



ASAM XIL

Generic Simulator Interface

Part 1 of 5

Programmers Guide

Version 2.2.0

Date: 2020-03-13

Base Standard

Disclaimer

This document is the copyrighted property of ASAM e.V.
Any use is limited to the scope described in the license terms. The license
terms can be viewed at www.asam.net/license

Table of Contents

Foreword	9
1 Introduction	10
1.1 Overview	10
1.2 Motivation	10
1.3 What is Hardware in the Loop Simulation	11
1.4 Technical Approach	12
1.5 Technology Independence	12
2 Relations to Other Standards	14
2.1 Backward Compatibility to Earlier Releases	14
2.2 References to Other Standards	15
2.3 Versioning.....	15
3 General Concepts	16
3.1 XIL Test System Architecture	16
3.2 Overview Framework	18
3.2.1 General.....	18
3.2.2 Initialization	18
3.2.3 Framework Variables	18
3.2.4 Acquisition and Stimulation	19
3.3 Overview Testbench.....	22
3.3.1 General.....	22
3.3.2 Ports of the XIL Testbench API	22
3.4 ASAM Data Types.....	23
3.5 Instance Creation	23
3.5.1 Implementation Manifest File	24
3.5.1.1 File content.....	24
3.5.1.2 File Naming convention and storage location	28
3.5.2 Framework and Testbench Factory.....	29
4 Framework	31
4.1 Configuration.....	31
4.1.1 Framework Configuration File	31
4.1.2 Configuring the Framework	35
4.1.3 Shutting down the Framework.....	37
4.1.4 Customize Automatic port management of the framework	38
4.1.5 Object model of the framework configuration	39
4.2 Mapping	41
4.2.1 Overview	41
4.2.2 The Mapping XML Schema.....	42

4.2.2.1	Identifier Lists	44
4.2.2.2	Framework Label Group.....	47
4.2.2.3	Identifier Mapping.....	48
4.2.2.4	String Mapping	50
4.2.2.5	Raster Mapping	51
4.2.2.6	Conversion Tables	51
4.2.2.7	Units	52
4.2.2.8	Computation Tables	54
4.2.2.9	Versioning	55
4.2.3	How a Mapping is used by Test Cases	55
4.2.3.1	The Mapping Info API.....	55
4.2.3.2	Identifier Mapping.....	57
4.2.3.3	String Mapping	57
4.2.3.4	Raster Mapping	57
4.2.4	Tasks of the Framework.....	57
4.2.4.1	Reading mapping XML files	57
4.2.4.2	Consolidating mapping object trees	57
4.2.4.3	Implementing the Mapping Info API	58
4.2.4.4	Resolve the Mapping during Variable Creation.....	60
4.2.4.5	Perform automatic string mappings	60
4.2.4.6	Error handling.....	60
4.2.5	Consistency Rules	60
4.3	Framework Variables	62
4.3.1	What is a Framework Variable	62
4.3.2	Framework variable Classes	63
4.3.2.1	Scalar Variables	66
4.3.2.2	Vector Variables	67
4.3.2.3	Matrix Variables.....	68
4.3.2.4	Curve Variables	69
4.3.2.5	Map Variables	71
4.3.3	Quantities.....	73
4.3.3.1	Scalar Quantity Classes	74
4.3.3.2	Complex Quantity Data Classes	77
4.3.3.3	Usage of Quantities	79
4.3.4	Unit and Physical Dimensions	82
4.3.4.1	Unit Class	84
4.3.4.2	Physical Dimension Class	84
4.3.5	MetaData	84
4.3.6	Calculation with quantity objects	85
4.3.6.1	Overview.....	85
4.3.6.2	Addition and Subtraction	86
4.3.6.3	Multiplication and Division	86
4.3.6.4	Computation Table	87
4.3.6.5	Comparison	90
4.3.7	Data Type Conversion	90
4.3.8	Unit Conversion	90
4.3.9	Usage of Mathematical Operations	91
4.3.9.1	ABSOLUTE and RELATIVE.....	91
4.3.9.2	NEUTRAL UNIT	91
4.3.9.3	Mathematical and Physical Constants	92
4.3.9.4	Example.....	92
4.4	Measuring	93
4.4.1	Motivation	93
4.4.2	Setting up an Acquisition.....	94
4.4.2.1	Introduction.....	94
4.4.2.2	Using the Acquisition Interface	94

4.4.2.3	Acquisition States	96
4.4.2.4	Using the AcquisitionConfiguration Interface	96
4.4.2.5	Principle of Triggered Acquisition	98
4.4.2.6	Synchronized Data Acquisition	102
4.4.3	Setting up a Recording	105
4.4.3.1	Introduction	105
4.4.3.2	Configuring an AcquisitionRecorder	106
4.4.3.3	Configuring a Recorder	106
4.4.3.4	Using RecorderResult Readers and Writers	107
4.4.3.5	Working with the Recorded Results	109
4.4.3.6	Recorder States	111
4.4.4	Sequence of Acquisition and Recorder Commands	112
4.5	Stimulation.....	114
4.5.1	Motivation	114
4.5.2	Setting up a Stimulation	114
4.5.2.1	Introduction	114
4.5.2.2	Using the Stimulation Interface	115
4.5.2.3	Stimulation States	115
4.5.3	Setting up a Player	116
4.5.3.1	Introduction	116
4.5.3.2	Configuring a Player	117
4.5.3.3	Using a Player	118
4.5.3.4	Player States	119
5	Testbench	120
5.1	Common Functionalities.....	120
5.1.1	ValueContainer	121
5.1.1.1	Overview	121
5.1.1.2	General Value Container Classes	121
5.1.1.3	Specific Value Container Classes	123
5.1.2	Document Handling	124
5.1.3	Script	125
5.1.3.1	Overview	125
5.1.3.2	States of Script	126
5.1.3.3	Script Parameters	130
5.1.4	TargetScript	131
5.1.4.1	Overview	131
5.1.4.2	Parameters	132
5.1.4.3	Custom Properties	132
5.1.4.4	Usage of TargetScript	132
5.1.5	Signal Descriptions	134
5.1.5.1	General Remarks about Segment-Based Signals	137
5.1.5.2	Signal Segments	140
5.1.5.3	Converting SignalDescriptions to sample based representations ...	160
5.1.5.4	Using Signal Descriptions	161
5.1.5.5	Signal Description File	167
5.1.5.6	Usage of Parameterized SignalDescriptions	168
5.1.6	SignalGenerator	171
5.1.6.1	Interpretation of SignalDescriptions by the SignalGenerator	172
5.1.6.2	Parameters	176
5.1.6.3	Custom Properties	176
5.1.6.4	Usage of SignalGenerator (Stimulating Model Variables)	176
5.1.7	Document Handling for SignalGenerator and SignalDescriptionSet ...	178
5.1.8	Watcher	181
5.1.8.1	General	181
5.1.9	Duration	182



5.1.10	Meta Info	183
5.1.10.1	Metadata on Variables	183
5.1.10.2	Metadata on Conversion Methods	185
5.1.10.3	Metadata on Acquisition Rasters / Processor Tasks	187
5.1.11	Variable Ref and Value Representation Mode.....	188
5.1.12	Data Capturing.....	190
5.1.12.1	Introduction.....	190
5.1.12.2	Capture Configuration and Control	190
5.1.12.3	Untriggered Capturing	196
5.1.12.4	Triggered Capturing	197
5.1.12.5	Re-triggered Capturing	199
5.1.12.6	Obtain Capture Results from a Capture.....	201
5.1.12.7	Capture Result	202
5.1.12.8	Relation between Triggers, Capture State, Capture Events and Data Frames.....	206
5.1.12.9	Reader and Writer for CaptureResult.....	215
5.1.12.10	Usage examples for Capture and Capture Result	216
5.2	Model access Port	227
5.2.1	User Concept	227
5.2.1.1	General.....	227
5.2.1.2	MAPort Interface	227
5.2.1.3	States of the MAPort	229
5.2.2	Usage of MAPort.....	231
5.2.2.1	Creation and Configuration	231
5.2.2.2	Obtaining Information on Available Model Variables, Tasks and their Properties	233
5.2.2.3	Reading & Writing Model Variables	236
5.2.2.4	State dependency of Variables' Writeability and Readability	240
5.2.2.5	Relation between MAPort and Capturing / SignalGenerator	241
5.2.2.6	Pausing and stepwise execution of the simulation	242
5.3	Diagnostic Port	246
5.3.1	Overview	246
5.3.2	API	247
5.3.2.1	Communication Modes.....	248
5.3.2.2	ECU	248
5.3.2.3	Functional Groups	249
5.3.3	States of the DiagPort	249
5.3.4	Usage of this Port	251
5.3.4.1	Creation and Configuration	251
5.3.4.2	Getting The Ecu Object	252
5.3.4.3	Reading And Clearing The Fault Memory	253
5.3.4.4	Reading The Variant Coding Data	254
5.3.4.5	Reading Identification Data	254
5.3.4.6	Reading and Writing Values from and to the Eeprom by Alias Names	255
5.3.4.7	Reading from the Eeprom	256
5.3.4.8	Writing to the Eeprom.....	256
5.3.4.9	Implicit and Explicit Communication.....	257
5.3.4.10	Sending Hex Services with Explicit Communication	257
5.3.4.11	Executing Jobs	258
5.3.4.12	Reading Data from a Functional Group	259
5.3.4.13	Using the Basecontroller	260
5.3.5	Special Hints	260
5.3.5.1	Structure of Returned Collections	260
5.3.5.2	States in the Diagnostic Tool	260
5.4	ECUMPort	261



5.4.1	Overview	261
5.4.2	States of the ECUMPort	264
5.4.3	Usage of ECUMPort	266
5.4.3.1	Creation and Configuration	266
5.4.3.2	Getting Lists of Variable and Task Names	267
5.4.3.3	Read a Scalar Variable Value and its Properties	268
5.4.3.4	Read an Array Variable Value and its Properties	268
5.4.3.5	Read a Matrix Variable Value and its Properties	269
5.4.3.6	Capturing ECU Variables	270
5.5	ECUCPort	274
5.5.1	Overview	274
5.5.2	States of the ECUCPort	275
5.5.3	Usage of ECUCPort	277
5.5.3.1	Creation and Configuration	277
5.5.3.2	Accessing ECU Parameters	278
5.5.3.3	Getting the List of Variables of the ECUCPort	279
5.5.3.4	Manage ECU Memory Pages	280
5.6	EES Port	282
5.6.1	User Concept	282
5.6.1.1	General	282
5.6.1.2	Configuration and Execution of Electrical Errors	284
5.6.1.3	Triggers in EES	286
5.6.1.4	Electrical Errors	286
5.6.1.5	API	289
5.6.1.6	States of the EES Port	295
5.6.2	Usage of this Port	298
5.6.2.1	Port Creation and Configuration	298
5.6.2.2	Creating Error Configurations	299
5.6.2.3	Error Stimulation	302
5.6.2.4	Extension of Error Stimulation	302
5.6.2.5	Creating Error Objects	304
5.6.2.6	Loading Error Configurations from File	308
5.6.3	Special Hints	309
5.6.3.1	EES Hardware Limitations and Extensions	309
5.7	Network Port	310
5.7.1	User Concept	310
5.7.1.1	General	310
5.7.1.2	Network Port	311
5.7.1.3	States of the Network Port	312
5.7.2	Usage of the Network Port	314
5.7.2.1	Port Creation and Configuration	314
5.7.2.2	Object Model	315
5.7.2.3	Mapping to DBC and FIBEX	317
5.7.2.4	Navigating the Object Model	318
5.7.2.5	Send and Receive Frames	319
5.7.2.6	Send and Receive Signals	322
5.7.2.7	Capturing of Bus Signals	324
5.7.2.8	Capturing of Bus Frames	327
5.7.2.9	Replay of Bus Frames	330
5.7.2.10	Extending the CAN object model	331
6	Symbols and Abbreviated Terms	334
7	Bibliography	335



Appendix A.	Syntax of Watcher Conditions	336
A.1. Other restrictions	338
A.2. Syntax Overview.....	338
Appendix B.	Syntax of ConstSymbol Expressions	341
B.1. Other restrictions	343
B.2. Syntax Overview.....	343
Appendix C.	Key Value Pairs in CaptureResult MetaData	345
Appendix D.	Storage of Data in MDF4	346
D.1. Framework Measurement Data.....	346
D.1.1. Retriggered Measurements.....	347
D.2. Testbench Capture Data	348
D.2.1. Storage of Client Events	348
Appendix E.	Error Overview	350
Appendix F.	Deprecated Elements	392
Figure Directory		397
Table Directory		402

Foreword

The XIL standard is developed by ASAM to standardize the communication between test automation software and X-in-the-loop testbenches. It supports testbenches at all stages in the software development process – such as MIL¹, SIL², HIL³.

The application of the XIL standard allows users to combine the software and hardware tools independent of the vendor, thus enabling the reuse of the existing programmed tests for subsequent development stages (e.g. reuse of test cases for function models in software tests at a HIL simulator). This results in reduced costs for test programming and reduced training costs for testers.

The standard is divided into the following parts:

- Core specification
- Mapping rules for C# and Python
- C# implementation and usage examples
- C# test suite

The core specification includes a conceptual description (Programmer's Guide), the UML model (technology independent interface definition) of the Testbench and Framework API as well as various file format specifications.

The mapping rules define the transformation of the technology independent definitions into C# and Python specific representations. They include the C# interfaces and Python classes generated from the technology neutral UML model as well as standard implementations of a few, very fundamental interface classes (only C#).

The C# implementation and usage examples contain programming examples in C# which demonstrate how to implement and use the XIL interfaces. They also contain source code of common vendor independent base functions and components. The example implementations are intended to help test case developers and XIL vendors. They can be used and integrated into their products.

The C# test suite provides a set of test cases for the XIL Testbench API to help vendors and users to verify standard conformance of their C# implementations. The test cases are provided as C# source code. They can be automatically executed using the NUnit⁴ test framework and therefore easily integrated into automated build and test tool chains.

¹ Model-in-the-Loop

² Software-in-the-Loop

³ Hardware-in-the-Loop

⁴ NUnit: <http://nunit.org>

1 Introduction

1.1 OVERVIEW

ASAM XIL was developed to allow to exchange combinations of test automation software and test hardware.

The standard consists of two layers:

- Testbench API: This API separate the test hardware from the test software and allow a standardized access to the Hardware
- Framework API: This API allow the exchange of the test automation software by a standardized access point for test cases, so a decoupling of test cases from real and virtual test systems takes place

The testbench API covers the access to the following hardware:

- Model Access, provides access to the simulation model read and write parameters, capture and generate signals
- ECU Access, allows capturing and reading of measurement variables (M Port), is used for calibration (C Port)
- Stimulator functionality, i.e. the option to describe and evaluate real-time capable stimulation sequences,
- DIAG Access, i.e., the feature set to remote-control 3rd party ECU diagnostic tools, e.g. to use failure memory management options by diagnostic services.
- FIU Access, the capability to control electrical failure injection systems, to be used for short circuits, set to battery voltage or ground potential, open connection, etc. for ECU pins,
- Network Access, i.e. the option to read and write signal values on the automotive network systems, such as CAN, LIN, Flexray in a symbolic form.

The Framework serves as a central infrastructure to handle all variables related to their specific formats. Based on the framework the access to hardware is only realized by variables, which can be read, write, captured or used in the signal generation. These variables encapsulate the access to the testbench. The framework API offers the following possibilities:

- Mapping to exchange test case variables with port elements
- control of port life cycle
- configuration of used testbench ports

1.2 MOTIVATION

HIL technology has been developed over the years by only a few suppliers. Due to several reasons the architecture of these HIL systems was characterized by a direct rigid coupling of test automation software and used test hardware. Therefore test cases directly depend on the used test hardware. The end users perspective is, that not always the 'best' test software could be combined with the 'best' testing hardware.

Know-how could not be transferred from one testbench to the other. This resulted in additional training costs for employees. Switching to the newest testing technology and to a

new development process stage was difficult because of tool specific formats and test hardware compatibility issues. This led to the consequence that the base pre-condition for an exchange of test cases, e.g. between OEM and supplier, was not fulfilled.

The major goal of all standardization efforts is to allow for more reuse in test cases and to decouple test automation software from test hardware. Therefore the reuse of test cases within the same test automation software on different test hardware systems should be achieved. This will lead to a reduction of effort for test hardware integration into test automation software.

Software investments and test case development efforts can be long-term protected. End users may decide on test automation software system on a perspective of many years without the coercion of being coupled to one test hardware supplier.

Starting with version 2.0 the standard addresses two major parts of important functionality, that have not been covered in previous versions:

- *Framework*, which is completely new designed and contains broadly extended functionality such as variable measuring and mapping as well as managing of ports already known from HIL API 1.0.2. The *Framework* chapter deals with functionality, that is based on the
- *Testbench*, which comprises the ports, known from HIL API 1.0.2. The ports were extended slightly with respect to missing functionality in the previous standard version, such as configuration and initialization. In order to give test developers standardized access to CAN busses, the Network Port was completely new designed and added to the port family.

1.3 WHAT IS HARDWARE IN THE LOOP SIMULATION

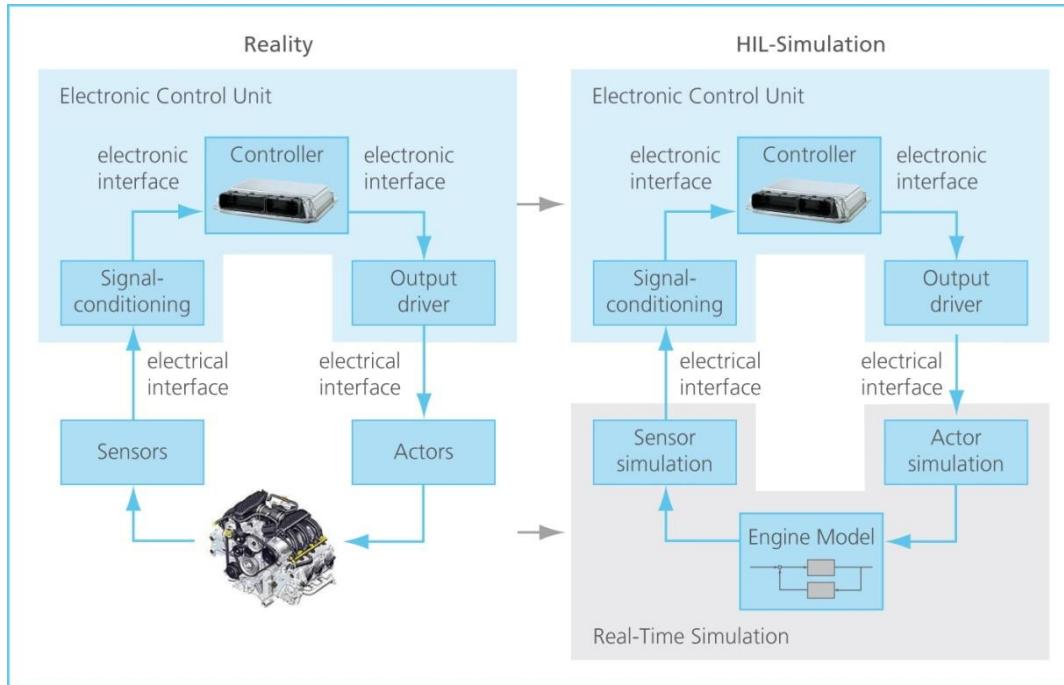


Figure 1: Principle of Hardware-in-the-Loop Simulation

Hardware-in-the-Loop (HIL) simulation has become a well-established verification technology applied in many ECU development projects today.

By means of XIL technology function tests can be shifted to earlier development stages to increase the maturity of new software and/or electronics components.

Cost and time expensive test drive cycles which have been performed in former times directly in vehicle or on conventional testbenches can be substituted by simulation based operations.

Tests of failure situations or tests of dangerous maneuvers can be shifted into the computer, at least in parts of the complete test program.

The major advantage is the capability to automate these testbenches. This allows to reproduce all test cycles and to operate these testbenches 24 h per day.

A closed control loop of today's automotive electronic system as shown in the left part of [Figure 1](#) (Controller, output driver, actors, plant, e.g. an engine, sensors and the input side signal conditioning) is substituted in parts. The electrical interfaces are retained. Sensors and actors are either replaced by full simulated versions or they are even attached as original physical load component in the testbench setup.

The plant part of the control loop, i. e. in this example the engine, is replaced completely by a simulation model, which can be calculated in the appropriate model precision in real-time.

1.4 TECHNICAL APPROACH

All interfaces of the XIL API can be created via factory⁵ methods. This provides the following advantages:

- All classes, represented by UML-based XIL API object model can be transformed into interfaces.
- This frees the test code from vendor-specific name space information, which would be necessary for calling constructors. Thus, code is free of vendor-specific information, what facilitates the exchange of tests among different testbenches.
- C# (this is the primary programming language provided by vendors in the market and covered by cross tests) does not provide constructors within its interface concept.

Factory methods may have arguments, in order to provide a comfortable way of interface creation, and to ensure, that all relevant data has been provided by the user to create the interface properly. However, not all factory methods do have arguments. In that case, the user is responsible for a proper configuration of the interface. Otherwise, an exception is thrown during runtime, indicating that the interface configuration is not valid.

1.5 TECHNOLOGY INDEPENDENCE

Today's XIL test automation systems use very different description technologies to define the test cases, e. g. the script language Python or C#.

Graphical or tabular based notations might also be used but underneath transform to the mentioned languages.

ASAMs goal has always been to define technology independent standards. Therefore the object model of the XIL API is defined in UML. This UML model is mapped to different

⁵ With XIL 2.0, all constructors have been replaced by factory methods.

programming languages. As a result of the mapping process, all XIL API classes are available in each of the supported programming languages either as interface definitions or using native data types. A mapping guideline is available for each programming language which describes how the UML model is converted to the programming language. Technology references are available for the programming languages C# and Python (for C# see [5] and for Python see [6]).

There are samples that explain how to use the interfaces described in the UML model. These samples can be found in a subdirectory within the technology reference directory. The separation of UML based reference model also allows adding other technologies later without the need to modify the API model itself.

2 Relations to Other Standards

2.1 BACKWARD COMPATIBILITY TO EARLIER RELEASES

This specification is based on ASAM XIL version 2.1. ASAM XIL version 2.2 is compatible with version 2.1. Compatibility means

- Clients using the XIL 2.1 API do not get broken if the used server implementations upgrade to XIL 2.2,
- Server implementations of XIL 2.2 are able to support clients based on XIL 2.1 if the rules listed below are adhered to.

Backward compatibility of minor releases is accomplished by the measures described in the following sub sections.

Changes to interfaces

- Minor XIL releases do not remove methods, properties or interfaces. Elements that become obsolete are preserved in the UML model as well as in the technology references. However, they are marked deprecated via a stereotype in the UML model.
- Minor XIL releases do not change the signature (i.e. name, parameters and return value) of methods or properties. In case of changes a new method / property is introduced and the existing method / property is marked deprecated via a stereotype in the UML model. If necessary to avoid name clashes, a sequential number is appended to the name of the new method / property (e.g. GetValue2).

Changes to file formats (schema files)

- In case of changes to a schema file by a minor XIL release, the original version is retained as part of the standard. However, it is marked as deprecated by moving it to a sub folder named “DEPRECATED” next to the new version of the schema file.

Required support of deprecated methods, properties and interfaces by XIL servers

- Upgrading an existing XIL server to a new XIL version must meet the following conditions for XIL compliance:
 - Existing implementations of API elements (i.e. methods, properties and interfaces) that are deprecated in the new XIL version must be preserved in the new server version. It is not allowed to remove implemented API elements because they are marked as deprecated.
 - The replacement of implemented, deprecated API elements must also be implemented. It is not allowed to provide only the implementation for the deprecated API elements.
- The first time implementation of an XIL server must meet the following conditions for XIL compliance:
 - At least the newest version of an API element (i.e. method, property or interface) must be implemented (their deprecated versions may be omitted).
 - It is not allowed to implement deprecated API elements only.

Required support of deprecated versions of XIL file formats by XIL servers

- For writing, the current version of the respective XIL file format must be used.
- On reading, the current and all deprecated versions of the respective XIL file format must be supported. It is not allowed to support the deprecated format versions only.

General remarks on deprecated elements

- An overview of all deprecated API elements (methods, properties, interfaces etc.) and their replacement can be found in [Appendix F](#).
- Deprecated API elements as well as deprecated schema file versions are likely to be removed with the next major version of the standard.
- Clients are recommended not to use deprecated elements any more and to switch to their replacement at the next opportunity.
- If supported by a technology reference, the deprecation marks in the generic UML model are converted to annotations at the generated interfaces (see also [\[5\]](#)). This should help users to detect the use of deprecated elements in their XIL clients based on the deprecation warnings generated by the compiler or IDE.

2.2 REFERENCES TO OTHER STANDARDS

In the XIL API the General Expression Syntax is used for defining watcher conditions. Not all possible functions and operators of the ASAM GES [\[2\]](#) are required. For Details, see Appendix [Syntax of Watcher Conditions](#).

Measurement data includes numerical data, e. g. within CaptureResult or RecordingResult objects. For streaming this data to file, ASAM XIL uses the definition of Measurement Data Format (MDF) [\[3\]](#).

2.3 VERSIONING

Versioning of the XIL standard is based on three numbers: <major>.<minor>.<maintenance>. These numbers also determine the interface version of the API defined by the standard.

The numbers of the version can be retrieved from the Testbench and Framework interface. [Table 1](#) gives the names of the respective properties and their values for XIL 2.2.

Table 1 Version Number and Version Properties

Number	Property name	Value
Major	MajorNumber	2
Minor	MinorNumber	2
Maintenance	RevisionNumber	0

The numbers returned by these properties reflect the XIL version that the respective implementation is based on. The implementation is responsible for returning the correct numbers.

3 General Concepts

The ASAM XIL standard specifies two major APIs, namely the Framework API and the Testbench API. The next section will explain the general architecture, which is the basis of the different interfaces.

3.1 XIL TEST SYSTEM ARCHITECTURE

The following picture gives an overview of a test system in general.

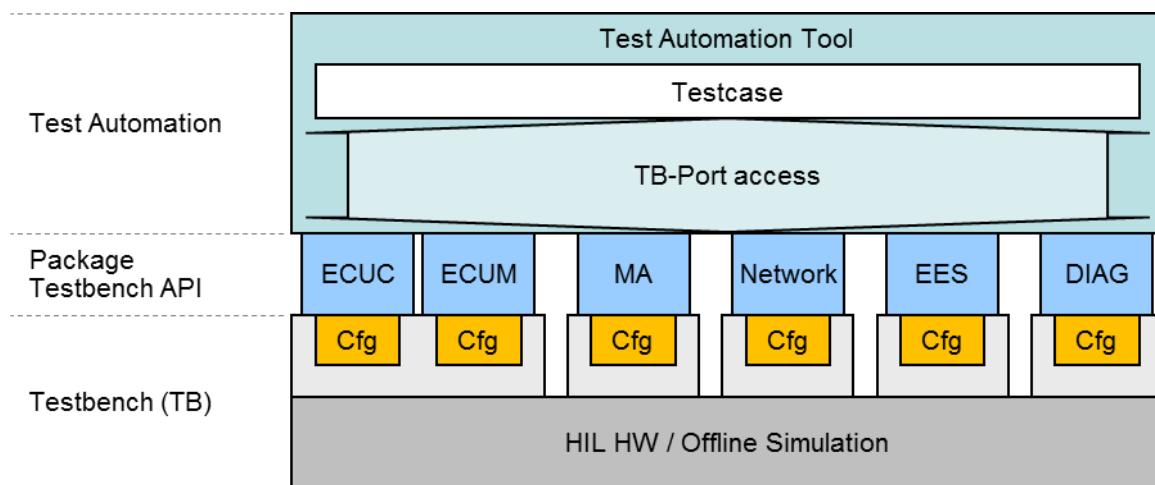


Figure 2: XIL Testbench API with direct port access

A typical XIL testbench consists of a system under test, which is a coupled system of ECUs, hardware components, and simulated parts of the system, depicted by the dark grey box at the bottom. The interaction between these parts is managed by different tools of different vendors, shown in the light grey boxes. These tools serve different purposes, such as accessing ECUs, or offer capabilities to interface bus systems, like CAN. The simulated parts might be executed by simulation tools or on the basis of compiled code. These tools have their proprietary means for configuration (orange blocks).

A first step of standardization has been achieved with ASAM HIL API 1.0.2 providing standardized interfaces to the different types of tools by means of specific Testbench ports in a Testbench API. ECUC port and ECUM port offer standardized access to ECUs via a calibration and a measurement interface. The MA port deals with model access, the DIAG port with a diagnosis interface. The EES port offers means for electric error simulation. XIL API 2.0 additionally introduces the Network port for accessing CAN networks.

Based on these testbench port interfaces, a test automation tool could utilize a standardized access to variables and signals on these ports without dealing with proprietary details of different vendors and their tools. However, in such a setup the test case has to implement start and shutdown of the testbench ports, which is highly dependent of the test system. In order to access variables and signals the test case has also to deal with port-specific addresses and data types.

These drawbacks have been resolved with XIL API 2.0. The testbench port access is still available (see right side of [Figure 3](#)) for compatibility reasons and in order to provide some detailed functionality, that is not provided by the framework.

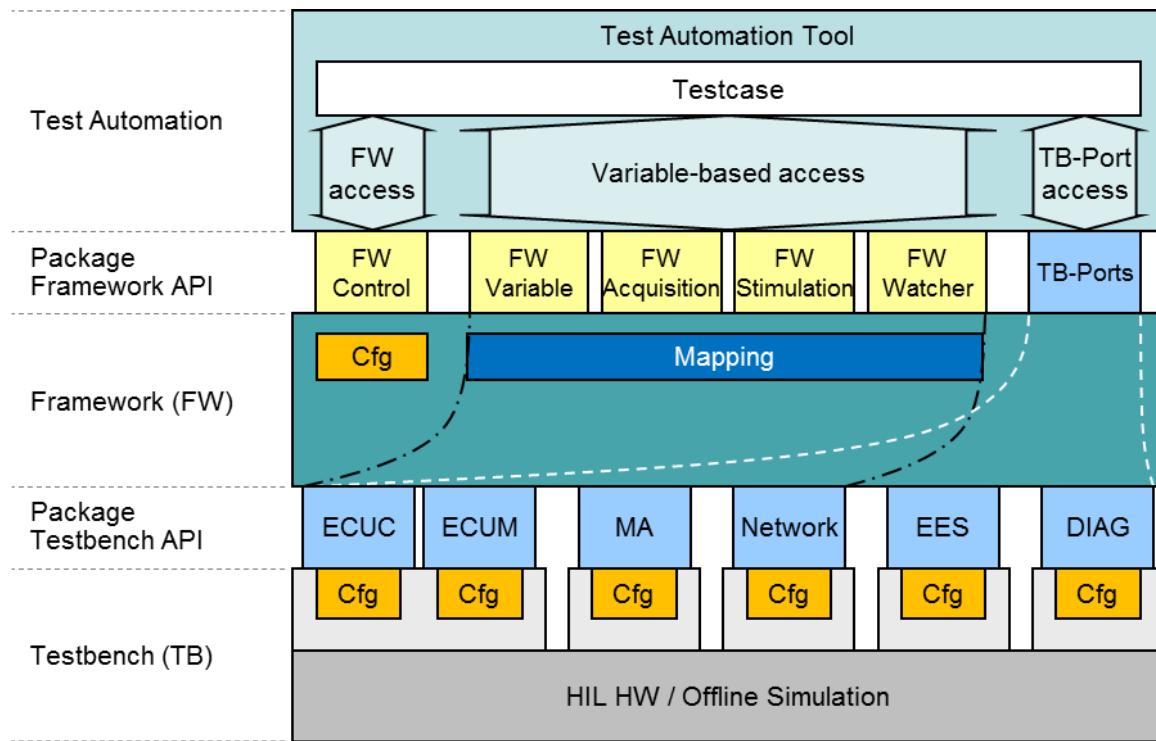


Figure 3: Test system with XIL Framework API and XIL Testbench API

The major benefit of XIL API 2.0 is to introduce port independence to test cases by using an object-oriented access to variables on Testbench ports. This kind of port abstraction is provided by the framework layer between test automation and testbench. According to some testbench specific configuration within the framework, the different ports are started and shutdown in a specified order.

At the present time it is a challenging task for test developers to achieve effective decoupling between test cases and testbenches in order to improve test reuse. The test developer's work will be greatly facilitated by one of the new XIL API 2.0 framework's basic features: mapping. This provides the standardized assignment of mapped values:

- Abstract identifiers for variables on test case site and corresponding concrete identifiers for variables on simulation model site,
- physical units on the test case and on the testbench site,
- data types on the test case and on the testbench site.

Thus, mapped values can differ during the development process. The user just has to adjust the mapping – the test case remains unaffected. Mapped values such as the data type or physical unit of a vehicle's velocity may change on the testbench side because different precisions or different simulation model suppliers are involved in the development process while the same test case implementation is being reused.

Based on a **MappingInfoAPI** a testcase can retrieve all information, which is part of the mapping definition. Based on the variable objects, the framework also provides objects for port-independent signal recording, signal generation, and event watching and triggering. A more detailed description is given in the following sections.

3.2 OVERVIEW FRAMEWORK

3.2.1 GENERAL

The Framework class is the central class for managing the set of testbench ports and their configuration, mapping information as well as overall recording and stimulation. The Framework class provides factories for other framework related objects (see [Figure 4](#)).

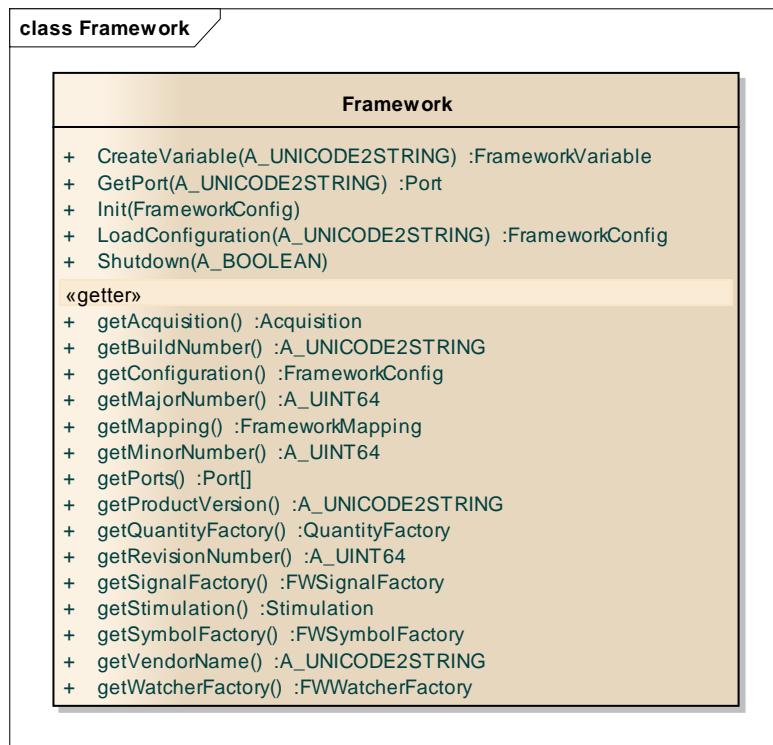


Figure 4: The Framework class

3.2.2 INITIALIZATION

The XIL API 2.0 framework simplifies initialization of the testbench ports. The complete test system is initialized based on a configuration, which can be loaded from file using the `LoadConfiguration()` method. It references other mapping and port configuration files in order to setup the complete framework with all relevant ports. For more details see the section [Configuration](#). The `Init()` method is called in order to initialize all ports according to the previously loaded configuration in a specified order. After successful initialization the test automation system is ready to perform framework variable accesses, or more precisely, all testbench ports are in their target state as specified in the configuration.

3.2.3 FRAMEWORK VARIABLES

The most fundamental concept of XIL API 2.0 is to provide framework variable objects which abstract from the Testbench ports and memory addresses of variables. The following figure shows the `FrameworkVariable` class.

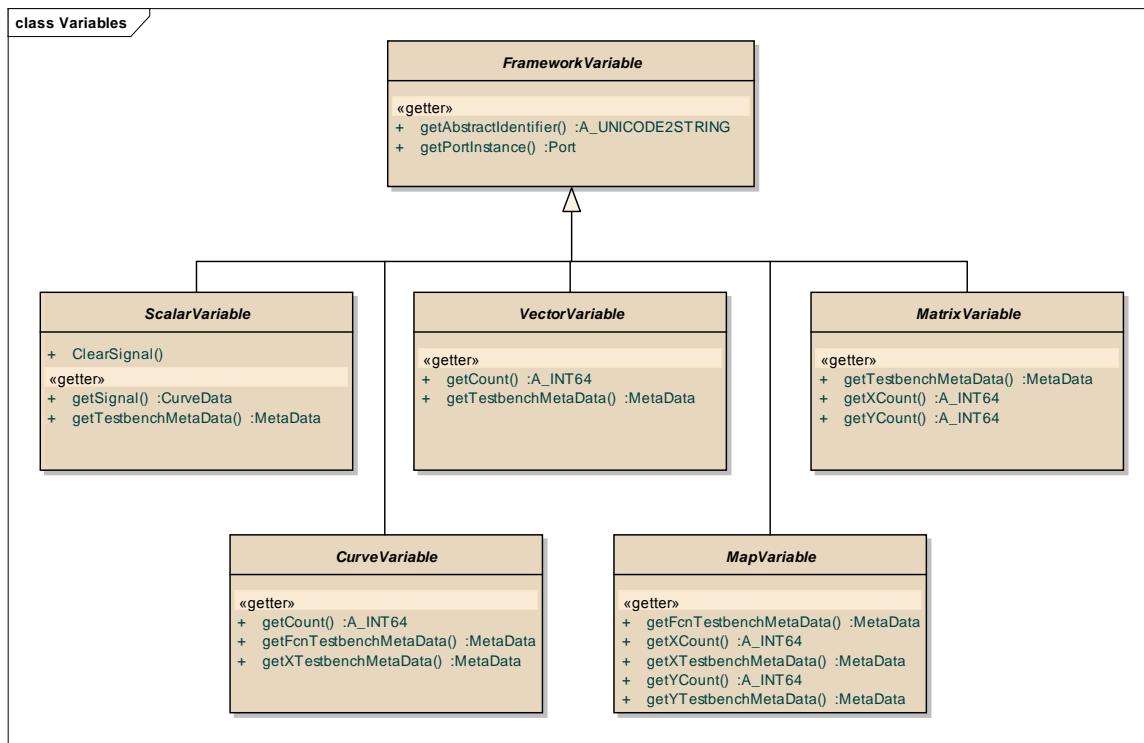


Figure 5: The FrameworkVariable class

Framework variables are created using the `Framework.CreateVariable()` method using an `AbstractIdentifier` string. This string has to be defined in the mapping configuration, where it is related to a specific testbench port. The mapping also contains meta data about unit and data type of the variable, as well as unit and data type of the testbench port value, which the variable represents. This allows an automatic unit and type conversion at any framework access.

The value of a framework variable is encapsulated by a `Quantity` object. A quantity has a physical dimension, e.g. length, a unit, e.g. meter, and a value.

Framework variables exist as scalar variables as well as vector, matrix, curve, and map variables. More details on methods and framework mechanisms utilizing framework variables can be found in section [Framework Variables](#).

Note: In XIL API 2.0 framework variables cannot be used for the `DiagPort` and `EESPort`.

3.2.4 ACQUISITION AND STIMULATION

Data acquisition is an important part of test systems. With the `Acquisition` object (see [Figure 6](#)), which can be obtained from the `Framework`, the data flow from the testbench ports to the framework is configured and controlled. The configuration is based on framework variables, which carry the relation to the testbench port and information about unit and data type.

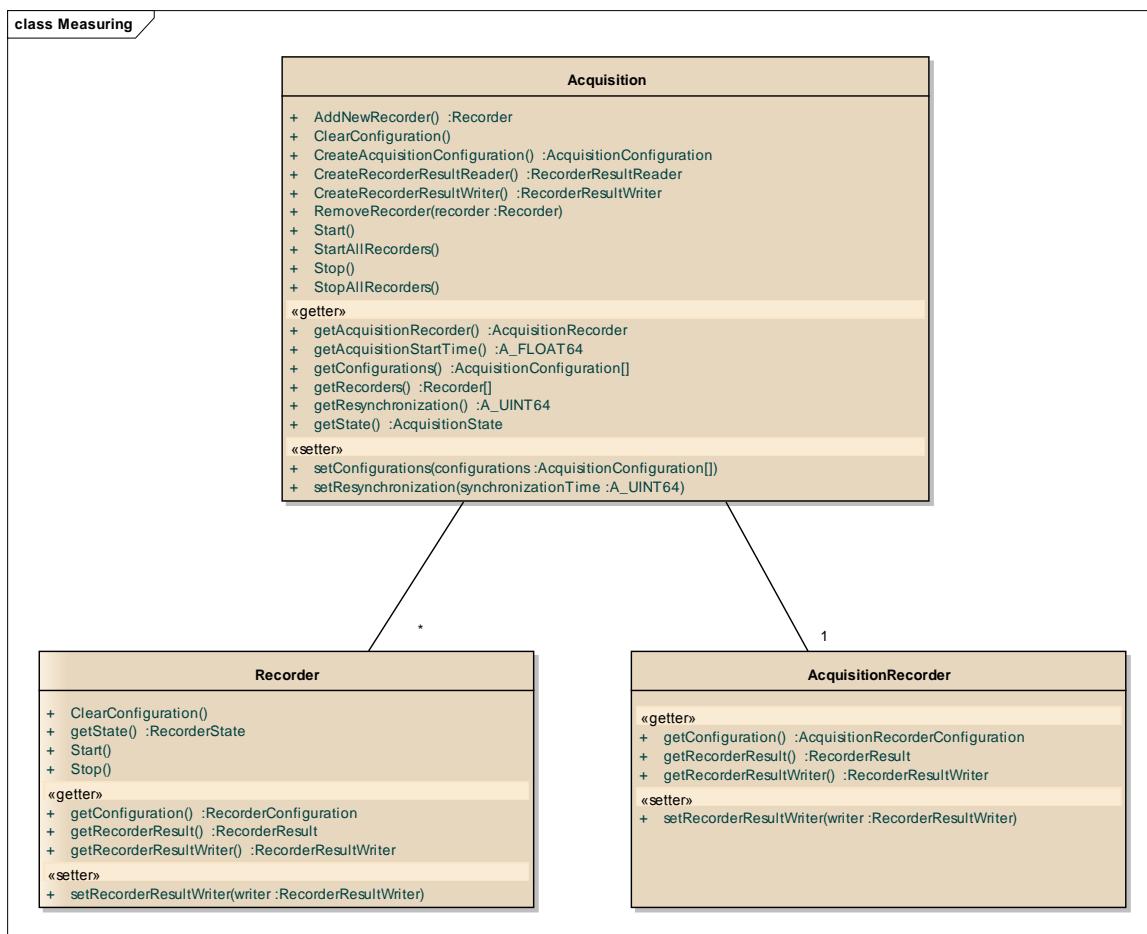


Figure 6: The Acquisition class

Several recorders can be created. In order to reduce the amount of data, which is transferred to the host PC, the measurement process can be configured via triggers and a downsampling factor. A recorder can store signals from different ports. A RecorderResult can be held in memory or stored to file, e.g. in MDF4 format. Details are given in section [Measuring](#).

In a similar way, stimulation is configured. [Figure 7](#) shows the Stimulation class.

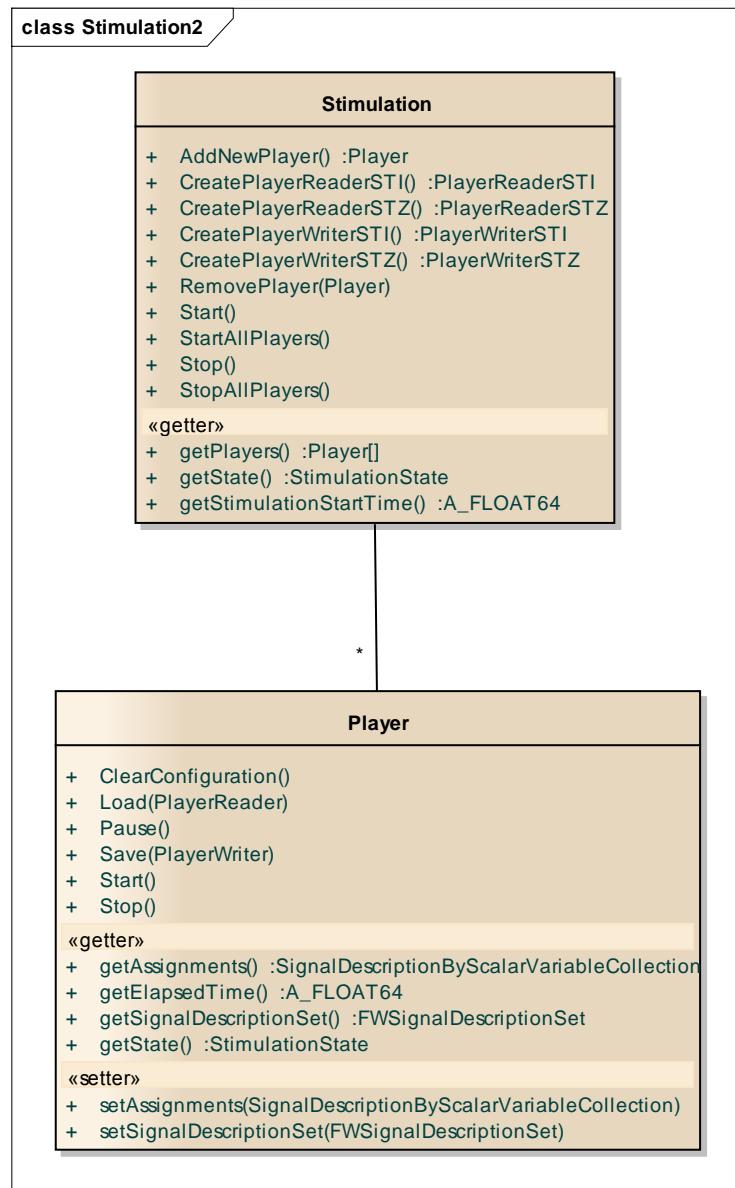


Figure 7: The **Stimulation** class

Several Player instances can be configured. Signal descriptions are similar as known from HIL API 1.0. The information about the specific port variable is again taken from the framework variables, which have been assigned in the configuration. A player can also use a previously recorded signal from file for stimulation. For details refer to section [Stimulation](#).

Watcher objects are used to generate trigger events, e.g. for recordings. The trigger conditions are defined based on General Expression Syntax (GES) strings. In order to refer to framework variables, their `AbstractIdentifier` is used. Watchers can be defined for variables of different testbench ports.

3.3 OVERVIEW TESTBENCH

3.3.1 GENERAL

The testbench API covers access to the Testbench ports with their hardware and software components.

XIL API provides vendor independent access to the functionalities of a XIL simulator via port interface definitions for the different kinds of ports. Each tool vendor can provide an implementation of these interfaces, which is specific for his tool set. Thus, the user of the XIL Testbench API gains standardized access to the tools of different vendors.

The XIL Testbench API covers the functional areas

- Model access,
- ECU access,
- Network access,
- Diagnostics access, and
- Electrical error simulation.

Each of these functional areas is represented by one or several ports. In contrast to HIL API 1.0.x, in XIL API 2.0 the port initialization is supported by means of standardized configuration methods and objects.

Note: Usually, initialization is performed by the framework. However, testcases can directly program initialization sequences.

3.3.2 PORTS OF THE XIL TESTBENCH API

The following class diagram shows the different XIL Testbench ports.

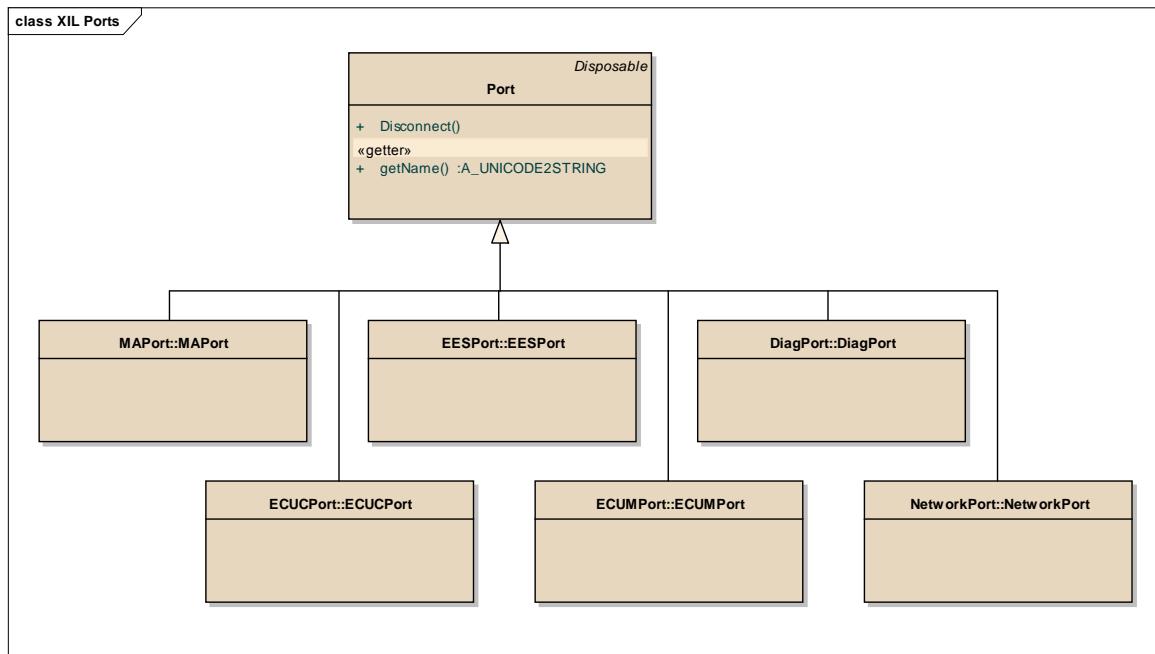


Figure 8: XIL Ports

The Model Access port (`MAPort`) provides access to the simulation model. It is possible to read and to write data and to capture and to generate signals.

The Diagnostic port (`DiagPort`) communicates with a diagnostic system to read data via diagnostic services from an ECU or Functional Group.

The EES port (`EESPort`) controls electrical error simulation hardware. It allows to set different types of errors.

The ECU Calibration port (`ECUCPort`) communicates with an electronic control unit system. It provides access to ECU internal calibration values.

The ECU Measurement port (`ECUMPort`) communicates with an electronic control unit system. The ECUM port allows to capture and to read measurement variables.

The Network Access port (`NetworkPort`) provides access to networks, such as CAN or FlexRay.

Note: Please note that in XIL API 2.0 only CAN networks are supported.

Support for FlexRay or other buses will be part of a future version of XIL API.

3.4 ASAM DATA TYPES

ASAM data types are used in the entire UML model. These define the type system for all scalar basic data types. All complex data types use these base types (e.g., see chapter [ValueContainer](#) and chapter [Quantities](#)). More information about the ASAM data types is available in [1].

The following basic ASAM data types are used in the XIL API UML model:

- `A_BOOLEAN`
- `A_BYTEFIELD`
- `A_FLOAT64, A_FLOAT32`
- `A_INT64, A_INT32, A_INT_16, A_INT8`
- `A_UINT64, A_UINT32, A_UINT_16, A_UINT8`
- `A_UNICODE2STRING`

The ASAM data types are included in the model in the sub package '`XILTypes.ASAMDataTypes`'.

3.5 INSTANCE CREATION

In order to maximize independence of test cases from test systems it is not sufficient to just standardize interfaces of a test system. Rather there should be a generic way to obtain corresponding instances from any vendor providing a XIL implementation.

Therefore the factory approach is applied. That means instances are obtained from methods of already existing objects. For example a `MAPort` instance is created by invoking the `CreateMAPort` method on an object implementing the `MAPortFactory` interface. In turn such a `MAPortFactory` object can be obtained from a `Testbench` object providing a corresponding property.

Following this approach one can obtain instances for any XIL interface and of any vendor. Preconditions are the existence of the respective vendor's `Testbench` or `Framework` object and the implementation of the desired interface by that vendor.

Methods for instance creation are called factory methods. Classes that exclusively serve the instance creation of other objects are called factory classes. Since factory classes and methods are also standardized, instance creation is possible in a vendor independent manner.

Creation of the top-level factories `Testbench` and `Framework` is done based on descriptive information that vendors must provide together with their XIL implementations. This description (Implementation Manifest) is standardized to enable creation in a vendor independent manner. The respective specification can be found in [Implementation Manifest File](#). There are two special interfaces (`TestbenchFactory` and `FrameworkFactory`) for utilization of the descriptive information. They are explained in [Framework and Testbench Factory](#).

3.5.1 IMPLEMENTATION MANIFEST FILE

A standard-compliant implementation must comprise a description file providing information for locating and instance creation of the vendor specific `Testbench` and/ or `Framework` class. This file is called Implementation Manifest. Its content and storage location are standardized as described in the following chapters.

3.5.1.1 FILE CONTENT

The manifest file's structure must obey the schema definition in the `ImplementationManifest.xsd` that is part of the standard. The manifest is actually a list of references to C# or Python classes implementing either the `Testbench` or the `Framework` interface. These references are represented by XML elements. As shown in [Figure 9](#) there are four different reference types corresponding to the different interfaces (`Testbench` or `Framework`) and the different implementation technologies (C# or Python). For instance a `NetTestbenchImplementation` element is used to refer to a C# class implementing the `Testbench` interface, whereas `PyFrameworkImplementation` elements are used for Python classes implementing the `Framework` interface.

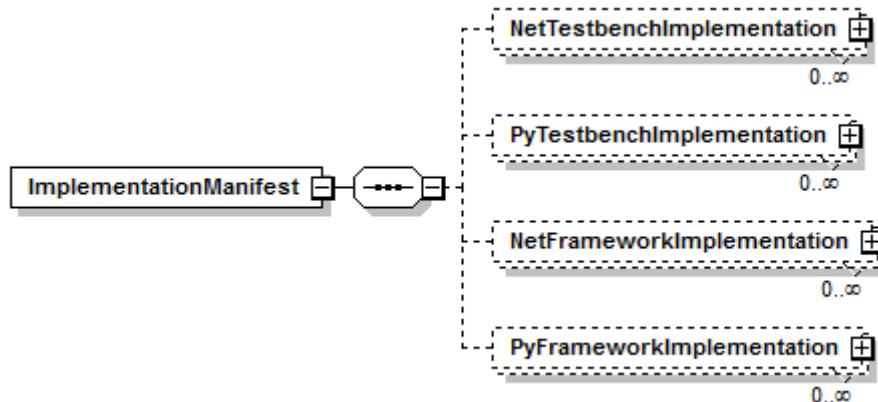


Figure 9: Implementation Manifest files contain a list of elements referring to C# or Python classes that implement the `Testbench` or `Framework` interface

Basically a manifest file may contain any number of references to different Testbench or Framework implementations. These references are uniquely identified by their type and their attribute assignments. The attributes of references to C# classes can be gathered from [Figure 10](#) and [Figure 11](#). References to Python classes have the same attributes (see [Figure 12](#) and [Figure 13](#)).

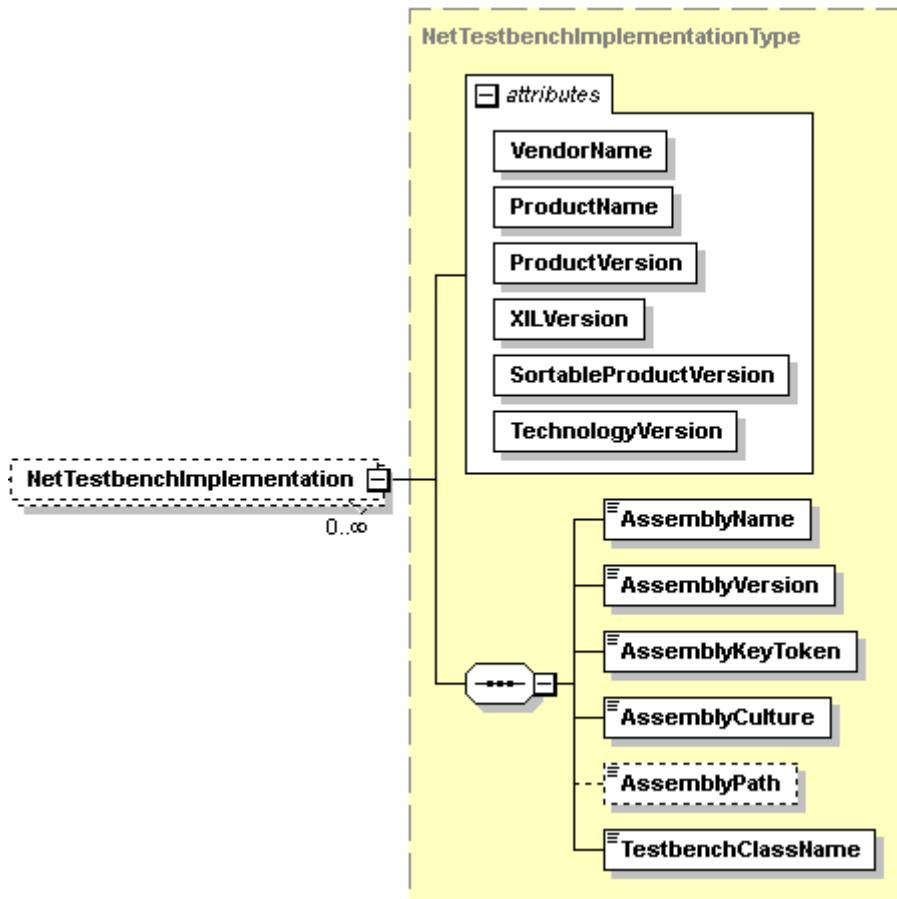


Figure 10: **NetTestbenchImplementation** element of the Implementation Manifest

Commonly XIL implementations are delivered as part of software products. The reference attributes **VendorName**, **ProductName** and **ProductVersion** identify such a software product by its name, version and vendor. The **XILVersion** attribute specifies the XIL standard's version the respective implementation is based on. And finally reference type and the attribute **TechnologyVersion** determine the implementation technology (C# or Python) and technology version of the respective XIL implementation.

The attribute "SortableProductVersion" is an internal attribute which, in the case of several product versions, serves to chronologically sort or determine the most recent product version. The client is therefore able to sort the entries belonging to a product according to product actuality independent of the product versioning scheme of the respective manufacturer and to automatically use the latest product version installed on the respective test stand. The **SortableProductVersion** is a string that consists of three non-negative integer numbers separated by dot (e.g. "3.5.1") and denoted as <major>, <minor> and <maintenance> numbers.

Note: The manufacturers are free in the assignment of the <major>, <minor> and <maintenance> numbers. Each vendor must, however, ensure that all of the

TestbenchImplementation entries in its product X / all FrameworkImplementation entries of its framework follow this rule: If ProductVersion V2 is larger than ProductVersion V1, the corresponding SortableProductVersion SV2 is larger than the SortableProductVersion SV1 and vice versa.

Note: To compare the SortableProductVersion, the numeric values of the <major>, <minor> and <maintenance> numbers are compared with each other with decreasing priority.

