



BSI Standards Publication

Road vehicles — Open Test sequence eXchange format (OTX)

Part 2: Core data model specification and requirements

National foreword

This British Standard is the UK implementation of [ISO 13209-2:2022](#). It supersedes [BS ISO 13209-2:2012](#), which is withdrawn.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 02712 9

ICS 43.020; 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 July 2022.

Amendments/corrigenda issued since publication

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

INTERNATIONAL STANDARD

ISO
13209-2

Second edition
2022-07-26

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

Part 2: Core data model specification and requirements

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

Partie 2: Exigences et spécifications du modèle de données central



Reference number
ISO 13209-2:2022(E)

AVA Documents
Document Sharing Hub
© ISO 2022



COPYRIGHT PROTECTED DOCUMENT

© ISO 2022, Published in Switzerland

All rights reserved. Unless otherwise specified, 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
Ch. de Blandonnet 8 • CP 401
CH-1214 Vernier, Geneva, Switzerland
Tel. +41 22 749 01 11
Fax +41 22 749 09 47
copyright@iso.org
www.iso.org

Contents

Page

Foreword	vi
Introduction	vii
1 Scope	1
2 Normative references	1
3 Terms, definitions and abbreviated terms	2
3.1 Terms and definitions	2
3.2 Abbreviated terms	3
4 Requirements and recommendations	4
4.1 General	4
4.2 Basic principles for requirements and recommendations definition	4
4.3 Clustering of requirements and recommendations	4
4.4 Entries priorities	4
4.5 General format and language aspects	5
4.6 Test sequence development process support	6
4.7 Language feature details	7
4.7.1 Declarations	7
4.7.2 Data types	8
4.7.3 Expressions	9
4.8 Boundaries	12
5 Introduction to modelling in UML and XSD	14
5.1 General aspects	14
5.2 Class diagrams	14
5.2.1 General	14
5.2.2 Class	14
5.2.3 Inheritance relationships	15
5.2.4 Aggregation relationships	16
5.3 Mapping to the XML Schema Definition language (XSD)	16
5.3.1 General	16
5.3.2 Mapping rules	17
5.3.3 Full mapping example	17
6 OTX principles	20
6.1 General	20
6.2 XML format	20
6.3 Imperative and structured programming paradigm	21
6.4 Graphical authoring of OTX sequences	21
6.5 Specification/realisation concept	21
6.6 Modular OTX extension concept and OTX-based runtime architecture	21
6.7 Context concept	23
6.8 Validities concept	24
6.9 Signature concept	27
7 OTX core data model specification	28
7.1 General	28
7.2 High-level overview of the OTX core data model	29
7.3 Document root	30
7.3.1 Description	30
7.3.2 Syntax	31
7.3.3 Semantics	31
7.3.4 Example	34
7.4 Imports	35
7.4.1 Description	35
7.4.2 Syntax	35
7.4.3 Semantics	35

	7.4.4	Example	36
7.5		Global declarations	36
	7.5.1	Description	36
	7.5.2	Syntax	36
	7.5.3	Semantics	37
	7.5.4	Example	40
7.6		Validity terms	41
	7.6.1	Description	41
	7.6.2	Syntax	41
	7.6.3	Semantics	42
	7.6.4	Example	43
7.7		Signatures	43
	7.7.1	Description	43
	7.7.2	Syntax	44
	7.7.3	Semantics	44
7.8		Procedure signatures	45
	7.8.1	Description	45
	7.8.2	Syntax	45
	7.8.3	Semantics	45
	7.8.4	Example	46
7.9		Procedures	47
	7.9.1	Description	47
	7.9.2	Syntax	47
	7.9.3	Semantics	48
	7.9.4	Example	50
7.10		Floating comments	51
	7.10.1	Description	51
	7.10.2	Syntax	51
	7.10.3	Semantics	51
	7.10.4	Example	52
7.11		Parameter declarations	52
	7.11.1	Description	52
	7.11.2	Syntax	53
	7.11.3	Semantics	53
	7.11.4	Example	54
7.12		Local declarations	55
	7.12.1	Description	55
	7.12.2	Syntax	55
	7.12.3	Semantics	55
	7.12.4	Example	56
7.13		Nodes	56
	7.13.1	Overview	56
	7.13.2	Node	58
	7.13.3	Action node	59
	7.13.4	Compound nodes	62
	7.13.5	End nodes	84
7.14		Actions	91
	7.14.1	Overview	91
	7.14.2	Syntax	91
	7.14.3	General considerations	92
	7.14.4	Assignment	92
	7.14.5	ProcedureCall	92
	7.14.6	ByteFieldModifiers	98
	7.14.7	ListModifiers	101
	7.14.8	MapModifiers	104
7.15		Terms	106
	7.15.1	Overview	106
	7.15.2	Literal terms	107

7.15.3	Dereferencing terms	112
7.15.4	Creation terms	114
7.15.5	Conversion terms	117
7.15.6	Integer conversion terms	121
7.15.7	Logic operations	124
7.15.8	Relational operations	126
7.15.9	Mathematical operations	129
7.15.10	ByteField operations	133
7.15.11	List-related terms	136
7.15.12	Map-related terms	137
7.15.13	Exception-related terms	139
7.15.14	Validity concept related terms	140
7.16	Universal types	141
7.16.1	Overview	141
7.16.2	PackageName	141
7.16.3	OtxName and OtxLink	142
7.16.4	NamedAndSpecified	143
7.16.5	MetaData	145
7.16.6	Variable access	147
7.16.7	Declarations	149
7.16.8	Visibility	162
7.16.9	Flow	162
Annex A	(normative) OTX data types	165
Annex B	(normative) Scope and memory allocation	170
Annex C	(normative) Comprehensive checker rule listing	172
Annex D	(normative) Extension mechanism	185
Annex E	(normative) Schema annotations	188
Annex F	(informative) OTX home and URI recommendation	190
Bibliography		191

Foreword

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

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

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

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

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

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

This second edition cancels and replaces the first edition ([ISO 13209-2:2012](http://www.iso.org/iso/13209-2:2012)), which has been technically revised.

The main changes are as follows:

- introduction of new extension interfaces, e.g. for procedure realisations, compound nodes, and **NamedAndSpecified**;
- made **ForEachLoop** more convenient;
- new terms added (e.g. **RoundToNearest**);
- introduction of **MutexLock**;
- added new checker rules;
- deprecated **TerminateLanes** node.

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

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

Introduction

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

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

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

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

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

This page deliberately left blank

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

Part 2: Core data model specification and requirements

1 Scope

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

The requirements are derived from the use cases described in [ISO 13209-1](#). They are listed in the requirements section.

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

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

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

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

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

2 Normative references

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

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

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

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

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

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

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

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

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

3 Terms, definitions and abbreviated terms

3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in [ISO 13209-1](#) and the following apply.

