



BSI Standards Publication

Road vehicles — Open Test sequence eXchange format (OTX)

Part 3: Standard extensions and requirements

National foreword

This British Standard is the UK implementation of ISO 13209-3:2022. It supersedes BS ISO 13209-3:2012, which is withdrawn.

The UK participation in its preparation was entrusted to Technical Committee AUE/16, Data Communication (Road Vehicles).

A list of organizations represented on this committee can be obtained on request to its committee manager.

Contractual and legal considerations

This publication has been prepared in good faith, however no representation, warranty, assurance or undertaking (express or implied) is or will be made, and no responsibility or liability is or will be accepted by BSI in relation to the adequacy, accuracy, completeness or reasonableness of this publication. All and any such responsibility and liability is expressly disclaimed to the full extent permitted by the law.

This publication is provided as is, and is to be used at the recipient's own risk.

The recipient is advised to consider seeking professional guidance with respect to its use of this publication.

This publication is not intended to constitute a contract. Users are responsible for its correct application.

© The British Standards Institution 2022
Published by BSI Standards Limited 2022

ISBN 978 0 539 02713 6

ICS 43.040.15; 43.180

Compliance with a British Standard cannot confer immunity from legal obligations.

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 31 August 2022.

Amendments/corrigenda issued since publication

Date	Text affected
------	---------------

**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





COPYRIGHT PROTECTED DOCUMENT

© ISO 2022

All rights reserved. Unless otherwise specified, or required in the context of its implementation, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

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

Published in Switzerland

Contents

Page

Foreword.....	viii
Introduction.....	ix
1 Scope.....	1
2 Normative references.....	1
3 Terms, definitions and abbreviated terms.....	1
3.1 Terms and definitions.....	1
3.2 Abbreviated terms.....	2
4 Requirements and recommendations.....	3
4.1 Basic principles for requirements and recommendations definition.....	3
4.2 Entries priorities.....	3
4.3 Requirement listing.....	3
5 Extension overview.....	6
5.1 General.....	6
5.2 Dependencies.....	6
5.3 Basic characteristics of the OTX extensions.....	8
6 OTX DateTime extension.....	9
6.1 General.....	9
6.2 Terms.....	9
6.2.1 Overview.....	9
6.2.2 Syntax.....	9
6.2.3 Semantics.....	10
7 OTX DiagCom extension.....	12
7.1 General.....	12
7.2 General considerations.....	13
7.2.1 Communication channels.....	13
7.2.2 Diagnostic services.....	13
7.2.3 Diagnostic communication patterns.....	15
7.2.4 Special-purpose diagnostic data types.....	19
7.3 Data types.....	20
7.3.1 Overview.....	20
7.3.2 Syntax.....	20
7.3.3 Semantics.....	20
7.4 Exceptions.....	23
7.4.1 Overview.....	23
7.4.2 Syntax.....	23
7.4.3 Semantics.....	24
7.5 Variable access.....	25
7.5.1 Overview.....	25
7.5.2 Syntax.....	25
7.5.3 Semantics.....	26
7.6 Actions.....	26
7.6.1 Overview.....	26
7.6.2 ComChannel related actions.....	26
7.6.3 ComParameter related actions.....	28
7.6.4 DiagService related actions.....	29
7.7 Terms.....	40
7.7.1 Overview.....	40
7.7.2 ComChannel related terms.....	41
7.7.3 DiagService related terms.....	45
7.7.4 Request related terms.....	49
7.7.5 Result related terms.....	50
7.7.6 Response related terms.....	50

	7.7.7	Parameter related terms.....	55
	7.7.8	ComParam related terms.....	61
	7.7.9	Event related terms	64
8		OTX DiagDataBrowsing extension.....	65
	8.1	General.....	65
	8.2	Data types.....	66
	8.2.1	Overview	66
	8.2.2	Syntax.....	66
	8.2.3	Semantics.....	66
	8.3	Variable access.....	67
	8.3.1	Overview	67
	8.3.2	Syntax.....	67
	8.3.3	Semantics.....	67
	8.4	Terms.....	68
	8.4.1	Overview	68
	8.4.2	Syntax.....	68
	8.4.3	Semantics.....	68
9		OTX EventHandling extension.....	72
	9.1	General.....	72
	9.2	Data types.....	72
	9.2.1	Overview	72
	9.2.2	Syntax.....	73
	9.2.3	Semantics.....	73
	9.3	Variable access.....	74
	9.3.1	Overview	74
	9.3.2	Syntax.....	74
	9.3.3	Semantics.....	74
	9.4	Actions.....	74
	9.4.1	Overview	74
	9.4.2	Syntax.....	74
	9.4.3	Semantics.....	74
	9.4.4	Example.....	76
	9.5	Terms.....	76
	9.5.1	Overview	76
	9.5.2	Event terms	77
	9.5.3	Event source terms.....	78
	9.5.4	Event property terms.....	81
	9.5.5	Exception terms.....	83
10		OTX Flash extension.....	84
	10.1	General.....	84
	10.2	Data types.....	85
	10.2.1	Overview	85
	10.2.2	Syntax.....	85
	10.2.3	Semantics.....	86
	10.3	Exceptions.....	88
	10.3.1	Overview	88
	10.3.2	Syntax.....	88
	10.3.3	Semantics.....	88
	10.4	Variable access.....	88
	10.4.1	Overview	88
	10.4.2	Syntax.....	89
	10.4.3	Semantics.....	89
	10.5	Actions.....	89
	10.5.1	Overview	89
	10.5.2	Syntax.....	89
	10.5.3	Semantics.....	89
	10.5.4	Example.....	89

10.6	Terms.....	92
10.6.1	Overview	92
10.6.2	Flash job related terms.....	93
10.6.3	Flash session related terms.....	95
10.6.4	Flash block related terms.....	99
10.6.5	Flash block segment related terms.....	104
10.6.6	Security related terms	106
10.6.7	Own ident related terms.....	109
10.6.8	Enumeration related terms	110
11	OTX HMI extension.....	112
11.1	General.....	112
11.1.1	General considerations.....	112
11.1.2	Dialogs.....	113
11.1.3	Custom screens.....	113
11.1.4	Custom screen usage example	114
11.2	Data types.....	115
11.2.1	Overview	115
11.2.2	Syntax.....	115
11.2.3	Semantics.....	115
11.3	Exceptions.....	117
11.3.1	Overview	117
11.3.2	Syntax.....	117
11.3.3	Semantics.....	117
11.4	Variable access.....	118
11.4.1	Overview	118
11.4.2	Syntax.....	118
11.4.3	Semantics.....	118
11.5	Actions.....	118
11.5.1	Overview	118
11.5.2	Dialog related actions.....	119
11.5.3	Custom screen related actions.....	125
11.6	Terms.....	129
11.6.1	Overview	129
11.6.2	Syntax.....	130
11.6.3	Semantics.....	131
11.7	Signatures.....	134
11.7.1	Overview	134
11.7.2	Syntax.....	134
11.7.3	Semantics.....	134
12	OTX i18n extension.....	136
12.1	General.....	136
12.2	Data types.....	136
12.2.1	Overview	136
12.2.2	Syntax.....	136
12.2.3	Semantics.....	137
12.3	Exceptions.....	137
12.3.1	Overview	137
12.3.2	Syntax.....	137
12.3.3	Semantics.....	138
12.4	Variable access.....	138
12.4.1	Overview	138
12.4.2	Syntax.....	138
12.4.3	Semantics.....	139
12.5	Terms.....	139
12.5.1	Overview	139
12.5.2	Locale settings related terms	140
12.5.3	Translation related terms.....	141