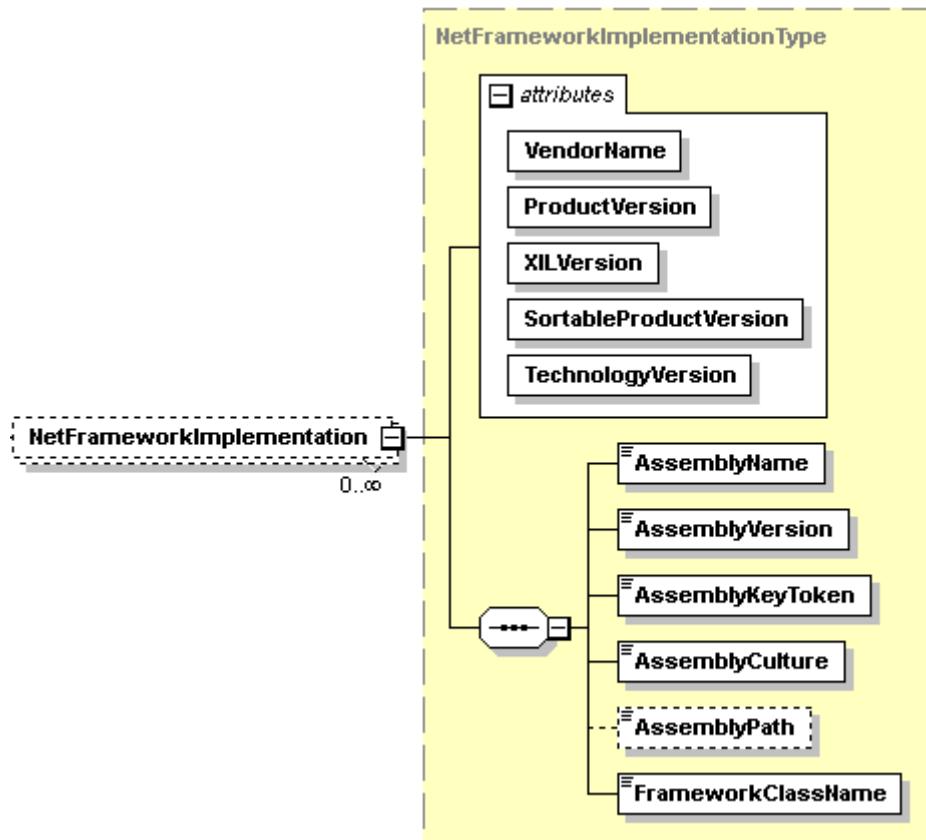


Figure 11: NetFrameworkImplementation element of the Implementation Manifest

The content of a reference element provides technical information on code module and class name of the respective Testbench or Framework implementation. This information is required to find and load the module and create Testbench or Framework instances via reflection mechanisms. Since this kind of information highly depends on the implementation technology there are different reference elements for C# and Python classes.

References to C# Testbench classes comprise the following elements as shown in [Figure 10](#). The elements AssemblyName, AssemblyVersion, AssemblyKeyToken, AssemblyCulture give the so called strong name of the assembly containing the class that implements the Testbench interface. The element TestbenchClassName holds the full qualified name of that class.

It is recommended to install the respective assemblies in the global assembly cache (GAC). So they can be found and loaded by means of their strong name. If not installed in the GAC, the assembly location (file system folder) must be specified in the manifest file by the optional element AssemblyPath.

[Figure 11](#) shows the elements of references to Framework classes implemented in C#. The only difference to Testbench pointers is the `FrameworkClassName` element which replaces the `TestbenchClassName` element.

References to Testbench classes implemented in Python comprise the following elements as shown in [Figure 12](#). `TestbenchClassNameModule` specifies the full qualified name of the Python module containing the Testbench class. And `TestbenchClassName` gives the name of that class. There is a third, optional element `LibPaths` holding the list of file system folders where the Python modules are stored.

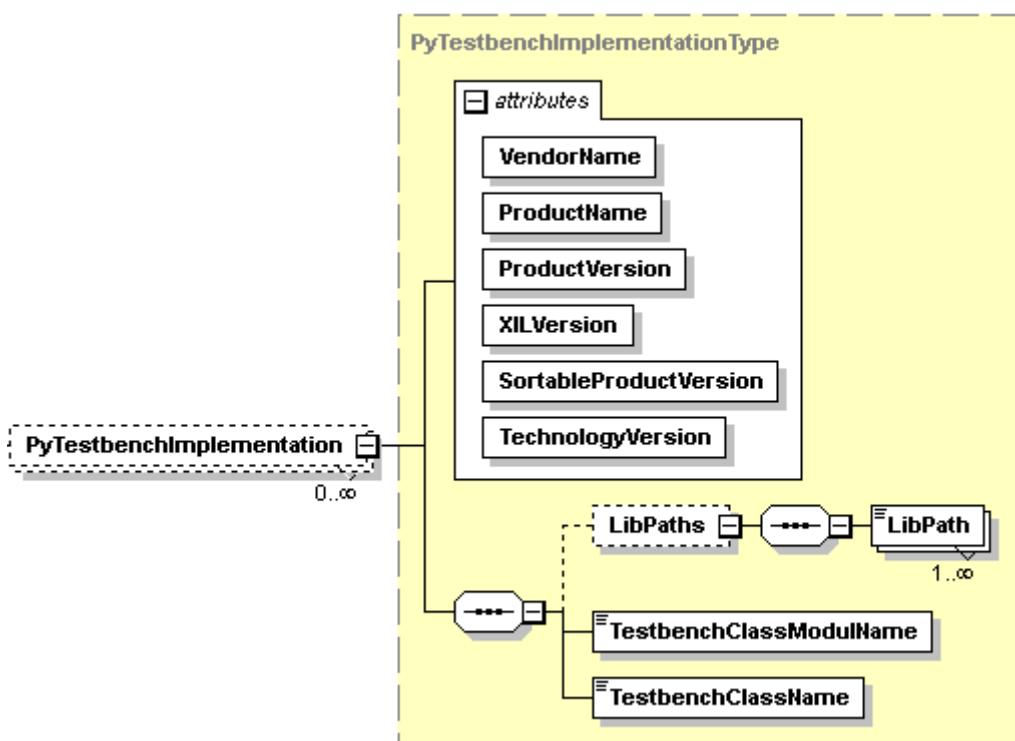


Figure 12: `PyTestbenchImplementation` element of the Implementation Manifest

References to Framework classes are quite similar (see [Figure 13](#)). They just have their elements giving the module and class named `FrameworkClassNameModule` and `FrameworkClassName`.

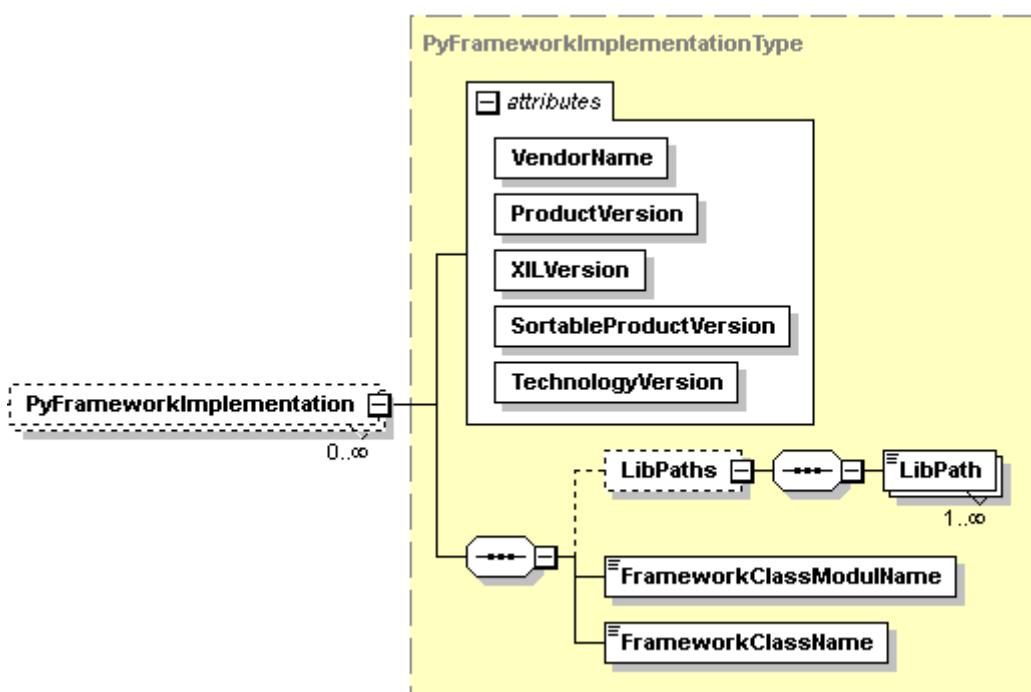


Figure 13: PyFrameworkImplementation element of the Implementation Manifest

3.5.1.2 FILE NAMING CONVENTION AND STORAGE LOCATION

Principally each Implementation Manifest file may contain any number of references to different Testbench and Framework classes. These can be classes of different products or even different vendors or implementation technologies. However it is recommended that each vendor provides one (or more) separate Implementation Manifest file(s) for its XIL implementation(s). So it is not necessary to merge and store manifest information of different vendors in a single file. But it is up to the client to support several manifest files.

The name of each Implementation Manifest file must start with the name of its vendor to avoid name clashes between different vendors. Vendors distributing two or more manifest files (e.g. with different products or product versions) must ensure unique names for their files. Implementation Manifest files must have the extension “imf”.

The Implementation Manifest file(s) of a XIL implementation must be copied to a specific file system folder during installation. The following [Table 2](#) specifies the storage location on Microsoft operating systems.

Table 2 Storage locations for Implementation Manifest files (environment variables are enclosed by % signs)

Operating System	Folder
Microsoft Windows XP	%ALLUSERSPROFILE%\ASAM\XIL\Implementation
Microsoft Windows Vista	%PROGRAMDATA%\ASAM\XIL\Implementation
Microsoft Windows 7	%PROGRAMDATA%\ASAM\XIL\Implementation
Microsoft Windows 8	%PROGRAMDATA%\ASAM\XIL\Implementation
Microsoft Windows 10	%PROGRAMDATA%\ASAM\XIL\Implementation

It is up to the vendor to ensure manifest files are copied to the proper location.

3.5.2 FRAMEWORK AND TESTBENCH FACTORY

The standard defines two interfaces that serve the creation of Testbench and Framework instances in a vendor independent manner. Testbench instances can be created by calling one of the methods `CreateVendorSpecificTestbench` or `CreateVendorSpecificTestbench2` of the `TestbenchFactory` (see Figure 14). `CreateVendorSpecificTestbench` selects the desired Testbench implementation based on vendor name, product name and product version. If there are multiple implementations matching the search criteria the implementation whose XIL version equals the `TestbenchFactory`'s XIL version is chosen. If none of the matching Testbench implementations has that XIL version an implementation of a different XIL version is chosen in an undefined way. `CreateVendorSpecificTestbench2` provides an extra parameter that allows the client to explicitly specify the XIL version of the Testbench implementation to be instantiated. However, passing a XIL version different from the XIL version of the used `TestbenchFactory` may cause problems because of version compatibility issues. Depending on XIL version and technology it may not even be possible to instantiate a Testbench implementation of a different XIL version or, after instantiation its functionality may be truncated. Please refer to the technology references for further information and possible solutions.

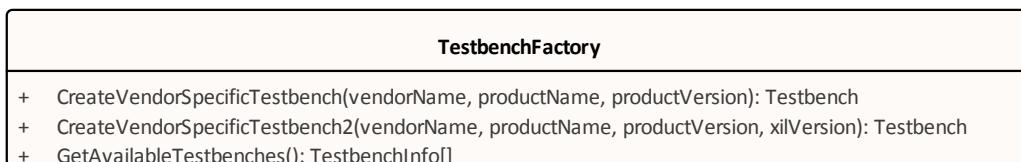


Figure 14: TestbenchFactory class

Framework instances can be obtained by calling one of the methods `CreateVendorSpecificFramework` or `CreateVendorSpecificFramework2` of the `FrameworkFactory` (see Figure 15). `CreateVendorSpecificFramework` selects the Framework implementation based on the vendor name and product version. If there are multiple implementations matching the search criteria the implementation whose XIL version equals the `FrameworkFactory`'s XIL version is chosen. If none of the matching Framework implementations has that XIL version an implementation of another XIL version is chosen in an undefined way. `CreateVendorSpecificFramework2` provides an extra parameter that allows the client to explicitly specify the XIL version of the Framework implementation to be instantiated. Please be aware of the same possible version compatibility issues as described for `CreateVendorSpecificTestbench2` above.

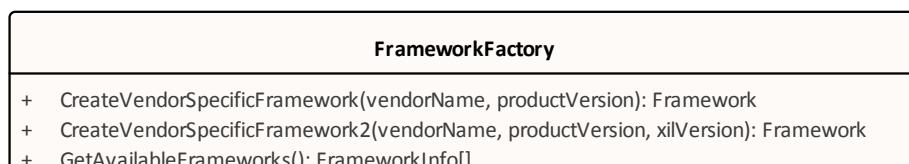


Figure 15: FrameworkFactory class

There is a reference implementation of the `TestbenchFactory` and `FrameworkFactory` interface that comes with the XIL standard (C# only). The reference implementation relies on the Implementation Manifest files (see [Implementation Manifest File](#)) in order to find and create instances of vendor specific `Testbench` and `Framework` classes. Instance creation of `TestbenchFactory` and `FrameworkFactory` implementations takes place by usual programming language mechanisms (e.g. calling the `new` operator). Please, consult the technology references for information and examples on using the reference implementation.

4 Framework

4.1 CONFIGURATION

The definition of items (test quantities, units, testbench ports etc.) used by test cases and created or managed by the framework is called configuration. This includes the items' settings and relations. The configuration is stored in several XML files that have to be loaded into the framework. [Figure 16](#) outlines the different configuration files and their relations.

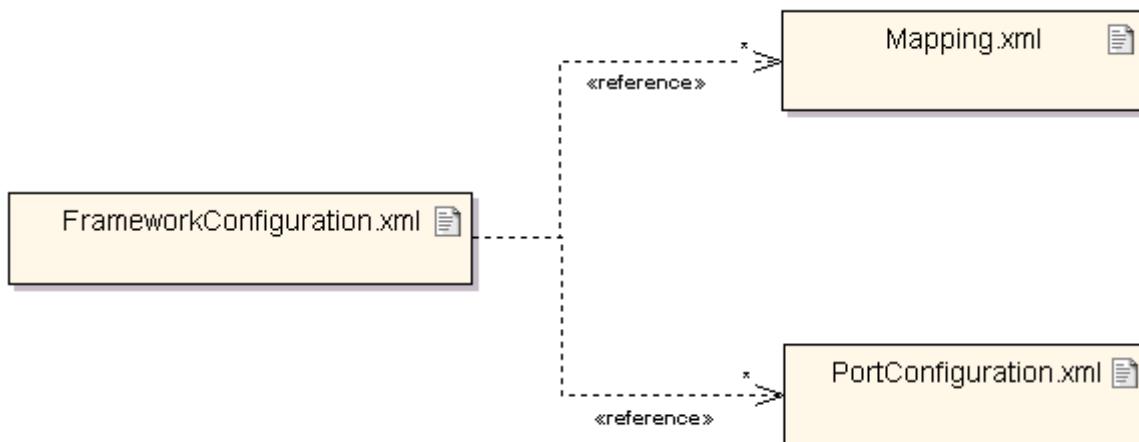


Figure 16: Framework configuration files

The framework configuration file is the central configuration file. It references a set of mapping files and a set of port configuration files. Its file structure must obey the schema definition in the FrameworkConfiguration.xsd that is part of the standard.

The mapping files particularly contain information on the available framework variables, e.g. data types, unit definitions, associations to specific testbench port labels and conversions. The mapping file format is standardized, for a detailed description see chapter [Mapping](#).

The port configuration files encapsulate all the port specific configuration settings required to setup the testbench ports. These files are not standardized, since the port settings are specific to the respective tool or port provider.

The following chapters give a detailed description of the framework configuration file, show how to initialize the framework with a certain configuration and explain how to customize the management of testbench ports by the framework.

4.1.1 FRAMEWORK CONFIGURATION FILE

The framework configuration file is an XML file comprising two top-level elements as shown in [Figure 17](#). The `MappingFileList` declares the mapping files to be used. The `PortDefinitionList` provides information on the testbench ports to be created and configured by the framework (e.g. `MAPorts` or `ECUCPorts`).

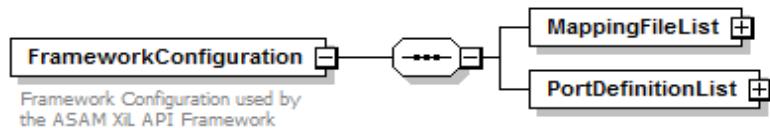


Figure 17: Top level elements of framework configuration file

The `MappingFileList` is a list of file references. Its structure is shown in Figure 18. Each reference is represented by a `MappingFile` element whose content is a file path. The `Name` attribute of the `MappingFile` element can be used to qualify content or purpose of the respective mapping file (e.g. unit definitions, project or test stand specific mappings). The `Name` attribute is not interpreted by the framework.

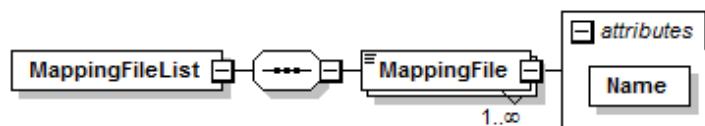


Figure 18: Structure of mapping file references in the framework configuration file

Purpose of the `PortDefinitionList` element is to define count, name, type and configuration of the testbench ports to be managed by the framework. The `PortDefinitionList` contains exactly one `PortDefinition` element for each testbench port. However there are different types of `PortDefinition` elements as shown in Figure 19. For each port type (e.g. `MAPort`) there is a corresponding `PortDefinition` type (e.g. `MAPortDefinition`).

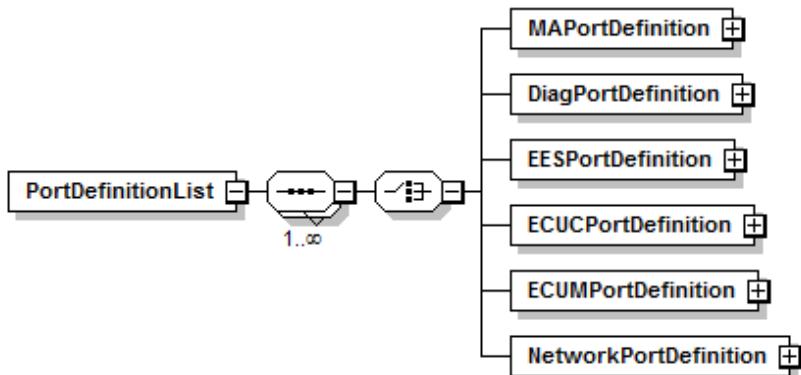


Figure 19: Specific port definition types in the framework configuration file

A `PortDefinition` has three functions. First of all it declares the type of port to be created, assigns a unique port name (e.g. for reference by the mapping) and selects the specific port implementation. The last aspect is crucial since several XIL implementations (of different vendors) may be installed on the same host. The second function of a `PortDefinition` is the assignment of the proper port configuration. The port configuration may depend on test stand, device under test or test project. Finally, `PortDefinitions` control the startup and shutdown sequence applied to a number of ports and determine the ports' target states to be established.

The different types of PortDefinition elements are very similar. They only differ in their domain of the TargetState element. That's why structure and semantics are explained on the example of the MAPortDefinition (see [Figure 20](#)).

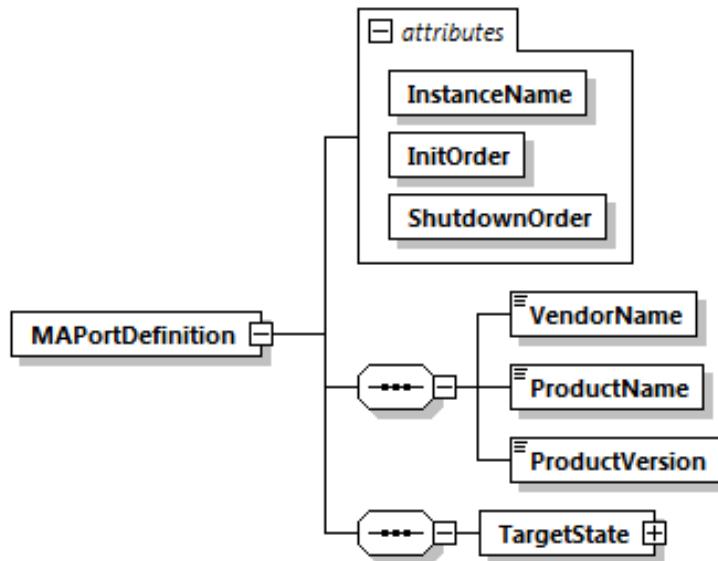


Figure 20: Structure of MAPortDefinition elements in the framework configuration file

As already mentioned the type of PortDefinition determines the type of port to be created (e.g. a MAPortDefinition creates a MAPort). A PortDefinition has four child elements. The triple of VendorName, ProductName and ProductVersion element selects a specific XIL implementation for the port creation (resolution to implementation classes is possible by means of Implementation Manifest files, see [Implementation Manifest File](#)). The fourth element TargetState provides the port state to be set up by the framework in the process of port configuration and startup. It also provides the parameter values needed to be passed to the port instance during setup. See [Figure 21](#) for an example of a TargetState structure.

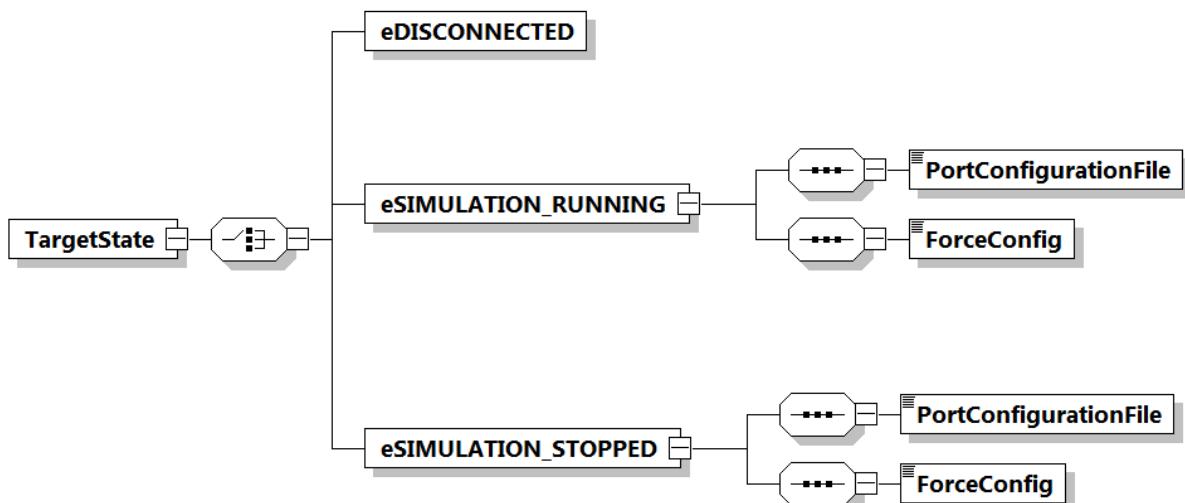


Figure 21: Structure of the TargetState element of the MAPortDefinition

The TargetState element has exactly one child element whose name (and type) determines the target state. The possible target state names depend on the port type and hence on the type of the PortDefinition element. The parameter values required to establish the respective target state are represented by child elements of the one named after the target state. The set of these parameter child elements depends on the target state and may also be empty. Please refer to [Table 3](#) for an overview of the target states grouped by PortDefinition type and their corresponding parameter sets.

Table 3 TargetState elements and contained parameters depending on the PortDefinition type in the framework configuration file

PortDefinition type	Permitted target states	Parameters
MAPortDefinition	eDISCONNECTED	---
	eSIMULATION_STOPPED	PortConfigurationFile
	eSIMULATION_RUNNING	ForceConfig
DiagPortDefinition	eDISCONNECTED	---
	eCONNECTED	PortConfigurationFile
EESPortDefinition	eDISCONNECTED	---
	eCONNECTED	PortConfigurationFile
	eDOWNLOADED	PortConfigurationFile
	eACTIVATED	ErrorConfigurationFile
ECUCPortDefinition	eDISCONNECTED	---
	eOFFLINE	PortConfigurationFile
	eONLINE	PortConfigurationFile LoadingType
ECUMPortDefinition	eDISCONNECTED	---
	eMEASUREMENT_STOPPED	PortConfigurationFile
	eMEASUREMENT_RUNNING	
NetworkPortDefinition	eDISCONNECTED	---
	eSTOPPED	PortConfigurationFile
	eRUNNING	

Each PortDefinition has three attributes. The InstanceName attribute determines the name to be assigned to the port instance on creation. This name must be unique across all PortDefinitions in a framework configuration file. It's up to the framework to ensure this uniqueness. The InstanceName is used as port reference, e.g. in mapping files or to obtain created port instances from the framework.

The InitOrder and ShutdownOrder attributes determine the start and shutdown sequence applied by the framework to the set of ports. Both attributes must be assigned a nonnegative integer value. Ports with smaller InitOrder are started before ports with greater InitOrder. Semantics of the ShutdownOrder is inverse, i. e. ports with greater ShutdownOrder are stopped before ports with smaller ShutdownOrder. The start and shutdown sequence of ports with identical InitOrder or ShutdownOrder is not defined.

4.1.2 CONFIGURING THE FRAMEWORK

Applying a configuration means setting up testbench ports and loading the required mapping information according to the framework configuration file. Therefore this is a precondition to executing test cases using the framework.

Configuration is a two-stage process (see [Figure 22](#)). The first step is loading the framework configuration file. Therefore the Framework interface provides a LoadConfiguration method that returns a FrameworkConfig object. LoadConfiguration shall check consistency of the framework configuration file and the referenced mapping files. It always returns a new FrameworkConfig object and does not invalidate existing FrameworkConfig objects. The currently active configuration is not modified (e.g. by creating or deleting ports). So LoadConfiguration may be called any number of times and independently from any other methods of the Framework interface.

The second configuration step consists of creating, configuring and starting the testbench ports as defined in the configuration as well as applying the mapping information. This second step is also called initialization. Therefore the Framework interface provides the Init method, that receives the FrameworkConfig object created in step one as parameter.

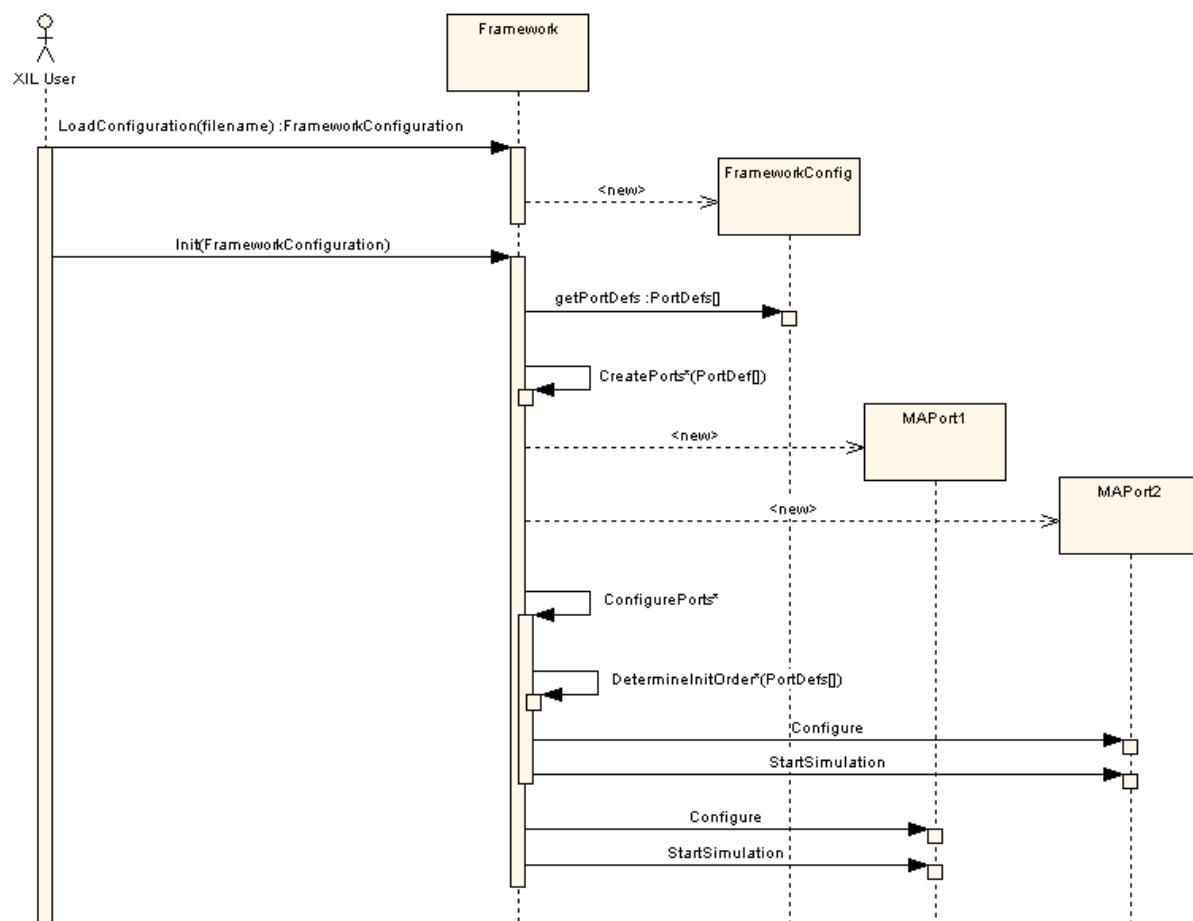


Figure 22 Framework configuration process example (* for descriptive purpose only, not part of the interface)

The process of port setup to be implemented by the framework's `Init` method is more complex and requires detailed explanation ([Figure 22](#) shows an example with two `MAPort` instances). Initially all port instances defined in the configuration by means of `PortDefinition` entries are created. Thereafter every port instance is configured and the requested target state is established.

Port configuration is realized by loading the port configuration file via the port's `LoadConfiguration` method and passing the resulting `PortConfig` object to the port's `Configure` method (see [Figure 23](#)). This makes a port connecting to the underlying tool or hardware and applying the vendor and tool specific settings from its configuration file. The configuration file is retrieved from the `PortConfigurationFile` element of the corresponding `PortDefinition`.

Having a port successfully configured the framework tries to switch the port to the target state defined by the `TargetState` element of its `PortDefinition`. Target state as well as port methods being called to establish that state are port type specific. For instance, an `ECUMPort` will be in state `eMEASUREMENT_STOPPED` after successful configuration. But if the requested target state of this port is `eMEASUREMENT_RUNNING`, the framework will invoke `StartMeasurement`.

Some of the methods that need to be invoked to switch port states may require parameters. The values of those parameters are obtained from the corresponding `PortDefinition` (see chapter [4.1.5](#)).

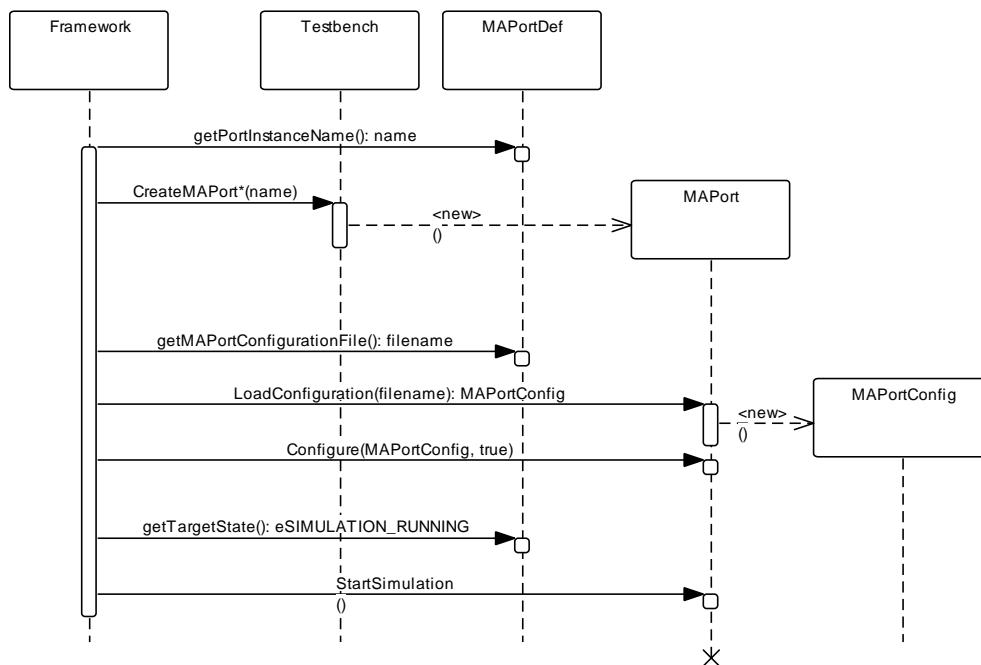


Figure 23 Example for port creation and setup process by the framework (* for descriptive purpose only, not part of the interface)

The order of port instance creation is not standarized and hence left open to the Framework implementation. But the order of port configuration and target state establishing is client controlled and can be specified by means of the `InitOrder` attribute in the framework configuration (see chapter [Framework Configuration File](#)). The `Init` method implementation

must honor the `InitOrder` values and apply the resulting start order. This is also illustrated by [Figure 22](#).

If the `Init` method encounters a problem during port configuring it stops processing of further ports and returns with an error. In that case it is up to the client to take care of remaining port configuration tasks (see chapter [Customize Automatic port management of the framework](#)) or apply any other error handling strategy.

After initialization the framework's `Ports` property returns a list containing the port instances created by the framework. The property `Configuration` points to the `FrameworkConfig` object passed to `Init`.

Note: At the beginning of the initialization process existing port instances are disposed. That's why it is recommended to call `Shutdown` on a framework that has already been initialized before initializing a new configuration. Otherwise a clean shutdown of existing port instances cannot be ensured.

4.1.3 SHUTTING DOWN THE FRAMEWORK

Shutdown means stopping, disconnecting and disposing all existing port instances. Therefore the `Framework` interface provides the `Shutdown` method. `Shutdown` invokes `Disconnect` and then `Dispose` on a port before the port is removed from the framework's list of managed ports. [Figure 24](#) illustrates the shutdown process using the example of two `MAPort` instances.

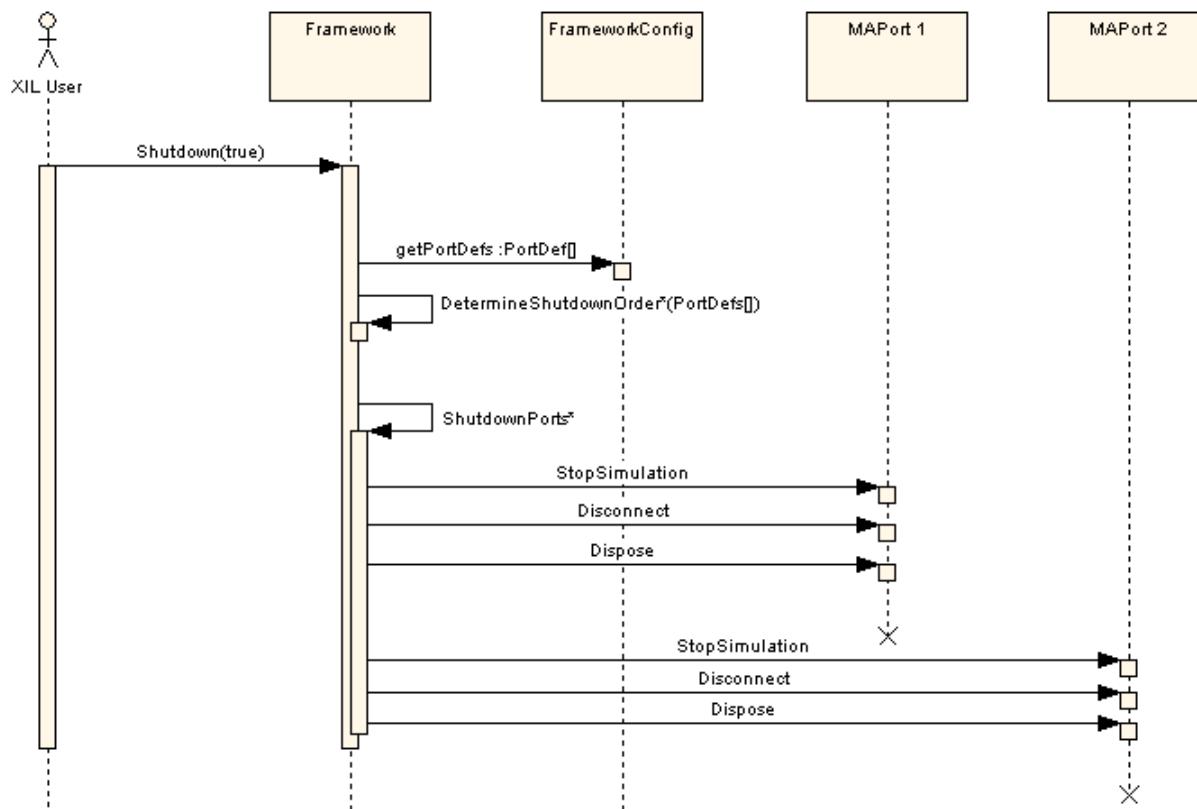


Figure 24 Framework shutdown process example (* for descriptive purpose only, not part of the interface)

Analogous to the initialization order there is a shutdown order that is client controlled and can be specified by means of the `ShutdownOrder` attribute in the framework configuration (see chapter [Framework Configuration File](#)). The `Shutdown` method must honor the `ShutdownOrder` values and apply the resulting shutdown order.

The exception handling of `Shutdown` (failure to disconnecting or disposing a port) depends on its parameter `force`. If set to `FALSE` `Shutdown` stops processing further ports and returns with an error. In that case it is up to the client to take care of remaining shutdown tasks (see chapter [Customize Automatic port management of the framework](#)) or apply any other error handling strategy. If parameter `force` is set to `TRUE` `Shutdown` ignores the error and skips on to the next port.

After successful shutdown the framework's `Ports` property returns an empty list and the properties `Configuration` and `Mapping` return `Null` values. Note that all references to testbench ports are invalidated by calling `Shutdown`.

Calls to `Shutdown` without prior initialization are ignored and do not yield an error.

4.1.4 CUSTOMIZE AUTOMATIC PORT MANAGEMENT OF THE FRAMEWORK

The `Init` and `Shutdown` method of the `Framework` interface provide automatic creation and configuration of testbench ports as well as automatic shutdown. Hence the client is freed from these tasks and can focus on execution of test cases. However there are use cases that require customization of the initialization and shutdown process.

Testbench setups containing a `MAPort` and an `ECUCPort` are prominent examples that regularly require an advanced startup procedure because the `ECUCPort` needs the clamps properly activated via the `MAPort` before it can switch to state `eONLINE`. That means one cannot set the `TargetState` of the `ECUCPort` to `eONLINE` in the framework configuration but to `eOFFLINE`. So after the framework's `Init` method has returned it is up to the client to activate the clamps via the corresponding framework variable (or directly on the `MAPort`) and finally call `StartOnlineCalibration` on the `ECUCPort` to finish the startup procedure. [Figure 25](#) shows this approach.

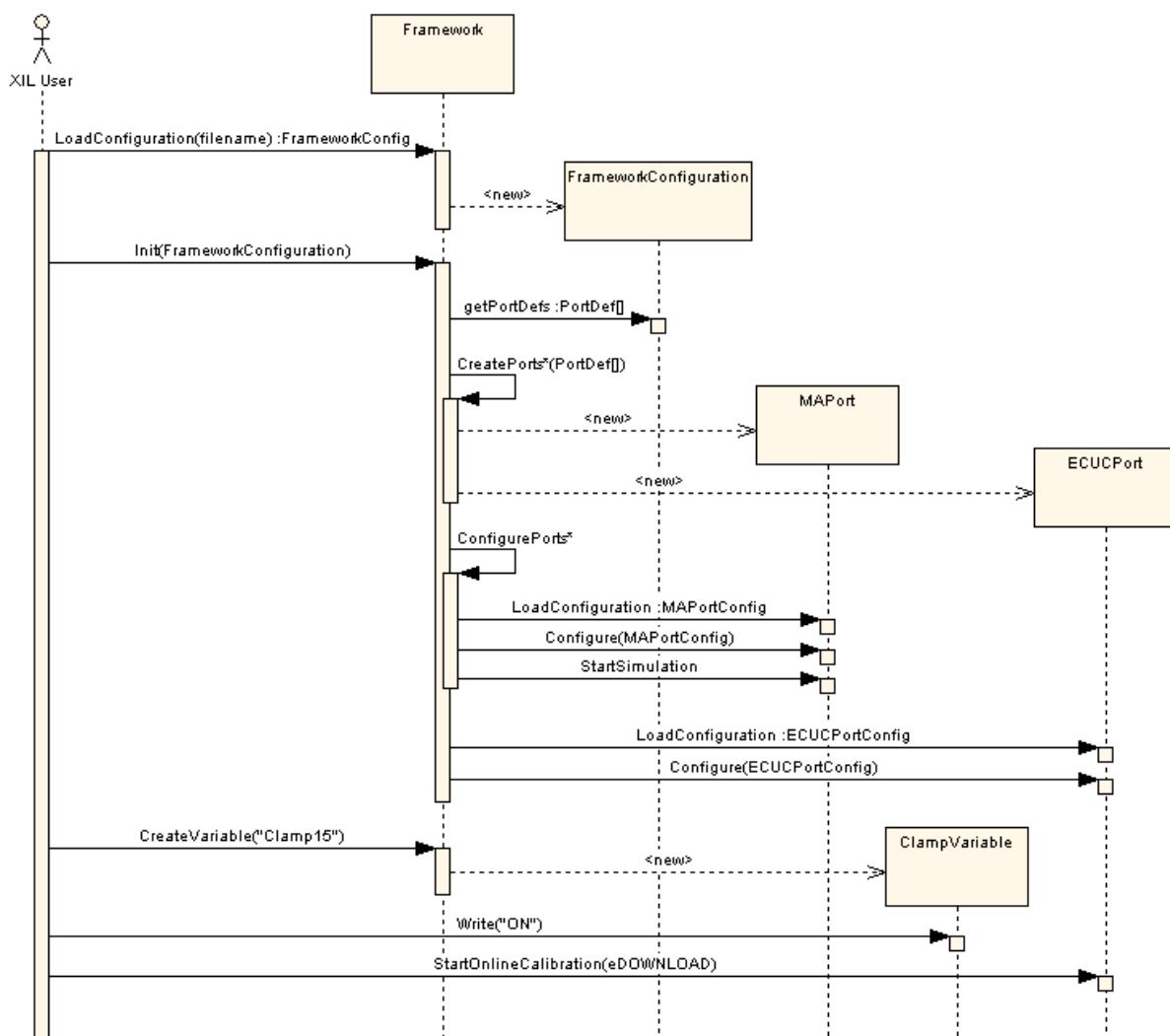


Figure 25 Example for customized Framework configuration process (* for descriptive purpose only, not part of the interface)

Another use case for client controlled port management is the implementation of a specific error handling strategy that is applied if automatic port setup or shut down fails.

The Framework interface provides several properties that facilitate client controlled port management. The testbench ports created and used by the framework can be retrieved via the Ports property or by the method GetPort. The currently active framework configuration (i.e. configuration passed to the most recent call to `Init`) can be accessed via the Configuration property. And finally there is also an access path to the mapping information via the Mapping property. Note, the last one is only a shortcut for `Configuration.Mapping`, since the mapping information is part of the framework configuration.

4.1.5 OBJECT MODEL OF THE FRAMEWORK CONFIGURATION

The `FrameworkConfig` instance created by the framework's `LoadConfiguration` method resembles structure and information of the framework configuration file (see chapter [Framework Configuration File](#)). [Figure 26](#) shows the class diagram.

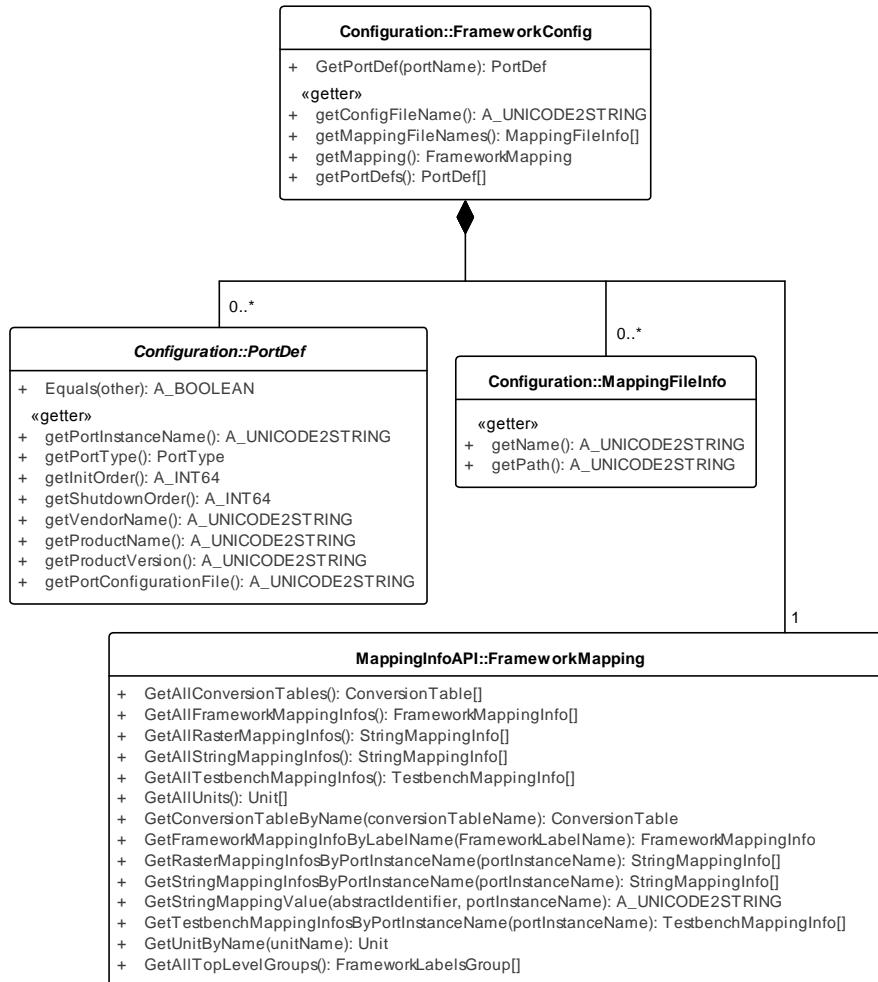


Figure 26: Framework configuration object model

All information required by the framework to create and configure a specific port instance is stored in a **PortDef** object. **PortDef** objects can be retrieved all at once using the **Ports** property of **FrameworkConfig** or individually by passing a port name to the **GetPortDef** method. Note, **PortDef** objects can be of different types, depending on the type of port they refer to (see [Figure 27](#)). The type information can be retrieved via the **PortDef**'s **PortType** property.

Every type of **PortDef** has a specific set of properties for the retrieval of parameter values that are required to drive the port to a certain target state. (The requested target state can be obtained from the **TargetState** property.) However, depending on **TargetState** the values of those parameter properties can be available or not. So, reading the value of a parameter that is not needed by the port to reach the targeted state results in an **eFW_UNAVAILABLE_PROPERTY** error. For example, accessing **ErrorConfigurationFile** on an **EESPortDef** when **TargetState** is **eDISCONNECTED** yields that error.

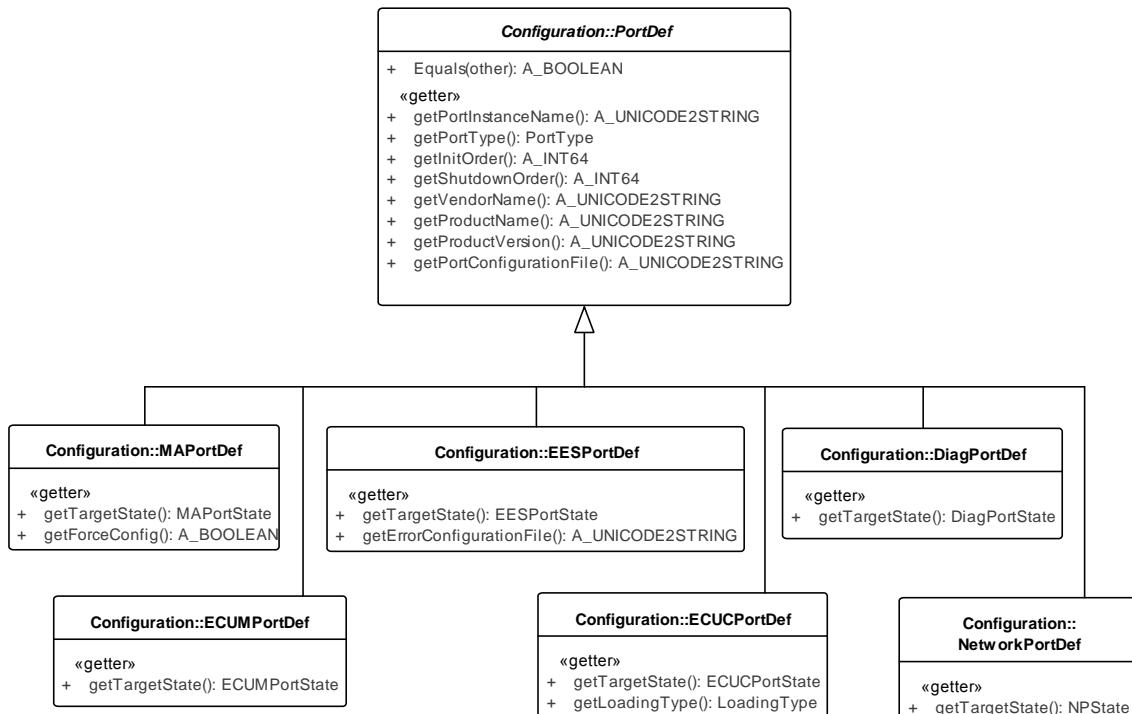


Figure 27: Port definition interfaces of the framework configuration object model

Besides port configuration related information the `FrameworkConfig` interface also provides access to mapping information. Use the `Mapping` property to obtain the appropriate interface. For a detailed description of the Mapping Interface refer to chapter [Mapping](#). The list of mapping files the mapping information has been loaded from can be retrieved from the `MappingFileNames` property.

Please note that the introduced interfaces provide no means for any manipulation or writing of configuration files.

4.2 MAPPING

4.2.1 OVERVIEW

The purpose of the mapping is to allow the decoupling of the framework side from the testbench side. This makes it possible to write test cases that are independent of the test tool employed as well as the underlying testbench.

In order to achieve this goal the main task of the mapping is to associate abstract identifiers on the framework side (framework labels) with concrete identifiers on the testbench side (testbench labels). For the conversion between two identifiers' values different physical units and conversion tables can be defined. The conversion is then handled automatically by the framework. Additionally the names of rasters as well as filenames can be kept abstract when using the mapping.

Thus the mapping has mainly three purposes:

- **Identifier mapping:** Maps a framework label to a testbench label. Optionally various conversion parameters can be used to customize this mapping. This mapping is resolved automatically by the framework.
- **String mapping:** Maps framework strings to testbench strings. For example this can be used to map abstract filenames on the framework side to concrete filenames on the testbench side. This mapping is resolved automatically by the framework whenever the test case calls a method with a string parameter.
- **Raster mapping:** Maps abstract raster names on the framework side to concrete raster names on the testbench side. This mapping is resolved automatically by the framework.

Each of these mapping types will be explained in more detail in the following chapters.

4.2.2 THE MAPPING XML SCHEMA

The mapping consists of one or more XML files that are referenced by the main framework configuration file. The structure of the XML files is defined by the mapping XSD. The following illustration shows the top levels of the XSD's hierarchy.

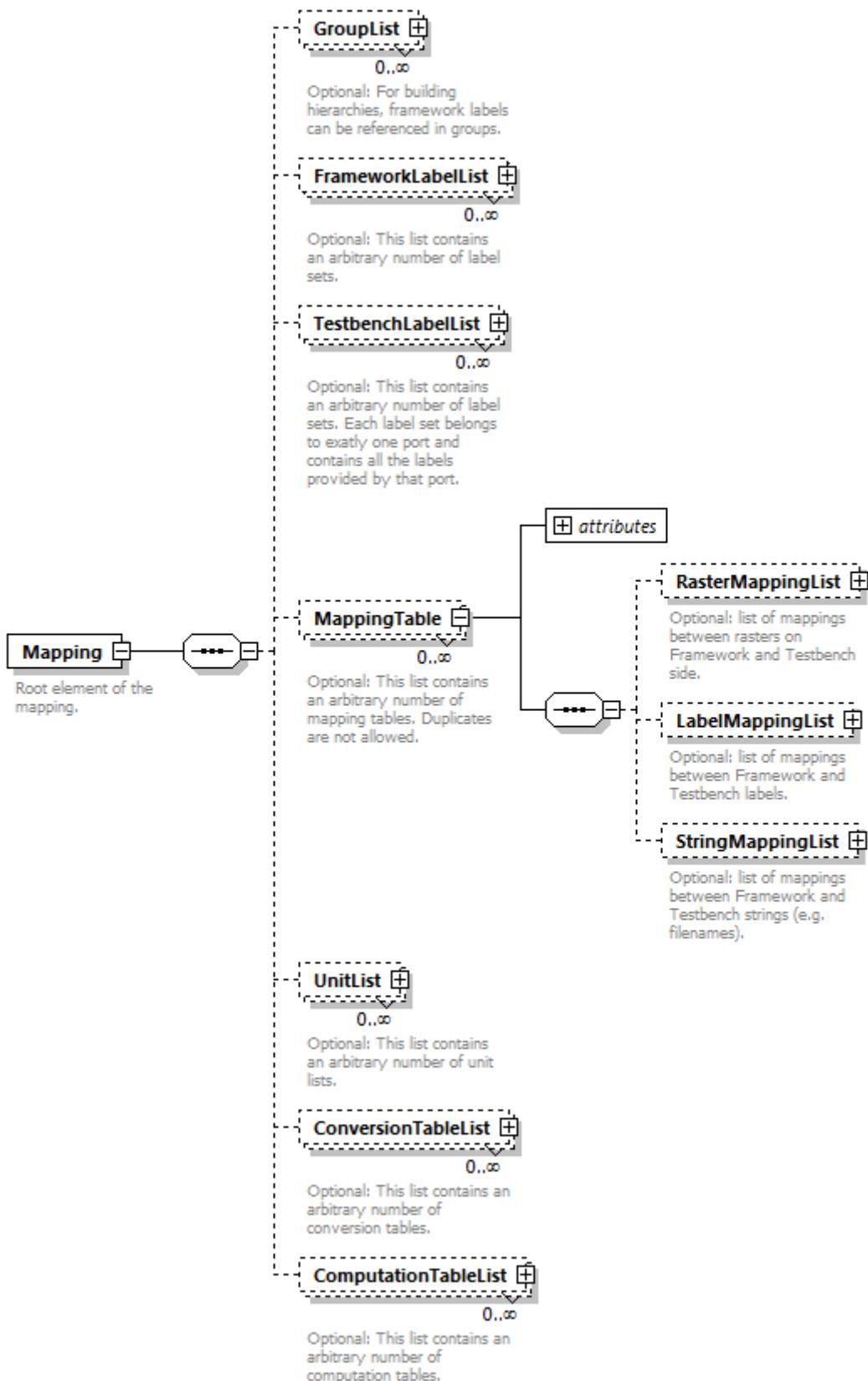


Figure 28: Top level entities of the mapping XSD

In the XSD the following top level entities are defined:

- **GroupList:** This list contains a hierarchical structure of group definitions. It allows grouping of framework labels.
- **FrameworkLabelList:** This list contains abstract identifiers that will be used on the framework side. Thus these identifiers are used in the test cases.
- **TestbenchLabelList:** This list holds the concrete identifiers that are used on the testbench side.
- **MappingTable:** The mapping table establishes the relationship between framework and testbench identifiers. It also contains the mapping of raster names as well as the mapping of strings, e.g. filenames.
- **UnitList:** This list holds all the units used by framework and testbench identifiers.
- **ConversionTableList:** This list contains the conversion tables used by the mappings between framework and testbench identifiers.
- **ComputationTableList:** This list contains an arbitrary number of computation tables. Each of these tables describes how to perform a multiplication or division operation on physical dimensions.

Each of these entities will be explained in more detail in the following chapters.

Splitting the mapping into multiple XML files is highly recommended for reasons of clarity and comprehensibility. For example a mapping could be split into separate files for every port used, or different sets of units could be outsourced to separate files to make them easily exchangeable.

4.2.2.1 IDENTIFIER LISTS

The mapping contains two types of identifiers: framework labels and testbench labels. Each of the two types is defined within a corresponding label list, which is specified by the XML elements `TestbenchLabelList` and `FrameworkLabelList`, respectively.

A Framework label is defined by a unique id, a data type and an optional physical unit. The usage of the physical units depends on the type of the label: while simple values may only use the regular `UnitId`, curve and map types may also have units for their axes. `FrameworkMapping` class allows retrieving a collection of all available framework labels and their identifiers from within the test case.

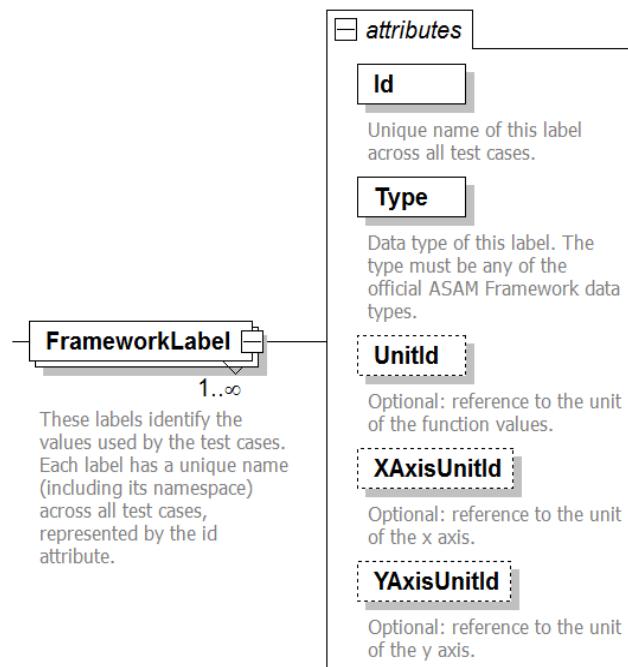


Figure 29: XSD structure of a framework label

A Testbench label is defined by a port specific id, a data type and an optional physical unit. That means that labels with the same identifier may exist on different ports. Additionally metadata (e.g. min and max values) can be provided. Analogous to the unit ids, metadata may be provided for each axis separately.

In contrast to the framework labels the data type is not simply specified by a single enumeration element. Instead one of the three elements “TestbenchSimpleType”, “TestbenchCurveType” and “TestbenchMapType” has to be chosen. This is due to the fact that depending on the selected element different attributes and sub elements can be specified. For example the element “TestbenchMapType” has an individual type attribute for the X-axis, the Y-axis and the function value axis. Additionally each of the three axes has its own set of metadata.

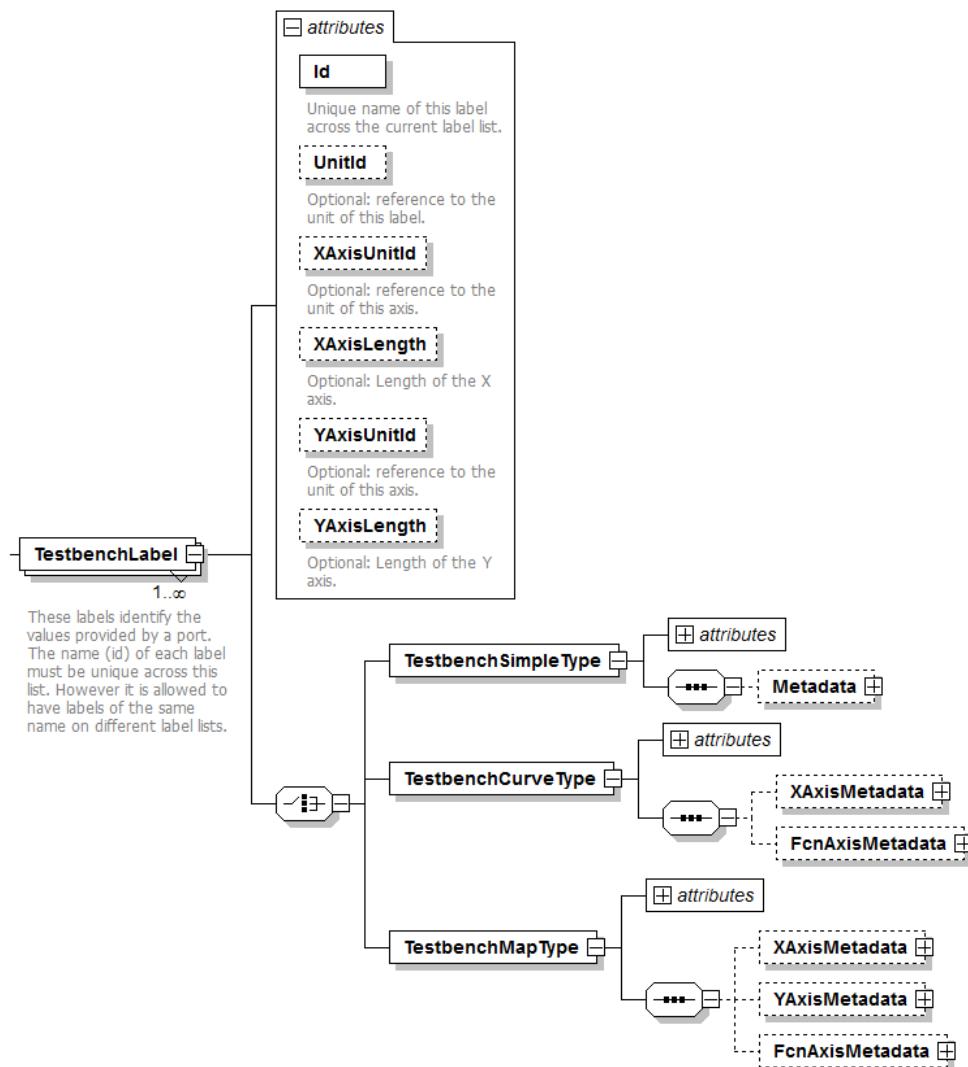


Figure 30: XSD structure of a testbench label

The purpose of the metadata is to represent constraints of the model. For example the min and max values may be used to prevent testbench variables from getting assigned invalid and possible harmful values. Please note that these metadata are defined on both the testbench and the framework side.