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

- ISO Online browsing platform: available at <https://www.iso.org/obp>
- IEC Electropedia: available at <https://www.electropedia.org/>

3.1.1

attribute

<UML> property of a UML class

3.1.2

attribute

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

3.1.3

constant

identifier of a non-writable memory location

3.1.4

context

environmental circumstances which influence *test sequence* ([3.1.11](#)) execution

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

3.1.5

expression

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

3.1.6

identification routine

method or software by which a diagnostic application identifies contextual information

3.1.7

procedure signature

description of the interface of an OTX procedure

3.1.8

reference

value which refers to data in memory

3.1.9

session

instance of *test sequence* ([3.1.11](#)) execution

3.1.10

term

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

3.1.11

test sequence

test procedure (3.1.12) defining a full test

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

3.1.12

test procedure procedure

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

3.1.13

validity

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

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

3.1.14

variable

identifier of a writable memory location

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

3.2 Abbreviated terms

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

4 Requirements and recommendations

4.1 General

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

4.2 Basic principles for requirements and recommendations definition

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

- OTX requirements or recommendations specify the conditions that the OTX data model and format shall satisfy,
- all stakeholders (system suppliers, OEMs, tool suppliers), which offer diagnostic test procedures are expected to implement and follow the requirements of this document,
- the content of OTX documents and the quality of the information is the responsibility of the originator,
- the runtime system defines an OTX home directory. It is the entry directory for all relative file and directory access (see [Annex F](#)).

4.3 Clustering of requirements and recommendations

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

Table 1 — Main requirements clustering

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

4.4 Entries priorities

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

- **SHALL:**
The requirement represents stakeholder-defined characteristics, the absence of which will result in a deficiency that cannot be compensated by other means.
- **SHOULD:**
If the recommendation-defined characteristic is not or not fully implemented in the data model, it does not result in a deficiency, because other features in the data model can be used to circumvent this.

4.5 General format and language aspects

Core_R01 – Machine readable format

Priority: SHALL

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

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

Core_R02 – Platform independence

Priority: SHALL

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

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

Core_R03 – Well-defined syntax and semantics

Priority: SHALL

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

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

Core_R04 – Universal language

Priority: SHALL

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

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

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

Core_R05 – Minimal language

Priority: SHOULD

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

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

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

Core_R06 – Structured programming approach

Priority: SHALL

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

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

Core_R07 – Imperative structure

Priority: SHALL

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

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

Core_R08 – Extensibility

Priority: SHALL

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

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

4.6 Test sequence development process support

Core_R09 – Embed non-machine readable content

Priority: SHALL

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

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

Core_R10 – High level test procedure

Priority: SHALL

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

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

Core_R11 – Exchange high level test procedure

Priority: SHALL

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

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

Core_R12 – Exchange a fully functional test procedure

Priority: SHALL

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

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

Core_R13 – Exchange an intermediate stage test procedure

Priority: SHALL

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

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

Core_R14 – Floating comments

Priority: SHALL

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

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

4.7 Language feature details

4.7.1 Declarations

Core_R15 – Declarations

Priority: SHALL

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

Core_R16 – Initialisation

Priority: SHALL

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

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

Core_R17 – Constant declarations

Priority: SHALL

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

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

Core_R18 – Variable declaration

Priority: SHALL

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

Description: OTX shall support the declaration of variables.

Core_R19 – Input parameter declarations

Priority: SHALL

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

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

Core_R20 – Output parameter declarations

Priority: SHALL

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

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

Core_R21 – Two-way parameter declarations

Priority: SHOULD

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

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

NOTE This is the combination of input and output parameters.

4.7.2 Data types

Core_R22 – Strong typing

Priority: SHALL

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

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

Core_R23 – Data types

Priority: SHALL

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

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

Core_R24 – Extension mechanism for data types

Priority: SHALL

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

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

Core_R25 – Raw memory data type

Priority: SHALL

Description: OTX shall support a raw memory data type.

Core_R26 – Integer data type

Priority: SHALL

Description: OTX shall support an integer data type.

Core_R27 – Floating-point data type

Priority: SHALL

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

Core_R28 – String data type

Priority: SHALL

Description: OTX shall support a string data type.

Core_R29 – Boolean data type

Priority: SHALL

Description: OTX shall support a Boolean data type.

Core_R30 – Container data type

Priority: SHALL

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

4.7.3 Expressions

Core_R31 – Expressions

Priority: SHALL

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

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

Core_R32 – Extension mechanism for expressions

Priority: SHALL

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

Core_R33 – Literal expressions

Priority: SHALL

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

Core_R34 – Dereferencing expressions

Priority: SHALL

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

Core_R35 – Combined expressions: functions

Priority: SHALL

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

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

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

Core_R36 – Basic function set

Priority: SHALL

Description: OTX shall support the basic functions:

- mathematical: addition, subtraction, multiplication, division, negation, modulo, power,
- bitwise: conjunction, disjunction, negation,
- relations: equality, greater than, less than,
- logical: conjunction, disjunction, negation,
- string: concatenation.

Core_R37 – Statements

Priority: SHALL

Rationale: Statements are needed to represent single commands or higher-level control structures in a procedure. In order to achieve Turing-completeness, a set of basic statements with different semantics

shall be part of OTX. Statements are made up out of distinct parts, for example, in a loop statement, parts are a condition which is a logic expression, and a sub-sequence of commands. A procedure call statement consists of other parts, namely the called procedures name and a list of parameters.

Description: OTX shall support statements.

Core_R38– Extension mechanism for statements

Priority: SHALL

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

Core_R39 – Blocks of statements

Priority: SHALL

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

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

Core_R40 – Block statement

Priority: SHALL

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

Core_R41 – Assignment statement

Priority: SHALL

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

Core_R42 – Call procedure statement

Priority: SHALL

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

Core_R43 – Branch statement

Priority: SHALL

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

Core_R44 – Parallelisation statement

Priority: SHALL

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

Core_R45 – Loop statement

Priority: SHALL

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

Core_R46 – Continue statement

Priority: SHALL

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

Core_R47 – Break statement

Priority: SHALL

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

Core_R48 – Return statement

Priority: SHALL

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

Core_R49 – Exception handler statement

Priority: SHALL

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

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

Core_R50 – Throw statement

Priority: SHALL

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

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

Core_R51 – Validity information for statements

Priority: SHALL

Rationale: Enable/disable statements according to validity information.

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

4.8 Boundaries

Core_B01 – No graphical procedure layout information

Priority: SHALL

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

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

Core_B02 – Performance

Priority: SHALL

Rationale: OTX is solely a data format.

Description: No requirements on OTX performance exist.

Core_B03 – No versioning and configuration management

Priority: SHALL

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

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

Core_B04 – Licensing

Priority: SHALL

Rationale: OTX is solely a data format.

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

Core_B05 – No explicit memory management

Priority: SHALL

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

Core_B06 – No exception handling for environment fail

Priority: SHALL

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

Core_B07 – No global variables

Priority: SHALL

Description: OTX shall not allow variables with global scope.

Core_B08 – No string translation utilities

Priority: SHALL

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

Core_B09 – No unit localization utilities

Priority: SHALL

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

5 Introduction to modelling in UML and XSD

5.1 General aspects

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

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

5.2 Class diagrams

5.2.1 General

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

5.2.2 Class

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

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

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

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

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

- The top field contains the name of the class, e.g. **Contact**. For abstract classes, the class name is italic. The stereotype «**XSDcomplexType**» denotes that this class shall be mapped to an XSD complex type in the corresponding XML schema (stereotypes for the problem domain XSD).
- The second field contains the attributes of the class, e.g. the string labelled **contactId**, another string labelled **name** and an integer labelled **age**. The type descriptor is denoted directly after the colon behind the label. The stereotypes «**XSDelement**» and «**XSDattribute**» specify that **name** and **age** map to elements, whereas **contactId** maps to an attribute (in the XSD domain). Each

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

attribute in the class also carries cardinality information in square brackets, e.g. [0..1], [1..*], [2], etc. The cardinality defines how many instances of the attribute may appear in an instance of the class. If there is no cardinality shown, a default cardinality of [1] applies. Furthermore, a default value for the attribute may be specified.