	12.5.4 Quantity related terms	145
13	OTX Logging extension	147
13.1	General	147
13.2	Data types	148
13.2.1	Overview	148
13.2.2	Syntax	148
13.2.3	Semantics	148
13.3	Variable access	149
13.3.1	Overview	149
13.3.2	Syntax	149
13.3.3	Semantics	149
13.4	Actions	150
13.4.1	Overview	150
13.4.2	Syntax	150
13.4.3	Semantics	150
13.4.4	Example	151
13.5	Terms	152
13.5.1	Overview	152
13.5.2	Syntax	152
13.5.3	Semantics	152
14	OTX Math extension	153
14.1	General	153
14.2	Terms	154
14.2.1	Overview	154
14.2.2	Syntax	154
14.2.3	Semantics	154
15	OTX Measure extension	156
15.1	General	156
15.2	Data types	157
15.2.1	Overview	157
15.2.2	Syntax	157
15.2.3	Semantics	157
15.3	Exceptions	157
15.3.1	Overview	157
15.3.2	Syntax	157
15.3.3	Semantics	158
15.4	Variable access	159
15.4.1	Overview	159
15.4.2	Syntax	159
15.4.3	Semantics	159
15.5	Signatures	159
15.5.1	Overview	159
15.5.2	Syntax	159
15.5.3	Semantics	160
15.6	Actions	161
15.6.1	Overview	161
15.6.2	Syntax	161
15.6.3	Semantics	162
15.7	Terms	164
15.7.1	Overview	164
15.7.2	Measurement related terms	165
15.7.3	Event related terms	168
16	OTX quantities extension	169
16.1	General	169
16.2	Data types	172
16.2.1	Overview	172

16.2.2	Syntax.....	172
16.2.3	Semantics.....	172
16.3	Exceptions.....	173
16.3.1	Overview.....	173
16.3.2	Syntax.....	173
16.3.3	Semantics.....	174
16.4	Variable access.....	174
16.4.1	Overview.....	174
16.4.2	Syntax.....	174
16.4.3	Semantics.....	175
16.5	Terms.....	175
16.5.1	Overview.....	175
16.5.2	Quantity and unit related terms.....	176
16.5.3	Overloading semantics.....	180
17	OTX StringUtil extension	182
17.1	General.....	182
17.2	Data types.....	182
17.2.1	Overview.....	182
17.2.2	Syntax.....	182
17.2.3	Semantics.....	182
17.3	Exceptions.....	183
17.3.1	Overview.....	183
17.3.2	Syntax.....	183
17.3.3	Semantics.....	184
17.4	Variable access.....	184
17.4.1	Overview.....	184
17.4.2	Syntax.....	184
17.4.3	Semantics.....	185
17.5	Terms.....	185
17.5.1	Overview.....	185
17.5.2	Syntax.....	185
17.5.3	Semantics.....	186
	Annex A (normative) Comprehensive checker rule listing.....	193
	Annex B (normative) OTX DiagCom extension data type mappings	197
	Annex C (normative) OTX DiagMetaData auxiliary for the OTX DiagCom extension	201
	Annex D (informative) OTX DiagComRaw extension for resource-restrained systems	206
	Annex E (informative) OTX job extension.....	217
	Bibliography.....	228

Foreword

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

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

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

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

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

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

This second edition cancels and replaces the first edition (ISO 13209-3:2012), which has been technically revised.

The main changes are as follows:

- DiagMetaData: introduced `ComChannelGroup` and `EcuVariantGroup`;
- EventHandling: introduced `CompositeEventSource`, `GetEventSourceFromEvent`, `IsEventHasException`;
- DiagCom: introduced `textIdTarget`, `GetParameterValueTextId`;
- added new checker rules.

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

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

Introduction

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

Today, the automotive industry mainly relies on paper documentation and/or proprietary authoring 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.

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

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

ISO 639-1, *Codes for the representation of names of languages — Part 1: Alpha-2 code*

ISO 3166-1, *Codes for the representation of names of countries and their subdivisions — Part 1: Country code*

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

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

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

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

3 Terms, definitions and abbreviated terms

3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO 13209-1, ISO 13209-2 and the following apply.

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

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

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

3.1.1

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 (3.1.2) 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 and recommendations

4.1 Basic principles for requirements and recommendations definition

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

- a) OTX requirements or recommendations 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 document.

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

4.2 Entries priorities

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

— SHALL:

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

— SHOULD:

If the recommendations-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: A 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: A 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 a functionality to store in a selected format.

Extensions_R06 – Enable developer to use OTX in place of ODX Java jobs

Priority: SHALL

Rationale: A 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: A 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 to establish a communication channel to a particular ECU, to request parameters of a diagnostic service which is sent to the ECU and to analyse 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: A 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, 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: A 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). There shall be a way to get further information about an event, for example, 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: A 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: A 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, for example, the transformation of a physical value from one unit-system to another (e.g. representing a distance by kilometres or miles). 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 (e.g. trigonometric and logarithmic functions).

5 Extension overview

5.1 General

This document represents the specification of the OTX standard extensions for data model version "1.0.0". <https://standards.iso.org/iso/13209/-3/ed-2/en/> includes code on the OTX extensions.

[Annex A](#) contains a comprehensive listing of all checker rules which shall be followed. The rules are needed because some constraints existing on OTX documents cannot be ensured by XSD validation alone. These constraints need to be checked by additional checker applications.

5.2 Dependencies

[Figure 1](#) shows a UML package diagram^[5] 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 ISO 13209-2. 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.