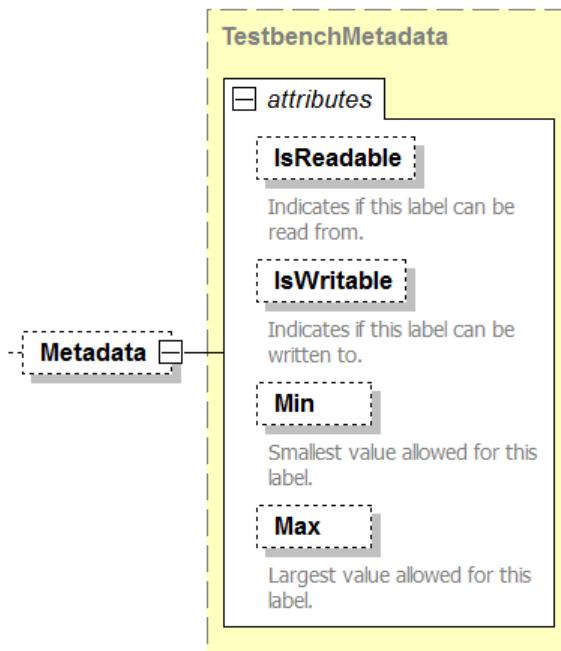


Figure 31: XSD structure of the testbench metadata

4.2.2.2 FRAMEWORK LABEL GROUP

Optional GroupList elements are available within the Mapping. This is a collection of Group elements.

Each Group element has a mandatory id. The id of a group does not need to be unique; see below, in the rules of merging paragraph, how possible conflicts are solved. Group elements can reference an arbitrary number of FrameworkLabelId elements. One FrameworkLabelId can be referenced by multiple Group elements. Each Group element can have a collection of subgroups.

When processing the Mapping, all the GroupList elements from all the mapping files are merged together into a single list of group definitions. The rules of merging are the following:

- Merging is applied recursively to all groups in the resulting list of group definitions. Because group ids are not unique, it is possible that multiple children groups of the same parent to have the same id; they may be originating from different group lists defined in different mapping files, or different group lists defined in the same file or even from the same group list. So, all the Group elements that have the same parent and the same id are merged into a single Group with the corresponding id that has all the framework label references and all the sub-groups from all the merging groups.
- If the same framework label is referenced multiple times in one group (that can be the result of merging), then only one reference is kept, and the duplicates are dropped.

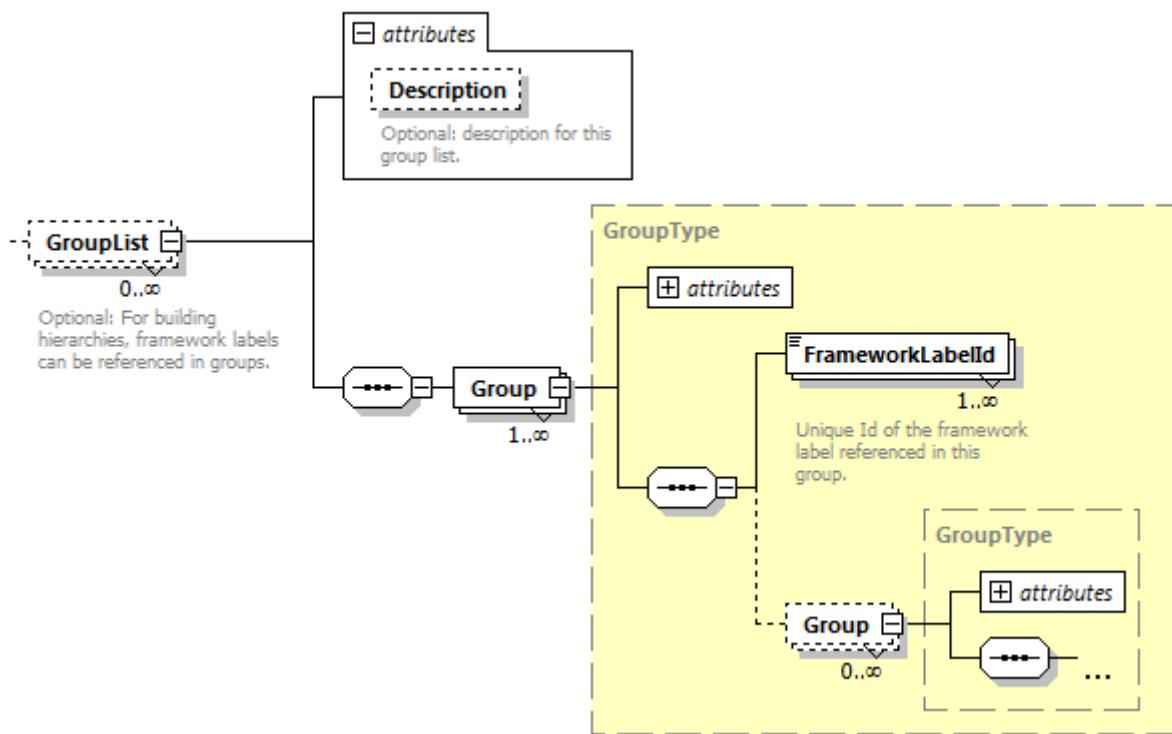


Figure 30: XSD structure of `GroupList`

4.2.2.3 IDENTIFIER MAPPING

The mapping between identifiers is defined by the label mapping list. In the simplest case a single mapping consists of the id of a framework label and a testbench label (alongside its port id). The framework will then handle the data exchange between the two labels during runtime. This includes automatic conversion between the two labels' units.

Additionally a conversion table can be referenced by the mapping (see chapter [Conversion Tables](#)).

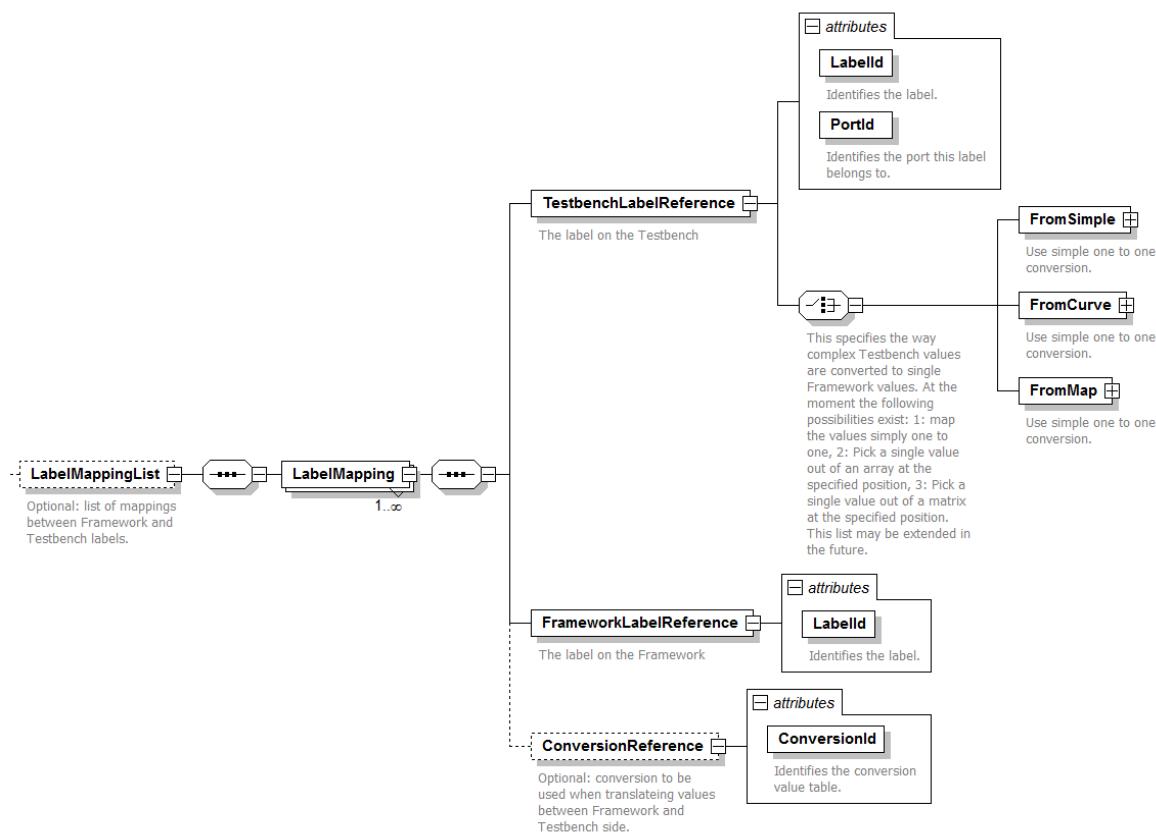


Figure 32: XSD structure of the label mapping

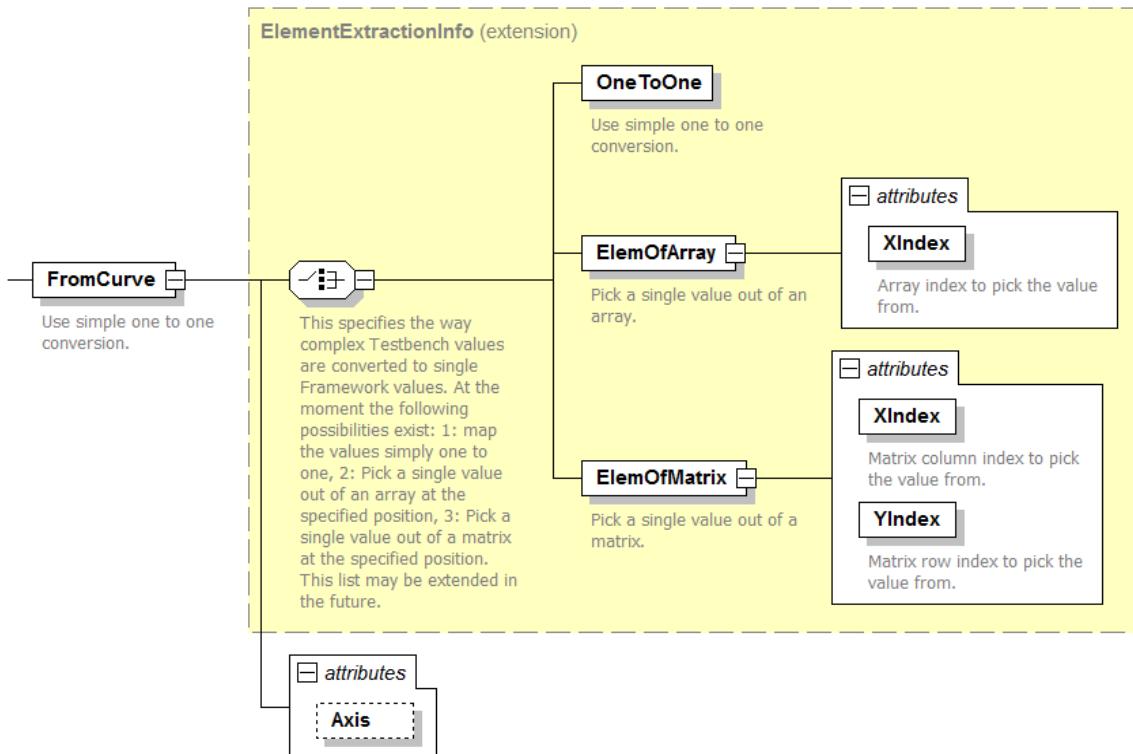


Figure 33: XSD structure of the FromCurve mapping selector

More complex cases, where one or both of the labels have a compound data type (a curve or a map) can also be handled. For that purpose three options are available:

- **FromSimple**: the testbench label has a simple data type
- **FromCurve**: the testbench label is a curve
- **FromMap**: the testbench label is a map

Each of these three elements offers three more options as sub elements:

- **OneToOne**: the testbench label's value is directly mapped to the framework label's value (e.g. simple to simple, curve to curve or map to map)
- **ElemOfArray**: one value out of an array (e.g. curve axis) from the testbench label is mapped to a framework label's simple value
- **ElemOfMatrix**: one value out of a matrix (e.g. map function values) from the testbench label is mapped to a framework label's simple value

If a value is to be taken from an axis, this axis can be specified via the "Axis" attribute.

4.2.2.4 STRING MAPPING

The string mapping allows establishing a connection between strings on the framework side and strings on the testbench side. For example quite often filenames (e.g. STI files) may differ between testbenches. In that case it is useful to reference those files via abstract filenames. These names can then be mapped to concrete testbench filenames using the mapping. For that purpose the mapping may contain multiple lists of general string to string mappings associated with a specific port. Each of these mappings consists of two strings (one for the framework side, one for the testbench side).

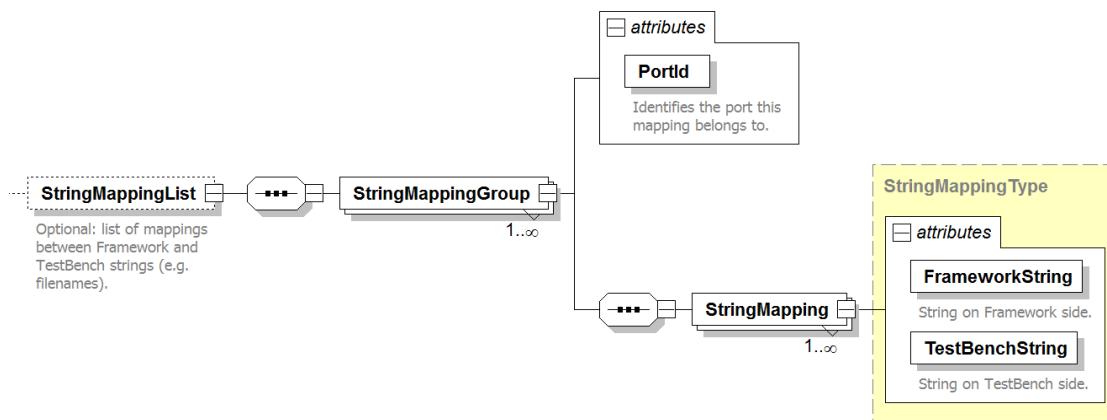


Figure 34: XSD structure of the string mapping

The string mapping is used whenever a method with a string parameter is called by the test case. In this situation the framework automatically searches the mapping for the past parameter's value. If the value is found as a `FrameworkString` it is automatically replaced by its mapped `TestBenchString`. If it is not found the parameter's value remains unchanged.

Please note that this automatic string mapping mechanism is only applied in one direction. Strings returned from method calls to the testbench side will never get mapped by the framework.

Please also note that for this mechanism to work it is important that every string mapping is unique. This means that each `FrameworkString` defined in the mapping file (independent of the `PortId` of the `StringMappingGroup`) must be unique.

4.2.2.5 RASTER MAPPING

The assignment of abstract raster names on the framework side to concrete raster names on the testbench side is also part of the mapping. To represent these relationships every mapping table contains a list of raster mappings. Each raster mapping consists of two raster names (one for the framework side, one for the testbench side) and a port identifier. This identifier specifies which port the raster is associated with.

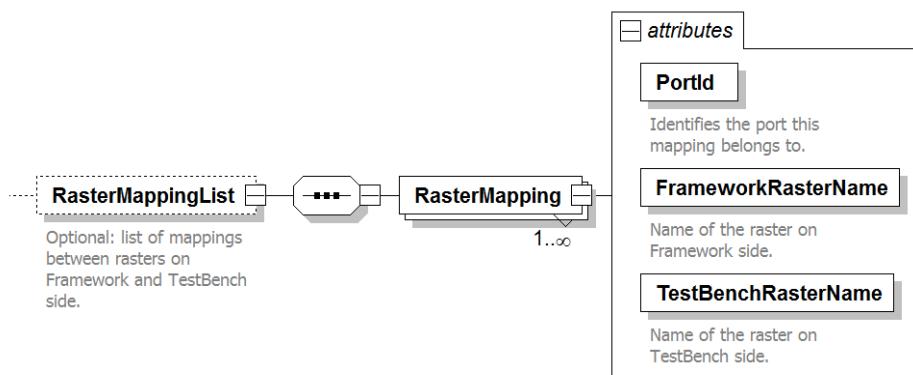


Figure 35: XSD structure of the raster mapping

These mappings are automatically used by the framework whenever a method with a raster name as a parameter is called by a test case.

Please note that raster mappings take priority over general string mappings. E.g. if a string is mapped both via a raster mapping and a general string mapping, the raster mapping is applied first.

4.2.2.6 CONVERSION TABLES

The idea behind the conversion tables is to allow the mapping of concrete string values of variables on the testbench side to more abstract string values on the framework side. Thus every mapping between a framework label and a testbench label may optionally reference a conversion table. During runtime every string value passed from the framework side to the testbench side and vice versa is translated using this conversion table.

The conversion tables themselves are basically a simple list of string pairs. When a value gets translated, it is looked up in the conversion table. If it is found, the corresponding string of the string pair is extracted. That string is then assigned to the target label.

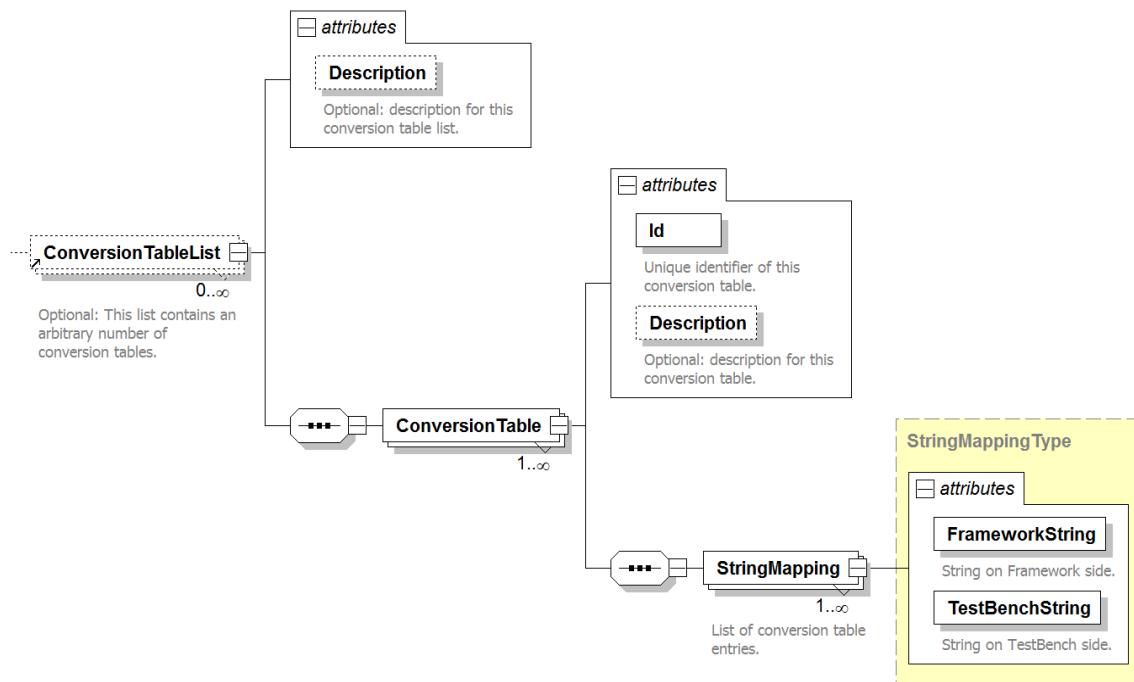


Figure 36: XSD structure of the conversion tables

Example:

The following conversion table is attached to a mapping:

```
<ConversionTable Id="ExampleTable">
    <StringMapping TestbenchString="1" FrameworkString="Gear 1"/>
</ConversionTable>
```

When the value “Gear 1” is assigned to the framework variable, the value will be translated automatically by the framework. Thus on the testbench side the value “1” will be used.

4.2.2.7 UNITS

The mapping allows defining units which serve two purposes: On the one hand they provide information about the values in the framework. On the other hand they are used for automatic conversion between framework and testbench values (e.g. miles per hour to kilometers per hour). For that reason each identifier on the framework and the testbench side can be associated with a unit.

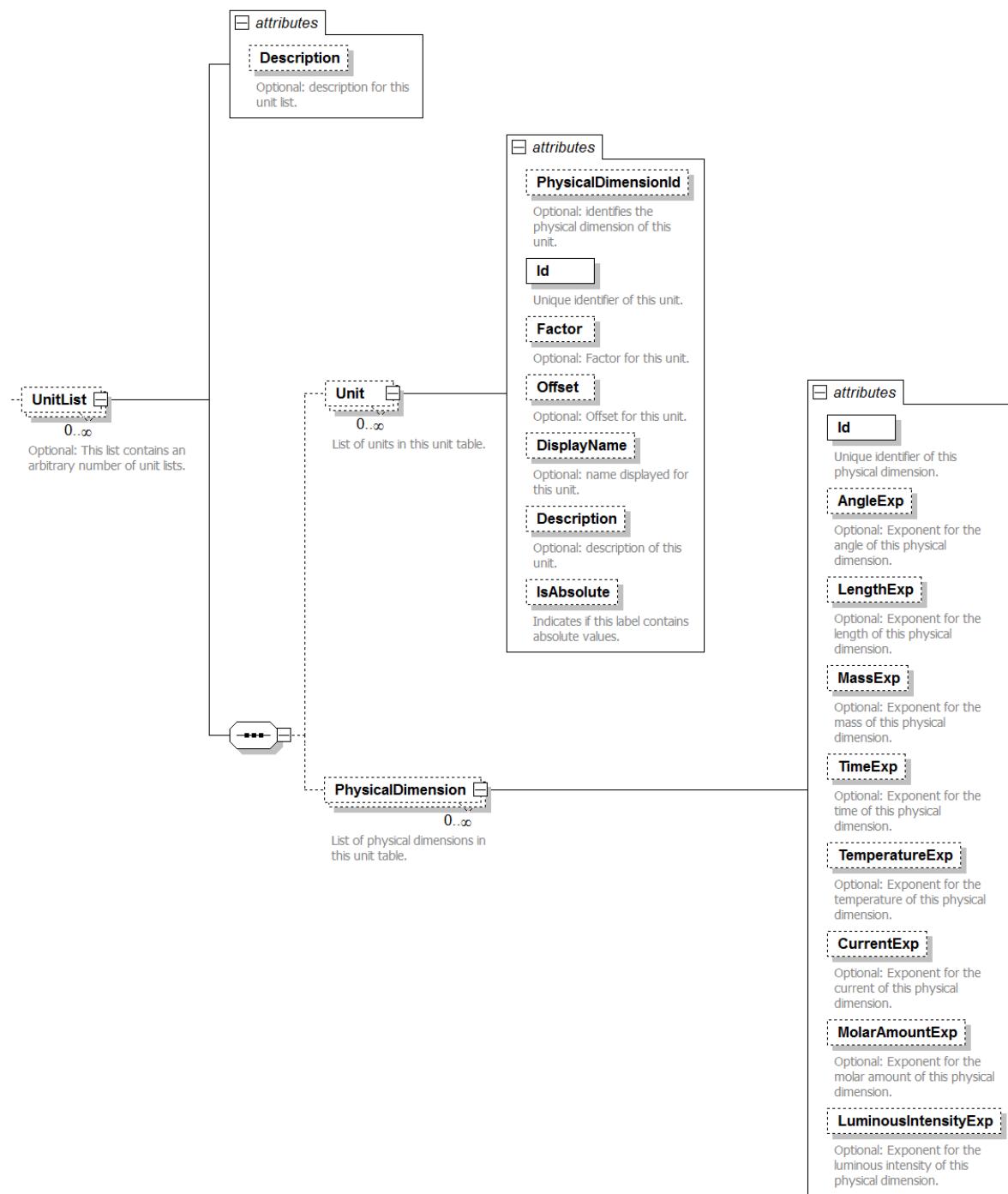


Figure 37: XSD structure of the units

The units themselves are specified by a factor, an offset and a physical dimension. The factor and the offset are simple scalar values with relation to the corresponding fundamental unit. For example the unit "miles per hour" has a factor of 2.369 and no offset with relation to the fundamental unit "meters per second".

The physical dimensions are defined by specifying exponents for the SI units. E.g. the physical dimension for "meters per second" would have an exponent of 1 for the dimension "length" and an exponent of -1 for the dimension "time". All other exponents would be set to 0.

One special case worth mentioning are physical dimensions including angles. Although degrees resp. radians are no SI unit they can be expressed via AngleExp. For example the physical dimension “degrees per second” can be defined by setting AngleExp to 1 and TimeExp to -1.

Unit information is available for both the framework and testbench sides. If no unit information is available on the framework side the information from the testbench side is used.

4.2.2.8 COMPUTATION TABLES

The purpose of the computation tables is to define how multiplication and division operations of framework variables are handled if physical dimensions are involved. For every allowed operation (meaning a multiplication of two physical dimensions and the corresponding division) one computation table has to be defined. If an operation is to be performed during runtime and no matching computation table is found an exception will be thrown.

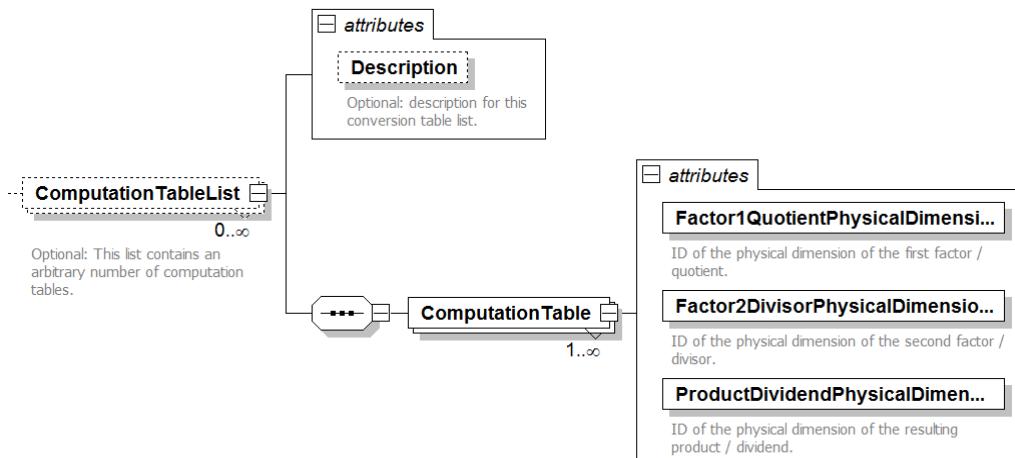


Figure 38: XSD structure of the units

As shown in the illustration each computation table consists of three physical dimension IDs. These identify the first factor resp. quotient, the second factor resp. divisor and the resulting product resp. dividend. Of course for each of these IDs a corresponding physical dimension has to be defined in the mapping files.

Example:

For reasons of clarification consider the following example:
Two physical dimensions are defined in the mapping files:

Table 4 Physical Dimension Example

ID	Exponents	Description
Length	LengthExp = 1	Length in m
Area	LengthExp = 2	Area in m ²

To allow performing multiplication and division operations with these physical dimensions the following computation table would have to be added to the mapping:

```
<ComputationTable
  Factor1QuotientPhysicalDimensionId="Length"
  Factor2DivisorPhysicalDimensionId="Length"
  ProductDividendPhysicalDimensionId="Area"/>
```

From this information the framework can then deduce, that Length * Length equals Area, and that Area / Length equals Length.

4.2.2.9 VERSIONING

The versioning of the mapping files and the XSD schema is done via the `xmIn's namespace attribute`. Inside a XML mapping instance the namespace "`http://www.asam.net/XILAPI/Mapping/1.0`" has to be used. The version number at the end has to match the version of the XSD file the XML was built against.

4.2.3 HOW A MAPPING IS USED BY TEST CASES

This chapter describes how the different mapping types are used by test cases.

4.2.3.1 THE MAPPING INFO API

The mapping info API allows the test case to retrieve information about the loaded mapping during runtime. This information includes the mapped identifiers, available units, physical dimensions, string mappings and conversion tables.

To obtain this information from within a test case, use the class `FrameworkMapping`. An instance of that class may be obtained by calling `GetConfiguration` on the main `Framework` object. The resulting `FrameworkConfig` object then offers a method `GetMapping`.

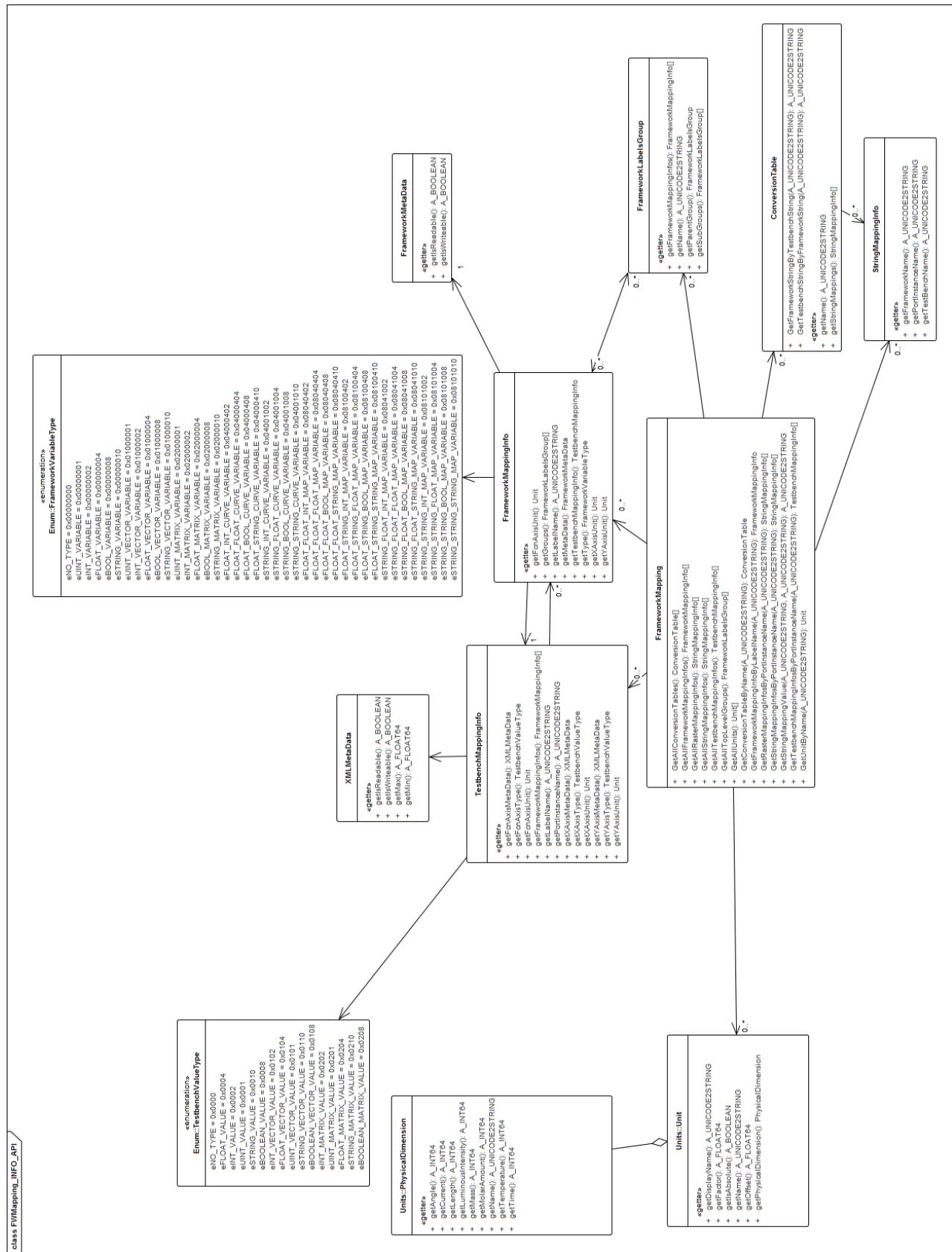


Figure 39: Framework Mapping Info API

4.2.3.2 IDENTIFIER MAPPING

The framework provides access to all framework labels via the method `CreateVariable` of the class `Framework`. This method takes an abstract identifier as its only parameter. Inside the framework this identifier is then resolved via the mapping. That way a connection between the framework label and the testbench label mapped to it is established.

The method returns a `FrameworkVariable` that has been set up with all the information specified in the mapping (e.g. the data type). Subsequent read and write requests on this variable will be handled by the framework using the configured conversion rules.

4.2.3.3 STRING MAPPING

Several methods of the framework or the testbench ports expect filenames as parameters. When calling such a method the framework will automatically try to map the given string parameter. If a proper mapping is found, the string gets replaced with its mapped value. If no mapping is found the string remains unchanged.

4.2.3.4 RASTER MAPPING

Like general string mappings raster mappings are not explicitly accessed by the test case. These are implicitly used by the framework capturing components.

4.2.4 TASKS OF THE FRAMEWORK

This chapter describes several framework tasks regarding the mapping. Some of the tasks (e.g. the creation of variables and the handling of rasters) are not mentioned here since they are explained in great detail in the corresponding chapters of this specification.

4.2.4.1 READING MAPPING XML FILES

As explained in chapter [Overview](#) a mapping consists of one or more XML files. One task of the framework is to read in these XML files. To help with this task the ASAM provides an official mapping reader. This reader parses a given XML file and generates the corresponding object model. If multiple mapping files were referenced by the framework configuration this leads to multiple object models. These have to be consolidated by the framework as explained below.

4.2.4.2 CONSOLIDATING MAPPING OBJECT TREES

After each required XML has been read in it is demanded of the framework to consolidate all object models into a single one. This can easily be done by concatenating all of the lists (labels, units etc.) in each model. After all object models have been merged the resulting model has to be validated. For that purpose a list of consistency rules has been specified. See chapter [Consistency Rules](#) for the complete list. The model has to be checked against each of these rules by the framework.

4.2.4.3 IMPLEMENTING THE MAPPING INFO API

The Mapping Info API allows the test cases to access mapping related information during runtime. The framework has to provide the required implementation for the corresponding interfaces. Please see [Figure 39](#) for details on which classes and methods have to be provided.

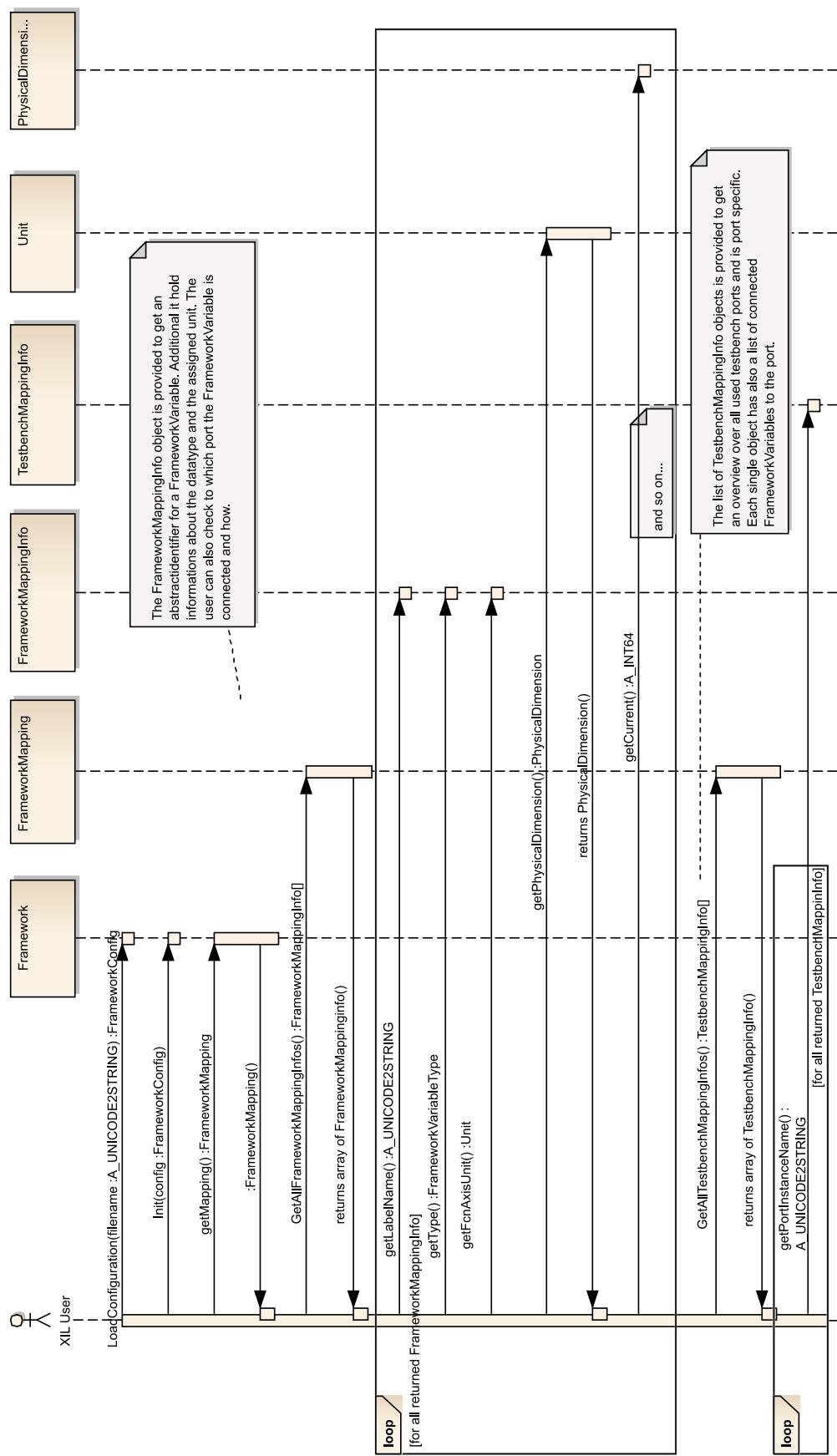


Figure 40: Access via MappingInfo API

4.2.4.4 RESOLVE THE MAPPING DURING VARIABLE CREATION

Framework variables may be created by the test case via the method `CreateVariable` of the class `Framework`. In the implementation of this method the mapping has to be used to find the testbench variable corresponding to the given framework variable.

4.2.4.5 PERFORM AUTOMATIC STRING MAPPINGS

In every method offered by the framework that accepts string parameters a mapping has to be performed. The string value passed in by the test case has to be checked against all string mappings defined in the mapping file. If a corresponding mapping entry is found the framework string has to be replaced by the proper testbench string. If no mapping is found the string is not to be changed.

4.2.4.6 ERROR HANDLING

Whenever a mapping related error occurs (e.g. one of the consistency rules is not satisfied) a proper exception has to be thrown by the framework. This exception has to be derived from `XILAPIException` and must contain a meaningful error message.

4.2.5 CONSISTENCY RULES

Every mapping XML file is checked for conformity to the XSD schema when it is loaded by the mapping reader. But even if the XML file matches the XSD schema that does not mean that the mapping is valid. For example a framework label id may have been used twice, which would lead to an exception during runtime.

Therefore the framework has to ensure that a set of consistency rules is adhered to. This check has to be performed after all (one or more) mapping files have been read by the mapping reader and their object models have been merged by the framework. The consistency rules that are to be checked are explained below.

Table 5 Consistency rules

#	Requirement	Error criterion derived from requirement
Duplicates		
D1	All framework labels' identifiers must be unique.	The framework label lists contain two FrameworkLabels with the same id.
D2	For each port all testbench labels' identifiers must be unique.	There is a testbench label list which contains two testbench labels with the same id. There are two testbench label lists with the same PortId that each contains a testbench label with the same id.
D3	Every unit's identifier must be unique.	The unit lists contain two units with the same id.
D4	Every physical dimension's identifier must be unique.	The unit lists contain two physical dimensions with the same id.
D5	Every conversion table's identifier must be unique.	The conversion table lists contain two tables with the same id.
D6	The content of a conversion table must be unambiguous in both directions.	A conversion table contains one string multiple times on the left hand side. A conversion table contains one string multiple times on the right hand side.

#	Requirement	Error criterion derived from requirement
D7	For each port the raster mappings have to be unambiguous from the test case's point of view.	There are two raster mappings with the same PortId and the same FrameworkRasterName.
D8	All string mappings have to be unambiguous from the test case's point of view.	There is a string mapping where one string occurs multiple times on the left hand side. There are two string mappings in which the same string occurs on the left hand side.
D9	No IDs reserved by the framework may be used in the mapping.	A unit with the ID "NO_UNIT" exists in the mapping. A physical dimension with the ID "NO_DIM" exists in the mapping.
D10	Label mappings have to be unambiguous from the test case's point of view.	There is a label mapping list where one framework label is referenced multiple times. There are two different label mapping lists that reference the same framework label.
D11	Identifiers of all sibling framework label groups must be unique.	A framework label group contains two children groups with the same id.
D12	All framework label references in a framework label group must be unique.	A framework label group references the same framework label multiple times.

Unresolved references

R1	A referenced physical dimension has to be defined.	A unit or computation table references a PhysicalDimensionId that is not defined.
R2	A referenced unit has to be defined.	A framework label references a UnitId that is not defined. A testbench label references a UnitId that is not defined.
R3	A referenced conversion table has to be defined.	The label mapping lists contain a ConversionId that is not defined.
R4	A referenced label has to be defined.	The label mapping lists contain a TestbenchLabelId that is not defined. The label mapping lists contain a FrameworkLabelId that is not defined.
R5	A label may only have UnitIds defined for axes if it is of the type curve or map.	A label has an X-axis UnitId although its type is neither curve nor map. A label has a Y-axis UnitId although its type is not map.

Conversion

K1	If two labels are mapped, the types of the labels and the mapping have to match.	There is a label mapping in which the type of the two referenced labels and of the mapping does not match. E.g. "IntVariable" is mapped to "BooleanMatrixValue" via "OneToOneMapping" and „FromSimple“.
K2	If a mapping uses values from an axis that axis has to be specified	There is a mapping using "FromCurve" or "FromMap" and that is not of the type

#	Requirement	Error criterion derived from requirement
	via the corresponding attribute.	"OneToOne". For that mapping the attribute "Axis" is not set.
K3	If a mapping exists between a framework and a testbench label both labels' units must be either absolute or relative.	There are no two labels A and B connected via a mapping where label A's unit's attribute "IsAbsolute" is set to a value different from label B's unit's attribute "IsAbsolute".

4.3 FRAMEWORK VARIABLES

4.3.1 WHAT IS A FRAMEWORK VARIABLE

Framework variables play a central role in the XIL-API standard. They allow an abstract vendor independent mechanism for data access by providing a connection between test case values and "physical" testbench port values. Framework variables are the primary data access mechanism in the XIL API Framework.

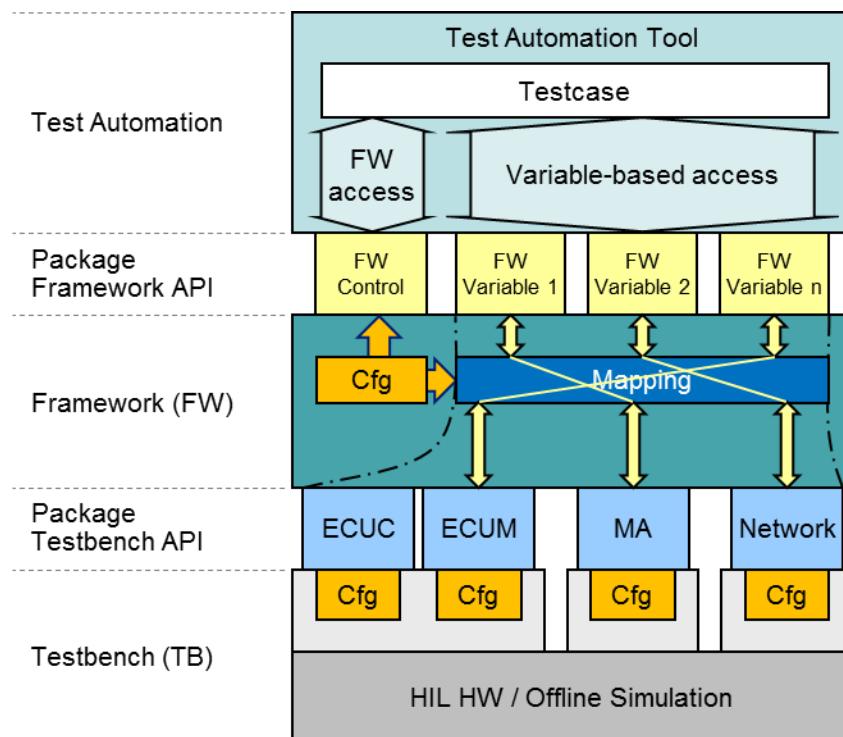


Figure 41 Data access using Framework variables

As it is shown in [Figure 41](#) test cases access the values/variables of a testbench port through framework variable objects that are connected to the corresponding testbench port. The connection between a framework variable and a variable of a testbench port is a multiple-to-one connection (multiple framework variables can be connected to the same testbench port variable) which is defined by the *Mapping* of the Framework. For the unambiguous identification of this connection the abstract identifier of a Framework variable is used. This unique abstract identifier is connected to a "concrete" identifier on the corresponding testbench port. For more information about Mapping, see chapter [Mapping](#).

Note: The current version of the XIL standard supports Framework variables for the following Testbench ports:

- Model Access (MA)
- ECU(MC)
- Network

The following Testbench ports are currently not supported:

- EES
- Diagnostic

Note: Framework variables always use the physical value representation mode for the quantities that they refer to. This applies to all operations where Framework variables are used: read / write, measuring and stimulation.

Note: If the user wants to read variables from the ECUMPort it is necessary that these variables are announced to the port as precondition. The announcement is realized with the method `ECUMPort.setMeasuringVariables`.

Since framework variables are accessed through their abstract identifiers it is guaranteed that test cases are independent of the underlying test system. By modifying the mapping of the Framework (and so the information about which abstract identifier is connected to which concrete identifier of a testbench port) the system can be easily adapted to new test system configurations without the need of changing the test cases.

For Example a test case can use the Framework variable with the abstract identifier "engine_speed" which is mapped in one case to the concrete identifier "myModel/myengine1/speed" of the MA Port and in another case with different mapping to the concrete identifier "myECU/engineSpeed" of the ECU(MC) Port. Without any modifications the test case can access the MA Port in the first case and the ECU(MC) port in the second case.

There are two ways to create an instance of a framework variable object:

1. Using the `CreateVariable()`method of the Framework

In this case test cases provide the abstract identifier of the framework variable and use the `CreateVariable()` method of the Framework.

2. Loading signals from file

Measuring uses framework variables in order to define the signals, which are to be recorded. The access of a variable's signal values is achieved by the `ScalarVariable.Signal` property. These signals can also be written to file. If a previously recorded file is loaded, the access to signal values is defined in a similar way. The reader returns an object, the `RecorderResult`. Calling `RecorderResult.getVariables()` returns a list of `ScalarVariable` objects. These variables can be accessed using the `ScalarVariable.Signal` property. However, these variables do not have any port mapping. Due to this fact, any `Read()`, `Write()`, or `SetVariables()` with one of these variables in the list will cause an error. For details, see chapter Measuring.

4.3.2 FRAMEWORK VARIABLE CLASSES

The XIL API standard defines the following five main types of the framework variable:

- Scalar variables

- Vector variables
- Matrix variables
- Curve variables
- Map variables

[Figure 42](#) shows an overview of the five main variable types of the framework variable.

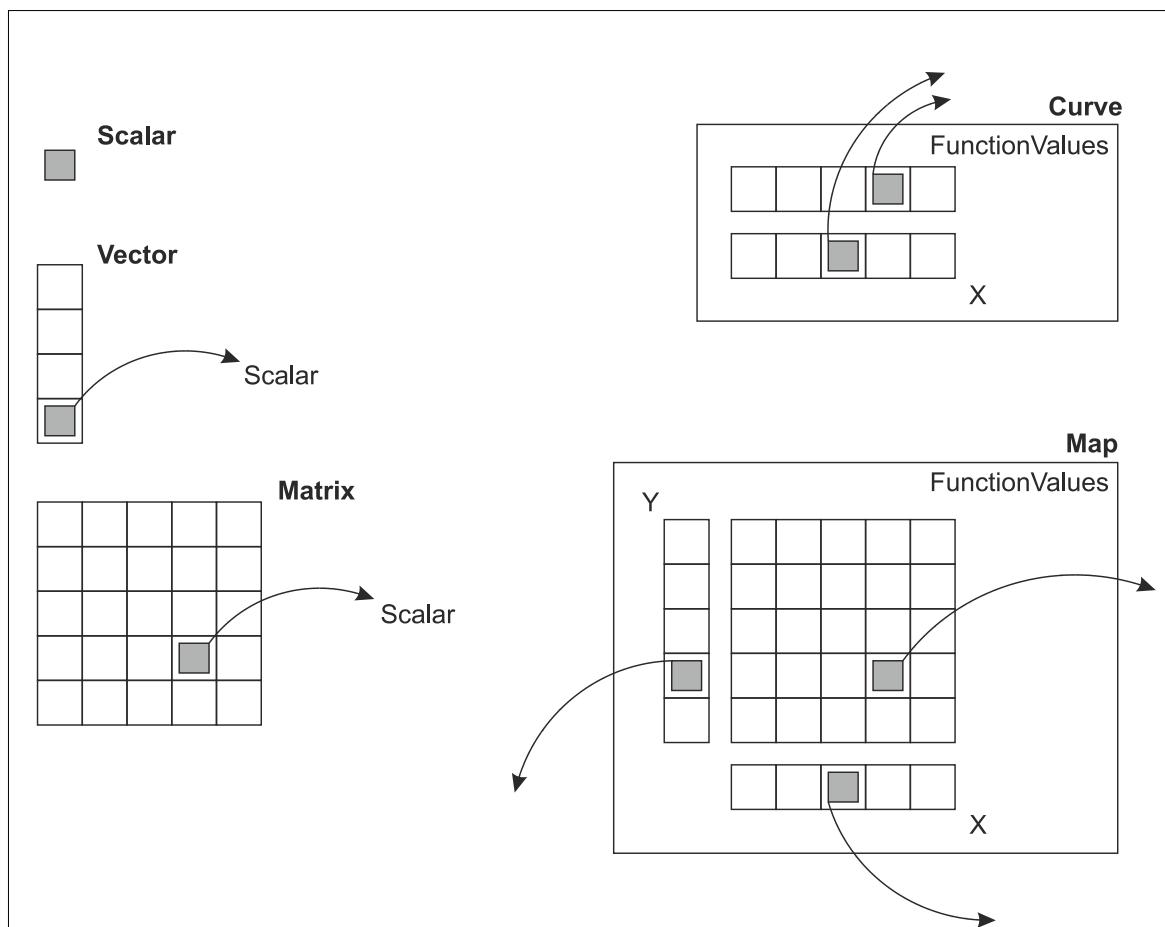


Figure 42: Main framework variable types

Each of the five main framework variable types is represented by its own framework variable class as it is shown in [Figure 43](#) below.

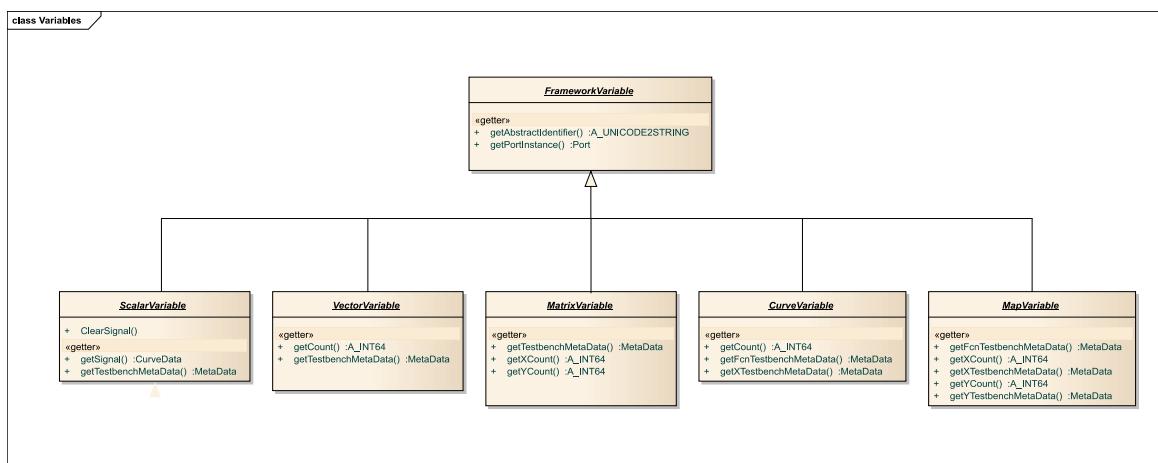


Figure 43: Main framework variable classes

The properties `AbstractIdentifier` and `PortInstance` are defined for all framework variables. The property `AbstractIdentifier` can be used to retrieve the abstract identifier of the framework variable. The property `PortInstance` returns the Testbench port that is connected to the framework variable. The framework variable has no port mapping, if the framework variable was created by loading a file (see chapter [Mapping](#)). In this case the `PortInstance` property returns an empty object.

There are concrete child classes (derived from the five main variable classes) for each supported data type. The Read and Write access methods are defined only for these child classes in order to be able to provide an inherent type safety for framework variable objects.

Each framework variable has meta-data information (e.g. default unit), which is stored in its `MetaData` object and defined in the mapping (see chapter [Mapping](#)). For example the default unit (if available) of such a `MetaData` object is used for the quantity objects that are returned by the `Read()` Method of a framework variable. The `Write()` method of a framework variable can be used with quantities that have different units as the default unit of the framework variable. In such cases an automatic conversion of the quantity into the default unit of the framework variable should be executed.

Test cases use quantities to work with the values of a framework variable object. These quantity objects are strongly typed and encapsulate a numeric value and a corresponding physical unit. This approach allows the implementation of intelligent framework variables that test cases can utilize for retrieving or setting values in the required units.

Note: Since both the data type and unit of a testbench port and Framework variable can be different it is necessary to convert the value of a framework variable when accessing a testbench port variable and vice versa. For these conversions the following main rules apply:

- Direction Framework → Testbench Port
 1. Framework datatype conversion into Float (e.g. `IntQuantity->FloatQuantity`)
 2. Unit conversion (always `FloatQuantity->FloatQuantity` in new unit)
 3. Testbench datatype conversion from Float (e.g. `FloatQuantity->IntQuantity`)
 4. Quantity to BaseValue conversion (e.g. `IntQuantity-> IntValue`)
- Direction Testbench Port → Framework
 1. BaseValue to Quantity conversion (e.g. `IntValue->IntQuantity`)
 2. Testbench datatype conversion to Float (e.g. `IntQuantity->FloatQuantity`)

3. Unit conversion (always FloatQuantity->FloatQuantity in new unit)
4. Framework datatype conversion (e.g. FloatQuantity->IntQuantity)

An example of creating and using a scalar Framework variable can be seen below:

C# Example Code:

```
IFloatVariable A = (IFloatVariable)FrameworkControl.CreateVariable("engine speed");
FloatQuantity aValue = A.Read();
double nativeValue = aValue.Value;
```

4.3.2.1 SCALAR VARIABLES

Scalar variables are used for representing scalar values. The following scalar variable classes are defined:

- UIntVariable
- IntVariable
- FloatVariable
- BoolVariable
- StringVariable

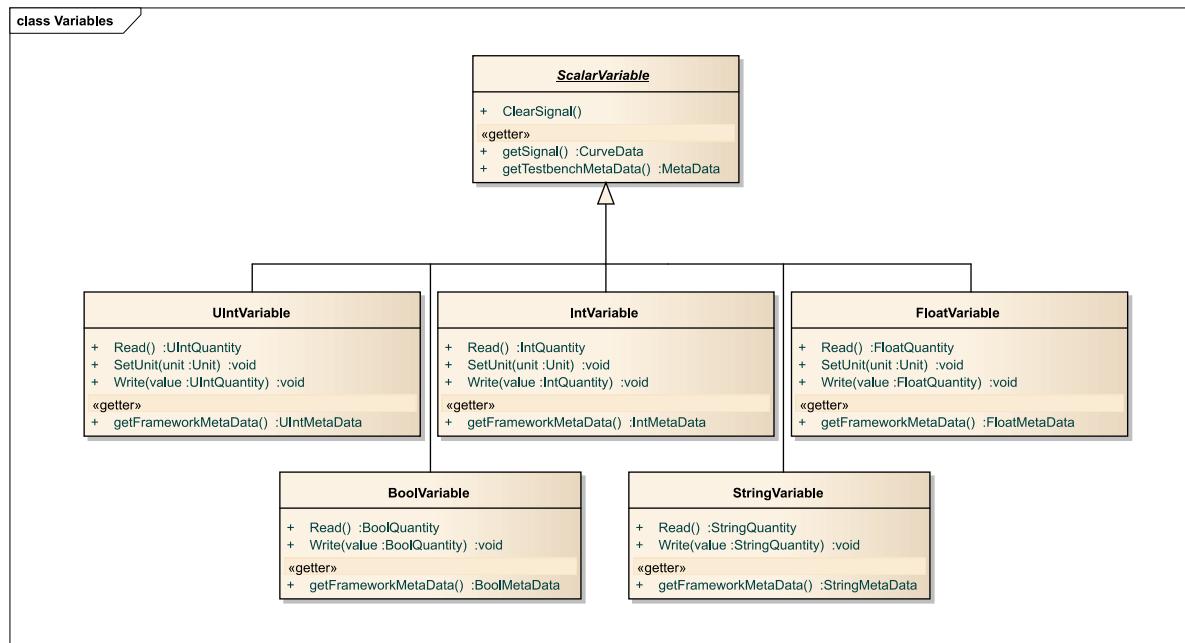


Figure 44: Scalar variable classes

All scalar variables have `Read()` and `Write()` methods for accessing their values. The property `Signal` of the scalar framework variables is used to retrieve captured data (see chapter [Measuring](#)). Scalar framework variables have also metadata information (including the default unit). This metadata information is stored for both sides of a framework scalar variable by the `FrameworkMetaData` (Framework variable) and `TestbenchMetaData` (Testbench variable) properties.

For scalar framework variables with unit information (**UIntVariable**, **IntVariable** and **FloatVariable**) the `SetUnit()` method can be used for changing the unit of the

Framework variable. This method will only change the unit information of the FrameworkMetaData property.

Subsequent Read() accesses will return a quantity with a value based on this newly set unit. Also accessing the getSignal property of a ScalarVariable will return a quantity with all values according to the new unit, regardless of being recorded before or after calling SetUnit().

Note: A scalar framework variable can be also mapped to one element of a Vector or Matrix Testbench variable (see chapter [Identifier Mapping](#)).

