaitForEvent** node that only listens for this event type, and without requiring additional code to analyze the type of even as required with a regular screen event.

IsScreenClosedEvent is an **otx:BooleanTerm**. Its members have the following semantics:

— **<event> : event:EventValue [1]**

Represents the **Event** whose type shall be tested.

— **<screen> : ScreenVariable [0..1]**

Optionally specifies the particular **Screen** which fired the event.

11.6.3.5 ScreenIsOpen

This is a term used to verify that a **Screen** is open and active. A **Screen** is open and active if it has been opened by using an **OpenScreen** action, it has not been dismissed by the user and it has not been closed by using a **CloseScreen** action.

IMPORTANT — Due to the fact that there may be multiple parallel lanes, and that a screen engine normally works in a different thread, if the ScreenIsOpen term returns true there is actually no guarantee that the screen is still open on the next step.

ScreenIsOpen is an **otx:BooleanTerm**. Its members have the following semantics:

— **<screen> : ScreenVariable [1]**

This element represents the variable which is a handle to the screen that shall be checked.

11.6.3.6 MessageTypeTerm

The abstract type **MessageTypeTerm** is an **otx:SimpleTerm**. It serves as a base for all concrete terms which return a **MessageType** value (see 11.2.3.3). It has no special members.

11.6.3.7 MessageTypeValue

This term returns the **MessageType** stored in a **MessageType** variable. For more information on value-terms and the syntax and semantics of the **valueOf** attribute and **<path>** element, please refer to Part 2 of ISO 13209.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

Throws:

— **otx:OutOfBoundsException**
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).

11.6.3.8 MessageTypeLiteral

This term shall return a **MessageType** value (see 11.2.3.3) from a hard-coded literal.

MessageTypeLiteral is a **MessageTypeTerm**. Its members have the following semantics:

— **value : MessageTypes={INFO|WARNING|ERROR|YESNO_QUESTION|YESNOCANCEL_QUESTION} [1]**

This attribute shall contain one of the values defined in the **MessageTypes** enumeration.

11.6.3.9 ConfirmationTypeTerm

The abstract type **ConfirmationTypeTerm** is an **otx:SimpleTerm**. It serves as a base for all concrete terms which return a **ConfirmationType** value (see 11.2.3.4). It has no special members.

11.6.3.10 ConfirmationTypeValue

This term returns the **ConfirmationType** stored in a **ConfirmationType** variable. For more information on value-terms and the syntax and semantics of the **valueOf** attribute and **<path>** element, please refer to Part 2 of ISO 13209.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

Throws:

— **otx:OutOfBoundsException**
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).

11.6.3.11 ConfirmationTypeLiteral

This term shall return a **ConfirmationType** value (see 11.2.3.4) from a hard-coded literal.

ConfirmationTypeLiteral is a **ConfirmationTypeTerm**. Its members have the following semantics:

— **value : ConfirmationTypes={YES|NO|CANCEL} [1]**

This attribute shall contain one of the values defined in the **ConfirmationTypes** enumeration.

11.7 Signatures

11.7.1 Overview

As specified by Part 2 of ISO 13209, OTX extensions may define new specialialized types of signatures by extending **`otx:SignatureRealisation`**. The OTX HMI extension uses this extensibility by adding the **`ScreenSignature`** type which allows in-document, high-level interface specifications to screen definitions which are used by the **`OpenScreen`** action, as specified in 11.5.3.3.1.

11.7.2 Syntax

Figure 70 shows the syntax of the HMI extension's signature types.

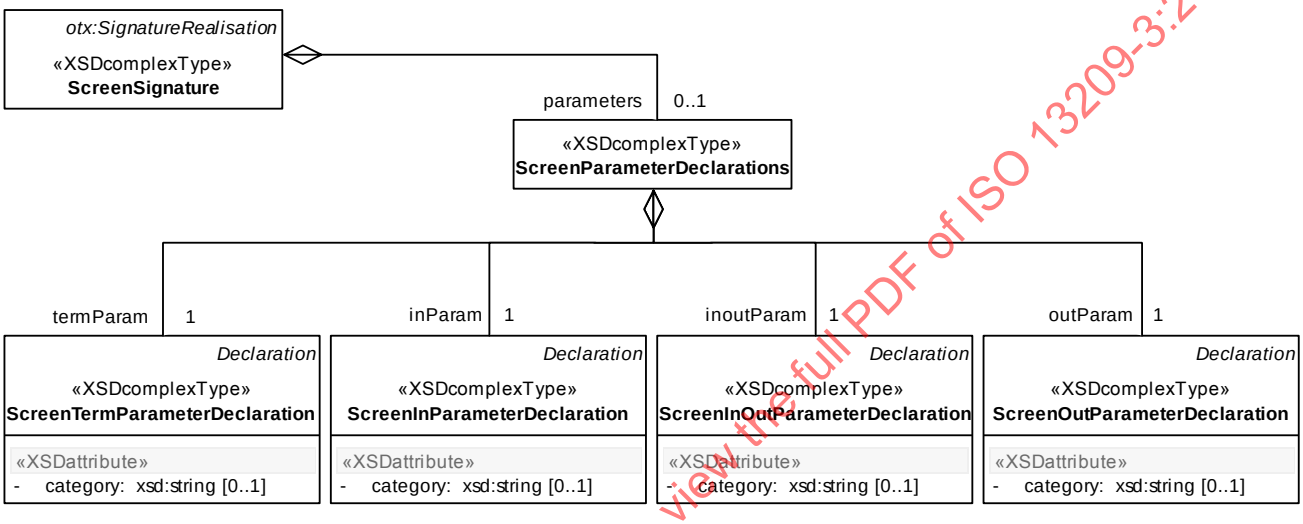

Figure 70 — Data model view: HMI signatures

NOTE The XSD complex type **`ScreenParameterDeclarations`** is of **`<xsd:choice> [1..*]`** content-type, which is not explicitly shown in the figure above.

11.7.3 Semantics

11.7.3.1 General

The basic semantics common to all kinds of signatures are specified in Part 2 of ISO 13209.

11.7.3.2 ScreenSignature

The **`ScreenSignature`** is a specialisation of the OTX Core type **`otx:SignatureRealisation`** that adds additional interface description functionality along with the OTX Core type **`otx:ProcedureSignature`**. A **`ScreenSignature`** represents the interface between **`OpenScreen`** actions and the runtime specific screen definitions.

Typically, all screens supported by a runtime system should be described in a dedicated set of OTX documents – "screen description documents" – which contain the signatures of all screen definitions available for the OTX author. This allows the author to create test sequences that use a pre-established UI.

Screen signatures shall also be used to verify that the arguments given by an **`OpenScreen`** action are complete.

Since **ScreenSignature** is an **otx:SignatureRealisation**, screen signatures have to be globally defined in OTX documents: They are located under the **<signatures>** element right below the root element **<otx>**, as defined by Part 2 of ISO 13209.

The members of **ScreenSignature** have the following semantics:

— **<parameters> : ScreenParameterDeclarations [0..1]**

This contains a list of parameters of different types. They describe which input- and output-values a certain screen needs or provides. The parameters of a specific screen signature are the counterparts of the arguments of a **ScreenOpen** action (cf. 11.5.3.3.1). Since all parameter types are derived from the **otx:Declaration** type as defined in Part 2 of ISO 13209, the parameters have a name, a specification and a data type declaration (not specified here).

ScreenParameters is of **<xsd:choice> [1..*]** content-type which allows an arbitrary-length list of parameter sub-elements of the following types:

— **<termParam> : ScreenTermParameter**

This represents the counterpart to the **<termArg>** type of the **OpenScreen** action (cf. 11.5.3.3.1). It declares an input parameter for a screen whose value shall be computed once out of a term given in a corresponding term argument of an **OpenScreen** action. The value shall be shown by a suitable widget on the screen.

— **category : xsd:string [1]**

This attribute indicates the category of the parameter, see below for details.

— **<inParam> : ScreenInParameter**

This represents the counterpart to the **<inArg>** type of the **OpenScreen** action (cf. 11.5.3.3.1). It declares an input parameter which shall be linked to an OTX variable (by executing an **OpenScreen** action at runtime). Value changes of the variable shall automatically trigger an update of the respective widget on the screen.

— **category : xsd:string [1]**

This attribute indicates the category of the parameter, see below for details.

— **<inoutParam> : ScreenInOutParameter**

This represents the counterpart to the **<inoutArg>** type of the **OpenScreen** action (cf. 11.5.3.3.1). It declares an bidirectional input/output parameter which shall be linked to an OTX variable (by executing an **OpenScreen** action at runtime). Value changes in the variable shall automatically trigger an update of the respective widget on the screen and vice versa, if the user changes the value on the screen, the new value shall be reflected in the linked variable.

— **category : xsd:string [1]**

This attribute indicates the category of the parameter, see below for details.

— **<outParam> : ScreenOutParameter**

This represents the counterpart to the **<outArg>** type of the **OpenScreen** action (cf. 11.5.3.3.1). It declares an output parameter of the screen which shall be linked to an OTX variable (by executing an **OpenScreen** action at runtime). If the user changes the value on the screen via the corresponding input widget, the new value shall be reflected in the linked variable.

— **category : xsd:string [1]**

This attribute indicates the category of the parameter, see below for details.

As specified above, each of the parameter types contains an additional **category** attribute. It is an optional hint to the runtime system regarding the usage of the associated parameter. Some runtime systems might not have a specific screen definition corresponding to a given screen signature, or do not support the concept of screen definitions at all. Such systems can use the **category** to attach semantic meaning to certain arguments of an **OpenScreen** action and choose an appropriate representation for the values.

Runtime systems are not required to implement this functionality. Any text can be used; however, the following have standardized meanings:

— **TITLE**: Parameter should be rendered as a title.

— **MESSAGE**: Parameter should be rendered as a message.

— **GRAPH**: Parameter should be displayed with a visual graphical representation.

— **WARNING**: Parameter should be displayed as a warning.

— **BUTTON**: Parameter should be rendered as a button.

— **CHECKBOX**: Parameter should be rendered as a checkbox.

— **INPUT**: Parameter should be rendered as input mask

— **CHOICE**: Parameter should be rendered as a choice (applies to **otx:List** and **otx:Map** only)

12 OTX i18n extension

12.1 Introduction

The OTX i18n (Internationalization) extension provides access to data types, terms and actions for translating strings, quantity units and values to the language and unit system of the locale of the runtime system.

Due to the international reach of vehicle manufacturers and the existence of research labs, production plants and repair shops across the globe, it is necessary to provide an API that will make a test sequence agnostic of the particularities of the language and the system present on the target region. Thus, all strings that will be presented to the user must be stored in a common format, referenced by keys and translated on the fly.

12.2 Data types

12.2.1 Overview

The OTX i18n extension introduces a single data type named **`TranslationKey`**, as described in the following.

12.2.2 Syntax

The syntax of the **`TranslationKey`** datatype declaration of the OTX i18n extension is shown in Figure 71.

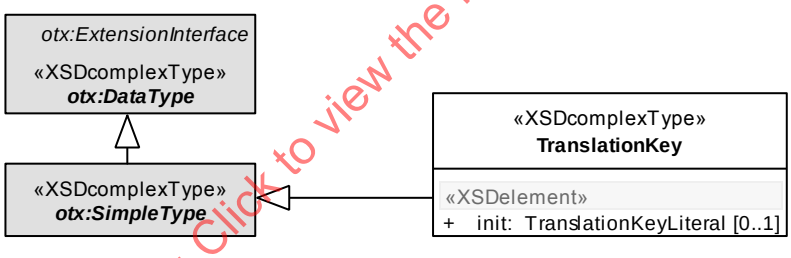

Figure 71 — Data model view: i18n data types

12.2.3 Semantics

12.2.3.1 General

The data types in the OTX i18n extension are derived from **`otx:SimpleType`**.

12.2.3.2 TranslationKey

A **`TranslationKey`** is a reference to a unique string message which can be internationalized (e.g. by corresponding entries in a thesaurus database). The concept used in OTX is similar to the concept used in many programming languages, where all messages that must be shown to the user are externalized and referenced by keys.

The actual retrieval procedure is defined by the runtime system.

`TranslationKey` is an **`otx:SimpleType`**. Its members have the following semantics:

— **`<init> : TranslationKeyLiteral [0..1]`**

This optional element stands for the initialisation of the identifier at declaration time. Initialisation is done by a hard-coded text ID in the OTX document.

— **`value : xsd:string [1]`**

This attribute contais the text ID value.

IMPORTANT — If the `TranslationKey` declaration is not explicitly initialized (omitted `<init>` element), the default value shall be the empty string.

12.3 Exceptions

12.3.1 Overview

All elements referenced in this clause are derived from the OTX Core **`Exception`** type as defined by Part 2 of ISO 13209. They represent the full set of exceptions added by the OTX i18n extension.

12.3.2 Syntax

The syntax of all OTX i18n exception type declarations is shown in Figure 72.

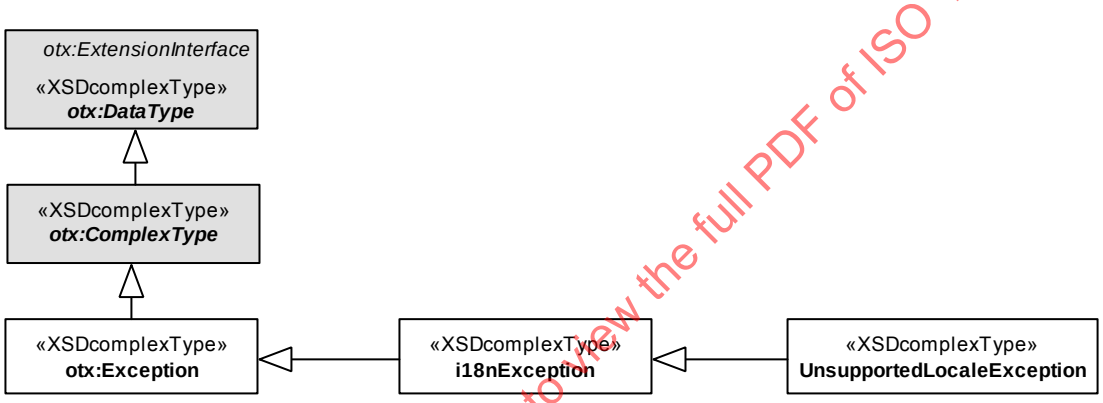

Figure 72 — Data model view: i18n exceptions

12.3.3 Semantics

12.3.3.1 General

Since all OTX i18n exception types are implicit exceptions whithout initialisation parts, they can not be declared constant.

12.3.3.2 i18nException

The **`i18nException`** is the super class for all exceptions in the i18n extension. An **`i18nException`** shall be used in case the more specific exception types described in the remainder of this section do not apply to the problem at hand.

12.3.3.3 UnsupportedLocaleException

The **`UnsupportedLocaleException`** shall be thrown when a locale related operation fails because the runtime system does not support the target locale.

12.4 Variable access

12.4.1 Overview

As specified in Part 2 of ISO 13209, OTX extensions shall define a variable access type for each datatype they define (exception types inclusively). All variable access types are derived from the OTX Core **`Variable`** extension interface. The following specifies all variable access types defined for the i18n extension.

12.4.2 Syntax

Figure 73 shows the syntax of the i18n extension's variable access types.

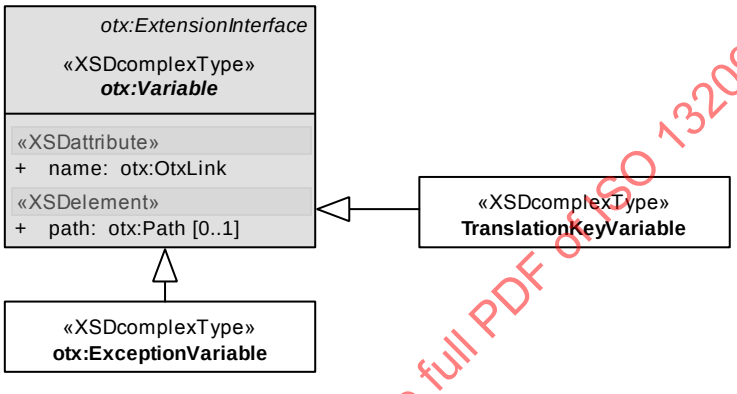

Figure 73 — Data model view: i18n variable access types

12.4.3 Semantics

The general semantics for all variable access types shall apply. Please refer to Part 2 of ISO 13209 for further details.

12.5 Terms

12.5.1 Overview

All of the i18n terms shown in Figure 74 extend the **`otx:Term`** extension interface (directly and indirectly), as defined by Part 2 of ISO 13209.

The i18n extension introduces the abstract type **`TranslationKeyTerm`** which serves as the base type for all i18n terms yielding **`TranslationKey`** values. **`TranslationKeyTerm`** itself is based on the abstract OTX Core term **`otx:StringTerm`**. Therefore, a **`TranslationKeyTerm`** can be applied in any place where an **`otx:StringTerm`** is required.

Other i18n terms extend the abstract OTX Core terms **`otx:ListTerm`**, **`otx:StringTerm`** and **`otx:BooleanTerm`**, furthermore **`quant:QuantityTerm`** is used for the localisation of quantities (please refer to Clause 17).

The i18n terms are assigned to the following categories:

— **Locale settings.** Terms in this category are related to locale settings of diagnostic applications.

— **Translation related.** This category is for terms which represent diverse translation functionality.

— **Quantity related.** These are terms used for localizing quantities.

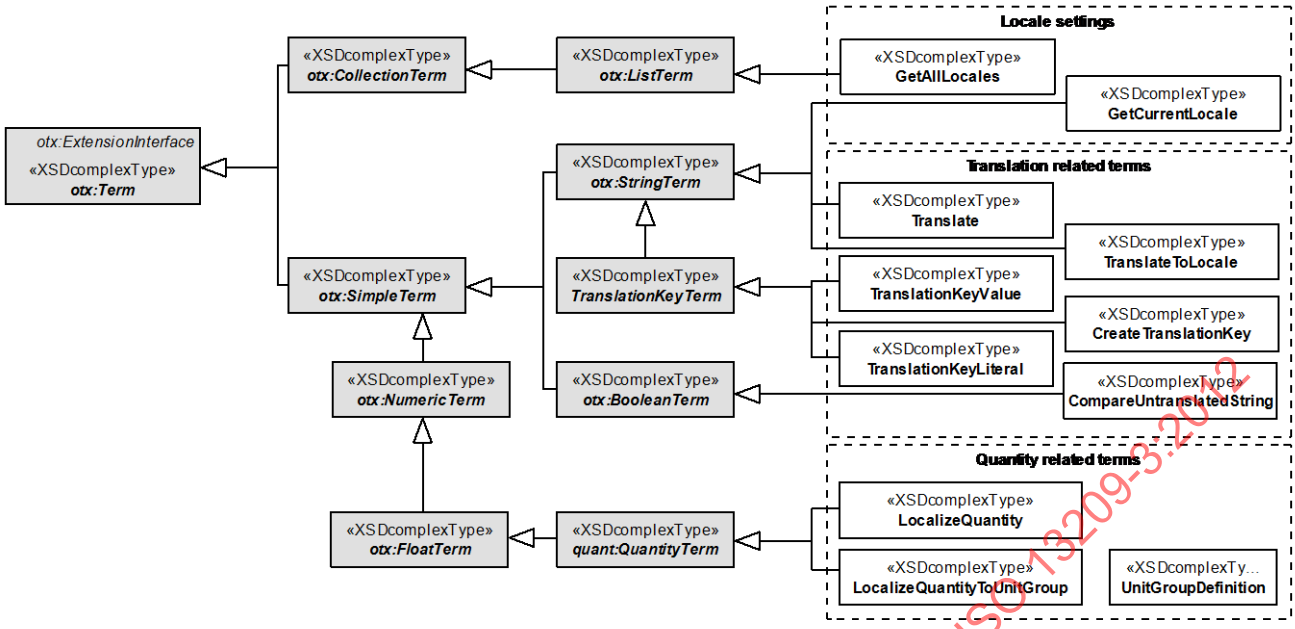

Figure 74 — Data model view: i18n term categories

12.5.2 Locale settings related terms

12.5.2.1 Description

The terms in this category are designed for retrieving locale settings of diagnostic applications.

12.5.2.2 Syntax

Figure 75 shows the syntax of all locale setting related terms of the i18n extension.

Figure 75 — Data model view: Locale setting related terms

12.5.2.3 Semantics

12.5.2.3.1 GetCurrentLocale

The **`GetCurrentLocale`** term shall retrieve the current locale code in use by the runtime system. The returned locale code shall be a combination of the ISO 639-1 two-letter language code followed by a hyphen, and then the uppercase two letter country code as defined by ISO 3166. Optionally, a variant code may be added in case of additional customizations (headed by another hyphen). The variant codes are not defined by this standard.

If no current locale is selected, the system shall return the default locale.

EXAMPLE Following the rules above, a returned locale shall be formed like e.g. **`"de-CH-1901"`** (for the variant of German orthography dating from the 1901 reforms, as seen in Switzerland).

`GetCurrentLocale` is an **`otx:StringTerm`**. It has no members.

12.5.2.3.2 GetAllLocales

The term **`GetAllLocales`** should retrieve all available locales from the runtime system that are supported and for which translations are available.

The fact that a runtime system returns a locale does not guarantee that all translations and units are available. Rather, this method returns the locales that can be used, regardless of data availability. It is however recommended that runtime systems consult their translation data store before returning the list of locales, so the results should be close to the actual available data.

The returned value shall be a list of strings using the same locale format as specified for the **`GetCurrentLocale`** term (see above).

This term allows querying some of the capabilities of the underlying runtime system. It is useful information e.g. for the **`TranslateToLocale`** term, as it is known before hand what can be used as valid locale input.

`GetAllLocales` is an **`otx:ListTerm`** without any further members.

12.5.3 Translation related terms

12.5.3.1 Description

The terms in this category are designed for managing, translating and comparing **`TranslationKey`** values.

12.5.3.2 Syntax

Figure 76 shows the syntax of all translation related terms of the i18n extension.

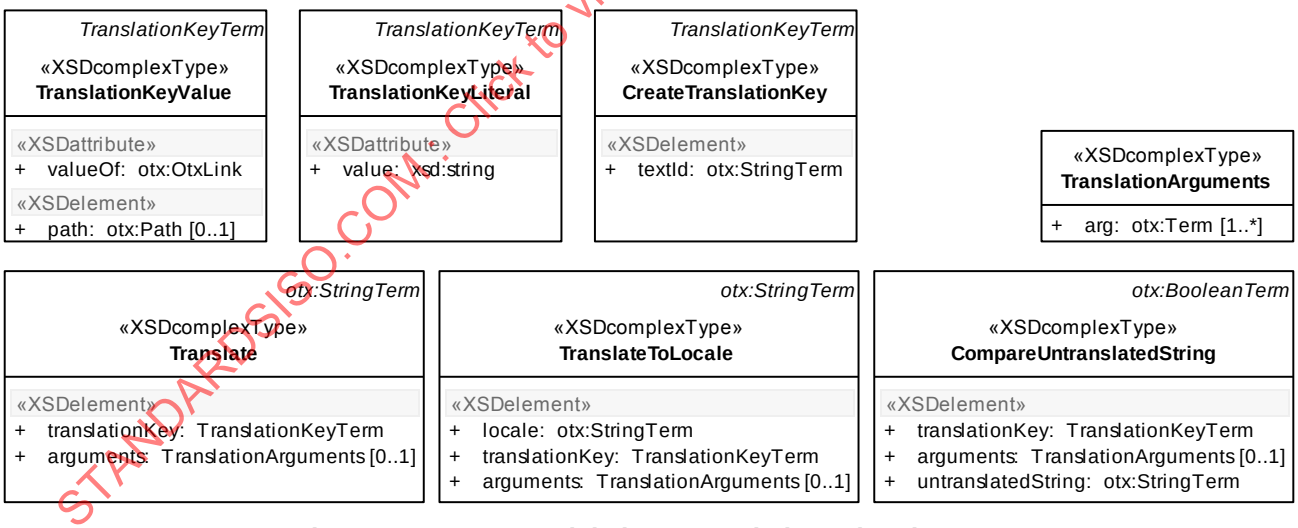

Figure 76 — Data model view: Translation related terms

12.5.3.3 Semantics

12.5.3.3.1 TranslationKeyTerm

The abstract type **`TranslationKeyTerm`** is an **`otx:StingTerm`**. It serves as a base for all concrete terms which return a **`TranslationKey`**. It has no special members.

IMPORTANT — If the OTX Core conversion term `otx:ToString` is applied to a `TranslationKey`, the internal text ID string value contained in the `TranslationKey` shall be returned.

12.5.3.3.2 TranslationKeyValue

This term returns the **TranslationKey** stored in a **TranslationKey** variable. For more information on value-terms and the syntax and semantics of the **valueOf** attribute and **<path>** element, please refer to Part 2 of ISO 13209.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

Throws:

— **otx:OutOfBoundsException**
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).

12.5.3.3.3 TranslationKeyLiteral

This term shall be used to create a **TranslationKey** data object based on a hard-coded text ID. The text ID is a reference to an external thesaurus system. It is assumed that the runtime system contains a data storage that knows how to create a **TranslationKey** data based on the literal.

NOTE The creation of the object should always work, as the data should not be loaded from the runtime system

TranslationKeyLiteral is a **TranslationKeyTerm**. Its members have the following semantics:

— **value : xsd:string [1]**

The text ID represents a simple string that is used by the runtime system as a reference to its internal storage of localized string translations. The exact usage and translation is not defined in the standard.

12.5.3.3.4 CreateTranslationKey

The **CreateTranslationKey** term creates a **TranslationKey** out of a given string. The string is used as the text ID that will be used to create the **TranslationKey**.

This term allows dynamically creating translation keys as a result of, for example, accessing specific parts of ODX data sources.

CreateTranslationKey is a **TranslationKeyTerm**. Its members have the following semantics:

— **<textId> : otx:StringTerm [1]**

The string term value provided that will be used to generate a translation key.

12.5.3.3.5 Translate

The **Translate** term accepts a **TranslationKey** which may be supplemented by additional translation arguments for message parameter substitution (if required by the associated thesaurus entry). It shall return a localized string in the current user language.

It is assumed that the runtime system contains a user or system selected locale which will be used to automatically perform the translation.

Recommendation: If no translation is available in the current locale, the runtime system may use a fall back strategy that consists on consulting a family that share a common base language. Otherwise the default language may be used. This fallback strategy is out of scope of the specification.

In case that the translation is unknown by the runtime system, the translation key itself shall be returned as the translation. This is to avoid error conditions in the system due to incorrect translations.

IMPORTANT — It is not in the scope of this standard to specify or expect a certain kind of thesaurus database structure. However, concerning compound thesaurus entries (messages with parameters), thesaurus entries should be formed in a similar way like the string patterns specified by the Java class `MessageFormat` (`java.text.MessageFormat`). This allows identifying parameters in the pattern unambiguously, e.g. the parameters `{0}` and `{1}` in `"The resistance of injector {0} is {1}"`. This provided, `Translate` shall function like `MessageFormat.format(String pattern, Object[] arguments)`, where the arguments substitute the message parameters according to their position in the `arguments` array.

`Translate` is an **`otx:StringTerm`**. Its members have the following semantics:

— **`<translationKey> : TranslationKeyTerm [1]`**

This element represents a unique key that the system shall use to search its internal database for a translation. Once a translation is found and parameter substitution has taken place, the resulting message string shall be returned.

— **`<arguments> : TranslationArguments [0..1]`**

This optional element represents a list of arguments for the translation. The arguments shall be evaluated first before being inserted into the translated message. The order of arguments is important; the first argument shall substitute message parameter **`{0}`**, the second parameter **`{1}`**, and so on.

— **`<arg> : Term [1..*]`**

Represents an argument which will be substituted in the resulting messages at the corresponding parameter's position. Non-**`String`** arguments shall be converted automatically to **`String`** prior to parameter substitution.

EXAMPLE Consider a thesaurus entry in English `ID1: "The resistance of injector {0} is {1}"` or similar, in German `ID1: "Der Widerstand des Injektors {0} ist {1}"`. Also consider a quantity `Q` which represents `10 Ohm`. If the current locale is English, applying `Translate(ID1, [3, Q])` will produce the English output `"The resistance of injector 2 is 10 Ohm."` or likewise, if the current locale is German, the output `"Der Widerstand des Injektors 2 ist 10 Ohm."`.

12.5.3.3.6 TranslateToLocale

The **`TranslateToLocale`** term shall perform a similar function to the **`Translate`** term, but instead of using the current locale it shall use a target locale that is given as an argument to the call, formed after the rules of the ISO 639-1 standard.