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

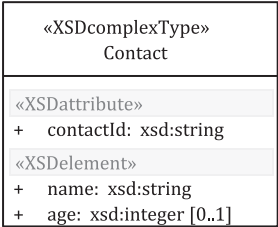


Figure 1 — UML representation of a class

5.2.3 Inheritance relationships

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

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

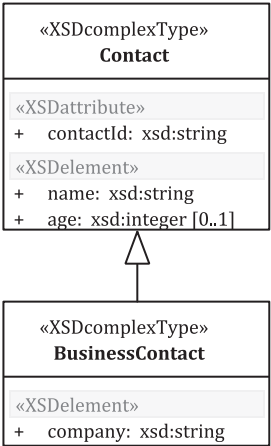


Figure 2 — UML representation of inheritance relationship

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

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

UML diagrams usually show certain aspects or fractions of the overall data model. Therefore, diagrams exist where a super-class of another class is not shown. To still reflect the fact that a class is a child of the hidden super-class, a special notation form has been chosen in this document, as presented in

[Figure 3](#): an additional property is shown to the upper right of the class **BusinessContact**, denoting its super-class named **Contact**.



Figure 3 — Alternative UML representation of inheritance relationship

5.2.4 Aggregation relationships

Besides the inheritance relationship, a pair of classes may also have an aggregation²⁾ relationship. Aggregation relationships are used if an object of one class is contained in an object of another class.

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

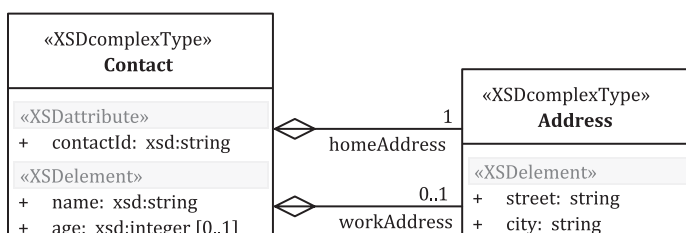


Figure 4 — UML representation of aggregation relationship

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

5.3 Mapping to the XML Schema Definition language (XSD)

5.3.1 General

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

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

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

5.3.2 Mapping rules

The following mapping rules apply.

- A class is mapped onto an XSD complex type with XSD sequence content as a default content type [the order of elements in an XSD sequence is significant (!)]. In cases where the stereotype of the class is not **«XSDcomplexType»**, analogous rules are applied and explained in the place needed.
- Attributes of a class are mapped onto attributes of the XSD complex type, if the **«XSDataAttribute»** stereotype is defined for the UML attribute. A cardinality of [1] will set the XSD attribute property **use = "required"**; a cardinality of [0..1] will result in **use = "optional"**. If there is no cardinality given, the default, **use = "optional"**, applies.
- Attributes of a class are mapped onto sub elements of the XSD complex type, if the **«XSDataElement»** stereotype is defined. The cardinality of the attribute is mapped to the cardinality properties of the sub element (XSD element properties **minOccurs**, **maxOccurs**). As an example, a cardinality of [1..*] will result in **minOccurs = "1"** and **maxOccurs = "unbounded"**. If there is no cardinality given, the default, **minOccurs = "1"** and **maxOccurs = "1"**, applies.
- Classes connected to another class via an aggregation relationship are mapped onto sub elements in the corresponding XSD complex type of the containing class by the following rules: The role names at the aggregation (contained end) are mapped onto the sub element **name** property. The name of a contained class is the **type** property of the corresponding sub element. The cardinality of an aggregation (contained end) is mapped to the cardinality of the sub element (XSD element properties **minOccurs**, **maxOccurs**).
- An inheritance relationship between a super-class **Super** and a child class **Child** is mapped onto the XSD extension **base** property of the XSD complex type **Child**, e.g. **base = "Super"**. Multiple inheritance (more than one super-class) cannot be mapped correctly to XSD, because XSD allows single inheritance only.
- An abstract marked class is mapped to an XSD complex type with the property **abstract = "true"**.

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

- **«XSDsequence»**, **«XSDchoice»**, **«XSDall»**, these are stereotypes for classes representing an XSD sequence, choice or all content-model separately or outside of an XSD complex type;
- **«XSDsimpleType»** to make a class represent a simple content type, e.g. classes derived from string, integer;
- **«XSDtopLevelElement»**, this makes a class represent a document root element (only used once in the OTX data model for the **<otx>** element definition itself).

5.3.3 Full mapping example

5.3.3.1 UML model example

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

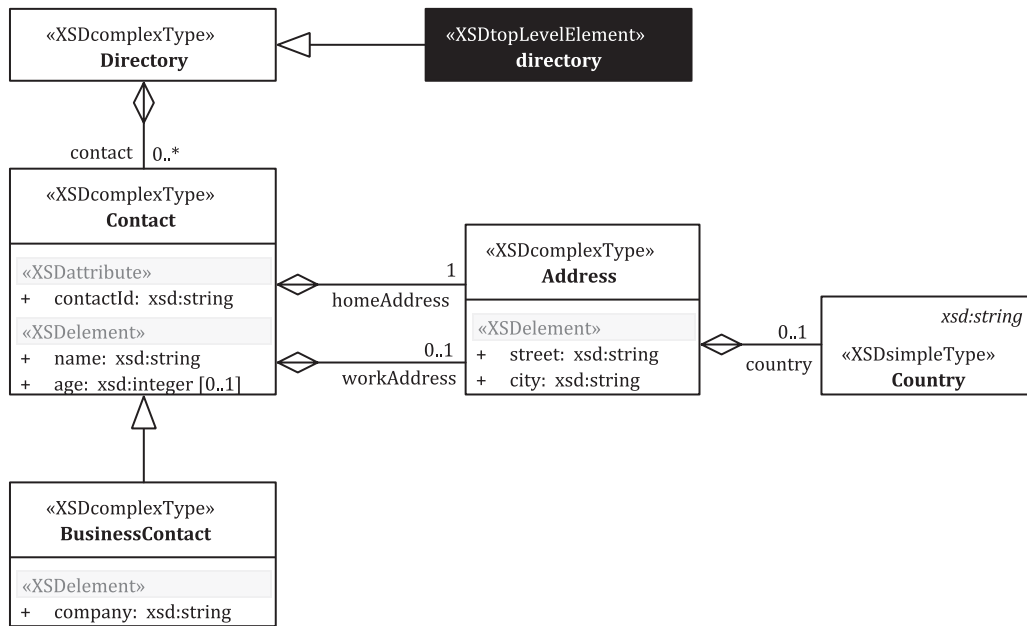


Figure 5 — Mapping example: “Directory” UML model

5.3.3.2 UML to XML translation

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

- The root of every such document shall be the top-level element **<directory>**. Since the corresponding **directory** class is derived from **Directory**, an unbounded list of **<contact>** elements is allowed below **<directory>**.
- The **<contact>** elements are of **Contact** type, so each of them has an attribute **contactId** and the elements **<name>**, **<age>** (optional), **<homeAddress>** and **<workAddress>** (optional).
- Simple type elements like **<name>** (of **xsd:string** type) or **<age>** (of **xsd:integer** type) are described implicitly within the aggregating **Contact** class.
- Features that are complex themselves, like the **<homeAddress>** (of **Address** type), are described explicitly by an own **Address** class, which is connected to **Contact** by an aggregation relation³⁾.
- The elements **<homeAddress>** and **<workAddress>** have simple string sub elements **<street>** and **<city>** and also **<country>**.
- In UML, **<country>** is modelled in a special way: it is a simple string and could have been modelled as such, but here the alternative display of inheritance relationships shall be exemplified (see [Figure 3](#)).