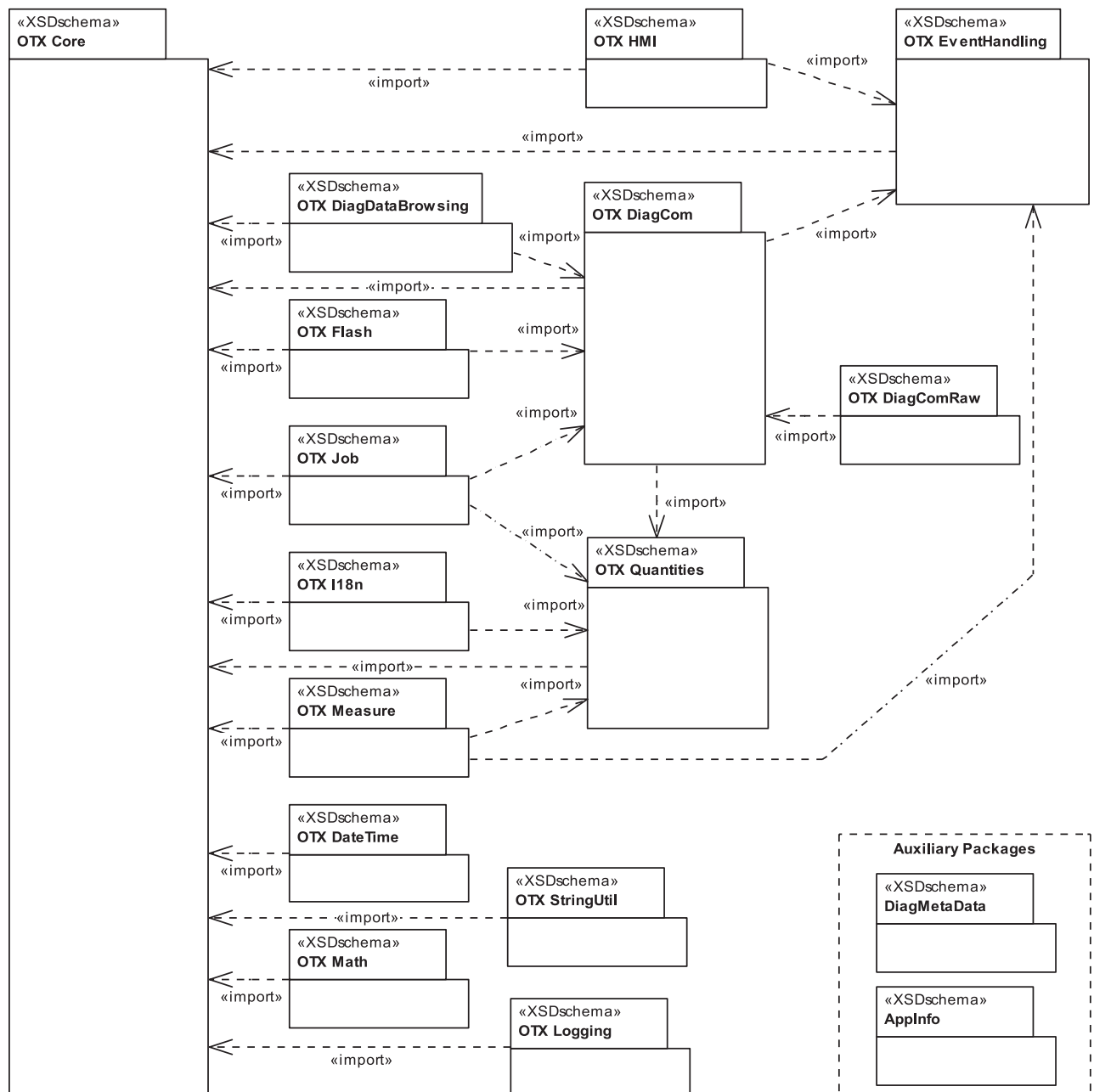


Figure 1 — Overview: OTX schema dependencies

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 document. 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 not 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](#), which shall be followed. For the AppInfo auxiliary, please refer to ISO 13209-2.

[Annex E](#) describes how Java jobs of the MVCI system can be used in OTX test sequences.

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_DiagDataBrowsing.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, analysing 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 based on References [11] and [12]. 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:

Table 2 (continued)

Extension	Namespace	Prefix
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 General

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.

<i>otx:IntegerTerm</i> «XSDcomplexType» GetTimestamp	<i>otx:StringTerm</i> «XSDcomplexType» FormatDate «XSDelement» + timestamp: otx:NumericTerm + format: otx:StringTerm [0..1]	<i>otx:StringTerm</i> «XSDcomplexType» FormatDuration «XSDelement» + duration: otx:NumericTerm + format: otx:StringTerm [0..1]
--	--	---

Figure 3 — Data model view: DateTime 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™¹⁾ 2 Platform Standard Ed. 6.

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

6.2.3.2 FormatDate

Results from **FormatDate** should be used for user representations only. These values should not be used inside the test logic. The reason for this is that the exchangeability is not guaranteed across different run time systems (e.g. era, time zone).

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

- In case there is no custom format specified, the returned string shall be formatted according to the rules given by ISO 8601.
- If a custom format is given (by the `<format>` element), the string shall be formatted according to the custom 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. Non-numeric outputs (e.g. the name of a month) shall be translated automatically to the currently set locale (see [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

1) Java is an example of a suitable product available commercially. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO of this product.

Table 3 (continued)

Specifier(s)	Meaning	Presentation	Example
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, 04, 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 `SimplifyDateFormat.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 the 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.

— `<format> : 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`

It is thrown if the timestamp value is negative or the pattern format is wrong.

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), only the specifiers defined in [Table 4](#) should be used.

Accordingly, the values expressed shall not exceed the "carry-over-points" of 12 months, 30 days, 24 hours, 60 min and 60 s.

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, 0-29 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, 03, 002

EXAMPLE 1 For a given duration of 203 443 ms (this is 3 min, 23 s and 443 ms), a pattern like, e.g. "This took about 'm' minutes and 's' seconds." would produce the following formatted output: "This took about 3 min and 23 s."

If no custom format is given, the ISO 8601 conform date output shall be formatted equivalent to the custom pattern "'P'y-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 203 443 ms (this is 3 min, 23 s and 443 ms), the following standard-format output will be produced: "P0-00-00T00:03:23.443".

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.

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

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

Throws:

— `otx:OutOfBoundsException`

It is thrown if the duration value is negative or the pattern format is wrong.

7 OTX DiagCom extension

7.1 General

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.

IMPORTANT — 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 (see the ISO 22901^[8] and ISO 22900^[7] series) 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.

IMPORTANT — 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.

In contrast to working at a symbolic level, the DiagComRaw extension (see [Annex D](#)) can be used to work at a raw data (binary) level for diagnostic communication.

NOTE An additional functionality is specified in the DiagComPlus extension.

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 the OTX this instance is called a **ComChannel1**, designating a logical connection between the test sequence and the intended communication target. A **ComChannel1** 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 **ComChannel1** 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 **ComChannel1**, and the capability of creating channels to point to specific ECU variants or retrieve the currently active ECU variant name of a **ComChannel1**.

NOTE It is an explicit design goal of the OTX DiagCom extension to be useable with any diagnostic communication kernel. However, the concepts of **ComChannels** 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.