NOTE Using this term forces a translation to a specific language. This might be desirable for specific situations such as for feedback send to a support desk.

`TranslateToLocale` is an **`otx:StringTerm`**. Its members have the following semantics:

— **`<locale> : otx:StringTerm [1]`**

The translation process shall use this string as the target locale for the translation. The locale is expected to be formed after the rules of the ISO 639-1 standard, as explained for the **`GetCurrentLocale`** term (see 12.5.2.3.1).

— **`<translationKey> : TranslationKeyTerm [1]`**

This element represents a unique key that the system shall use to search its internal database for a translation. Once a translation is found and parameter substitution has taken place, the resulting message string shall be returned.

— **<arguments> : TranslationArguments [0..1]**

This optional element represents a list of arguments for the translation. The arguments shall be evaluated first before being inserted into the translated message. The order of arguments is important; the first argument shall substitute message parameter **{0}**, the second parameter **{1}**, and so on.

— **<arg> : Term [1..*]**

Represents an argument which will be substituted in the resulting messages at the corresponding parameter's position. Non-**String** arguments shall be converted automatically to **String** prior to parameter substitution.

Throws:

— **UnsupportedLocaleException**
If the runtime system does not support the given locale.

12.5.3.3.7 CompareUntranslatedString

The **CompareUntranslatedString** term compares whether an untranslated string equals at least one of the translations of a given translation key. While searching for a match, each available locale shall be considered by the runtime. The term shall return **true** if and only if a matching translation can be found.

EXAMPLE The **CompareUntranslatedString** term is useful in cases where e.g. an ECU responds in the form of a hard-coded string, e.g. "OFFEN" (German for "OPEN"). **CompareUntranslatedString** may now be used by an OTX author to find out if whether this is a translation for a given translation key and use that information for further purposes. This is also important at authoring time since OTX editor tools might show the key translation in the current locale of the editor, thus making comparisons like myOpenCloseResponseGerman=="OPEN" possible/visible, and therefore localizing the editor tool.

CompareUntranslatedString is an **otx:BooleanTerm**. Its members have the following semantics:

— **<translationKey> : TranslationKeyTerm [1]**

This element represents a unique key that the system shall use to search its internal database for a matching translation which matches the untranslated string. If message parameters exist, argument substitution has to be performed first, prior to comparison.

— **<arguments> : TranslationArguments [0..1]**

This optional element represents a list of arguments for the translation (cf. **Translate** term). The arguments shall be evaluated first before being inserted into the translated message. The order of arguments is important; the first argument shall substitute message parameter **{0}**, the second parameter **{1}**, and so on.

— **<arg> : Term [1..*]**

Represents an argument which will be substituted in the resulting messages at the corresponding parameter's position. Non-**String** arguments shall be converted automatically to **String** prior to parameter substitution.

— **<untranslatedString> : otx:StringTerm [1]**

Represents the string which shall be tested for a match.

12.5.4 Quantity related terms

12.5.4.1 Description

The terms in this category are designed for managing quantities with respect to locale settings.

12.5.4.2 Syntax

Figure 77 shows the syntax of all quantity related terms of the i18n extension.

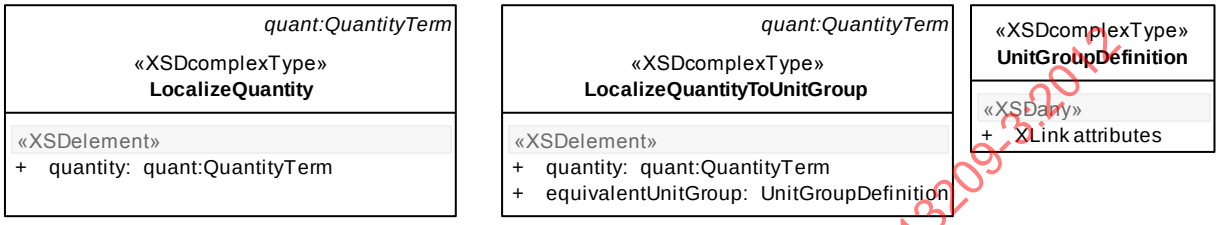

Figure 77 — Data model view: Quantity related terms

12.5.4.3 Semantics

12.5.4.3.1 Referring to unit group definitions

The **`LocalizeQuantityToUnitGroup`** term uses the **`UnitGroup`** type in order to refer to unit group definitions located in an external resource. The OTX i18n extension reuses the unit definition data model specified by the ODX standard (see **`UNIT-SPEC`** data type in 7.3.6.7 in ISO 22901-1:2008). Concerning references from OTX to **`UNIT-SPEC`** entries, the rules below shall apply.

IMPORTANT — Any elements of the OTX i18n terms that work with unit groups shall link to required ODX unit group definitions by using *simple XLinks* only, as specified by [W3C XLink]. This means that the `xlink:type` attribute shall always be set to `"simple"`. Furthermore, the `xlink:href` attribute should follow the pattern `"{URI}#{SHORT-NAME}"`, where `{URI}` represents the `UNIT-SPEC` resource and `{SHORT-NAME}` identifies the unit group definition by its ODX `SHORT-NAME` property. The pattern corresponds to a *shorthand notation XPointer*, as specified by Clause 3.2 in [W3C XPtr]. However, in case the shorthand notation is not sufficient to address unit group definitions, the full XPointer notation may be used (e.g. when one ODX-document contains more than one `UNIT-SPEC` container).

EXAMPLE For linking to the unit definition for **`"EU_Metric"`** given in the exemplary **`UNIT-SPEC`** in 17.1, the element has the form of **`<unit xlink:type="simple" xlink:href="unit-spec.xml#EU_Metric]"/>`**.

This is related to the OTX Quantities extension – please refer to Clause 17 for further details.

12.5.4.3.2 LocalizeQuantity

The **`LocalizeQuantity`** term is used to create a localized version of a given **`Quantity`**.

A **`Quantity`** contains a value and display unit information. However, the display unit might be unsuitable for the current locale (e.g. when a distance-type quantity with a display unit of miles should be displayed to a user who is used to dealing with metric units). Because OTX test sequences should remain agnostic of localisation details, it is necessary to express conversions in such a way that both the display unit and the value of a **`Quantity`** can be localized in a consistent way.

NOTE The conversion must take into account factors such as the unit groups, known units and physical dimensions known to the system. From the point of view of the OTX sequence, quantities are just a data containers and the whole conversion process happens in the background. For naïve implementations, it is acceptable that the returned value is exactly the same as the given value.

Localization in the **LocalizeQuantity** term shall always be performed using the current locale.

LocalizeQuantity is a **quant:QuantityTerm**. Its members have the following semantics:

— **<quantity> : quant:QuantityTerm [1]**

This represents the **Quantity** that shall be localized to the current locale.

Throws:

— **quant:InvalidConversionException**
If the **Quantity** can not be converted for any reason.

12.5.4.3.3 LocalizeQuantityToUnitGroup

The **LocalizeQuantityToUnitGroup** term shall create a version of a **Quantity** localized to a specific unit group.

There are two different types of unit groups: country and equivalent unit groups. This term shall create a new **Quantity** containing the display unit given by the new group that is equivalent to the display unit of the original **Quantity**.

NOTE It is assumed that the runtime system contains a list of known and valid equivalent unit groups. In case that the runtime system decides to implement a naïve solution, it is valid to return the same quantity as the one that has been received.

LocalizeQuantityToUnitGroup is a **quant:QuantityTerm**. Its members have the following semantics:

— **<quantity> : quant:QuantityTerm [1]**

This represents the quantity that shall be localized using the given country unit group name.

— **<equivalentUnitGroup> : UnitGroupDefinition [1]**

Represents the **UNIT-GROUP** definition that shall be used as the target for unit localization.

The element allows all attributes from the namespace **"http://www.w3.org/1999/xlink"**, as defined by the W3C XLink recommendation [W3C XLink]. For the usage of the attributes, the rules given in 12.5.4.3.1 shall apply.

Throws:

— **quant:UnknownUnitException**
if the target unit group is unknown by the runtime system

— **quant:InvalidConversionException**
if the conversion is physically not possible (i.e. conversion from a length to a voltage measurement)

Associated checker rules:

— Quantities_Chk001 – correct unit linking

13 OTX Job extension

13.1 Introduction

The OTX Job extension is designed for supporting access to the MCD 3D Job API. This use case is highly specific to OTX sequences that are used for replacing Java Job code inside an MVCI system [ISO 22900]. A MVCI Java Job can be understood as a macro sequence providing low-level diagnostic business logic, e.g. for implementing security access challenge-response mechanisms. From the perspective of a tester application (on top of the MVCI API) a Java Job looks and behaves like a normal diagnostic service – it is identified by a name and has a set of request and response parameters. From the perspective of a Java Job this means that it is being called with a set of input parameters, and has to create and deliver a result structure back to its caller. This implies the following requirements for OTX sequences that take the place of a Java Job:

a) It needs a well defined entry point

b) It has to be able to create and parameterize result and response structures (as opposed to a tester application that only has to parameterize request structures)

c) It has to communicate results (and other information) to it's caller

These requirements are addressed by the OTX Job extension. As they are highly specific to the MVCI standard, in the following actions and terms sections directly refer to the MVCI standard documentation where applicable.

13.2 Exceptions

13.2.1 Overview

All elements referenced in this clause are derived from the OTX Core **`Exception`** type as defined by Part 2 of ISO 13209. They represent the full set of exceptions added by the OTX Job extension.

13.2.2 Syntax

The syntax of all exception types defined for the OTX Job extension is shown in Figure 78.

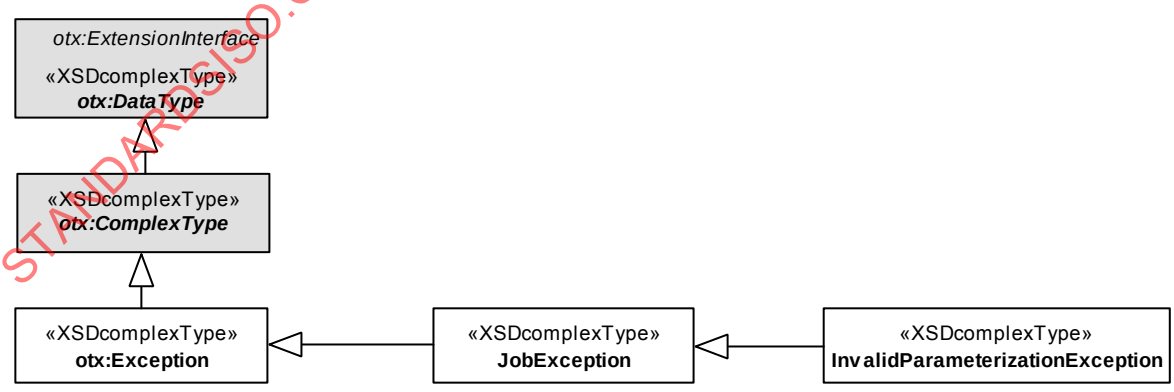

Figure 78 — Data model view: Job exceptions

13.2.3 Semantics

13.2.3.1 General

Since all OTX Job exception types are implicit exceptions whithout initialisation parts, they can not be declared constant.

13.2.3.2 JobException

The **`JobException`** is the super class for all exceptions in the Job extension. A **`JobException`** shall be used in case the more specific exception types described in the remainder of this section do not apply to the problem at hand.

13.2.3.3 InvalidParameterizationException

The **`InvalidParameterizationException`** shall be thrown when the type of one or more response paramters created by a Job OTX sequence do not match with the response database template of the underlying communication system.

For example when the OTX Job tries to add a response parameter A of type Integer where the response structure of the Job demands a parameter A of type String, the OTX runtime shall throw an **`InvalidParameterizationException`**.

13.3 Actions

13.3.1 Overview

The OTX Job extension introduces the actions **`SendFinalResult`**, **`SendIntermediateResult`**, **`SetJobInfo`**, **`SetProgressInfo`**, **`AddElement`**, **`AddBranchByName`**, **`AddBranchByIndex`**, **`AddBranchByValud`** and **`AddEnvDataByDtc`** as specified in the following.

13.3.2 Syntax

Figure 79 shows the syntax of all actions in the OTX Job extension.

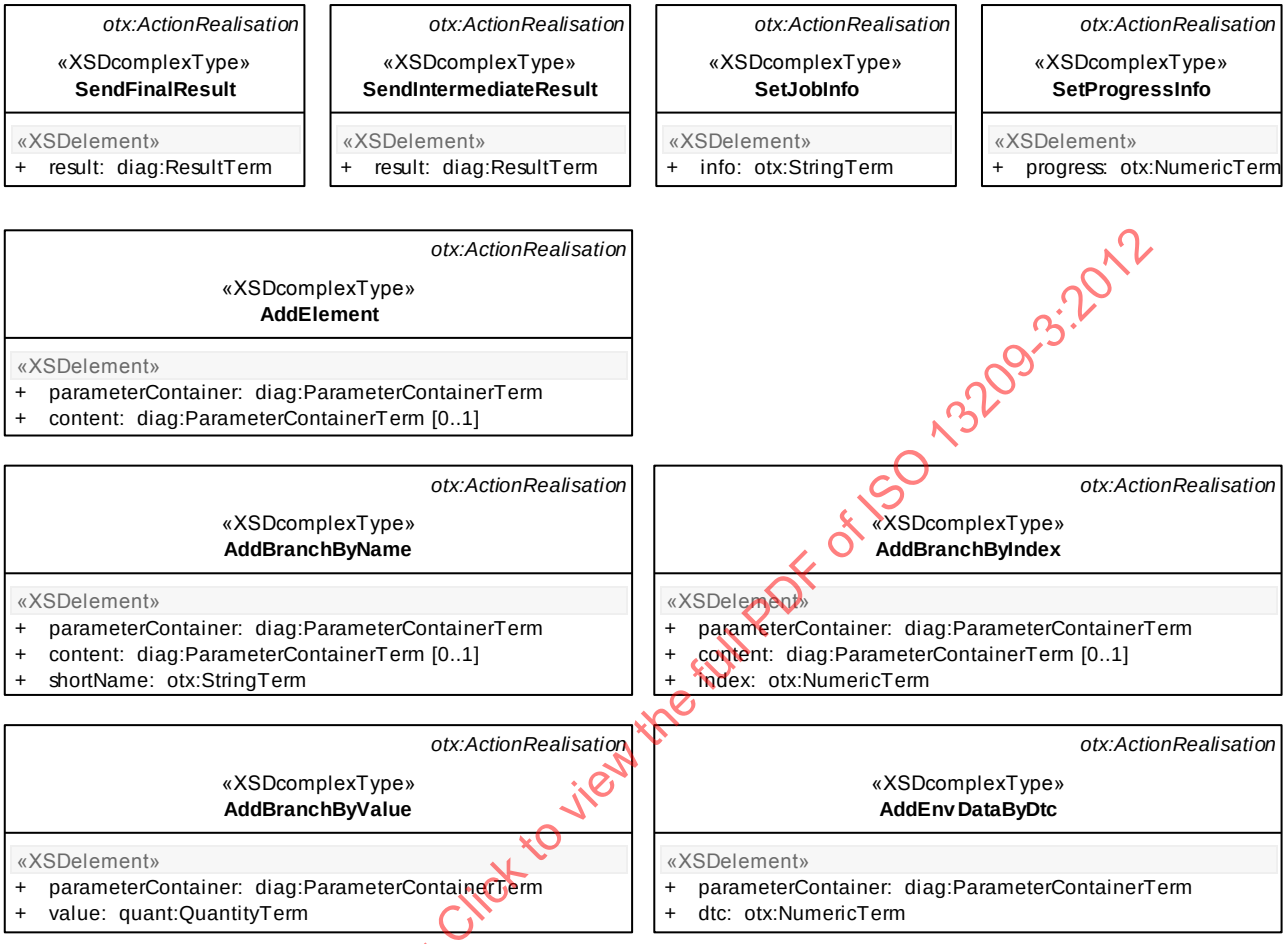

Figure 79 — Data model view: Job actions

13.3.3 Semantics

13.3.3.1 SendFinalResult

`SendFinalResult` shall send the last result before the job exits to the instance which started the job. Please also refer to MVCI API documentation for method **`MCDJobApi::sendFinalResult`**.

The members of **`SendFinalResult`** have the following semantics:

— **`<result> : diag:ResultTerm [1]`**

This represents the result which shall be sent to the tester. The term **`diag:ResultTerm`** is specified in the DiagCom extension (please refer to Clause 6).

Throws:

— **`otx:TypeMismatchException`**
If the result structure to be sent does not match with the result template defined for this Job by the underlying communication kernel.

— **`diag:InvalidStateException`**
If a final result has already been sent previously.

13.3.3.2 SendIntermediateResult

SendIntermediateResult shall send the value of a **diag:ResultTerm** as an intermediate result to the instance which started the job. Please also refer to the MVCI API documentation for the method **MCDJobApi::sendIntermediateResult**.

— **<result> : diag:ResultTerm [1]**

This represents the result which shall be sent to the tester as an intermediate result. The term **diag:ResultTerm** is specified in the DiagCom extension (please refer to Clause 6).

Throws:

— **otx:TypeMismatchException**
If the result structure to be sent does not match with the result template defined for this Job by the underlying communication kernel

— **diag:InvalidStateException**
If a final result has been sent previously (no more intermediate results allowed).

13.3.3.3 SetJobInfo

SetJobInfo shall set a job information which is sent to the calling tester. Please also refer to MVCI API documentation for method **MCDJobApi::setJobInfo**.

The members of **SetJobInfo** have the following semantics:

— **<info> : otx:StringTerm [1]**

Represents a string which will be sent to the tester as job information.

Throws:

— **diag:InvalidStateException**
If a final result has been sent previously (no more job info messages allowed).

13.3.3.4 SetProgressInfo

SetProgressInfo shall send progress information in the form of an integer value to the tester. The value represents a percentage to completion and should be a number between 0 and 100. In case the provided value is smaller than 0, the OTX runtime shall set the value to 0. In case the provided value is higher than 100, the OTX runtime shall set the value to 100. Please also refer to MVCI API documentation for method **MCDJobApi::setProgress**.

The members of **SetProgressInfo** have the following semantics:

— **<info> : otx:NumericTerm [1]**

Represents the progress information as percentage to completion (0..100). **Float** values shall be truncated.

Throws:

— **diag:InvalidStateException**
If a final result has been sent previously (no more progress info messages allowed).

13.3.3.5 AddElement

AddElement is used to add a parameter or parameter structure to a point in a response structure. Please also refer to MVCI API documentation for method **MCDResponseParameters::addElement** (in case that no **<content>** element is provided) and **MCDResponseParameters::addElementWithContent** (in case a **<content>** element is provided).

The members of **AddElement** have the following semantics:

— **<parameterContainer> : diag:ParameterContainerTerm [1]**

The parameter where new element(s) shall be added

— **<content> : diag:ParameterContainerTerm [0..1]**

The element(s) to be added to the **ParameterContainer**. If this element is omitted, the MVCI method **MCDResponseParameter::addElement** should be called. Otherwise the MVCI method **MCDResponseParameter::addElementWithContent** should be called.

Throws:

— **otx:TypeMismatchException**
If the element **<content>** to be added does not match the result template defined for this Job by the underlying communication kernel.

13.3.3.6 AddBranchByName

AddBranchByName is used to add a set of response parameters to the response structure according to a multiplexer database definition given by its short name. Please also refer to MVCI API documentation for method **MCDResponseParameters::addMuxBranch** (in case no **<content>** element is provided) and **MCDResponseParameters::addMuxBranchWithContent** (in case a **<content>** element is provided).

The members of **AddBranchByName** have the following semantics:

— **<parameterContainer> : diag:ParameterContainerTerm [1]**

The parameter where new element(s) shall be added.

— **<content> : diag:ParameterContainerTerm [0..1]**

The element(s) to be placed in the newly created multiplexer branch. If this element is omitted, the MVCI method **MCDResponseParameter::addMuxBranch** should be called. Otherwise the MVCI method **MCDResponseParameter::addMuxBranchWithContent** should be called.

— **<shortName> : otx:StringTerm [1]**

The name of the multiplexer branch to be added.

Throws:

— **otx:TypeMismatchException**
If the element **<content>** to be added does not match the result template defined for this Job by the underlying communication kernel.

— **InvalidParameterizationException**
If the **<shortName>** of the branch to be added does not match the result template defined for this Job by the underlying communication kernel.

13.3.3.7 AddBranchByIndex

AddBranchByIndex is used to add a set of response parameters to the response structure according to a multiplexer database definition given by its index. Please also refer to MVCI API documentation for method **MCDResponseParameters::addMuxBranchByIndex** (in case no **<content>** element is provided) and **MCDResponseParameters::addMuxBranchByIndexWithContent** (if **<content>** element is provided).

The members of **AddBranchByIndex** have the following semantics:

— **<parameterContainer> : diag:ParameterContainerTerm [1]**

The parameter where new element(s) shall be added.

— **<content> : diag:ParameterContainerTerm [0..1]**

The element(s) to be placed in the newly created multiplexer branch. If this element is omitted, the MVCI method **MCDResponseParameter::addMuxBranchByIndex** should be called. Otherwise the MVCI method ,,**MCDResponseParameter::addMuxBranchByIndexWithContent** should be called.

— **<index> : otx:NumericTerm [1]**

The index of the multiplexer branch to be added. **Float** values shall be truncated.

Throws:

— **otx:TypeMismatchException**
If the element **<content>** to be added does not match the result template defined for this Job by the underlying communication kernel.

— **otx:OutOfBoundsException**
If the **<index>** of the branch to be added does not exist in the result template defined for this Job by the underlying communication kernel.

13.3.3.8 AddBranchByValue

AddBranchByIndex is used to add a set of response parameters to the response structure according to a multiplexer parameter value. Please also refer to MVCI API documentation for method **MCDResponseParameters::addMuxBranchByMuxValue**.

The members of **AddBranchByValue** have the following semantics:

— **<parameterContainer> : diag:ParameterContainerTerm [1]**

This element represents the parameter where new element(s) shall be added.

— **<value> : quant:QuantityTerm [1]**

The value of the multiplexer parameter of the branch to be added.

Throws:

— **InvalidParameterizationException**
If the **<value>** of the multiplexer parameter to be added does not match the result template defined for this Job by the underlying communication kernel.

13.3.3.9 AddEnvDataByDtc

AddEnvDataByDtc is used to add an environment data parameter structure to the response structure according to the value of a DTC. Please also refer to MVCI API documentation for method **MCDResponseParameters::addEnvDataByDTC**.

The members of **AddEnvDataByDtc** have the following semantics:

— **<parameterContainer> : diag:ParameterContainerTerm [1]**

The parameter where new element(s) shall be added.

— **<dtc> : otx:NumericTerm [1]**

The value of the DTC for which environment data structures shall be added. **Float** values shall be truncated.

Throws:

— **InvalidParameterizationException**
If the **<dtc>** of the environment data to be added is not recognized by the underlying communication kernel.

13.3.4 Example

The example below shows uses of the actions defined for the OTX Job extension.

EXAMPLE Sample of OTX-file "JobActionsExample.otx"

```
<action id="a1">
  <specification>Send the final result ()stored in variable jobResult</specification>
  <realisation xsi:type="job:SendFinalResult">
    <job:result xsi:type="diag:ResultValue" valueOf="jobResult"/>
  </realisation>
</action>

<action id="a2">
  <specification>Send the intermediate result which is in variable jobResult</specification>
  <realisation xsi:type="job:SendIntermediateResult">
    <job:result xsi:type="diag:ResultValue" valueOf="jobResult"/>
  </realisation>
</action>

<action id="a3">
  <specification>Set the job information to "PENDING"</specification>
  <realisation xsi:type="job:SetJobInfo">
    <job:info xsi:type="StringLiteral" value="PENDING"/>
  </realisation>
</action>

<action id="a4">
  <specification>Set the progress to 50%</specification>
  <realisation xsi:type="job:SetProgressInfo">
    <job:progress xsi:type="IntegerLiteral" value="50"/>
  </realisation>
</action>
```

13.4 Terms

13.4.1 Overview

The terms introduced by the OTX Job extension are specified in the following.

13.4.2 Syntax

Figure 80 shows the syntax of all terms in the OTX Job extension.

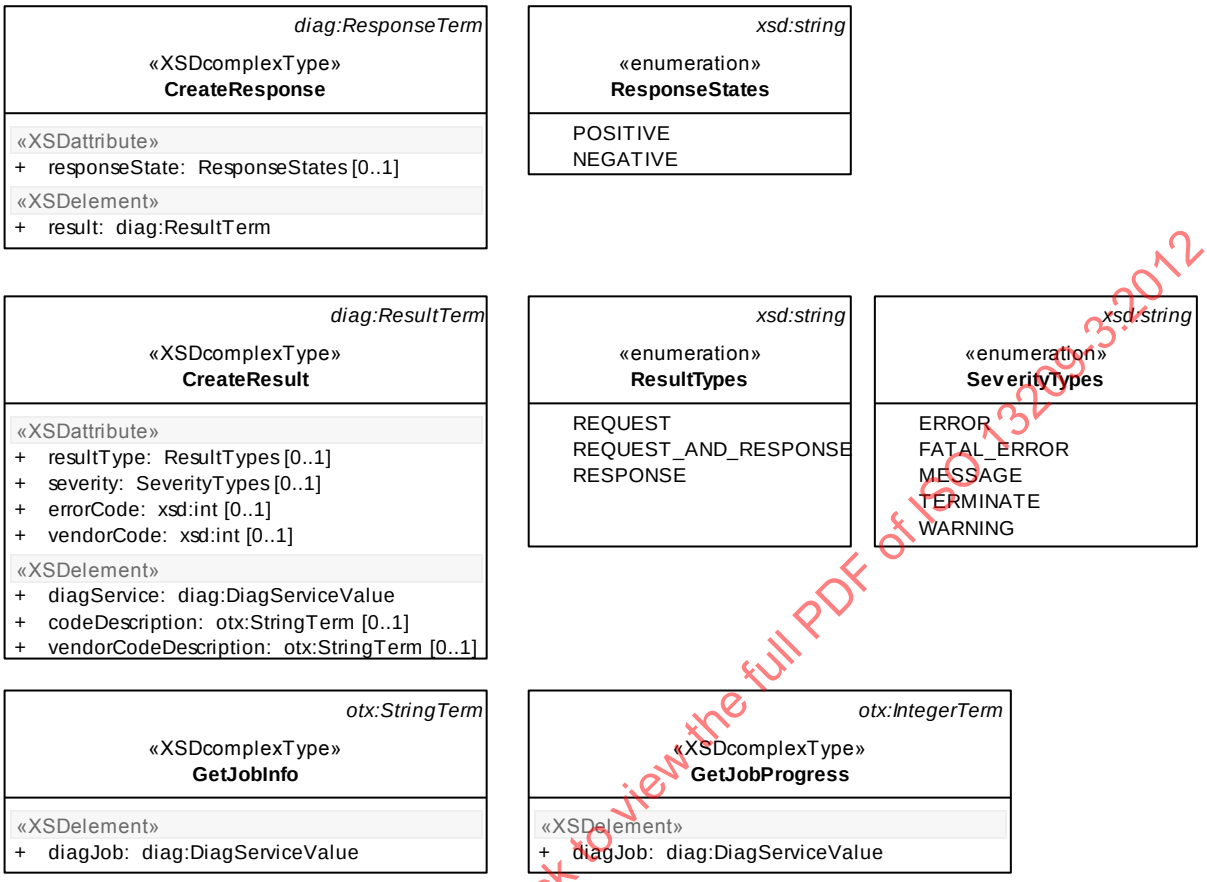

Figure 80 — Data model view: Job terms

13.4.3 Semantics

13.4.3.1 CreateResponse

`CreateResponse` creates a response inside a result object according to a given response state. Please also refer to MVCI API documentation for method **`MCDResponses::add`**.

`CreateResponse` is a **`diag:ResponseTerm`**. Its members have the following semantics:

— **`responseState : ResponseStates={POSITIVE|NEGATIVE} [0..1]`**

This optional attribute determines whether a positive or negative response should be created. Allowed values are defined by the **`ResponseState`** enumeration. If the attribute is not set, the default value **`POSITIVE`** shall apply implicitly.

— **`<result> : diag:ResultTerm [1]`**

The result the new response shall be added to. The term **`diag:ResultTerm`** is specified in the DiagCom extension (please refer to Clause 7).