4.3.2.2 VECTOR VARIABLES

Vector variables are used for representing vector values (arrays). The following vector variable classes are defined:

- UIntVectorVariable
- IntVectorVariable
- FloatVectorVariable
- BoolVectorVariable
- StringVectorVariable

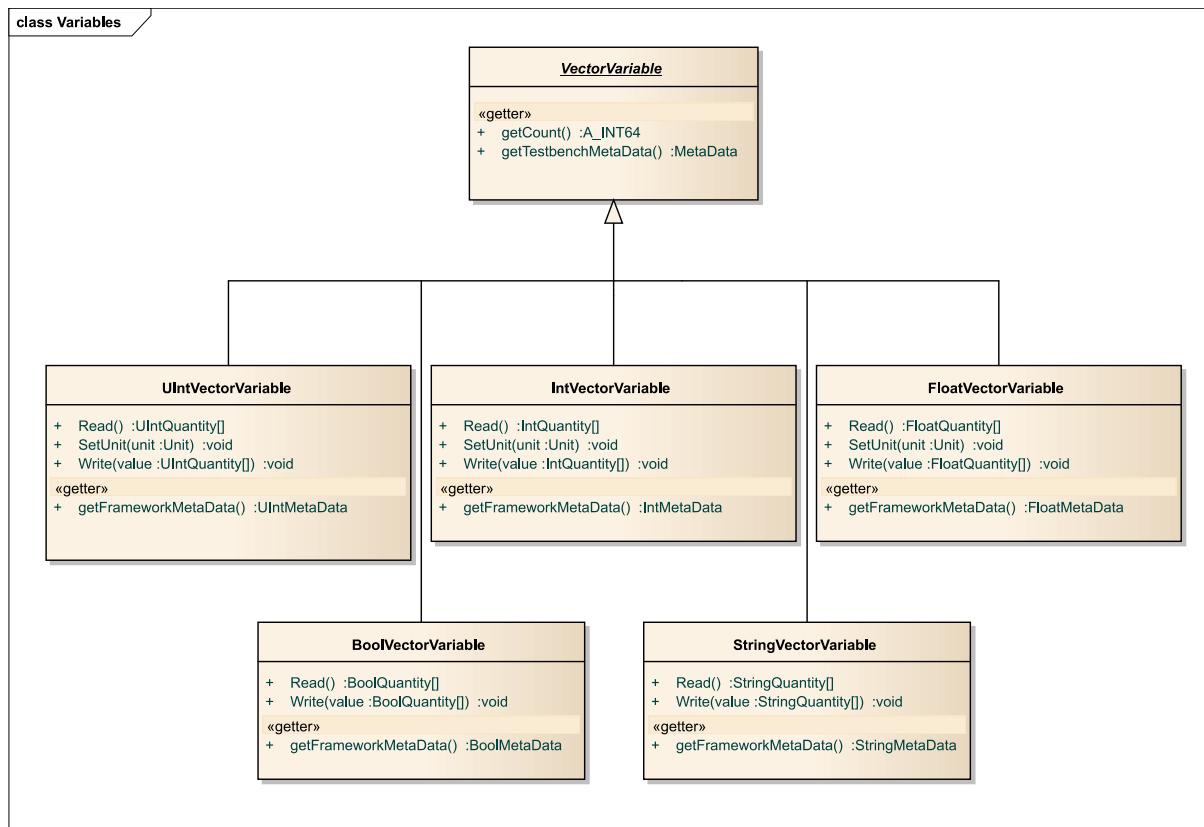


Figure 45: Vector variable classes

All vector variable classes provide the methods Read(), Write() and Item() for accessing the values of a vector variable. The Read() and Write() methods access all

elements of a vector variable as a 1-D array. By using the method `Item()` one element of a Vector variable at a specific index can be retrieved as a scalar variable.

Note: Indexing is 0 based for vector, matrix, etc... Variables of the XIL API standard.

In addition to these access methods all vector variable classes provide the property `Count` to return information about the number of elements of the vector variable.

The metadata information (including the default unit) of the vector framework variables is stored for both sides of a framework vector variable by the `FrameworkMetaData` (Framework variable) and `TestbenchMetaData` (Testbench variable) properties.

For vector framework variables with unit (`UIntVectorVariable`, `IntVectorVariable` and `FloatVectorVariable`) the `SetUnit()` method can be used for changing the unit of the Framework variable. This method will only change the unit information of the `FrameworkMetaData` property.

The quantity object of the Vector-Variable is a native, language specific 1-D array (see the programming language-specific type mapping rules in the document [5], [6]).

4.3.2.3 MATRIX VARIABLES

Matrix variables are used for representing matrices (2 dimensional arrays). The following matrix variable classes are defined:

- `UIntMatrixVariable`
- `IntMatrixVariable`
- `FloatMatrixVariable`
- `BoolMatrixVariable`
- `StringMatrixVariable`

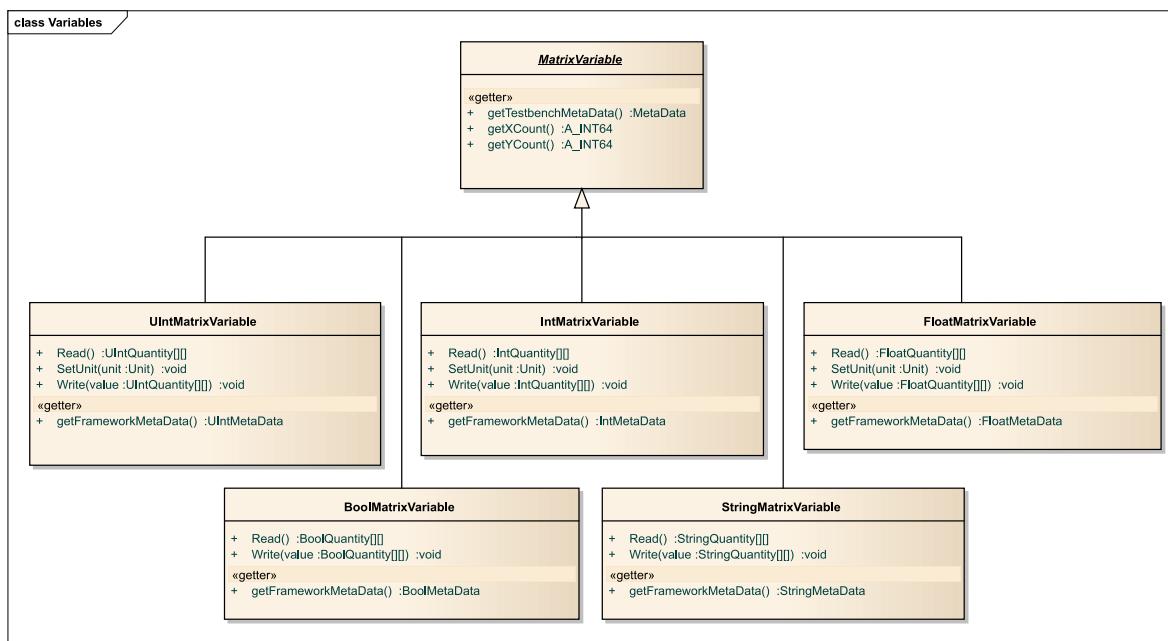


Figure 46: Matrix variable classes

All matrix variable classes provide the methods `Read()` and `Write()` for accessing the values of a matrix variable. The `Read()` and `Write()` methods access all elements of a matrix variable as a 2-D array.

In addition to these access methods all matrix variable classes provide the properties `XCount` and `YCount` to return information about the number of elements in the x and y dimensions of the matrix variable

The metadata information (e.g.: default unit) of the matrix framework variables is stored for both sides of a framework matrix variable by the `FrameworkMetaData` (Framework variable) and `TestbenchMetaData` (Testbench variable) properties.

For matrix framework variables with unit information (`UIntMatrixVariable`, `IntMatrixVariable` and `FloatMatrixVariable`) the `SetUnit()` method can be used for changing the unit of the Framework variable. This method will only change the unit information of the `FrameworkMetaData` property.

The quantity objects of the Matrix-Variable are native, language specific 2-D arrays (see the programming language-specific type mapping rules in the document [5], [6]).

4.3.2.4 CURVE VARIABLES

Curve variables are application oriented variables that are mainly used as calibration data of curve (1D table) values. Similar to the HiL API 1.x `CurveValue` classes the X vector data belonging to a curve variable must be either monotonously increasing or decreasing and the length of this X vector data must be equal to the length of the function value data.

The following curve variable classes are defined, which are all strongly-typed sub-classes of the common base class `CurveVariable`

- `FloatIntCurveVariable`
- `FloatFloatCurveVariable`
- `FloatBoolCurveVariable`
- `FloatStringCurveVariable`
- `StringIntCurveVariable`
- `StringFloatCurveVariable`
- `StringBoolCurveVariable`
- `StringStringCurveVariable`

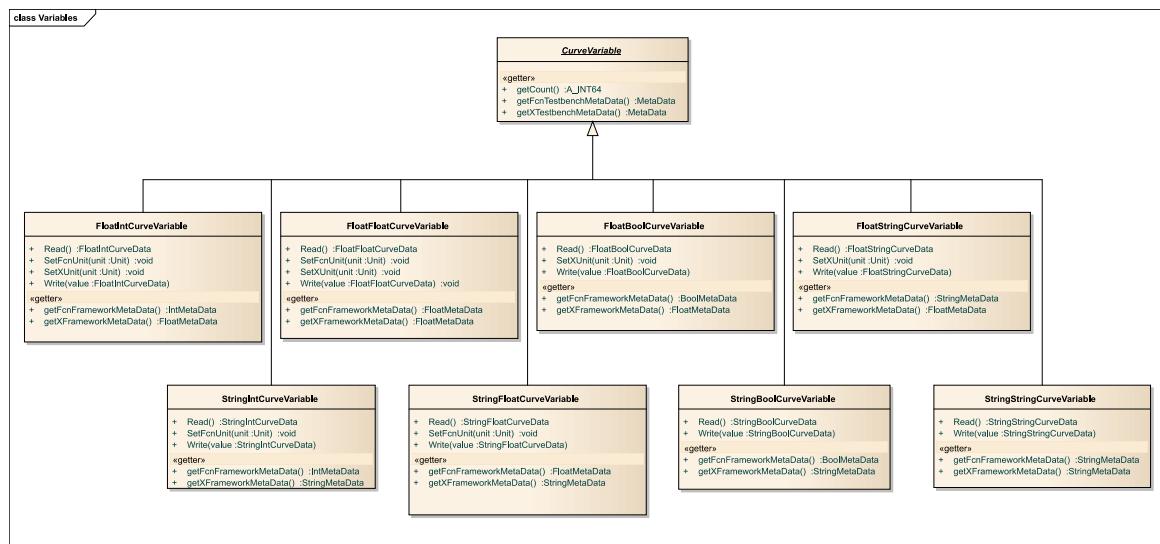


Figure 47: Curve variable classes

All curve variable classes provide the methods `Read()` and `Write()` for accessing their values. The `Read()` and `Write()` methods access all elements of a curve variable as a curve (1D table), for a detailed description of this data, see chapter [Complex Quantity Data Classes](#). In addition to these access methods all curve variable classes provide the property `Count` to return information about the number of elements of the curve variable.

Curve variable classes have metadata information. This metadata information is stored for both sides of a framework curve variable:

- the `XFrameworkMetaData` (Framework variable) and `XTestbenchMetaData` (Testbench variable) properties are used for the X vector data
- the `FCnFrameworkMetaData` (Framework variable) and `FCnTestbenchMetaData` (Testbench variable) properties are used for the functional value data.

A summary of the curve framework variables and the available methods for setting the unit of a curve framework variable is shown below (x = method is available, - = method is not available):

Table 6 setUnit() Methods of the Curve Framework variables

Curve Framework Variable	setXUnit()	setFcnUnit()
FloatFloatCurveVariable	x	x
FloatIntCurveVariable	x	x
FloatBoolCurveVariable	x	-
FloatStringCurveVariable	x	-
StringFloatCurveVariable	-	x
StringIntCurveVariable	-	x
StringBoolCurveVariable	-	-
StringStringCurveVariable	-	-

4.3.2.5 MAP VARIABLES

Map variables are similar to the curve variables. They are application oriented variables that are mainly used as calibration data of a map (2D table values). Similar to the HiL API 1.x MapValue classes the X and Y vector data belonging to a map variable must be either monotonously increasing or decreasing and the length of the X and Y vector data must be equal to the number of rows and columns of the function value data, respectively.

The following map variable classes are defined:

- FloatFloatIntMapVariable
- FloatFloatFloatMapVariable
- FloatFloatBoolMapVariable
- FloatFloatStringMapVariable
- FloatStringIntMapVariable
- FloatStringFloatMapVariable
- FloatStringBoolMapVariable
- FloatStringStringMapVariable
- StringFloatIntMapVariable
- StringFloatFloatMapVariable
- StringFloatBoolMapVariable
- StringFloatStringMapVariable
- StringStringIntMapVariable
- StringStringFloatMapVariable
- StringStringBoolMapVariable
- StringStringStringMapVariable

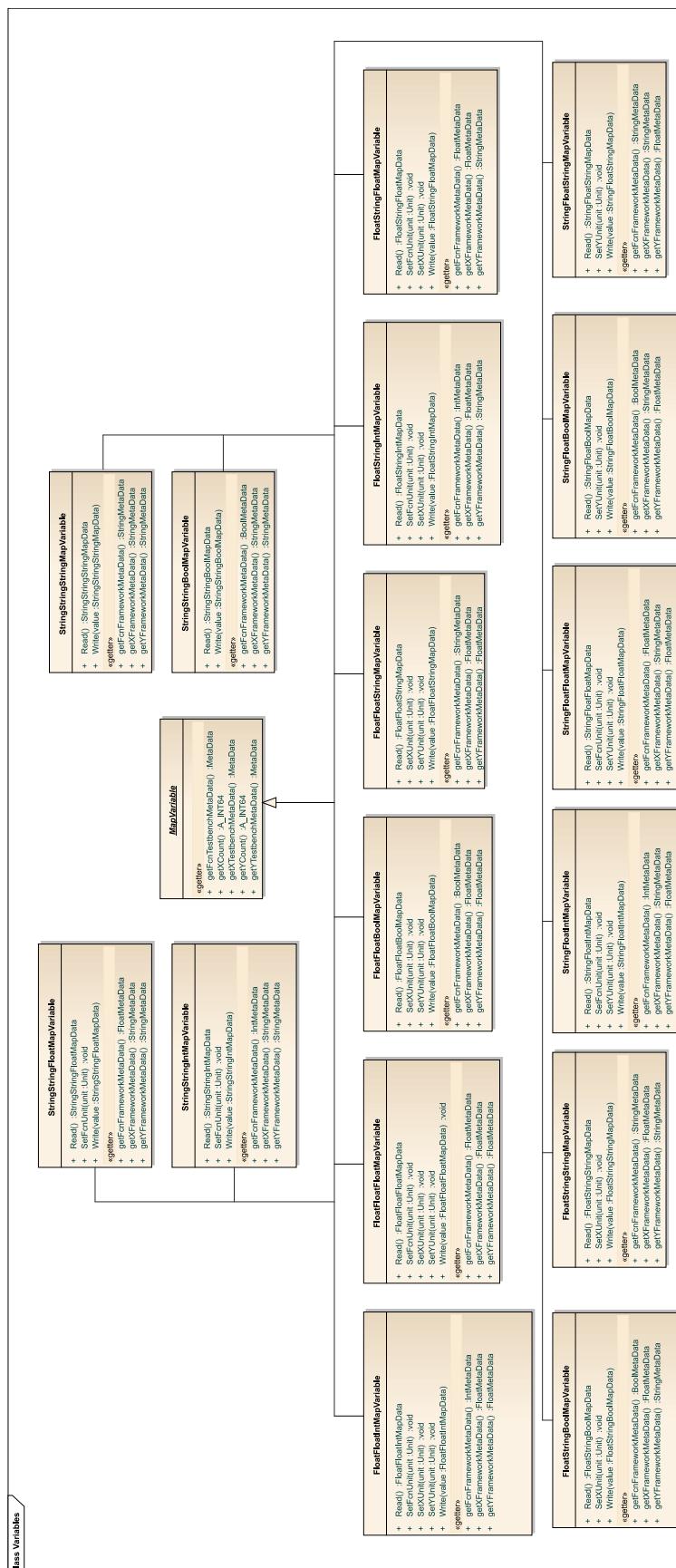


Figure 48: Map variable classes

All map variable classes provide the methods `Read()` and `Write()` for accessing their values. The `Read()` and `Write()` methods access all elements of a curve variable as a map (2D table), for a detailed description of this data see chapter [Complex Quantity Data Classes](#).

In addition to these access methods all map variable classes provide the properties `XCount` and `YCount` to return information about the number of elements in the x and y dimensions of the map variable

Map variable classes have metadata information. This metadata information is stored for both sides of a framework map variable:

- the `XFrameworkMetaData` (Framework variable) and `XTestbenchMetaData` (Testbench variable) properties are used for the X vector data
- the `YFrameworkMetaData` (Framework variable) and `YTestbenchMetaData` (Testbench variable) properties are used for the Y vector data
- the `FcnFrameworkMetaData` (Framework variable) and `FcnTestbenchMetaData` (Testbench variable) properties are used for the functional value data.

A summary of the map framework variables and the available methods for setting the unit of a map framework variable is shown below (x = method is available, - = method is not available):

Table 7 setUnit() Methods of the Map Framework variables

Framework Map Variable	<code>setXUnit()</code>	<code>setYUnit()</code>	<code>setFcnUnit()</code>
FloatFloatFloatMapVariable	x	x	x
FloatFloatIntMapVariable	x	x	x
FloatFloatBoolMapVariable	x	x	-
FloatFloatStringVariable	x	x	-
FloatStringFloatMapVariable	x	-	x
FloatStringIntMapVariable	x	-	x
FloatStringBoolMapVariable	x	-	-
FloatStringStringMapVariable	x	-	-
StringFloatFloatMapVariable	-	x	x
StringFloatIntMapVariable	-	x	x
StringFloatBoolMapVariable	-	x	-
StringFloatStringMapVariable	-	x	-
StringStringFloatMapVariable	-	-	x
StringStringIntMapVariable	-	-	x
StringStringBoolMapVariable	-	-	-
StringStringStringMapVariable	-	-	-

4.3.3 QUANTITIES

Quantities are used for representing values within the XIL API 2.0 framework. They encapsulate a value with its associated unit. (Quantities that represent string or boolean values have no units, for details, see Note in the following chapter). Quantity objects are

immutable. The unit of a quantity is described by the unit object belonging to this quantity. The XIL API standard defines scalar and complex quantity data classes to represent the values of the Framework variables.

4.3.3.1 SCALAR QUANTITY CLASSES

The scalar quantity classes of the XIL API are used when representing the values of the scalar, vector and matrix framework variables. The following basic ASAM data types are supported by the scalar quantity classes of the XIL API 2.0 framework:

- `A_UINT64` – `UIntQuantity` class
- `A_INT64` – `IntQuantity` class
- `A_FLOAT64` – `FloatQuantity` class
- `A_BOOLEAN` - `BoolQuantity` class
- `A_UNICODE2STRING` – `StringQuantity` class

Note: To provide consistency and allow the same mechanisms for accessing Boolean and string values within the XIL API framework, `BoolQuantities` and `StringQuantities` are defined. These quantities have no units.

More information about the above listed ASAM data types is available in [1]. When the UML model of the XIL API standard is transformed to different programming languages (Python [6], C# [5]), the ASAM data types are mapped to native, language specific data types.

The class diagram of the scalar quantity classes and the relation between quantities, units and physical dimensions is shown below:

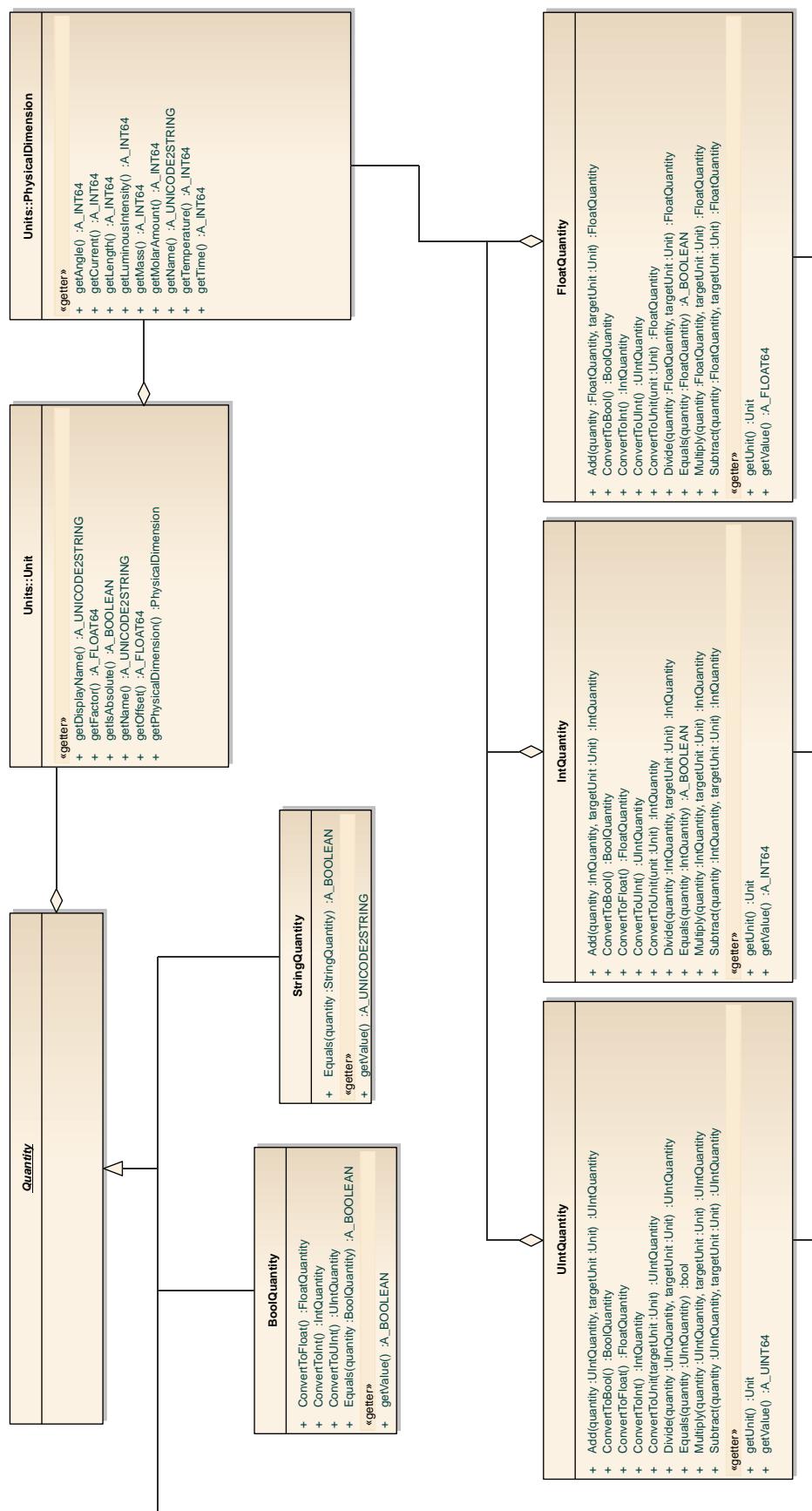


Figure 49: Scalar Quantity classes

The `value` property of a `Quantity` object can be used to retrieve the value of a `Quantity`. For quantities that represent numeric values (`UIntQuantity`, `IntQuantity` and `FloatQuantity`) the property `Unit` is used to retrieve information about the unit of the quantity.

The conversion methods: `ConvertToBool()`, `ConvertToInt()`, `ConvertToInt()`, `ConvertToFloat()` return a new quantity object based on the following conversion rules:

Table 8 Rules for the quantity conversion methods

Quantity	Conversion Method	Rule to apply	New Quantity	Unit of the new Quantity
BoolQuantity	<code>ConvertToFloat()</code>	True ---> 1 False --> 0	FloatQuantity	Empty Unit ¹
	<code>ConvertToInt()</code>	True ---> 1 False --> 0	IntQuantity	Empty Unit ¹
	<code>ConvertToUInt()</code>	True ---> 1 False --> 0	UIntQuantity	Empty Unit ¹
UIntQuantity	<code>ConvertToFloat()</code>	²	FloatQuantity	Same Unit as original Quantity
	<code>ConvertToInt()</code>	<code>MaxUInt / 2 --> MaxInt</code> <code>MaxUInt --> -1</code>	IntQuantity	Same Unit as original Quantity
	<code>ConvertToBool()</code>	<code>!0 --> True</code> <code>otherwise --> False</code>	BoolQuantity	No Unit
IntQuantity	<code>ConvertToFloat()</code>	²	FloatQuantity	Same Unit as original Quantity
	<code>ConvertToUInt()</code>	<code>MaxInt --> MaxUInt / 2</code> <code>-1 --> MaxUInt</code>	UIntQuantity	Same Unit as original Quantity
	<code>ConvertToBool()</code>	<code>!0 --> True</code> <code>otherwise --> False</code>	BoolQuantity	No Unit
FloatQuantity	<code>ConvertToInt()</code>	<code>float --> round --> int</code>	IntQuantity	Same Unit as original Quantity
	<code>ConvertToUInt()</code>	<code>float --> round --> uint</code>	UIntQuantity	Same Unit as original Quantity
	<code>ConvertToBool()</code>	<code>!0.0 --> True</code> <code>otherwise --> False</code>	BoolQuantity	No Unit

¹ An empty unit has all its elements set to 0 except for its property `Factor` which is set to 1.

² Have the baseline value more than 16 decimal places, the 16th decimal place will be rounded, e.g.: 18446744073709551615 --> 1.8446744073709552E+19

The method `ConvertToUnit(targetUnit: Unit)` converts the value of the quantity to the requested unit (`targetUnit`) and returns a new quantity object with this converted value and new unit.

The methods are described in the chapters [Calculation with quantity objects](#), [Data Type Conversion](#), [Unit Conversion](#) and [Usage of Mathematical Operations](#).

For representing the values of vector and matrix framework variables native language specific 1-D and 2-D arrays of the quantity objects are used.

4.3.3.2 COMPLEX QUANTITY DATA CLASSES

Complex quantity data classes and objects are used for representing the values of the application oriented complex framework variable types (Curve- and Map-Variables).

The following complex quantity data classes for Curve variables are defined:

- `FloatIntCurveData`
- `FloatFloatCurveData`
- `FloatBoolCurveData`
- `FloatStringCurveData`
- `StringIntCurveData`
- `StringFloatCurveData`
- `StringBoolCurveData`
- `StringStringCurveData`

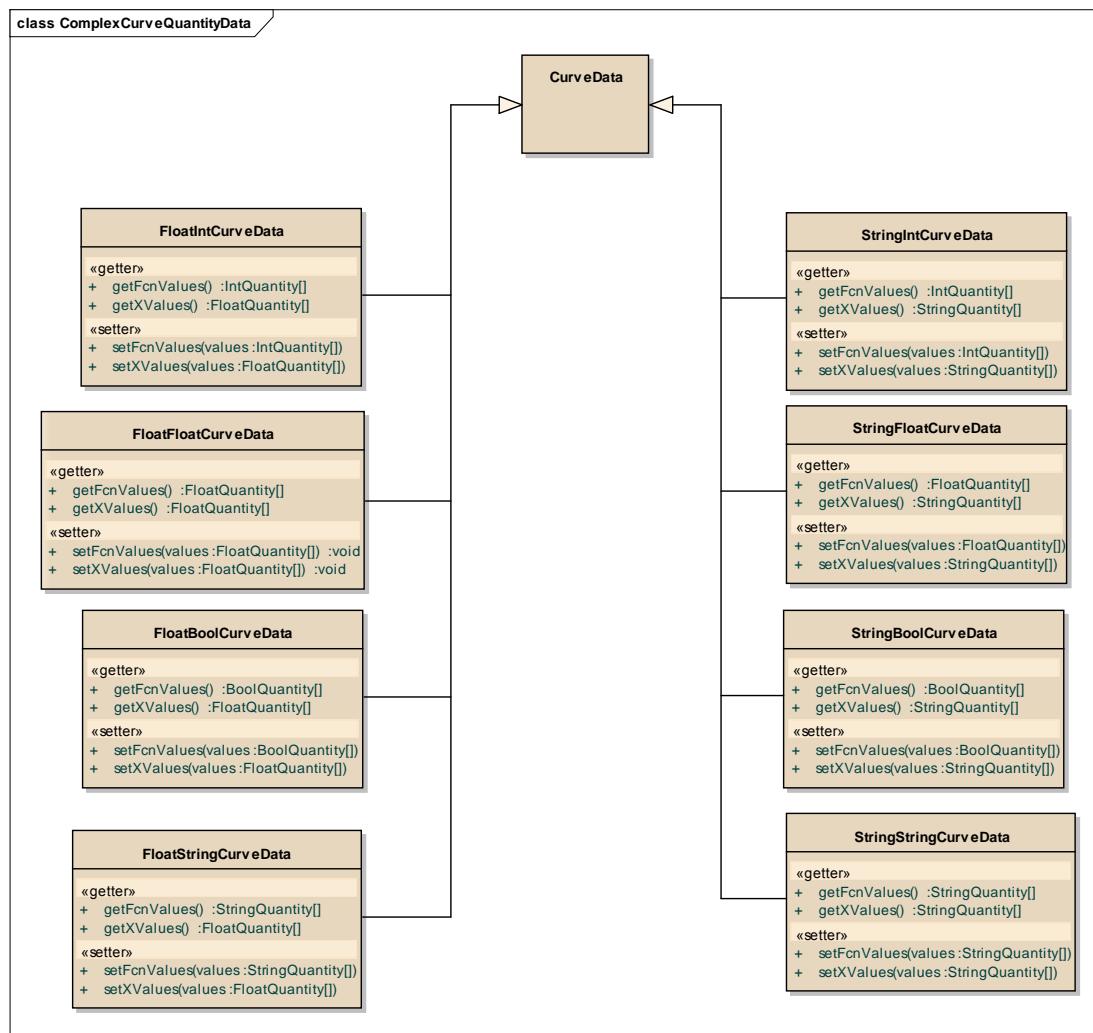


Figure 50: Complex Curve Quantity classes

All complex curve quantity classes provide the properties `FcnValues` and `XValues` to get and set the functional values and the X vector data of a curve quantity.

The X Vector and functional value data is represented as a native, language specific 1-D array.

The following complex quantity data classes for Map variables are defined:

- `FloatFloatIntMapData`
- `FloatFloatFloatMapData`
- `FloatFloatBoolMapData`
- `FloatFloatStringMapData`
- `FloatStringIntMapData`
- `FloatStringFloatMapData`
- `FloatStringBoolMapData`
- `FloatStringStringMapData`
- `StringFloatIntMapData`
- `StringFloatFloatMapData`
- `StringFloatBoolMapData`
- `StringFloatStringMapData`
- `StringStringIntMapData`
- `StringStringFloatMapData`
- `StringStringBoolMapData`
- `StringStringStringMapData`

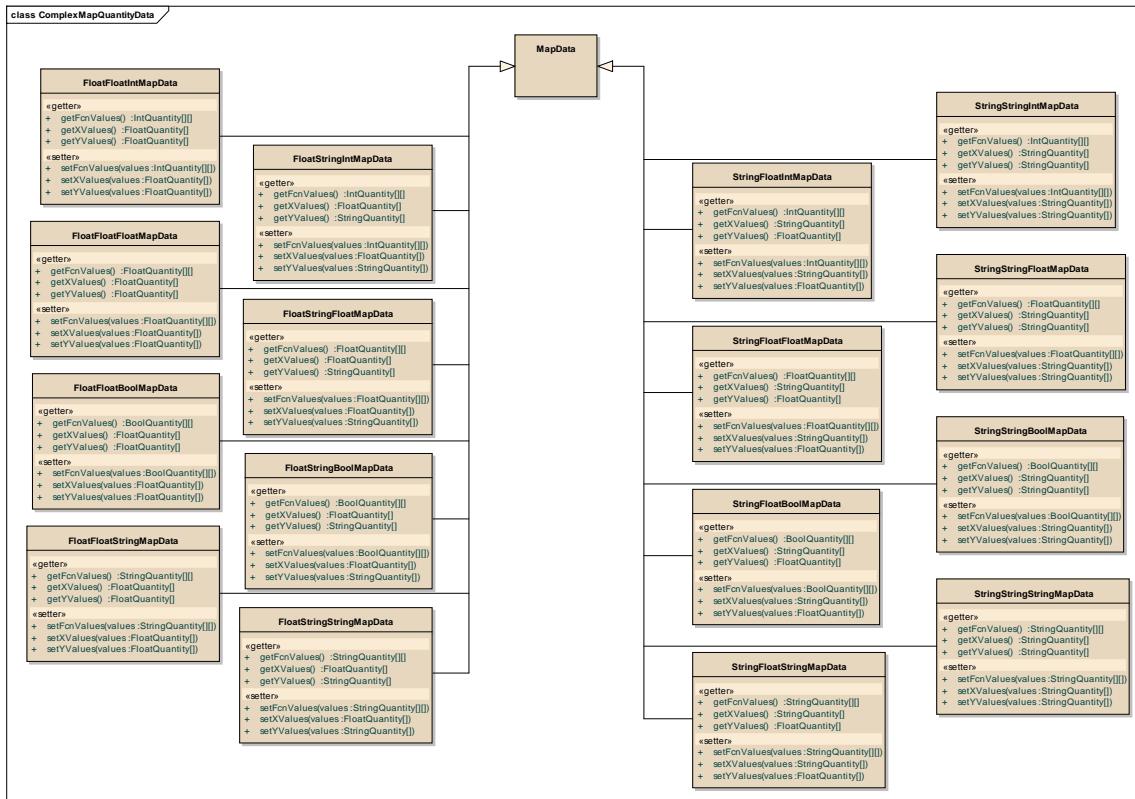


Figure 51: Complex Map Quantity classes

All complex map quantity classes provide the properties `FcnValues`, `XValues` and `YValues` to get and set the functional values, X vector and Y vector data of a map quantity. The X vector and Y vector is represented as a native, language specific 1-D array.

The functional value data is represented as a native, language specific 2-D array.

4.3.3.3 USAGE OF QUANTITIES

Quantity objects are created either implicitly by the `Read()` method (as return value) of a framework variable, the `Signal` property (to access the curve quantity result of a framework variable recording), or by using one of the factory methods of the `QuantityFactory` Class.



Figure 52: QuantityFactory class

When quantity objects are created implicitly using the `Read()` method of a Framework variable the unit for the quantity is set according to the unit of the Framework variable (`FrameworkMetaData` property of a `FrameworkVariable`).

Using one of the methods of the `QuantityFactory` class enables the explicit creation of a quantity object.

The factory methods for the scalar quantities that have no units (`CreateBoolQuantity()` and `CreateStringQuantity()`) have only one parameter (`value`), which is used for setting the value of the quantity to be created.

The factory methods for the scalar quantities with unit (`IntQuantity`, `UIntQuantity`, and `FloatQuantity`) have two parameters: `value` – for setting the value and `unitName` – for assigning a `Unit` object to the quantity to be created.

The units that are available in the Framework are defined in the mapping and identified using a string identifier (see [Example](#)). One of these string identifiers has to be used for selecting the unit of the quantity.

Native language-specific 1-D and 2-D arrays of the quantity objects are used by the `Vector` and `Matrix` Framework variables. This 1-D and 2-D quantity object arrays are created using the native language specific methods.

Note: It is possible to create a 1-D or 2-D quantity object array that contains quantities with different units.

Note: When using a 1-D or 2-D quantity object array with the `Write()` method of a `Vector` or `Matrix` Framework variable the physical dimension of the quantity object's unit must match the physical dimension of the Framework variable, otherwise an exception is generated.

The factory methods for the complex quantity data types will create an empty complex quantity data object. The elements of these complex quantity data objects (`Curve` and `Map`) have to be set by using the properties `FcnValues`, `XValues` and `YValues`.

Note: An empty complex quantity data object is the container for the complex quantity data object with empty `FcnValues`, `XValues` and `YValues` arrays (array length = 0).

The following sequence diagram shows the usage of quantity objects with the `Read()` and `Write()` methods of framework variables.

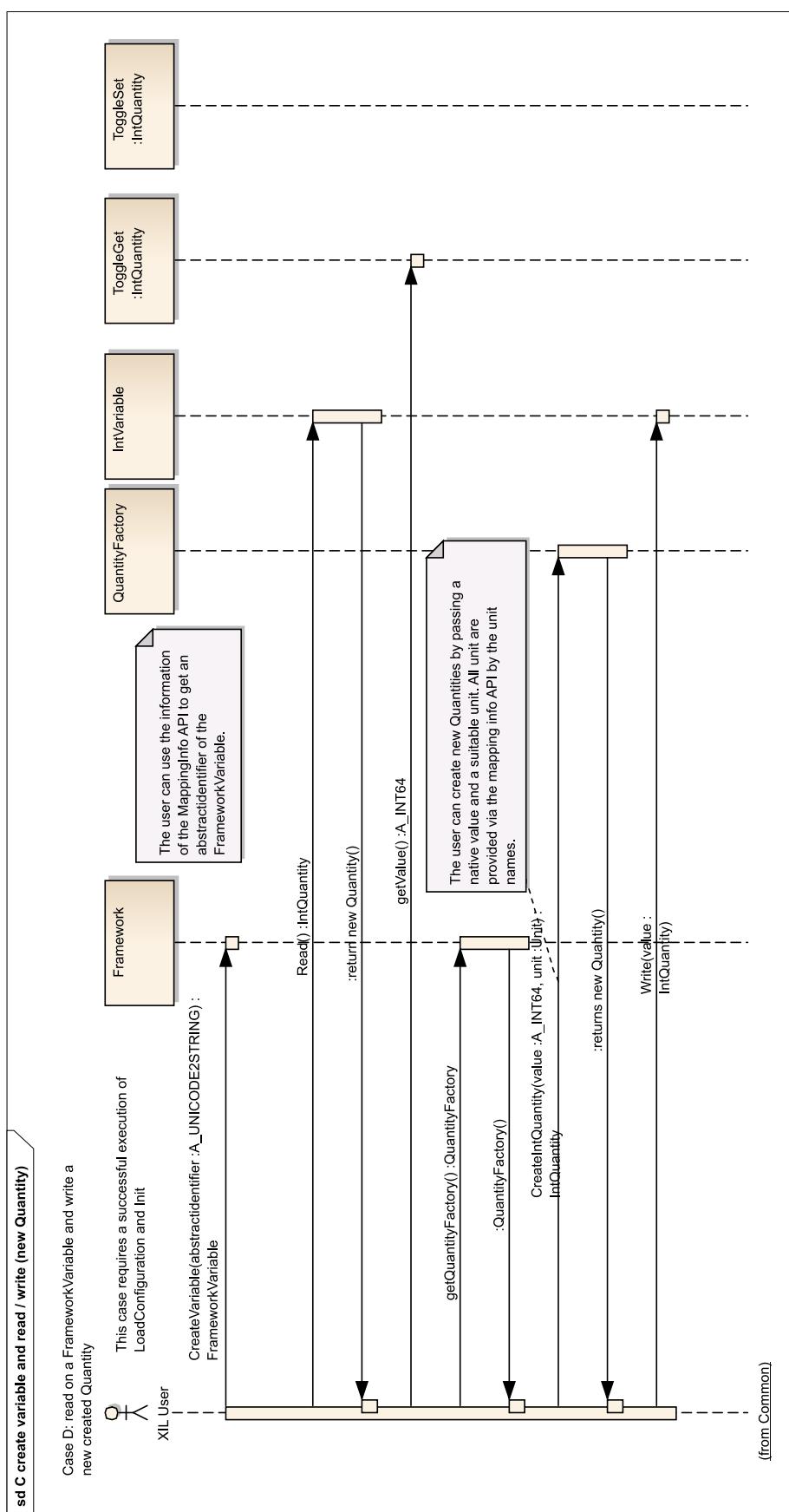


Figure 53: Using quantities with Framework variables

Besides using the quantity objects with the `Read()`/`Write()` methods of Framework variables it is also possible to do calculations with them (see chapter [Calculation with quantity objects](#) and following).

4.3.4 UNIT AND PHYSICAL DIMENSIONS

As it has already been mentioned before, units are represented by unit objects in the XIL API framework. These unit objects are used mainly in combination with a value to form quantities. The idea behind using unit objects is to provide a flexible, extendable mechanism for defining units. To achieve this, a unit object consists of a factor and an offset in combination with a physical dimension object.

For the description of a physical dimension, 7 base physical dimensions from the SI standard plus an additional 'angle' dimension are used. The base dimensions and their corresponding symbols are: electric current (I), length (L), luminous intensity (J), mass (M), molar amount (N), temperature (Θ), and time (T). The `PhysicalDimension` class contains the exponents for these 7 SI dimensions (plus an 'angle' dimension) to describe a general physical dimension. For example, the unit [m/s] has the physical dimension L/T, where the exponent for the dimension length (unit [m]) is 1 and the exponent for the dimension time (unit [s]) is -1. This concept allows the definition of physical dimension objects that can be reused with different units. By using different factor and offset values with the same physical dimension object different units for a physical value can be defined. This is shown in the table below:

Table 9 Definition of Units and Physical Dimensions

Unit	PhysicalDimension	DisplayName	offset-si-to-unit	factor-si-to-unit	LengthExp	MassExp	TimeExp	CurrentExp	TemperatureExp	MolarAmountExp	LuminousIntensityExp	AngleExp
Dist_km	Distance	[km]	0,000	0,001	1	0	0	0	0	0	0	0
Dist_m	Distance	[m]	0,000	1,000	1	0	0	0	0	0	0	0
Force_kN	Force	[kN]	0,000	0,001	1	1	- 2	0	0	0	0	0
Force_N	Force	[N]	0,000	1,000	1	1	- 2	0	0	0	0	0
Temp_Cel	Temperature	[°C]	-273,150	1,000	0	0	0	0	1	0	0	0
Temp_Far	Temperature	[°F]	-459,670	1,800	0	0	0	0	1	0	0	0
Vel_Kl_p_Nw	Velocity	[Klafte r/Nach t-wache]	0,000	7659,574	1	0	- 1	0	0	0	0	0
Vel_km_h	Velocity	[km/h]	0,000	3,600	1	0	- 1	0	0	0	0	0
Vel_mph	Velocity	[miles/h]	0,000	2,2369	1	0	- 1	0	0	0	0	0
Vel_mps	Velocity	[m/s]	0,000	1,000	1	0	- 1	0	0	0	0	0
Angle_deg	Angle	[deg]	0,000	57,296	0	0	0	0	0	0	0	1
Angle_rad	Angle	[rad]	0,000	1,000	0	0	0	0	0	0	0	1
Area_m²	Area	[m²]	0,000	1,000	2	0	0	0	0	0	0	0
Time_h	Time	[h]	0,000	2,778	0	0	1	0	0	0	0	0

The following rule describes how to convert the physical values for different units:

Rule of conversion: **UnitValue = BaseUnitValue * factor-si-to-unit + offset-si-to-unit**

Note: Special conversion considerations have to be applied in the case of a framework variable that represents “relative” values. In this case the offset-si-to-unit in the conversion rule has to be ignored.

An example for a framework variable representing a “relative” value could be a variable that represents a temperature difference value. In this case converting the difference value of 25 °C to K (Kelvin) must provide 25 K as a result (and not 298 K).

The `IsAbsolute` property of a unit defines if a framework variable represents “absolute” or “relative” values.

The mapping (see chapter [Mapping](#)) contains information about the physical dimensions and units for the XIL API Framework.

4.3.4.1 UNIT CLASS

The information stored in the Unit class is used for allowing the conversion of quantities with different units.

4.3.4.2 PHYSICAL DIMENSION CLASS

The physical dimension class is used for the comparison and for calculations of quantities.

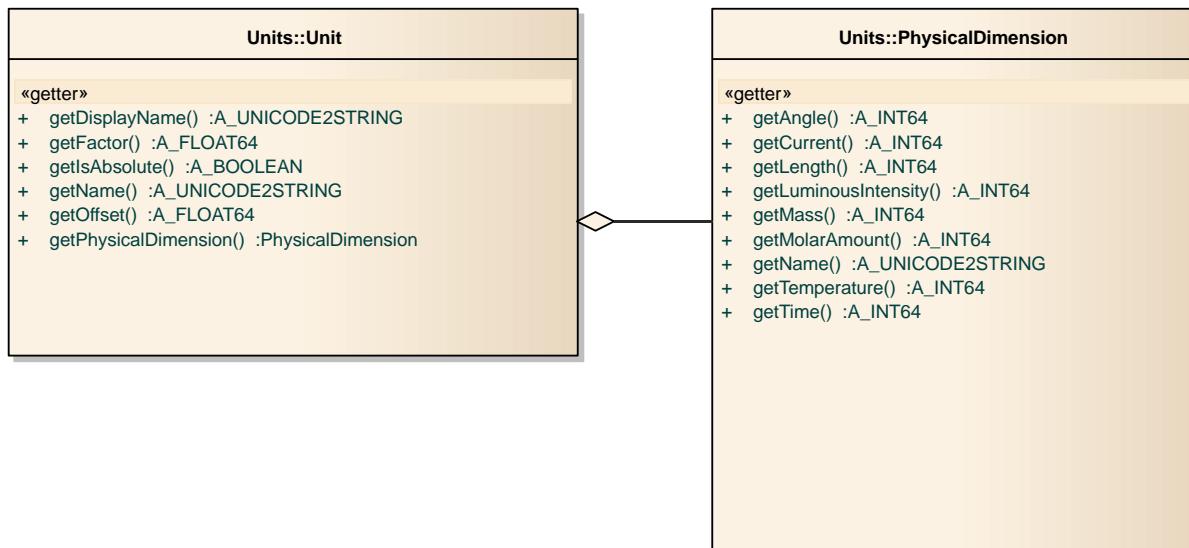


Figure 54: The Unit and PhysicalDimension classes

4.3.5 METADATA

For each framework variable, MetaData objects are defined for both sides of a test environment (Testcase/framework and Testbench). The MetaData class is a container class that holds the following information:

- to store the unit information of a framework variable (`Unit` property)
- to define the valid value range for a framework variable (`Min/Max` properties)
- to define if a variable is readable (`IsReadable` property)
- to define if a variable is writable (`IsWritable` property)

The information is defined in the mapping and used for the instantiation of a framework variable object.

Note: The XIL API standard doesn't specify a consistency check for the data stored in the MetaData objects.

Note: The MappingInfoAPI offers a possibility to check if the Meta data in the XML mapping file is filled or not filled.

Note: The XIL API standard doesn't require the evaluation of the meta-data information about the valid value range (`Min/Max` properties) and readability and writability (`IsReadable/IsWritable` properties) of a framework variable when accessing a framework variable. The `IsReadable` property means you can read and recording this framework variable. The `IsWritable` property indicates it is possible to write and stimulate the framework variable. The default value from these properties is true.

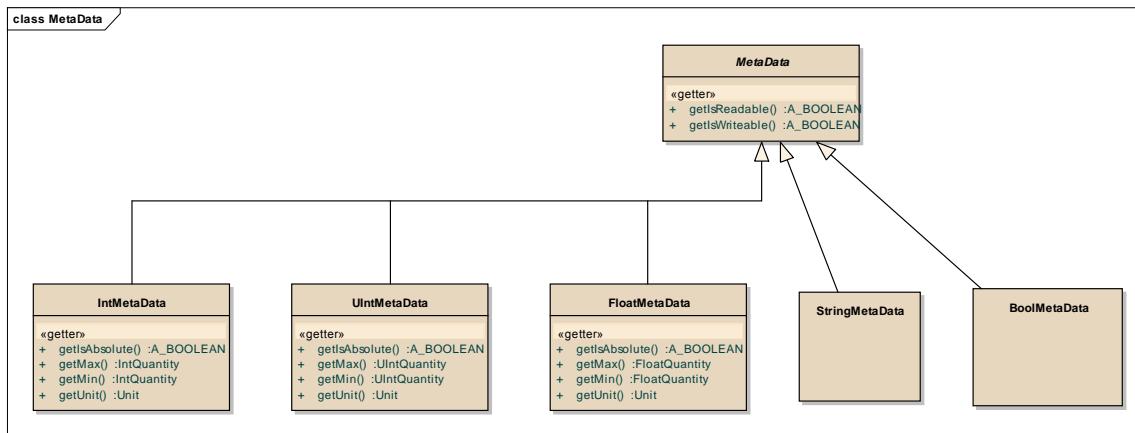


Figure 55: The MetaData classes

4.3.6 CALCULATION WITH QUANTITY OBJECTS

4.3.6.1 OVERVIEW

Since quantities are used for representing values within the XIL-API Framework, it is possible to perform calculations on quantities. However, it is important to mention that calculations are only supported for quantities that represent numerical values:

- UintQuantity
- IntQuantity
- FloatQuantity

All other quantities (BoolQuantity, StringQuantity and any complex quantity data classes) do not support calculations.

The following table summarizes the mathematical operations that are supported for the calculations on quantities:

Table 10 Mathematical Operations

Operation	Method name	Content
Addition	Add	A + B
Subtraction	Subtract	A - B
Multiplication	Multiply	A * B
Division	Divide	A / B

The mathematical operations of a quantity object are available through the corresponding methods of the quantity object. Please, note that mathematical operations are supported only for quantities of the same type. All mathematical operation methods require the specification of a target unit as a method parameter. This target unit has to be defined in the mapping file, otherwise the operation will be rejected with an exception.

The result of a calculation is a new quantity object, which is returned by the mathematical operation method of the corresponding quantity. This new quantity object (result) has the same type as the quantity objects of the mathematical operation. Its unit has been specified as target unit for the mathematical operation.

If the resulting value of the calculation exceeds the range, which may be represented for this new quantity object (result), an exception is thrown.

4.3.6.2 ADDITION AND SUBTRACTION

Addition and subtraction operations can only be realized with quantities, whose units are based on identical physical dimensions. Furthermore, the target unit specified for the result of the calculation must be based also on the same physical dimension.

The calculation is executed on the basis of base SI units. The value of each quantity object of the calculation will be converted before executing the calculation into the respective SI unit (see chapter [Unit Conversion](#)). Afterwards, the result will be converted into the target unit. The target unit also defines whether the calculated result will be used as absolute or relative value in further calculations.

4.3.6.3 MULTIPLICATION AND DIVISION

These calculations will be also executed on the bases of SI Units. Multiplication and division operations can be realized on the base of different physical dimensions.

All operations, which are used during runtime, have to be predefined in the computation table as part of the mapping file. If the physical dimensions of the operands and the result do not fit with one definition in the computation table, an exception is thrown. As an additional plausibility check, the consistency of physical dimensions with the target dimension must be proven. This is used as a fallback, if the computation table contains inconsistent entries.

$$\begin{aligned} \text{Product} &= 1.\text{Factor} * 2.\text{Factor} \\ \text{Quotient} &= \text{Dividend} / \text{Divisor} \end{aligned}$$

The exponents of complex physical dimensions can be obtained from simpler physical dimensions by using the calculation rules for exponents:

$$\begin{aligned} A^n * A^m &= A^{(n+m)} \\ A^n / A^m &= A^{(n-m)} \end{aligned}$$

Where A is a physical dimension (e.g., length), n and m are integers, and '^' is the power operator (raising the left operand to the power given by the right operand).

Example:

In square brackets are the relevant physical dimensions A are shown.

```
Area_in_m² = Dist_in_m * Dist_in_m
[Length = 2] = [Length = 1] + [Length = 1]
[Length^(1+1)]  
  
Dist_in_m = Area_in_m² / Dist_in_m
[Length = 1] = [Length = 2] - [Length = 1]
[Length^(2-1)]  
  
Vel_km_h = Dist_km / Time_h
[Length = 1, Time = -1] = [Length = 1, Time = 0] - [Length = 0, Time = 1]
[Length^(1-0)] and [Time^(0-1)]
```

4.3.6.4 COMPUTATION TABLE

The computation table is part of the mapping file (see chapter [Mapping](#)). Together with the XIL standard a computation table is published, which provides the most common equations of automotive business.

First of all this table defines all physical dimensions. Based on these physical dimensions, the user has to define units, which are not SI base units (e.g. miles per hour). The equations in the computation table do not depend on units, they only define the target physical dimension. For example: Force * Length => Torque. This example shows, that a second, different entry in the table will make sense: Force * Length => Energy.

The XIL user performs computations based on quantity objects with their units. The numerical software implementation first checks the physical dimensions according to the provided units, whether the operation of such physical quantities is defined in the computation table. Together with the target unit, which is provided with the computation method, in the example above it can be proved, whether the resulting quantity is meant as a torque or as energy.

The mapping computation table defines the multiplication and division in one table. The division represents the inverse function of the multiplication. Therefore note the column headings.

The computation table provided with the XIL standard is based on commonly used physical identifiers. If no common identifiers are known, the names were chosen according to the mathematical definition (e.g. 3rd derivative of velocity). In general, the concept of user defined identifiers can also be used here. This is illustrated in the following example:

Predefined common physical identifiers

Table 11 Dimension definitions with common physical identifiers

Physical Dimension	LengthExp	CurrentExp	LuminousIntensityExp	MassExp	MolarAmountExp	TemperatureExp	TimeExp	AngleExp
Distance	1	0	0	0	0	0	0	0
Time	0	0	0	0	0	0	1	0
Velocity	1	0	0	0	0	0	-1	0
Acceleration	1	0	0	0	0	0	-2	0
Jerk	1	0	0	0	0	0	-3	0
3rd derivative of velocity	1	0	0	0	0	0	-4	0
Area	2	0	0	0	0	0	0	0
Volume	3	0	0	0	0	0	0	0
kinematic viscosity	2	0	0	0	0	0	-1	0
Volumetric flow rate	3	0	0	0	0	0	-1	0

Table 12 Computation table with common physical identifiers

Product / Dividend	1. Factor / Quotient	2. Factor / Divisor
Distance	Velocity	Time
Velocity	Acceleration	Time
Acceleration	Jerk	Time
Jerk	3rd derivative of velocity	Time
Area	Distance	Distance
Area	Kinematic viscosity	Time
Volume	Area	Distance
Volume	Volumetric flow rate	Time

User defined identifiers

Note: As an example of user defined identifiers, a different language is chose (here: German). In general, any other user defined identifiers may be used, e.g. "speed" instead of "Velocity".

Typically, user defined identifiers will be introduced for unit identifiers (e.g. "Speed_in_miles_per_hour"), which have to be part of the mapping file. These units refer to the identifiers of physical dimensions.

However, the concept of user defined identifiers provides a high degree of flexibility.

Table 13 Dimension definitions with user defined identifiers (example German)

Physical Dimension	LengthExp	CurrentExp	LuminousIntensityExp	MassExp	MolarAmountExp	TemperatureExp	TimeExp	AngleExp
Länge	1	0	0	0	0	0	0	0
Zeit	0	0	0	0	0	0	1	0
Geschwindigkeit	1	0	0	0	0	0	-1	0
Beschleunigung	1	0	0	0	0	0	-2	0
Ruck	1	0	0	0	0	0	-3	0
3. Ableitung d. Geschwindigkeit	1	0	0	0	0	0	-4	0
Fläche	2	0	0	0	0	0	0	0
Volumen	3	0	0	0	0	0	0	0
kinematische Viskosität	2	0	0	0	0	0	-1	0
Volumenstrom	3	0	0	0	0	0	-1	0

Table 14 Computation table with user defined identifiers

Product / Dividend	1. Faktor / Quotient	2. Faktor / Divisor
Länge	Geschwindigkeit	Zeit
Geschwindigkeit	Beschleunigung	Zeit
Beschleunigung	Ruck	Zeit
Ruck	3. Ableitung d. Geschwindigkeit	Zeit
Länge	Länge	Länge
Länge	kinematische Viskosität	Zeit
Volumen	Fläche	Länge
Volumen	Volumenstrom	Zeit

4.3.6.5 COMPARISON

A comparison is defined for all quantity datatypes. However, for datatypes UINT, INT, and FLOAT, the operands need to have an identical physical dimension. The comparison result is always of datatype BOOLEAN.

Table 15 Comparison

Operation	Method name	Content
Comparison	Equals	A == B

4.3.7 DATA TYPE CONVERSION

Datatype conversions are carried out using Quantity objects.

Table 16 Data Type Conversions on Quantities

UINT	INT	FLOAT	BOOLEAN
ConvertToBool	ConvertToBool	ConvertToBool	
ConvertToInt		ConvertToInt	ConvertToInt
	ConvertToUInt	ConvertToUInt	ConvertToUInt
ConvertToFloat	ConvertToFloat		ConvertToFloat

The returned object is a new Quantity object. For datatype conversions apply the same rules as in [Rules for the quantity conversion methods](#).

4.3.8 UNIT CONVERSION

Unit conversions are carried out using Quantity objects. Unit conversion is defined only for units with identical physical dimension. Otherwise, an exception is thrown. They are only defined for Quantities of datatype UINT, INT, or FLOAT. The numerical calculations are always done based on a FLOAT datatype internally, regardless of the quantity's datatype in order to maintain the highest precision. However, precision can get lost if an integer quantity

is converted and the conversion covers magnitudes (e.g. a length of mm to m). The quantity returned is a new object with the desired unit.

The target unit defines if the calculated result will be used as absolute or relative value in further processings.

The following algorithm will be used for this conversion:

- Quantities with absolute values (`IsAbsolute` property is TRUE) :

$$U_T = \frac{U_A - O_A}{F_A} F_T + O_T$$

- Quantities with relative values (`IsAbsolute` property is FALSE) :

$$U_T = \frac{U_A F_T}{F_A}$$

Where:

U_T = For the calculation specified Target Unit

F_T = Factor value belonging to the Target Unit

O_T = Offset value belonging to the Target Unit

U_A = Unit of the Quantity whose value has to be converted

F_A = Factor value belonging to the unit of the Quantity whose value has to be converted

O_A = Offset value belonging to the unit of the Quantity whose value has to be converted

Example:

Length in Miles (Target Unit) := Length in m (Add) + Length in mm

Note: Unit conversion is defined only for units with identical values of the `IsAbsolute` property. Otherwise, an exception is thrown.

4.3.9 USAGE OF MATHEMATICAL OPERATIONS

4.3.9.1 ABSOLUTE AND RELATIVE

The `IsAbsolute` property of a unit determines, whether the offset is used for unit conversion. An absolute temperature of 20°C equals 293K. A relative temperature (temperature difference) of 20°C (`IsAbsolute` = FALSE) equals 20K.

In general mathematical calculations it is not possible to automatically propagate the `IsAbsolute` properties of the operands to the result. Due to this fact, the resulting quantity will be interpreted according to the `IsAbsolute` property of the target unit, which is provided for any mathematical operation.

4.3.9.2 NEUTRAL UNIT

For mathematical or physical constants, results of mathematical operations, as well as datatype conversion of BOOLEAN quantities a neutral unit is required. The neutral unit is not part of the mapping file. It is predefined in the framework as follows:

- Dimension Identifier: NO_DIM
- Unit Description: NO_UNIT
- Unit DisplayName: [] (empty)
- IsAbsolute: TRUE

Dimension less quantities (NO_UNIT) may be part of a product or fraction without being part of the computation table. The resulting unit will be the same as the other factor or part of the fraction.

In addition, XIL users may define their own units without physical dimension, e.g. "TorqueRatioUnit" or "no_apples". Such units have to be part of the vendor specific unit definitions. In contrast to NO_UNIT, user specific units have to part of the computation table, if they should be used in computations.

Note: In comparison operations or conversions, NO_UNIT and different user defined units without physical dimension will be treated as different physical dimensions (e.g. "no_apples" and "no_oranges").

4.3.9.3 MATHEMATICAL AND PHYSICAL CONSTANTS

Quantities can be generated by the Quantity Factory for any datatype. The value has to be provided (e.g. 3.14159 for π), as well as a unit, if the datatype is either INT, UINT, or FLOAT. For constants also the unit NO_UNIT can be used.

4.3.9.4 EXAMPLE

The XIL user can utilize computations, as well as unit and datatype conversions in his test context. Due to this fact, it is not necessary to switch the unit for a framework variable permanently. Using the provided conversion methods, quantities of different kind can be combined in a computation.

```
1 // Declare a FloatQuantity.  
2 IFloatQuantity qFloat;  
3  
4 // Create IntQuantity and UIntQuantity with initial values and units.  
5 IIntQuantity qInt = myQuantityFactory.CreateIntQuantity(10, lengthUnit);  
6 IUIntQuantity qUInt = myQuantityFactory.CreateUIntQuantity(22, lengthUnit);  
7  
8 // Convert both factors explicitly to float Quantities (based on the target  
9 // type).  
10 qFloat = (qInt.ConvertToFloat()).Multiply(qUInt.ConvertToFloat(),  
11 areaUnit);  
12  
13 // Convert the product explicitly to float after the calculation.  
14 qFloat = qInt.Multiply(qUInt.ConvertToInt(), areaUnit).ConvertToFloat();  
15  
16 // Use the first factor Quantity as target Quantity.  
17 qInt = qInt.Multiply(qUInt.ConvertToInt(), areaUnit);  
18  
19 // Explicitly data type conversion.  
20 qFloat = qInt.ConvertToFloat();
```

4.4 MEASURING

4.4.1 MOTIVATION

In the context of XIL API measuring describes the process of collecting time traces of variables, e. g. for analyzing the effects of changes within simulation models and ECUs. This chapter shows the measuring capabilities of XIL API. This comprises the configuration of sophisticated triggers, the measurement of variables of different ports in parallel (e. g. simulation model and ECU), and the assembling of the measured data into a coherent result (e. g. MDF4 file).