5.3.3.3 XSD result

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

Sample of directory

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

```

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

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

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

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

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

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

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

```

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

5.3.3.4 XML instance document

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

Sample showing some directory entries

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

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

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

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

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

6 OTX principles

6.1 General

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

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

6.2 XML format

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

- XML is designed to represent hierarchically structured information, which applies especially for test sequences;
- the basic XML rules for well-formedness eliminate the need for defining an own set of such basic grammar rules specific for OTX;
- checking XML documents for well-formedness is a well-defined task for which a great many number of solutions exist off-the-shelf; thus, there is no need to implement well-formedness-checkers specifically for OTX;
- by modelling the detail correlations of the OTX grammar by using XSD, validity checking can be done by off-the-shelf solutions;

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

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

6.3 Imperative and structured programming paradigm

OTX follows the paradigm of imperative, structured programming:

- **imperative:** OTX procedures define sequences of commands for the computer to perform;
- **structured:** there are no explicit jumps allowed in the OTX. Instead, branches, loops, exception handlers, return, continue, break and throw statements as well as procedure calls define jumps implicitly and in a **controlled manner**. Therefore, the OTX also defines a block structure, as typically found in all languages that follow the structured programming paradigm.

6.4 Graphical authoring of OTX sequences

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

6.5 Specification/realisation concept

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

- **Specification stage:** a test sequence may be described at specification level only. This is helpful for early stages in the test sequence development process, when not all of the details for creating an executable test sequence are known. The specification stage allows the development of test sequences at a prose level, which is human readable, but not fully executable—i.e. the overall sequence logic may already be there, but single steps in the sequence are filled by prose only, so a runtime interpreter cannot make use out of it. However, starting from this stage, the test sequence can be continuously broken down by the following stages.
- **Intermediate stage:** OTX also allows intermediate stage test sequences, which occur while transforming a specification stage test sequence towards a realisation stage sequence. Such an intermediate stage test sequence may contain parts that are already fully executable, whereas other parts are still at specification stage. The sequence author uses the human readable information from the specification stage parts and implements those parts following the specification instructions by adding realisation counterparts. At any step in this process, the sequence is validly saveable and exchangeable.
- **Realisation stage:** At this stage, there are no more specification-only parts in the sequence. The sequence is then fully executable. Note that even at this stage, specification parts of the sequence may still be contained, but not without their corresponding realisation. With other words, this coexistence represents an “executable specification”.

6.6 Modular OTX extension concept and OTX-based runtime architecture

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

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

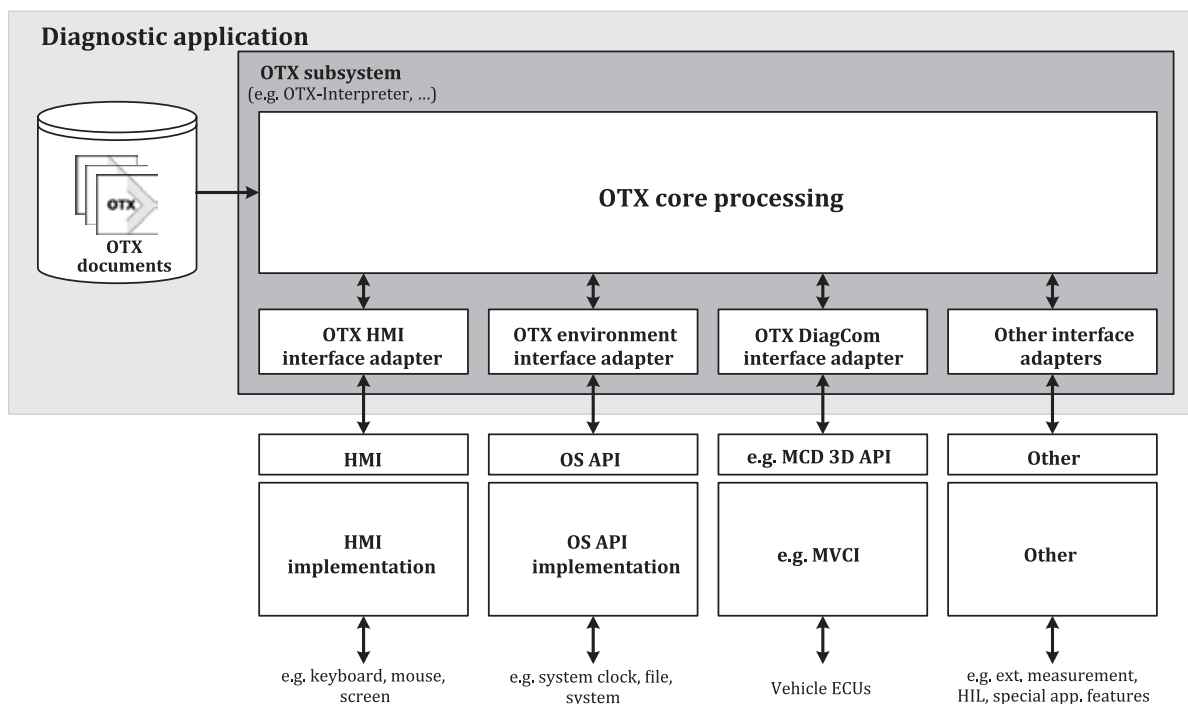


Figure 6 — OTX-based runtime architecture

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

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

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

6.7 Context concept

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

- Vehicle related: data about the currently diagnosed vehicle's model, vendor, identification number, engine-type or other identification data concerning the physical and electronical vehicle configuration.
- Diagnostic application related: application version, name, used vehicle communication interface type or version or also diverse settings of the application.
- User related: login name, idle time, user access rights, etc.
- Application area related: information about the location where the test sequences are applied, to make the difference, e.g. between manufacturing, engineering or workshop appliance.

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

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

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

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

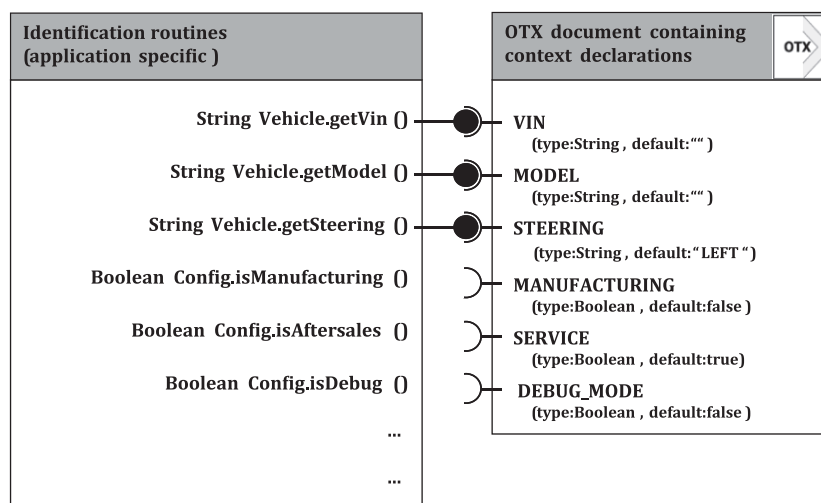


Figure 7 — Mapping context declarations to identification routines

The OTX context concept has several advantages.