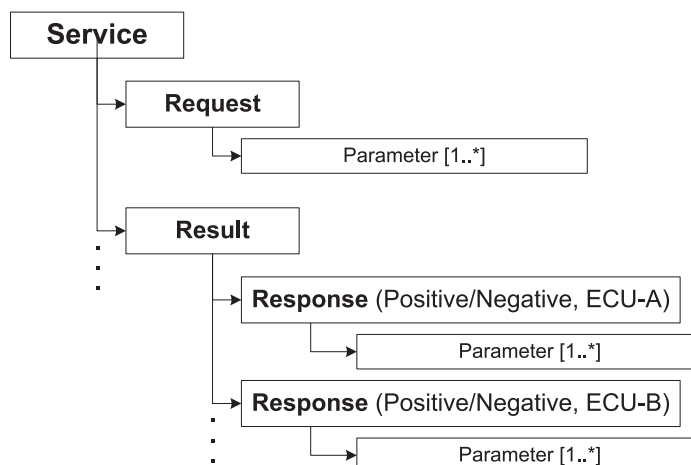


Figure 4 — Diagnostic service request - result structure

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 `<stepByName>` and `<stepByIndex>` method defined by the `<path>` element (please refer to the remainder of this subclause, as well as ISO 13209-2 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:

- the first result of the queue can be accessed by using the `GetFirstResult` term;
- all results currently in the queue can be retrieved as an OTX list by using the `GetAllResults` term;
- 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.

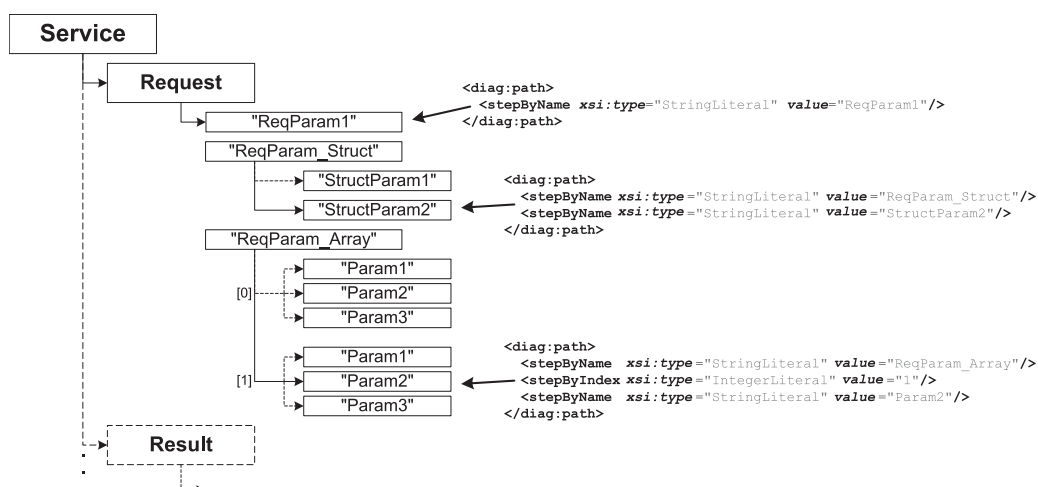


Figure 5 — Complex request structure with `<stepByName>` and `<stepByIndex>`

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.

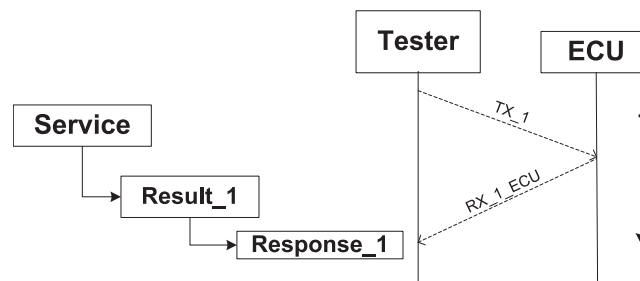


Figure 6 — One-shot service, physical addressing, single-part response

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.

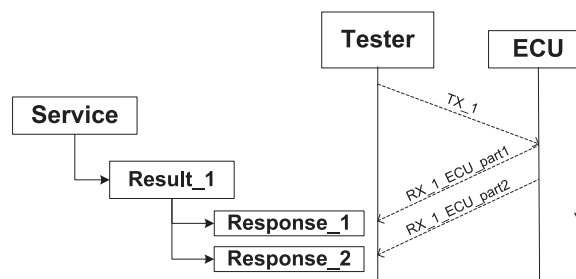


Figure 7 — One-shot service, physical addressing, multi-part responses

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.

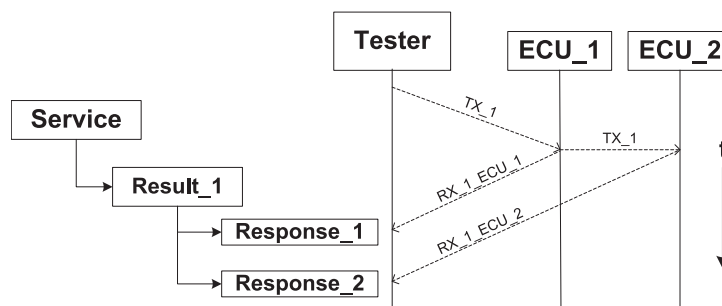


Figure 8 — One-shot service, functional addressing, single-part response

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 (`ComChannel1`) 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.

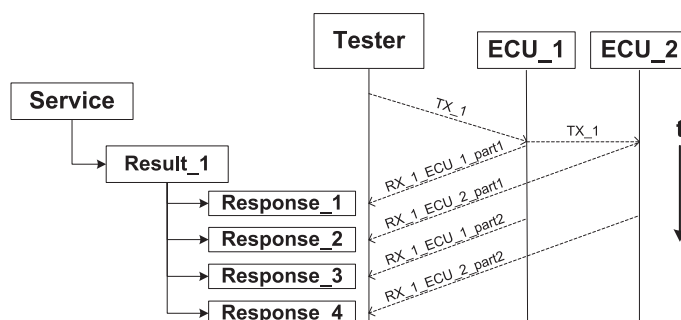


Figure 9 — One-shot service, functional addressing, multi-part responses

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.

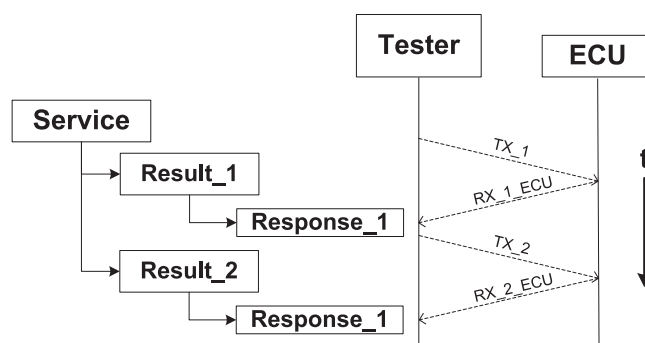


Figure 10 — Repeated service, physical addressing, single-part response

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.

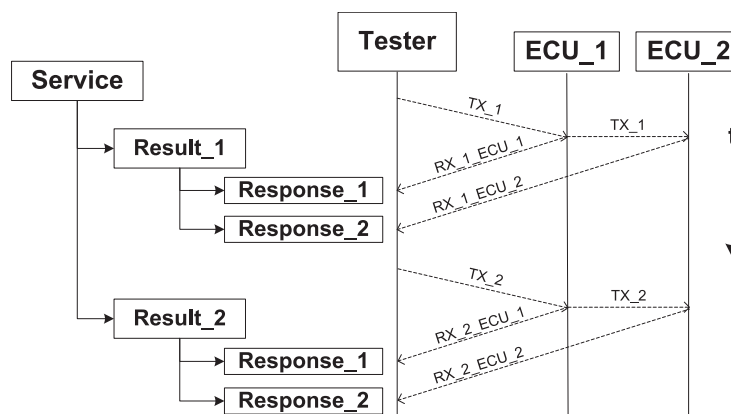


Figure 11 — Repeated service, functional addressing, single-part response

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.