XIL API uses the [Framework Variables](#) in order to configure measurements and access measurement data. However, these variables must be of type `ScalarVariable`. The Measuring of entire complex data types such as `VectorVariable`, `MatrixVariable`, `CurveVariable` or `MapVariable` is not supported explicitly. Nevertheless, single elements of these complex data types can be mapped to a single `ScalarVariable`, in order to get the corresponding time traces during the measuring process.

Measuring in XIL API consists of two different processes: acquisition and recording.

The acquisition process encapsulates the data acquisition from different testbench ports to the framework. It collects all measurement data into a single data pool, together with a common time basis. This time basis is set to 0 when acquisition starts. The acquisition can be configured by different downsampling and triggering options. The acquisition process has to be started and stopped explicitly.

Recordings are necessary to make subsets of the acquisition data available to the user. A recording result (i.e., a subset of the acquisition data) can exist either in memory as a `RecorderResult` object (similar to the `CaptureResult` known from HIL API 1.0) or as a file (e. g. in MDF4 format). The test developer can use these recording results for further analysis within the test case. Again, configuration options can be used in order to focus on relevant parts of the acquired data.

4.4.2 SETTING UP AN ACQUISITION

4.4.2.1 INTRODUCTION

The XIL API acquisition process collects the time traces of all variables of interest from different testbench ports into a common data pool, which is held in the framework. All testbench ports can be employed, which support capturing (ECUMPort, MAPort, NetworkPort). All time traces will have a common time basis with a common start time and end time. However, the synchronicity is limited to the different delay times between framework and the different testbench ports. For details see chapter [Synchronized Data Acquisition](#).

To setup the *Acquisition*, XIL API provides two interfaces (see [Figure 56](#)).

4.4.2.2 USING THE ACQUISITION INTERFACE

The test developer obtains the *Acquisition* object via the Framework's attribute *Acquisition*.

Note: The *Acquisition* is a single instance per Framework. When first accessing *Acquisition*, the Framework creates the instance.

The *Acquisition* interface has methods to configure, start, stop and ClearConfiguration the acquisition – and two factory methods to create recorder objects. The acquisition setup consists of a list of *AcquisitionConfiguration* objects, each of which describes a different measurement configuration (trigger conditions, measurement raster) for a specified set of variables from either the same or different testbench ports. Details are described in the following section.



Figure 56: Acquisition and AcquisitionConfiguration

Start of Acquisition

The Framework acquisition is started by calling the method `Start()` of the `Acquisition` object.

While the acquisition is running, the framework periodically fetches measurement data from the testbench ports. This results in a data stream from the testbench ports to the `Acquisition` object according to its configuration.

Note: While the acquisition is running, no re-configuration is allowed. The user has to stop the current acquisition first for reconfiguring.

Stop of Acquisition

With the `Stop()` method the framework acquisition is stopped.

The stop of the acquisition terminates the data stream between testbench ports and the `Acquisition` object. All data samples, which have been captured in testbench ports up to the stop call will be transferred to the framework.

Note: Stop of acquisition implicitly terminates all running Recorder objects.

ClearConfiguration

The `ClearConfiguration()` method can be used to clear the configuration of the `Acquisition` object. Afterwards, a new configuration has to be set.

4.4.2.3 ACQUISITION STATES

The Acquisition interface has two states: eSTOPPED and eSTARTED.

The following table shows the methods, which are allowed in each state indicated by an “x” sign. If calls are not allowed, an exception is thrown. Most of the calls are allowed in all states.

Table 17: Acquisition States

Acquisition Operation	eSTOPPED	eSTARTED
Method CreateAcquisitionConfiguration	x	x
Method CreateRecorderResultReader	x	x
Method CreateRecorderResultWriter	x	x
Method AddNewRecorder	x	x
Method Start	x	
Method Stop		x
Method ClearConfiguration	x	
Method RemoveRecorders	x	
Method StartAllRecorders	x	x
Method StopAllRecorders	x	x
Property setConfigurations	x	
Property setResynchronization	x	
Property getConfigurations	x	x
Property getRecorders	x	x
Property getState	x	x
Property getAcquisitionRecorder	x	x
Property getAcquisitionStartTime	x	x
Property getResynchronization	x	x

4.4.2.4 USING THE ACQUISITIONCONFIGURATION INTERFACE

A valid acquisition configuration consists of a list with at least one instance of the AcquisitionConfiguration class. The instances have to be created by the Acquisition instance to which the acquisition configuration is applied. Each AcquisitionConfiguration contains groups of measurement variables, which share a common set of capturing parameters, e.g. triggering conditions. The configuration contains the following attributes:

Table 18 AcquisitionConfiguration

Acquisition Attribute	Description
Name	<p>Name of the configuration.</p> <p>Note: A unique name must be given to each configuration that is created by one Acquisition instance. This is used to uniquely identify signals from a RecorderResult.</p> <p>Note: Each configuration that is part of the list of configurations that is set as the configuration of the Acquisition must have a name.</p>
TaskName	Raster identifier
Downsampling	Factor for downsampling
StartWatcher	Trigger for starting of acquisition
StartDelay	Delay in seconds for the start watcher
StopWatcher	Trigger for stopping of acquisition
StopDelay	Delay in seconds for the stop watcher
Retriggering	<p>Retriggering means that after receiving a stop trigger the Acquisition is again waiting for a start trigger. The Retriggering attribute is an integer that is interpreted as follows:</p> <p>Retriggering = 0, no retriggering. This means that the given start trigger condition is watched once.</p> <p>Retriggering = N, where N is the (positive) number of consecutive start/stop trigger sequences. (Example: N = 1 means that the start trigger condition is watched two times, first time as normal trigger, second time as a first retrigger.)</p> <p>Retriggering = -1, defines an infinite number of start/stop trigger sequences.</p>
EnforceNonRealtimeTriggerException	<p>Realtime triggering is possible only for trigger conditions that involve signals from a single testbench port. Those conditions can be evaluated locally.</p> <p>If trigger conditions involve signals from different testbench ports, these conditions will be evaluated in the Framework. As a consequence, no realtime behavior can be guaranteed.</p> <p>If the EnforceNonRealtimeTriggerException attribute is set to true, an exception is thrown if triggering cannot be carried out in realtime (e. g. if the trigger condition contains trigger variables from different testbench ports).</p>
Variables	This attribute specifies a set of variables that are to be measured according to this

Acquisition Attribute	Description
	AcquisitionConfiguration. This set contains framework variables that are mapped to different testbench ports. The XIL API offers an abstraction from testbench ports for testcases and configuration.

Note: The Framework will distribute its configuration settings to the related testbench functionality (e. g. testbench ports or testbench common interfaces). The underlying hardware (testbench ports) has to provide the functionality. If the setup fails, an error exception is thrown (post condition).

The test developer needs to be aware of the following situations:

- Any AcquisitionConfiguration can contain framework variables of different testbench ports. The framework internally splits the configuration into captures for the different testbench ports, which perform the capture (incl. data logging of the variable, evaluating trigger, task, downsampling etc.) If this is not possible, but required by the user's configuration, a testbench exception is thrown.
- Each framework variable must be used only once within the entire acquisition in order to allow for the object oriented access to the measured data via the framework variable's Signal method. The collected data within the framework can now be used for recordings or for evaluating the recorder's trigger conditions, which are performed in the framework.
- While the acquisition is running, the user can create, configure and start new recorder objects.

4.4.2.5 PRINCIPLE OF TRIGGERED ACQUISITION

Performing triggered acquisition means that data collection according to an acquisition group can be started and stopped via triggers. An acquisition group is part of the entire acquisition, that is configured by exactly one acquisition configuration. Data collection is not started or stopped until certain trigger conditions are met. The test developer can specify a start and a stop trigger to define the condition for each acquisition group.

If triggers are enabled for an acquisition group, the trigger conditions are evaluated on the capture system when measuring is started. A capture system in this sense is a HIL hardware or Offline Simulation, that is able to provide capture data via a capture interface on a testbench port. In XIL 2.0, the following ports provide capture interfaces: MAPort, ECUMPort, NetworkPort.

If trigger conditions contain variables of *only one* port, these trigger conditions are evaluated on the corresponding capture system. Data is transferred to the framework only if the trigger conditions are met. This means that the data flow between the capture systems and the framework is not continuous. This reduces the measurement data throughput.

Note: If trigger conditions contain variables of *different* ports, the trigger conditions are evaluated within the framework. Data is transferred to the framework without regards to the trigger conditions. This means that the data flow between the capture systems and the framework is continuous. Compared to the former case, this may increase the measurement data throughput between testbench and framework.

To describe trigger conditions, so-called Watcher classes are used. Since the Watcher classes in the Framework are very similar to Watcher classes used in the Testbench domain,

we describe here only the differences between the two domains. The general description of the Watcher classes is given in the Testbench chapter in section [Watcher](#).

The Framework Watcher interfaces are located in the namespace ASAM.XIL.Interfaces.Framework.Common.Watcher. To distinguish them from the interfaces in the corresponding Testbench namespace, they carry the prefix 'FW' to indicate that they are Framework-specific. They are shown in [Figure 57](#).

Furthermore, only one difference to the Testbench Watcher classes must be mentioned: While the ConditionWatcher in the Testbench domain uses a mapping of abstract names to simulation variable paths ('Defines'), this mapping is not required for the FWConditionWatcher. In the FWConditionWatcher, the abstract names occurring in the condition expression are interpreted as abstract variable identifiers from the Framework mapping (see section [Mapping](#)). Thus, the FWConditionWatcher does not have a 'Defines' property.

The condition of a condition watcher can only contain scalar framework variables that are one to one mapped to a testbench variable.

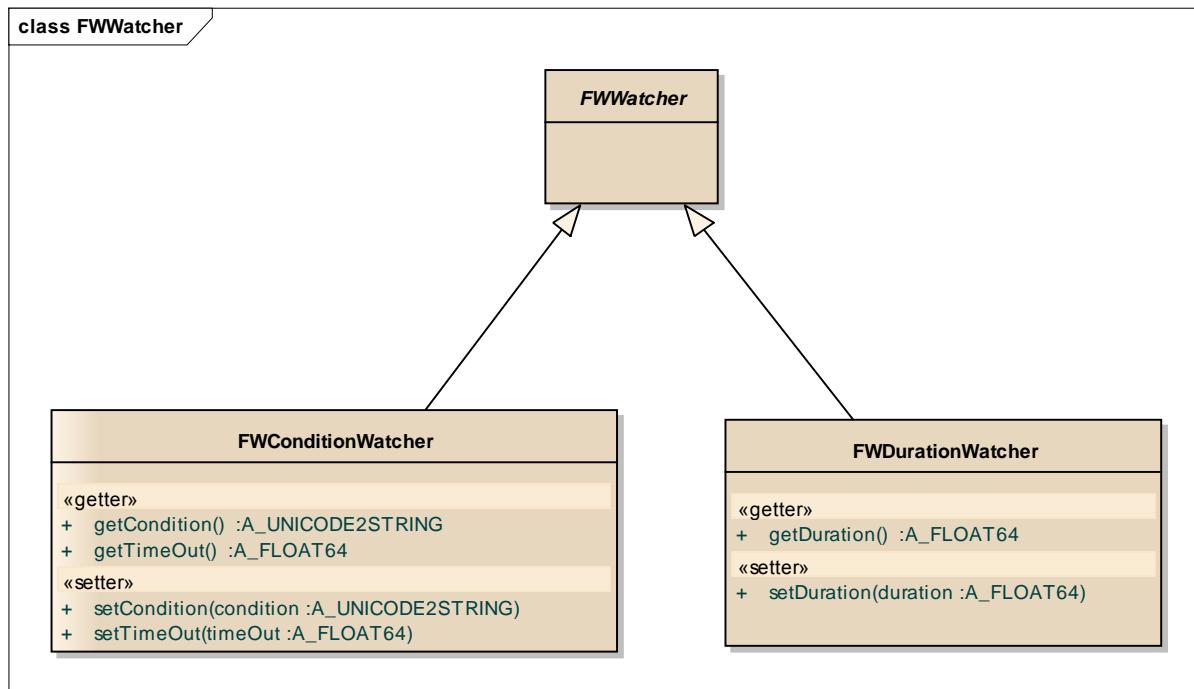


Figure 57: Framework Watcher

However, one additional aspect concerning the FWConditionWatcher must be noted: Since scalar variables in the Framework are generally considered as Quantities, i.e. they have a value and a physical unit, special considerations have to be made about the condition expression. As in HIL API 1.0, the condition expressions are based on the ASAM General Expression Syntax (GES) (see [Syntax of Watcher Conditions](#)). However, because in XIL API all variables have an associated unit, this information also has to be represented by GES expressions. The GES expressions themselves do not allow for the specification of units of identifiers. For this purpose, a set of rules is defined as to how GES expressions are to be interpreted with regard to units.

The following rules are to be applied:

- The identifiers occurring in a GES expression are to be interpreted as abstract variable identifiers which must be defined in the mapping table.
- When evaluating an expression, for each occurring variable identifier an assumption has to be made about the corresponding unit. For all conditions used in the Framework, the default Framework unit as defined in the mapping table should be used.

The user has to make sure that operators in an expression are applied only to variables with compatible units (e.g., no comparison between speed and distance variables are allowed). Errors of this kind will not be detected by the API.

The illustration below shows an example for triggered measurements:

- The data flow between testbench and framework begins with the explicit interface call `Acquisition.Start()`. Note, that at this point the time is equal to 0.0 s. Likewise, the data flow ends with the call of `Acquisition.Stop()` at an elapsed time relative to `Acquisition.Start()`.
- The Framework Variables `Reference_Position` and `Real_Position_1` belong to one acquisition group, which has been configured by one corresponding `AcquisitionConfiguration` without using any trigger conditions for start and stop. Thereby, the names `Reference_Position` and `Real_Position_1` are the abstract identifier attributes of these Framework Variables. Thus, data flow starts with `Acquisition.Start()` and ends with `Acquisition.Stop()`.
- The signals of `Real_Position_2` and `Real_Position_3` have transient behavior that a user wants to analyze. Only an interesting part of these signals need to be measured for this, in order to reduce the data transfer between testbench and framework. Details to this are given within the next step.
- The Framework Variables `Real_Position_2` and `Real_Position_3` belong to two different acquisition groups, which have been configured by two different `AcquisitionConfiguration` objects respectively with different trigger settings. First, let's have a look at the `AcquisitionConfiguration` containing `Real_Position_2`: The start trigger condition is “`posedge (Reference_Position, 45)`” which means, that it triggers at a positive (rising) edge of 45. The condition term contains the abstract identifier of the Framework Variable and refers the General Expression Syntax (GES). The stop condition is defined by a duration watcher (`Duration = 0,2`). Note, that the duration watcher starts with the trigger event. Thus, the total length of the measured interval is (`Duration - Delay`), in this example $0,2\text{ s} - (-0,05\text{ s}) = 0,25\text{ s}$. `AcquisitionConfiguration` containing `Real_Position_3` has the start trigger condition “`negedge (Reference_Position, 45)`”, triggering at the falling edge.

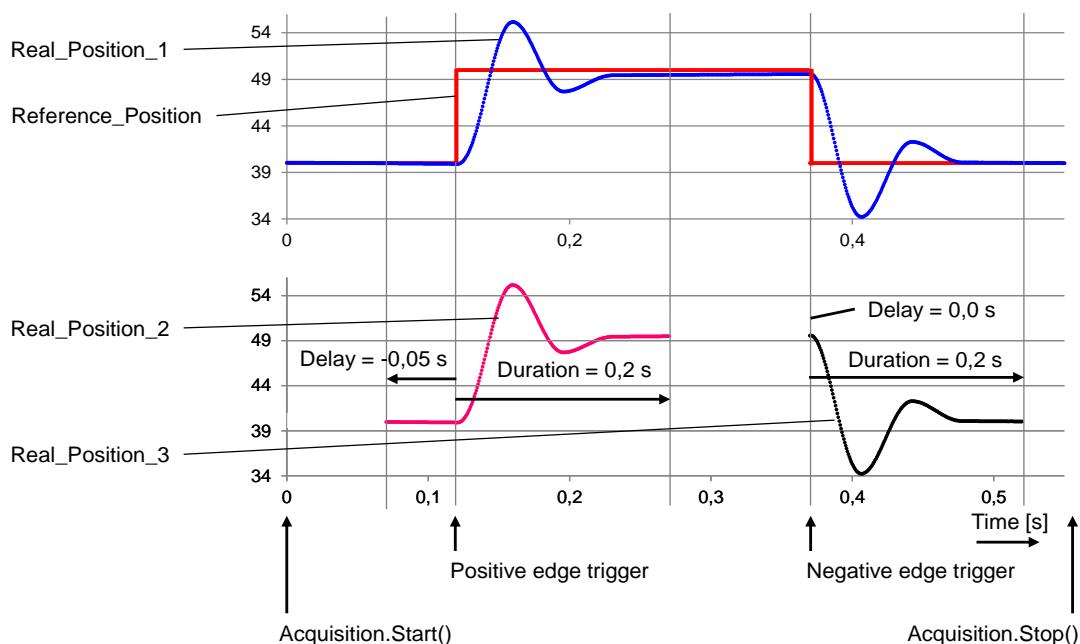


Figure 58: Example of different AcquisitionConfiguration Objects

Note: If a user collects large amounts of data in a capture system by an untriggered acquisition using high-frequency tasks, the operability of the framework might be reduced. Thus, users should be aware of controlling and reducing data transfer between testbench ports and framework by triggering and downsampling.

The next figure shows an acquisition with a longer time interval. The `AcquisitionConfiguration` objects responsible for data measuring of `Real_Position_4` and `Real_Position_5` have a downsampling higher than 1, to reduce the samples per time, and thus the data flow between testbench and framework.

Note, that the `AcquisitionConfiguration` objects responsible for measuring of `Real_Position_2` and `Real_Position_3` are configured with the `Retriggering` attribute set.

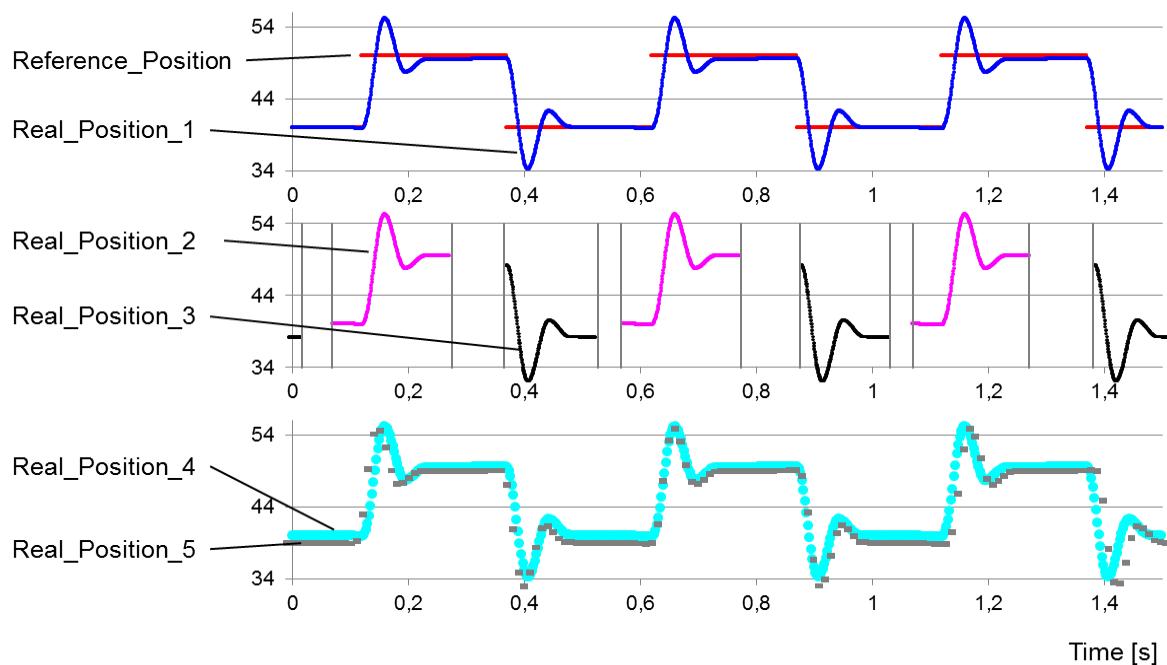


Figure 59: Acquisition with longer interval

The next figure shows another plot indicating data flow respective no data flow between testbench and framework. A thick line indicates data flow, which means data is collected with respect to the trigger condition. Thin lines indicate that there is no data flow.

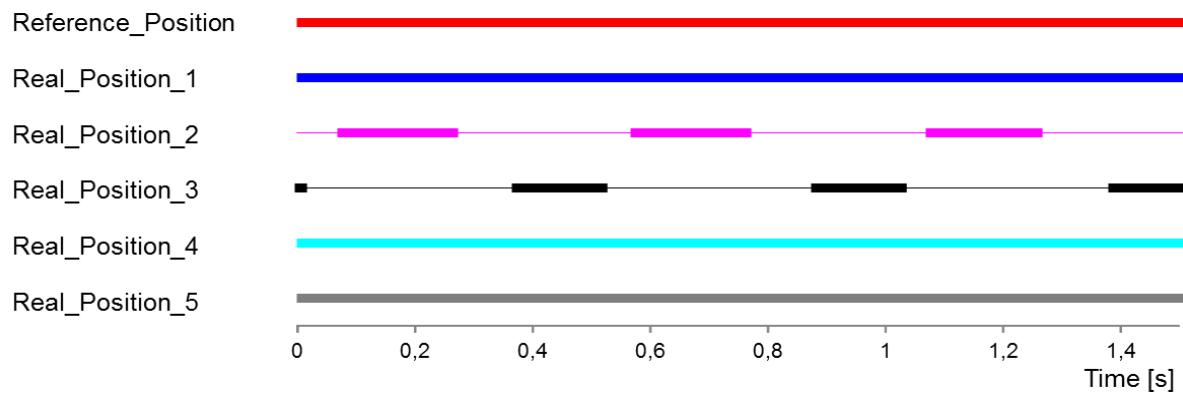


Figure 60: Acquisition data flow

4.4.2.6 SYNCHRONIZED DATA ACQUISITION

Measured signals from different hardware with different hardware timers usually are not synchronous. The XIL Framework can synchronize these signals to a common time base and therefore, it provides standardized interfaces to configure and enable the synchronization. The Framework supports synchronized data acquisition among signals across different capture systems accessed by testbench ports, which support capturing (MAPort, ECUMPort, NetworkPort).

Hardware timers usually are simple counters which start counting, once the hardware is switched on. During data acquisition, each measurement value of a capture system is given a time stamp by the corresponding timer. During data acquisition, those timers have different

initial values (due to different times, when they have been switched on) and they also may drift apart due to slightly different frequencies.

To make it more clearly, let's compare the hardware timers with clocks at different places in the world, which provide time stamps for measured values (see figure below): Montreal and Sydney provide measurement data in their own time base. Of course the time difference to these places will vary according to where in the world you are, but let's fix Munich as the place, from where this scenario is observed. Thus, Munich is supposed to be the framework time as the common time base to make measured signals from different locations comparable. At acquisition start, Munich is 6 hours ahead of Montreal, which means there is an offset of 6 ours; i. e. 4 am in Montreal is 10 am in Munich. However, the frequencies of timers differ, resulting in a mere 4 hours of difference between both cities at acquisition end in this example.

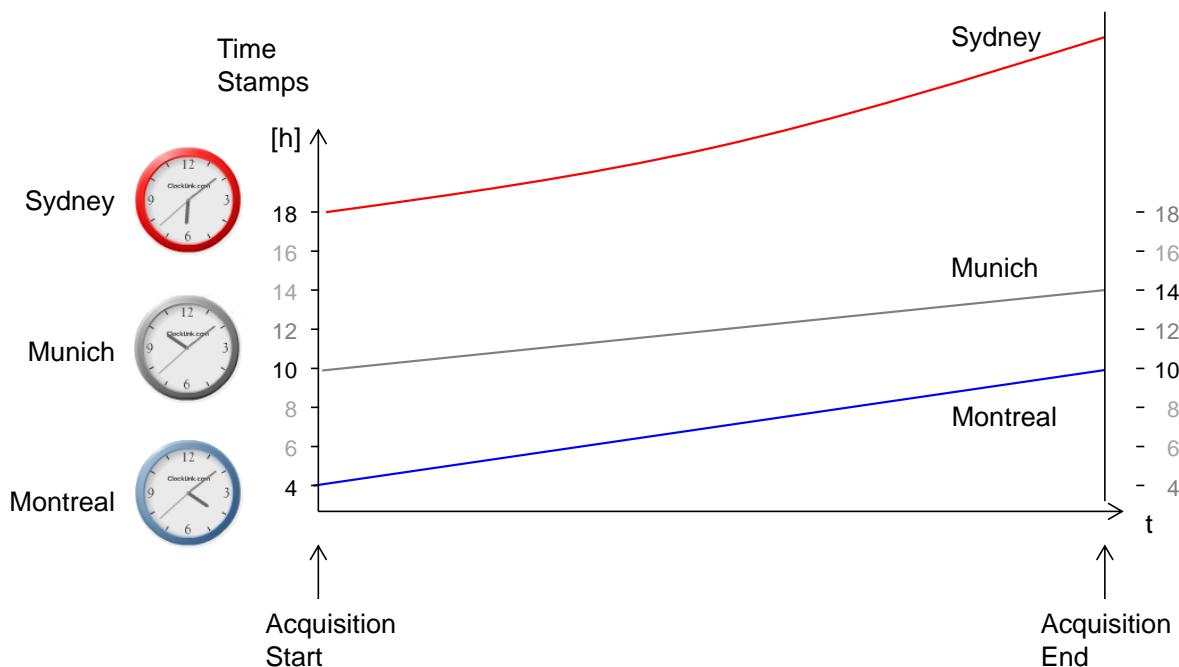


Figure 61: Effects of offsets and drifting timers (without resynchronization)

But it gets even more complex: Not all timers run at a constant frequency. The reason for this: changing environmental factors over time may affect the timer frequency, e. g. a rising temperature results in a higher timer frequency (see the significant curve progression of the Sydney time). Consequently, the framework compensates all these effects, if resynchronization is switched on (see figure below) using correction factors for offset and drift. Thus, the framework is turning the values of time stamps for Sydney back to Munich time, for Montreal it is advancing them. If signals are measured with equidistant time stamps at a local time, this transformation may lead to non-equidistant time stamps within the common time base, where the data is synchronized to. At acquisition start, all time stamps are reset to 0.

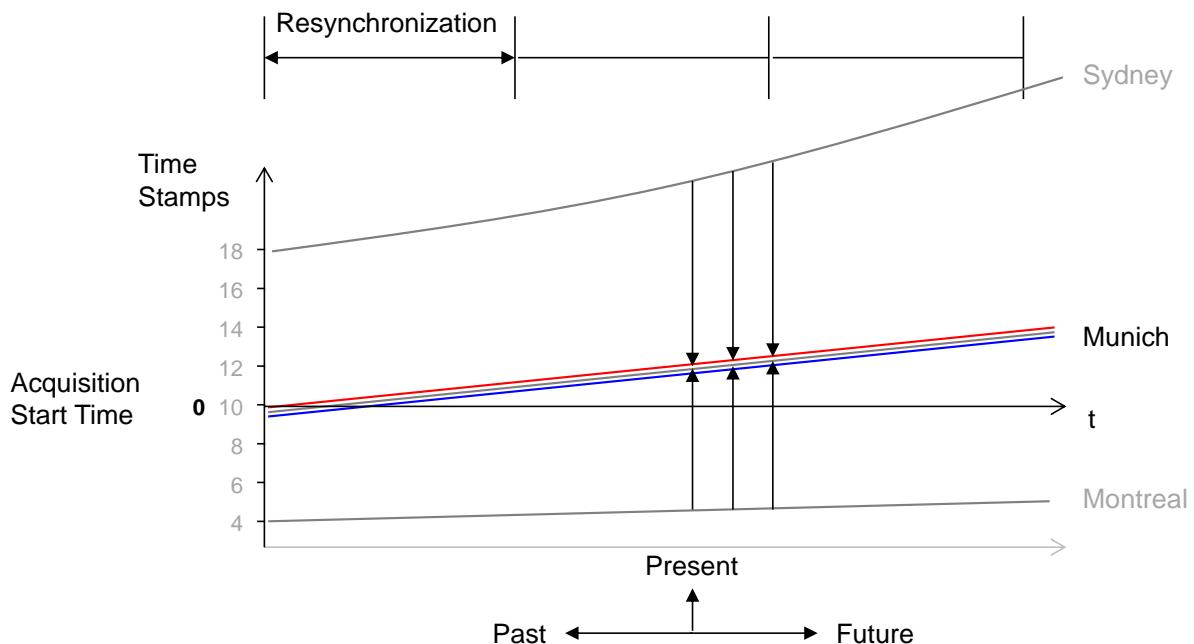


Figure 62: Effects of offsets and drifting timers corrected with resynchronization

The Resynchronisation property specifies the time interval in [ms], after that the framework recalculates the correction factors periodically in order to also deal with non-constant timer frequencies. A negative value of Resynchronization means, that the framework's resynchronization mechanism is switched off.

Note, that the acquisition and the recording result objects have a past, present and future. This means that the framework adds recently synchronized measured values at the present arrow to the result object, whereas measured data, which already has been added remains unchanged, since it is history. Thus, no post processing is done of data, the framework has already added to in the past. Remember: The past is not controllable, but observable – the future is not observable, but controllable. This means that the framework uses (observes) data that has been collected in the past, in order to compute new correction factors, which will be used for timestamp adjustment (control) in the near future of the next resynchronization interval.

Each testbench port, which supports capturing, provides the method GET_DAQ_CLOCK. This method returns the time of its port in seconds. This time base is also used for the acquisition signals, as it is also known from XCP. Internally, the Framework requests the current hardware time from the capture system using the `getDAQClock` method of the corresponding testbench port (e. g. MAPort). Note, that this method call has some latency time between starting the call and receiving the result. This may be one reason for some inaccuracy within the synchronized data acquisition.

The value of `AcquisitionStartTime` is the number of seconds since the Unix epoch time (i.e., the time unit is Unix time or POSIX time or Unix timestamp). This is the number of seconds that have elapsed since January 1, 1970 (midnight UTC/GMT), not counting leap seconds (in ISO 8601: 1970-01-01T00:00:00Z. Thus, the epoch is Unix time 0 (midnight 1/1/1970).

4.4.3 SETTING UP A RECORDING

4.4.3.1 INTRODUCTION

While the acquisition configuration defines the overall set of data, which is available in the framework, recording provides access to specific parts of the acquisition data. A recording makes acquisition data persistent, either in memory as a RecorderResult or in a file (e.g. MDF4). XIL API offers two different possibilities to the test developer, the AcquisitionRecorder and the Recorder.

The AcquisitionRecorder provides the user with a convenient way to define and access measured data. It is started and stopped implicitly with the call of acquisition start and stop, respectively. The user has access to a RecorderResult.

In addition, the user can obtain several Recorder instances via the Acquisition.AddNewRecorder() method. Each Recorder produces a RecorderResult. Thus, the user can set up independent Recorder objects during a running acquisition according to his particular needs. Recorder objects offer the possibility of an additional triggering in order to further focus on specific aspects within the acquired signals. As in the case of the AcquisitionRecorder, data access is possible via the ScalarVariable.Signal property and RecorderResult objects.

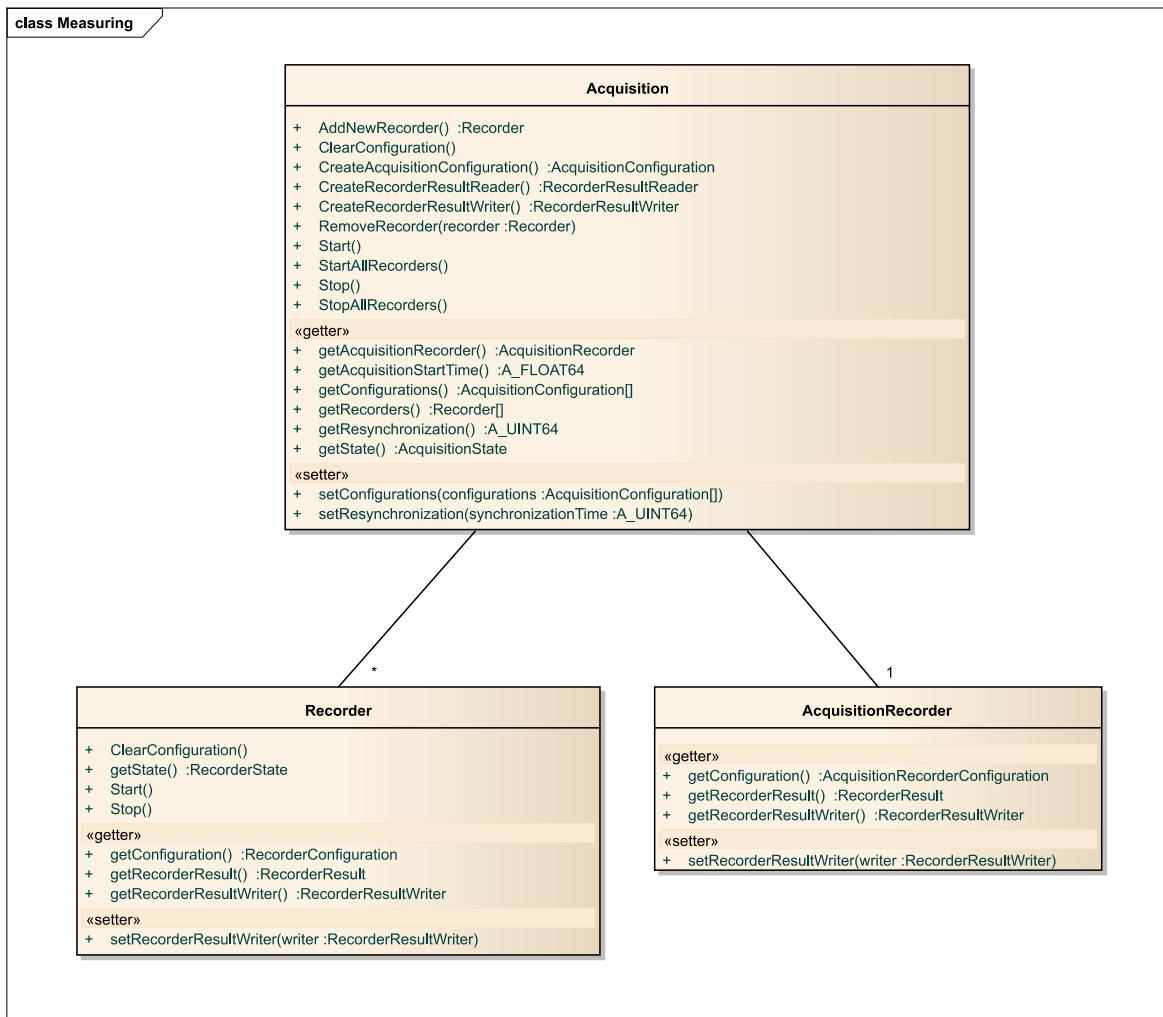


Figure 63: AcquisitionRecorder and Recorder

4.4.3.2 CONFIGURING AN ACQUISITIONRECORDER

The user can configure the AcquisitionRecorder (single instance per Acquisition) with the following configuration options of the AcquisitionRecorderConfiguration:

- Downsampling
- Framework Variables

The framework variables must be of type ScalarVariable. The measuring of complex data types such as VectorVariable, MatrixVariable, CurveVariable or MapVariable is not supported.

Note: The AcquisitionRecorder has no start and stop trigger conditions since it is started and stopped implicitly with the acquisition.

With the setRecorderResultWriter method a RecorderResultWriter can be assigned to the AcquisitionRecorder.

One framework variable instance can be recorded only once in the AcquisitionRecorder or one recorder. Therefore, it must be also part of the variable's list within the acquisition. This ensures that each framework variable has its unique timetrace, which can be obtained via the ScalarVariable.Signal property or the RecorderResult.

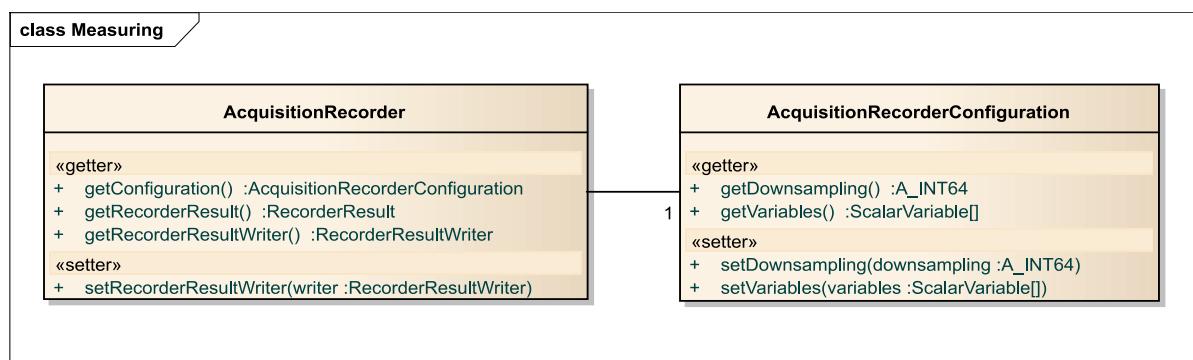


Figure 64: AcquisitionRecorder and its Configuration

4.4.3.3 CONFIGURING A RECORDER

The user can create any number of Recorder objects within the test case. In contrast to the AcquisitionRecorder, with Recorder objects the user can use start and stop triggers to control the start and end of the recordings. Thus, the list of Recorder settings is as follows:

- Downsampling
- StartWatcher (Conditions according GES)
- StartDelay
- StopWatcher (Conditions according GES)
- StopDelay
- Retriggering
- Variables

The Framework variables must be of type ScalarVariable. The measurement of complex data types such as VectorVariable, MatrixVariable, CurveVariable or MapVariable is not supported.

With the `setRecorderResultWriter` method a RecorderResultWriter can be assigned to the Recorder.

One framework variable instance can be recorded only once in the AcquisitionRecorder or one recorder. Therefore, it must be also part of the variable's list within the acquisition. This ensures that each framework variable has its unique timetrace, which can be obtained via the `ScalarVariable.Signal` property or the `RecorderResult`.

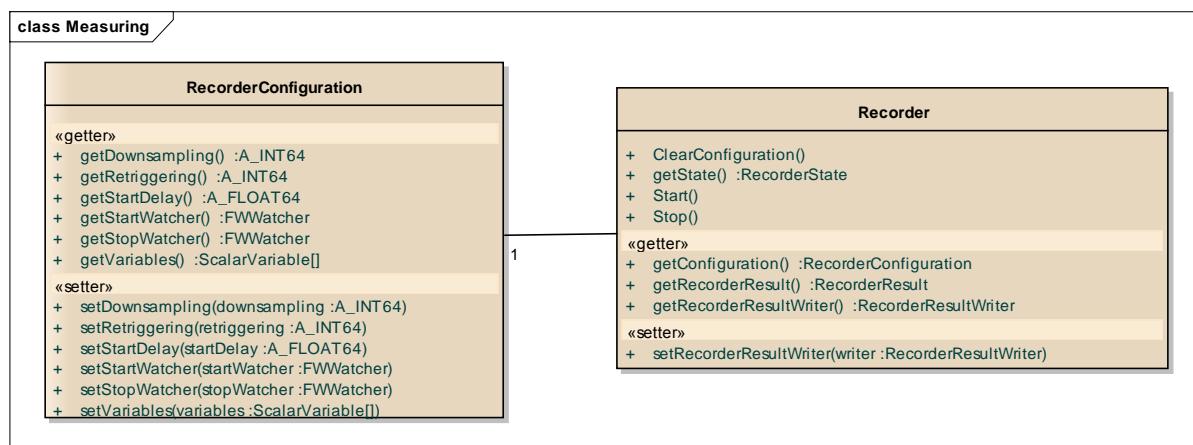


Figure 65: AcquisitionRecorder and RecorderConfiguration

4.4.3.4 USING RECORDERRESULT READERS AND WRITERS

This section is valid for Recorder and ResultRecorder objects as well. To make it easier and more understandable, in the following text of this section all descriptions refer to the Recorder, also saying, that it is valid for AcquisitionRecorder as well.

The recorder is able to stream RecorderResult objects either to memory or to a file using RecorderResultWriter objects. RecorderResult objects that have been streamed into a file, can be read back to memory by using a RecorderResultReader.

The following figure gives an overview of the reader and writer classes that XIL API provides for RecorderResult objects.

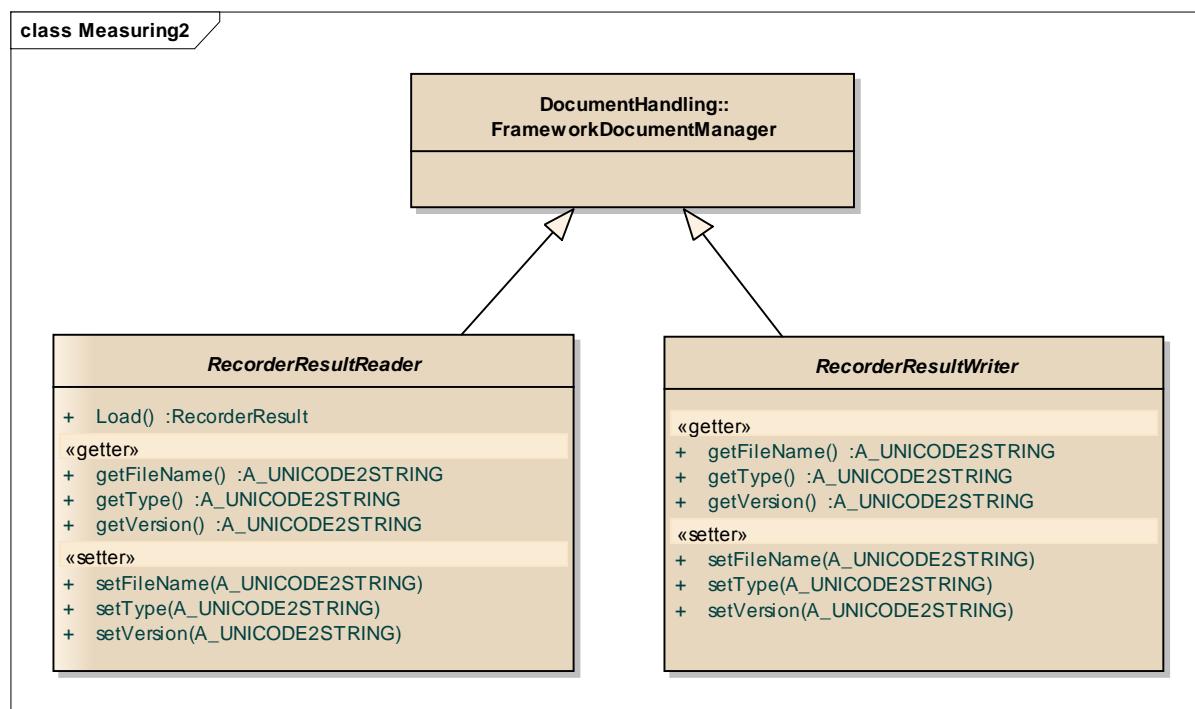


Figure 66: RecorderResultReader and RecorderResultWriter

Both, the RecorderResultWriter and the RecorderResultReader can be created using the corresponding factory methods CreateRecorderResultWriter and CreateRecorderResultReader of the Acquisiton object. The following table shows the attributes, which can be used to configure the readers and writers.

Table 19 Attributes of RecorderResultReader and RecorderResultWriter

Attribute	Description
FileName	Specifies the complete file name as string.
Type	Specifies the type (format) that is expected as string. Example: type: "MDF" XIL API 2.0 currently supports "MDF" and "MEMORY". This allows, to define also proprietary types other than "MDF" and "MEMORY". Please refer to the specific vendor's documentation in order to get a complete list of types supported by the vendor. For details of MDF storage see annex D.1.
Version	Specifies the version of the configured type as string. Example: type: version "4.1" Syntax: major.minor[.maintenance] major, minor, maintenance are represented by numbers (0 - 99) default is an empty string. If the type is "MEMORY", the version information is ignored.

The writer objects are assigned to a Recorder's RecorderResultWriter attribute. Data streaming (either to memory or into file) is done implicitly by the recorder while it is running.

By default, the Recorder has a RecorderResultWriter of type "MEMORY" assigned. This means, that recorder results are written to memory, but will not become persistent automatically. See also section [Save data to file](#).

The writer of the Recorder can only be changed, if the Recorder is stopped. Change in this sense means, either a new writer object is assigned to the Recorder's Writer attribute or the attributes of an assigned Writer are changed. The data produced by the Recorder (either in ScalarVariable.Signal or within the RecorderResult) still exists even after changing the Writer until the Acquisition is being restarted.

Start of Recording

After calling the Start method, the Recorder is ready for recording. If a start trigger condition has been set, recording is started after the match of a valid trigger condition. If no start trigger is set, the recording starts immediately after calling the start method. A recorder can also be started before starting the Acquisition.

Stop of Recording

The recording stops either if the last stop trigger condition is matched (if a stop trigger condition exists) or after the Stop method is called. Also, Acquisition.Stop() implicitly stops all recorders.

Get Results

If the user calls the method GetRecorderResult() during a running recording, the recorder result contains all measured data up to this point. In contrast to the Fetch() method known from the Capture class in HIL API 1.0.2, GetRecorderResult() returns all data from start of the recording. The framework internally calls the Fetch() method of the underlying testbench ports, which are defined by the mappings of the recorded framework variables.

Important Notes:

- In the case of subsequent calls of GetRecorderResult() the latest RecorderResult will contain also the complete data of previous calls, since the recording was started.
- If a Recorder is stopped and subsequently restarted within a running acquisition, RecorderResults will be appended, similar to the behavior of StartWatcher and Retriggering. This applies also to stored results by an attached writer.

Note: Stopping the Acquisition does not mean that recorded data is lost. Test cases can still refer to previously recorded results. The user can explicitly remove storage results by calling the method ScalarVariable.ClearSignal().

A new acquisition has a time base, beginning with time 0.0 seconds at acquisition start. The acquisition can be reconfigured, with respect of rasters, downsamplings etc. Thus, a new start of the acquisition results implicitly in discarding all formerly recorded data.

4.4.3.5 WORKING WITH THE RECORDED RESULTS

The user can access the recorded signal of a ScalarVariable in an object oriented manner via its Signal property.

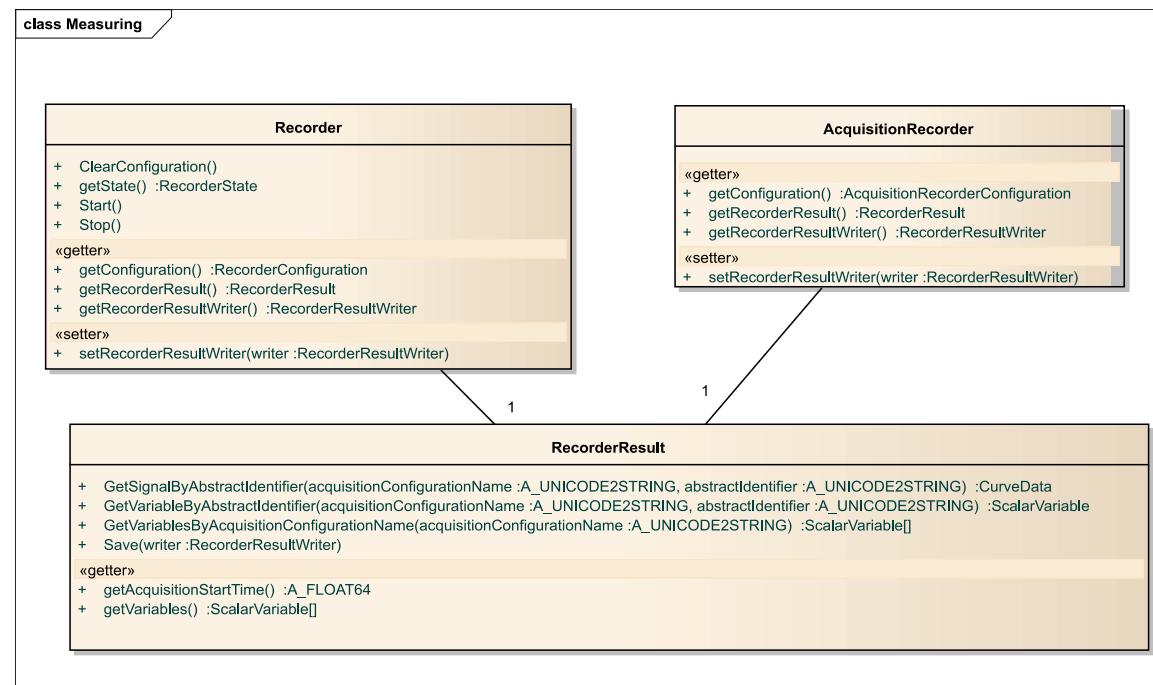


Figure 67: AcquisitionRecorder and Access of the Measured Data

If the user accesses the `Signal` property during a running acquisition, he gets the measured data up to this point. The framework invokes the corresponding `Fetch` method on the respective testbench ports internally. The second option to work with results recorded via a `SignalRecorder` is obtaining and using a `RecorderResult` object.

Retrieve a single signal

The measured data that has been assigned to a specific `ScalarVariable` object (by adding the variable object to an `AcquisitionConfiguration` and a `RecorderConfiguration` or `AcquisitionRecorderConfiguration`) can be accessed via the `ScalarVariable`'s `Signal` property. The type of the returned value is a sub-type of `CurveData` (e.g., `FloatFloatCurveData`) that depends on the type of the measured variable. Correspondingly, the measured values and the time axis, which can be accessed via `CurveData`'s `FcnValues` and `XValues` property, respectively, are of a type which also depends on the measured variable. This base type of these properties is `Quantity`[].

A second way of accessing measured data is the use of a `RecorderResult` object. This can be obtained by the `RecorderResult` property of a `Recorder` or `AcquisitionRecorder`. A `RecorderResult` provides several methods to access one or more measured signals. Individual signal data can be accessed via the method `GetSignalByAbstractIdentifier`. This method expects two parameters, the abstract identifier of the measured variable and the name of the `AcquisitionConfiguration` in which this variable was configured. The return value is a `CurveData` object equivalent to the object obtained by `ScalarVariable`'s `Signal` property. A similar method to access individual signal data is `GetVariableByAbstractIdentifier`. This method has the same parameters, but it returns a `ScalarVariable` instead of a `CurveData` object. The returned `ScalarVariable` represents the measured variable, but is not connected to any `Port` instance. The user can access the measured signal via its `Signal` property. To obtain

all measured signals contained in a specific `AquisitionConfiguration`, the method `GetVariablesByAcquisitionConfigurationName` is available. Finally, to get all measured signals contained in a `RecorderResult`, the property `Variables` of the `RecorderResult` can be employed.

Load data from file

The `Acquisition` object provides the factory method `CreateRecorderResultReader` in order to create a new `RecorderResultReader` instance. The `RecorderResultReader` has an attribute `FileName` and a method `Load`, which returns the loaded `RecorderResult` from file.

As corresponding variables will be created during `RecorderResultReader.Load`, the user can obtain instances of `FrameworkVariables` from the formerly imported `RecorderResult`. Furthermore, the user can retrieve the measured data from the `ScalarVariable.Signal` property.

MDF 4/4.1 provides the information, which is needed to generate framework variables from a measurement file.

For details see the following section [Save data to file](#).

Note: Variables, which have been loaded from a data file cannot be used in port read or write actions directly, because they do not have a port mapping.

Save data to file

If the Recorder has a writer, that writes to file, the Recorder itself streams data to file automatically during recording.

An existing `RecorderResult` can be saved to file using the `Save` method explicitly. This can be necessary, if a `RecorderResultWriter` has been used, that writes to memory or if a `RecorderResult` was loaded from file using a reader and has been changed using XIL API methods afterwards.

4.4.3.6 RECORDER STATES

The Recorder interface has two stats: `eSTOPPED` and `eSTARTED`.

The following table shows the method calls, which are allowed in each state indicated by an "x" sign. If calls are not allowed, an exception is thrown. Most of the calls are allowed in all states.

Table 20: Recorder States

Recorder Operation	eSTOPPED	eSTARTED
Method Start	x	
Method Stop		x
Method ClearConfiguration	x	
Property getRecorderResult	x	x
Property getConfiguration	x	x
Property getRecorderResultWriter	x	x
Property getState	x	x
Property setRecorderResultWriter	x	

4.4.4 SEQUENCE OF ACQUISITION AND RECORDER COMMANDS

The figure below shows an example, how the Acquisition together with the AcquisitionRecorder can be used. Please compare with [Figure 60](#), which gives the explanation of the Acquisition configuration.

- The gray rectangle comprises the acquisition, together with its start and stop command, step (5) and (18).
- Step (1) configures the Acquisition.
- Step (2) configures the AcquisitionRecorder.
- Step (3) and (4) create a further Recorder *before* Acquisition.Start().
- After Acquisition.Start(), Recorder_1 is started (6).

Note: this could be done before Acquisition.Start. In this case, the recorder would start with the acquisition synchronously with Acquisition.Start(), like the AcquisitionRecorder does.

Please note that the amount of data for the “pink” and “black” variables is already limited by the Acquisition configuration. In these examples Recorder 1 records the signals without triggering. The gaps between the thick lines indicate that the Acquisition does not contain data for these time spans.

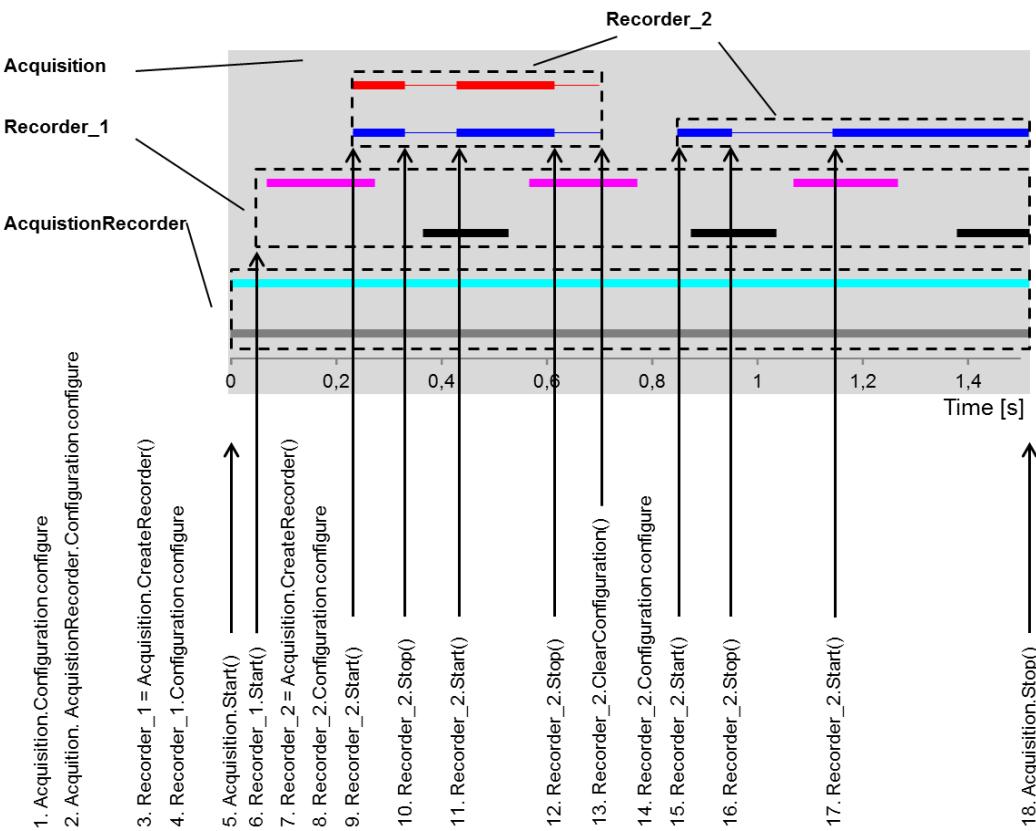


Figure 68: Sequence of Acquisition and Recorder Commands

- Step (7) and (8) create Recorder_2 *while* Acquisition is started.
- Then, a sequence of start and stop commands is called on Recorder_2 in steps (9), (10), (11) and (12). The start and stop commands distinguish, which part of the incoming data is taken into account for the RecorderResult (thick vs. thin lines in the rectangle of Recorder 2).
Please note that although the Acquisition is configured to continuously get the “red” and “blue” variables, outside of the time span of running Recorder 2 no data is recorded.
- Step (13) resets the Recorder 2, which means, it cannot be restarted, without reconfiguring before. Also, the RecorderResult is cleared.
Note: The `ScalarVariable.Signal` still contains the data of the previous recording, until either calling `ScalarVariable.ClearSignal()` or a new `Acquisition.Start()`. Signal data of an existing variable is always appended to already existing signal values.
- Step (14) reconfigures Recorder_2. Note, that also the recorded FrameworkVariable changed (the “red” variable is not included anymore).
- Another sequence of start and stop commands is following on Recorder_2. Note: in this case, Recorder_2 is not stopped explicitly by a command. It stops implicitly with `Acquisition.Stop()`.

4.5 STIMULATION

4.5.1 MOTIVATION

When testing ECUs via XIL API, signals play an important role in different use cases. In many test cases, simulation model variables are stimulated, measured and the measured data has to be compared with reference signals. For these use cases, the XIL API introduces the Stimulation package of the Framework.

This chapter shows the stimulation capabilities of XIL API. This comprises the configuration of sophisticated sets of different signals represented by classes already known from HIL API 1.0 (`SignalDescriptionSet`, see sections [Introduction](#) and [Signal Descriptions](#)). The `Player` interface uses the `FWSignalDescriptionSet` interface to perform the stimulation of testbench variables.

The stimulation facilities provided by the Framework API are based on two interfaces: the `Stimulation` interface and the `Player` interface.

- The `Player` interface aggregates a number of signal descriptions and defines what variables are to be stimulated by these signals. This is described by a collection called `FWSignalDescriptionByScalarVariableCollection`, which associates simulation variables to signal descriptions. As in the other parts of the Framework API, simulation variables are accessed via instances of the `FrameworkVariable` interface, which abstracts a variable from its specific `Testbench` port. However, these variables must be of type `ScalarVariable`. The stimulation of entire complex data types such as `VectorVariable`, `MatrixVariable`, `CurveVariable` or `MapVariable` is not supported explicitly. Nevertheless, single elements of these complex data types can be mapped to a single `ScalarVariable`, in order to stimulate single elements of complex data types.
- The `Stimulation` interface aggregates and manages different players. Therefore, it provides factory and remove methods for players, as well as global start and stop methods. Within the stimulation process, the `Stimulation` interface plays a role similar to the `Acquisition` interface within the measuring process: while the `Acquisition` interface controls the data flow from the testbench ports to the framework via `Recorders`, the `Stimulation` interface controls the ‘reverse direction’, allowing to globally start and stop the stimulation process and aggregate signals via the `Player` interface. Various players can be set up, started, paused and stopped independently from each other. Furthermore, the `Stimulation` interface provides factory methods for the `PlayerReader` and `PlayerWriter` classes, which are required to load and save individual Players from or to a file.

4.5.2 SETTING UP A STIMULATION

4.5.2.1 INTRODUCTION

The stimulation interface provides methods for adding new players and creating `PlayerReaders` and `PlayerWriters`. By means of `Stimulation.Start` and `Stimulation.Stop` methods, a commonly time basis for different players can be utilized.

Note: The stimulation interface provides no options, to configure time synchronization, as known from data acquisition in the measuring section.

4.5.2.2 USING THE STIMULATION INTERFACE

The test developer obtains the Stimulation object via the Framework's attribute `Stimulation`.

Note: The Stimulation is a single instance per Framework. When first accessing `Stimulation`, the Framework creates the instance.

After obtaining the Stimulation instance, instances of the `Player` interface can be created via `AddNewPlayer()`. This method adds a Player to the Stimulation's internal list of Players and returns it. The complete list of Players can be accessed via the `Players` attribute. To remove a specific Player from the list, the `RemovePlayer()` method is used.

When all Players have been configured, the Stimulation's `Start()` method must be called to enable the stimulation process. Then, individual Players can be started or stopped as desired (see section [Using a Player](#) for details). To start or stop all Players at once, the `StartAllPlayers()` and `StopAllPlayers()` methods are available.

To load signal information into a player or to store it, several reader and writer classes are available. As in HIL API 1.0.2, these classes use the STI and STZ file formats. The corresponding factory methods are provided by the Stimulation instance:

`CreatePlayerReaderSTI()`, `CreatePlayerWriterSTI()`,
`CreatePlayerReaderSTZ()`, and `CreatePlayerWriterSTZ()`.