Throws:

— **`InvalidParameterizationException`**
If the **`<responseState>`** of the respsonse to be added is invalid.

13.4.3.2 CreateResult

CreateResult creates a **Result** object that can be returned to the called of the OTX job sequence. Please also refer to MVCI API documentation for method **MCDJob::createResult.**

CreateResult is a **diag:ResultTerm**. Its members have the following semantics:

— **resultType : ResultTypes={REQUEST|RESPONSE|REQUEST_AND_RESPONSE} [0..1]**

The type of result which shall be created, allowed values are defined by the **ResultTypes** enumeration. If the attribute is not set, the default value **REQUEST_AND_RESPONSE** shall apply implicitly.

— **severity : SeverityTypes={ERROR|FATAL_ERROR|MESSAGE|TERMINATE|WARNING} [0..1]**

A severity which shall be associated with the result (if applicable), allowed values are defined by the **ResultTypes** enumeration. If the attribute is not set, the default value **ERROR** shall apply implicitly.

— **errorCode : xsd:int [0..1]**

An error code which shall be associated with the result (if applicable).

— **vendorCode : xsd:int [0..1]**

A vendor code which shall be associated with the result (if applicable).

— **<diagService> : diag:DiagServiceValue [1]**

Represents the **DiagService** on which the term operates and which the result shall be tailored to.

— **<codeDescription> : otx:StringTerm [0..1]**

A code description which shall be associated with the error code of the result (if applicable).

— **<vendorCodeDescription> : otx:StringTerm [0..1]**

A vendor code description to be associated with the vendor code of the result (if applicable).

Throws:

— **InvalidParameterizationException**
If the **<resultType>** or **<severity>** of the result to be added is invalid.

13.4.3.3 GetJobInfo

GetJobInfo retrieves information out of a DiagService object that encapsulates an OTX job. Please also refer to MVCI API documentation for method **MCDJob::getJobInfo**.

GetJobInfo is an **otx:StringTerm**. Its members have the following semantics:

— **<diagJob> : diag:DiagServiceValue [0..1]**

This diagnostic service object representing a job, where job information should be retrieved.

13.4.3.4 GetJobProgress

GetJobProgress retrieves progress information out of a DiagService object that encapsulates an OTX job. Please also refer to MVCI API documentation for method **MCDJob::getProgress**.

GetJobProgress is an **otx:IntegerTerm** Its members have the following semantics:

— **<diagJob> : diag:DiagServiceValue [0..1]**

This diagnostic service object represents the job from which job progress information shall be retrieved.

13.4.4 Example

The example below shows uses of the terms defined for the OTX Job extension.

EXAMPLE Sample of OTX-file "JobTermsExample.otx"

```
<?xml version="1.0" encoding="UTF-8"?>
<otx name="JobTermsExample" package="org.iso.otx.examples" id="jobTermEx"
  version="1.0" timestamp="2012-03-18T14:40:10" xmlns="http://iso.org/OTX/1.0.0"
  xmlns:diag="http://iso.org/OTX/1.0.0/DiagCom"
  xmlns:job="http://iso.org/OTX/1.0.0/Job"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <imports>
    <import package="org.iso.otx.signatures" document="JobInterfaces" prefix="jobsig"/>
  </imports>

  <procedures>
    <procedure name="execute" implements="jobsig:SingleEcuJob" visibility="PUBLIC" id="p1">
      <specification>Demonstration of OTX Job extension capabilities </specification>
      <realisation>
        <parameters>
          <inParam name="comChannelHandle" id="ip1">
            <realisation><dataType xsi:type="diag:ComChannel"/></realisation>
          </inParam>
          <inParam name="jobHandle" id="ip2">
            <realisation><dataType xsi:type="diag:DiagService"/></realisation>
          </inParam>
        </parameters>
        <declarations>
          <variable name="jobResult" id="v1">
            <realisation><dataType xsi:type="diag:Result"/></realisation>
          </variable>
          <variable name="jobResponse" id="v2">
            <realisation><dataType xsi:type="diag:Response"/></realisation>
          </variable>
        </declarations>
        <flow>
          <!-- do something useful here -->

          <action id="createResult">
            <specification>Create a result</specification>
            <realisation xsi:type="Assignment">
              <result xsi:type="diag:ResultVariable" name="jobResult"/>
              <term xsi:type="job:CreateResult">
                <job:diagService xsi:type="diag:DiagServiceValue" valueOf="jobHandle"/>
              </term>
            </realisation>
          </action>
          <action id="createResponse">
            <specification>Create a response and set response state </specification>
            <realisation xsi:type="Assignment">
              <result xsi:type="diag:ResponseVariable" name="jobResponse"/>
              <term xsi:type="job:CreateResponse" responseState="POSITIVE">
                <job:result xsi:type="diag:ResultValue" valueOf="jobResult"/>
              </term>
            </realisation>
          </action>
```

```
          <action id="setParameterValue">
            <specification>Set response parameter value</specification>
            <realisation xsi:type="diag:SetParameterValue">
              <diag:parameter xsi:type="diag:GetParameterByPath">
                <diag:parameterContainer xsi:type="diag:ResponseValue" valueOf="jobResponse"/>
                <diag:path>
                  <stepByName xsi:type="StringLiteral" value="ResponseParameterA"/>
                </diag:path>
              </diag:parameter>
              <diag:value xsi:type="IntegerLiteral" value="42"/>
            </realisation>
          </action>
          <action id="sendFinalResult">
            <specification>Send the result</specification>
            <realisation xsi:type="job:SendFinalResult">
              <job:result xsi:type="diag:ResultValue" valueOf="jobResult"/>
            </realisation>
          </action>
        </flow>
      </realisation>
    </procedure>
  </procedures>
</otx>
```

13.5 Standard signature definitions

13.5.1 General

The following specifies the standard signatures of the OTX Job extension. The signatures shall be part of every OTX application implementing the Job extension. They can be used by authors for developing local versions of single ecu jobs, flash jobs and security access jobs, following a uniform interface. By creating procedures which implement these signatures, exchangeability is improved.

The signatures are all part of the document **`org.iso.otx.signatures.JobInterfaces`**. Please refer to Annex D for the full XML specification (normative OTX instance document "JobInterfaces.otx").

13.5.2 SingleEcuJob

13.5.2.1 Description

The **`SingleEcuJob`** standard signature defines an interface for a job entry procedure which implements a so called Single ECU Job according to ISO 22901 (ISO ODX).

13.5.2.2 Declaration

The declaration of the **`SingleEcuJob`** signature is shown below. The signature is contained in the normative OTX instance document "JobInterfaces.otx" which is defined in Annex D.

```
  <signature name="SingleEcuJob" id="sej">
    <specification>Declaration of a single job entry point</specification>
    <realisation xsi:type="ProcedureSignature">
      <parameters>
        <inParam name="comChannelHandle" id="sej_p1">
          <specification> Communication Channel from Tester Environment</specification>
          <realisation>
            <dataType xsi:type="diag:ComChannel"/>
          </realisation>
        </inParam>
        <inoutParam name="jobHandle" id="sej_p2">
          <specification> Job handle with parameters from outside </specification>
          <realisation>
            <dataType xsi:type="diag:DiagService"/>
          </realisation>
        </inoutParam>
      </parameters>
    </realisation>
  </signature>
```

13.5.2.3 Semantics

The parameters of the **FlashJob** standard signature have the following semantics:

— **comChannelHandle : ComChannel [inParam]**

This input parameter represents the communication channel on which the job shall operate.

— **<jobHandle> : DiagService [inoutParam]**

This input/output parameter represents the job handle which shall be used to store the input and output parameters of the job.

13.5.3 FlashJob

13.5.3.1 Description

The **FlashJob** standard signature defines an interface for a job which shall be used by a diagnostic application for updating an ECU's flash memory.

13.5.3.2 Declaration

The declaration of the **FlashJob** signature is shown below. The signature is contained in the normative OTX instance document "JobInterfaces.otx" which is defined in Annex D.

```
<signature name="FlashJob" id="fj">
  <specification>Declaration of a flash job entry point</specification>
  <realisation xsi:type="ProcedureSignature">
    <parameters>
      <inParam name="comChannelHandle" id="fj_p1">
        <specification> Communication Channel from Tester Environment</specification>
        <realisation>
          <dataType xsi:type="diag:ComChannel"/>
        </realisation>
      </inParam>
      <inParam name="Session" id="fj_p2">
        <specification>Session variable from Tester</specification>
        <realisation>
          <dataType xsi:type="flash:FlashSession"/>
        </realisation>
      </inParam>
      <inoutParam name="jobHandle" id="fj_p3">
        <specification> Job handle with parameters from outside </specification>
        <realisation>
          <dataType xsi:type="diag:DiagService"/>
        </realisation>
      </inoutParam>
    </parameters>
  </realisation>
</signature>
```

13.5.3.3 Semantics

The parameters of the **FlashJob** standard signature have the following semantics:

— **comChannelHandle : ComChannel [inParam]**

This input parameter represents the communication channel on which the job shall operate.

— **jobHandle : DiagService [inoutParam]**

This input/output parameter represents the job handle (a **DiagService** object) which shall be used to store the input and output parameters of the job.

— **sessionHandle : FlashSession [inParam]**

This input Parameter represents the **FlashSession** on which the job shall operate.

13.5.4 SecurityAccessJob

13.5.4.1 Description

The **SecurityAccessJob** standard signature defines an interface which shall be used calculating an ECU's security access.

13.5.4.2 Declaration

The declaration of the **SecurityAccessJob** signature is shown below. The signature is contained in the normative OTX instance document "JobInterfaces.otx" which is defined in Annex D.

```
<signature name="SecurityAccessJob" id="saj">
  <specification>Declaration of a security access job entry point</specification>
  <realisation xsi:type="ProcedureSignature">
    <parameters>
      <inParam name="requestParameters" id="saj_p1">
        <realisation>
          <dataType xsi:type="diag:Request"/>
        </realisation>
      </inParam>
      <inParam name="comChannelHandle" id="saj_p2">
        <specification> Communication Channel from Tester Environment</specification>
        <realisation>
          <dataType xsi:type="diag:ComChannel"/>
        </realisation>
      </inParam>
      <inoutParam name="jobHandle" id="saj_p3">
        <specification> Job handle with parameters from outside </specification>
        <realisation>
          <dataType xsi:type="diag:DiagService"/>
        </realisation>
      </inoutParam>
    </parameters>
  </realisation>
</signature>
```

13.5.4.3 Semantics

The parameters of the **SecurityAccessJob** standard signature have the following semantics:

— **comChannelHandle : ComChannel [inParam]**

This input parameter represents the communication channel on which the job shall operate.

— **jobHandle : DiagService [inoutParam]**

This input/output parameter represents the job handle (a **DiagService** object) which shall be used to store the input and output parameters of the job.

— **requestParameters : Request [inParam]**

This input parameter represents the **Request** which contains the information to calculate the security access.

14 OTX Logging extension

14.1 Introduction

The OTX Logging extension provides functionality which allows for explicitly writing log-messages to a logging-resource.

Following the approach of the de-facto-standard *log4j* (which is a Java™-based logging framework), the extension uses so called *severity-levels* which are associated to log-messages, and *log-levels* which can be set in the logging framework. Depending on the currently set log-level and the severity-level of a log-message fired by an OTX sequence, the message gets logged or is discarded. For that reason, a log-level represents a certain threshold which must be exceeded by the severity-level of a log-message in order to be written into the logging-resource.

The severity-levels for log-messages are shown in Table 5 (in decreasing order of severity).

Table 5 — Severity-levels

Severity	Description
FATAL	Severe errors that cause premature termination.
ERROR	Other runtime errors or unexpected conditions.
WARN	Other runtime situations that are undesirable or unexpected, but not necessarily "wrong".
INFO	Interesting runtime events.
DEBUG	Detailed information on the flow through the sequence.
TRACE	Even more detailed information.

Available log-levels are shown in Table 6 (in increasing order of logging verbosity).

Table 6 — Log-levels

Thresold	Description
OFF	Nothing will be logged.
FATAL	Messages with severity **FATAL** will be logged.
ERROR	Messages with severity **ERROR** or above will be logged.
WARN	Messages with severity **WARN** or above will be logged.
INFO	Messages with severity **INFO** or above will be logged.
DEBUG	Messages with severity **DEBUG** or above will be logged.
TRACE	Messages with severity **TRACE** or above will be logged.
ALL	All messages will be logged (this is the default setting).

OTX authors may control which kind of log-messages make it into the logfile and which not by simply setting the log-level to the desired threshold value. For instance, if the current log-level is set to **ERROR**, a log-message with a severity of **FATAL** passes the log-level threshold, whereas a log-message with a rather uninteresting severity of **TRACE** will not pass the threshold.

NOTE The OTX Logging extension makes no assumptions nor does it define any rules concerning the resource into which log-messages are written. It is entirely up to the specific OTX application whether the messages are written to a

text-file, a log-queue or a database etc. Also the extension does not define any actions for the handling of the log-resource like e.g. clearing the log etc. OTX applications may provide specific functionality for such use cases.

14.2 Data types

14.2.1 Overview

The OTX Logging extension defines two data types. These are the enumerations **`LogLevel`** and **`SeverityLevel`**.

14.2.2 Syntax

The syntax of the datatype declarations of the OTX Logging extension is shown in Figure 81.

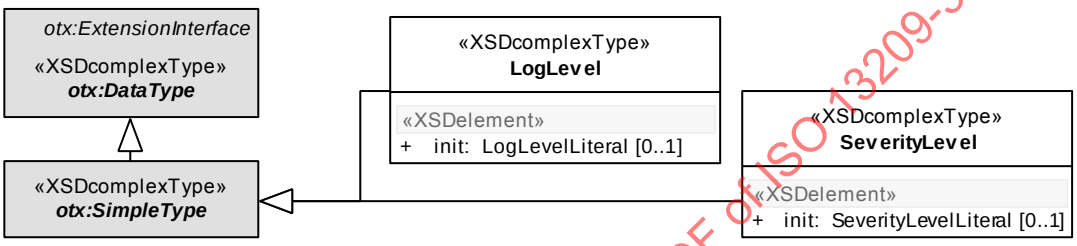

Figure 81 — Data model view: Logging data types

14.2.3 Semantics

14.2.3.1 General

The enumeration types in the OTX Logging extension are based on **`otx:SimpleType`**.

14.2.3.2 LogLevel

`LogLevel` is an enumeration type describing log thresholds used by the **`SetLogLevel`** action (cf. 14.4.3.1). The allowed enumeration values are specified in Table 6.

IMPORTANT — `LogLevel` values may occur as operands of comparisons (cf. Part 2 of ISO 13209, relational operations). For this case, the following order relation shall apply:
`ALL < TRACE < DEBUG < INFO < WARN < ERROR < FATAL < OFF`.

IMPORTANT — When applying `otx:ToString` on a `LogLevel` value, the resulting string shall be the name of the enumeration value, e.g. `otx:ToString(TRACE)="TRACE"`. Furthermore, applying `otx:ToInteger` shall return the index of the value in the `LogLevels` enumeration (smallest index is `0`). The behaviour is undefined for other conversion terms (cf. Part 2 of ISO 13209).

`LogLevel` is an **`otx:SimpleType`**. Its members have the following semantics:

— **`<init> : LogLevelLiteral [0..1]`**

This optional element stands for the hard-coded initialisation value of the identifier at declaration time.

— **`value : LogLevels={ALL|TRACE|DEBUG|INFO|WARN|ERROR|FATAL|OFF} [1]`**

This attribute shall contain one of the values defined in the **`LogLevels`** enumeration.

IMPORTANT — If the `LogLevel` declaration is not explicitly initialized (omitted `<init>` element), the default value shall be `ALL`.

14.2.3.3 SeverityLevel

SeverityLevel is an enumeration type describing the severity of a log message written by a **WriteLog** action (cf. 14.4.3.2). The allowed enumeration values are specified in Table 5.

IMPORTANT — SeverityLevel values may occur as operands of comparisons (cf. Part 2 of ISO 13209, relational operations). For this case, the following order relation shall apply:
TRACE < DEBUG < INFO < WARN < ERROR < FATAL.

IMPORTANT — When applying otx:ToString on a SeverityLevel value, the resulting string shall be the name of the enumeration value, e.g. otx:ToString(TRACE)="TRACE". Furthermore, applying otx:ToInteger shall return the index of the value in the SeverityLevels enumeration (smallest index is 0). The behaviour is undefined for other conversion terms (cf. Part 2 of ISO 13209).

SeverityLevel is an **otx:SimpleType**. Its members have the following semantics:

— **<init> : SeverityLevelLiteral [0..1]**

This optional element stands for the hard-coded initialisation value of the identifier at declaration time.

— **value : SeverityLevels={TRACE|DEBUG|INFO|WARN|ERROR|FATAL} [1]**

This attribute shall contain one of the values defined in the **SeverityLevels** enumeration.

IMPORTANT — If the SeverityLevel declaration is not explicitly initialized (omitted <init> element), the default value shall be TRACE.

14.3 Variable access

14.3.1 Overview

As specified in Part 2 of ISO 13209, OTX extensions shall define a variable access type for each datatype they define (exception types inclusively). All variable access types are derived from the OTX Core **Variable** extension interface. The following specifies all variable access types defined for the Logging extension.

14.3.2 Syntax

Figure 82 shows the syntax of the Logging extension's variable access types.

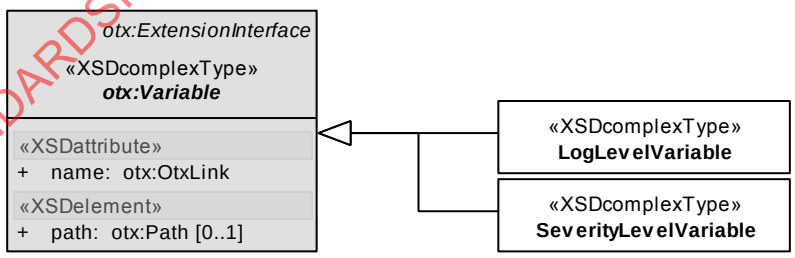

Figure 82 — Data model view: Logging variable access types

14.3.3 Semantics

The general semantics for all variable access types shall apply. Please refer to Part 2 of ISO 13209 for details.

14.4 Actions

14.4.1 Overview

There are two complementary action types defined for the OTX Logging extension: **`SetLogLevel`** for setting the log-level and **`WriteLog`** for writing a log-message to a file. Both types extend the **`ActionRealisation`** extension interface as defined by Part 2 of ISO 13209.

14.4.2 Syntax

Figure 83 shows the syntax of all actions in the OTX Logging extension.

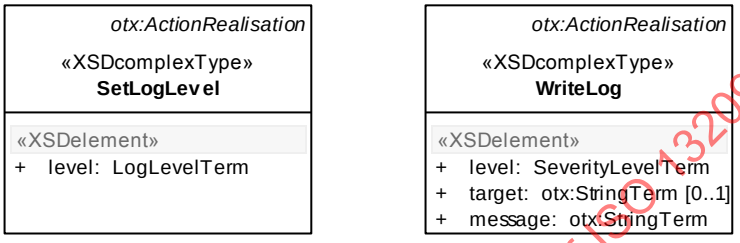

Figure 83 — Data model view: Logging actions

14.4.3 Semantics

14.4.3.1 SetLogLevel

As outlined in the introduction (see 14.1), the **`SetLogLevel`** action shall cause the OTX runtime system to set the log-level threshold to a given value.

The members of **`SetLogLevel`** have the following semantics:

— **`<level> : LogLevelTerm [1]`**

This element represents the log-level which shall be set in the OTX runtime's logging framework (cf. 14.2.3.2 for **`LogLevel`** values).

14.4.3.2 WriteLog

As outlined in the introduction (see 14.1), the **`WriteLog`** action shall cause the OTX runtime system to write a log-message into a logging-resource provided that the severity-level of that message is higher or equal than the currently set log-level threshold. The particular logging-resource to which the log-message shall be written may be identified by the optional **`<target>`** element. Otherwise (if no explicit target is given), the location of the logging-resource depends on the specific runtime system settings.

The members of **`WriteLog`** have the following semantics:

— **`<level> : SeverityLevelTerm [1]`**

This element represents the severity-level of the log-message (cf. 14.2.3.3 for **`SeverityLevel`** values).

— **`<target> : otx:StringTerm [0..1]`**

The optional element shall be used for locating the resource to which the message shall be written. The target should be defined by a URI. Other resource-location mechanisms may also be used.

— **<message> : otx:StringTerm [1]**

This string value represents the log-message. The OTX runtime shall compare the severity-level of the message to the current log-level after the rules given in Table 6. If the conditions for writing the message hold, the log-message shall be appended to the logging-resource.

Throws:

— **otx:InvalidReferenceException**
If the log-resource given by the **<target>** element is not available or not accessible.

14.4.4 Example

The usage of **SetLogLevel** and **WriteLog** is shown below. First, the log-level is set to **"ERROR"**, then two log-messages with severity-level **"INFO"** resp. **"FATAL"** are triggered. The first message's severity does not pass the log-level threshold, so only the latter message will be logged.