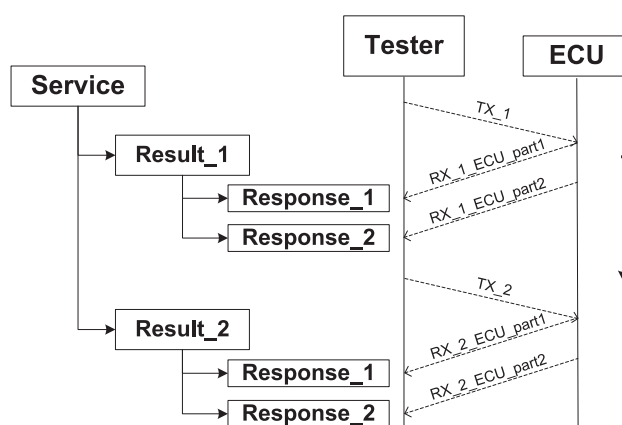


Figure 12 — Repeated service, physical addressing, multi-part responses

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.

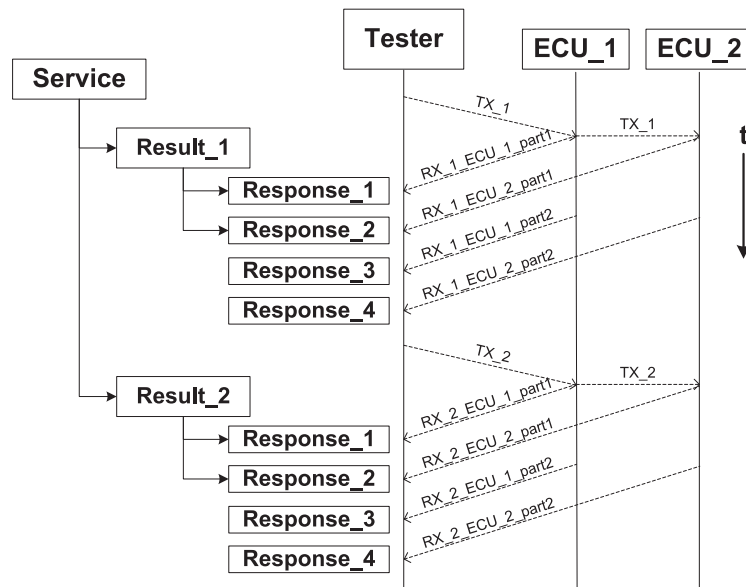


Figure 13 — Repeated service, functional addressing, multi-part responses

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 its sending of cyclic responses has to manually select and execute the appropriate diagnostic service for telling the ECU to stop.

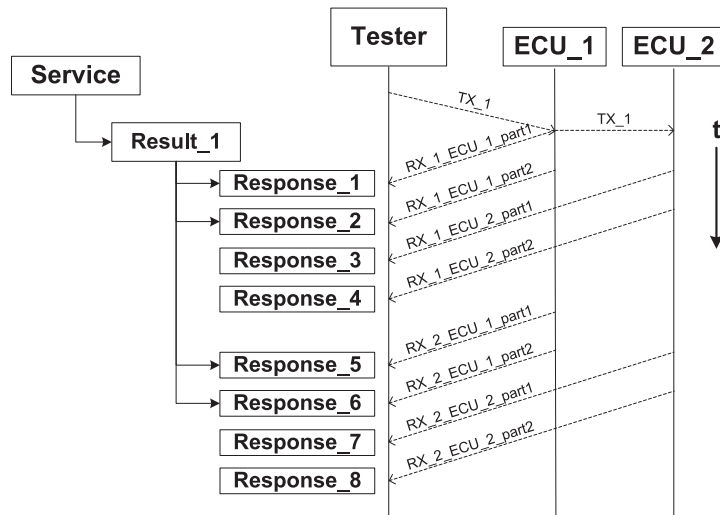


Figure 14 — One-shot service, functional addressing, cyclical multi-part responses

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 ubiquitous diagnostic datatypes like DTCs or freeze frame data. Looking at a DTC, it is a structured data type, with a set of structure parameters defined by SAE J1979^[9] and 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 accessed 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.