The attribute `StimulationStartTime` is the time of the last call to the `Stimulation.Start()` method. The initial value is -1.0 indicating, that `Stimulation.Start()` has not been called yet. The value of `StimulationStartTime` is the number of seconds since the Unix epoch time (i.e., the time unit is Unix time or POSIX time or Unix timestamp). This is the number of seconds that have elapsed since January 1, 1970 (midnight UTC/GMT), not counting leap seconds (in ISO 8601: 1970-01-01T00:00:00Z. Thus, the epoch is Unix time 0 (midnight 1/1/1970).

The current state of the global stimulation can be obtained via the `State` attribute. The available states are detailed in the following section.

4.5.2.3 STIMULATION STATES

The Stimulation instance has two states: `eSTOPPED` and `eSTARTED`.

The following table shows the method calls, which are allowed in each state. They are indicated by an "x" sign. If calls are not allowed, an exception is thrown. Most of the calls are allowed in all states.

Table 21: Stimulation States

Stimulation Operation	eSTOPPED	eSTARTED
Method CreatePlayerReader	x	x
Method CreatePlayerWriter	x	x
Method AddNewPlayer	x	x
Method Start	x	
Method Stop		x
Method ClearConfiguration	x	
Method RemovePlayer	x	
Method StartAllPlayers	x	x
Method StopAllPlayers	x	x
Property getPlayers	x	x
Property getState	x	x
Property getStimulationStartTime		x

4.5.3 SETTING UP A PLAYER

4.5.3.1 INTRODUCTION

A Player replays signals from a given `FWSignalDescriptionSet` to corresponding variables on the testbench. A `FWSignalDescriptionSet` comprises one or more signal descriptions. A signal description consists of one or multiple segments, e.g. a ramp, followed by sine, or simply a constant signal. Many other segment types are also defined by the XIL API. Such a signal description is independent of any variables in the simulation model. In order to apply a `FWSignalDescription` to a specific testbench variable, the player uses an assignment of a framework `ScalarVariable` to the signal description.

In order to not duplicate the discussion of signal creation and of the signal classes, the reader is referred to section Signal Descriptions of the Testbench chapter, where a general introduction and description of the signal classes can be found. In the following, we describe only aspects that are unique to the Framework's signal classes.

Generally, almost identical classes are used for signal description in the Framework and Testbench namespaces. In order to distinguish them, the Framework signal classes carry the prefix 'FW'. Thus, e.g., there is a class `SegmentSignalDescription` in the Testbench namespace and a corresponding class `FWSegmentSignalDescription` in the Framework.

The following table shows the namespaces which contain interfaces that correspond to each other (prefixed with 'ASAM.XIL.Interfaces'):

Table 22 Namespaces in ASAM.XIL.Interfaces

Namespace in Framework	Namespace in Testbench
Framework.Stimulation.Signal	Testbench.Common.Signal
Framework.Common.Symbol	Testbench.Common.Symbol
Framework.Common.Watcher	Testbench.Common.WatcherHandling

One unique aspect of the Framework in contrast to the Testbench lies in the representation of simulation variables by FrameworkVariable objects. Due to this difference, the class StringSymbol in the Testbench domain (describing a model variable's path as a string) is replaced by the class FWScalarVariableSymbol which uses a ScalarVariable object to represent this relationship.

Since the Framework does not make use of the ValueContainer classes, the SignalValueSegment, which depends on the SignalValue container, is missing from the Framework signal classes.

Also for this reason, the method 'CreateSignalValue' of the Testbench SignalDescription class is not duplicated in FWSignalDescription, but is replaced with the method 'CreateCurveData', returning a Framework-specific CurveData instance instead of a SignalValue instance. Also, the method 'CreateSignalGroupValue' is present only in the Testbench SignalDescriptionSet class.

4.5.3.2 CONFIGURING A PLAYER

Figure 69 shows the class diagram of the Player class and its relation to the FWSignalDescriptionSet class.

The FWSignalDescriptionByScalarVariableCollection class represents the assignments of simulation variables to signal descriptions.

First of all, it is necessary to configure the FWSignalDescriptionSet of the Player via its FWSignalDescriptionSet attribute. In the next step, the framework variables have to be assigned to the corresponding signal descriptions by setting up the FWSignalDescriptionByScalarVariableCollection obtained by the Player's Assignments attribute. The Assignments contain the relations between a framework variable and the corresponding signal description. Such assignments can be added using the Add() method and pass a framework ScalarVariable object together with a FWSignalDescription object as part of a given FWSignalDescriptionSet.

Note: The signal descriptions used in the Assignments, must be part of the player's FWSignalDescriptionSet. If this is not the case, an exception is thrown.

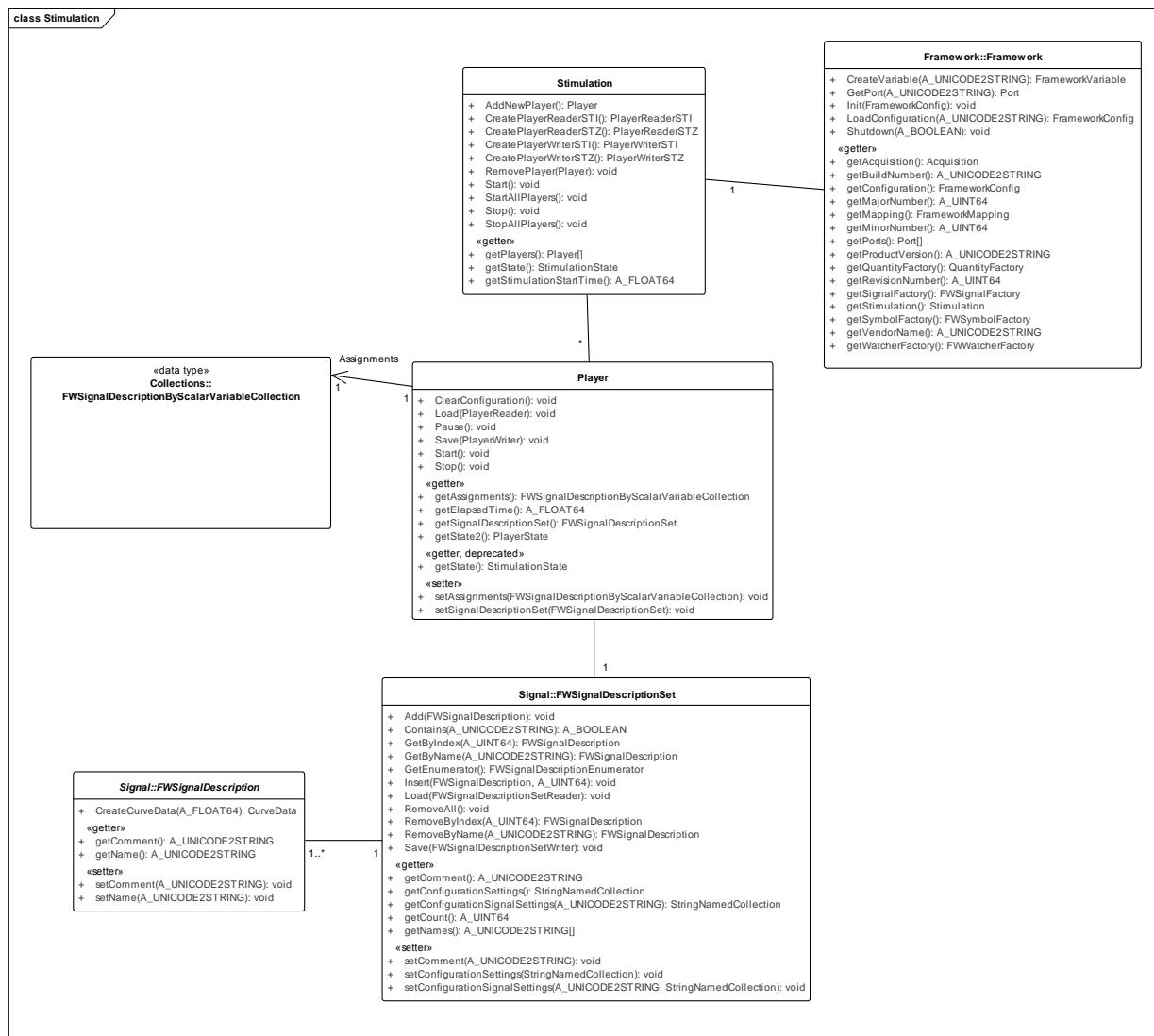


Figure 69: Player and relations to SignalDescriptionSet and Assignments

4.5.3.3 USING A PLAYER

Start of Signal Playing

After calling the `Start()` method, the player starts to playback signals to testbench variables. Therefore, the framework implicitly prepares and downloads all necessary information to the related testbench port. After a player is paused, the following call of the `start` method the player will continue from the latest time position.

The precondition to start a player is a started Stimulation. If a player is started before the stimulation has been started, the player starts its signal playing implicitly, when calling `Stimulation.Start()`.

Note: If a player has been started while the stimulation is in state `eSTOPPED`, the player's state changes to `eSTARTED`. The player's internal time remains at zero, until the stimulation is started.

Stop of Signal Playing

The method `Stop()` terminates signal playing and sets the time pointer to the beginning of the signal description. This method can be used to stop and reset the stimulus before reaching the end of its definition.

Pause of Signal Playing

Holds the signal playing at the current time step. The signal playing can be continued with the `Start()` method. In this state (`ePAUSED`), the stimulus does not write to the model. It is possible to call `Pause()` and `Start()` several times during signal playing.

Save of Player Information

`Save` stores the `FWSignalDescriptionSet` and Assignment information in either `.sti` or `.stz` format due to the given writer (see [Signal Description File](#)).

Load of Player Information

`Load` restores the `FWSignalDescriptionSet` and Assignment information in either `.sti` or `.stz` format due to the given reader (see [Signal Description File](#)).

Note: In order to load the player information correctly, the abstract identifiers of the given Framework Variables must exist in the framework's mapping, since the player needs access to the corresponding variables on the testbench. If the abstract identifier cannot be found in the mapping, an exception is thrown.

4.5.3.4 PLAYER STATES

The Player interface has three states: `eSTOPPED`, `eSTARTED` and `ePAUSED`.

The following table shows the method calls, which are allowed in each state indicated by a "x" sign. If calls are not allowed, an exception is thrown.

Table 23: Player states

Player Operation	eSTOPPED	ePAUSED	eSTARTED
Method Start	x	x	
Method Stop		x	x
Method Pause			x
Method ClearConfiguration	x		
Method Load	x		
Method Save	x		
Property <code>getSignalDescriptionSet</code>	x	x	x
Property <code>getAssignments</code>	x	x	x
Property <code>getState2</code>	x	x	x
Property <code>getElapsed Time</code>	x	x	x
Property <code>setSignalDescriptionSet</code>	x		
Property <code>setAssignments</code>	x		

5 Testbench

5.1 COMMON FUNCTIONALITIES

This chapter describes functionalities of the XIL Testbench that are not specific for one port. They are located in the package Common. The following table gives an overview of the sub packages in Common.

Table 24 Packages of Common part

Sub package name	Description
Capturing and CaptureResult	Interfaces for capturing and handling of captured data.
DocumentHandling	Base interface and overview of derived interfaces that are used to read or write content from/to the file system in different file formats.
Duration	Different representations for duration values.
Error	Testbench error codes and common classes for error handling.
MetaInfo	Interfaces for querying meta data of test bench parameters (e.g. data types and units of test bench variables).
Script	Abstract base interfaces for configuration and execution control of user provided scripts. Refer to package SignalGenerator and TargetScript for concrete types of user scripts.
Signal	Interfaces for (analytical) waveform descriptions as well as reading and writing these waveforms from and to STI / STZ files.
SignalGenerator	Signal player for continuous, time-accurate stimulation of test bench variables including interfaces for reading and writing SignalGenerator configurations from and to STI / STZ files.
Symbol	References to constant values, test bench variables or signal descriptions used by the waveform description interfaces in the Signal package.
TargetScript	Player for user scripts that are executed synchronously with the application on the target system.
ValueContainer	Set of classes designed to represent the values of test bench variables (e.g. scalar, matrix or map values). Together with the ASAMTypes and the Collection classes, the value container classes form the fundamental type system used throughout the UML model.
VariableRef	Interfaces representing references to test bench variables. These interfaces are used across all read, write, capture and stimulate methods.
WatcherHandling	Interfaces for trigger definitions used by capturing, signal generation, breakpoints etc.

5.1.1 VALUECONTAINER

5.1.1.1 OVERVIEW

The ValueContainer package provides a set of data classes used for (runtime) data exchange between the XIL client and the target system that is controlled via the XIL interfaces. In particular, these data classes are used to represent the values of the system variables of the target system. The data classes are divided into two categories:

- General container classes representing either scalar values or arrays with multiple elements that are accessed by integer indices, e.g. vectors and matrices. Concrete sub classes are available for the most important element types like boolean, integer, float and string. Some examples are ScalarFloatValue, StringVectorValue and BooleanMatrixValue.
- Container classes related to specific usage scenarios. Examples are the CurveValue and MapValue, that are used for calibration access, and the SignalValue and SignalGroupValue being used as numerical representation for signal descriptions.

Note: XIL also uses collections (named collections and indexed collections) of the data classes in the ValueContainer package. However, no collection classes are defined in the ValueContainer package. Instead, the collection classes built into the used programming language are used. The collection classes declared by XIL can be found in package XILTypes\Collections of the generic UML model. Refer to [7] for the mapping of these classes to language specific collection classes.

All container classes are derived from a common base class named `BaseValue`. The concrete type of a value container instance when referenced as `BaseValue` can be determined via its properties `ContainerType` and `ElementType`. `ContainerType` indicates whether the value is a scalar value or a composite value like matrix or curve value. It uses the enumeration `ContainerDataType` as type identifier. The property `ElementType` determines the primitive data type of the contained element(s) like `int`, `float`, `string`. It uses the `PrimitiveDataType` enumeration as type identifier.

The `Value` property of a value container instance returns a copy (not a reference) of the contained internal data objects. This also applies to any further, type specific data access methods and properties. For example, the `Xvector` property of a `MapValue` object returns a copy of its `VectorValue` instance. This is to make sure, the value of a value container instance cannot be implicitly changed by altering the retrieved data. However, copying is not required if the internal data object itself is immutable (e.g. if it is an `int` or `string` value).

The following chapters explain the general and the specific value container classes in more detail.

5.1.1.2 GENERAL VALUE CONTAINER CLASSES

General ValueContainer classes represent scalar, vector and matrix values (Figure 70).

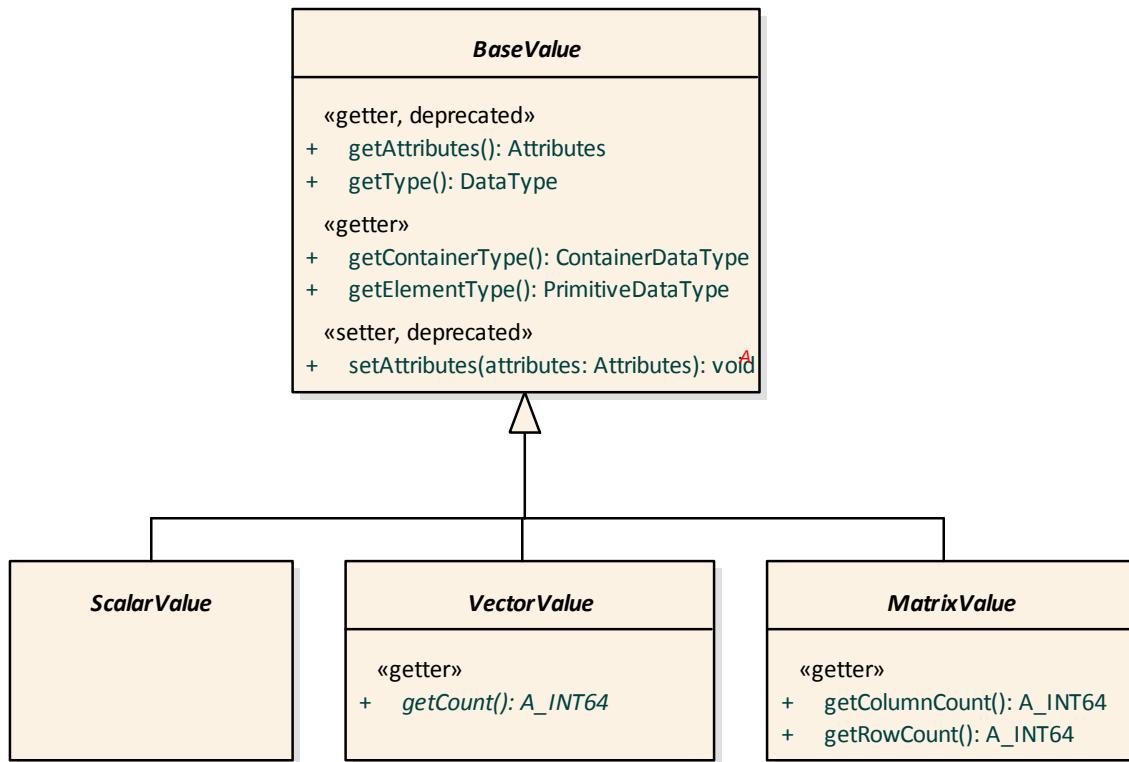


Figure 70: General Value classes

The `VectorValue` or `MatrixValue` classes are homogenous, i.e. all elements are of the same type. That is why there is a sub class of `VectorValue` and `MatrixValue` for every supported element type. Their class names are prefixed with `Int`, `Uint`, `Float`, `String` and `Boolean` corresponding to their element type. [Figure 71](#) shows the element type specific sub classes using `ScalarValue` and `VectorValue` as example. The sub classes of `MatrixValue` not shown are analogous.

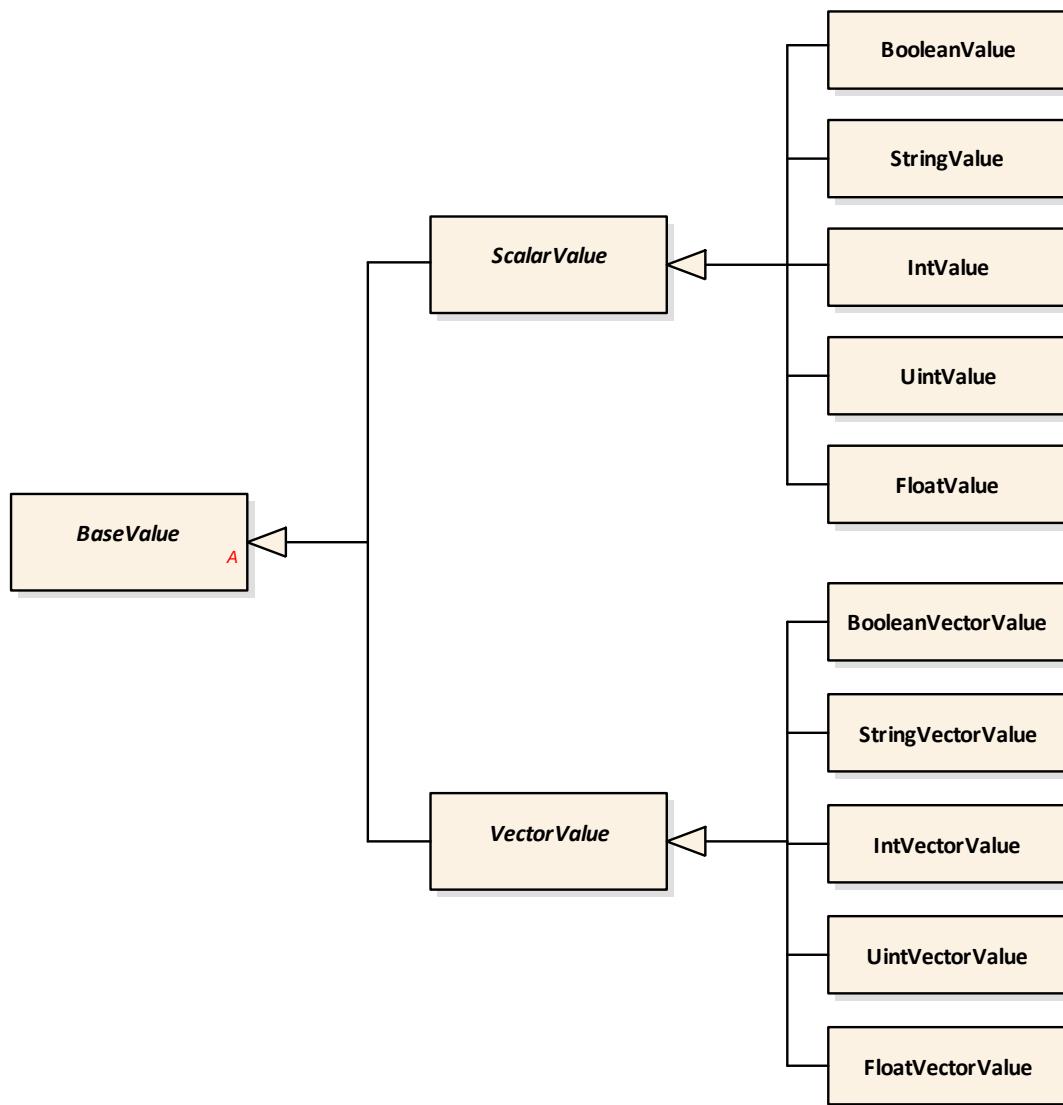


Figure 71: Element type specific sub classes of ScalarValue and VectorValue

The **ScalarValue** classes `IntValue`, `UintValue`, `FloatValue`, `StringValue` and `BooleanValue` represent a single value of the particular data type.

The **VectorValue** classes `IntVectorValue`, `UintVectorValue`, `FloatVectorValue`, `StringVectorValue` and `BooleanVectorValue` represent an ordered sequence of values. The `Count` property returns the number of values in the collection.

The **MatrixValue** classes `IntMatrixValue`, `UintMatrixValue`, `FloatMatrixValue`, `StringMatrixValue` and `BooleanMatrixValue` represent a two dimensional array of values. The `ColumnCount` and `RowCount` properties return the number of values per column and row in the matrix.

5.1.1.3 SPECIFIC VALUE CONTAINER CLASSES

Some of the `ValueContainer` classes are dedicated to more specific usage scenarios. They are depicted in [Figure 72](#)

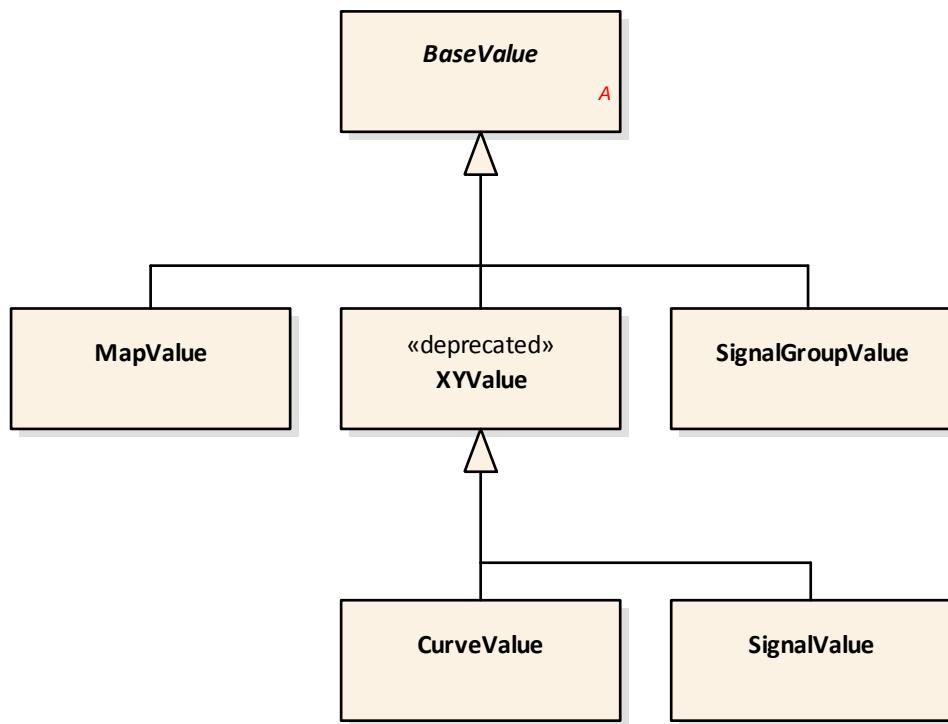


Figure 72: Specific Value classes

The **MapView** and **CurveValue** classes are widely used for calibration of curve (1D table) and map (2D table) values. Their X and Y vectors must be either monotonously increasing or decreasing and the number of rows / columns of their function values must be equal to the length of their Y / X vector.

The **SignalValue** and **SignalGroupValue** are used as numerical waveform description.

5.1.2 DOCUMENT HANDLING

The classes derived from the abstract class **DocumentManager** are designed to save and load data to/from files. Each class derived from the **DocumentManager** provides a **Load()** and a **Save()** function to store data in a particular file format. E.g. sub classes are defined for reading and writing signal descriptions, signal generator properties, capture results, and EES port configurations.

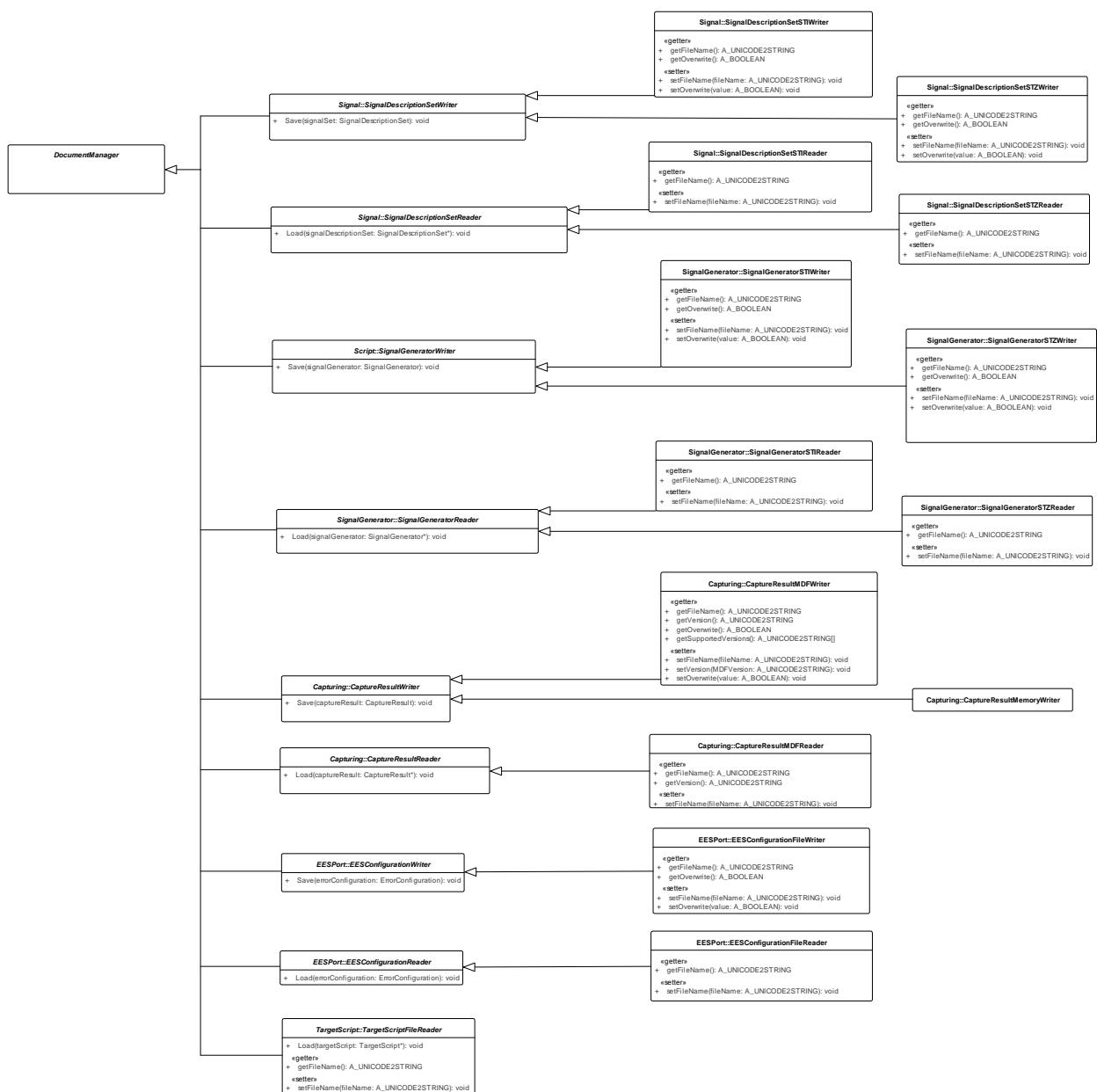


Figure 73: DocumentHandling in XIL

5.1.3 SCRIPT

5.1.3.1 OVERVIEW

Script is a common configuration and execution control interface for any type of user-defined script to be executed by the target system connected to a port. It is the base interface for the more specific interfaces TargetScript and SignalGenerator as shown in 74. This section describes the concepts and properties common to all Script objects independent from their specific type. For the specific features of TargetScript and SignalGenerator refer to 5.1.4 and 5.1.6.

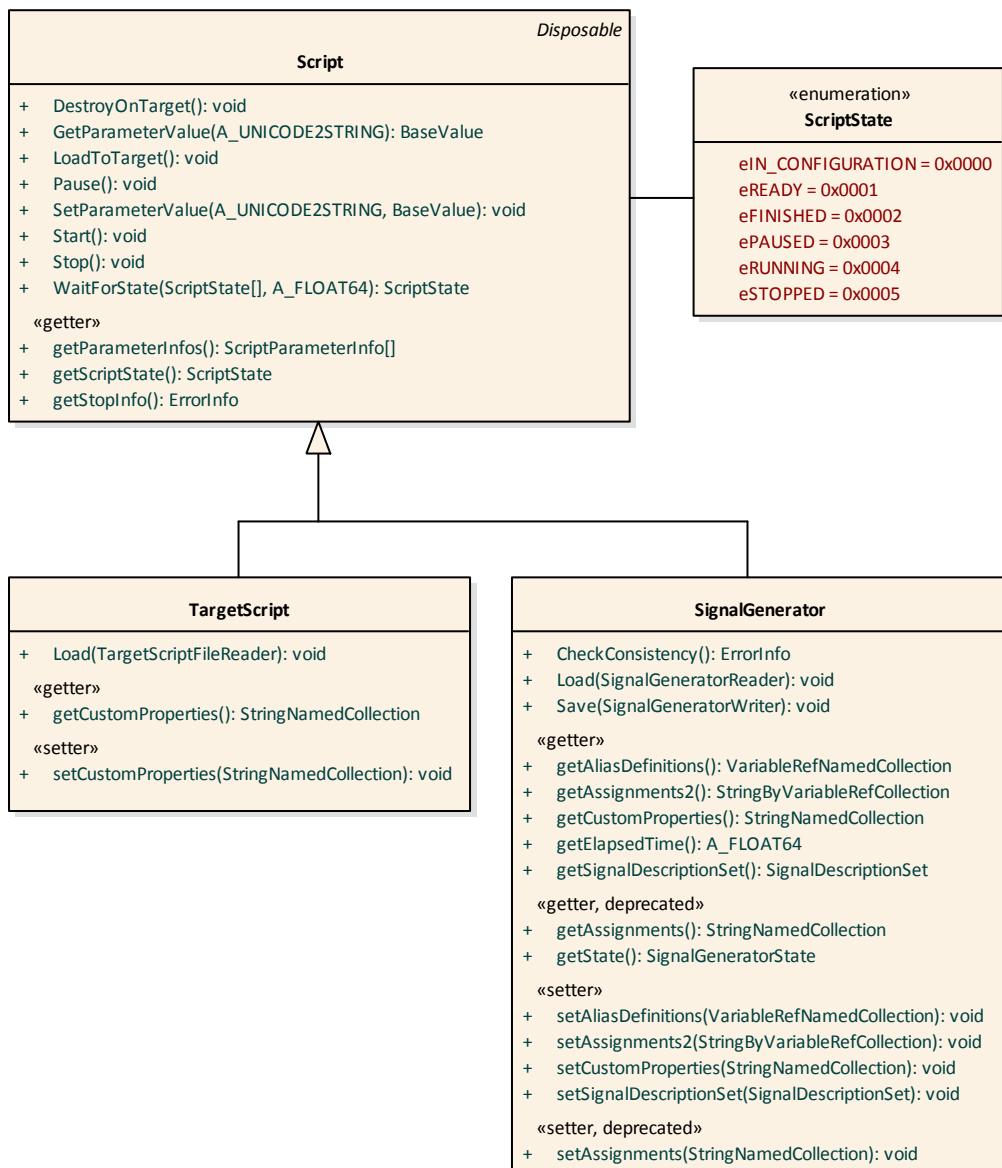


Figure 74: Script and relations to TargetScript and SignalGenerator

The `Script` interface does not have a factory, since it is only the base interface for the more specific script interfaces. That means you cannot directly create instances of `Script`, instead use the factory methods for `TargetScript` and `SignalGenerator`.

5.1.3.2 STATES OF SCRIPT

The following figure depicts the possible states and valid transitions of a `Script` object as well as the methods and events that trigger these transitions. See [Table 26](#) for a description of the `Script` states.

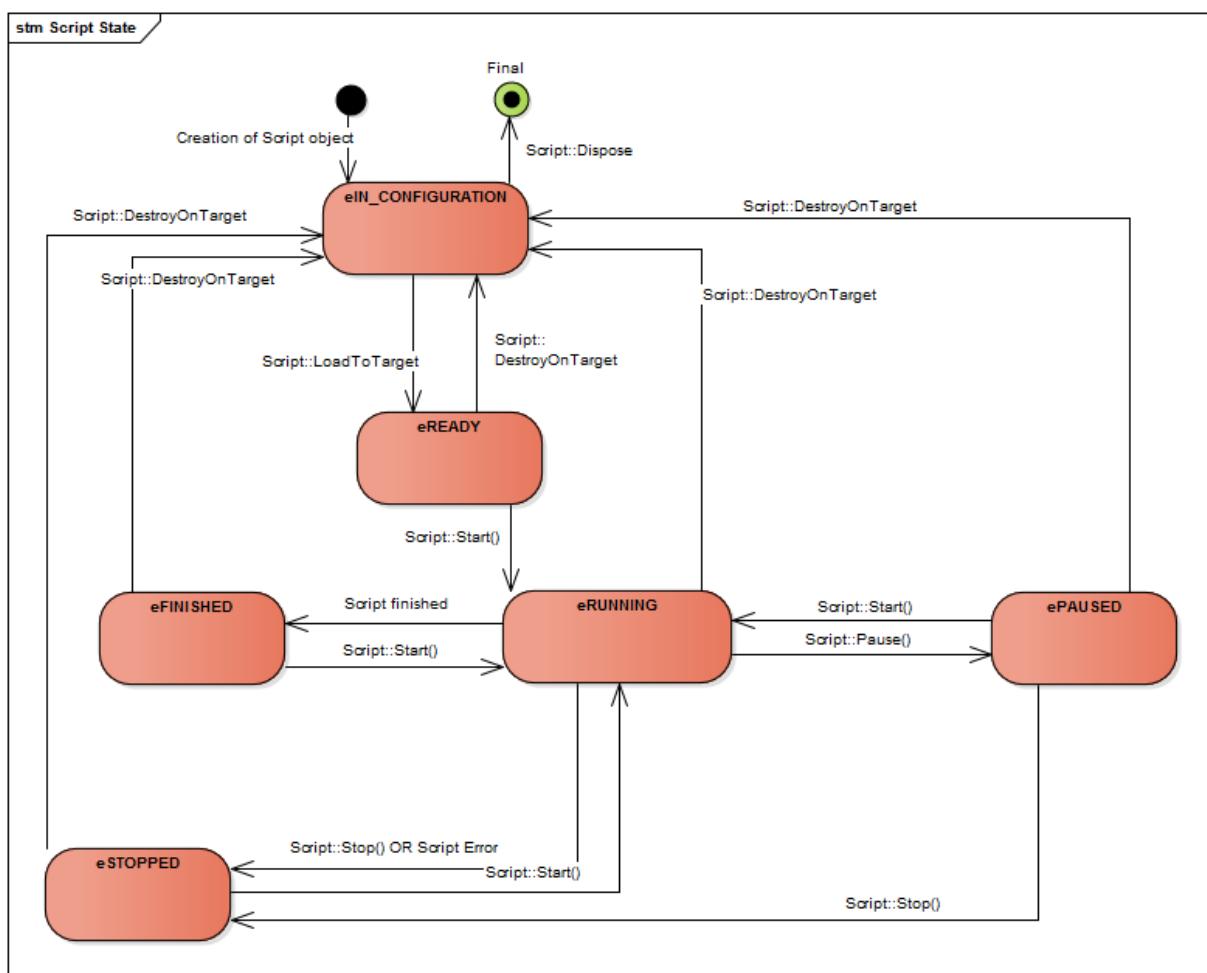


Figure 75: States and state transitions of Script objects

The state chart also shows all methods that trigger state transitions. However, state transitions only take place if execution of the respective method is completed successfully, hence no errors occur. This means in particular that all preconditions of the method must be met. Otherwise the Script's state remains unchanged. Methods, which trigger a state change, throw an exception if the intended state change could not take place.

Table 25 Script states

State	Description
eIN_CONFIGURATION	The Script has not been loaded to the target system, its content can be modified (e.g. loaded from a file), parameters for the execution can be set or modified. This is the (initial) state after creation of a Script object or after the Script has been removed from the target system.
eREADY	The Script has been loaded to the target system and is ready to run. Setting of execution parameters is restricted to eSTART_PARAMETERS.
eRUNNING	The Script is executed and its content takes effect. Note, that the Script's actual execution state is usually influenced by the state of the Port the Script

	is connected to. So although in state eRUNNING the Script's execution may be frozen or stopped when the associated Port is paused or stopped. Dependencies on the Port state are documented in the sections describing the respective Port type.
eFINISHED	Script execution has been successfully completed, i.e. has come to its natural end. Values of the Script's output parameters (if available) can be retrieved.
ePAUSED	Script execution is interrupted, but can be resumed from the point of interruption, i.e. execution status is preserved.
eSTOPPED	Script execution has been aborted, e.g. due runtime error or due client request (call of the Script's Stop method). Output parameter values are not available, but abort information can be obtained from the Script's StopInfo property.

Whether or not a method of the `Script` interface may be called depends on the `Script` state. Some of the methods are bounded to specific states. The following table gives an overall view of the permitted states for all `Script` methods and the methods specific to the derived interfaces `SignalGenerator` and `TargetScript`.

Table 26 Script, TargetScript and SignalGenerator methods in script object states

	eIN_CONFIGURATION	eREADY	eFINISHED	eRUNNING	ePAUSED	eSTOPPED
Script methods and properties						
Method DestroyOnTarget		X	X	X	X	X
Property getParameterInfos	X	X	X	X	X	X
Method GetParameterValue	X	X	X	X	X	X
Property GetScriptState	X	X	X	X	X	X
Property getStopInfo						X
Method LoadToTarget	X					
Method Pause					X	
Method SetParameterValue	X	X	X			X
Method Start		X	X		X	X
Method Stop					X	X
Method WaitForState	X	X	X	X	X	X
SignalGenerator specific methods and properties						
Method CheckConsistency	X					
Property getAliasDefinitions	X	X	X	X	X	X
Property getAssignments	X	X	X	X	X	X
Property getAssignments2	X	X	X	X	X	X
Property getCustomProperties	X	X	X	X	X	X
Property getElapsedTime	X	X	X	X	X	X
Property getSignalDescriptionSet	X	X	X	X	X	X
Property getState	X	X	X	X	X	X
Method Load	X					
Method Save	X	X	X	X	X	X
Property setAliasDefinitions	X					
Property setAssignments	X					
Property setAssignments2	X					
Property setCustomProperties	X	X	X	X	X	X
Property setSignalDescriptionSet	X					
TargetScript specific methods and properties						

Property	getCustomProperties	X	X	X	X	X	X
Method	Load	X					
Property	setCustomProperties	X	X	X	X	X	X

Please note, that a Script's behavior is usually influenced by the state of the Port the Script is connected to. These dependencies are documented in the sections describing the respective Port type. For dependencies on the MAPort state please refer to section [5.2.2.5](#).

5.1.3.3 SCRIPT PARAMETERS

Script objects may have parameters. Their count, type and meaning depend on the Script content and hence is specified by the author of the script file. With its ParameterInfos property the Script provides a means to query the available parameters and their properties. The values of the parameters can be set and read with the generic SetParameterValue and GetParameterValue methods.

The Script interface distinguishes between input and output parameters. Output parameters are read-only and read access is restricted to Script state eFINISHED. Input parameters are readable and writable. However, write access is restricted to specific states depending on the parameter type. So input parameters of type eLOAD_PARAMETER can be written in state eIN_CONFIGURATION only, whereas eSTART_PARAMETERS may also be modified after the Script has been downloaded to the target system (characterized by the states eFINISHED, eREADY, eSTOPPED).

Note: The values of output parameters cannot be queried if the Script execution has been aborted (Script state eSTOPPED). In this case use the StopInfo property to retrieve information on the abort cause.

As already mentioned, there is a ParameterInfos property that can be queried for parameter names and descriptive information on the available parameters. This includes type (eLOAD_PARAMETER, eSTART_PARAMETER or eOUTPUT_PARAMETER), data type and array dimensions (in the form of a DataTypeInfo object) as well as a human readable parameter description. The ParameterInfos property holds a list of ScriptParameterInfo objects, one for each available parameter. The ScriptParameterInfo interface is shown in [Figure 76](#).

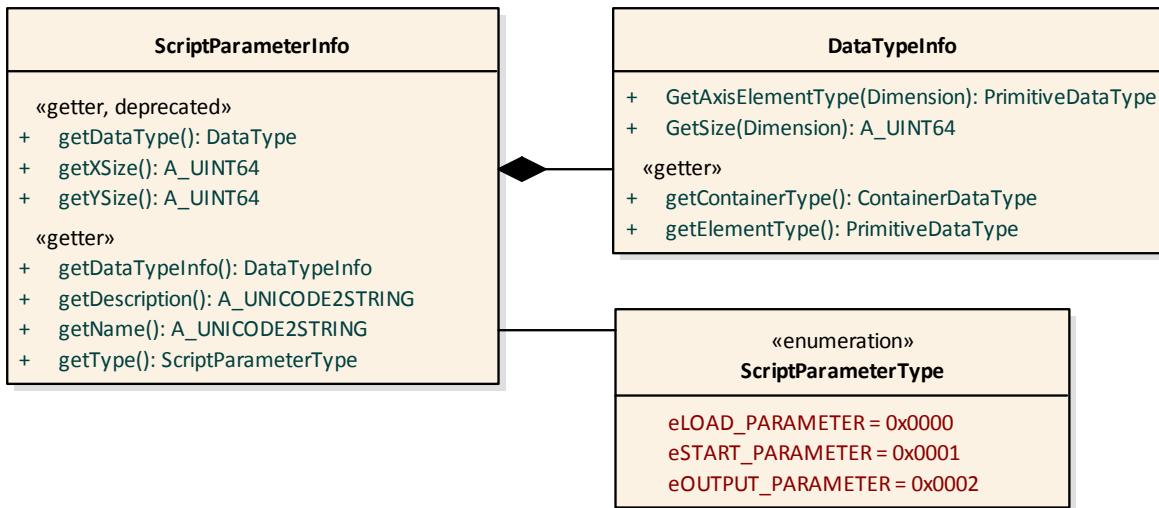


Figure 76: `ScriptParameterInfo` interface

For an example on how to use script parameters when executing a `Script` refer [Figure 78](#) illustrating the execution of `TargetScripts`.

5.1.4 TARGETSCRIPT

5.1.4.1 OVERVIEW

`TargetScript` is a specialization of the more generic `Script` concept (see chapter [Script](#)). It represents an executable, user-defined script that is used to monitor or stimulate variables synchronously in the real-time application. The `TargetScript` interface is used for loading, starting und controlling the execution of scripts commonly also known as real-time scripts. [Figure 77](#) gives an overview of the interfaces that play a role in this context. For state dependencies of the `TargetScript`'s methods refer to [Table 26](#).

Unlike the `SignalDescription` that represents the executable content of `SignalGenerator`, XIL does neither standardize an interface for programmatically creating `TargetScript` content nor a file format to store such content. So it is up to the user to properly create script files that can be processed by the used `TargetScript` implementation (e.g. by using vendor specific authoring tools or libraries provided by the respective XIL vendor).

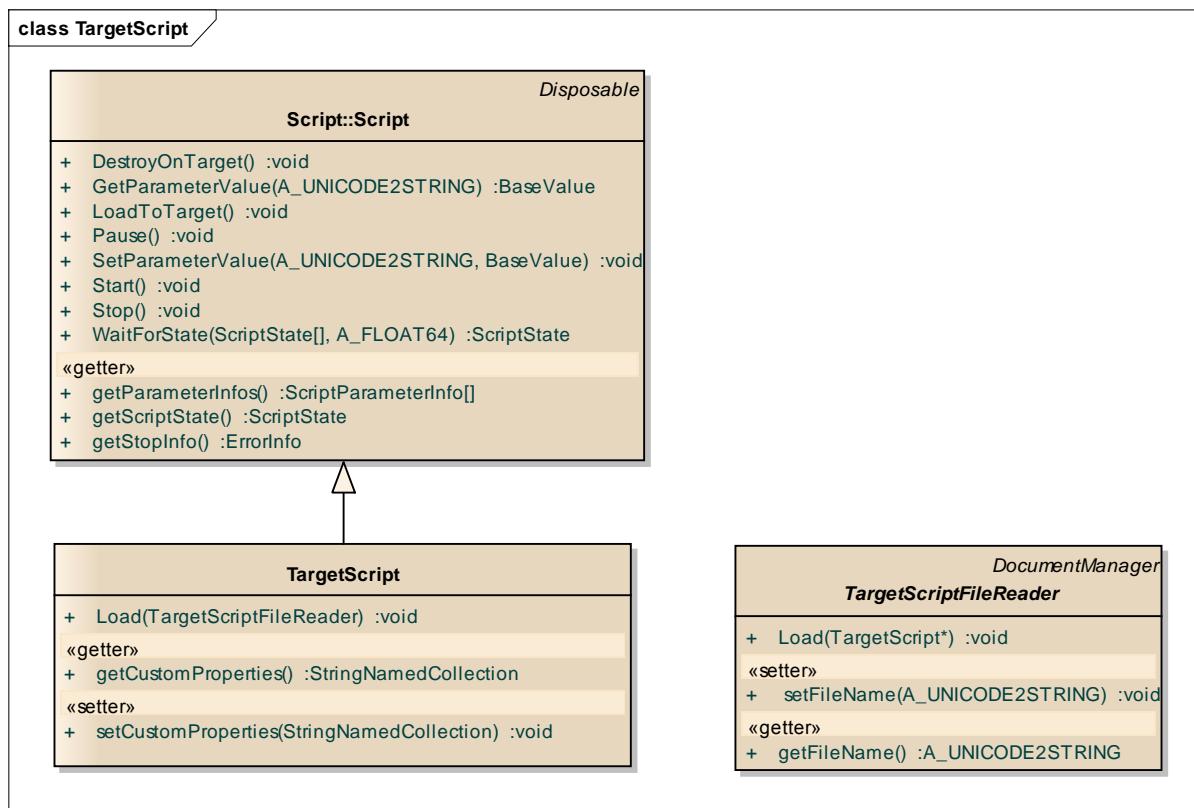


Figure 77: TargetScript interface

5.1.4.2 PARAMETERS

The script parameters of TargetScript have default values stored in the corresponding script file. It is up to the author of a script file to specify these default values. The TargetScript's parameters that can be read and set by the methods `GetParameterValue` and `SetParameterValue` are initialized with these default values every time the script file is loaded into the TargetScript instance with the `Load` method.

5.1.4.3 CUSTOM PROPERTIES

TargetScript extends the base interface `Script` by custom properties which is a generic way to configure vendor specific settings. Use the property `CustomProperties` for this purpose.

Note: Custom properties are always optional, i.e. other methods (`LoadToTarget`, `Start` etc.) do not require these properties to be set by the client. If the vendor's implementation requires a custom property that is not set by the client it is up to the implementation to assign a feasible value (although the system might not work with optimum performance in this case).

Invalid custom properties or values set by the client are signalled by the methods that make use of these properties (`LoadToTarget`, `Start` etc.) but not by the property `CustomProperties`.

5.1.4.4 USAGE OF TARGETSCRIPT

Figure 78 illustrates the general approach on executing a TargetScript that is available in form of a user-defined script file. The first step is the creation of the required TargetScript

instance via a factory method of the respective Port the TargetScript should be connected to. The example uses a simulator as target system. So the TargetScript instance is created via the `CreateTargetScript` method of the `MAPort`. For details on the creation and configuration of the `MAPort` refer to [5.2.2.1](#).

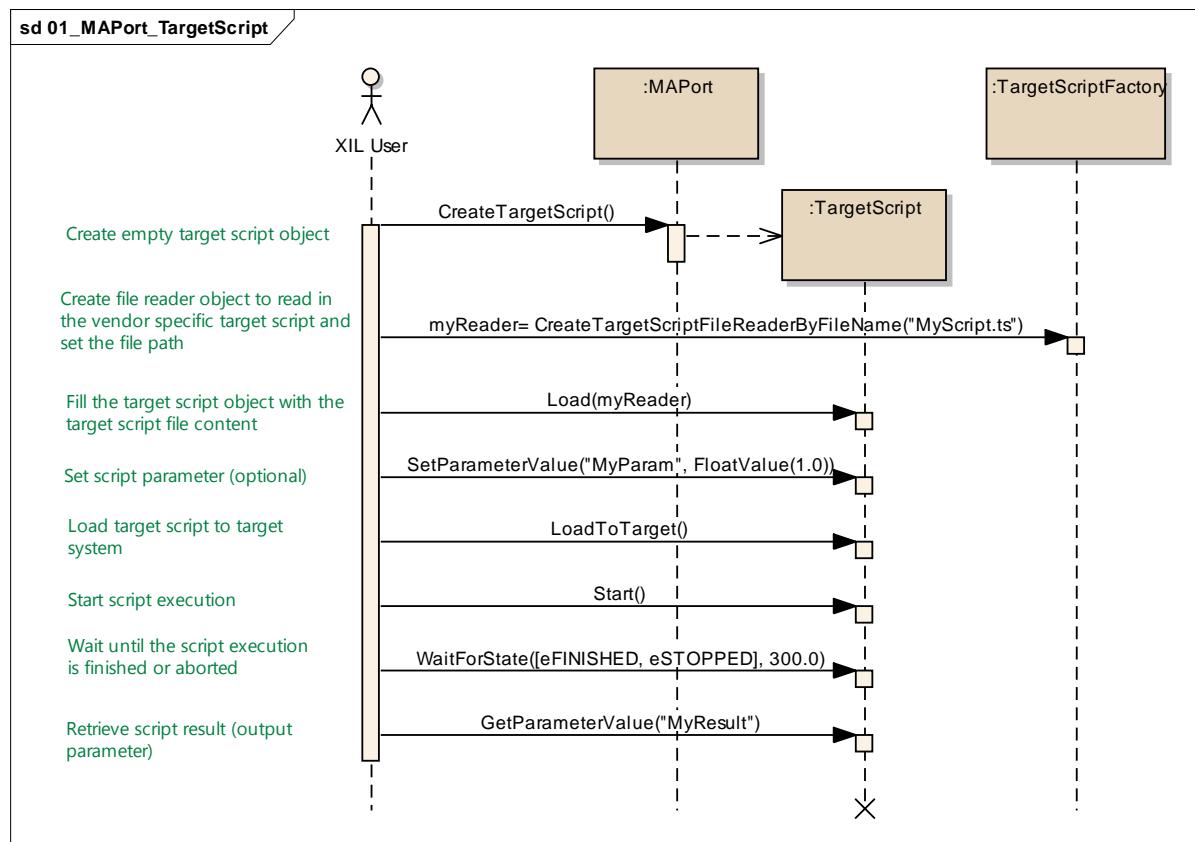


Figure 78: Example for executing a TargetScript

The second step is to read in the script file content into the `TargetScript` instance with its `Load` method. This requires an appropriate `TargetScriptFileReader` instance which supports the used vendor specific file format. That reader instance is configured with the file path. Loading the script file implicitly initializes all script parameters to their default values. But the client can adjust their values if required. (Note: Depending on parameter type adjustments might also be permitted after download to the target system. This avoids the need of repeated downloads in case the same script should be run several times with different parameterization.) For further information on the parameter concept refer to 0.

Having the script successfully loaded and parameterized it is downloaded to the target system and prepared for execution with the method `LoadToTarget`. Now the script is ready to run. Calling the `Start` method starts the execution.

In the example the client just wants to wait for the execution to be finished. For this use case it is not necessary to poll the script state. Instead `WaitForState` is called passing the states `eFINISHED` (natural end) and `eSTOPPED` (execution abort).

The final step shown in the example is the retrieval of script results. The script may provide results in the form of output parameters that can be queried on successful execution with method `GetParameterValue`.

5.1.5 SIGNAL DESCRIPTIONS

When testing ECUs via XIL simulation, signals play an important role in different use cases. In many test cases, model variables are stimulated. In other tests, variables are captured and the captured data has to be compared with reference signals. For these use cases, the XIL API introduced the classes `SignalDescription`, `SignalDescriptionSet` and `SignalGenerator`, as shown in the figure below.

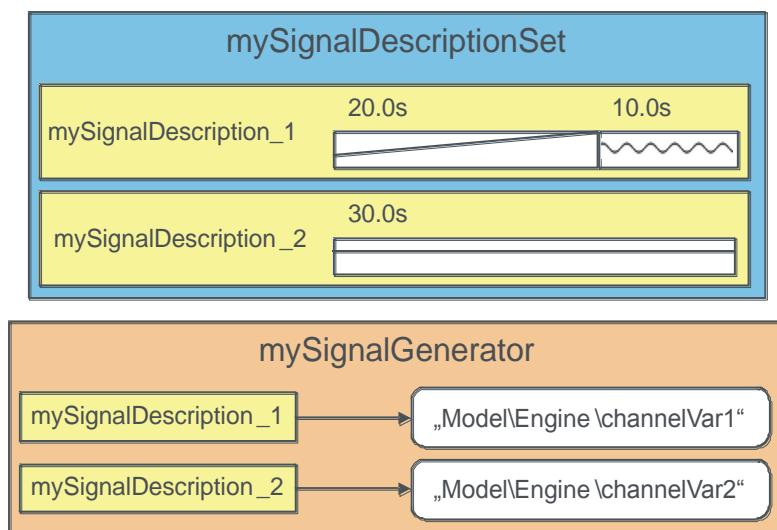


Figure 79: SignalDescriptions and SignalGenerator

A signal description consists of one or multiple segments, e.g. a ramp, followed by sine, which is denoted as "mySignalDescription_1" in the figure, or simply a constant signal denoted as "mySignalDescription_2". Many other segment types are also defined by the XIL API (see below). Such a signal description does not have any relation to variables of the simulation model. It can be used e.g. as a reference signal. Multiple signals are aggregated in a signal description set.

In order to use signals for stimulation, a signal generator is used. A signal generator relates signals to model variables and controls the signal generation process.

When modeling signals, an advanced specification is possible: The following figure shows a ramp signal, denoted as "modulateSignal" and a sine signal ("mySignalDescription_1") whose amplitude is specified by the ramp. The resulting signal is depicted besides the signal generator. The majority of segment parameters can be specified by other signals (or even by simulation model variables).

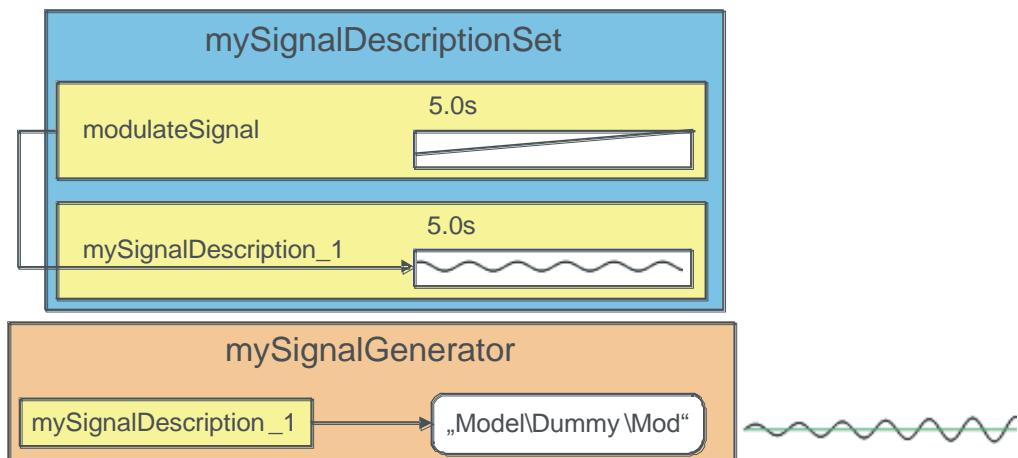


Figure 80: Modulate Signal Parameter by further Signals

Another possibility to describe a kind of modulated signals is the use of operational signal descriptions: An operational signal adds or multiplies two signals, as shown in Figure 81.

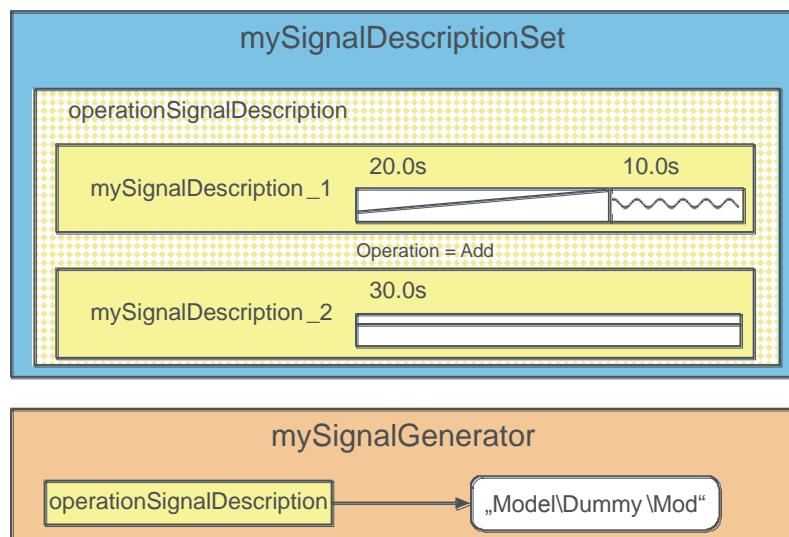


Figure 81: SignalDescriptions and SignalGenerator

In order to compare a signal description for example with sample data, i.e. signals that are defined by a couple of points in time and corresponding functional values, it is helpful to transform the signal description into an equivalent format (see Figure 82). This transformation can be done by calling the method `CreateSignalValue()` on a `SignalDescription` with the sample raster as parameter. This returns a `SignalValue` (see chapter [ValueContainer](#)) as numerical representation. Similarly, a `SignalDescriptionSet` is transformed to a numerical representation using its `CreateSignalGroupValue()` method, which returns a `SignalGroupValue`. Refer to chapter [5.1.5.3](#) for details on the transformation performed by these methods.

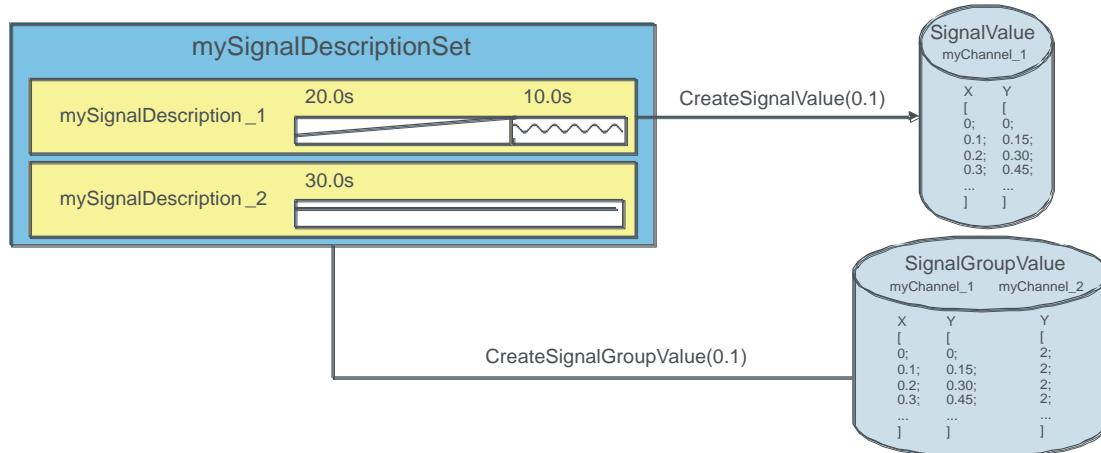


Figure 82: SignalDescriptions and SignalGenerator (data transformation)

In general the signal description is used to describe a signal for general purpose usage. A signal can be described by using synthetic waveform elements like ramp or sine and/or with elements which contain the signal points in form of numerical data.