EXAMPLE Sample of OTX-file "LoggingExample.otx"

```
<?xml version="1.0" encoding="UTF-8"?>
<otx name="LoggingExample" package="org.iso.otx.examples" id="otx1"
  version="1.0" timestamp="2010-03-18T14:40:10"
  xmlns="http://iso.org/OTX/1.0.0"
  xmlns:log="http://iso.org/OTX/1.0.0/Logging"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

  xsi:schemaLocation="http://iso.org/OTX/1.0.0 Core/otx.xsd
  http://iso.org/OTX/1.0.0/Logging StandardInterface/otxIFD_Logging.xsd">

  <procedures>
    <procedure name="main" visibility="PUBLIC" id="p1">
      <specification>Demonstration of OTX Logging extension capabilities</specification>
      <realisation>
        <flow>

          <action id="a1">
            <specification>Set log-level to ERROR</specification>
            <realisation xsi:type="log:SetLogLevel">
              <log:level xsi:type="log:LogLevelLiteral" value="ERROR"/>
            </realisation>
          </action>

          <action id="a2">
            <specification>Trigger a INFO log-message</specification>
            <realisation xsi:type="log:WriteLog">
              <log:level xsi:type="log:SeverityLevelLiteral" value="INFO"/>
              <log:target xsi:type="StringLiteral" value="file:///c:/myLog.txt"/>
              <log:message xsi:type="StringLiteral" value="This will not be logged."/>
            </realisation>
          </action>

          <action id="a3">
            <specification>Trigger a FATAL log-message</specification>
            <realisation xsi:type="log:WriteLog">
              <log:level xsi:type="log:SeverityLevelLiteral" value="FATAL"/>
              <log:target xsi:type="StringLiteral" value="file:///c:/myLog.txt"/>
              <log:message xsi:type="StringLiteral" value="Houston, we have a problem."/>
            </realisation>
          </action>

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

14.5 Terms

14.5.1 Overview

The terms of the OTX Logging extension are related to the handling of the enumerations **`LogLevel`** and **`SeverityLevel`** (see 14.2).

14.5.2 Syntax

Figure 84 shows the syntax of all terms in the OTX Logging extension.

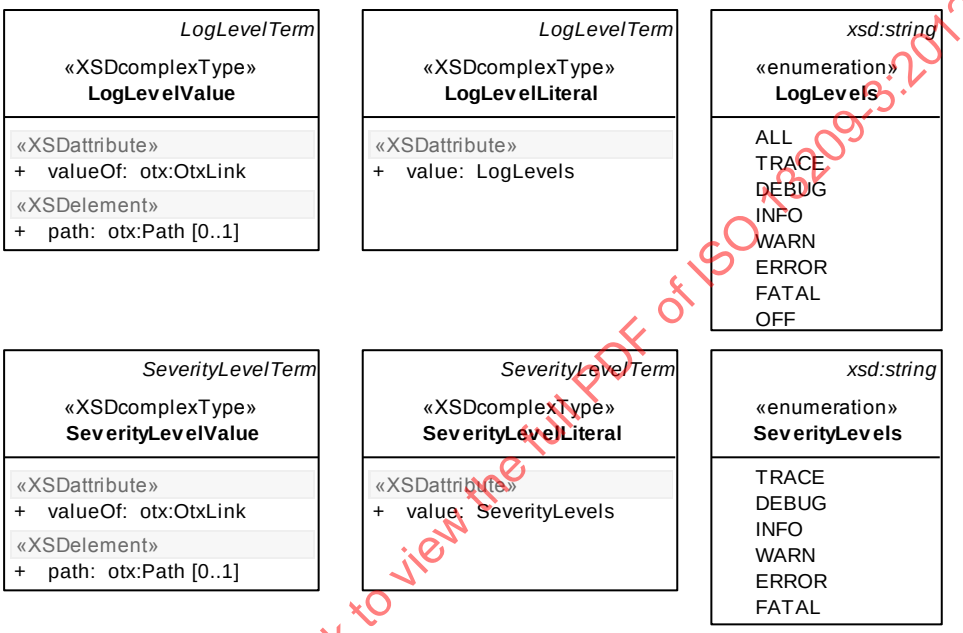

Figure 84 — Data model view: Logging terms

14.5.3 Semantics

14.5.3.1 LogLevelTerm

The abstract type **`LogLevelTerm`** is an **`otx:SimpleTerm`**. It serves as a base for all concrete terms which return a **`LogLevel`** value (see 14.2.3.2). It has no special members.

14.5.3.2 LogLevelValue

This term returns the **`LogLevel`** stored in a **`LogLevel`** variable. For more information on value-terms and the syntax and semantics of the **`valueOf`** attribute and **`<path>`** element, please refer to Part 2 of ISO 13209.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

Throws:

— **`otx:OutOfBoundsException`**
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).

14.5.3.3 LogLevelLiteral

This term shall return a **LogLevel** value (see 14.2.3.2) from a hard-coded literal.

LogLevelLiteral is a **LogLevelTerm**. Its members have the following semantics:

— **value : LogLevels={ALL|TRACE|DEBUG|INFO|WARN|ERROR|FATAL|OFF} [1]**

This attribute shall contain one of the values defined in the **LogLevels** enumeration.

14.5.3.4 SeverityLevelTerm

The abstract type **SeverityLevelTerm** is an **otx:SimpleTerm**. It serves as a base for all concrete terms which return a **SeverityLevel** value (see 14.2.3.3). It has no special members.

14.5.3.5 SeverityLevelValue

This term returns the **SeverityLevel** stored in a **SeverityLevel** variable. For more information on value-terms and the syntax and semantics of the **valueOf** attribute and **<path>** element, please refer to Part 2 of ISO 13209.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)#

Throws:

— **otx:OutOfBoundsException**
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).

14.5.3.6 SeverityLevelLiteral

This term shall return a **SeverityLevel** value (see 14.2.3.3) from a hard-coded literal.

SeverityLevelLiteral is a **SeverityLevelTerm**. Its members have the following semantics:

— **value : SeverityLevels={TRACE|DEBUG|INFO|WARN|ERROR|FATAL} [1]**

This attribute shall contain one of the values defined in the **SeverityLevels** enumeration.

15 OTX Math extension

15.1 Introduction

This OTX extension provides a collection of mathematical terms which are not covered by the OTX Core but may be required for some use cases.

15.2 Terms

15.2.1 Overview

The OTX Math extension provides terms which OTX authors may use for trigonometric, logarithmic and exponential calculations. Since all terms specified here return **`Float`** type values they are derived from the **`otx:FloatTerm`** type as defined by Part 2 of ISO 13209.

15.2.2 Syntax

Figure 85 shows the syntax of all terms of the Math extension.

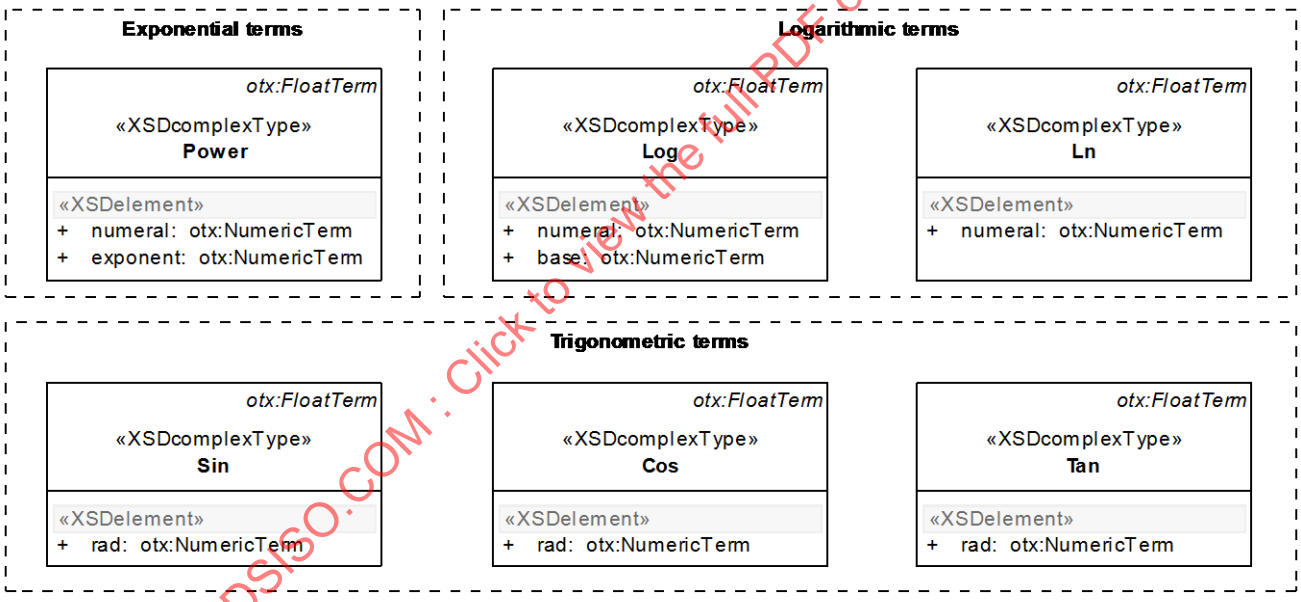

Figure 85 — Data model view: Math terms

15.2.3 Semantics

15.2.3.1 Power

The **`Power`** term shall be used for calculating the power of a given number using a given exponent.

`Power` is an **`otx:FloatTerm`**. Its members have the following semantics:

— **`<numeral> : otx:NumericTerm [1]`**

Represents the numeric value on which the power shall be calculated.

— **`<exponent> : otx:NumericTerm [1]`**

Represents the numeric value which shall serve as the exponent in the calculation.

IMPORTANT — Special cases concerning the `<numeral>` and `<exponent>` arguments shall be taken into account (e.g. special `Float` values like `0`, `-0`, `INF`, `-INF` and `NaN` as well as special argument combinations). The reference implementation for these special cases is the Java method `java.lang.Math.pow(double a, double b)`. There shall be no deviation from this implementation.

15.2.3.2 Log

The **`Log`** term shall be used for calculating the logarithm of a given number to a given base.

`Log` is an **`otx:FloatTerm`**. Its members have the following semantics:

— **`<numeral> : otx:NumericTerm [1]`**

Represents the numeric value on which the logarithm shall be calculated. If the value is **`Integer`**, it shall be automatically promoted to **`Float`**.

— **`<base> : otx:NumericTerm [1]`**

Represents the numeric value which shall serve as the logarithmic base of the calculation. If the value is **`Integer`**, it shall be automatically promoted to **`Float`**.

IMPORTANT — Special cases concerning the `<numeral>` and `<base>` arguments shall be taken into account (e.g. special `Float` values like `0`, `-0`, `INF`, `-INF` and `NaN` as well as special argument combinations). The reference implementation for these special cases is the Java method `java.lang.Math.log(double a)` in combination with the Java /-operator (since `java.Math` only provides the natural logarithm, the OTX `Log(base,numeral)` equals Java `log(numeral)/log(base)`). There shall be no deviation from this implementation.

15.2.3.3 Ln

The **`Ln`** term shall be used for calculating the natural logarithm of a given number.

`Ln` is an **`otx:FloatTerm`**. Its members have the following semantics:

— **`<numeral> : otx:NumericTerm [1]`**

Represents the numeric value on which the logarithm shall be calculated.

IMPORTANT — Special cases concerning the `<numeral>` argument shall be taken into account (e.g. special `Float` values like `0`, `-0`, `INF`, `-INF` and `NaN`). The reference implementation for these special cases is the Java method `java.lang.Math.log(double a)`. There shall be no deviation from this implementation.

15.2.3.4 Sin

The **`Sin`** term shall be used for calculating the sine of a given angle (in radians).

`Sin` is an **`otx:FloatTerm`**. Its members have the following semantics:

— **`<rad> : otx:NumericTerm [1]`**

Represents the angle from which the sine shall be calculated (radian interpretation).

IMPORTANT — Special cases concerning the `<rad>` argument shall be taken into account (e.g. special `Float` values like `0`, `-0`, `INF`, `-INF` and `NaN`). The reference implementation for these special cases is the Java method `java.lang.Math.sin(double a)`. There shall be no deviation from this implementation.

15.2.3.5 Cos

The **Cos** term shall be used for calculating the cosine of a given angle (in radians).

Cos is an **otx:FloatTerm**. Its members have the following semantics:

— **<rad> : otx:NumericTerm [1]**

Represents the angle from which the cosine shall be calculated (radian interpretation).

IMPORTANT — Special cases concerning the <rad> argument shall be taken into account (e.g. special Float values like 0, -0, INF, -INF and NaN). The reference implementation for these special cases is the Java method java.lang.Math.cos(double a). There shall be no deviation from this implementation.

15.2.3.6 Tan

The **Tan** term shall be used for calculating the tangent of a given angle (in radians).

Tan is an **otx:FloatTerm**. Its members have the following semantics:

— **<rad> : otx:NumericTerm [1]**

Represents the angle from which the tangent shall be calculated (radian interpretation)

IMPORTANT — Special cases concerning the <rad> argument shall be taken into account (e.g. special Float values like 0, -0, INF, -INF and NaN). The reference implementation for these special cases is the Java method java.lang.Math.tan(double a). There shall be no deviation from this implementation.

16 OTX Measure extension

16.1 Introduction

The OTX Measure extension provides actions, terms and data types for basic measurement and control operations. Its purpose is to extend OTX to the requirements of vehicle manufacturing.

In manufacturing a significant amount of the overall test steps are electric and electronic measurement and control actions that are not related to a standardised diagnostic ECU-communication as it is described in the OTX DiagCom extension. The OTX Measure extension shall serve as a simple interface to describe these electronic and electric measurement and control actions.

16.2 Data types

16.2.1 Overview

The OTX Measure extension introduces a single data type named **`Measurement`**, as described in the following.

16.2.2 Syntax

The syntax of the **`Measurement`** datatype declaration of the OTX Measure extension is shown in Figure 86.

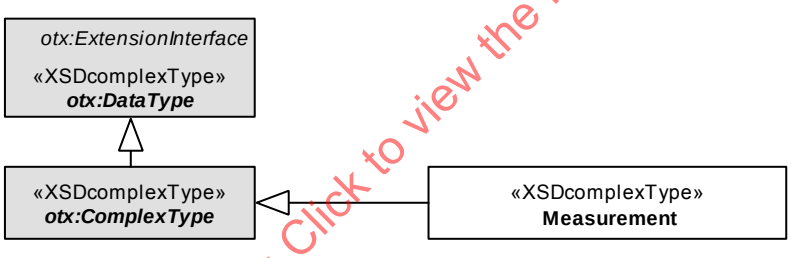

Figure 86 — Data model view: Measure data types

16.2.3 Semantics

16.2.3.1 General

Since the OTX Measure data types have no initialisation parts, they can not be declared constant.

16.2.3.2 Measurement

Measurement serves as container for a specific measurement. It includes a timestamp of the measurement, the status of the measurement and the measured quantity.

The internal properties of a measurement can be acquired by the terms **`GetMeasurementQuantity`**, **`GetMeasurementTimestamp`**, **`GetMeasurementStatus`**, **`IsValidMeasurement`** as well as **`GetMeasurementValue`**.

Since the **`Measurement`** data type has no initialisation parts, a **`Measurement`** can not be declared constant.

16.3 Exceptions

16.3.1 Overview

All elements referenced in this clause are derived from the OTX Core **`Exception`** type as defined by Part 2 of ISO 13209. They represent the full set of exceptions added by the OTX Measure extension.

16.3.2 Syntax

The syntax of all OTX Measure exception type declarations is shown in Figure 87.

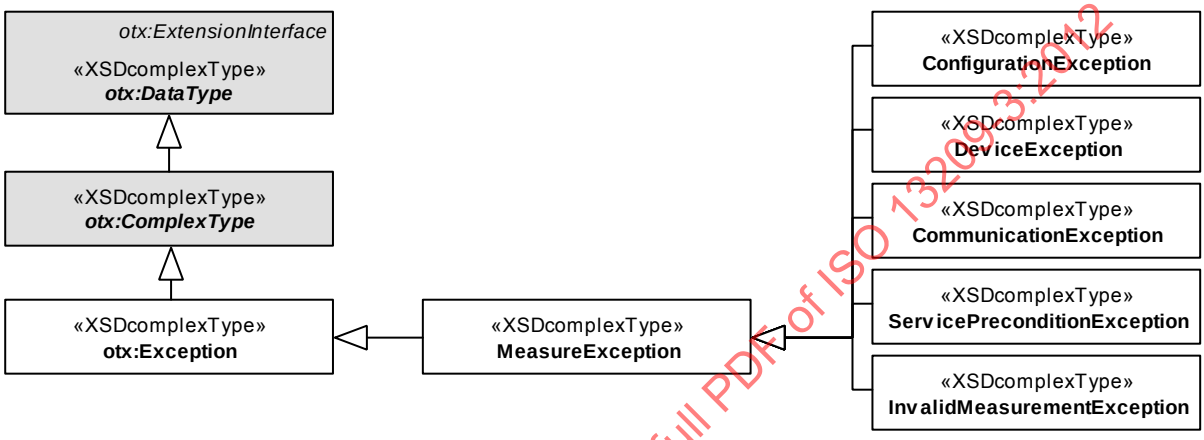

Figure 87 — Data model view: Measure exceptions

16.3.3 Semantics

16.3.3.1 General

Since all OTX Measure exception types are implicit exceptions whithout initialisation parts, they can not be declared constant.

16.3.3.2 MeasureException

The **`MeasureException`** is the super class for all exceptions in the Measure extension. A **`MeasureException`** shall be used in case the more specific exception types described in the remainder of this section do not apply to the problem at hand.

16.3.3.3 ConfigurationException

A **`ConfigurationException`** is thrown if there is a configuration problem, e.g. if a channel is not a legal channel for an intended operation.

16.3.3.4 CommunicationException

A **`CommunicationException`** is thrown in case when the communication to a device did not succeed, e.g. there is no answer from the device or an error occurred in the communication infrastructure.

16.3.3.5 DeviceException

A **`DeviceException`** is thrown if there is a measurement device problem. The physical device is reachable but has problems and sends a hint, e.g. that a contact is broken.

16.3.3.6 ServicePreconditionException

The **`ServicePreconditionException`** is thrown if a precondition is not met which is vital for the execution of the demanded device service. This may happen e.g. if the minimal speed is not yet reached for a break test.

16.3.3.7 InvalidMeasurementException

The **`InvalidMeasurementException`** shall be thrown when an invalid measurement is received from a measurement device. By contrast to **`ServicePreconditionException`** (see above), a thrown **`InvalidMeasurementException`** means that the measurement device is fine, but but the measured value is nevertheless regarded as invalid by the device. Since there are cases where invalid measurements pose to exceptional situation or the production of additional return values shall not be hindered, the throw of this exception can be controlled by the optional **`suppressInvalidMeasurementException`** flag of **`ExecuteDeviceService`** action (see 16.6.3.2).

16.4 Variable access

16.4.1 Overview

As specified in Part 2 of ISO 13209, OTX extensions shall define a variable access type for each datatype they define (exception types inclusively). All variable access types are derived from the OTX Core **`Variable`** extension interface. The following specifies all variable access types defined for the i18n extension.

16.4.2 Syntax

Figure 88 shows the syntax of the Measure extension's variable access types.

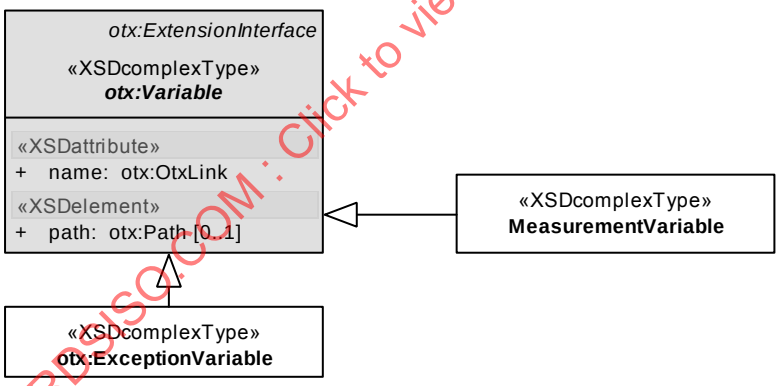

Figure 88 — Data model view: Measure variable access types

16.4.3 Semantics

The general semantics for all variable access types shall apply. Please refer to Part 2 of ISO 13209 for further details.

16.5 Signatures

16.5.1 Overview

As specified by Part 2 of ISO 13209, OTX extensions may define new specialized types of signatures by extending **`otx:SignatureRealisation`**. The OTX Measure extension uses this extensibility by adding the

DeviceSignature type which allows in-document, high-level interface specifications of measurement devices and their capabilities.

16.5.2 Syntax

Figure 89 shows the syntax of the Measure extension's signature types.

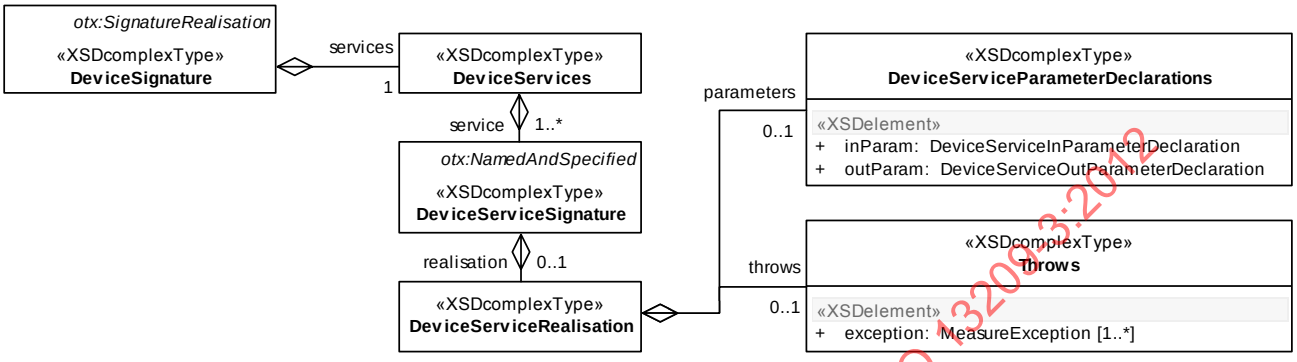

Figure 89 — Data model view: Measure signatures

NOTE The XSD complex type **DeviceServiceParameterDeclarations** is of **<xsd:choice> [1..*]** content-type, which is not explicitly shown in the figure above.

16.5.3 Semantics

16.5.3.1 General

The basic semantics common to all kinds of signatures are specified in Part 2 of ISO 13209.

16.5.3.2 DeviceSignature

Each measurement device which is attached to a diagnostic application and which is used by an OTX test sequence shall be described by a device signature.

The device ids (the signature's **name** attribute) are symbolic and have to be mapped by some runtime configuration to the concrete measurement and control device drivers. For the mapping, the use of a signature's meta data element is recommended by this standard.

A **DeviceSignature** describes the measurement and control interface of a measurement device or probe. It comprises a collection of (sub-)signatures for each device service that can be called by an **ExecuteDeviceService** action (see 16.6.3.2). The parameters described for such a service serve as a blueprint for the arguments of an **ExecuteDeviceService** action.

DeviceSignature is an **otx:SignatureRealisation**. Its members have the following semantics:

— **<services> : DeviceServices [0..1]**

This container element holds a collection of device service signatures describing the set of services available for a measurement device.

— **<service> : DeviceServiceSignature [1..*]**

Describes a measurement device's service that can be called by an **ExecuteDeviceService**.

— **name : otx:OtxName [0..1]** (derived from **otx:NamedAndSpecified**)

This represents the service's name. **ExecuteDeviceService** actions shall identify the to-be-executed device service by using this.

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

This optional string should be used by OTX authors to specify the purpose and properties/parameters of a device service for human readers.

— **<metadata> : otx:MetaData [0..1]** (derived from **otx:NamedAndSpecified**)

In case that a diagnostic application needs to associate any further (tool-specific) information to a device service, this element shall be used.

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

This is the formal counterpart of the **<specification>**. It contains a list of parameter descriptions adhering to a device service.

— **<parameters> : DeviceServiceParameterDeclarations [0..1]**

This simple container element represents the list of arguments for a device service call. The content-type of **DeviceServiceParameterDeclarations** is **<xsd:choice> [1..*]** which allows an arbitrary-length list of in- and output parameters of a device service.

NOTE While it might be pretty seldom that more than one out parameter is described, there are cases in which the device serves as a kind of gateway or is a complex device like an ECOS measurement device which is able to return a variety of return values like for instance **numberOfUpperLimitViolations**, **MeasurementsAfterStopTrigger**, etc.

— **<inParam> : DeviceServiceInParameterDeclaration**

Describes an input parameter for a service. This is needed for measurement services which require additional arguments for parametrizing their execution.

DeviceServiceInParameterDeclaration is based on type **otx:Declaration**. Therefore, a **<inParam>** element has a name, a specification and a data type declaration (please refer to Part 2 of ISO 13209 for details about declarations).

— **<outParam> : DeviceServiceOutParameterDeclaration**

Describes an output parameter for the requested service.

DeviceServiceOutParameterDeclaration is based on type **otx:Declaration**. Therefore, a **<outParam>** element has a name, a specification and a data type declaration (please refer to Part 2 of ISO 13209 for details about declarations).

— **<throws> : Throws [0..1]**

This shall declare an arbitrary-length list of measure exception types which this device service may potentially throw.

— **<exception> : MeasureException [1..*]**

Describes an exception type which may possibly be thrown by the enclosing device service.

16.6 Actions

16.6.1 Overview

The OTX Measure extension introduces one action named **`ExecuteDeviceService`**, as described in the following.

16.6.2 Syntax

Figure 90 shows the syntax of the Measure extension's signature types.

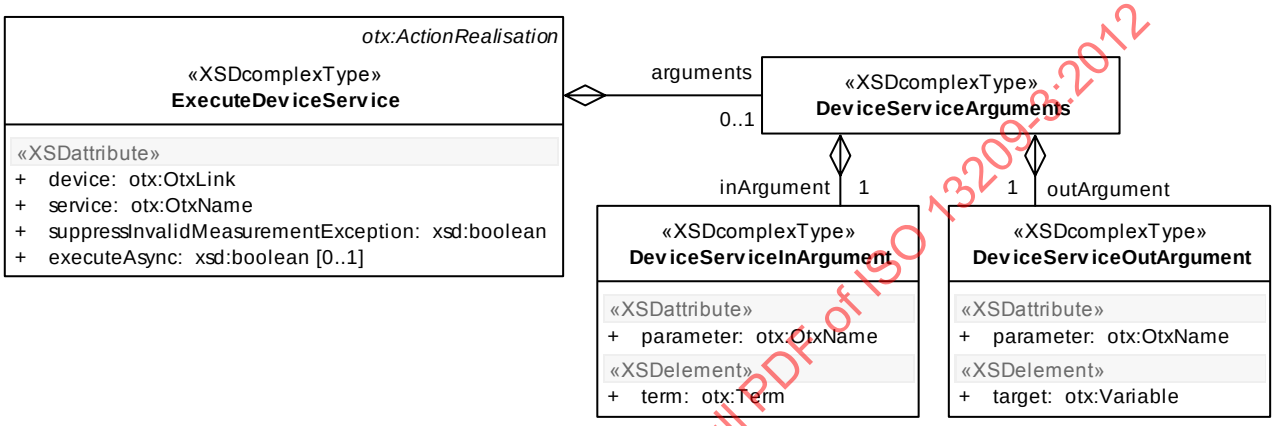

Figure 90 — Data model view: Measure actions

NOTE The XSD complex type **`DeviceServiceArguments`** is of **`<xsd:choice> [0..*]`** content-type, which is not explicitly shown in the figure above.

16.6.3 Semantics

16.6.3.1 General

The basic semantics common to all kinds of OTX actions are specified in Part 2 of ISO 13209.

16.6.3.2 ExecuteDeviceService

The **`ExecuteDeviceService`** action shall execute a service provided by a measurement device. The action connects to physical devices from where it retrieves measurements. The OTX author may choose which of the retrieved measurements shall be assigned to OTX variables. The devices to which the **`ExecuteDeviceService`** action connects shall be described by device signatures, as specified in 16.5 (**`DeviceSignature`** type).

NOTE There are devices which need to be configured prior to execution of a specific service. Configuration should be done by previously executing the respective configuration services (triggered also by **`ExecuteDeviceService`** actions). This allows e.g. setting parameters of the test equipment or controlling the object under test (e.g. setting the speed on a roller test bench).

The members of the **`ExecuteDeviceService`** type have the following semantics:

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

This option tells the communication backend to make this device service execution non-blocking. This means that if **`executeAsync`** is set to **`true`**, the OTX execution flow will immediately move on to the next **`Action`**, without waiting for the result of the **`ExecuteDeviceService`** action. An OTX sequence can make use of the **`DeviceEventSource`** term (refer to 16.7.3.3.1) to be notified when a new result

from an asynchronously executed device service has arrived. When this happens, the OTX variable(s) which are linked to a service's output parameter(s) will potentially contain a new value.

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

This flag shall affect only those device services which declare **InvalidMeasurementException** in their **<throws>** block. For other device services, it shall have no effect.

When the flag is set to **false** (the default), any **InvalidMeasurementException** produced by the executed device service will be passed on to the OTX sequence by the **ExecuteDeviceService** action. This supports the most common use case when it is senseless to continue a test sequence if no valid measurement could be produced. This ensures that later uses of the term **isValidMeasurement** will always return **true** – therefore the OTX author does not have to check each measurement if it is valid, and he can treat exceptional cases of invalid measurements by using ordinary OTX exception handling mechanisms.

Otherwise, if the flag is set to **true**, the action shall hand over invalid-state **Measurement** values to the OTX sequence and suppress any throw of an **InvalidMeasurementException**. An example for the use of invalid measurements: an embedded system that measures a measurement profile at a fixed rate or in a loop might produce a few invalid measurements as well (because the measuring situation for these measurements was bad) but the measurement process should not be interrupted by an exception. In this case, it may be important to know nevertheless which measurements were invalid and which were not.

An invalid measurement does return a quantity, but the value can be arbitrary. The unit (if any) is defined by the measuring device and its configuration at measuring time.

— **device : otx:OtxLink [1]**

This attribute identifies the measurement device to execute the service on. The link shall point to the corresponding **DeviceSignature** for that device.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

— Measure_Chk001 – correct target for ExecuteDeviceService and DeviceEventSource

— **service : otx:OtxName [1]**

Identifies the service which shall be executed. The service name shall be defined within the corresponding service declaration within the **DeviceSignature**.

Associated checker rules:

— Measure_Chk002 – executed device service is declarated in device signature

— **<arguments> : DeviceServiceArguments [0..1]**

The content-type of this simple container element is **<xsd:choice> [1..*]** which allows an arbitrary-length list of in- and output arguments for the to-be-executed device service's parameters.

— **<inArgument> : DeviceServiceInArgument**

Represents an input argument for an input parameter of the to-be-executed device service. An input argument may be omitted **if and only if** there's 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:

— **parameter : otx:OtxName [1]**

This attribute represents the target parameter to which the argument shall be assigned.

— **<term> : otx:Term [1]**

Represents the value to be used as input argument for the service parameter. The value data type shall match to the parameter data type as declared in the corresponding device's signature.

— **<outArgument> : DeviceServiceOutArgument**

Represents an ouput argument for an output parameter of the to-be-executed device service. Output arguments may be omitted **freely** (e.g. in the case when there is no interest in one of the returned data). The parameter is identified by name, the argument is a variable:

— **parameter : otx:OtxName [1]**

This attribute represents the output parameter whose value shall be assigned to the target OTX variable.

— **<target> : otx:Variable [1]**

Represents the OTX variable to hold the value of the output parameter of the device service. The variable's data type shall match to the parameter data type as declared in the corresponding device's signature.

Associated checker rules:

— Measure_Chk003 – correct ExecuteDeviceService arguments

— Measure_Chk004 – ExecuteDeviceService input argument omission

— Measure_Chk005 – No Path in ExecuteDeviceService ouput arguments

Throws:

The exceptions that this action may throw depend on the **<throws>** declarations defined for the executed device service in the corresponding device signature (this is similar to **otx:ProcedureCall** which throws exceptions according to the called procedure).

16.7 Terms

16.7.1 Overview

The OTX Measure extension introduces two categories of terms, the first of which describes terms that allow measurement value handling while the other supports the handling of events fired from measurement devices. Figure 91 provides an overview of the OTX Measure term categories.

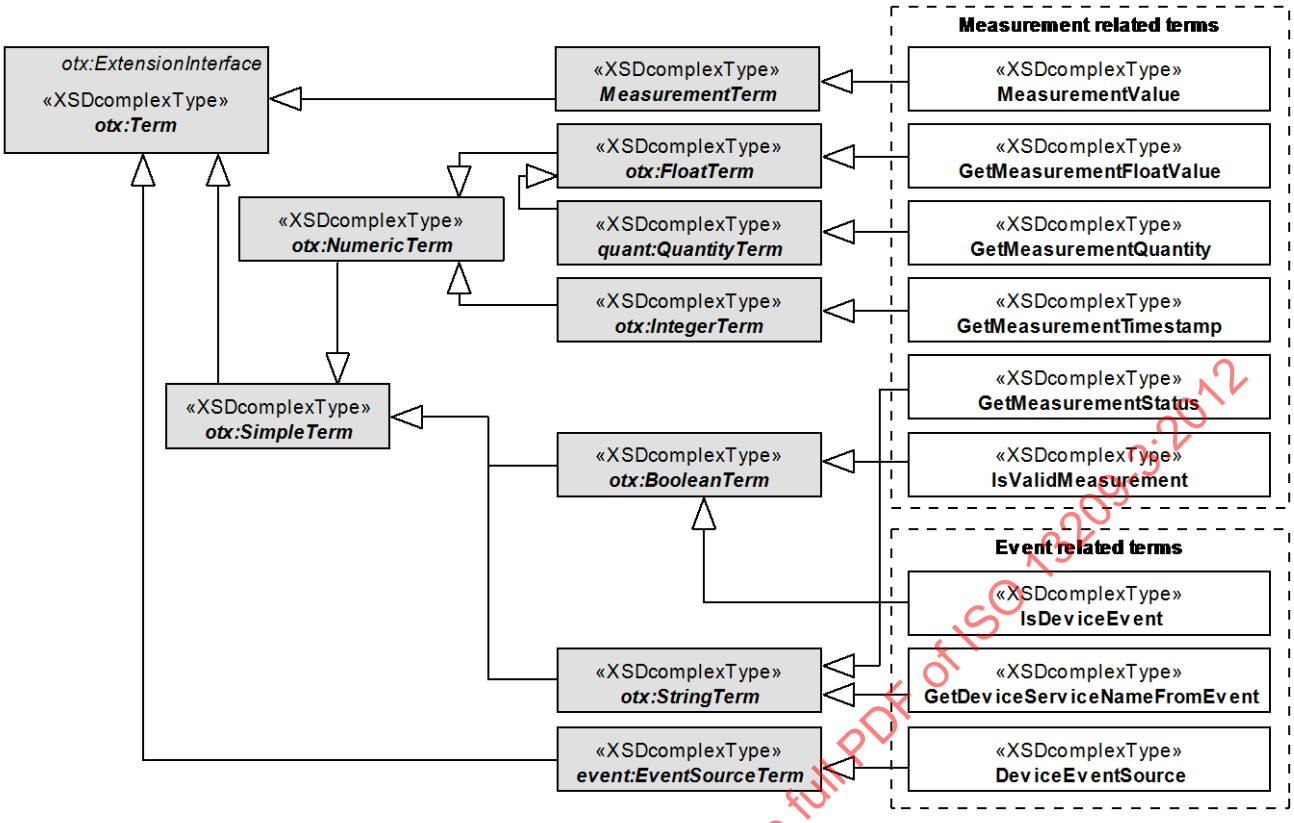

Figure 91 — Data model view: Measurement term categories

16.7.2 Measurement related terms

16.7.2.1 Description

The primary purpose of the measurement related terms is to get information out of **`Measurement`** objects which have been retrieved from a measurement device by executing an **`ExecuteDeviceService`** action.

16.7.2.2 Syntax

Figure 92 shows the syntax of the measurement related terms of the Measure extension.

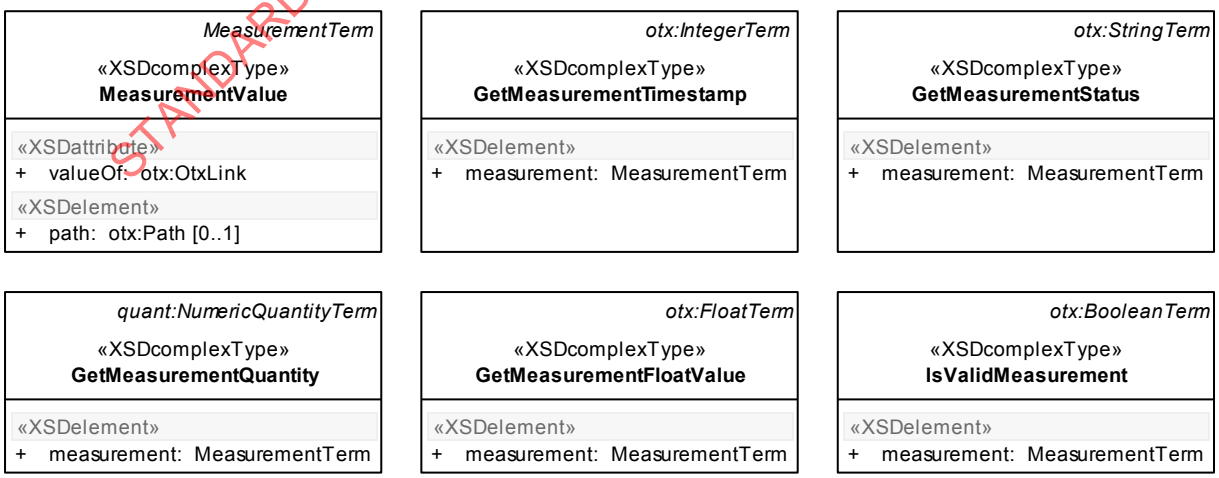

Figure 92 — Data model view: Measurement related terms

16.7.2.3 Semantics

16.7.2.3.1 MeasurementTerm

The abstract type **`MeasurementTerm`** is an **`otx:Term`**. It serves as a base for all concrete terms which return a **`Measurement`**. It has no special members.

16.7.2.3.2 MeasurementValue

This term returns the **`Measurement`** stored in a **`Measurement`** variable. For more information on value-terms and the syntax and semantics of the **`valueOf`** attribute and **`<path>`** element, please refer to Part 2 of ISO 13209.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

Throws:

— **`otx:OutOfBoundsException`**
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).

— **`otx:InvalidReferenceException`**
If the variable value is not valid (no value was assigned to the variable before).

16.7.2.3.3 GetMeasurementTimestamp

Get the timestamp of a measurement, expressed in milliseconds elapsed since 1970-01-01 00:00:00 UTC (cf. **`time:GetTimestamp`** as specified in the OTX DateTime extension in Clause 6). If no timestamp exists, the measurement is not valid and an exception shall be thrown.

`GetMeasurementTimestamp` is an **`otx:IntegerTerm`**. Its members have the following semantics:

— **`<measurement> : MeasurementTerm [1]`**

Represents the measurement whose timestamp shall be acquired.

Throws:

— **`measure:InvalidMeasurementException`**
If the measurement contains no timestamp (invalid measurement).

16.7.2.3.4 GetMeasurementStatus

Get the status of a measurement. The status of the measurement does reflect the situation of its generation. The returned status shall be a string. This standard does not specify a set of allowed values for the returned status strings – however, a listing of commonly used status strings is recommended below.

Recommendation for common status values:

— Status **`"ok"`**: This state is used for ordinarily measured values (the normal case).

— Status **`"generated"`**: This state is commonly used for measurements whose value was generated during an invalid state of the system under test. This state applies only to measurement devices which return a (fake) value despite the invalid machine state (for other devices, the **`ExecuteDeviceService`** action would have thrown a **`ServicePreconditionException`**). This situation may occur e.g. when the rpm of an engine shall be measured but the engine is not running and does not provide the rpm signal; therefore the measurement device assumes at this point that the rpm is 0, thus it fakes a 0

measurement. Other than that this state may also be used for measurements whose value was not measured but generated, e.g. when a measurement device is running in a simulation mode.

— Status **"interpolated"**: Commonly used for measurements whose value was not directly measured but calculated.

— Status **"invalid"**: Commonly used for measurements whose value could neither be measured correctly nor interpolated or faked etc.

— Status **"normalized"**: Commonly used for measurements whose value has been normalized, e.g. if a filter in the measurement device or driver has cut off outliers in a frequency measurement.

— Status **"outdated"**: Commonly used for measurements whose value is outdated. This is the case if the value is present but not current enough for the used service.

GetMeasurementStatus is an **otx:StringTerm**. Its members have the following semantics:

— **<measurement> : MeasurementTerm [1]**

Represents the measurement whose status shall be acquired.

16.7.2.3.5 GetMeasurementQuantity

Get the measured quantity value from a measurement (see Clause 17 about quantities).

GetMeasurementQuantity is a **quant:QuantityTerm**. Its members have the following semantics:

— **<measurement> : MeasurementTerm [1]**

The measurement whose quantity value shall be acquired.

Throws:

— **measure:InvalidMeasurementException**
If the measurement is invalid.

16.7.2.3.6 GetMeasurementFloatValue

Get the raw float value of a measurement as it has been received from the measurement device, disregarding any physical unit information.

GetMeasurementValue is an **otx:FloatTerm**. Its members have the following semantics:

— **<measurement> : Measurement [1]**

The measurement whose raw float value shall be acquired.

Throws:

— **measure:InvalidMeasurementException**
If the measurement is invalid.

16.7.2.3.7 IsValidMeasurement

Evaluates the status of a measurement. As the status constants are not fixed by this standard (see listing of recommended states given for **GetMeasurementStatus** term above) this action can be used to determine whether the measurement can be used or not.

A measurement shall be considered valid if it contains at least a quantity and a timestamp. Furthermore the status of the measurement should correspond to the above: if the status is **"invalid"**, **"outdated"** or has a value with equivalent semantics, this action should return **false**. If the status is **"generated"** or equivalent, it is application specific whether the measurement is to be considered valid or not.

IsValidMeasurement is an **otx:BooleanTerm**. Its members have the following semantics:

— **<measurement> : MeasurementTerm [1]**

The measurement whose status shall be evaluated.

16.7.3 Event related terms

16.7.3.1 Description

The terms introduced in the following support the event handling mechanisms as described for the OTX EventHandling extension in Clause 8. The terms can be used for creating event sources listening for events fired by measurement device (**DeviceEventSource**), for querying the type of event (**IsDeviceEvent**) and for identifying the particular device and service which fired an event (**GetDeviceServiceFromEvent**).

16.7.3.2 Syntax

Figure 93 shows the syntax of the event related terms of the Measure extension.

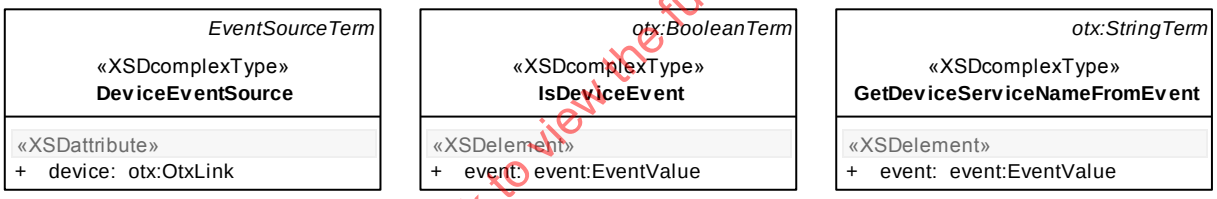

Figure 93 — Data model view: Event related terms

16.7.3.3 Semantics

16.7.3.3.1 DeviceEventSource

The **DeviceEventSource** term accepts a link to a **DeviceSignature** of a device that is to be made an event source. This term enables an OTX sequence to use a measurement device as a source for events in the context of the OTX EventHandling extension (please refer to Clause 8). A measurement device (driver) shall trigger an event every time a new output parameter from one of its services has arrived. The **DeviceEventSource** term is the complementary functionality to the asynchronous execution feature of the **ExecuteDeviceService** action: when **ExecuteDeviceService** is used with **executeAsync** attribute set to **true**, the only way to be notified of incoming measurement values for the executed device service is to use it as an event source through the **DeviceEventSource** term.

DeviceEventSource is an **event:EventSourceTerm**. Its members have the following semantics:

— **device : otx:OtxLink [1]**

Represents the to-be-monitored device. If an output parameter of an earlier triggered device service becomes available, the event shall be fired, causing an embedding **event:WaitForEventAction** to exit.

Associated checker rules:

— Measure_Chk001 – correct target for ExecuteDeviceService and DeviceEventSource

16.7.3.3.2 IsDeviceEvent

The **IsDeviceEvent** term accepts an **EventValue** term yielding an **Event** object that has been raised by the OTX runtime, as a result of declaring a measurement device as an event source by using the term **DeviceEventSource**. The term shall return **true** if and only if the **Event** originates from a **DeviceEventSource** term.

IsDeviceEvent is an **otx:BooleanTerm**. Its members have the following semantics:

— **<event> : event:EventValue [1]**

Represents the **Event** whose type shall be tested.

16.7.3.3.3 GetDeviceServiceNameFromEvent

The **GetDeviceServiceNameFromEvent** term accepts an **EventValue** term yielding an **Event** object that has been raised by the OTX runtime, as a result of declaring a measurement device as an event source by using the term **DeviceEventSource**. It shall return a string which contains the device and service name of the device and service that caused the event. By using this term, an OTX sequence can wait for an **Event** raised by a device receiving a new result and then evaluate which service of that device caused the event.

The returned string value shall be composed out of two parts: "**devicename.servicename**", where "**devicename**" and "**servicename**" are the names as given by the corresponding **DeviceSignature**.

GetDeviceServiceNameFromEvent is an **otx:StringTerm**. Its members have the following semantics:

— **<event> : event:EventValue [1]**

Represents the event that was raised after executing a device service.

Throws:

— **otx:TypeMismatchException**
If the specified event has not been raised by a **DeviceEventSource**.

17 OTX Quantities extension

17.1 Introduction

The **Quantity** data types specified in this extension offer an additional layer of abstraction on top of the numeric data types provided by the OTX Core as specified by Part 2 of ISO 13209. The **Quantity** type contains additional information about a value's physical unit, allowing it to describe actual measurement values. This allows e.g. the OTX DiagCom extension (see Clause 6) to use quantities for getting data in and out of diagnostic services.

A **Quantity**, as mentioned, contains information about a physical unit besides the actual value. To do this, OTX Quantities reuses the unit definition data model specified by the ODX standard (see **UNIT-SPEC** data type in 7.3.6.7 in ISO 22901-1:2008). The intention is to use ODX for defining a set of units that can then be referenced by elements of the OTX Quantities extension. Please note that the ODX **UNIT-SPEC** can be used separately from the rest of the ODX standard. As an example, a minimal **UNIT-SPEC** definition is provided below (see following page).

The way an ODX **UNIT-SPEC** is defined allows an OTX runtime system to automatically convert **Quantity** values into different units, as long as these are defined as equivalent units in ODX. Thus, an OTX runtime is able to automatically perform basic arithmetic operations on **Quantity** operands, so for example an addition operation on a **Quantity** containing a 'km/h' value with another **Quantity** containing a value in 'm/h'. To achieve this, an OTX runtime is expected to perform any arithmetic involving quantities using an internal presentation of the quantities' values that is normalized to the SI base unit(s) underlying the unit of the **Quantity**. For example, to add a **Quantity** with a unit of "miles per hour" to another **Quantity** with a unit of "kilometres per hour", the OTX runtime should convert both quantities' values to the underyling base SI dimensions (in this case "meters per second") before adding both quantities' values. In subsequent sections, the user-assigned unit of a **Quantity** is referred to as a *display unit*, while the corresponding SI-dimensioned unit is called *base unit*. Accordingly, the quantities' value in display units is called physical or display value, while the value in base SI dimensions is referred to as internal or normalized value.

EXAMPLE An XML instance of the ODX **UNIT-SPEC**.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<ODX MODEL-VERSION="2.2.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <DIAG-LAYER-CONTAINER ID="UNIT-SPEC-DLC">
    <SHORT-NAME>DLC_UnitSpec</SHORT-NAME>
    <LONG-NAME>DLCUnitSpec</LONG-NAME>
    <ECU-SHARED-DATAS>
      <ECU-SHARED-DATA ID="UNIT-SPEC-ESD">
        <SHORT-NAME>ESD_UnitSpec</SHORT-NAME>
        <DIAG-DATA-DICTIONARY-SPEC>

          <UNIT-SPEC>

            <UNIT-GROUPS>
              <UNIT-GROUP OID="EU_Metric">
                <SHORT-NAME>EU_Metric</SHORT-NAME>
                <CATEGORY>COUNTRY</CATEGORY>
                <UNIT-REFS>
                  <UNIT-REF ID-REF="km"/>
                  <UNIT-REF ID-REF="m"/>
                  <UNIT-REF ID-REF="mm"/>
                </UNIT-REFS>
              </UNIT-GROUP>

              <UNIT-GROUP OID="UK_Imperial">
                <SHORT-NAME>UK_Imperial</SHORT-NAME>
                <CATEGORY>COUNTRY</CATEGORY>
                <UNIT-REFS>
                  <UNIT-REF ID-REF="mi"/>
                  <UNIT-REF ID-REF="ft"/>
                  <UNIT-REF ID-REF="in"/>
                </UNIT-REFS>
              </UNIT-GROUP>
```

```
  <UNIT-GROUP OID="TravelDistance">
    <SHORT-NAME>TravelDistance</SHORT-NAME>
    <CATEGORY>EQUIV-UNITS</CATEGORY>
    <UNIT-REFS>
      <UNIT-REF ID-REF="mi"/>
      <UNIT-REF ID-REF="km"/>
    </UNIT-REFS>
  </UNIT-GROUP>
</UNIT-GROUPS>

<UNITS>
  <UNIT ID="km">
    <SHORT-NAME>km</SHORT-NAME>
    <LONG-NAME>kilometers</LONG-NAME>
    <DISPLAY-NAME>km</DISPLAY-NAME>
    <FACTOR-SI-TO-UNIT>1000</FACTOR-SI-TO-UNIT>
    <PHYSICAL-DIMENSION-REF ID-REF="PD-m"/>
  </UNIT>

  <UNIT ID="s">
    <SHORT-NAME>s</SHORT-NAME>
    <LONG-NAME>seconds</LONG-NAME>
    <DISPLAY-NAME>s</DISPLAY-NAME>
    <PHYSICAL-DIMENSION-REF ID-REF="PD-s"/>
  </UNIT>

  <UNIT ID="km_h">
    <SHORT-NAME>km_h</SHORT-NAME>
    <LONG-NAME>kilometers per hour</LONG-NAME>
    <DISPLAY-NAME>km/h</DISPLAY-NAME>
    <FACTOR-SI-TO-UNIT>3.6</FACTOR-SI-TO-UNIT>
    <PHYSICAL-DIMENSION-REF ID-REF="PD-m_s"/>
  </UNIT>


  <UNIT ID="min">
    <SHORT-NAME>min</SHORT-NAME>
    <LONG-NAME>minutes</LONG-NAME>
    <DISPLAY-NAME>min</DISPLAY-NAME>
    <FACTOR-SI-TO-UNIT>60</FACTOR-SI-TO-UNIT>
    <PHYSICAL-DIMENSION-REF ID-REF="PD-s"/>
  </UNIT>

  <UNIT ID="m">
    <SHORT-NAME>m</SHORT-NAME>
    <LONG-NAME>meters</LONG-NAME>
    <DISPLAY-NAME>m</DISPLAY-NAME>
    <PHYSICAL-DIMENSION-REF ID-REF="PD-m"/>
  </UNIT>

  <UNIT ID="mm">
    <SHORT-NAME>mm</SHORT-NAME>
    <LONG-NAME>millimeters</LONG-NAME>
    <DISPLAY-NAME>mm</DISPLAY-NAME>
    <FACTOR-SI-TO-UNIT>0.001</FACTOR-SI-TO-UNIT>
    <PHYSICAL-DIMENSION-REF ID-REF="PD-m"/>
  </UNIT>

  <UNIT ID="mi">
    <SHORT-NAME>mi</SHORT-NAME>
    <LONG-NAME>mile</LONG-NAME>
    <DISPLAY-NAME>mi</DISPLAY-NAME>
    <FACTOR-SI-TO-UNIT>6.213712E-4</FACTOR-SI-TO-UNIT>
    <OFFSET-SI-TO-UNIT>0.0</OFFSET-SI-TO-UNIT>
    <PHYSICAL-DIMENSION-REF ID-REF="PD-m"/>
  </UNIT>

  <UNIT ID="ft">
    <SHORT-NAME>ft</SHORT-NAME>
    <LONG-NAME>foot</LONG-NAME>
    <DISPLAY-NAME>ft</DISPLAY-NAME>
    <FACTOR-SI-TO-UNIT>3.28084</FACTOR-SI-TO-UNIT>
    <OFFSET-SI-TO-UNIT>0.0</OFFSET-SI-TO-UNIT>
    <PHYSICAL-DIMENSION-REF ID-REF="PD-m"/>
  </UNIT>

  <UNIT ID="in">
    <SHORT-NAME>in</SHORT-NAME>
    <LONG-NAME>inch</LONG-NAME>
    <DISPLAY-NAME>in</DISPLAY-NAME>
    <FACTOR-SI-TO-UNIT>39.37008</FACTOR-SI-TO-UNIT>
    <OFFSET-SI-TO-UNIT>0.0</OFFSET-SI-TO-UNIT>
    <PHYSICAL-DIMENSION-REF ID-REF="PD-m"/>
  </UNIT>
</UNITS>
```

```
          <PHYSICAL-DIMENSIONS>
            <PHYSICAL-DIMENSION ID="PD-m">
              <SHORT-NAME>km</SHORT-NAME>
              <LENGTH-EXP>1</LENGTH-EXP>
            </PHYSICAL-DIMENSION>

            <PHYSICAL-DIMENSION ID="PD-s">
              <SHORT-NAME>s</SHORT-NAME>
              <TIME-EXP>1</TIME-EXP>
            </PHYSICAL-DIMENSION>

            <PHYSICAL-DIMENSION ID="PD-m_s">
              <SHORT-NAME>m_s</SHORT-NAME>
              <LENGTH-EXP>1</LENGTH-EXP>
              <TIME-EXP>-1</TIME-EXP>
            </PHYSICAL-DIMENSION>

            <PHYSICAL-DIMENSION ID="PD-m_ss">
              <SHORT-NAME>km_h</SHORT-NAME>
              <LENGTH-EXP>1</LENGTH-EXP>
              <TIME-EXP>-2</TIME-EXP>
            </PHYSICAL-DIMENSION>
          </PHYSICAL-DIMENSIONS>

        </UNIT-SPEC>
      </DIAG-DATA-DICTIONARY-SPEC>
    </ECU-SHARED-DATA>
  </ECU-SHARED-DATAS>
</DIAG-LAYER-CONTAINER>
</ODX>
```