Sample of DTC

```
<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 ISO 13209-2. 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](#).

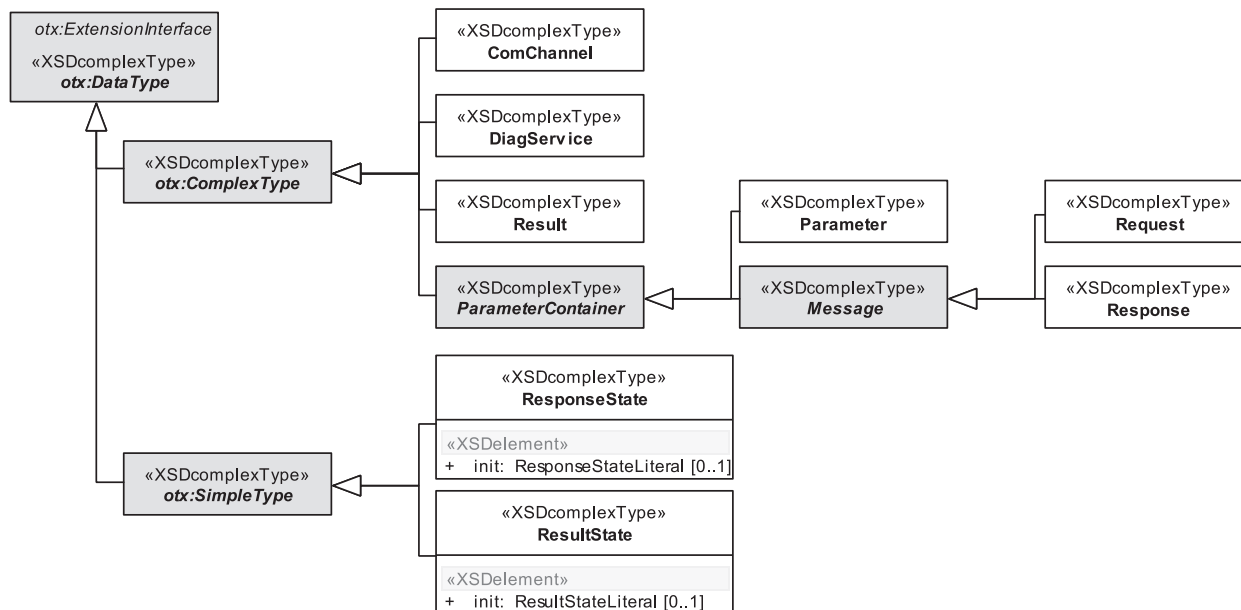


Figure 15 — Data model view: DiagCom data types

7.3.3 Semantics

7.3.3.1 General

The OTX DiagCom data types have no initialization parts (except for the enumeration types **ResponseState** and **ResultState**); therefore, these cannot 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 an 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 an 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 an MVCI/ODX based system, a **Result** handle represents an **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 an MVCI/ODX based system, a **Parameter** handle represents an **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 an MVCI/ODX based system, a **Response** handle represents an **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 an MVCI/ODX based system, a **Request** handle represents an **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.

The mapping from an MVCIServer is shown in [Table 5](#) — Relation between OTXResultState and MCDEXecutionState.

Table 5 — Relation between OTXResultState and MCDEXecutionState

OTX ResultState	MCDEXecutionState
ALL_FAILED	eALL_FAILED
ALL_INVALID	eALL_INVALID_RESPONSE
ALL_NEGATIVE	eALL_NEGATIVE
ALL_POSITIVE	eALL_POSITIVE
FAILED	eCANCELED_DURING_EXECUTION eCANCELED_FROM_QUEUE eFAILED
INVALID	eINVALID_RESPONSE
NEGATIVE	eNEGATIVE
POSITIVE	-----

Please note that the value **Positive** will never occur.

IMPORTANT — **ResultState** values may occur as operands of comparisons (see ISO 13209-2, 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 (see ISO 13209-2).

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

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

This optional element stands for the hard-coded initialization 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 (see ISO 13209-2, 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 (see ISO 13209-2).

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

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

This optional element stands for the hard-coded initialization 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 subclause are derived from the OTX core **Exception** type as defined by ISO 13209-2. 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](#).

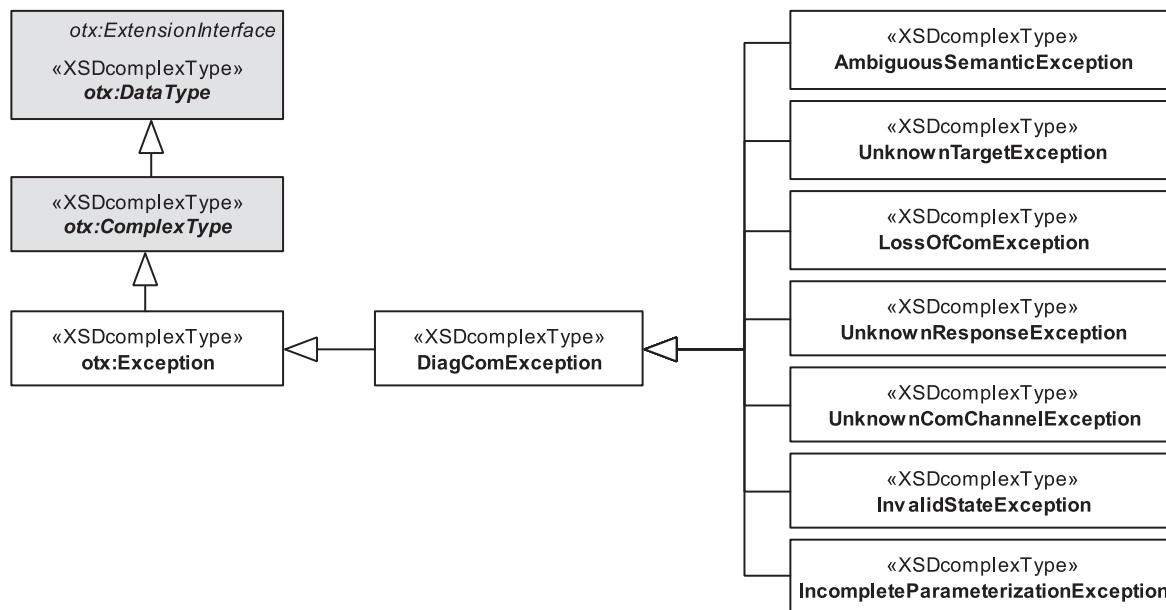


Figure 16 — Data model view: DiagCom exceptions

7.4.3 Semantics

7.4.3.1 General

Since all OTX DiagCom exception types are implicit exceptions without initialization parts, they cannot 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 subclause do not apply to the problem at hand.

IMPORTANT — All terms and action realisations in this extension may potentially throw this exception.

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](#)). If no **<responseParameters>** are defined, no **UnknownResponseException** will be thrown.

7.4.3.7 UnknownComChannelException

This exception is thrown in case a **Response** handle cannot 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 is 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 did not have a default value.

7.5 Variable access

7.5.1 Overview

As specified in ISO 13209-2, 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.

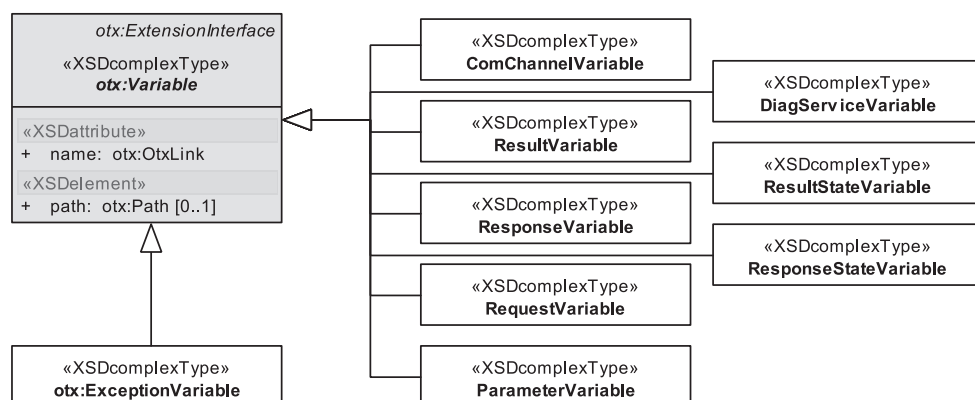


Figure 17 — Data model view: **DiagCom** variable access types

7.5.3 Semantics

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

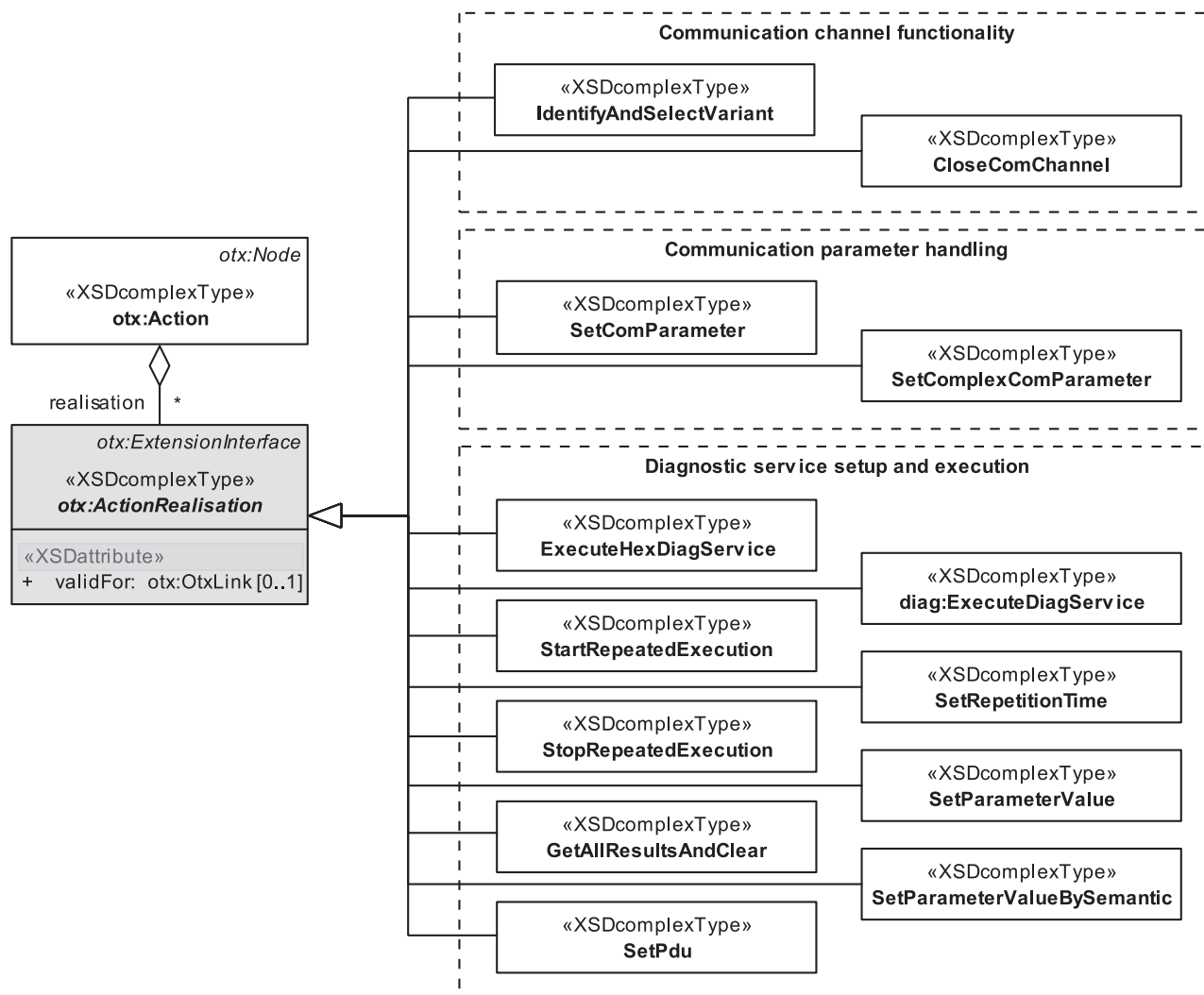


Figure 18 — Data model view: DiagCom actions overview

7.6.2 ComChannel related actions

7.6.2.1 Description

All actions described in this subclause 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.

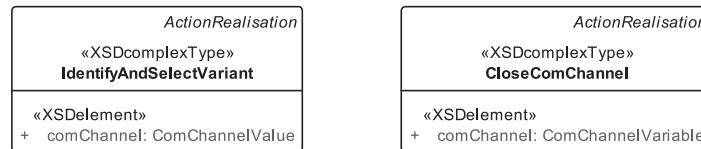


Figure 19 — Data model view: `ComChannel` related actions

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 document cannot 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 can 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**

It is thrown if communication to the ECU was interrupted during the variant identification process.

IMPORTANT — If a variant identification returns without identifying a variant, a **DiagComException** shall be thrown.