- Transparency: working with context data is almost similar to working with global constants for OTX authors.
- Compatibility: integrating OTX into existing diagnostic applications does not require a change of the original context identification/management strategy of the application or the diagnostic session management, etc.
- Exchangeability: since OTX documents declare each context item they require explicitly and in a well-defined place, using an OTX sequence originating from a foreign diagnostic application only requires an adjustment of the mapping between context declarations and identification routines (and possibly the implementation of new identification routines). In this way a context dependent OTX document can be “docked” to heterogeneous diagnostic applications.
- Cooperation: by agreeing upon an OTX-file or a set of OTX-files declaring commonly used context data, two or more parties can improve their cooperation concerning OTX test sequence development.
- Simulation: various context situations can be simulated by, e.g. feeding simulated data to identification routines.
- Aside from the above topics, the context concept is crucial for the validities concept which is described in [6.8](#).

6.8 Validities concept

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

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

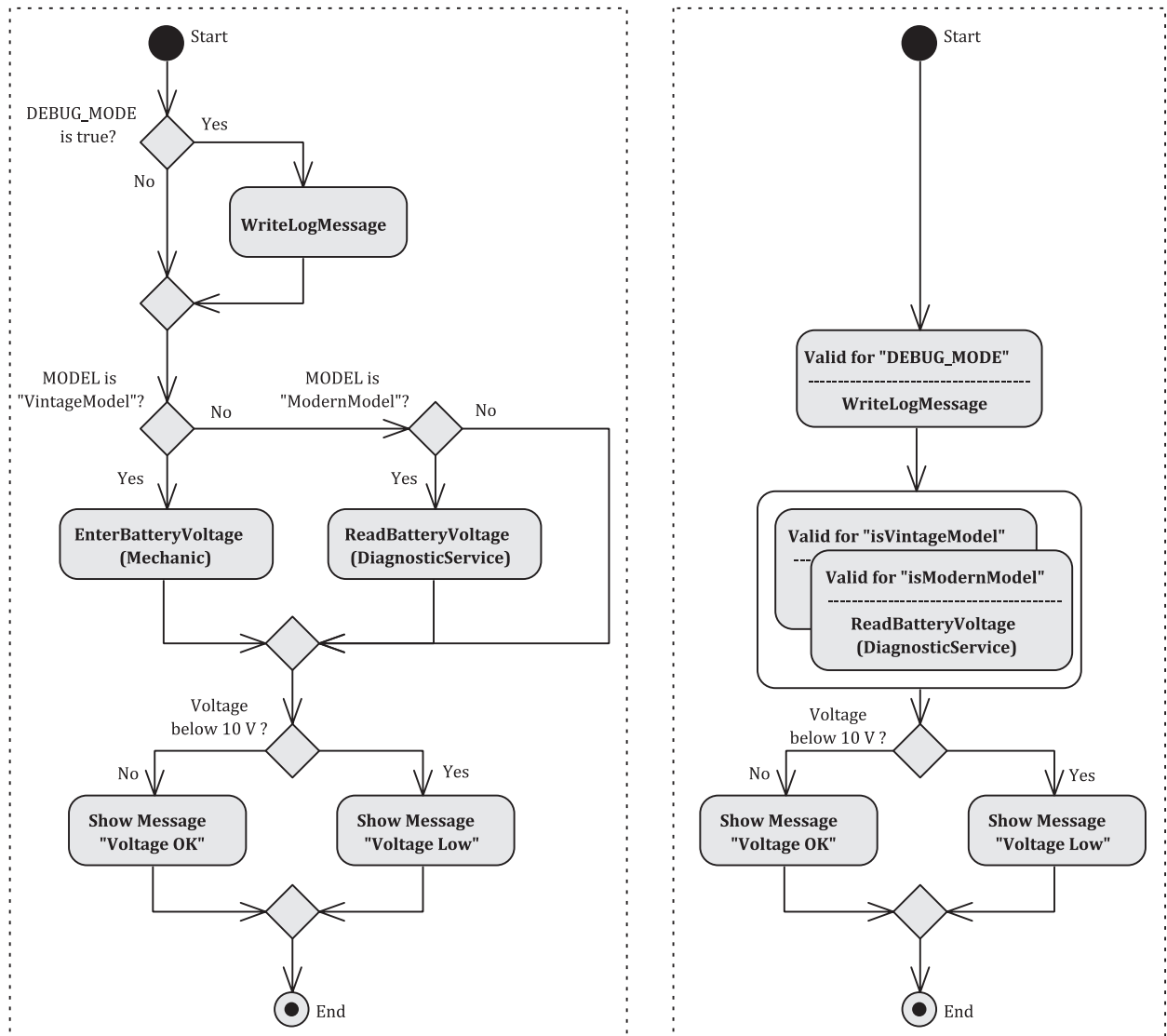


Figure 8 — Validities concept, example of use

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

- a) If the diagnostic application's runtime system is in debug mode, it shall write a log message.
- b) The battery voltage of a vehicle shall be measured. According to the type of vehicle under test (a vintage model, a modern model or neither of these, e.g. a bicycle), this shall happen in different ways:
 - 1) vintage model: the voltage shall be measured and entered into the system manually.
 - 2) modern model: the voltage can be queried automatically (by using a diagnostic service).
 - 3) it is some other type of vehicle with no battery: nothing can be measured.
- c) If the measured voltage is too low, a message "voltage low" shall be shown, otherwise "voltage OK".

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

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

- b) The step for measuring the voltage comes in two different flavours: one is valid only for “isVintageModel”, the other only for “isModernModel”. If none is valid, no action is specified (there is some other type of vehicle). The validities “isVintageModel” and “isModernModel” are validity terms analysing the current value of the “MODEL” context item.
- c) The voltage value is dynamic: it is neither predictable at authoring time nor depending on the context. It is only known at runtime. It is not a configuration. That is why validities are not used here.

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

The advantages of this approach are the following.

- A distinction is drawn between decisions based on static context data (e.g. enumerations of expected context values known at authoring time) and dynamic data (which is computed at runtime).
- Validity information controls the flow implicitly regarding context, not explicitly by normal dynamic branch conditions. This fact is also reflected in the more compact way of representation, the “pure” test sequence logic becomes more apparent.
- Sets of commonly used validities can be stored separately, in a central place. This prevents a lot of redundancy, since authors can reuse preconfigured validities for their test sequences. This also improves maintainability.
- Filtering: OTX authoring system may allow the author to configure certain context situations in a simulated environment. This produces very compact views of test sequences for a given context (mask steps that are not valid).
- Filtered test sequences may be extracted and/or exported. This results in test sequences which are tailored for a specific context.
- Enables the signature concept which is described later in this document.