The entry point is the class `SignalDescriptionSet` which acts as a container for signals to group several signals to one signal-set.

The `SignalDescription` is the abstract base class of `OperationSignalDescription` and `SegmentSignalDescription`.

The class `OperationSignalDescription` adds or multiplies (depends on operation property) 2 signals (left and right signal).

The `SegmentSignalDescription` is used to define a signal waveform based on a temporal sequence of different segments. Thus the `SegmentSignalDescription` is an indexed collection of signal-segments.

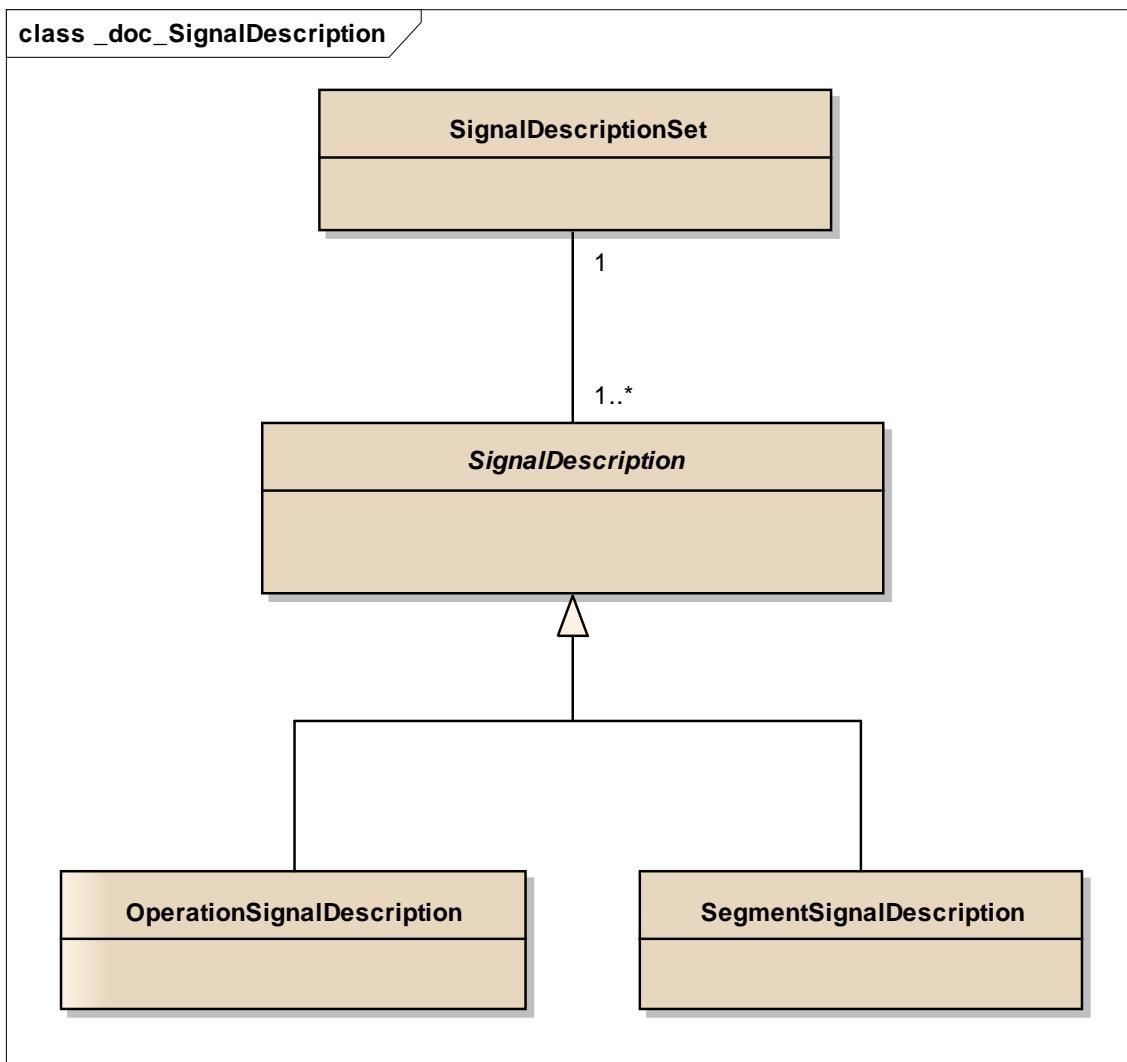


Figure 83: SignalDescription relations

5.1.5.1 GENERAL REMARKS ABOUT SEGMENT-BASED SIGNALS

A segment is the smallest unit that describes the signal form completely for a defined time period. Properties of a segment are:

Type: Each segment has a read-only property type that indicates the kind of the segment for post-analysis (`SignalSegment.getType() : SegmentTypes`).

Comment: Each segment has an optional property comment that can be used by the tester to write a description linked to the segment definition, for example to help to understand the complete signal definition.

Duration: Most of the segments have the property duration that specifies the length in time. The unit of duration is second. The `SignalValueSegment` and the `OperationSegment` have no duration property.

StopTrigger: In addition to the Duration property some segments support the property StopTrigger. This property is used to define a stop trigger for the segment. It is allowed to use a DurationWatcher with a TimeSpanDuration or a ConditionWatcher.

The StopTrigger overrides the Duration property: The evaluation of the segment will be stopped, if the given Watcher returns TRUE.

If no Watcher object is assigned to the StopTrigger property, the segment will be evaluated according to the Duration parameter. It will stop after the given duration is reached.

The other segment parameters/properties are segment specific. For example the SineSegment has the parameters amplitude, offset, period and phase.

All segment parameters use a symbolic mapping. This means that each parameter is defined via the abstract `Symbol` class, and the concrete type of the parameter is one of the sub-classes of `Symbol`. This mechanism allows different sub-classes to implement different definitions (e.g., calculation algorithms) for a segment parameter.

The sub-classes of `Symbol` are the `ConstSymbol` class (the segment parameter has a constant numeric value, i.e. does not change during execution. However the value is an expression whose parameters can be set before the execution is started. See [5.1.5.6](#) for details), the `SignalSymbol` class (the parameter's value is determined by another `SignalDescription` and hence may change dynamically during execution) and the `StringSymbol` class (the parameter's value is determined by a variable of the target system and hence may change dynamically during execution).

Thus, besides having a constant value for a segment parameter, the parameter can also be modulated. An example for this is the amplitude modulation of a `SineSegment`. It is not possible to modulate the duration of the segment by another signal, thus the duration property only accepts the `ConstSymbol`.

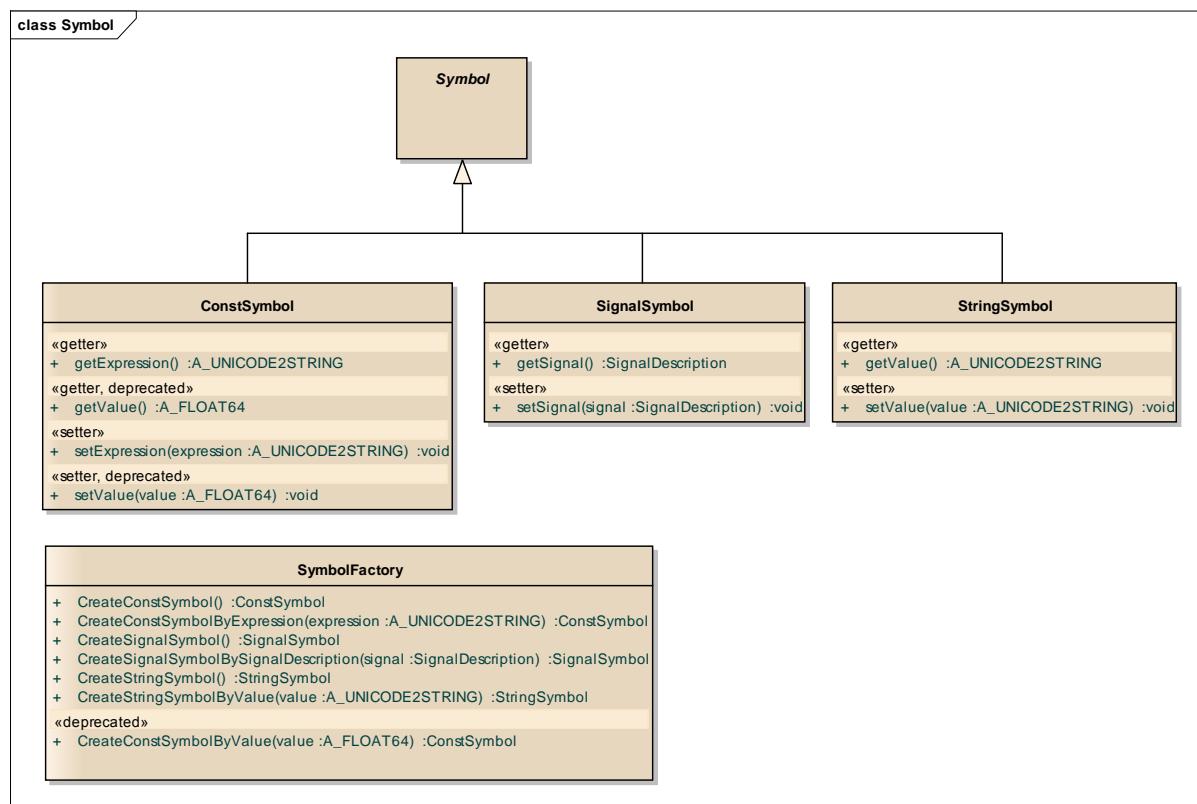


Figure 84: Symbol

Additionally segments can be combined together by operations. So you can for example add a ramp signal to a noise signal. This operation can be done by the `OperationSegment` that can be used in the same way as the native segments.

List of segments:

Synthetic Waveform Segments:

- `ConstSegment`
- `RampSegment`
- `IdleSegment`
- `NoiseSegment`
- `RampSlopeSegment`
- `SineSegment`
- `SawSegment`
- `PulseSegment`
- `ExpSegment`

Data Oriented Segments:

- `SignalValueSegment`
- `DataFileSegment`

Complex Segments:

- `OperationSegment`
- `LoopSegment`

5.1.5.2 SIGNAL SEGMENTS

ConstSegment

The ConstSegment is used to generate a part (segment) of the signal with a constant signal flow. The amplitude of the signal is on a constant value during the whole duration of the segment.

Mathematical description

$$f(t) = A$$

A : Amplitude of the signal

Graphical Representation Example:



Figure 85: ConstSegment

Table 27 Parameters ConstSegment

Parameter	Description
Duration	Duration / run time of the segment Unit: Seconds [s] Range: [0 < Duration <= MAX(A_FLOAT64)]
StopTrigger	To define a stop trigger by using a ConditionWatcher or DurationWatcher. A defined StopTrigger will override the Duration Parameter. Unit: - Range: -
Value	Value which is used as signal amplitude. Unit: - Range: [MIN(A_FLOAT64) <= Value <= MAX(A_FLOAT64)]

RampSegment

The RampSegment is used to generate a part (segment) of the signal with a ramp-shaped signal flow. The amplitude of the signal follows a straight line according to a linear equation. The slope of the line is calculated from the given start- and stop-amplitude of the ramp and the duration of the segment ($\Delta y/\Delta x$)

Mathematical description

$$f(t) = \frac{y_2 - y_1}{T_D} \cdot t + y_1$$

y_1 : Start amplitude

y_2 : Stop amplitude

T_D : Duration

Graphical Representation Example:

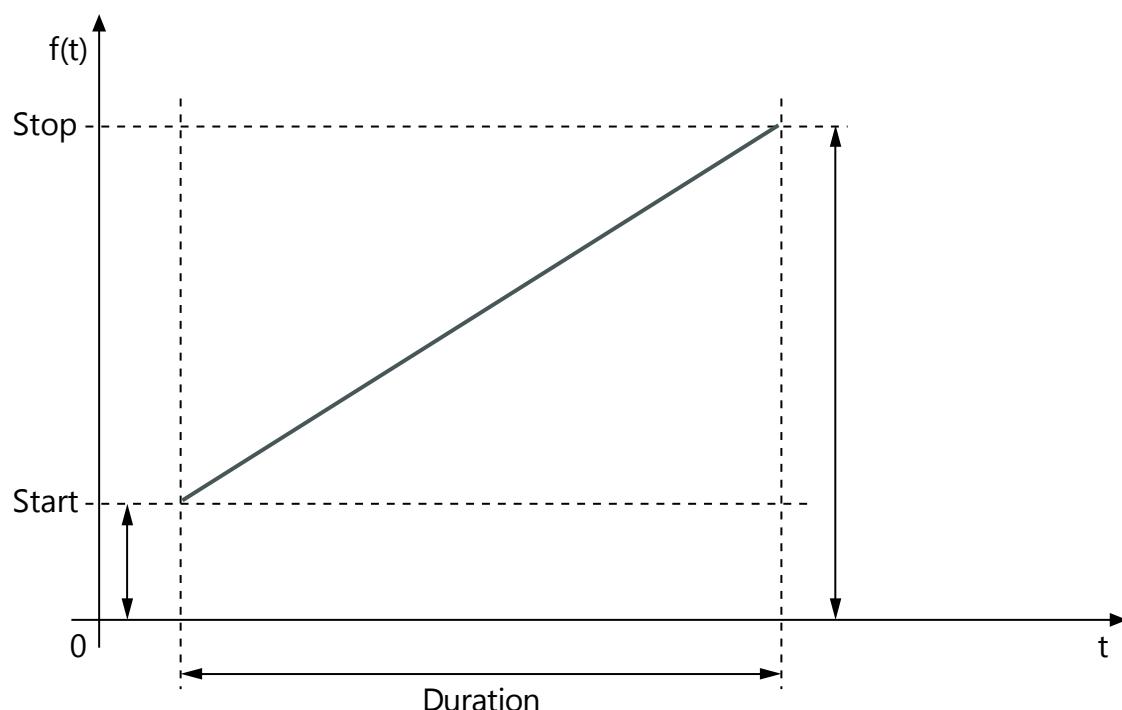


Figure 86: RampSegment

Table 28 Parameter RampSegment

Parameter	Description
Duration	Duration / run time of the segment Unit Seconds [s] Range: [0 < Duration <= MAX(A_FLOAT64)]
Start	Start value of the amplitude Unit - Range: [MIN(A_FLOAT64) <= Start <= MAX(A_FLOAT64)]
Stop	Stop value of the amplitude Unit: - Range: [MIN(A_FLOAT64) <= Stop <= MAX(A_FLOAT64)]

IdleSegment

The `IdleSegment` sets the signal generation into idle mode for the given duration. During this idle time the signal generator will not write to the corresponding model variable, respectively the memory location of the model variable.

The `IdleSegment` is normally used to allow manipulations from outside the current signal generation to take effect. If the variable is not written by such process outside the current signal generation, the variable is left untouched and will keep its value during the idle time. Please refer to chapter [5.1.6.1](#) for further details on the interpretation of the `IdleSegment` by the signal generator.

In case of evaluating an `IdleSegment` to produce numerical data e. g. when using the method `CreateSignalValue` of class `SignalDescription`, the `Nan` value (defined by [9]) must be generated (see chapter [5.1.5.3](#)). Please refer to technology references [5] and [6] for representation and handling of `Nan` values.

Mathematical description

none

Graphical Representation Example:

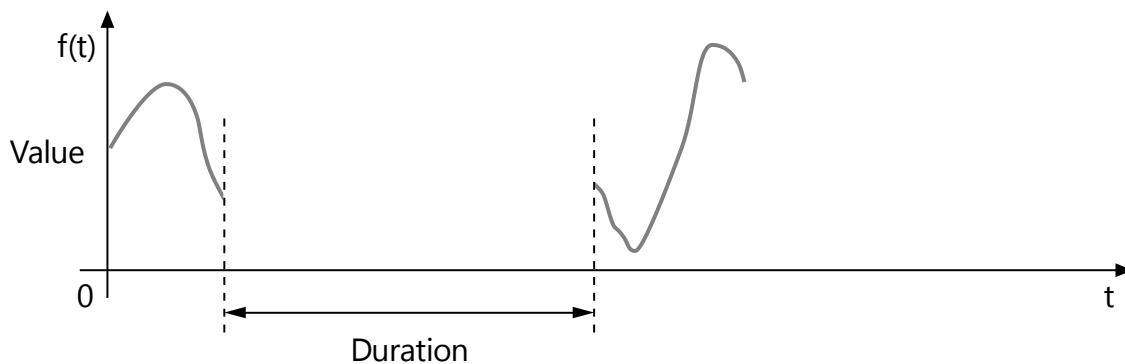


Figure 87: IdleSegment

Table 29 Parameter IdleSegment

Parameter	Description
Duration	Duration / run time of the segment Unit: Seconds [s] Range: $[0 < \text{Duration} \leq \text{MAX(A_FLOAT64)}]$
StopTrigger	To define a stop trigger by using a ConditionWatcher or DurationWatcher. A defined StopTrigger will override the Duration Parameter. Unit: - Range: -

NoiseSegment

The NoiseSegment is used to generate a part (segment) of the signal with gaussian noise. That means that the amplitude of the signal is gaussian distributed.

In each model step one noise value is calculated by using a random generator. The generated random value is than applied against the gaussian distribution to get amplitude values according to the gaussian bell-shaped curve.

Mathematical description

Gaussian Distribution:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Box-Muller-Method:

From two standard independent random numbers u_1 and u_2 in the range 0..1 (e.g. generated via `random()`) two standard normal-distributed and independent random numbers z_1 and z_2 will be created.

$$z_1 = \sqrt{-2 \cdot \ln(1 - u_1)} \cdot \cos(2\pi \cdot u_2)$$

and

$$z_2 = \sqrt{-2 \cdot \ln(1 - u_1)} \cdot \sin(2\pi \cdot u_2)$$

With

$$x_i = \mu + \sigma \cdot z_i$$

It is possible to generate normal distributed random numbers x_i with any mean and sigma parameters you need.

μ : Mean value

σ : Standard deviation

Note: The Box-Muller-Method is used by the Python function `random.gauss(mu, sigma)`.

Graphical Representation Example:

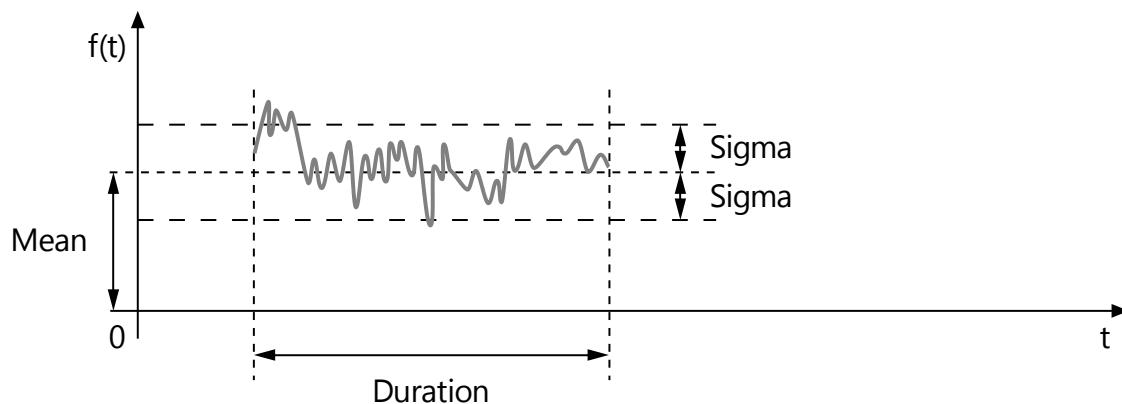


Figure 88: NoiseSegment

Table 30 Parameter NoiseSegment

Parameter	Description
Duration	Duration / run time of the segment Unit: Seconds [s] Range: $[0 < \text{Duration} \leq \text{MAX (A_FLOAT64)}]$
Mean	Mean value, where the Gaussian distribution is moving Unit: - Range: $[\text{MIN (A_FLOAT64)} \leq \text{Mean} \leq \text{MAX (A_FLOAT64)}]$
Sigma	Standard deviation of the signal amplitude against the mean value Unit: - Range: $[\text{MIN (A_FLOAT64)} \leq \text{Sigma} \leq \text{MAX (A_FLOAT64)}]$
Seed	Start value of the random generator Unit: - Range: $[\text{MIN(A_INT32)} \leq \text{Seed} \leq \text{MAX(A_INT32)}]$
StopTrigger	To define a stop trigger by using a ConditionWatcher or DurationWatcher. A defined StopTrigger will override the Duration Parameter. Unit: - Range: -

RampSlopeSegment

The RampSlopeSegment is used to generate a part (segment) of the signal with a ramp-shaped signal flow. The amplitude of the signal follows a straight line according to a linear equation.

The segment form is similar to RampSegment. Only the parameters are different.

Mathematical description

$$f(t) = m \cdot t + b$$

m : Slope of the line

b : Offset of the line

Graphical Representation Example:

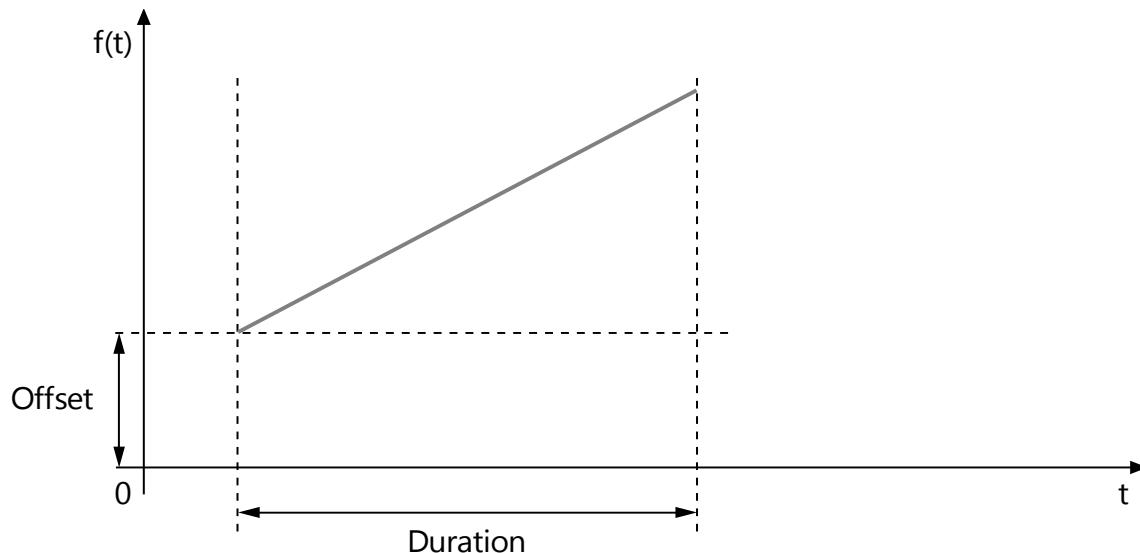


Figure 89: RampSlopeSegment

Table 31 Parameter RampSlopeSegment

Parameter	Description
Duration	Duration / run time of the segment Unit: Seconds [s] Range: [0 < Duration <= MAX (A_FLOAT64)]
Offset	Offset of the ramp Unit: - Range: [MIN (A_FLOAT64) <= Offset <= MAX (A_FLOAT64)]
Slope	Slope of the ramp Unit: - Range: [MIN (A_FLOAT64) <= Slope <= MAX (A_FLOAT64)]
StopTrigger	To define a stop trigger by using a ConditionWatcher or DurationWatcher. A defined StopTrigger will override the Duration Parameter. Unit: - Range: -

SineSegment

The SineSegment is used to generate a part (segment) of the signal with a sine-shaped signal flow. The amplitude of the signal follows a periodical sine-waveform.

Mathematical description

$$f(t) = A \cdot \sin\left(\frac{2\pi}{T} \cdot (t + \varphi \cdot T)\right) + b$$

A : Amplitude of the Signal

T : Cycle time

φ : Initial phase shift as a factor of the cycle time

b : Offset of the Signal

Graphical Representation Example:

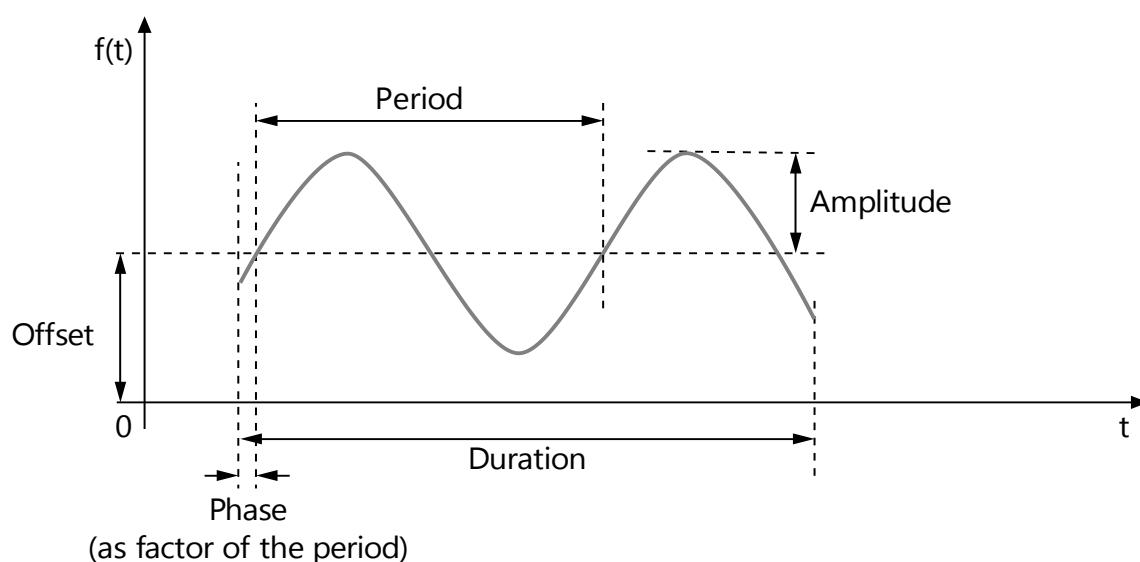


Figure 90: SineSegment

Table 32 Parameter SineSegment

Parameter	Description
Duration	Duration / run time of the segment Unit: Seconds [s] Range: [0 < Duration <= MAX (A_FLOAT64)]
Offset	Offset of the sine waveform Unit: - Range: [MIN (A_FLOAT64) <= Offset <= MAX (A_FLOAT64)]
Period	Cycle time of the sine waveform Unit: - Range: [0 < Period <= MAX (A_FLOAT64)]
Amplitude	Amplitude of the sine waveform Unit: - Range: [MIN (A_FLOAT64) <= Amplitude <= MAX (A_FLOAT64)]
Phase	Initial phase shift as positive or negative factor of the cycle time Unit: - Range: [-1.0 <= Phase <= +1.0] (0.25 is equal to 90° phase shift, -0.33 is equal to -120° phase shift) The variation of the period with another signal is not equivalent to frequency modulation.
StopTrigger	To define a stop trigger by using a ConditionWatcher or DurationWatcher. A defined StopTrigger will override the Duration Parameter. Unit: - Range: -

SawSegment

The SawSegment is used to generate a part (segment) of the signal with a saw tooth shaped or triangle shaped signal flow. The amplitude of the signal follows a periodical saw tooth waveform.

Mathematical description

$$f(t) = \begin{cases} \frac{A}{t_r} t + b & 0 < t + \varphi \cdot T < t_r \\ A - \frac{A}{t_f}(t - t_r) + b & t_r < t + \varphi \cdot T < t_f \end{cases}, t_r = T * \delta, t_f = T - t_r, t_r \neq 0, t_f \neq 0$$

A : Amplitude of the Signal

T : Cycle time

δ : Duty cycle (ratio of rise-time to cycle-time) as factor of the cycle time

t_r : Rise time

t_f : Fall time

φ : Initial phase shift as factor of the cycle time

b : Offset of the Signal

Graphical Representation Example:

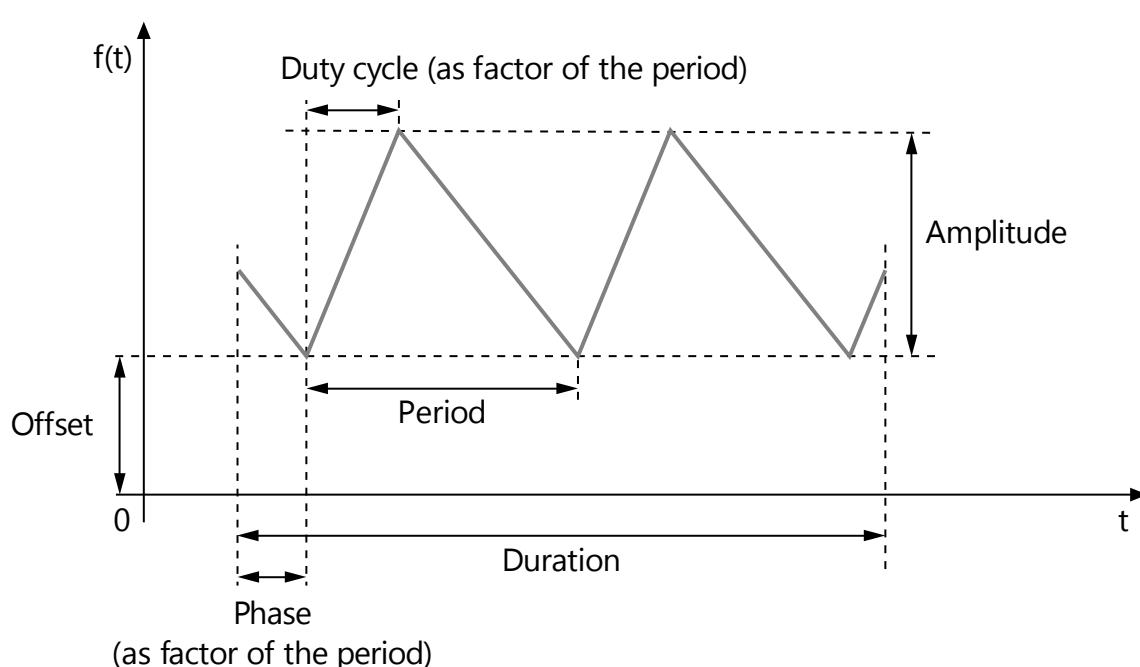


Figure 91: SawSegment

Table 33 Parameter SawSegment

Parameter	Description
Duration	Duration / run time of the segment Unit: Seconds [s] Range: [0 < Duration <= MAX (A_FLOAT64)]
Offset	Offset of the saw tooth waveform Unit: - Range: [MIN (A_FLOAT64) <= Offset <= MAX (A_FLOAT64)]
Period	Cycle time of the saw tooth waveform Unit: - Range: [0 < Period <= MAX (A_FLOAT64)]
Amplitude	Amplitude of the saw tooth waveform Unit: - Range: [MIN (A_FLOAT64) <= Amplitude <= MAX (A_FLOAT64)]
Phase	Initial phase shift as positive or negative factor of the cycle time Unit: - Range: [-1.0 <= Phase <= +1.0] (0.25 is equal to 90° phase shift, -0.33 is equal to -120° phase shift)
DutyCycle	Ratio of raise-time to cycle-time as a positive factor Unit: - Range: [0.0 <= DutyCycle <= 1.0] (use 0.5 to get a triangular shaped signal) The variation of the period with another signal is not equivalent to frequency modulation.
StopTrigger	To define a stop trigger by using a ConditionWatcher or DurationWatcher. A defined StopTrigger will override the Duration Parameter. Unit: - Range: -

PulseSegment

The PulseSegment is used to generate a part (segment) of the signal with a rectangular-shaped signal flow. The amplitude of the signal follows a periodical rectangle-waveform.

Mathematical description

$$f(t) = \begin{cases} A+b & 0 < t + \varphi \cdot T < t_h \\ b & t_h < t + \varphi \cdot T < T \end{cases}, \quad t_h = T \cdot \delta$$

A : Amplitude of the Signal

T : Cycle time

δ : Duty cycle (ratio of high-time to cycle-time) as factor of the cycle time

t_h : High-time

φ : Initial phase shift as factor of the cycle time

b : Offset of the Signal

Graphical Representation Example:

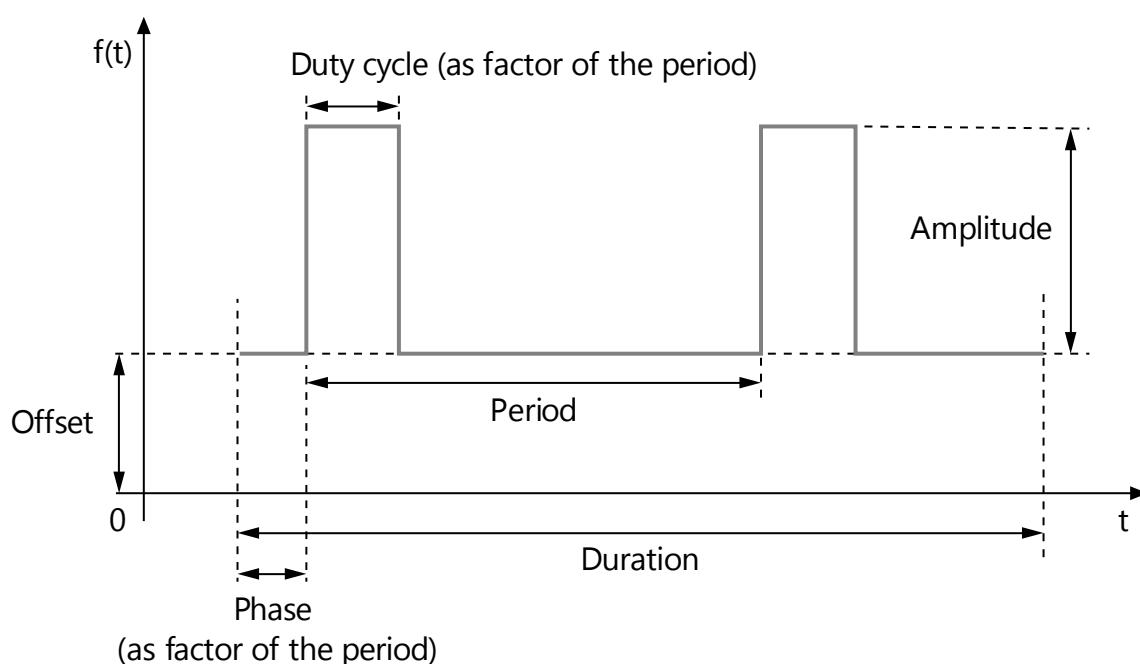


Figure 92: PulseSegment

Table 34 Parameter PulseSegment

Parameter	Description
Duration	Duration / run time of the segment Unit: Seconds [s] Range: [0 < Duration <= MAX (A_FLOAT64)]
Offset	Offset of the rectangle waveform Unit: - Range: [MIN (A_FLOAT64) <= Offset <= MAX (A_FLOAT64)]
Period	Cycle time of the rectangle waveform Unit: - Range: [0 < Period <= MAX (A_FLOAT64)]
Amplitude	Amplitude of the rectangle waveform Unit: - Range: [MIN (A_FLOAT64) <= Amplitude <= MAX (A_FLOAT64)]
Phase	Initial phase shift as positive or negative factor of the cycle time Unit: - Range: [-1.0 <= Phase <= +1.0] (0.25 is equal to 90° phase shift, -0.33 is equal to -120° phase shift)
DutyCycle	Ratio of high-time to cycle-time as a positive factor Unit: - Range: [0.0 <= DutyCycle <= 1.0] (use 0.5 to get a symmetric rectangular shaped signal, use 1.0 to get a constant value) The variation of the period with another signal is not equivalent to frequency modulation.
StopTrigger	To define a stop trigger by using a ConditionWatcher or DurationWatcher. A defined StopTrigger will override the Duration Parameter. Unit: - Range: -

ExpSegment

The ExpSegment is used to generate a part (segment) of the signal with an exponential-shaped signal flow. The amplitude of the signal follows an exponential curve.

Mathematical description

$$f(t) = A \cdot (1 - e^{-\frac{t}{\tau}}) + b$$

A : Amplitude of the Signal

τ : Time constant (tau)

b : Offset of the Signal

Graphical Representation Example:

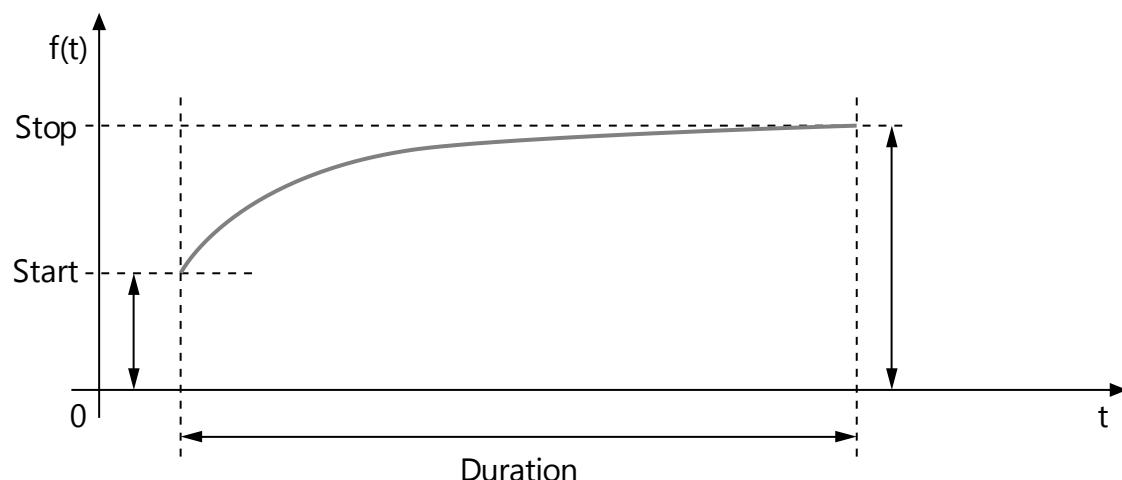


Figure 93: ExpSegment

Table 35 Parameter ExpSegment

Parameter	Description
Duration	Duration / run time of the segment Unit: Seconds [s] Range: [0 < Duration <= MAX (A_FLOAT64)]
Start	Start amplitude (Offset of the Signal) Unit: - Range: [MIN (A_FLOAT64) <= Start <= MAX (A_FLOAT64)]
Stop	Stop amplitude (Note: Amplitude of the Signal A = Stop – Start) Unit: - Range: [MIN (A_FLOAT64) <= Stop <= MAX (A_FLOAT64)]
StopTrigger	To define a stop trigger by using a ConditionWatcher or DurationWatcher. A defined StopTrigger will override the Duration Parameter. Unit: - Range: -
Tau	Time constant of the e-curve Unit: Seconds [s] Range: [0 < Tau <= MAX (A_FLOAT64)]

SignalValueSegment

The `SignalValueSegment` is used to generate a part (segment) of the signal which directly uses numerical data. The amplitude is the result of the data points of the numerical data and the given interpolation type.

Normally this segment is used to replay measured data.

The numerical (respective measured) data is stored in a `SignalValue` object (see chapter [ValueContainer](#)) which is given during creation of the segment or during configuration of the segment. The duration of the segment is derived from the time vector.

The serialization of the numerical data (e.g. `SignalDescriptionSet.Save()`) is done by generating a flat MATLAB-File with two vectors of type double. One vector describes the time vector, and the other vector describes the corresponding signal amplitude values.

The duration of the segment is implicitly derived from the time vector. For more information see chapter [Signal Description File](#).

An automatic offset correction of all time stamps is performed when playing the signal description by the `SignalGenerator`, so that the first time stamp equals the segment's start time.

Table 36 Parameter SignalValueSegment

Parameter	Description
SignalValue	SignalValue object which contains the time-vector and the data-vector Unit: time-vector: Seconds [s], data-vector: - Range: [MIN (A_FLOAT64) <= time, data <= MAX (A_FLOAT64)]
Interpolation	Interpolation method Unit: - Range: enum InterpolationTypes eFORWARD: Next data point will be used immediately (staircase forward) eBACKWARD: Actual data point will be used until next data point (staircase backward) eLINEAR: Linear interpolation

DataFileSegment

With the DataFileSegment it is possible to use numerical data, which is stored in a file, within a signal description. The data file is typically a measurement data file which contains some measured signals and one or more time axis resp. raster information. The DataFileSegment holds a link to the data file. The DataFileSegment does not store or serialize the used numerical data. So it is very easy to switch between different numerical data by referencing another file, e.g. a measurement data file from a newer measurement that should be used for the signal description. Another goal is to use the same measurement data file in different contexts, like stimulation and reference signal comparison.

The DataFileSegment allows the user to select a specific range of values from a data file. For this purpose, the properties Start and Duration of the DataFileSegment are used. These properties have the initial values '-INF' and '+INF'. The following table shows different combinations of values of these properties and how they affect the range of data values that is used for stimulation.

Table 37 Start and Duration Parameter Values of of DataFileSegment

Start	Duration	Comment
-INF	+INF	Replays the complete data file.
-INF	const2	Starting at the beginning of the numeric data in the file, the data file is replayed until the specified duration has passed.
const1	+INF	Starting at a specific time stamp, the complete file is replayed.
const1	const2	A specific section of the file is replayed.

If either const1 or const2 is out of the range of timestamps in the data file, an exception will be thrown. So the user has the possibility to adapt the Start or Duration of the method call or to exchange the used numerical data material.

An automatic offset correction of all time stamps is performed when playing the signal description by the SignalGenerator, so that the first time stamp equals the segment's start time.

Table 38 Parameter DataFileSegment

Parameter	Description
Duration	Duration / run time of the segment Unit: Seconds [s] Range: [0 < Duration <= +INF] Initial value: +INF Note: +INF represent the PositiveInfinity value of double corresponding to IEEE.
FileName	Name or link of the data file.
DataVectorName	Name of the data vector / signal to be use
TimeVectorName	Name of the time vector / raster to use
Interpolation	Interpolation method Unit: - Range: enum InterpolationTypes eFORWARD: Next data point will be used immediately (staircase forward) eBACKWARD: Actual data point will be used until next data point (staircase backward) eLINEAR: Linear interpolation
Start	Start point of the numerical data to be used. Unit: Seconds [s] Range: [-INF <= Start < +INF] Initial value: -INF Note: +INF or -INF represents the PositiveInfinity or the NegativInfinity value of double corresponding to IEEE.
StopTrigger	To define a stop trigger by using a ConditionWatcher or DurationWatcher. A defined StopTrigger will override the Duration Parameter. Unit: - Range: -
ChannelSource	The channel source of the two channels (signals) TimeVector and DataVector. The channel source typically contains the network node name or the task name.
ChannelPath	The channel path of the two channels (signals) TimeVector and DataVector. The channel path typically contains the device name or platform name.
GroupName	The group name of the two channels (signals) TimeVector and DataVector. The group name typically contains the task name, the raster name or the event name.
GroupSource	The group source of the two channels (signals) TimeVector and DataVector. The group source typically contains the network node name, the tool name or the task name.
GroupPath	The group path of the two channels (signals) TimeVector and DataVector. The group path typically contains the device name or platform name.

LoopSegment

The LoopSegment is used to repeat its containing child segments. The loop segment can be used to repeat sequences of synthetic segments, for example a trapezoid shaped wave form that should be evaluated/executed for a couple of times designed with a leading RampSegment for the rising edge, a following ConstSegment and a trailing RampSegment for the falling edge. The total number of executions/evaluations is given by the loop count property.

Table 39 Parameter LoopSegment

Parameter	Description
LoopCount	The number of times the child segments will be executed / evaluated. Unit: - Range: [1 <= LoopCount <= MAX (A_UINT64)]

OperationSegment (Operationtypes)

The OperationSegment is used to generate a part (segment) of the signal which is a combination of two other segments. The two segments are combined by a mathematical operation like addition or multiplication. The amplitude of the signal follows the calculated result. The duration of the resulting segment is derived from the shorter segment.

Mathematical description

$$f(t) = S_1(t) \text{ op } S_2(t) \quad , \text{op} = \text{operation}$$

S_1 : First segment / first operand

S_2 : Second segment / second operand

Table 40 Parameter OperationSegment

Parameter	Description
leftSegment	left segment object (left operand s1) Unit: - Range: -
rightSegment	right segment object (right operand s2) Unit: - Range: -
Operation	Operation which is used to calculate the corresponding signal Unit: - Range: enum OperationTypes eADD: Addition ($y(t) = s1(t) + s2(t)$) eMULT: Multiplication ($y(t) = s1(t) * s2(t)$)

5.1.5.3 CONVERTING SIGNALDESCRIPTIONS TO SAMPLE BASED REPRESENTATIONS

The use of SignalDescriptions often requires the calculation of concrete signal values for certain points in time or the conversion of the analytical description into a numerical, time-discrete representation. For this purpose, the SignalDescription interface provides the CreateSignalValue method. This method generates a list of consecutive values (samples) with equidistant time stamps in the form of a SignalValue.

While the use of equidistant time stamps (i.e. a constant sampling rate) is mandatory for the CreateSignalValue method, XIL does not restrict the way of time stamp selection in general. Hence, vendor specific conversion methods and tools may use dynamic sampling rate adaption or skip samples during constant value periods etc.

Basis for any conversion of a SignalDescription is the definition that each time interval defined by a SignalSegment is closed on the left and open on the right. Consequently the interval of the complete SignalDescription (which is the concatenation of SignalSegments) is also closed on the left, but open on the right. This results in the following conditions that each conversion method must meet, no matter if equidistant time stamps are used or not:

- **C1:** The first sample value refers to time stamp 0 and corresponds to the start value of the first `SignalSegment`.
- **C2:** Sample values within the time limits of a `SignalSegment` are determined by the segment's calculation rule. In particular, `IDLESegments` yield NaN as sample values.
- **C3:** The value of a sample exactly on the boundary of two consecutive `SignalSegments` corresponds to the start value of the successor segment. This is particularly important for discontinuous segment transitions (i.e. successor segment's start value differs from predecessor segment's end value) or if the successor segment is an `IDLESegment` (NaN as sample value).
- **C4:** The sample exactly on the right boundary of the last `SignalSegment` (i.e. at the endpoint of the `SignalDescription`) has the value NaN.
- **C5:** Samples with time stamps beyond the end of the `SignalDescription` have the value NaN. It is recommended that conversion methods do not return lists with samples beyond the endpoint of the `SignalDescription`.

For the conversion of a `SignalDescriptionSet` (i.e. a group of `SignalDescriptions` with common time base) into a numerical, time-discrete representation, the equally named interface provides the method `CreateSignalGroupValue`. The returned `SignalGroupValue` contains a sample value for each signal at each time stamp contained. The count of sample values is therefore the same for all signals and corresponds to the number of contained time stamps.

To ensure that the number of samples per signal is the same even with `SignalDescriptions` of different lengths, the otherwise shorter sample sequences are padded with NaN values at the end.

The conditions for the conversion of `SignalDescriptions` as mentioned above also apply to the conversion of `SignalDescriptionSets`. The restriction of `CreateSignalGroupValue` to the use of equidistant time stamps is also not fundamentally important. Vendor specific conversion methods and tools are permitted to use alternative sampling strategies.

5.1.5.4 USING SIGNAL DESCRIPTIONS

Each `SegmentSignalDescription` consists of one or more segments. The sequence diagram illustrates the creation of a `SignalDescription` that consists of several segments.

After creating instances of the segments, these instances are added to the `SegmentSignalDescription` object.

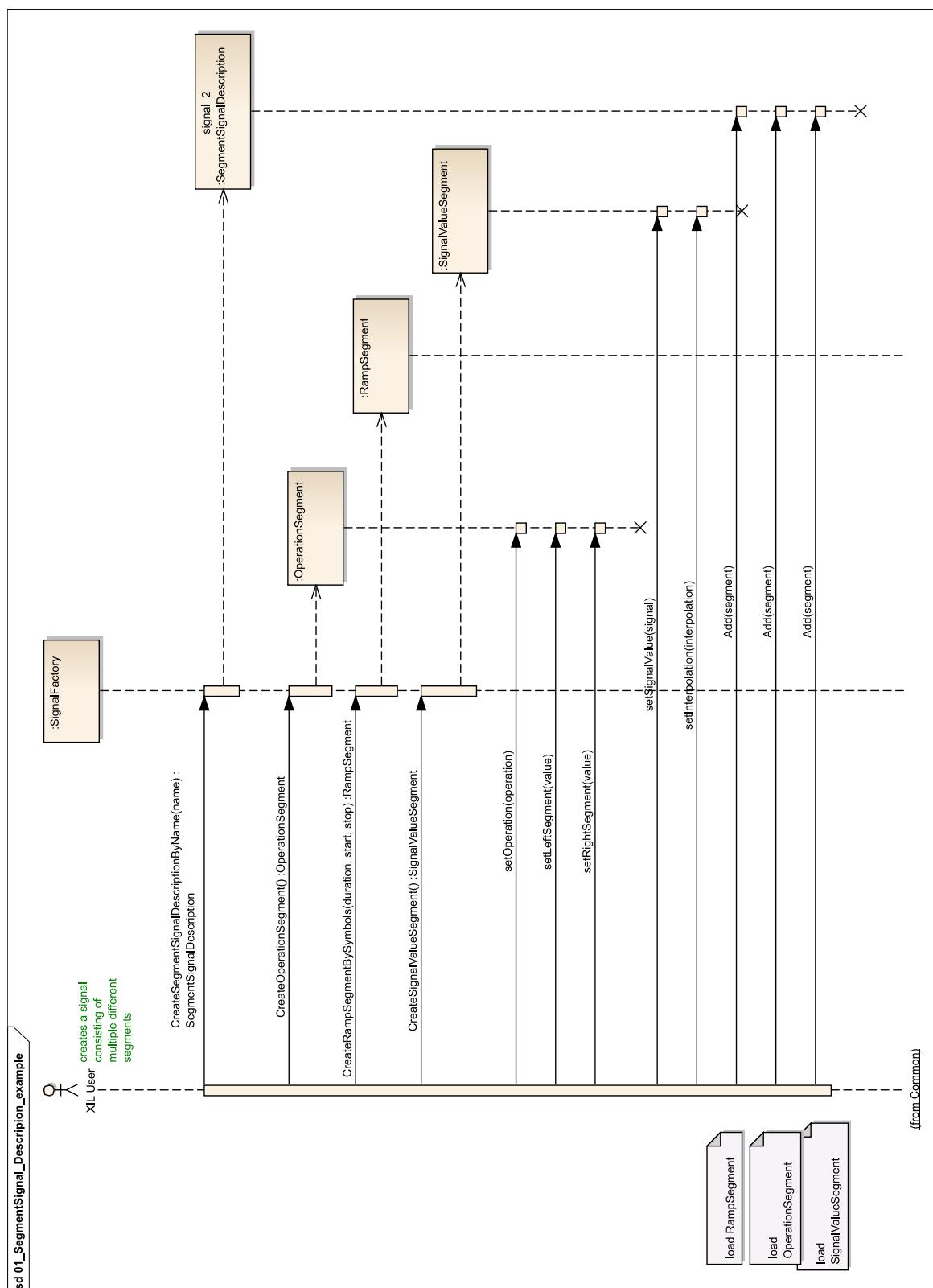


Figure 94: Create Segment Signal Description Example

Creating an Operation Signal

The sequence diagram [Figure 95](#) is describing the creation of an `OperationSignal` in detail. It consists of two `SegmentSignalDescriptions` which are combined by the given operation. The `SignalDescriptions` itself can have more than one signal segment inside. In this case the first has 2 signal segments (`RampSegment` and `SawSegment`) and the second has only one signal segment (`SineSegment`). The operation in this example is Multiplication.

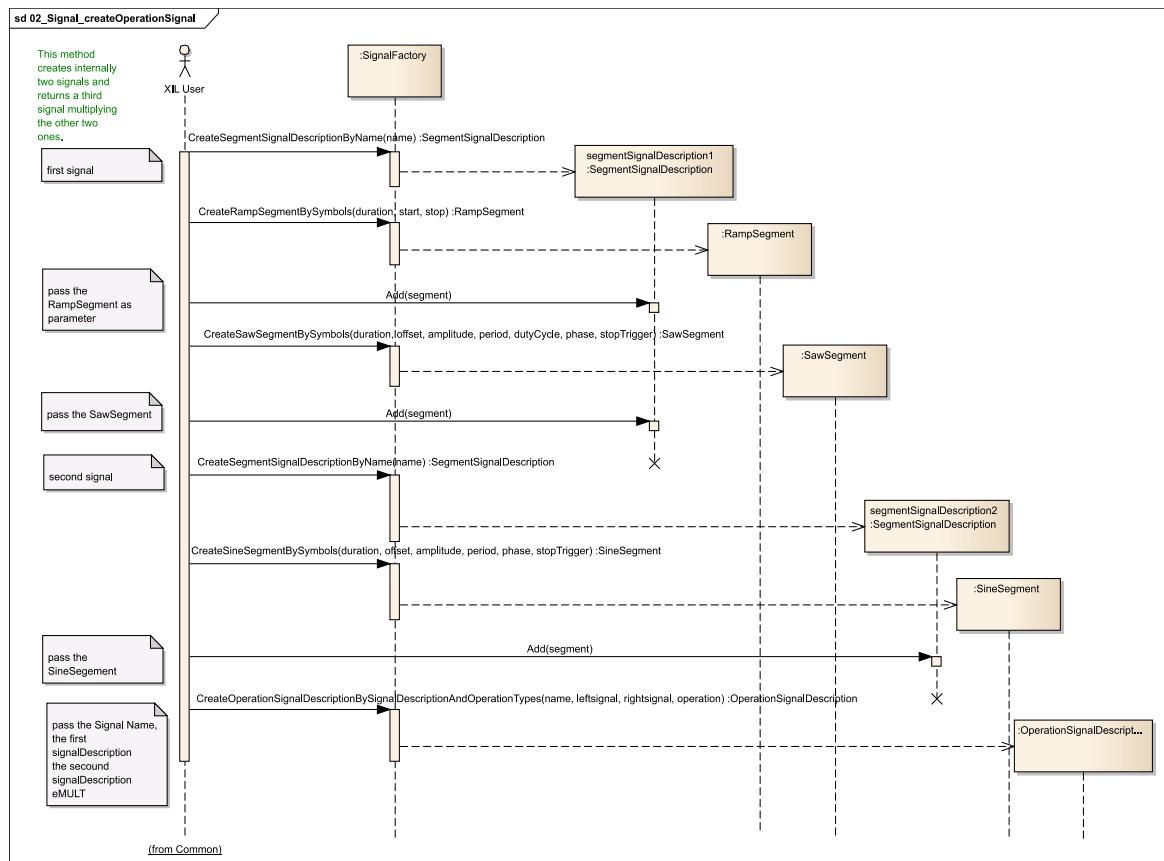


Figure 95: Create OperationSignal

Creating a wobble Signal

In this example (Figure 96) a periodic signal is created. The frequency property is described by a saw signal, so that the sine signal is wobbling.

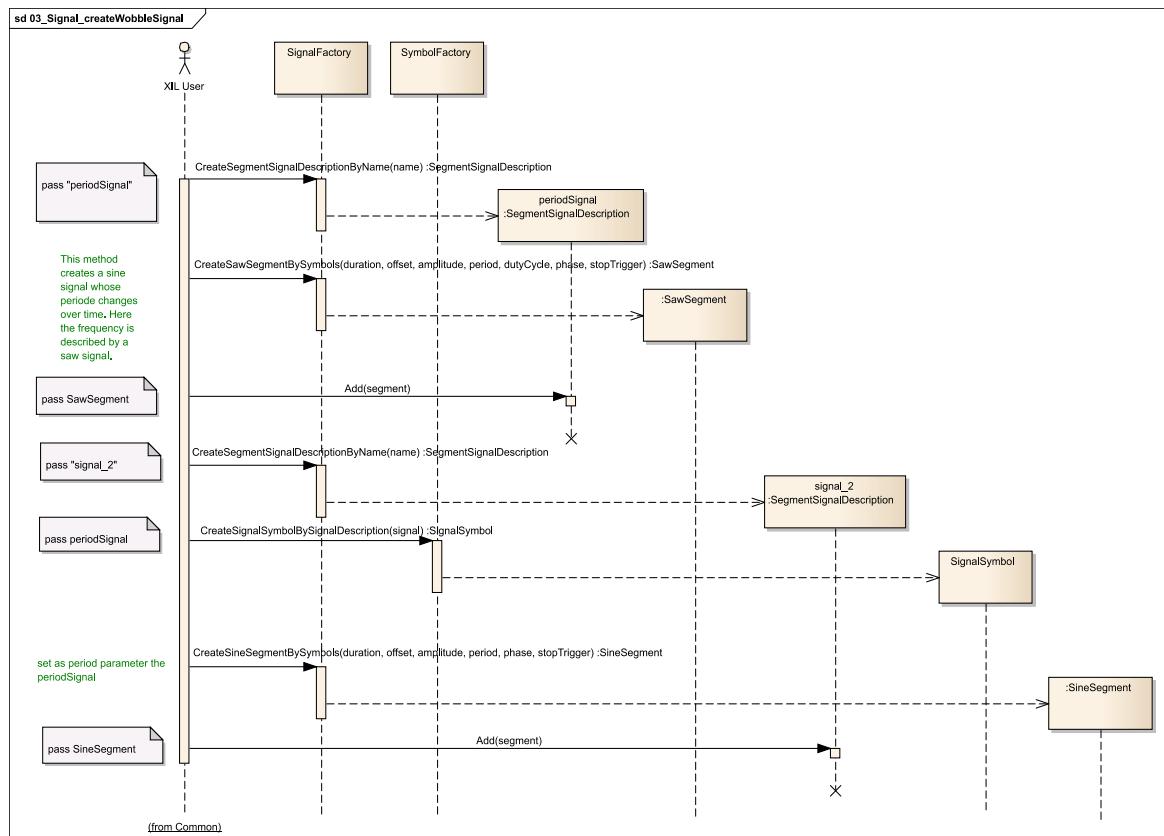


Figure 96: Create a wobbling signal

Creating a Signal Description Set

Figure 97 shows how a SignalDescriptionSet is created and how several SignalDescriptions (created as shown in the aforementioned figures) are added to it.

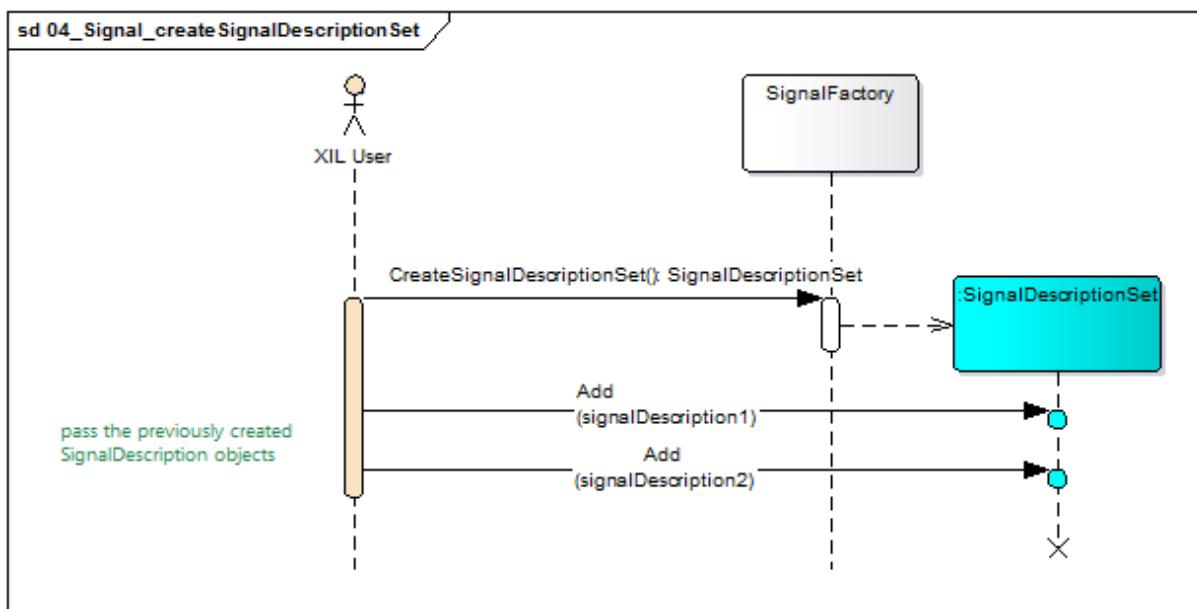


Figure 97: Create SignalDescriptionSet

Loading a Signal Description Set

The Figure 98 shows the alternative way to get a signal description set: via loading an existing set from a STI file. Then the set is queried for the names and the contained descriptions. Each of the descriptions is converted into a `SignalValue` object.