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 frees all resources and closes the channel at this point. If a diagnostic sequence uses a `ComChannel` handle after it has been freed by a `closeComChannel` action, the runtime system shall throw an `otx:InvalidReferenceException`.

Closing an uninitialized or already closed `ComChannel` shall perform no operation and report no errors. It shall be for all effects a NOP.

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 subclause 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 parameters handling related `ActionRealisation` types of the `DiagCom` extension.

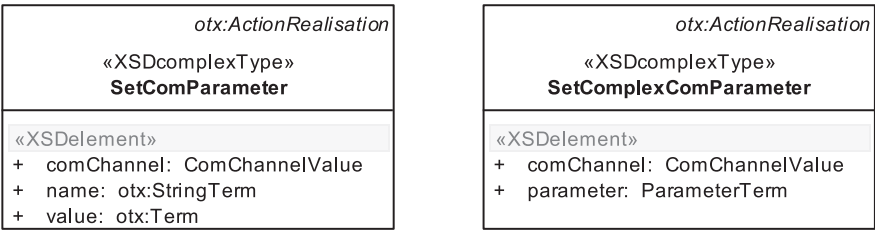


Figure 20 — Data model view: Communication parameter handling

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.

IMPORTANT — In case an ODX/MVCI system is used for vehicle communication, this action should implicitly control the LogicalLink State. The state should be adjusted for setting of COM Parameters. This requires state eONLINE.

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**

It is thrown if no communication parameter with the specified name exists.

- **otx:TypeMismatchException**

It is thrown 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**

It is thrown 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 subclause 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.

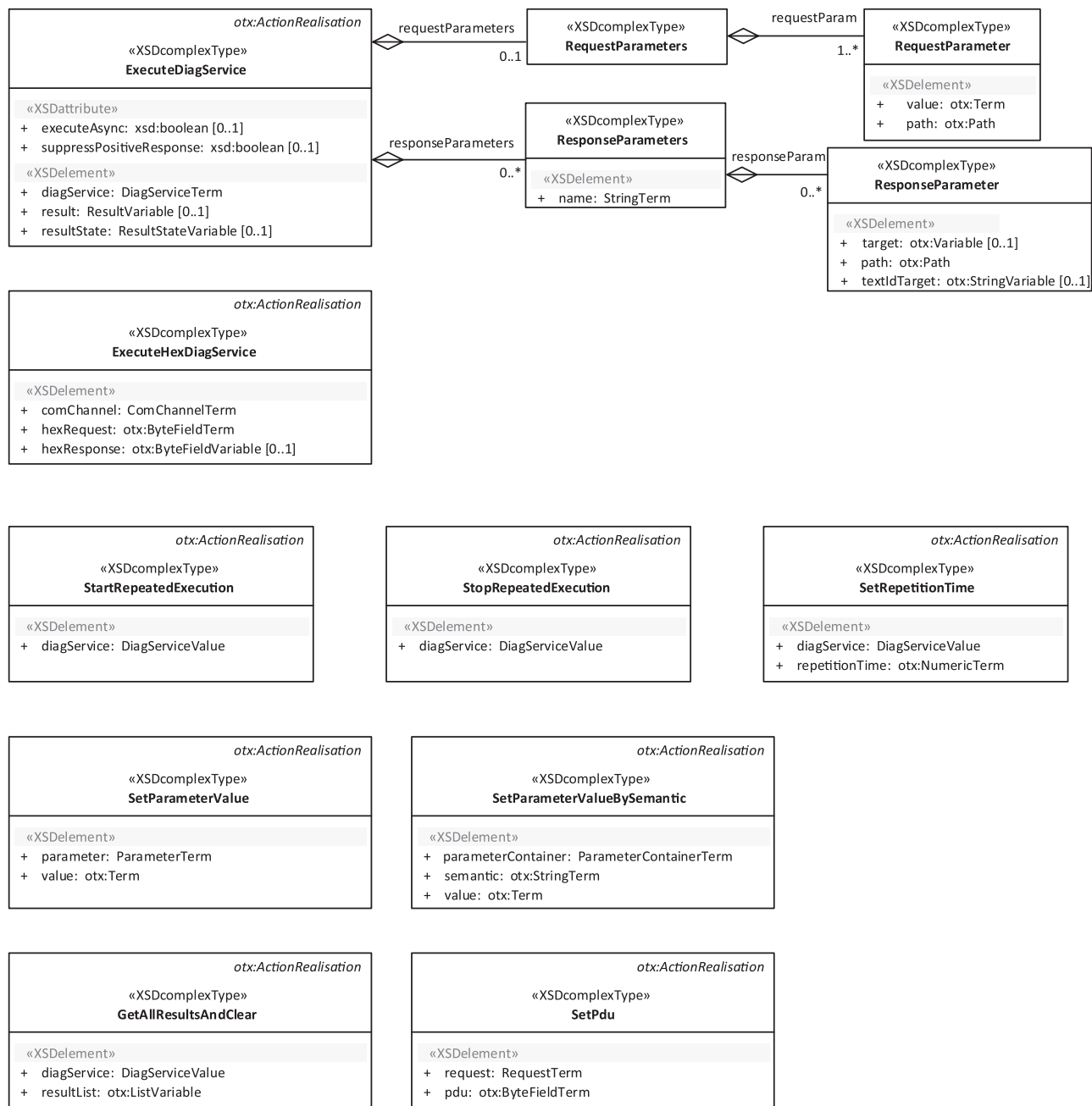


Figure 21 — Data model view: DiagService related actions

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:

- the **DiagService** to use;
- 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 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 subclause. 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 as defined by the UDS protocol^[6].

The mapping of the data types between ODX/MVCI and OTX is described in detail for inline mapping and dynamic response in [Annex B](#) which shall be followed.

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 shall 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 shall 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.

- 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, and there are any **<responseParameters>** elements, this shall result in an **UnknownResponseException**. If no **<responseParameters>** are defined, no **UnknownResponseException** will be thrown. 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 its 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 the **ExecuteDiagService** action is supposed to throw an exception in case an unexpected/unwanted response is encountered is simply a matter of providing a potentially 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 occurrence of a negative response to be an error, the author should only provide a mapping for the positive response's name. If the occurrence of the negative response is not considered an error and shall not cause an **UnknownResponseException**,

the author can simply provide a mapping for the negative response as well, which can be empty if the author does not care about the actual response parameters. The principle applies for any combination of responses (positive, negative, global negative, etc.); simply by providing an appropriate `<responseParameters>` element, the author can control whether the occurrence 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 [7.7.3.1](#)) 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 [\[6\]](#)).

— `<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 OTX values to request parameters 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 yielded 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 full path shall be used, that is all short names shall be contained, starting from the table row. In case of dynamic elements (e.g. `eEND_OF_PDU`, `eFIELD`, `eSTRUCT_FIELD`) an index as number can be inserted, instead of a short name.

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.

IMPORTANT — 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 response parameter values of a diagnostic service to OTX variables.