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

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

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

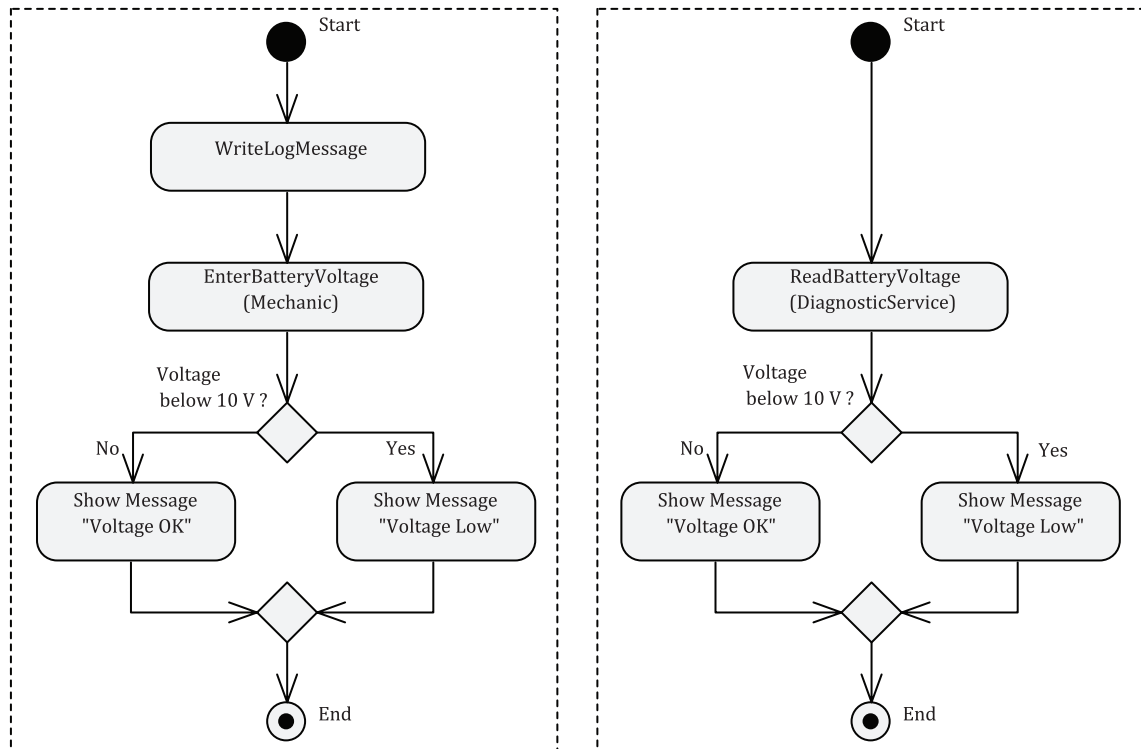


Figure 9 — Validities concept, filtering

6.9 Signature concept

The OTX signature concept is designed to support dynamic linking.

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

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

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

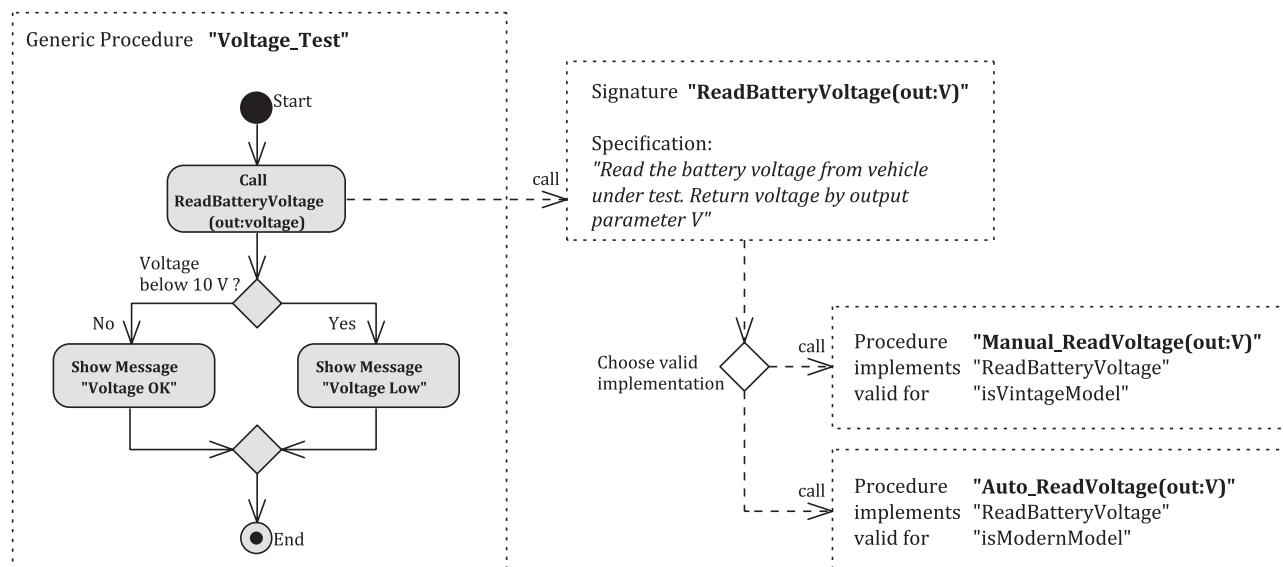


Figure 10 — Signatures concept

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

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

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

7 OTX core data model specification

7.1 General

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

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

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

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

- [Annex A](#) specifies the OTX data types and shall be followed.
- [Annex B](#) specifies the runtime behaviour concerning scope and memory allocation of OTX constants and variables and shall be followed.

- [Annex C](#) contains a comprehensive listing of all checker rules and shall be followed. The rules are needed because some constraints existing on OTX documents cannot be ensured by XSD validation alone. These constraints need to be checked by additional checker applications.
- [Annex D](#) describes the extension mechanism of the OTX core data model and shall be followed. By the creation of OTX extensions new features can be added to the format. Nevertheless, there are well-defined extension rules which shall be adhered by every correct OTX extension.
- [Annex E](#) describes the special schema annotations for exceptions and shall be followed.
- [Annex F](#) contains the definition of the OTX home directory and is recommended.