17.2 Data types

17.2.1 Overview

The OTX Quantities extension introduces the data types **`Quantity`** and **`Unit`**, as described in the following.

17.2.2 Syntax

The syntax of the datatype declarations of the OTX Quantities extension is shown in Figure 94.

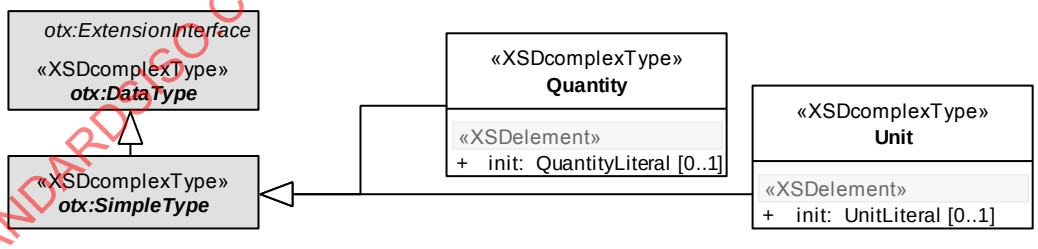

Figure 94 — Data model view: Quantities data types

17.2.3 Semantics

17.2.3.1 General

The following describes the runtime semantics of the OTX Quantities data types.

17.2.3.2 Quantity

A **`Quantity`** represents a numeral value which has a display unit associated with it. For instance, the value **`"5"`** is described more specifically by a **`Quantity`** that also contains information about the unit of the value,

e.g. **"5 km/h"**. Furthermore, a **Quantity** has an optional display precision property which has an effect on the **otx:ToString** conversion of a **Quantity** (17.5.3.3.1).

A **Quantity** can be initialized at declaration time.

The members of **Quantity** have the following semantics:

— **<init> : QuantityLiteral [0..1]**

This optional element represents the hard-coded value from which the declared **Quantity** shall be created. The literal includes a float-value, a display unit name and a display precision; when the **Quantity** is created, the float value shall be interpreted according to the display unit:

— **<numeral> : otx:FloatLiteral [1]**

Represents the hard-coded float-value from which the **Quantity** shall be created.

— **<displayUnit> : UnitLiteral [1]**

Represents the hard-coded display unit of the **QuantityLiteral**.

— **<displayPrecision> : otx:IntegerLiteral [0..1]**

Optionally represents the hard-coded display precision of the **Quantity** declaration (17.5.3.3.1).

See 17.5.2.3.4 for further details on term **QuantityLiteral**.

IMPORTANT — If a Quantity declaration is not explicitly initialized (omitted <init> element), the default value shall be a Quantity with a base value of 0.0 and a dimensionless unit.

17.2.3.3 Unit

A **Unit** represents a physical unit which is defined in a **UNIT-SPEC** (cf. 17.1). A **Unit** can be associated to a physical value when creating a **Quantity**, but it can also be used stand-alone, e.g. when comparing the display **Unit** of a **Quantity** to another **Unit** object.

A **Unit** can be initialized at declaration time.

The members of **Unit** have the following semantics:

— **<init> : UnitLiteral [0..1]**

This optional element describes the initialization value from which the **Unit** shall be created:

— **<value> : UnitDefinition [1]**

This element represents the hard-coded link to the appropriate **UNIT** definition in a **UNIT-SPEC** which shall be associated to the declared **Unit**. For linking, the element allows all attributes from the namespace **"http://www.w3.org/1999/xlink"**, as defined by the W3C XLink recommendation [W3C XLink]. For the usage of the attributes, the rules given in 17.5.2.3.1 shall apply.

See 17.5.2.3.8 for further details on term **UnitLiteral**.

IMPORTANT — If a Unit declaration is not explicitly initialized (omitted <init> element), the default value shall be a dimensionless unit.

17.3 Exceptions

17.3.1 Overview

All elements referenced in this clause are derived from the OTX Core **`Exception`** type as defined by Part 2 of ISO 13209. They represent the full set of exceptions added by the OTX Quantities extension.

17.3.2 Syntax

The syntax of all OTX Quantities exception type declarations is shown in Figure 95.

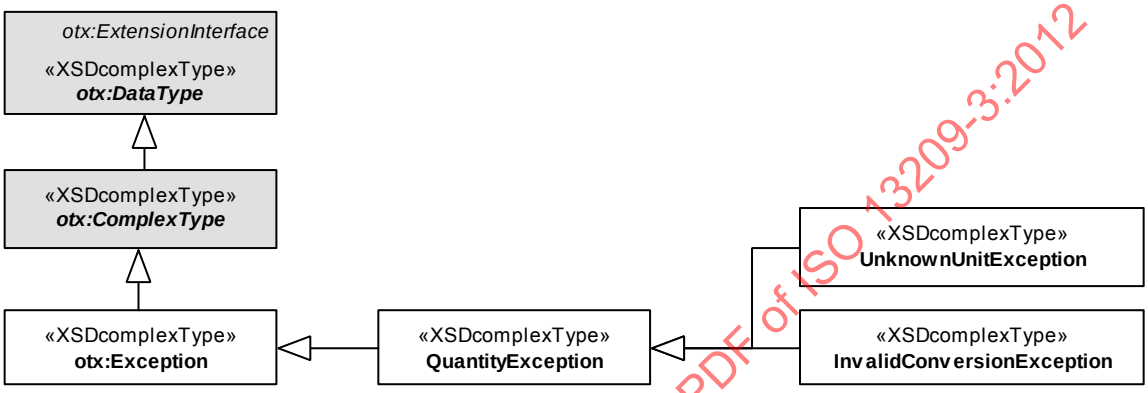

Figure 95 — Data model view: Quantities exceptions

17.3.3 Semantics

17.3.3.1 General

Since all OTX Quantities exception types are implicit exceptions whithout initialisation parts, they can not be declared constant.

17.3.3.2 QuantityException

The **`QuantityException`** type is the base type for all exceptions in the OTX Quantities extension. A **`QuantityException`** shall be used in case the more specific exception types described in the remainder of this section do not apply to the problem at hand.

17.3.3.3 UnknownUnitException

An **`UnknownUnitException`** shall be thrown if a referenced unit is not known by the runtime system. This exception can for instance occur when using the **`UnitLiteral`** term and passing a unit reference that does not exist in the system's **`UNIT-SPEC`**.

17.3.3.4 InvalidConversionException

An **`InvalidConversionException`** shall be thrown if the physical dimensions of **`Quantity`** operands in arithmetic operations are incompatible, e.g. if a speed is added to a voltage.

17.4 Variable access

17.4.1 Overview

As specified in Part 2 of ISO 13209, OTX extensions shall define a variable access type for each datatype they define (exception types inclusively). All variable access types are derived from the OTX Core **`Variable`** extension interface. The following specifies all variable access types defined for the OTX Quantities extension.

17.4.2 Syntax

Figure 96 shows the syntax of the Quantities extension's variable access types.

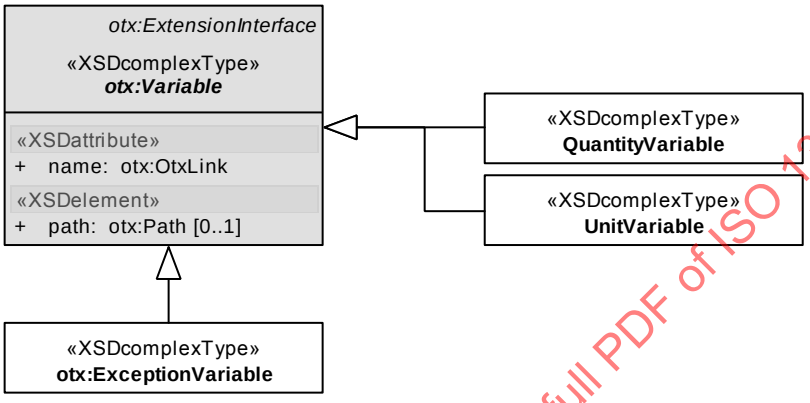

Figure 96 — Data model view: Quantities variable access types

17.4.3 Semantics

The general semantics for all variable access types shall apply. Please refer to Part 2 of ISO 13209 for further details.

17.5 Terms

17.5.1 Overview

All of the OTX Quantities terms shown in Figure 97 extend the **`Term`** extension interface as defined by Part 2 of ISO 13209. Information about the specific super class of a term is provided in the individual term description clauses below.

As shown in Figure 97, there are three OTX Quantity term categories:

— The first category contains terms yielding **`Quantity`** values; these are all based on the abstract term **`QuantityTerm`**.

— The second category contains terms which allow accessing various properties of a **`Quantity`**, such as the display value, base unit and display unit.

— The third category contains basic terms for **`Unit`** handling.