— `<target>` : `otx:Variable` [0..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 bytfields, parameters, 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 full path shall be used, that is all short names shall be contained, starting from the table row. In case of dynamic elements (e.g. `eEND_OF_PDU`, `eFIELD`, `eSTRUCT_FIELD`) an index as number can be inserted, instead of a short name.

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

— `<textIdTarget> : otx:StringVariable [0..1]`

This element specifies the OTX string variable the text identifier of the response parameter value shall be assigned to. If the value returned by the `ResponseParameter` does not have a text identifier an `otx:TypeMismatchException` shall be thrown.

NOTE In case an ODX/MVCI based system is used, the text identifier shall return the LongNameID of the related database object.

— `<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 (see [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 (see [7.7.5.3.8](#)).

Throws:

— `IncompleteParameterizationException`

It is thrown if one or more request parameters of the diag service have not been set and do not have a default value.

— `LossOfComException`

It is thrown if communication to the ECU was interrupted during diagnostic service execution.

— `UnknownTargetException`

It is thrown if no request or response parameter with the specified name in any of the parameter mappings exist.

— `UnknownResponseException`

It is thrown if execution of the diagnostic service returned a response that was not mapped by any `<responseParameters>` element.

— `otx:OutOfBoundsException`

It is thrown if a conversion cannot be made because an OTX value exceeds the limits of the target data type of a parameter of the vehicle communication component.

— `otx:TypeMismatchException`

It is thrown 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`.

If an OTX element is mapped to `<textIdTarget>` of a `ResponseParameter`, but the value returned by the `ResponseParameter` does not have a text identifier.

Associated checker rules:

- `DiagCom_Chk001` – no Path in `ExecuteDiagService` response parameter arguments (see [A.2.1](#));
- `DiagCom_Chk003` – target definition for `ResponseParameter` (see [A.2.3](#)).

An example for the `ExecuteDiagService` action is given in [7.6.4.4](#).

7.6.4.3.2 ExecuteHexDiagService

The `ExecuteHexDiagService` 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 shall 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 `ExecuteHexDiagService` 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`

It is thrown 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 behaviour depends on the underlying system. Especially if the repetition time value is 0 or lower than the physical possible repetition time. The `StartRepeatedExecution` action will return immediately, the results of the `DiagService` created by the repeated 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 do not 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 anymore.

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.

Associated checker rules:

- DiagCom_Chk002 – type-safe GetAllResultsAndClear (see [A.2.2](#)).

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 bytearray, an OTX list or an OTX map or an OTX **Quantity** as defined in [Clause 16](#).

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 on the parameter. Allowed input types are OTX simple types, OTX bytfields, OTX lists and OTX maps.

Throws:

— **otx:OutOfBoundsException**

It is thrown if the conversion cannot be made because the OTX value exceeds the limits of the target data type of a parameter of the vehicle communication component.

If the underlying system cannot set the parameter value, for example, because the parameter is a ResponseParameter or ConstantParameter.

— **otx:TypeMismatchException**

It is thrown 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 bytfield, list or map or an OTX **Quantity** as defined in [Clause 16](#).

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. While using 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 is 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 bytfields, lists and maps and OTX quantities.

Throws:

— **AmbiguousSemanticException**

It is thrown if there are none or more than one parameters present in the **ParameterContainerTerm** with the semantic value specified by the **<semantic>** element.