7.2 High-level overview of the OTX core data model

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

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

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

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

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

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

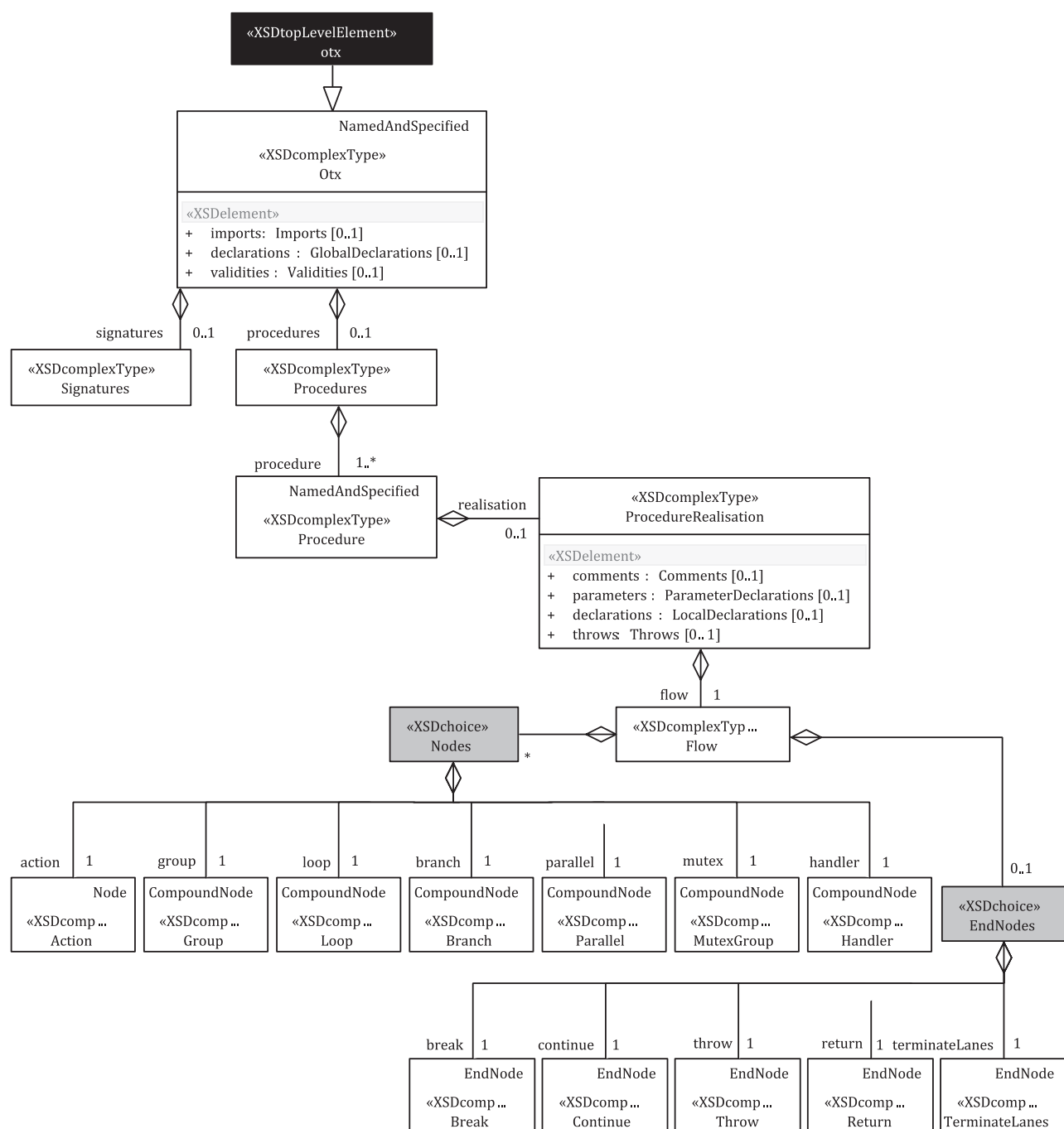


Figure 11 — High-level overview of the OTX core data model

7.3 Document root

7.3.1 Description

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

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

7.3.2 Syntax

Every OTX document starts with the document root element `<otx>` which is derived from the complex type `Otx`. The syntax is shown in [Figure 12](#).

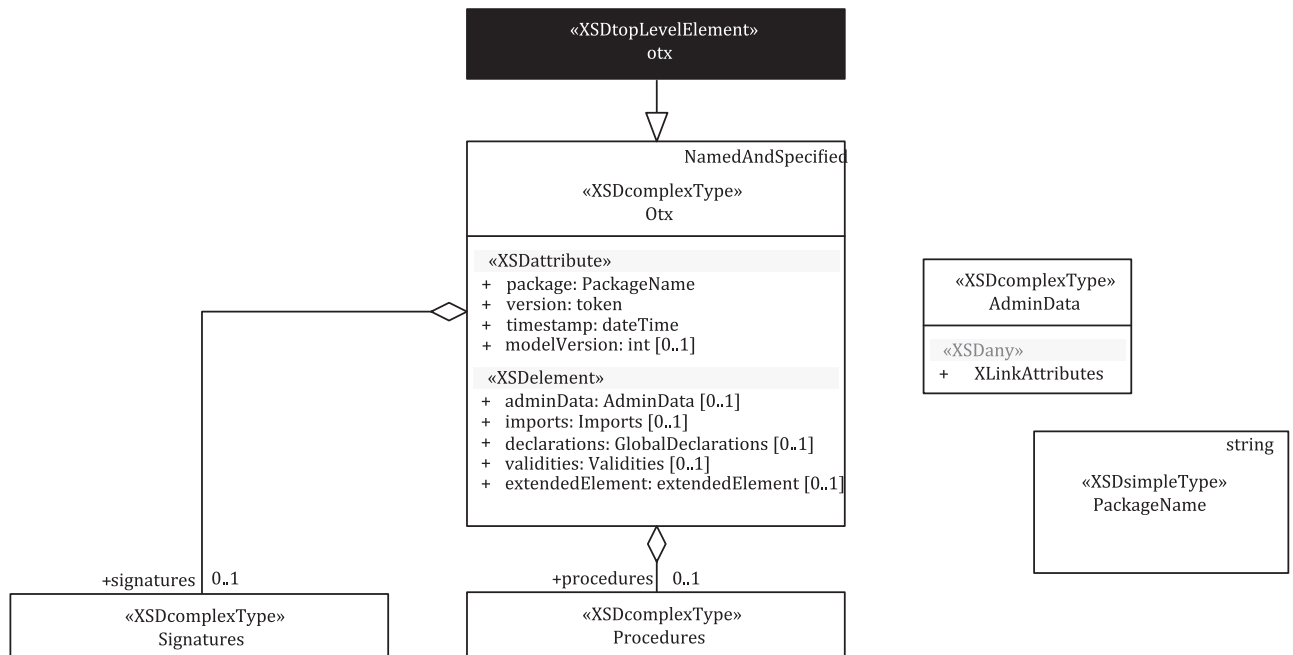


Figure 12 — Data model view: OTXdocument root

7.3.3 Semantics

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

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

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

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

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

Associated checker rules:

- Core_Chk001 – document name matches file name (see [C.2.1](#));
- Core_Chk002 – package-wide uniqueness of document names (see [C.2.2](#)).
- **package:** `PackageName` [1]

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

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

Recommendations:

- OTX documents can be stored in a file system or in a database. If stored in a file system, certain constraints on the organization of files and folders allow a simple implementation to find OTX documents easily. The same applies to the uniqueness of file names, which is ensured automatically by the file system if the OTX-files reside within a folder dedicated to their common package. The package name (in dotted notation) can simply be translated into a directory path or vice versa, whereas the OTX document name can be translated directly to the file name. This practice is strongly recommended. It is an analogy to the file system organization of packages and classes recommended by Reference [1], subclause 7.2.
- For the transport of larger OTX projects consisting out of numerous interlinked OTX-files and packages, the usage of the PDX archive format specified by the ISO 22901 series is strongly recommended. In the case of an MVCI[4]/ODX (see the ISO 22901 series) based system, this allows transporting OTX data along with the ODX data required by the OTX sequences.
- The file name extension should be *.ptx. The following values are predefined for CATEGORY:
 - OTX-MAIN for the main OTX document which is the entry point for the runtime execution. The element is optional. No more than one OTX document in the PDX can have the OTX-MAIN category. The referenced OTX document shall have a main procedure.
 - OTX-DATA for all other OTX documents.
- The file extension for OTX documents should be *.otx .
- The attribute F-XSD-VERSION shall contain the OTX version instead of ODX-version.
- The index xml can have an optional version number of the PDX file directly after the prolog in the format of a processing instruction. The format of the version is unspecified.
Sample: < ?PTX version = "1.2.3.4"? >
- Alternatively (if the latter recommendation is not applicable), the Java JAR file format is recommended. It represents a simple and popular archiving method. It is, like PDX, based on the ZIP archive format. Therefore, both approaches PDX and JAR could even be combined in one project file.

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

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

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

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

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

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

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

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

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

Overview of common XML namespace associations used for OTX:

- **xmlns** = "<http://iso.org/OTX/1.0.0>"
- **xmlns:diag** = "<http://iso.org/OTX/1.0.0/DiagCom>"
- **xmlns:hmi** = "<http://iso.org/OTX/1.0.0/HMI>"
- **xmlns:myext** = "(custom OTX extension namespace)"
- **xmlns:xsi** = "<http://www.w3.org/2001/XMLSchema-instance>"
- **xmlns:xmime** = "<http://www.w3.org/2005/05/xmlmime>"

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

- **version: xsd:token** [1]

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

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

- **timestamp: xsd:dateTime** [1]

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

- **modelVersion: xsd:int** [0..1]

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

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

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

This optional element shall contain meta information concerning the OTX document as a whole. For a description of **MetaData** refer to [7.16.5](#).