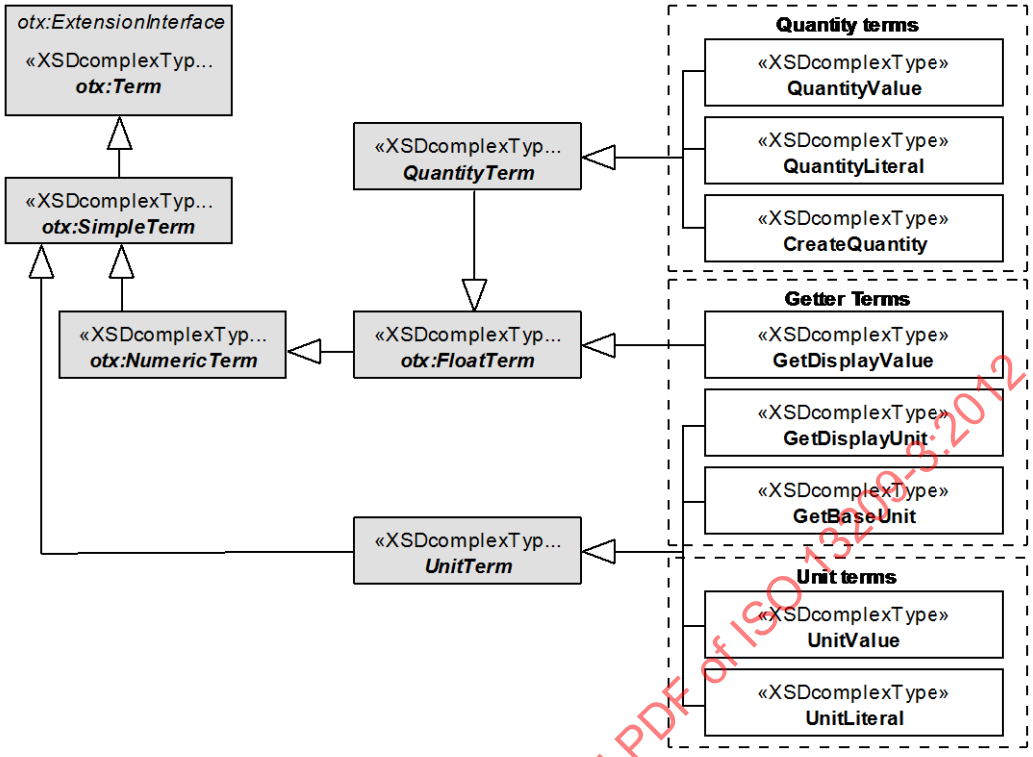

Figure 97 — Data model view: Quantities term categories

17.5.2 Quantity and Unit related terms

17.5.2.1 Description

The following specifies the terms for creating and accessing `Quantity` and `Unit` values.

17.5.2.2 Syntax

Figure 98 shows the syntax of all `Quantity` related terms of the Quantities extension.

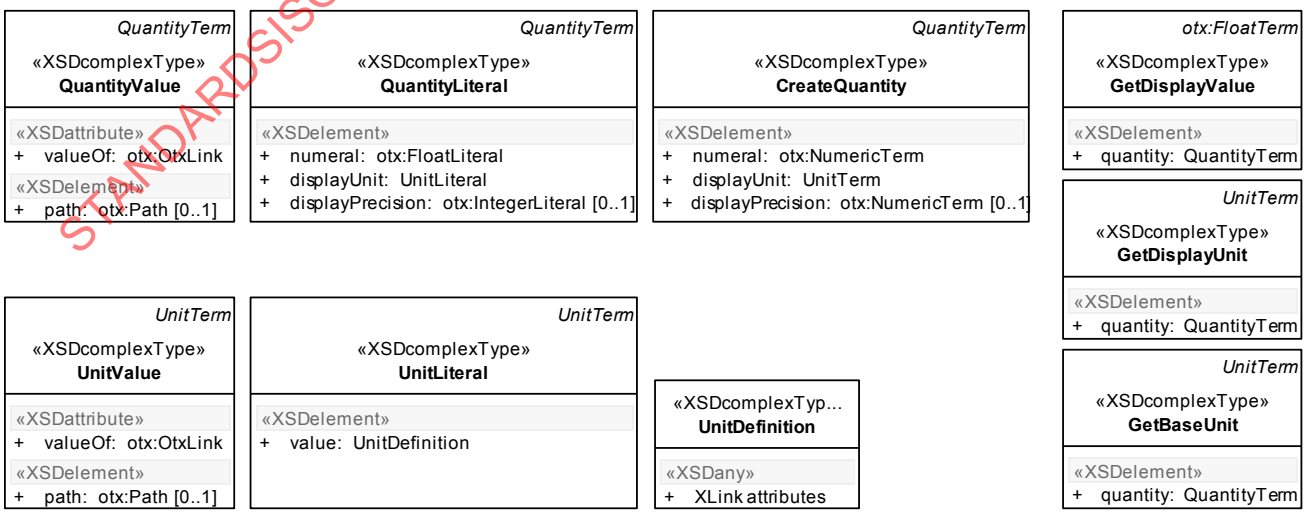

Figure 98 — Data model view: `Quantity` related terms

17.5.2.3 Semantics

17.5.2.3.1 Referring to unit definitions

Several terms in the OTX Quantities extension use the **Unit** type in order to refer to unit or unit group definitions located in an external resource. The extension reuses the unit definition data model specified by the ODX standard (see **UNIT-SPEC** data type in 7.3.6.7 in ISO 22901-1:2008). Concerning references from OTX to **UNIT-SPEC** entries, the rules below shall apply.

IMPORTANT — Any elements of the OTX Quantities terms that work with units shall link to required ODX unit definitions by using *simple XLinks* only, as specified by [W3C XLink]. This means that the `xlink:type` attribute shall always be set to `"simple"`. Furthermore, the `xlink:href` attribute value should follow the pattern of `"{URI}#{SHORT-NAME}"`, where `{URI}` represents the `UNIT-SPEC` resource and `{SHORT-NAME}` identifies the unit definition by its ODX `SHORT-NAME` property. The pattern corresponds to a *shorthand notation XPointer*, as specified by Clause 3.2 in [W3C XPtr]. However, in case the shorthand notation is not sufficient to address unit definitions, the full XPointer notation may be used (e.g. when one ODX-document contains more than one `UNIT-SPEC` container).

EXAMPLE For linking to the unit definition for **"mm"** given in the exemplary **UNIT-SPEC** in 17.1, the element has the form of **<unit *xlink:type*="simple" *xlink:href*="unit-spec.xml#mm"/>**.

17.5.2.3.2 QuantityTerm

The abstract type **QuantityTerm** is an **otx:FloatTerm**. It serves as a base for all concrete terms which return a **Quantity**. It has no special members.

17.5.2.3.3 QuantityValue

This term returns the **Quantity** stored in a **Quantity** variable. For more information on value-terms and the syntax and semantics of the **valueOf** attribute and **<path>** element, please refer to Part 2 of ISO 13209.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

Throws:

— **otx:OutOfBoundsException**
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).

17.5.2.3.4 QuantityLiteral

This term shall be used to create a **Quantity** object based on a hard-coded float value and display unit. The provided float value shall be interpreted as a display value (i.e. the value of the **Quantity** in given display units). Furthermore, the term optionally allows specifying a precision property which has an effect on the **otx:ToString** conversion of the resulting **Quantity** (17.5.3.3.1).

QuantityLiteral is a **QuantityTerm.** Its members have the following semantics:

— **<numeral> : otx:FloatLiteral [1]**

Represents the hard-coded value from which the **Quantity** shall be created. The value shall be interpreted in display units.

— **<displayUnit> : UnitLiteral [1]**

Represents the display unit of the **Quantity**. See 17.5.2.3.8 for further details on term **UnitLiteral**.

— **<displayPrecision> : otx:IntegerLiteral [0..1]**

Represents the hard-coded display precision of the **QuantityLiteral** (17.5.3.3.1).

17.5.2.3.5 CreateQuantity

The **CreateQuantity** term is the constructor for a **Quantity**. The provided numeric value shall be interpreted as a display value (i.e. the value of the **Quantity** in given display units). Furthermore, the term optionally allows specifying a precision property which has an effect on the **otx:ToString** conversion of the resulting **Quantity** (17.5.3.3.1).

The exact behaviour of **CreateQuantity** depends on the type of the passed numeric value:

— **Integer** or **Float** type: The value shall be interpreted according to the given display unit. Furthermore, the resulting internal value shall be **Float**, even for an **Integer** type argument. If a display precision is given, the property shall be set in the created **Quantity**; otherwise it shall remain unset (17.5.3.3.1).

— **Quantity** type: This is the copy-constructor-case which shall only work if the physical dimensions of both original and new **Quantity** match. Otherwise, an **InvalidConversionException** shall be thrown. If the physical dimensions match, the internal value of the original **Quantity** shall be copied into the new **Quantity**. Neither the original display unit nor the display precision shall be copied – instead, the new display unit and display precision specified in the term shall apply. If no display precision is given, the property shall remain unset (17.5.3.3.1).

CreateQuantity is a **QuantityTerm.** Its members have the following semantics:

— **<numeral> : NumericTerm [1]**

Represents the numeric value from which the **NumericQuantity** shall be created (in display units). The value can be either an **Integer**, a **Float** or another **Quantity**.

— **<displayUnit> : UnitTerm [1]**

Represents the display unit of the to-be-created **Quantity**. See 17.5.2.3.6 for details on **UnitTerm**.

— **<displayPrecision> : otx:NumericTerm [0..1]**

Optionally represents the display precision of the to-be-created **Quantity** (17.5.3.3.1). Float values shall be truncated.

Throws:

— **InvalidConversionException**
If **<numeral>** is a **Quantity** and its physical dimension does not match the physical dimension given by **<displayUnit>**.

17.5.2.3.6 UnitTerm

The abstract type **UnitTerm** is an **otx:SimpleTerm**. It serves as a base for all concrete terms which return a **Unit**. It has no special members.

17.5.2.3.7 UnitValue

This term returns the **Unit** stored in a **Unit** variable. For more information on value-terms and the syntax and semantics of the **valueOf** attribute and **<path>** element, please refer to Part 2 of ISO 13209.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

Throws:

— **otx:OutOfBoundsException**
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).

17.5.2.3.8 UnitLiteral

This term shall be used to create a **Unit** object based on a unit definition contained in **UNIT-SPEC** document. **UnitLiteral** allows referencing the unit definition by using XLink methodology [W3C XLink].

UnitLiteral is a **UnitTerm.** Its members have the following semantics:

— **<value> : UnitDefinition [1]**

This element represents the link to the **UNIT** definition in a **UNIT-SPEC** which shall be associated to the **Unit**. The element allows all attributes from the namespace **"http://www.w3.org/1999/xlink"** for linking, as defined by the W3C XLink recommendation [W3C XLink]. For the usage of the attributes, the rules given in 17.5.2.3.1 shall apply.

Throws:

— **UnknownUnitException**
If the given unit is not defined in the runtime system's unit specification.

Associated checker rules:

— Quantities_Chk001 – correct unit linking

— Quantities_Chk002 – no dangling unit definition links

17.5.2.3.9 GetDisplayValue

The **GetDisplayValue** term shall return the (dimensionless) **Float** value of a **Quantity** according to the **Quantity**'s display unit. Compare the **otx:ToFloat** term which, when applied to a **Quantity**, will result in the **Quantity**'s value in normalized SI unit representation.

GetDisplayValue is an **otx:FloatTerm.** Its members have the following semantics:

— **<quantity> : QuantityTerm [1]**

Represents the **Quantity** from which the numeral value shall be extracted.

17.5.2.3.10 GetDisplayUnit

The **GetDisplayUnitName** term shall extract the display unit out of a **Quantity** value (e.g. **"mp/h"**, **"km/h"**, **"h"**, **"sec"** etc.).

GetDisplayUnit is a **UnitTerm.** Its members have the following semantics:

— **<quantity> : QuantityTerm [1]**

Represents the **Quantity** from which the display unit shall be extracted.

Throws:

— **UnknownUnitException**
If the unit associated with the quantity is not defined in the system's unit specification.

17.5.2.3.11 GetBaseUnit

The **GetBaseUnit** term shall return the base unit of a **Quantity** value, according to its physical dimension (e.g. **"m"**, **"m/s"**, **"s"** etc.).

GetBaseUnit is a **UnitTerm.** Its members have the following semantics:

— **<quantity> : QuantityTerm [1]**

Represents the **Quantity** from which the base unit shall be extracted.

Throws:

— **UnknownUnitException**
If the base unit can not be obtained from the system's unit specification.

17.5.3 Overloading semantics

17.5.3.1 Description

Since **QuantityTerm** is based on OTX Core **FloatTerm**, **Quantity** values may be used in all places where **FloatTerm** or **NumericTerm** arguments are allowed. This is e.g. the OTX Core arithmetic terms, comparison terms and conversion terms, for which special rules shall apply when the operands are **Quantity** values. There are also places where general rules apply, e.g. where a display value can be used, discarding unit-information.

WARNING — Special care has to be taken by OTX authors when arithmetic operations are applied on Quantity values with display units involving an offset to the corresponding SI base unit. For instance, consider the operation 50°C - 10°C which yields 40°K (which is -233.15°C). Physically this is correct because the OTX runtime treats the operand 10°C as an absolute temperature quantity, not as a temperature difference. However, OTX authors unaware of the influence of unit offsets might expect a different result (40°C). To facilitate the handling of unit offsets, it is strongly recommended to use separate units for absolute values and difference values, where differerence values do not have an offset to the SI base unit. In the example above, the first operand should use an absolute temperature unit T[°C], while the second operand should use a difference temperature unit ΔT[°C]. With this, the operation 50°C - 10°C yields 313.15°K (40°C), which is the expected result.

17.5.3.2 Syntax

The syntax of the OTX Core arithmetic terms, comparison terms and conversion terms is specified in Part 2 of ISO 13209.

17.5.3.3 Semantics

17.5.3.3.1 Conversions

When applied to a **Quantity**, the **otx:ToFloat** term shall return the value of the **Quantity** normalized to the SI base units correlated to its display unit. For example a **Quantity** representing a speed value of 12.4 kilometers per hour will return a float value of 3.44 (as 12.4 km/h equal 3.44 m/s).

When applied to a **Quantity**, the **otx:ToString** term shall return a **String** containing the **Quanitity**'s display value followed by a space (Unicode character U+0020) followed by the **DISPLAY-NAME** of the unit definition of its display unit. For computing the string representation of the display value, the same rules as specified for **otx:ToString(Float)** shall apply. However, if the display precision property of the **Quantity** is set, the fixed-point-part shall be rounded to the decimal place given by the display precision property. Negative precision values are also allowed (expressing decimal positions to the left of the point). For instance, a **Quantity** representing a speed value of 12.35 kilometers per hour with a display precision of 1

will be rendered as "12.4 km/h", whereas a **Quantity** of 1234.5 kilometers and a precision of -2 shall be rendered as "1200 km", etc. For very large or very small values where **otx:ToString** yields a representation in scientific notation, the same rules shall apply, so for instance a **Quantity** of $1.123*10^5$ milliseconds with a display precision of 2 shall be rendered as "1.12E5 ms". Furthermore, if the display precision is greater than the number of decimal digits representing the fractional part, the string shall be stuffed by zero, e.g. a **Quantity** of 100.1 meters with a display precision of 3 yields "100.100 m".

When applied to a **Unit**, the **otx:ToString** term shall return a **String** containing the **DISPLAY-NAME** of the corresponding unit definition. For example a **Quantity** representing a speed value of 12.4 kilometers per hour will be rendered as "12.4 km/h".

IMPORTANT — For all other OTX Core conversion terms, the behaviour when applied to Quantity or Unit values is unspecified. However, OTX applications may provide custom implementations of the conversion terms for Quantity or Unit arguments, if required. Please refer to Part 2 of ISO 13209 for further information and restrictions on conversion terms.

17.5.3.3.2 Addition/Subtraction

When **Quantity** values are added or subtracted, the physical dimensions of the display unit of all **Quantity** operands shall be identical. That means that e.g. a distance **Quantity** shall only be added to another distance **Quantity** (or a scalar). Otherwise an **InvalidConversionException** shall be thrown, e.g. when a distance is added to a time.

If scalar operands exist, they shall be interpreted as normalized values according to the physical dimension of the **Quantity** operands. This allows e.g. the addition of 2 km + 1 m + 11 which will result in a **Quantity** of 2012 m.

The display unit of the resulting **Quantity** should be set to the SI base unit corresponding to the **Quantity**'s physical dimension. Furthermore, the display precision of the resulting **Quantity** shall be the maximum of the display precisions of the operands.

17.5.3.3.3 Multiplication, Division and Modulo

When **Quantity** values are multiplied or divided, a definition of the physical dimension of the resulting **Quantity** has to exist in the **UNIT-SPEC** available to the OTX system. That means that e.g. a distance **Quantity** can only be divided by a time **Quantity** if a distance/time unit is known to the system (e.g. km/h). Otherwise an **InvalidConversionException** shall be thrown.

Scalar operands shall be interpreted "as is"; this allows e.g. the multiplication of 2 * 2 km which will result in a **Quantity** of 4000 m.

The display unit of the resulting **Quantity** should be set to the SI base unit corresponding to the physical dimension resulting from the operation. Furthermore, the display precision of the resulting **Quantity** shall be the maximum of the display precisions of the operands.

17.5.3.3.4 Absolute Value and Negation

When the absolute value or the negation is computed from a **Quantity**, the display unit of the resulting **Quantity** should be set to the SI base unit corresponding to the physical dimension of the original **Quantity**. Furthermore, the display precision of the resulting **Quantity** shall be equal to the display precision of the original **Quantity**.

17.5.3.3.5 Relational operations

When **Quantity** values are compared using relational operators, an OTX runtime shall use the quantities' normalized values for comparison. So if e.g. a **Quantity** of 8 kilometers is to be compared with a **Quantity** of 10 miles, the runtime system shall convert both values into meters before doing the comparison.

Furthermore, the physical dimensions of the display unit of the **Quantity** values being compared shall be identical – e.g. it is allowed to compare distances with each other, but it is illegal to compare a distance to a time – in that case an **InvalidConversionException** shall be thrown.

If scalar operands exist, they shall be interpreted as normalized values according to the physical dimension of the **Quantity** operands. This allows e.g. the comparison of 2 km < 11 which will result in **false** (because the comparison is equivalent to comparing 2 km < 11m).

Concerning relational operations, **Unit** operands are also to be considered here: In comparisons, each **Unit** operand shall be treated as if it were a **Quantity** with a display value of 1 (this corresponds to applying **CreateQuantity(1,unit)** on the operands). This allows comparisons like e.g. m < km which equates to 1 m < 1 km which yields **true**. Hence, comparing units belonging to different physical dimensions shall also produce an **InvalidConversionException**.

17.5.3.3.6 Other operations

Generally whenever **Quantity** values are used in OTX actions or terms for which no specific definition is provided regarding the behaviour in the case of **Quantity** arguments, an OTX runtime shall use the **Quantity**'s **otx:ToFloat** value for computation. For instance, if a **Quantity** is used as an operand to the **math:Sin** term, the **Float** value (that is, the **Quantity**'s normalized value) shall be used as input for the operation.

18 OTX StringUtil extension

18.1 Introduction

This OTX extension provides a collection of data types and terms which operate on strings.

18.2 Data types

18.2.1 Overview

The OTX StringUtil extension defines one enumeration type named **`Encoding`**.

18.2.2 Syntax

The syntax of the **`Encoding`** declaration is shown in Figure 99.

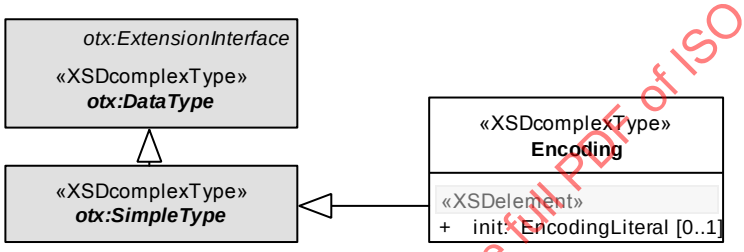

Figure 99 — Data model view: StringUtil data types

18.2.3 Semantics

18.2.3.1 General

The **`Encoding`** enumeration type of the OTX StringUtil extension is based on **`otx:SimpleType`**.

18.2.3.2 Encoding

`Encoding` is an enumeration type describing different kinds of string encodings which shall be supported by OTX applications.

The list of allowed enumeration values is defined as follows:

— **`US-ASCII`**: String encoding specified by [ISO/IEC 646]
— **`ISO-8859-1`**: String encoding specified by [ISO/IEC 8859-1]
— **`UTF-8`**: String encoding from the UTF encoding family, see [ISO/IEC 10646]
— **`UTF-16BE:`** String encoding from the UTF encoding family, see [ISO/IEC 10646]
— **`UTF-16LE:`** String encoding from the UTF encoding family, see [ISO/IEC 10646]
— **`UTF-16:`** String encoding from the UTF encoding family, see [ISO/IEC 10646]
— **`BIN:`** Simple radix-based string encoding for binary-dump strings
— **`OCT:`** Simple radix-based string encoding for octal-dump strings
— **`HEX:`** Simple radix-based string encoding for hexadecimal-dump strings

IMPORTANT — `Encoding` values may occur as operands of comparisons (cf. Part 2 of ISO 13209, relational operations). For this case, the following order relation shall apply:
`US-ASCII < ISO-8859-1 < UTF-8 < UTF-16BE < UTF-16LE < UTF-16 < BIN < OCT < HEX`.

IMPORTANT — When applying `otx:ToString` on an `Encoding` value, the resulting string shall be the name of the enumeration value, e.g. `otx:ToString(UTF-8)="UTF-8"`. Furthermore, applying `otx:ToInteger` shall return the index of the value in the `Encodings` enumeration (smallest index is `0`). The behaviour is undefined for other conversion terms (cf. Part 2 of ISO 13209).

`Encoding` is an **`otx:SimpleType`**. Its members have the following semantics:

— **`<init> : EncodingLiteral [0..1]`**

This optional element stands for the hard-coded initialisation value of the identifier at declaration time.

— **`value : Encodings={US-ASCII|ISO-8859-1|UTF-8|UTF-16BE|UTF-16LE|UTF-16|BIN|OCT|HEX} [1]`**

This attribute shall contain one of the values defined in the **`Encodings`** enumeration.

IMPORTANT — If the `Encoding` declaration is not explicitly initialized (omitted `<init>` element), the default value shall be `US-ASCII`.

18.3 Exceptions

18.3.1 Overview

All elements referenced in this clause are derived from the OTX Core **`Exception`** type as defined by Part 2 of ISO 13209. They represent the full set of exceptions added by the OTX StringUtil extension.

18.3.2 Syntax

The syntax of all OTX StringUtil exception type declarations is shown in Figure 100.

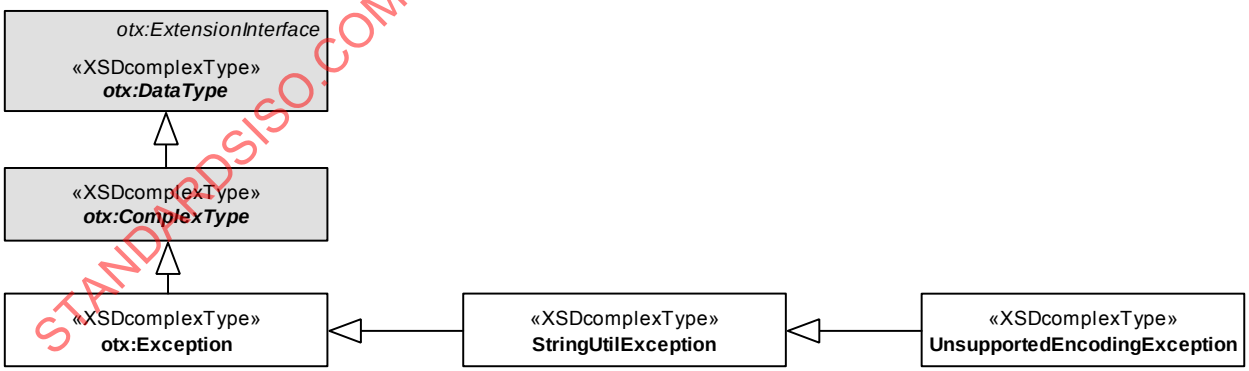

Figure 100 — Data model view: StringUtil exceptions

18.3.3 Semantics

18.3.3.1 General

Since all OTX StringUtil exception types are implicit exceptions whithout initialisation parts, they can not be declared constant.

18.3.3.2 StringUtilException

The **`StringUtilException`** is the super class for all exceptions in the StringUtil extension. A **`StringUtilException`** shall be used in case the more specific exception types described in the remainder of this section do not apply to the problem at hand.

18.3.3.3 UnsupportedEncodingException

An **`UnsupportedEncodingException`** is thrown if the given encoding to be used in instances of the **`Decode`** or **`Encode`** terms is not supported by the runtime system.

18.4 Variable access

18.4.1 Overview

As specified in Part 2 of ISO 13209, OTX extensions shall define a variable access type for each datatype they define (exception types inclusively). All variable access types are derived from the OTX Core **`Variable`** extension interface. The following specifies all variable access types defined for the StringUtil extension.

18.4.2 Syntax

Figure 101 shows the syntax of the StringUtil extension's variable access types.

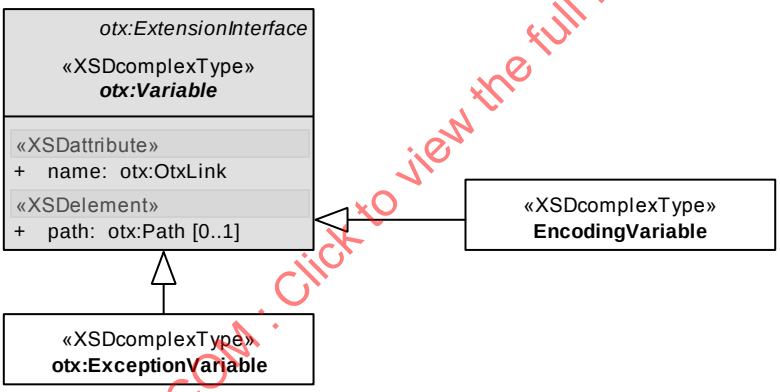

Figure 101 — Data model view: StringUtil variable access types

18.4.3 Semantics

The general semantics for all variable access types shall apply. Please refer to Part 2 of ISO 13209 for details.

18.5 Terms

18.5.1 Overview

The OTX StringUtil extension provides terms which OTX authors may use for analyzing and manipulating **`String`** values.

18.5.2 Syntax

Figure 102 shows the syntax of all terms of the StringUtil extension.

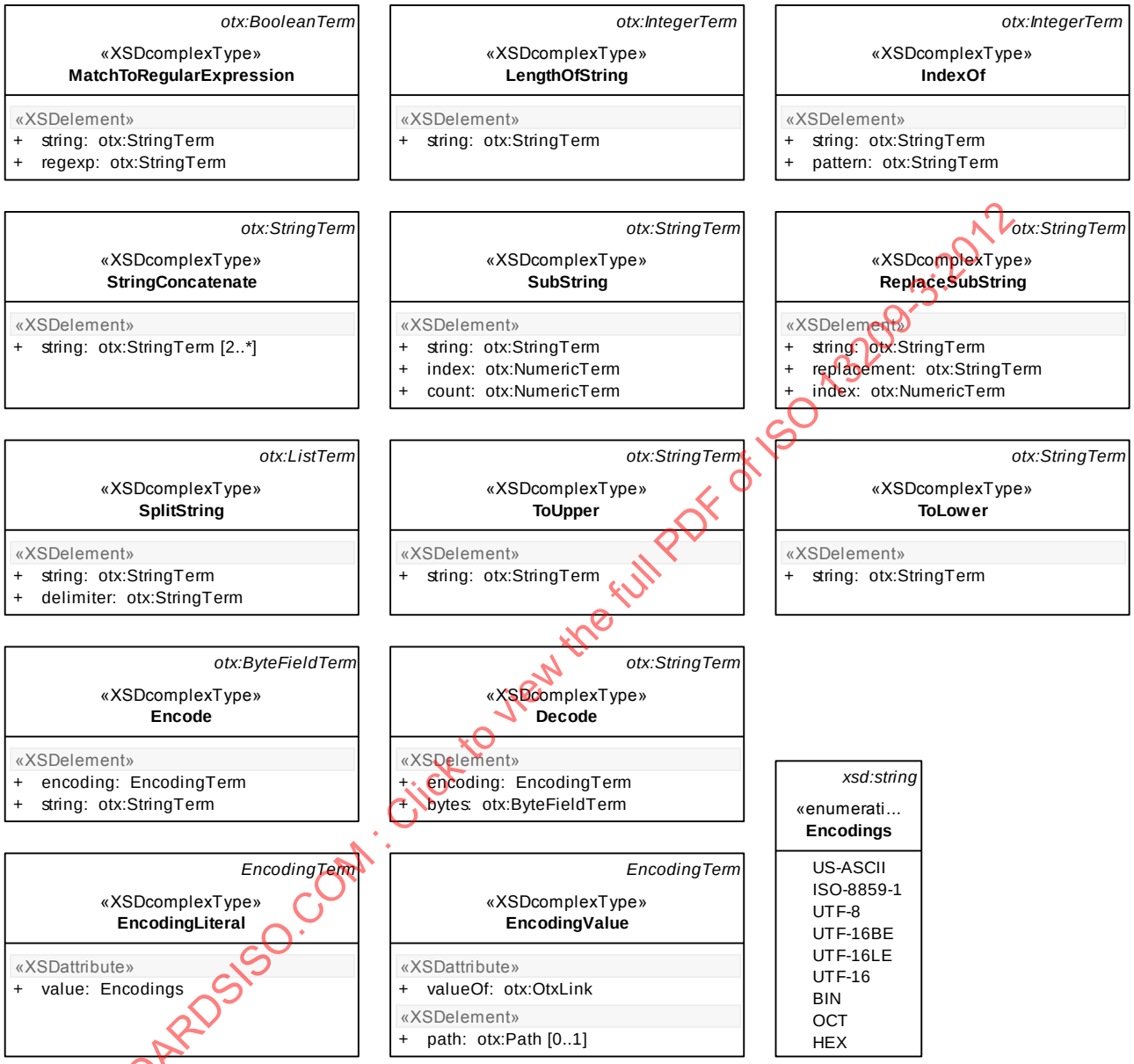

Figure 102 — Data model view: StringUtil terms

18.5.3 Semantics

18.5.3.1 ReplaceSubString

The **`ReplaceSubString`** term shall yield a new string value which is constructed out of an original string whose contents are overwritten by a replacement string, starting at a given position. If the replacement exceeds the right end of the original string, the length of the resulting string is expanded accordingly.