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

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

— **<adminData>: AdminData [0..1]**

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

— **<imports>: Imports [0..1]**

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

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

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

— **<validities>: Validities [0..1]**

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

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

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

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

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

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

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

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

7.3.4 Example

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

Sample for a DocumentRootExample

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

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

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

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

7.4 Imports

7.4.1 Description

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

7.4.2 Syntax

[Figure 13](#) shows the syntax of the **Imports** type.

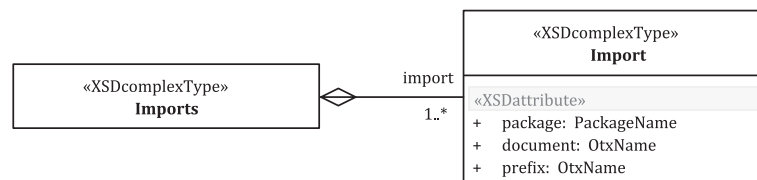


Figure 13 — Data model view: Imports

7.4.3 Semantics

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

— **package: PackageName [1]**

Contains the package which the imported document belongs to (see [7.16.2](#) for the **PackageName** type).

— **document: OtxName [1]**

Contains the document name of the document which shall be imported (see [7.16.3](#) for the **OtxName** type).

— **prefix: OtxName [1]**

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

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

Associated checker rules:

- Core_Chk003 – no dead import links (see [C.2.3](#));
- Core_Chk004 – no unused imports (see [C.2.4](#));
- Core_Chk006 – match of imported document's data model version (see [C.2.6](#)).

7.4.4 Example

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

Sample of imports

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

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

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

7.5 Global declarations

7.5.1 Description

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

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

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

7.5.2 Syntax

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

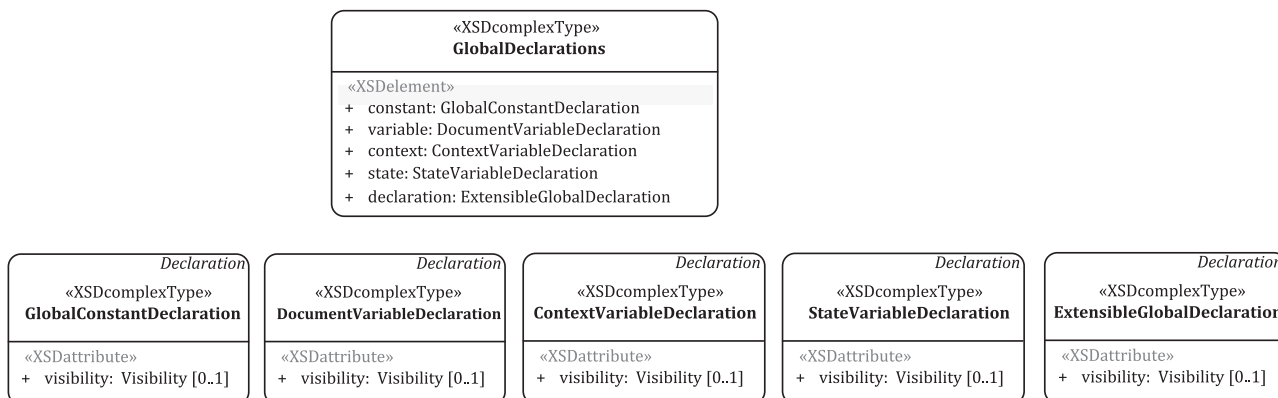


Figure 14 — Data model view: GlobalDeclarations

7.5.3 Semantics

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

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

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

- **<constant>: GlobalConstantDeclaration**

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

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

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

- **"PRIVATE"**: the constant shall be used only from procedures or validities defined in the **same document**. This means that the constant shall be invisible for other OTX documents. This is the default.
- **"PACKAGE"**: the constant shall be used only from procedures or validities defined in the **same package** like the local OTX document. This means that the constant shall be invisible for OTX documents belonging to other packages.
- **"PUBLIC"**: the constant can be used from **anywhere**.

Associated checker rules:

- Core_Chk051 – immutability of constants, input parameters and context variables (see C.2.51);
- Core_Chk009 – mandatory constant initialization (see C.2.9).
- **<variable>: DocumentVariableDeclaration**

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

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

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

- **<context>:** **ContextVariableDeclaration**

Declares a context variable. See [6.7](#) for further information about the OTX context concept.

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

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

- "**PRIVATE**": the context variable shall be used only from procedures or validities defined in the **same document**. This means that the context variable shall be invisible for other OTX documents. This is the default.
- "**PACKAGE**": the context variable shall be used only from procedures or validities defined in the **same package** as the local OTX document. This means that the context variable shall be invisible for OTX documents belonging to other packages.
- "**PUBLIC**": the context variable can be used from **anywhere**.

Associated checker rules:

- Core_Chk051 – immutability of constants, input parameters and context variables (see [C.2.51](#)).

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

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

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

Recommendations:

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

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

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

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

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

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

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

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

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

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

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

— **<state>: StateVariableDeclaration**

Declares a state variable.

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

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

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

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

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

Associated checker rules:

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

- Core_Chk065 – write only state variables (see [C.2.65](#)).
- **visibility: Visibility = {PRIVATE|PACKAGE|PUBLIC} [0..1]**

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

- "**PRIVATE**": the context variable shall be used only from procedures or validities defined in the **same document**. This means that the context variable shall be invisible for other OTX documents. This is the default.
 - "**PACKAGE**": the context variable shall be used only from procedures or validities defined in the **same package** as the local OTX document. This means that the context variable shall be invisible for OTX documents belonging to other packages.
 - "**PUBLIC**": the context variable can be used from **anywhere**.
- **<declarartion>: ExtensibleGlobalDeclaration**

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

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

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

- "**PRIVATE**": the context variable shall be used only from procedures or validities defined in the **same document**. This means that the context variable shall be invisible for other OTX documents. This is the default.
- "**PACKAGE**": the context variable shall be used only from procedures or validities defined in the **same package** as the local OTX document. This means that the context variable shall be invisible for OTX documents belonging to other packages.
- "**PUBLIC**": the context variable can be used from **anywhere**.

7.5.4 Example

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

- A global constant float named "**PI**", initialized by float value "**3.14159265**". Its visibility is "**PUBLIC**" which means that the constant can be accessed by any other importing OTX document.
- A Boolean document variable named "**dv**". The **visibility** attribute is not given explicitly in the document; nonetheless the variable is implicitly "**PRIVATE**". This setting is fixed for all document variables by the OTX schema.
- A context variable "**VIN**", which is connected to an identification routine. The metadata assigned to the context variable contains a mapping to the identification routine; it is an OTX procedure named "**getVehicleIdent**" belonging to the package "**identification**". The out-parameter "**vin**" yields the context value (the vehicle identification number of the currently diagnosed vehicle).
- A context variable "**HAS_SUNROOF**", there is no mapping to an identification routine given in the metadata. If there is no external mapping also, the context variable is not connected to the "real world" – in this case the default value **false** will apply.

Sample of GlobalDeclarations