EXAMPLE For the string "Hello world" whose contents shall be replaced at index 2 by the new string "yho", the resulting string is "Heyho world".

ReplaceSubString is an **otx:StringTerm**. Its members have the following semantics:

— **<string> : otx:StringTerm [1]**

Represents the original **String**.

— **<replacement> : otx:StringTerm [1]**

Represents the **String** value that shall replace the part starting at **index** of the original **String**.

— **<index> : otx:NumericTerm [1]**

The **Integer** value represents the position in the original **String** where the replacement starts (the first character in the original **String** has the index zero). **Float** values shall be truncated.

Throws:

— **otx:OutOfBoundsException**
If the index is negative or exceeds the length of the original string.

18.5.3.2 MatchToRegularExpression

The **MatchToRegularExpression** term shall return **true**, if and only if the regular expression performed on the input string is true.

MatchToRegularExpression is an **otx:BooleanTerm**. Its members have the following semantics:

— **<string> : otx:StringTerm [1]**

Represents the **String** which shall be analysed.

— **<regexp> : otx:StringTerm [1]**

This **String** value represents the regular expression, which shall be performed on the other **String** given by the **<string>** argument. The regular expression shall be conforming to the *PERL 5 Regular Expression Description Version 12*.

Throws:

— **otx:OutOfBoundsException**
If the regular expression is not conforming to the *PERL Regular Expression Description*.

18.5.3.3 StringConcatenate

The **StringConcatenate** term shall yield a new string which is the concatenation of two or more strings.

The term **StringConcatenate** is an **otx:StringTerm**. Its members have the following semantics:

— **<string> : otx:StringTerm [2..*]**

The elements represent the strings that shall be concatenated.

18.5.3.4 SubString

The **SubString** term shall return a sub-string read out of a given string. The to-be-read sub-string is defined by index and count arguments.

SubString is an **otx:StringTerm**. Its members have the following semantics:

— **<string> : otx:StringTerm [1]**

Represents the input string from which a sub-string shall be extracted.

— **<index> : otx:NumericTerm [1]**

Represents the index starting from which the sub-string shall be read. **Float** values shall be truncated.

— **<count> : otx:NumericTerm [1]**

Represents the number of characters to be read from the original string. Reading shall not exceed the last character in the input string (so the resulting sub-string length will be less than **<count>** in some cases). **Float** values shall be truncated.

Throws:

— **otx:OutOfBoundsException**
If index is negative or exceeds the length of the input string, or if count is negative.

18.5.3.5 LengthOfString

The term **LengthOfString** shall return the number of characters constituting a given string.

LengthOfString is an **otx:IntegerTerm**. Its members have the following semantics:

— **<string> : otx:StringTerm [1]**

Represents the input string whose length shall be retrieved.

18.5.3.6 IndexOf

The term **IndexOf** shall return the index within a string of the first occurrence of the specified pattern string. If the pattern is not included in the string, the return value shall be -1.

IndexOf is an **otx:IntegerTerm**. Its members have the following semantics:

— **<string> : otx:StringTerm [1]**

Represents the input string in which the pattern shall be searched.

— **<pattern> : otx:StringTerm [1]**

Represents the sub-string which shall be searched for in the input string.

18.5.3.7 SplitString

The term **SplitString** shall return a list of strings. The returned list shall contain each substring of the original string that is terminated by a given delimiter string or by the end of the string. The substrings in the resulting list shall be in the order in which they occur in the original string. If the delimiter does not occur in any part of the original string, then the resulting list shall have just one element, namely the original string. If the delimiter is the empty string, the original string shall be split into single characters. The search for delimiters in the string shall be case sensitive.

SplitString is an **otx:ListTerm**. Its members have the following semantics:

— **<string> : otx:StringTerm [1]**

Represents the original string which shall be split.

— **<delimiter> : otx:StringTerm [1]**

Represents the delimiter string. The original string is split at each place where a delimiter string occurs.

18.5.3.8 ToUpper

The term **ToUpper** shall return the uppercase counterpart of a given string.

ToUpper is an **otx:StringTerm**. Its members have the following semantics:

— **<string> : otx:StringTerm [1]**

Represents the input string whose uppercase counterpart shall be returned.

18.5.3.9 ToLower

ToLower shall return the lowercase counterpart of a given string.

The term **ToLower** is an **otx:StringTerm**. Its members have the following semantics:

— **<string> : otx:StringTerm [1]**

Represents the input string whose lowercase counterpart shall be returned.

18.5.3.10 Encode

The term **Encode** shall encode a **String** by using a given encoding. The result is a **ByteField** representing the encoded string.

NOTE **Encode** corresponds to the method **java.lang.String.getBytes(String charsetName)** of the Java™ programming language [1].

Encode is an **otx:ByteFieldTerm**. Its members have the following semantics:

— **<encoding> : EncodingTerm [1]**

This represents the encoding to be used. The set of standard encodings which shall be supported by any runtime system is given by the **Encoding** enumeration (see 18.2.3.2).

In case of **BIN**, **OCT** or **HEX** encoding, the input string shall only contain numbers and letters defined for the given radix. Allowed symbols are **{0,1}** for **BIN**, **{0-7}** for **OCT** and **{0-9,A-F}** for **HEX** encoding. Also the input string shall be of correct length according to the radix. This requires that for **BIN**, **len%8==0**, for **OCT** **len%3==0** and for **HEX** **len%2==0** shall apply, where **len** is the length of the input string. This enforces an unambiguous and exact mapping of e.g. every two **HEX** symbols to exactly one byte in the output. With this, applying e.g. **Encode("A1B2", HEX)** is allowed, but applying **Encode("A1B2", BIN)** and **Encode("A1B", HEX)** will cause an **OutOfBoundsException**.

— **<string> : otx:StringTerm [1]**

Represents the string which shall be transformed to a **ByteField** by using the given encoding.

Throws:

— **UnknownEncodingException**
if the encoding is not known by the runtime system.

— **otx:OutOfBoundsException**
if the string cannot be encoded with the given encoding.

18.5.3.11 Decode

The term **Decode** shall construct a **String** by decoding a given **ByteField** using a specified encoding.

NOTE **Decode** corresponds to the constructor **java.lang.String(byte[] bytes, String charsetName)** of the Java™ programming language [1].

Decode is an **otx:StringTerm**. Its members have the following semantics:

— **<encoding> : EncodingTerm [1]**

This element represents the encoding to be used. The set of standard encodings which shall be supported by any runtime system is given by the **Encoding** enumeration (see 18.2.3.2).

When using **BIN**, **OCT** or **HEX** for getting a dump of a **ByteField**, the resulting string shall not contain delimiters in between single byte dump parts, nor shall there be a prefix like e.g. **"0x"** marking the radix. Furthermore, leading zeros shall not be removed during bytewise translation. Therefore e.g. when applying **decode({00000000,11100001,00001011}, HEX)**, the result shall be **"00E10B"**.

— **<bytes> : otx:ByteFieldTerm [1]**

Represents the **ByteField** which shall be transformed to a **String** by using the given encoding.

Throws:

— **UnknownEncodingException**
if the encoding is not known by the runtime system.

— **otx:OutOfBoundsException**
if the bytes in the **ByteField** are not valid in the given encoding.

18.5.3.12 EncodingTerm

The abstract type **EncodingTerm** is an **otx:SimpleTerm**. It serves as a base for all concrete terms which return an **Encoding** value (see 18.2.3.2). It has no special members.

18.5.3.13 EncodingValue

This term returns the **Encoding** stored in an **Encoding** variable. For more information on value-terms and the syntax and semantics of the **valueOf** attribute and **<path>** element, please refer to Part 2 of ISO 13209.

Associated checker rules:

— Core_Chk053 – no dangling OtxLink associations (see Part 2 of ISO 13209)

Throws:

— **otx:OutOfBoundsException**
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).

18.5.3.14 EncodingLiteral

This term shall return an **`Encoding`** value (see 18.2.3.2) from a hard-coded literal.

`EncodingLiteral` is an **`EncodingTerm`**. Its members have the following semantics:

— **`value : Encodings={US-ASCII|ISO-8859-1|UTF-8|UTF-16BE|UTF-16LE|UTF-16|BIN|OCT|HEX} [1]`**

This attribute shall contain one of the values defined in the **`Encodings`** enumeration.

Annex A
(normative)

Comprehensive checker rule listing

A.1 Overview

The checker rules defined in the listing below extend the set of checker rules given by Part 2 of ISO 13209. The rules are categorized according to the OTX extension they belong to.

NOTE Checker rule names are composed of a prefix relating to respective extension, a running number and a long name, so for instance "Measure_Chk002 – correct ExecuteDeviceService arguments".

A.2 Listing

A.2.1 Checker rules for DiagCom extension

DiagCom_Chk001 – No Path in ExecuteDiagService response parameter arguments

Severity: Critical

Criterion: The **<target>** variable in any of an **ExecuteDiagService** action's response parameter mappings shall not be part of a compound data structure like an **otx:List** or **otx:Map**. Therefore the **<path>** member element of **<target>** shall not be used within any **ResponseParameter**.

A.2.2 Checker rules for EventHandling extension

Event_Chk001 – Correct data types of ThresholdExceededEventSource arguments

Severity: Critical

Criterion: In a **ThresholdExceededEventSource**, the data type of the monitored variable (given by the **<variable>** element) and the return type of the lower and upper threshold terms (given by **<lowerThreshold>** and **<upperThreshold>**) shall accord.

Event_Chk002 – No Path in MonitorChange related terms

Severity: Critical

Criterion: The **<variable>** in monitor change event related terms shall not be part of a compound data structure like an **otx:List** or **otx:Map**. Therefore the **<path>** member element of **<variable>** shall not be used in within the scope of such terms. This rule applies to the terms **MonitorChangeEventSource**, its descendant **ThresholdExceededEventSource** as well as **IsMonitorChangeEvent**.

A.2.3 Checker rules for Measure extension

Measure_Chk001 – correct target for ExecuteDeviceService and DeviceEventSource

Severity: Critical

Criterion: The **device** attribute of an **ExecuteDeviceService** action or a **DeviceEventSource** term shall refer to a **Signature** without realisation or a **Signature** with **DeviceSignature** realisation. No other **SignatureRealisation** type but the **DeviceSignature** type shall be allowed.

Measure_Chk002 – executed device service is declared in device signature

Severity: Critical

Criterion: The to-be-executed device service given by an **`ExecuteDeviceService`** action (value of the action's **`service`** attribute) shall be declared in the corresponding device signature (given by the action's **`device`** attribute).

Measure_Chk003 – correct ExecuteDeviceService arguments

Severity: Critical

Criterion: In an **`ExecuteDeviceService`** action, the given arguments shall match to the corresponding service parameter declaration in the corresponding device signature. This concerns parameter names as well as data types.

Measure_Chk004 – ExecuteDeviceService input argument omission

Severity: Critical

Criterion: For any omitted input argument (when calling a device service), an initial value shall be defined in the declaration of the corresponding parameter in the device signature.

Measure_Chk005 – No Path in ExecuteDeviceService ouput arguments

Severity: Critical

Criterion: The **`<target>`** variable in any of the **`ExecuteDeviceService`** action's output arguments shall not be part of a compound data structure like an **`otx:List`** or **`otx:Map`**. Therefore the **`<path>`** member element of **`<target>`** shall not be used within any **`DeviceServiceOutArgument`**.

A.2.4 Checker rules for HMI extension

HMI_Chk001 – correct list type for ChoiceDialog options

Severity: Critical

Criterion: The list given by the **`<options>`** element in a **`ChoiceDialog`** action shall be a list of strings. No other item types are allowed.

HMI_Chk002 – correct target for OpenScreen

Severity: Critical

Criterion: The **`screen`** attribute of an **`OpenScreen`** action shall refer to a **`Signature`** without realisation or a **`Signature`** with **`ScreenSignature`** realisation. No other **`SignatureRealisation`** type but the **`ScreenSignature`** type shall be allowed.

HMI_Chk003 – correct OpenScreen arguments

Severity: Critical

Criterion: In an **`OpenScreen`** action, the given arguments shall match to the corresponding screen parameter declaration in the corresponding screen signature. This concerns parameter names as well as data types. There shall be no arguments which have no counterpart parameter declaration in the signature.

HMI_Chk004 – OpenScreen term, input and input/output argument omission

Severity: Critical

Criterion: For any omitted term, input or input/output argument (when opening a screen by **`OpenScreen`** action), an initial value shall be defined in the declaration of the corresponding parameter in the screen signature.

HMI_Chk005 – no Path in connected OpenScreen arguments

Severity: Critical

Criterion: The **<variable>** in any of an **OpenScreen** action's in-, inout- and out-arguments shall not be part of a compound data structure like an **otx:List** or **otx:Map**. Therefore the **<path>** member element of **<variable>** shall not be used within any of these argument definitions.

A.2.5 Checker rules for Quantities extension

Quantities_Chk001 – correct unit linking

Severity: Critical

Criterion: For all elements of type **quant:Unit** or **i18n:UnitGroup**, only *simple XLinks* shall be allowed, as specified by [W3C XLink]. Therefore, **xlink:type="simple"** shall be set as attribute of **quant:Unit** or **i18n:UnitGroup** elements.

Quantities_Chk002 – no dangling unit definition links

Severity: Warning

Criterion: Unit definitions referenced by **UnitLiteral** terms should exist in the unit specification.

Comment: In addition to this checker rule, the exception **UnknownUnitException** is defined for **UnitLiteral**, since it cannot always be guaranteed that the unit specification available at authoring/checking time is identical to the unit specification available at runtime.

Annex B
(normative)

OTX DiagCom extension data type mappings

B.1 General considerations

OTX has a more simplified typing system when compared to the data type definitions of the ODX [ISO 22901] or MVCI [ISO 22900] standard. When executing test sequences that are using the OTX DiagCom extension together with an ODX/MVCI based communication component, an OTX runtime system needs to be able to map OTX data types to ODX/MVCI data types and vice versa. The following rules apply:

— When converting from OTX data types to ODX/MVCI data types, the runtime system has to follow the mapping laid out in the tables defined in the remainder of this annex.

  — The conversion has to work within the groupings as provided by the OTX data type definition. E.g. an **`otx:Float`** can be automatically converted to a MVCI **`A_FLOAT32`** or a MVCI **`A_FLOAT64`**, depending on the target data type required by the diagnostic data set.

  — In case a conversion can not be made because the value of an OTX variable exceeds the limits of the target data type the OTX runtime system should throw an **`otx:OutOfBoundsException`**.

  — In case a conversion can not be made because the value of an **`otx:String`** variable contains characters that can not be mapped to the target string data type the OTX runtime system should throw an **`otx:TypeMismatchException`**. This can happen when an **`otx:String`** containing Unicode characters outside of the ASCII range is supposed to be mapped onto an A_ASCIISTRING MVCI data type parameter.

— When converting from ODX/MVCI data types to OTX data types, the runtime system has to follow the mapping laid out in the tables defined in the remainder of this annex.

B.2 Mapping ODX data types to OTX data types

This mapping definition is provided for OTX runtime systems working in conjunction with an ODX-compliant communication component not conforming to the MVCI standard.

Table B.1 — Mapping ODX data types to OTX data types

ODX data type	OTX data type
`A_INT32`	`otx:Integer`
`A_UINT32`	`otx:Integer`
`A_FLOAT32`	`otx:Float`
`A_FLOAT64`	`otx:Float`
`A_ASCIISTRING`	`otx:String`
`A_UTF8STRING`	`otx:String`
`A_UNICODE2STRING`	`otx:String`
`A_BYTEFIELD`	`otx:ByteField`

B.3 Mapping MVCI data types to OTX data types

This mapping definition is provided for OTX runtime systems working in conjunction with an MVCI communication component.

Table B.2 — Mapping MVCI data types to OTX data types

MVCI data type	OTX data type
A_BOOLEAN	otx:Boolean
A_BITFIELD	otx:ByteField
A_BYTEFIELD	otx:ByteField
A_FLOAT32	otx:Float
A_FLOAT64	otx:Float
A_INT16	otx:Integer
A_INT32	otx:Integer
A_INT64	otx:Integer
A_UINT16	otx:Integer
A_UINT32	otx:Integer
A_UINT64	otx:Integer
A_UINT8	otx:Integer
A_ASCIISTRING	otx:String
A_UNICODE2STRING	otx:String

Annex C
(normative)

OTX DiagMetaData auxiliary for the OTX DiagCom extension

C.1 Description

In constrast to OTX extensions like OTX DiagCom or OTX HMI, the OTX DiagMetaData auxiliary has a special role. Its purpose is not to directly extend the feature set of OTX by providing additional actions, terms and data types for handling a certain runtime usecase – OTX diagnostic sequences can be built and executed without the use of the DiagMetaData auxiliary. Instead, this interface only serves an auxiliary function and has no runtime relevance. It is, however, quite important in the actual process of constructing OTX sequences with an OTX editor.

When an author is building an OTX sequence with OTX DiagCom functionality, one of the desired and most helpful features of an editor is to support the user by providing access to the underlying diagnostic data (e.g. ODX data). If the author wants to establish communication and creates a **`ComChannel`** using the **`GetComChannel`** term, the editor tool can list all available channels allowing the user to pick the one he wants to open. After the author has selected one of the channels, he might want to insert a **`CreateDiagService`** action into the sequence. Again, the editor can list all available diagnostic services for a selected communication channel by looking into the data.

Oftentimes, the author might want to create a sub-procedure and use a **`DiagService`** or **`Parameter`** as an input argument to reuse functionality and reduce redundancy in the project. This is possible by using standard OTX features; the sequence can be executed successfully without a problem. The issue here is, however, that the editor cannot support the author in creating the sub-procedure by providing a look into the diagnostic data at authoring time. If a sub-procedure expects a **`DiagService`** parameter, the editor cannot – without complex and time-consuming analysis – know on which communication channel to look for the wanted diagnostic service before actually executing code.

This is the (authoring time) information gap that the OTX DiagMetaData auxiliary is meant to close.

C.2 Syntax

The OTX DiagMetaData auxiliary represents a standalone XML schema which can be used within any **`<metaData>`** section of OTX documents. The auxiliary schema introduces the top-level XML element **`<diagMetaData>`**, the **`DiagMetaData`** type as well as the **`MetaDataPath`** type, as shown in Figure C.1 (please refer to Part 2 of ISO 13209 for details on the OTX Core type **`otx:MetaData`**).

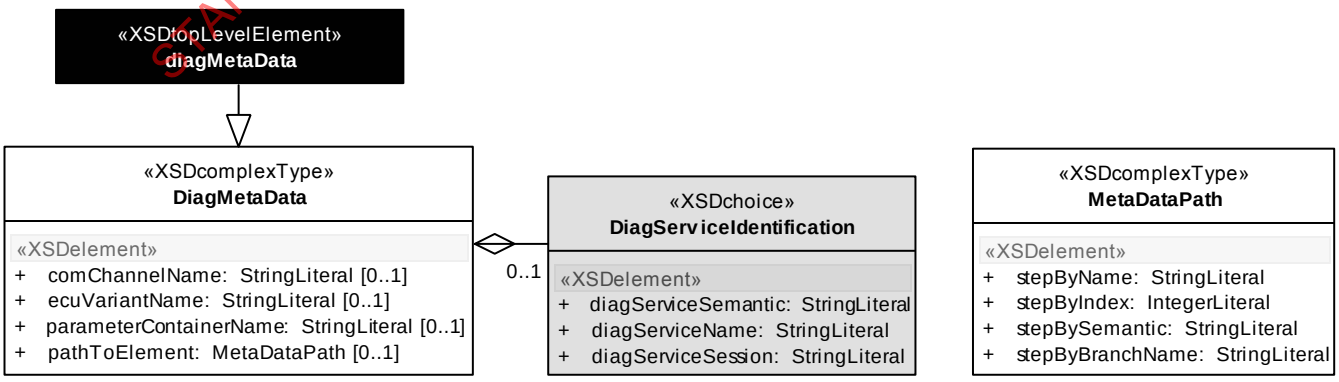

Figure C.1 — Data model view: OTX DiagMetaData auxiliary

NOTE The XSD complex type **MetaDataPath** is of **<xsd:choice> [1..*]** content-type, which is not explicitly shown in the figure above.

C.3 Semantics

The OTX DiagMetaData auxiliary provides a standardized XML structure for being used within **<metaData>** elements in OTX documents. More precisely, the **<diagMetaData>** element shall appear (if used) within the **xsd:any** content of a **<data>** element of a **<diagMetaData>** element. This way, the information needed to provide context to e.g. an input argument of a sub-procedure can be attached to the procedure without affecting the runtime semantics of the procedure.

IMPORTANT — The information in <diagMetaData> is not relevant for execution and shall only support the author by providing context information at authoring time.

The top-level **<diagMetaElement>** is of **DiagMetaData** type. It consists of a set of string literals. These are identifiers which represent the missing context information that is needed to locate the input arguments in the data used by the editor. E.g. if a **DiagService** is used as an input argument, the name of the communication channel (in ODX terminology: the **SHORT-NAME** of the **LOGICAL-LINK**) can be stored here; If a **Parameter** is used, the name of both communication channel and diagnostic service can be stored.

There are multiple elements an editor might need information about:

— **<comChannelName> : otx:StringLiteral [0..1]**

This element represents the name of the used communication channel.

— **<ecuVariantName> : otx:StringLiteral [0..1]**

This element represents the name of the identified variant.

— **DiagServiceIdentification : xsd:choice [0..1]**

While the communication channel and variant are represented by their unique names, a diagnostic service can be created in several different ways in OTX: By the OTX DiagCom extension's creation terms **diag:CreateDiagServiceByName** and **diag:CreateDiagServiceBySemantic** as well as by the OTX Flash extension's creation terms **flash:CreateFlashJob**, **flash:CreateFlashJobByName** and **flash:CreateFlashJobBySemantic**. To find data inside a diagnostic service that is used as an input argument for a procedure, the editor therefore needs the name, the semantic or the flash session's name, depending on what information is available. Therefore, this optional **xsd:choice** allows choosing between the following elements:

— **<diagServiceName> : otx:StringLiteral [1]**

The diagnostic service shall be identified in the data by the name given by this element.

— **<diagServiceSemantic> : otx:StringLiteral [1]**

The diagnostic service shall be identified in the data by the semantic given by this element.

— **<diagServiceSession> : otx:StringLiteral [1]**

The diagnostic service shall be identified in the data by the session-name given by this element.

— **<parameterContainerName> : otx:StringLiteral [0..1]**

This element represents the name of the parameter container which contains the parameter the editor is trying to find (e.g. a specific ECU response in a parameter container needs to be located).

— **<pathToElement> : MetaDataPath [0..1]** (this type is of **xsd:choice [1..*]** content)

This element describes the path to a specific parameter within a parameter hierarchy. A parameter path consists of a series of identifiers (steps) to find a specific parameter within a parameter tree structure:

— **<stepByName> : otx:StringLiteral [0..1]**

Represents a step by name in the path to the parameter.

— **<stepByIndex> : otx:StringLiteral [0..1]**

Represents a step by index in the path to the parameter. This is used for structures or fields where the items in the structure or field do not carry individual names.

NOTE Depending on the runtime implementation, the index of a parameter within a structure might not be deterministic. Use with care in this situation.

— **<stepBySemantic> : otx:StringLiteral [0..1]**

Identifies the parameter by semantic (a parameter can be identified via its semantic on each layer of the parameter structure).

— **<stepByBranchName> : otx:StringLiteral [0..1]**

The identifier of the branch element in case one is used in the parameter hierarchy.

NOTE In case of an ODX/MVCI based system, the system knows three elements which act as branches of a decision: **TABLE-ROW**, **CASE** and **ENV-DATA**. The element **<stepByBranchName>** can be used to store the name of such an element to reproduce the decision.

C.4 Example

EXAMPLE 1 Sample of "Procedure_A" in OTX-file "DiagMetaDataExample.otx"

```
<procedure name="Procedure_A" id="p1">
  <specification>Demonstration of DiagMetaData auxiliary features</specification>
  <realisation>
    <declarations>
      <variable name="cc" id="v1">
        <specification>comm channel handle for an ECU</specification>
        <realisation>
          <dataType xsi:type="diag:ComChannel"/>
        </realisation>
      </variable>
    </declarations>
    <flow>
      <action id="a1">
        <specification>Open comm channel to "MY_ECU", assign to cc</specification>
        <realisation xsi:type="Assignment">
          <result xsi:type="diag:ComChannelVariable" name="cc"/>
          <term xsi:type="diag:GetComChannel">
            <diag:identifier xsi:type="StringLiteral" value="MY_ECU"/>
            <diag:performVariantSelection xsi:type="BooleanLiteral" value="true"/>
          </term>
        </realisation>
      </action>
      <!-- Do something on that communication channel, e.g. execute a service, etc. -->

      <action id="a2">
        <specification>Call "Procedure_B", pass cc as an input argument</specification>
        <realisation xsi:type="ProcedureCall" procedure="Procedure_B">
          <arguments>
            <inArg param="channel">
              <term xsi:type="diag:ComChannelValue" valueOf="cc"/>
            </inArg>
          </arguments>
        </realisation>
      </action>
    </flow>
  </realisation>
</procedure>
```

EXAMPLE 2 Sample of "Procedure_B" in OTX-file "DiagMetaDataExample.otx"

```
<procedure name="Procedure_B" id="p2">
  <specification>Demonstration of DiagMetaData auxiliary features</specification>
  <realisation>
    <parameters>
      <inParam name="channel" id="par1">
        <specification>comm channel passed in from the calling procedure</specification>
        <metaData>
          <data key="dmd1">
            <dmd:diagMetaData>
              <dmd:comChannelName value="MY_ECU"/>
              <dmd:ecuVariantName value="VARIANT_A"/>
            </dmd:diagMetaData>
          </data>
        </metaData>
        <realisation>
          <dataType xsi:type="diag:ComChannel"/>
        </realisation>
      </inParam>
    </parameters>
    <flow>
      <!-- Do something on that communication channel, e.g. execute a service, etc. -->
    </flow>
  </realisation>
</procedure>
```

The OTX code snippets above demonstrate the use of DiagMetaData for a procedure call: In **Procedure_A**, a communication channel for an ECU **MY_ECU** is created and assigned to a **ComChannel** variable **cc**. This variable is passed to **Procedure_B** via procedure call. **Procedure_B** might be called by multiple other procedures, which means that **Procedure_B** can not know the actual name of the ECU at authoring time. The actual ECU the **ComChannel** connects to will only be known at runtime. When the OTX author accesses the **ComChannel** variable in **Procedure_B**, e.g. while implementing a **CreateDiagServiceByName** term, the editor cannot support the author. For instance, listing all available diagnostic services for that **ComChannel** variable would not be possible.

For supporting this, the author can provide the editor with the missing information by using the **<diagMetaData>** element. In the example above, the author specified the actual ECU name and the ECU variant name for which he wants support.

Annex D
(normative)

OTX standard signature documents

D.1 OTX Job standard signatures (JobInterfaces.otx)

```
<?xml version="1.0" encoding="UTF-8"?>
<otx name="JobInterfaces" package="org.iso.otx.signatures" id="otxjobstdsig"
  timestamp="2010-03-18T14:40:10" version="1.0"
  xmlns="http://iso.org/OTX/1.0.0"
  xmlns:diag="http://iso.org/OTX/1.0.0/DiagCom"
  xmlns:job="http://iso.org/OTX/1.0.0/Job"
  xmlns:flash="http://iso.org/OTX/1.0.0/Flash"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <signatures>

    <signature name="SingleEcuJob" id="sej">
      <specification>Declaration of a single job entry point</specification>
      <realisation xsi:type="ProcedureSignature">
        <parameters>
          <inParam name="comChannelHandle" id="sej_p1">
            <specification> Communication Channel from Tester Environment</specification>
            <realisation><dataType xsi:type="diag:ComChannel"/></realisation>
          </inParam>
          <inoutParam name="jobHandle" id="sej_p2">
            <specification> Job handle with parameters from outside </specification>
            <realisation><dataType xsi:type="diag:DiagService"/></realisation>
          </inoutParam>
        </parameters>
      </realisation>
    </signature>

    <signature name="FlashJob" id="fj">
      <specification>Declaration of a flash job entry point</specification>
      <realisation xsi:type="ProcedureSignature">
        <parameters>
          <inParam name="comChannelHandle" id="fj_p1">
            <specification> Communication Channel from Tester Environment</specification>
            <realisation><dataType xsi:type="diag:ComChannel"/></realisation>
          </inParam>
          <inParam name="Session" id="fj_p2">
            <specification>Session variable from Tester</specification>
            <realisation><dataType xsi:type="flash:FlashSession"/></realisation>
          </inParam>
          <inoutParam name="jobHandle" id="fj_p3">
            <specification> Job handle with parameters from outside </specification>
            <realisation><dataType xsi:type="diag:DiagService"/></realisation>
          </inoutParam>
        </parameters>
      </realisation>
    </signature>

    <signature name="SecurityAccessJob" id="saj">
      <specification>Declaration of a security access job entry point</specification>
      <realisation xsi:type="ProcedureSignature">
        <parameters>
          <inParam name="requestParameters" id="saj_p1">
            <realisation><dataType xsi:type="diag:Request"/></realisation>
          </inParam>
          <inParam name="comChannelHandle" id="saj_p2">
            <specification> Communication Channel from Tester Environment</specification>
            <realisation><dataType xsi:type="diag:ComChannel"/></realisation>
          </inParam>
          <inoutParam name="jobHandle" id="saj_p3">
            <specification> Job handle with parameters from outside </specification>
            <realisation><dataType xsi:type="diag:DiagService"/></realisation>
          </inoutParam>
        </parameters>
      </realisation>
    </signature>

  </signatures>
</otx>
```

Annex E
(informative)

Test sequence examples

E.1 Diagnostic Communication (IFD_DiagCom.xsd)

E.1.1 DiagComInlineExample.otx

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<otx name="DiagComInlineExample" package="otx.diagcom.example" id="otx1"
  version="1.0" timestamp="2001-12-31T12:00:00"
  xmlns="http://iso.org/OTX/1.0.0"
  xmlns:diag="http://iso.org/OTX/1.0.0/DiagCom"
  xmlns:quant="http://iso.org/OTX/1.0.0/Quantities"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <procedures>
    <procedure name="main" visibility="PUBLIC" id="p1">
      <realisation>
        <declarations>
          <variable name="comChannelHandle" id="v1">
            <realisation>
              <dataType xsi:type="diag:ComChannel"/>
            </realisation>
          </variable>
          <variable name="outputParamHandle" id="v2">
            <realisation>
              <dataType xsi:type="diag:Parameter"/>
            </realisation>
          </variable>
        </declarations>
        <flow>
          <action id="id_selectComChannel">
            <realisation xsi:type="Assignment">
              <result xsi:type="diag:ComChannelVariable" name="comChannelHandle"/>
              <term xsi:type="diag:GetComChannel">
                <diag:identifier xsi:type="StringLiteral" value="ODX_ComChannelName"/>
              </term>
            </realisation>
          </action>
          <action id="id_ExecuteDiagService">
            <realisation xsi:type="diag:ExecuteDiagService">
              <diag:diagService xsi:type="diag:CreateDiagServiceByName">
                <diag:comChannel xsi:type="diag:ComChannelValue" valueOf="comChannelHandle"/>
                <diag:name xsi:type="StringLiteral" value="ODX_DiagServiceName"/>
              </diag:diagService>
              <diag:requestParameters>
                <diag:requestParam>
                  <diag:value xsi:type="StringLiteral" value="ExampleParameterValue"/>
                  <diag:path>
                    <stepByName xsi:type="StringLiteral" value="ODX_RequestParameterShortName"/>
                  </diag:path>
                </diag:requestParam>
              </diag:requestParameters>
              <diag:responseParameters>
                <diag:name xsi:type="StringLiteral" value="ODX_PositiveResponseName"/>
                <diag:responseParam>
                  <diag:target xsi:type="diag:ParameterVariable" name="outputParamHandle"/>
                  <diag:path>
                    <stepByName xsi:type="StringLiteral" value="ODX_ResponseParameterShortName"/>
                  </diag:path>
                </diag:responseParam>
              </diag:responseParameters>
            </realisation>
          </action>
          <action id="id_CloseComChannel">
            <realisation xsi:type="diag:CloseComChannel">
              <diag:comChannel xsi:type="diag:ComChannelVariable" name="comChannelHandle"/>
            </realisation>
          </action>
        </flow>
      </realisation>
    </procedure>
  </procedures>
</otx>
```

E.1.2 DiagComDynamicExample.otx

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<otx name="DiagComDynamicExample" package="otx.diagcom.example" id="otx1"
  version="1.0" timestamp="2001-12-31T12:00:00"
  xmlns="http://iso.org/OTX/1.0.0"
  xmlns:diag="http://iso.org/OTX/1.0.0/DiagCom"
  xmlns:quant="http://iso.org/OTX/1.0.0/Quantities"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <procedures>
    <procedure name="main" visibility="PUBLIC" id="p1">
      <realisation>
        <declarations>
          <variable name="comChannelHandle" id="v1">
            <realisation>
              <dataType xsi:type="diag:ComChannel"/>
            </realisation>
          </variable>
          <variable name="diagServiceHandle" id="v2">
            <realisation>
              <dataType xsi:type="diag:DiagService"/>
            </realisation>
          </variable>
          <variable name="requestHandle" id="v3">
            <realisation>
              <dataType xsi:type="diag:Request"/>
            </realisation>
          </variable>
          <variable name="inputParamHandle" id="v4">
            <realisation>
              <dataType xsi:type="diag:Parameter"/>
            </realisation>
          </variable>
          <variable name="resultHandle" id="v5">
            <realisation>
              <dataType xsi:type="diag:Result"/>
            </realisation>
          </variable>
          <variable name="responseHandle" id="v6">
            <realisation>
              <dataType xsi:type="diag:Response"/>
            </realisation>
          </variable>
          <variable name="outputParamHandle" id="v7">
            <realisation>
              <dataType xsi:type="diag:Parameter"/>
            </realisation>
          </variable>
          <variable name="diagServiceResultQuantity" id="v8">
            <realisation>
              <dataType xsi:type="quant:Quantity"/>
            </realisation>
          </variable>
          <variable name="diagServiceResultValue" id="v9">
            <realisation>
              <dataType xsi:type="Integer"/>
            </realisation>
          </variable>
          <variable name="diagServiceResultUnit" id="v10">
            <realisation>
              <dataType xsi:type="quant:Unit"/>
            </realisation>
          </variable>
        </declarations>
        <flow>
          <action id="id_selectComChannel">
            <realisation xsi:type="Assignment">
              <result xsi:type="diag:ComChannelVariable" name="comChannelHandle"/>
              <term xsi:type="diag:GetComChannel">
                <diag:identifier xsi:type="StringLiteral" value="ODX_ComChannelName"/>
              </term>
            </realisation>
          </action>
          <action id="id_getServiceHandle">
            <realisation xsi:type="Assignment">
              <result xsi:type="diag:DiagServiceVariable" name="diagServiceHandle"/>
              <term xsi:type="diag:CreateDiagServiceByName">
                <diag:comChannel xsi:type="diag:ComChannelValue" valueOf="comChannelHandle"/>
                <diag:name xsi:type="StringLiteral" value="ODX_ExampleServiceShortName"/>
              </term>
            </realisation>
          </action>
```

```
        <action id="id_getRequestHandle">
          <realisation xsi:type="Assignment">
            <result xsi:type="diag:RequestVariable" name="requestHandle"/>
            <term xsi:type="diag:GetRequest">
              <diag:diagService xsi:type="diag:DiagServiceValue" valueOf="diagServiceHandle"/>
            </term>
          </realisation>
        </action>
        <action id="id_getParameterFromRequest">
          <realisation xsi:type="Assignment">
            <result xsi:type="diag:ParameterVariable" name="inputParamHandle"/>
            <term xsi:type="diag:GetParameterByPath">
              <diag:parameterContainer xsi:type="diag:RequestValue" valueOf="requestHandle"/>
              <diag:path>
                <stepByName xsi:type="StringLiteral" value="ODX_RequestParameterShortName"/>
              </diag:path>
            </term>
          </realisation>
        </action>
        <action id="id_setValueOfParameter">
          <realisation xsi:type="diag:SetParameterValue">
            <diag:parameter xsi:type="diag:ParameterValue" valueOf="inputParamHandle"/>
            <diag:value xsi:type="StringLiteral" value="TestStringValue"/>
          </realisation>
        </action>
        <action id="id_executeDiagService">
          <realisation xsi:type="diag:ExecuteDiagService">
            <diag:diagService xsi:type="diag:DiagServiceValue" valueOf="diagServiceHandle"/>
            <diag:result xsi:type="diag:ResultVariable" name="resultHandle"/>
          </realisation>
        </action>
        <action id="id_getResponseFromResult">
          <realisation xsi:type="Assignment">
            <result xsi:type="diag:ResponseVariable" name="responseHandle"/>
            <term xsi:type="diag:GetFirstResponse">
              <diag:result xsi:type="diag:ResultValue" valueOf="resultHandle"/>
            </term>
          </realisation>
        </action>
        <action id="id_getParameterFromResponse">
          <realisation xsi:type="Assignment">
            <result xsi:type="diag:ParameterVariable" name="outputParamHandle"/>
            <term xsi:type="diag:GetParameterByPath">
              <diag:parameterContainer xsi:type="diag:ResponseValue" valueOf="responseHandle"/>
              <diag:path>
                <stepByName xsi:type="StringLiteral" value="ODX_ResponseParameterShortName"/>
              </diag:path>
            </term>
          </realisation>
        </action>
        <action id="id_getResponseParameterQuantity">
          <realisation xsi:type="Assignment">
            <result xsi:type="quant:QuantityVariable" name="diagServiceResultQuantity"/>
            <term xsi:type="diag:GetParameterValueAsQuantity">
              <diag:parameter xsi:type="diag:ParameterValue" valueOf="outputParamHandle"/>
            </term>
          </realisation>
        </action>
        <action id="id_getResponseParameterValue">
          <realisation xsi:type="Assignment">
            <result xsi:type="IntegerVariable" name="diagServiceResultValue"/>
            <term xsi:type="quant:GetDisplayValue">
              <quant:quantity xsi:type="quant:QuantityValue" valueOf="diagServiceResultQuantity"/>
            </term>
          </realisation>
        </action>
        <action id="id_getResponseParameterUnit">
          <realisation xsi:type="Assignment">
            <result xsi:type="quant:UnitVariable" name="diagServiceResultUnit"/>
            <term xsi:type="quant:GetDisplayUnit">
              <quant:quantity xsi:type="quant:QuantityValue" valueOf="diagServiceResultQuantity"/>
            </term>
          </realisation>
        </action>
        <action id="id_closeComChannel">
          <realisation xsi:type="diag:CloseComChannel">
            <diag:comChannel xsi:type="diag:ComChannelVariable" name="comChannelHandle"/>
          </realisation>
        </action>
      </flow>
    </realisation>
  </procedure>
 </procedures>
</otx>
```

Annex F
(informative)

OTX DiagComRaw extension for resource-restrained systems

F.1 Introduction

IMPORTANT — The OTX DiagComRaw extension is a recommendation only; it is not a part of the standard set of OTX extensions. If an extension is needed which supports the use case described below, it should not be developed as a fully proprietary extension – instead, the OTX DiagComRaw recommendation described in the following should be followed.

The purpose of the OTX DiagComRaw extension is to provide the necessary OTX elements for performing diagnostic vehicle communication as a self-sufficient script. In contrast, the OTX DiagCom extension (cf. 6) is working on a symbolic level, only using identifier references to indicate the ECU or diagnostic service to use. This means that to execute an OTX sequence that is using elements from the DiagCom extension for vehicle communication it is always required to have a runtime component that is able to translate the symbolic level of the DiagCom extension to the bits&bytes level of actual ECU communication. Such functionality is e.g. provided by an MVCI [ISO 22900] runtime system component. The DiagComRaw extension addresses the use case where there is no such software component available in a runtime environment, requiring the OTX sequence to comprise any information that it necessary for performing diagnostic vehicle communication. To this end, DiagComRaw builds on DiagCom, extending specific DiagCom elements to contain additional information required for actual diagnostic vehicle communication. The added information is provided in the form of snippets of ODX [ISO 22901] diagnostic data descriptions.

Specifically, the following diagnostic use case has been considered: Creating diagnostic sequences that can run stand-alone e.g. on a target environment with limited resources.

NOTE 1 It is an explicit design goal of the DiagComRaw extension to reuse existing standards and definitions where possible. For this reason this extension only integrates data model definitions from both the OTX and ODX standard where possible. This way it becomes a trivial task e.g. to create a software converter that accepts an OTX DiagCom-based sequence and corresponding ODX data definitions as input and generates an all-in-one standalone OTX DiagComRaw sequence.

NOTE 2 Any of the definitions in this clause are derived from (and are aligned with) the ECU Variant Identification definitions from the ODX ISO standard. When in doubt, the ODX standard should be referred to for detailed information on how ECU Variant Identification should be performed and how the relevant ODX data is to be used.

The OTX DiagComRaw extension provides a new action and a new term which are both based on features specified by the OTX DiagCom extension: the **`ExecuteDiagService`** action and the **`GetComChannel`** term. These elements comprise additional information that is necessary to make an OTX sequence executable without requiring any added external content. Details on these additional information containers are provided in the following sections.

F.2 Using DiagComRaw

This clause explains how to use the OTX DiagComRaw extension for creating ECU variant-aware diagnostic sequences and how to use this OTX sequence at runtime.

F.2.1 Creating OTX data for variant identification

When an OTX sequence containing DiagComRaw elements is created, it needs to contain information from the ODX data set that is referenced by the **`ExecuteDiagService`** action nodes in the sequence. Two kinds of additional information are required: data to enable the DiagComRaw OTX sequence to identify the variants of ECUs present at runtime, as well as any ODX definitions that are necessary to correctly interpret the request and response parameters of diagnostic services according to the ECU variant present in the vehicle.

This clause describes how to add the required ODX data for performing ECU variant identification to an OTX sequence.

Any data that is required for performing ECU variant identification in a DiagComRaw OTX sequence needs to be added to the **VariantIdentExpectations** element of any **GetComChannel** action node in the OTX sequence. To this end, the author/tool creating the DiagComRaw OTX sequence has to perform the following steps:

Looking at the ECU **BASE-VARIANT** pointed to by the identifier of the **GetComChannel** term:

a) For each **DIAG-COMM** referenced by any **MATCHING-PARAMETER** within all **ECU-VARIANT-PATTERNs** in the **ECU-VARIANTs** based on that **BASE-VARIANT** （**ECU-VARIANTs** are to be iterated in alphabetical order）

   1) Create an OTX **<variantIdentService>** element containing this **DIAG-COMMs REQUEST** definition

   2) For each **OUT-PARAM-IF** of the **DIAG-COMMs** referenced by the **MATCHING-PARAMETER**

      i) Create an OTX **<RESPONSE>** element underneath the **<variantIdentService>** element contatining the relevant **RESPONSE** definition,

      ii) Create a **<responseAssignment>** element containing the **SHORT-NAME** of the response parameter referenced by the ODX **MATCHING-PARAMETER** and assign a unique id to the **uniqueParameterId** attribute of this **<responseAssignment>**

      iii) Add any **DOPs** or **UNIT** definitions referenced by any of the **REQUESTs** or **RESPONSEs** that were added in the previous steps to the **<DIAG-DATA-DICTIONARY-SPEC>** element in the **<variantIdentService>** container

   3) For each **ECU-VARIANT-PATTERN** in the **ECU-VARIANTs** inheriting from this ECU **BASE-VARIANT** in the ODX data (in the order defined in the ODX data)

      i) Create a **<variantIdentExpectation>** element, setting the **variantName** attribute to the **SHORT-NAME** of the **ECU-VARIANT** that contains the current **ECU-VARIANT-PATTERN**

      ii) For each **MATCHING-PARAMETER** element contained in this **ECU-VARIANT-PATTERN** element (in the order defined in the ODX data),

         I) Create a **<matchingPattern>** element in the current **<variantIdentExpectation>** element, setting its **expectedValue** attribute to the **EXPECTED-VALUE** attribute of the ODX **MATCHING-PARAMETER** and assigning the unique parameter id pointing to the appropriate **<variantIdentService>** element's **<responseAssignment>**, see steps above.

F.2.2 Creating OTX data for diagnostic service execution

When an OTX sequence containing DiagComRaw elements is created, it needs to contain information from the ODX data set that is referenced by the **ExecuteDiagService** action nodes in the sequence. Two kinds of additional information are required: data to enable the DiagComRaw OTX sequence to identify the variants of ECUs present at runtime, as well as any ODX definitions that are necessary to correctly interpret the request and response parameters of diagnostic services according to the ECU variant present in the vehicle. This clause describes how to add the required ODX data for executing ECU variant-aware diagnostic services.

Any data that is required for performing ECU variant-aware diagnostic services needs to be added to the **<odxSourceData>** element of any **ExecuteDiagService** action node in the OTX sequence. To this end, the author/tool creating the DiagComRaw OTX sequence has to perform the following steps:

a) Analyze the ODX inheritance hierarchy of the ECU the diagnostic service referenced by this **ExecuteDiagService** action is defined in,

b) For each **ECU-VARIANT** that has a **NOT-INHERITED** relationship to that **DIAG-COMM**, add a **<nonInheritingVariant>** element containing its **SHORT-NAME**, below **<odxSourceData>**,

c) If available, add the ODX **<REQUEST>** element definition of the **DIAG-COMM** as defined on the ECU **BASE-VARIANT** as an OTX **<REQUEST>** (**VariantEnabledRequest**) element, with the attribute **isBaseVariant** set to **true**

d) For each set of **ECU-VARIANTs** that contain identical versions of the **DIAG-COMMs REQUEST**, add an OTX **<REQUEST>** element, putting an **<includingVariant>** element for each **ECU-VARIANT** in the set, each containing the **SHORT-NAME** of the variant,

e) If available, add the ODX **<RESPONSE>** element definition of the **DIAG-COMM** as defined on the ECU **BASE-VARIANT** as an OTX **<RESPONSE>** (**VariantEnabledResponse**) element, with the **isBaseVariant** attribute set to **true**

f) For each set of **ECU-VARIANTs** that contain identical versions of the **DIAG-COMMs RESPONSE**, add an OTX **<RESPONSE>** element, putting an **<includingVariant>** element for each **ECU-VARIANT** in the set, each containing the **SHORT-NAME** of the variant, setting **isPositive** to **true** if the response in question is a positive response, **false** if otherwise

g) Add any **DOPs** or **UNIT** definitions referenced by any of the **REQUESTs** or **RESPONSEs** that were added in the previous steps to the **DIAG-DATA-DICTIONARY-SPEC** element in the **<odxSourceData>** container

F.2.3 Performing variant identification

To enable ECU variant-aware execution of diagnostic services in a DiagComRaw OTX sequence, the variant of an ECU present at runtime has to be identified before diagnostic ECU communication is attempted. ECU variant identification could either be performed by the DiagComRaw enabled runtime system when the **GetComChannel** term is executed, alternatively it might be delayed until the executing of the first OTX node which requires ECU variant information (e.g. **diag:ExecuteDiagService**, **diag:IsVariant**, **diag:GetComChannelEcuVariantName**).

NOTE When performing the following steps, diagnostic services only have to be executed once. This means that a runtime system is allowed to cache the results of previous variant identification-relevant serivces between evaluation of different **VariantPatterns** and **MatchingPatterns**.

To perform ECU variant identification using a DiagComRaw sequence, the following steps shall be performed:

Looking at the **<variantIdentExpectations>** element of the relevant **GetComChannel** action:

a) For each **<variantIdentExpectation>** element,

   1) For each **<variantPattern>** element,

      i) For each **<matchingPattern>** element,

         I) Execute the diagnostic service containing the response parameter referenced by the **uniqueParameterId** attribute of the **<matchingPattern>**

         II) Compare the runtime value of that response parameter to the **expectedValue** attribute of the **<matchingPattern>** element; if they are equal, this **MatchingPattern** is considered to match

      ii) In case each **MatchingPattern** matches

   2) In case each **VariantPattern** matches

b) The current **VariantIdentExpectation** is considered to match the ECU present in the vehicle (the name of the matching ECU variant can be retrieved from the **variantName** attribute of this **<variantIdentExpectation>** element, abort the variant identification operation

F.2.4 Executing variant-aware diagnostic services

After the variant of an ECU that is present in a vehicle at runtime has been identified, a DiagComRaw OTX sequence can perform variant-aware diagnostic communication with that ECU. That means that the ODX sequence can use the correct **REQUEST** and **RESPONSE** definitions of a diagnostic service depending on the variant of an ECU present in the vehicle.

To perform variant-aware diagnostic communication, the following steps have to be implemented when executing an **ExecuteDiagService** action:

a) Retrieve the name of the ECU variant identified for the **ComChannel** the **ExecuteDiagService** action is pointing to,

b) If the ECU variant name is contained in the list of **<nonInheritingVariant>** elements in the **<odxSourceData>** element associated with the **ExecuteDiagService** action, abort diagnostic service execution

c) Otherwise, iterate through the list of OTX **<REQUEST>** definitions associated with the **<odxSourceData>** element,

   1) If the current ECU variant name matches an **<includingVariant>** element of an OTX **<REQUEST>** element, execute this request definition (use this request definition for converting any request parameters from symbolic OTX level into bits&bytes level)

   2) If no match with an **<includingVariant>** element was found, execute the OTX **<REQUEST>** element where **isBaseVariant** is set to **true** (use this request definition for converting any request parameters from symbolic OTX level into bits&bytes level)

   3) If no OTX **<REQUEST>** element exists where **isBaseVariant** is set to **true**, abort diagnostic service execution

d) For interpreting the response of the executed diagnostic service, iterate through the list of OTX **<RESPONSE>** definitions associated with the **<odxSourceData>** element,

   1) If the current ECU variant name matches an **<includingVariant>** element of an OTX **<RESPONSE>**, use this response definition for converting any mapped response parameters from bits&bytes level into symbolic OTX level (the evaluation of the actual ECU response has to be implemented according to the relevant definitions of the ISO ODX standard, e.g. for determining whether a response definition matches the ECU response in the first place or whether the ECU returned a response that is not covered by any ODX data set)

   2) If no match with an **<includingVariant>** element was found, execute the OTX **<RESPONSE>** element where **isBaseVariant** is set to **true** and use this response definition for converting any mapped response parameters from bits&bytes level into symbolic OTX level

   3) If no OTX **<RESPONSE>** element exists where **isBaseVariant** is set to **true**, abort diagnostic service response conversion and raise an error (**diag:UnknownResponseException**)

F.3 Actions

F.3.1 Overview

The OTX DiagComRaw extension does not invent anything new; rather it is an extension to existing OTX DiagCom extension elements, adding information from ODX where necessary to enable a DiagComRaw-based OTX script to be executed without requiring any additional external systems (like an MVCI server).

Generally, there are two kinds of information required to be contained in a DiagComRaw script to make it executable: any **`ExecuteDiagService`** elements needs the required information to translate the symbolic service- and parameter names to the bits&bytes level required by protocol handlers, plus data needs to be included that allows a DiagComRaw based standalone runtime system to perform ECU variant identification on a communication channel (and subsequently execute diagnostic services as applicable for that channel).

NOTE In the OTX DiagComRaw data model (in the diagrams provided in this clause) referenced ODX elements are assumed to have a XML namespace ("odx"). The actual ODX data model as defined by the ODX ISO standard does not declare a target namespace; however the representation here has been chosen to enhance readability and make it more explicit where elements are referenced from the ODX data model. It is advised that for implementation purposes, the ODX data model should be enhanced to declare its own namespace to enable the usual XML schema mechanisms for validating XML instance documents or for flexibly interlocking other XML schemas (like the OTX schema) with the ODX schema by using the **`xsd:import`** mechanism.

F.3.2 Syntax

Figure F.1 shows the syntax of the DiagComRaw **`ExecuteDiagService`** action with its associated ODX-derived data container, as well as the **`ExecuteHexDiagService`** action.

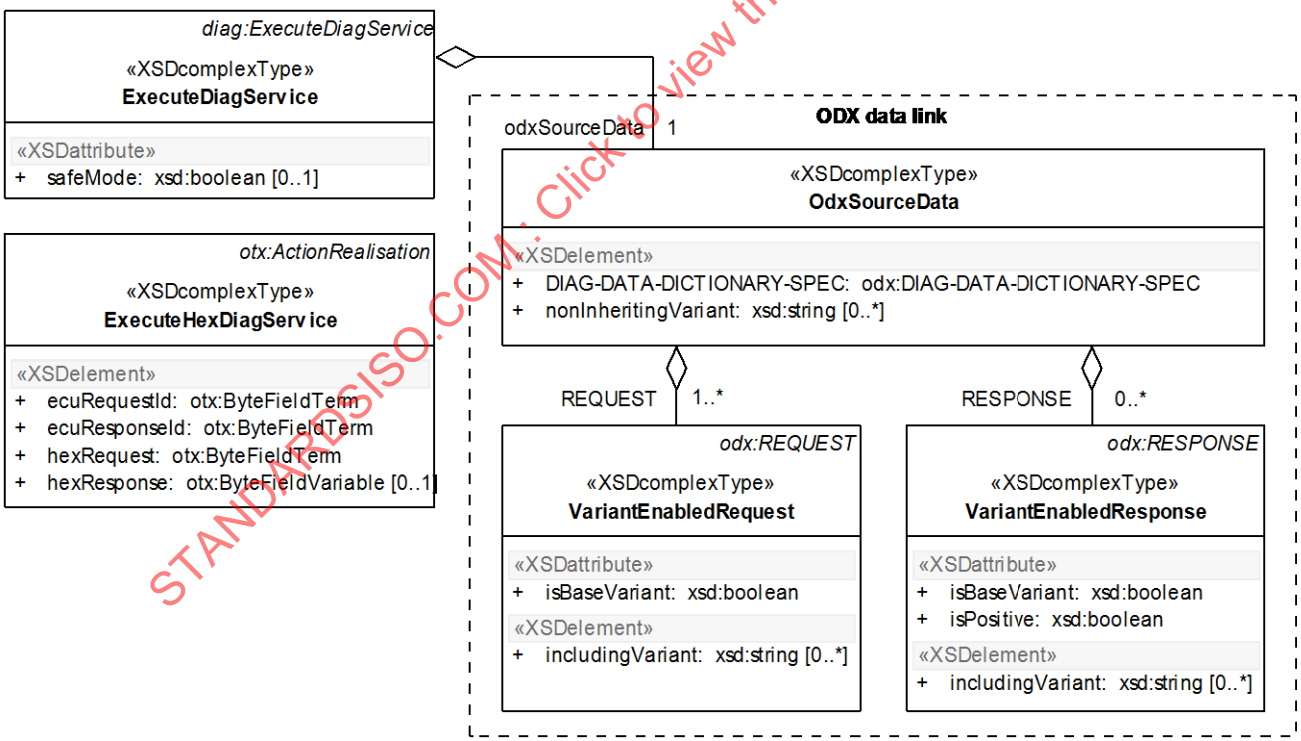

Figure F.1 — Data model view: DiagComRaw actions