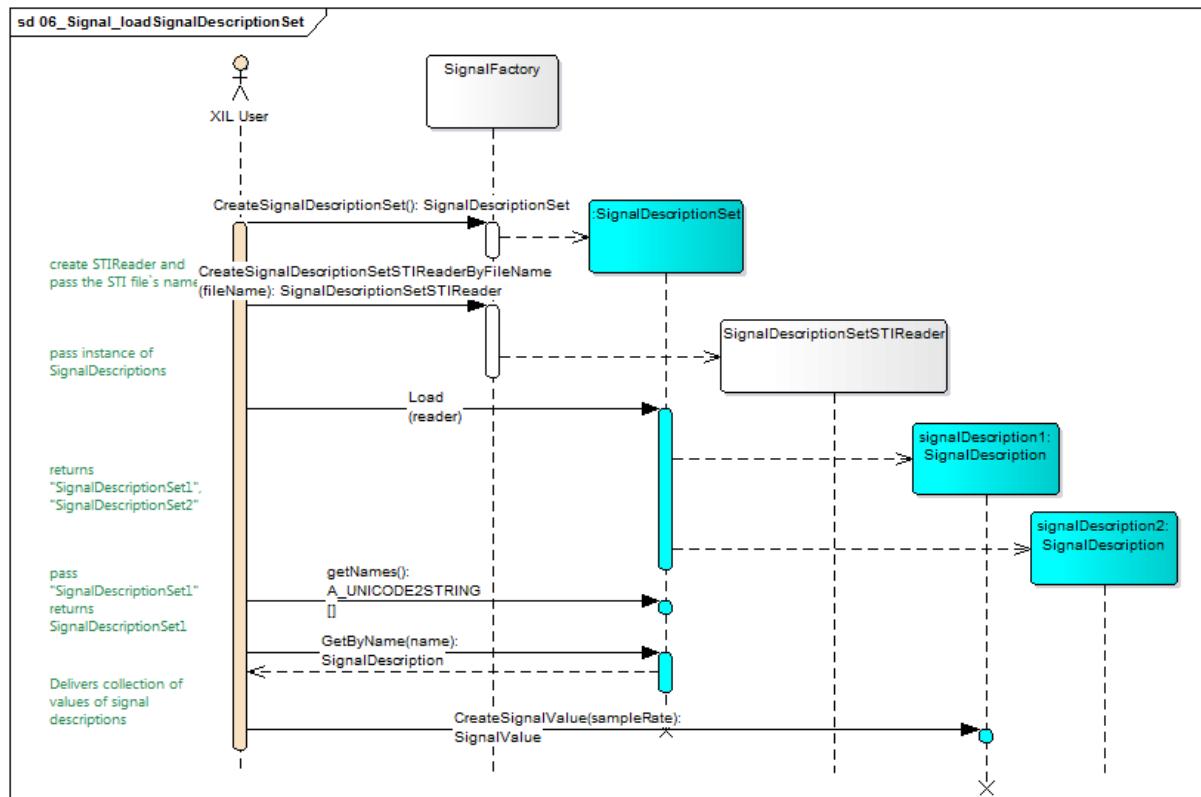


Figure 98: Load a `SignalDescriptionSet`

Saving a Signal Description Set

Figure 99 shows how to save a signal description set to a file for further reuse.

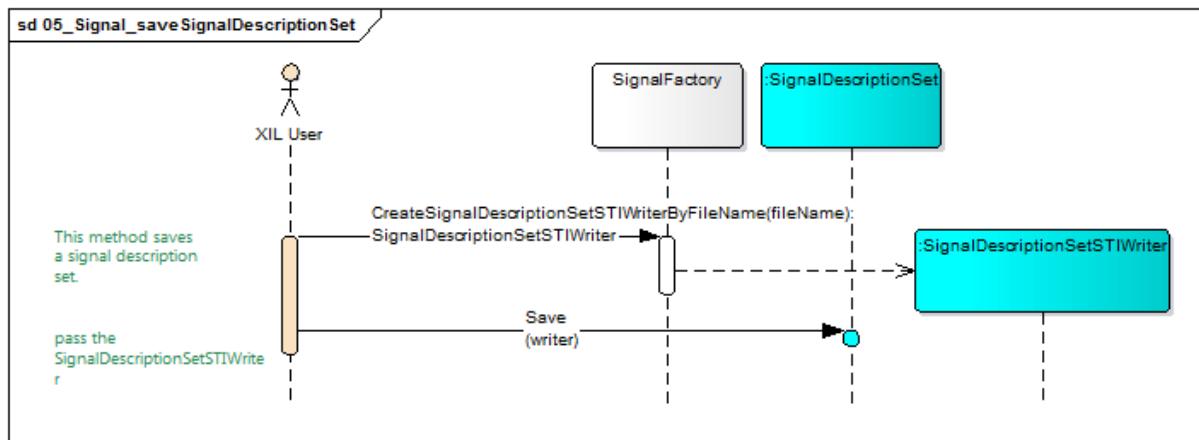


Figure 99: Save SignalDescriptionSet

5.1.5.5 SIGNAL DESCRIPTION FILE

The signal description file is used to serialize objects of type SignalDescriptionSet, and furthermore to serialize objects of type SignalGenerator.

The signal description file is an XML file with the file extension STI. The format of the STI file is defined via an XML schema definition file (see SignalDescriptionFormat.xsd).

All signals and segments are serialized in their corresponding XML tags. Due to performance issues the numerical values of the SignalValueSegment are serialized in a separate MATLAB file (.mat) and not in the XML file.

Each mat file contains two MATLAB arrays based on MATLAB data type double (64-Bit-IEEE-Floatingpoint). One array represents the time values and one array the signal values. Both arrays have the same length. Their dimension is (1 x N).

Each array is identified by its own name. The name of the time value array and the name of the signal value array inside a mat file are specified in the STI file.

Example:

```
MyTimeVector = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5];
MyDataVector = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0];
```

The MATLAB file use the file format Level 5 [4].

In addition it is possible to serialize the numerical data of multiple SignalValueSegments in one MATLAB file to get a better performance and to consume less memory. This is only possible, if the numerical data of the SignalValueSegments have the same time axis and the same length. E.g. the origin of the data was the same measurement and same raster. The MATLAB file will then contain one time vector and several data vectors which all are of the same length.

To achieve better data exchange by a STI file there is also the possibility to use a zip archive. Such a zip archive contains exactly one STI file and the eventually needed MATLAB files. The file extension of the zip archive is STZ.

The name of the zip-archive (STZ-file) and the name of the containing STI-file may be different.

5.1.5.6 USAGE OF PARAMETERIZED SIGNALDESCRIPTIONS

SignalDescriptions support expressions with placeholders to be assigned to SignalSegment parameters. This allows a very compact specification of a family of signals (characterized by the same SignalSegment sequence) with only one SignalDescription. The selection of a signal out of a family is realized by assignment of appropriate values to the placeholders and takes place just right before the stimulus start. This significantly increases reusability of SignalDescriptions.

A further benefit of using expressions with placeholders is the decoupling of clients from the signals' internal segment structure. Usually several parameters of a SignalDescription's segments are interrelated. So they must be changed in a consistent way to avoid distortions when adapting (e.g. scaling) a signal to specific needs. With expressions and placeholders properly applied a client does not need to know about the segment structure and the interrelations when using and adapting a predefined SignalDescription.

Figure 100 shows a SignalDescriptionSet with a step signal, which is a typical use case for a parameterized SignalDescription (with step height and offset being the parameters).

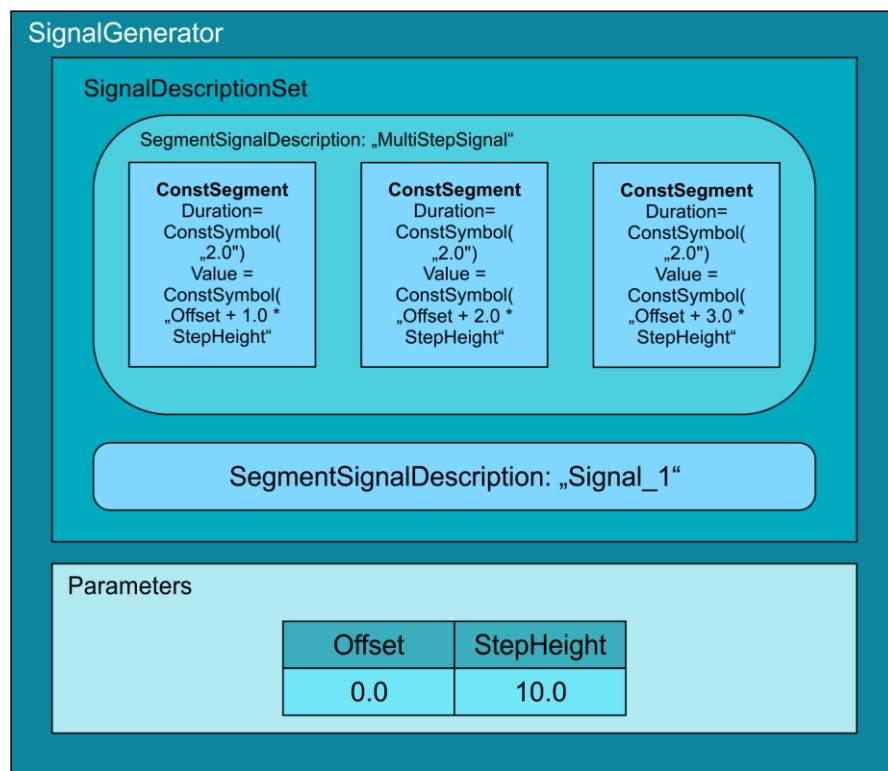


Figure 100: Usage of constants in SegmentSignalDescription

The principal approach to create a parameterized SignalDescription is illustrated in Figure 101, where the previously introduced step signal is used as example. It is assumed that an (empty) SignalDescriptionSet has already been created. So, first a new and empty SignalDescription for the MultiStepSignal must be created. Since the signal consists of 3 steps, it is necessary to create and add 3 properly configured ConstSegments to the SignalDescription.

The creation procedure is the same for all three steps. It is shown in detail only for the first step to keep the figure concise. For the same reason, the creation of `ConstSymbols`, that are assigned to the `ConstSegment` parameters, is only shown implicitly. First the `ConstSegment` is created and then its `Value` and `Duration` property are set. Pay attention to the expression “`Offset + 1.0 * StepHeight`” that is assigned to the `Value` property. It contains the placeholders “`Offset`” und “`StepHeight`” that refer to `SignalDescription` parameters defined later on. The expression “`2.0`” assigned to the `Duration` property does not contain any placeholders, hence specifying a fixed duration of 2 seconds. The syntax definition for the expressions permitted in `ConstSymbols` can be found in [Appendix B](#).

Finally parameter definitions for the used placeholders “`Offset`” und “`StepHeight`” are created and added to the `SignalDescriptionSet`. They are represented by instances of the `SignalDescriptionParameter` interface. There must be a `SignalDescriptionParameter` instance for each placeholder used in the `SignalDescriptions`. Otherwise the `SignalDescriptionSet` is inconsistent and cannot be executed.

When creating a parameter definition the parameter name and its default value must be specified. A parameter description can also be specified. These values determine the properties `Name`, `DefaultValue` and `Description` of the created `SignalDescriptionParameter` instance. Note that the parameter names must match the identifier syntax of ASAM GES [\[2\]](#).

Since `SignalDescriptionParameter` instances are a specialized form of `Script` parameters they also have the properties `Type` and `DataTypeInfo`. However, for `SignalDescriptionParameter` instances the value of these properties is predefined and cannot be set or changed. `Type` is always `eSTART_PARAMETER` and `DataTypeInfo` will always return `eSCALAR` as `ContainerType` and `eFLOAT` as `ElementType`.

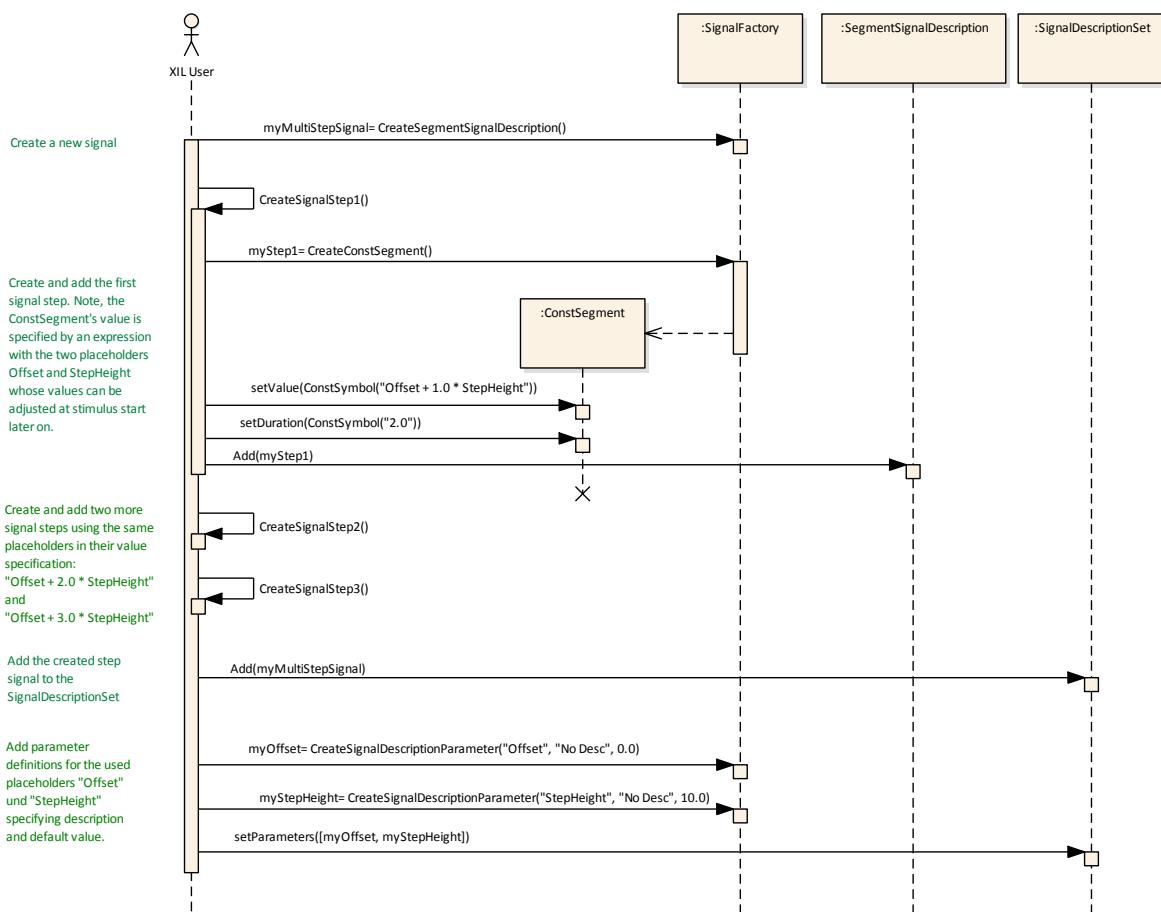


Figure 101: Signal createParameterizedSignalDescription

For an explanation on how to set the parameters and execute a parameterized `SignalDescription` please refer to [Figure 78](#) explaining the execution of TargetScripts which analogously applies to `SignalGenerators` except for the retrieval of return values (`SignalGenerators` do not have output parameters).

5.1.6 SIGNALGENERATOR

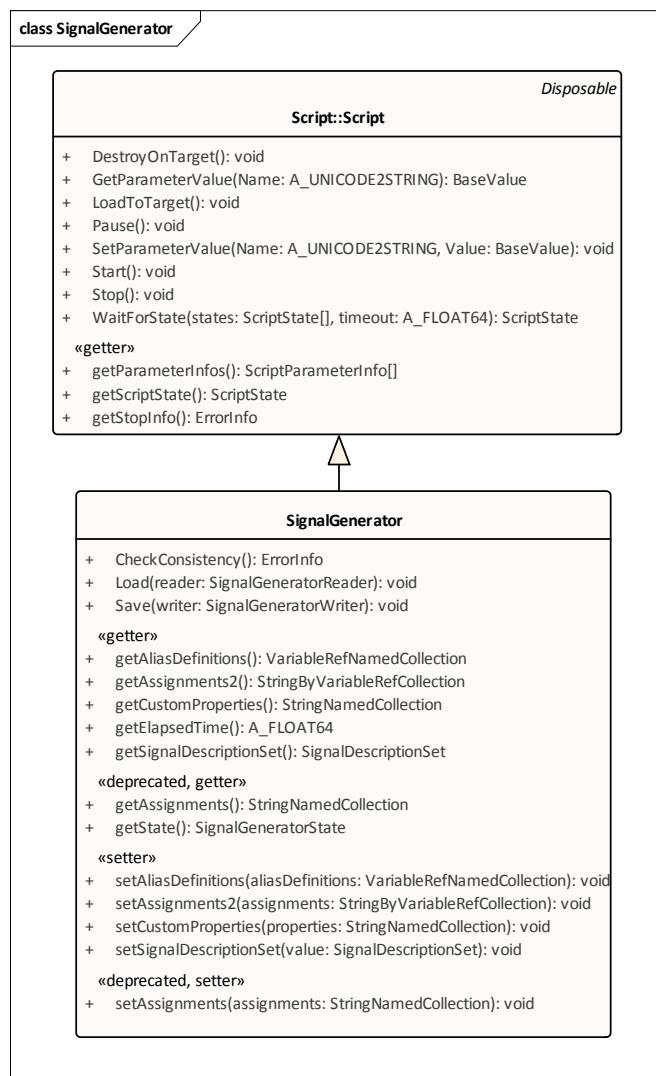


Figure 102: Signal Generator

A `SignalGenerator` defines stimuli and manages their execution. For the definition of a stimulus, a `SignalDescriptionSet` is referenced by the `SignalGenerator`. It also specifies the variables to be stimulated and assigns a `SignalDescription` from the `SignalDescriptionSet` to each of them. For the management of the stimulus, functionality is provided for downloading the stimulus to the target system, for starting, stopping, and pausing it and for observing its current state.

For state dependencies of the `SignalGenerator`'s methods refer to chapter 5.1.3.2. Note that state and state changes of the Port the `SignalGenerator` is connected to may have an effect on the stimulation by the `SignalGenerator`. Dependencies on the `MAPort` state are documented in chapter 5.2.2.5.

Please note, that also a variable's writeability has influence on the stimulation. The variable's writeability can be obtained using the `IsWriteable()` Method of the `VariableInfo` object (e. g. `MAPortVariableInfo` for the `MAPort`).

5.1.6.1 INTERPRETATION OF SIGNALDESCRIPTIONS BY THE SIGNALGENERATOR

The function of the SignalGenerator is the continuous, time-accurate tracking of variable values with respect to references in the form of SignalDescriptions. The time raster (cycle time) with which these value adjustments are performed in time-discrete systems is system-dependent and not predefined or restricted by XIL. In particular, an individual stimulation raster per variable and non-equidistant rasters are allowed. Furthermore SignalDescriptions that differ in length are permitted in the same SignalDescriptionSet. However, in order to ensure uniform behavior of stimulus files across different systems, SignalGenerator implementations must always meet the following conditions (Figure 103 to Figure 106 illustrate these conditions with examples):

- **C1:** The stimulation of all variables of a SignalGenerator refers to the same common time base represented by the generator's ElapsedTime property. Point zero of this stimulation time is defined by the initial writing of stimulation values. This also applies if this first stimulation step does not write all of the stimulated variables due to individual rasters. Leading IDLESegments are treated as non-IDLESegments when determining point zero of the stimulation time.
- **C2:** If a variable's first stimulation step takes place at a stimulation time $t_0 > 0$ (due to individual rasters), then the first written value is not the SignalDescription's initial value, but the value at position t_0 .

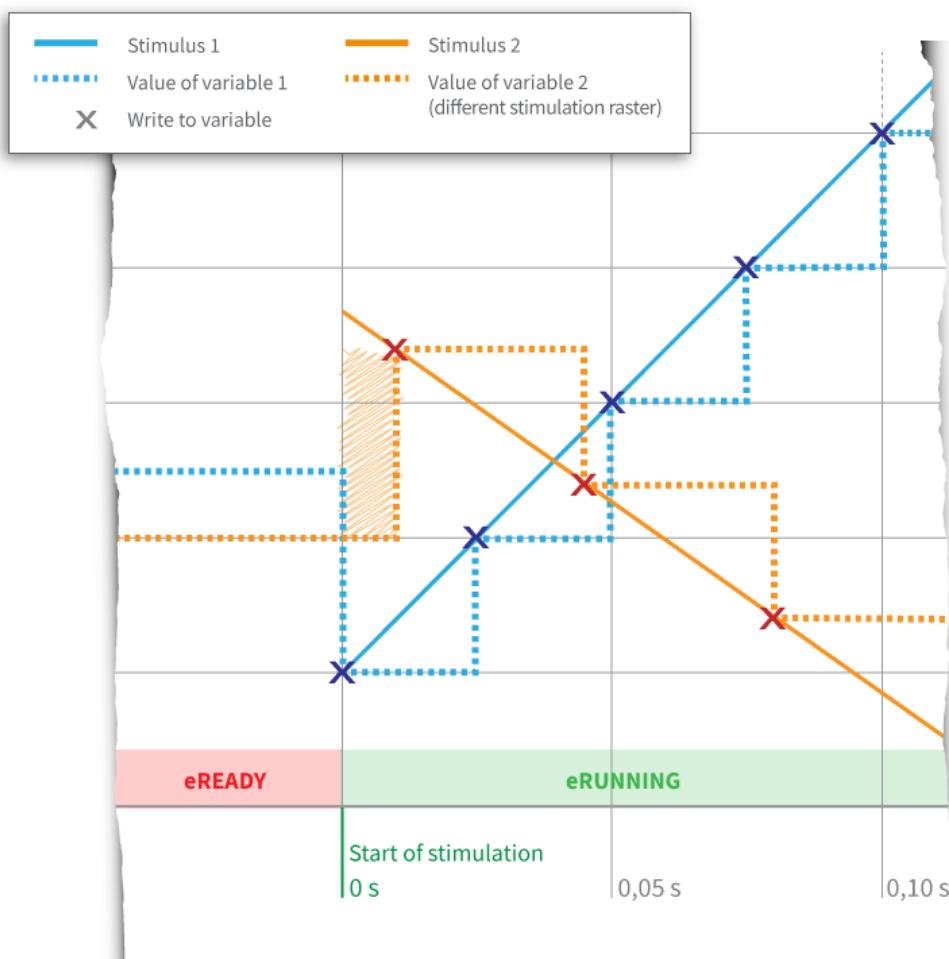


Figure 103: Initial stimulation values and zero point of elapsed stimulation time when stimulating two variables with equidistant but different stimulation rasters

- **C3:** If a stimulation step exactly coincides with the boundary of two successive SignalSegments, then the start value of the successor segment is used as stimulation value. This is particularly important for discontinuous segment transitions (i.e. successor segment's start value differs from predecessor segment's end value) or if the successor segment is an IDLESegment.

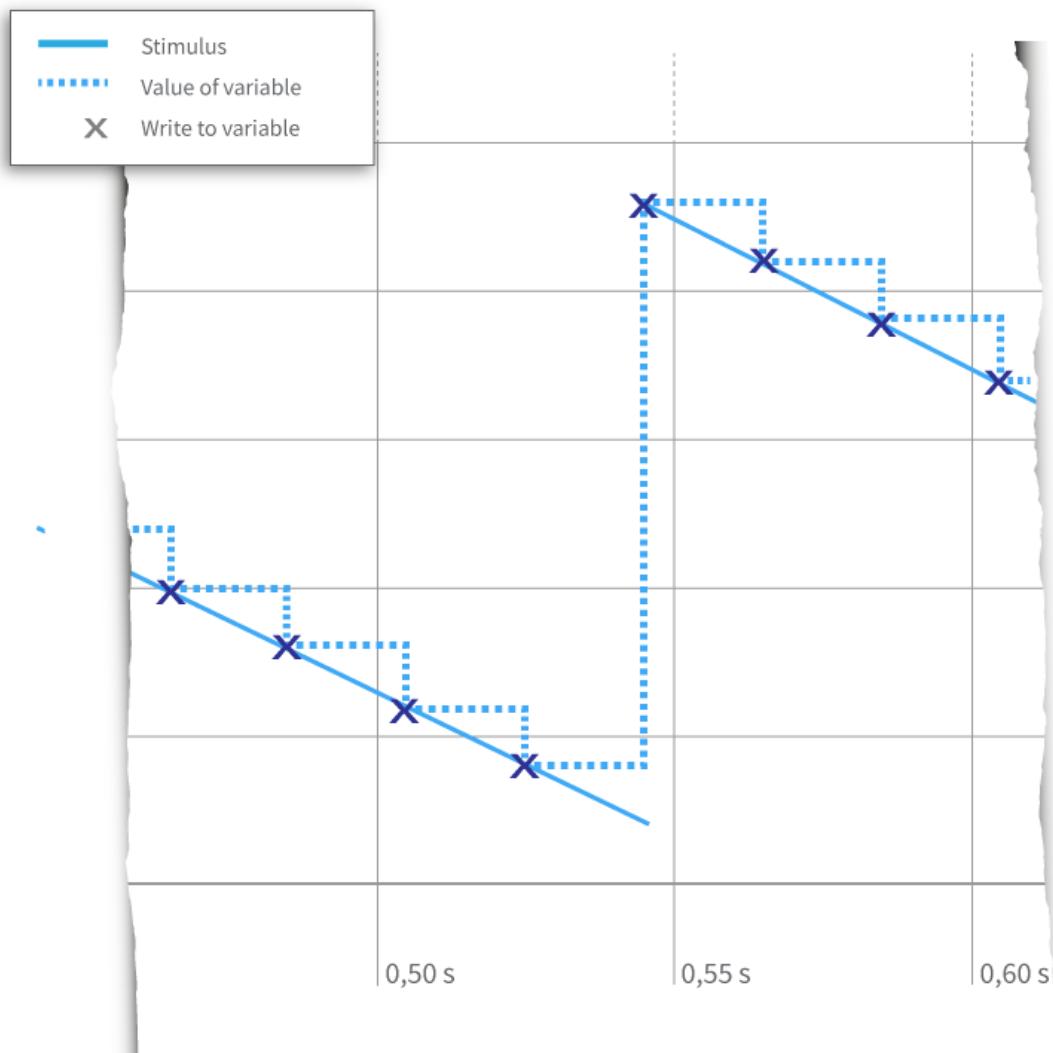


Figure 104: Course of stimulation at discontinuous segment transition (with equidistant stimulation raster)

- **C4:** If the stimulation is controlled by an IDLESegment, the value of the stimulated variable remains unaffected by the SignalGenerator. Instead, manipulations from outside the SignalGenerator, if any, will take effect and possibly change the variable value. In absence of such manipulations, the last written variable value remains unchanged until the end or even beyond the IDLESegment, hence showing a constant signal course.

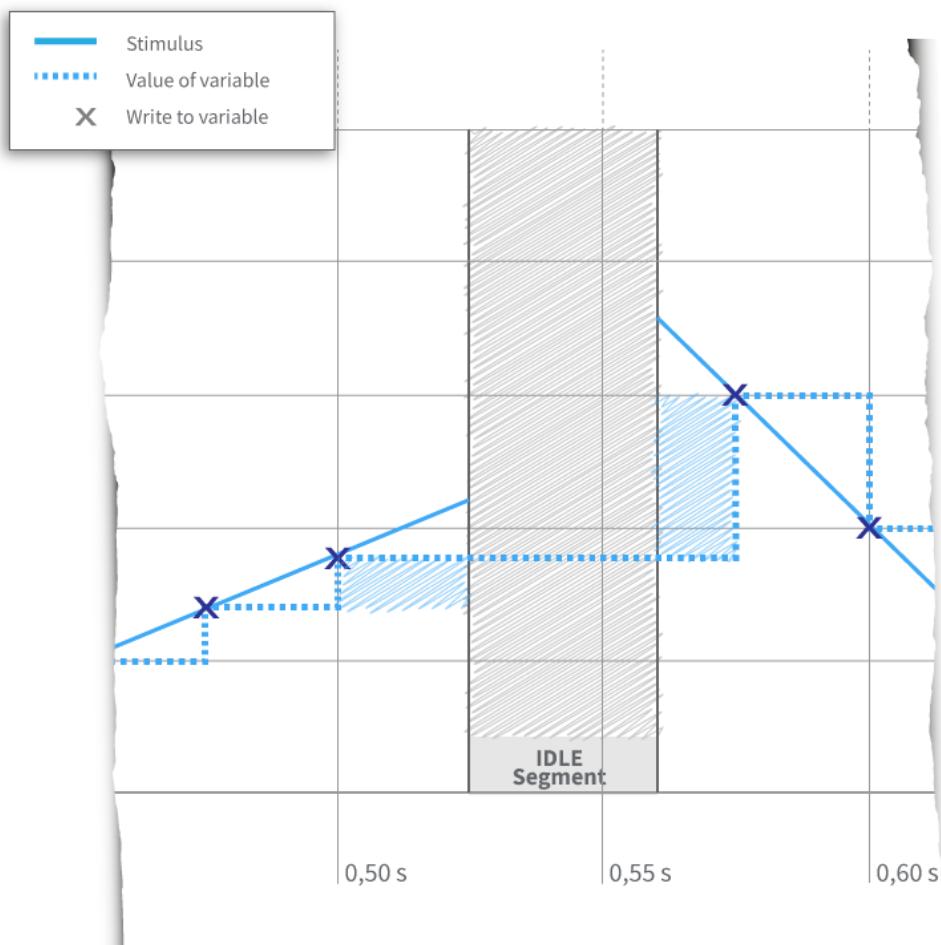


Figure 105: Course of stimulation during IDLESegment in absence of additional manipulations from outside the SignalGenerator and with equidistant stimulation raster used

- **C5:** Stimulation end is determined for each variable individually, since dependent on SignalDescription length and stimulation raster. It is determined by the following two conditions:
 - The last stimulation step is before the endpoint of the SignalDescription.
 - The stimulation step that would follow the last stimulation step if stimulation had not ended would be on or after the SignalDescription's endpoint.
- **C6:** Variables whose stimulation has ended while the stimulation of other variables is still in progress retain their last stimulation value, provided that no other manipulation from outside the SignalGenerator takes place.
- **C7:** The last time stimulation values are written, i.e. the stimulation step after which stimulation is completed for all variables, determines the point in time when the SignalGenerator switches to status eFINISHED.
- **C8:** In state eFINISHED all variables retain their last stimulation value, provided that no manipulations from outside the SignalGenerator take place. The same applies to state eSTOPPED. However, in the latter case, the last stimulation value is the last written value before the stimulation was aborted.

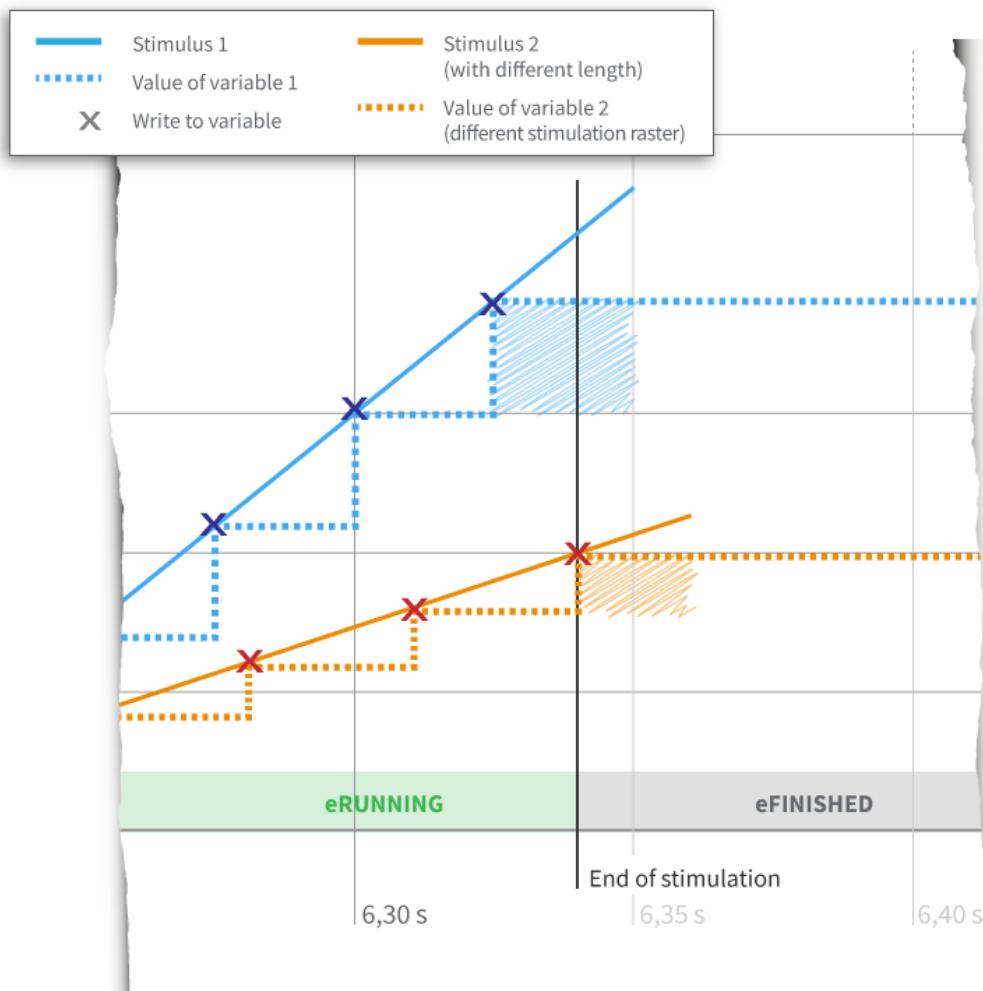


Figure 106: End of stimulation with final stimulation values and their persistence in absence of other signal manipulations (Equidistant but different stimulation rasters are used. SignalDescriptions have different lengths.)

- **C9:** Upon state change of the `SignalGenerator` to `eIN_CONFIGURATION` (i.e. call of method `Script:DestroyOnTarget`) all variables representing dynamic variables of the stimulated system (i.e. variables whose values dynamically change due to system internal processes) return to their dynamic behavior. This only applies if no further stimulation or manipulation is effective.

Note: The conditions above define several time periods in which the `SignalGenerator` seems to keep variables constant at certain values. This concerns the `IDLESegment`, the time after the last stimulation step while still in state `eRUNNING` and the status `eFINISHED`. However, this “holding” of values is only seemingly, for the following reasons:

- Actually, the `SignalGenerator` does not write values during the mentioned time periods. So the variable values might change due to manipulations from outside the `SignalGenerator`. Such external manipulations are write operations by the client, user interactions as well as stimulations by other `SignalGenerator` instances.
- Actually, there is no value specification by the `SignalDescription` giving a certain value at which the variable is to be held. The seemingly held value is

just the result of the last stimulation step. This value depends on the stimulation raster and does not necessarily correspond to the final value of the last active `SignalSegment` that would result from its calculation formula and parameterization.

If it is to be ensured that a variable has a certain value and is kept constant on that value for a period of time, this period must not be specified as an `IDLESegment`, but as a `ConstSegment`. Furthermore if it is to be ensured that a variable has a certain value when its stimulation ends, then there must be a corresponding `ConstSegment` at the end of the `SignalDescription`.

5.1.6.2 PARAMETERS

`SignalGenerator` does not have output parameters. The input parameters are restricted to parameter type `eSTART_PARAMETER` and to container type `eSCALAR` with element type `eFLOAT`. The parameter values, that can be read and set by the methods `GetParameterValue` and `SetParameterValue`, are initialized with their default values every time the `SignalGenerator` is loaded with its `Load` method.

Available parameters, their default values as well as optional descriptions are determined by user-defined parameter definitions. These parameter definitions are part of the `SignalDescriptionSet` and stored in the signal description file (STI/STZ). See [5.1.5.6](#) for an example illustrating the creation of parameterized `SignalDescriptions` and associated parameter definitions.

5.1.6.3 CUSTOM PROPERTIES

`SignalGenerator` extends the base interface `Script` by custom properties which is a generic way to configure vendor specific settings. Use the property `CustomProperties` for this purpose.

Note: Custom properties are always optional, i.e. other methods (`LoadToTarget`, `Start` etc.) do not require these properties to be set by the client. If the vendor's implementation requires a custom property that is not set by the client it is up to the implementation to assign a feasible value (although the system might not work with optimum performance in this case).

Invalid custom properties or values set by the client are signalled by the methods that make use of these properties (`LoadToTarget`, `Start` etc.) but not by the property `CustomProperties`.

5.1.6.4 USAGE OF SIGNALGENERATOR (STIMULATING MODEL VARIABLES)

The following explanations and figures give the basic aspects of using the `SignalGenerator` interface. Stimulation of simulation model variables is used as example.

Let us assume that a configured `MAPort` instance and a `SignalGeneratorFactory`, which can be obtained from the `Testbench`, are available. The first step, as depicted in [Figure 107](#), is the creation of a `SignalGenerator` instance. Then the stimulus content (especially the `SignalDescriptionSet`) is loaded from a STI file into the `SignalGenerator` using a `SignalGeneratorSTIReader`. After executing the `Load()` method, signal descriptions are referenced by the `SignalGenerator` via a `SignalDescriptionSet` object.

If the loaded STI file contains additionally information that assigns the contained SignalDescriptions to variables of the target system, the configuration might be complete at this point. Otherwise (as shown in the figure) these assignments have to be specified by adding key-value pairs to the Assignments collection of the SignalGenerator object. Keys in this collection are references to the target system variables to be stimulated. These references are represented by VariableRef objects that are created with the VariableRefFactory (not shown in detail in the figure). The values in the Assignments collection are the names of the SignalDescriptions to be used as stimulus.

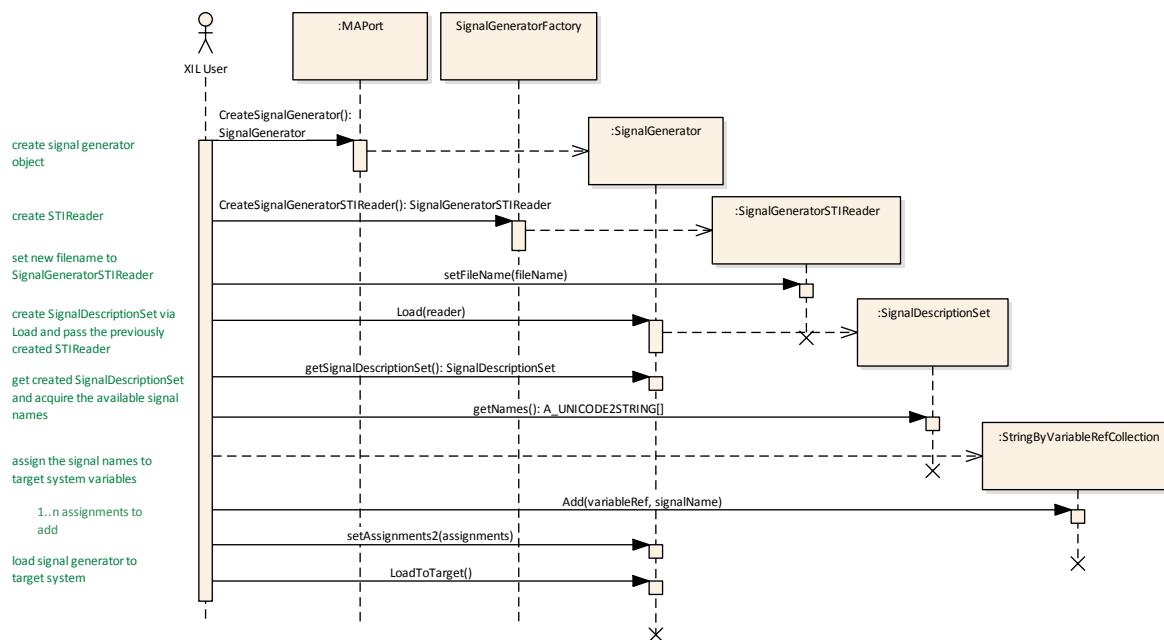


Figure 107: SignalGenerator example (part 1)

If the SignalDescriptions use StringSymbols, their alias names must be assigned to variables in the target system. If this assignment is not contained in the STI file or should be changed, another configuration step is necessary which is not shown in the figure. This configuration step sets or changes the AliasDefinition collection of the SignalGenerator. The collection's keys are the alias names used in the StringSymbols and the values are references to the assigned system variables. These references are represented by VariableRef objects and have the same restrictions as in the context of the Assignments collection.

After setting the assignments and alias definitions, the stimulus is configured. The next step is downloading the stimulus to the target system (e.g. the XIL simulator). Then, the stimulus can be started, paused, and stopped by calling the corresponding methods. This is shown in [Figure 108](#). This also shows how to retrieve status information like execution state or elapsed stimulation time. Finally, the SignalGenerator object can be saved including the new assignments, using a SignalGeneratorSTIWriter.

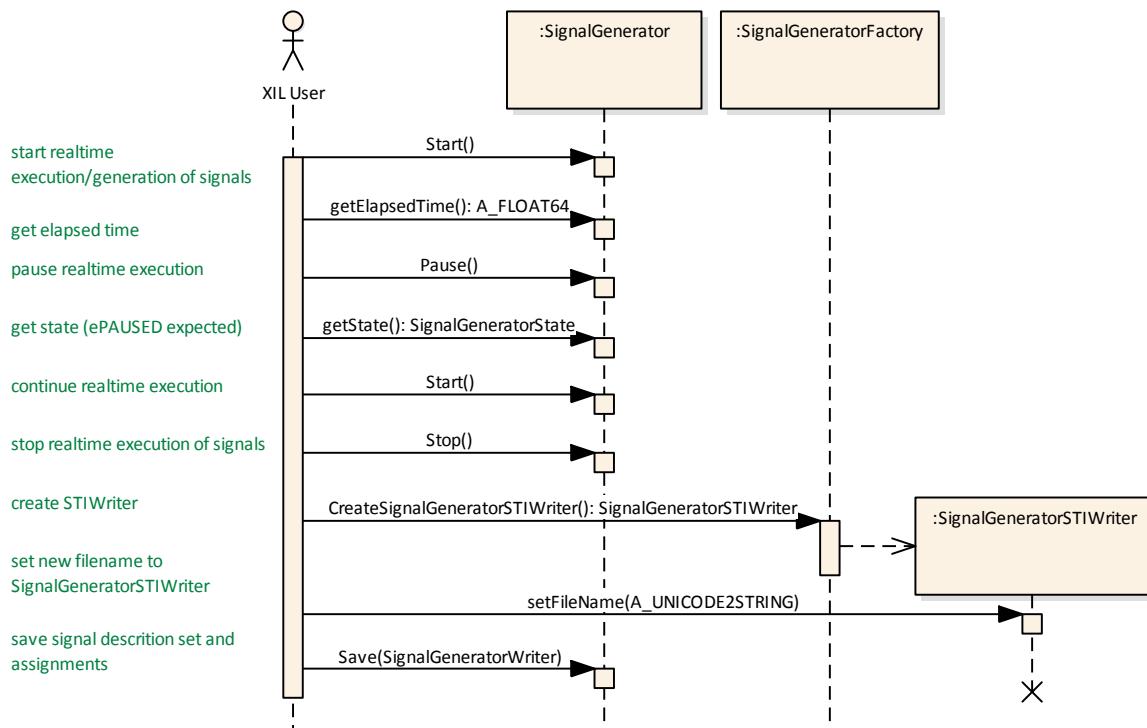


Figure 108: SignalGenerator example (part 2)

Note: Because of their nature `SignalDescriptions` can only be assigned to scalar variables or single elements of a vector or matrix variable. That means the `VariableRef` objects used as keys in the `Assignments` collection must either be `GenericVariableRefs` pointing to scalar variables, `VectorElementRefs` or `MatrixElementRefs`. `GenericVariableRefs` pointing to non-scalar variables are not allowed. The same restriction applies to the `AliasDefinitions` collection.

Note: The numerical stimulus values derived from the `SignalDescriptions` can be used to manipulate the physical value as well as the raw value of the stimulated system variable. The used representation mode is determined by the `ValueRepresentation` property of the respective `VariableRef` object in the `Assignments` collection. Raw value stimulation can be a possibility to stimulate system variables whose physical value is not of a numeric type, e.g. variables with a `TexttableCompuMethod` (enumeration variables).

Note: The `ValueRepresentation` property of the `VariableRef` objects in the `AliasDefinitions` collection determines whether the alias refers to the physical or the raw value of the assigned system variable.

5.1.7 DOCUMENT HANDLING FOR SIGNALGENERATOR AND SIGNALDESCRIPTIONSET

To save the whole content of a `SignalDescriptionSet` or to load a complete set of signals into a `SignalDescriptionSet` there are two interfaces: The `SignalDescriptionSetWriter` and the `SignalDescriptionSetReader` (see [Figure 109](#)). Similarly, the `SignalGeneratorWriter` and `SignalGeneratorReader` interfaces allow to save and load the content of a `SignalGenerator` (see [Figure 110](#)). This concept allows to load and save data in

different formats. For STI and STZ (signal description file formats of XIL, see [5.1.5.5](#)) the standard defines a set of specialized reader and writer interfaces.

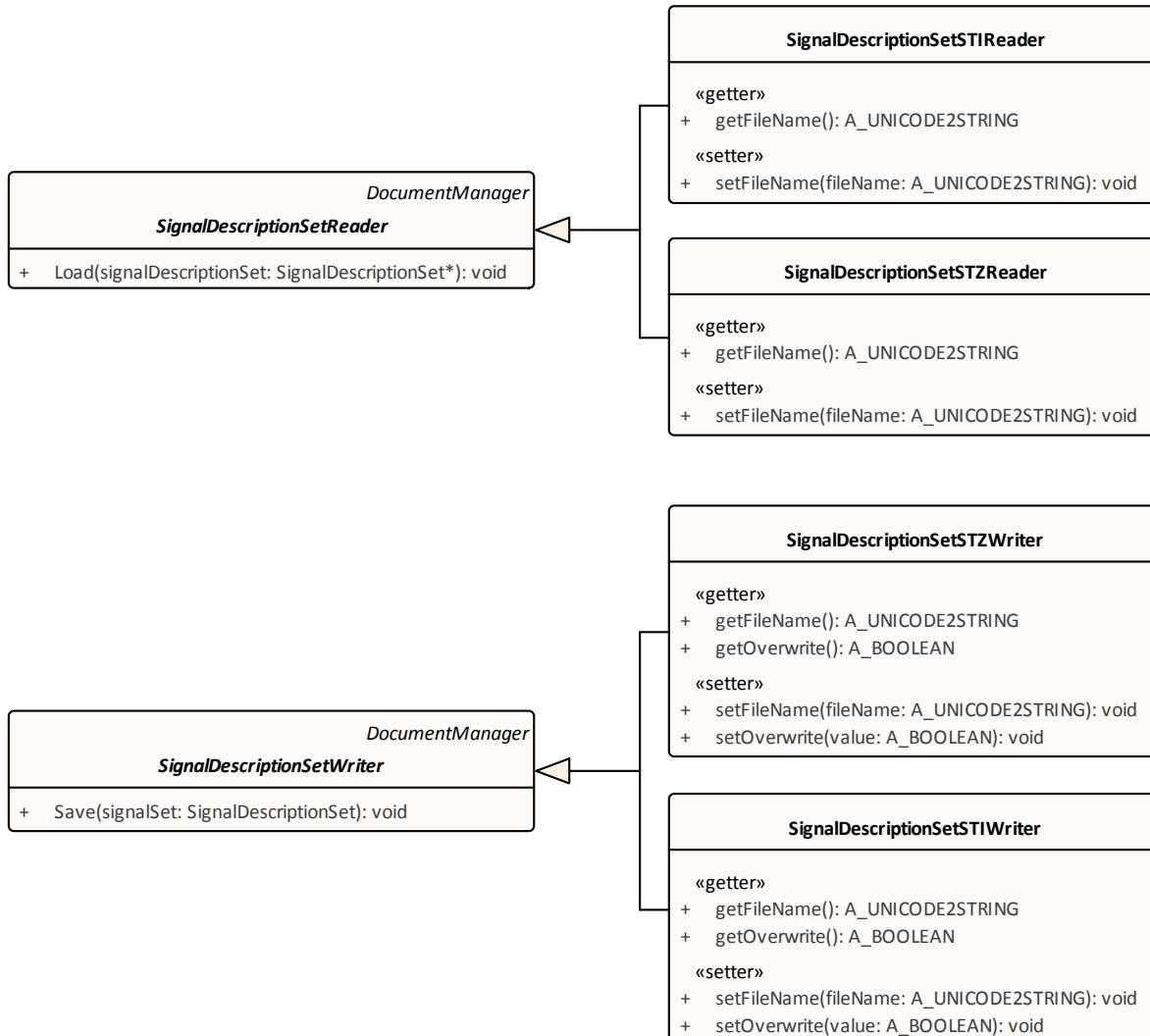


Figure 109: Signal Description Document Handling

SignalDescriptionSetReader & SignalDescriptionSetWriter

These classes are abstract super classes for the concrete reader and writer classes. These classes provide a Load or a Save method resp. to load and to save SignalDescriptionSet objects.

SignalDescriptionSetSTIReader/ SignalDescriptionSetSTZReader

This class handles the loading of a SignalDescriptionSet object stored in a STI/STZ file. The loaded data structure is stored in a SignalDescriptionSet object.

SignalDescriptionSetSTIWriter/ SignalDescriptionSetSTZWriter

This class handles the saving of SignalDescriptionSet objects in STI/STZ format.

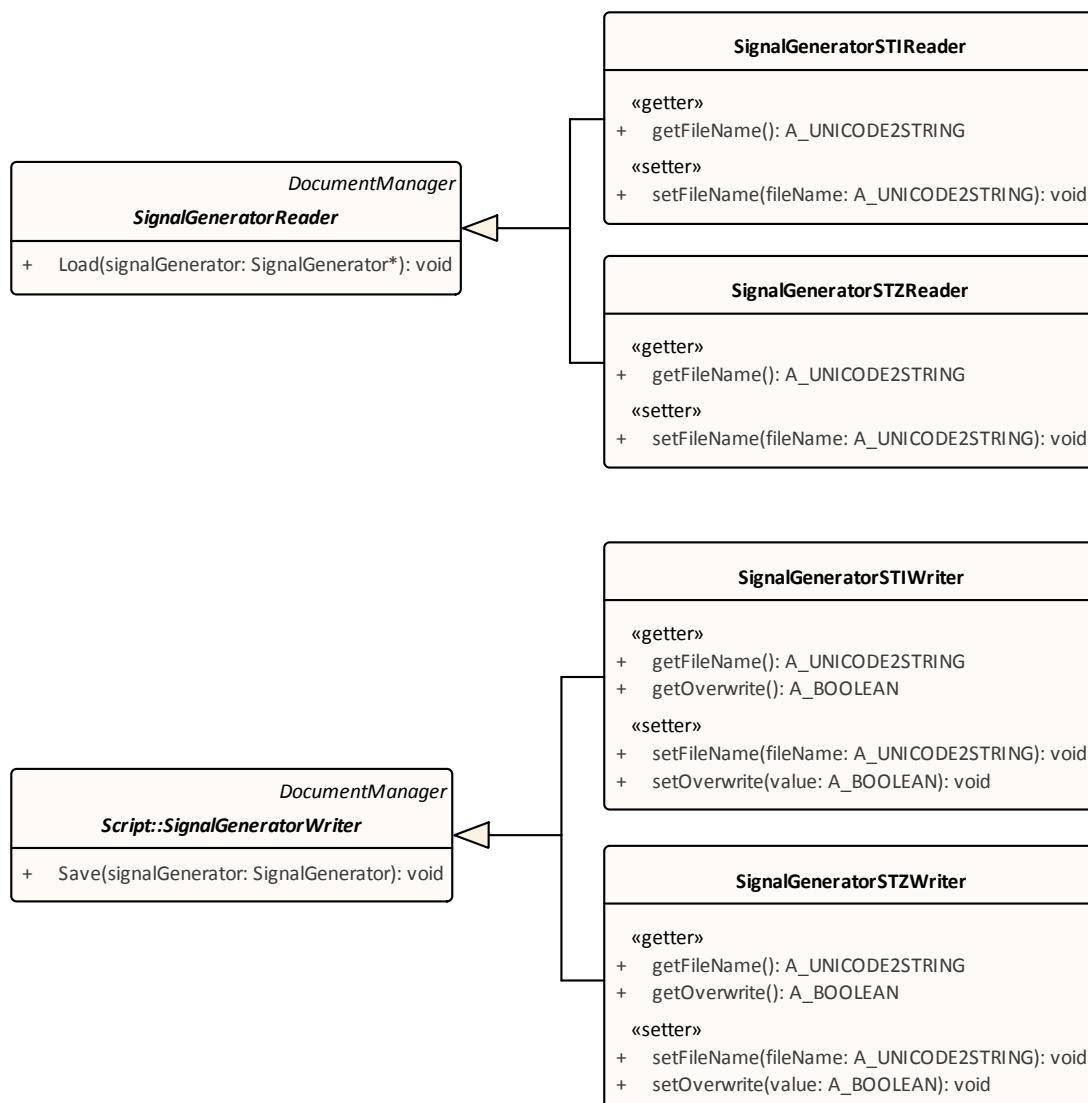


Figure 110: Signal Generator Document Handling

SignalGeneratorReader & SignalGeneratorWriter

These classes are abstract super classes for the concrete reader and writer classes. These classes provide a Load or a Save method resp. to load and to save SignalGenerator objects.

SignalGeneratorSTIReader/ SignalGeneratorSTZReader

This class handles the loading of a SignalGenerator object stored in a STI/ STZ file. The loaded data structure is stored in a SignalGenerator object.

SignalGeneratorSTIWriter/ SignalGeneratorSTZWriter

This class handles the saving of SignalGenerator objects in STI/ STZ format.

5.1.8 WATCHER

5.1.8.1 GENERAL

The Watcher is designed as a generic event generator. It can be used e.g. for the trigger definition of captures.

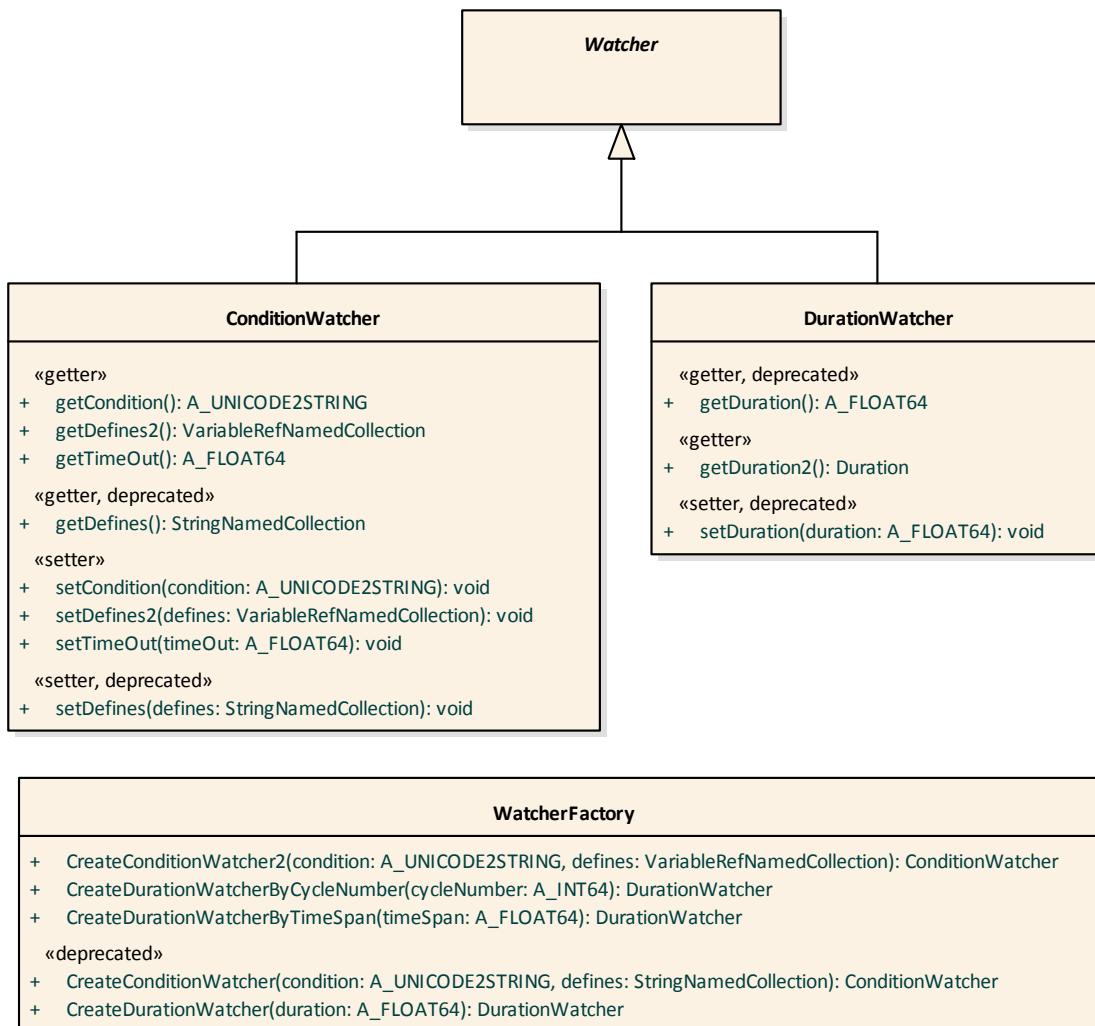


Figure 111: Testbench Watcher

XIL distinguishes two type of watcher:

1. Watchers that trigger if a certain condition becomes true (e.g. if a simulation variable takes on a specific value)
2. Watchers that trigger after a certain period of time

The first type of watchers is represented by the `ConditionWatcher` interface, the second type by the `DurationWatcher` interface.

DurationWatcher

The DurationWatcher triggers after a specified duration relative to the time of the watcher activation. For details on the behavior of the DurationWatcher in the context of a capture see section [Capture Configuration and Control](#).

ConditionWatcher

The ConditionWatcher triggers as soon as the specified condition becomes true after its activation. The condition is specified by an expression string that is stored in the property Condition. The expression syntax is defined in [Syntax of Watcher Conditions](#). The syntax of the condition is validated when the property Condition is set.

In the ConditionWatcher's condition aliases are used as placeholders for variables of the target system (e.g. signals or parameter of a simulation model). This keeps the conditions concise and human readable. It also decouples test cases from the technical names of the target system variables (e.g. pathnames within simulation models).

For the activation of a ConditionWatcher its aliases must be assigned to the system variables to be used. This assignment is defined by the ConditionWatcher's Defines collection. The aliases are the keys in the Defines collection. And the values are the assigned references to target system variables. These references are represented by VariableRef objects that are created using the VariableRefFactory (see section [Variable Ref and Value Representation Mode](#) for details). All aliases used in the condition expression must be contained in the Defines collection.

Note: Aliases must always be assigned to scalar variables or scalar elements of non-scalar variables. Hence, only GenericVariableRefs pointing to scalar variables, VectorElementRefs and MatrixElementRefs are allowed as values in the Defines collection.

Note: There are some method calls that block until a certain watcher condition becomes true. For example the Capture.Fetch method with the parameter whenFinished := TRUE blocks until the watcher used as stop trigger insists on a watcher condition to become true. In case of a ConditionWatcher this could cause the client to block endlessly if the specified condition does never become true. To prevent this case, the property TimeOut of the ConditionWatcher should be set. This ensures that the ConditionWatcher stops checking its condition and triggers its event as soon as the duration specified by TimeOut has expired. If a timeout value of zero is given, the ConditionWatcher fires its event immediately and the specified condition is ignored. The value -1.0 specifies an infinite timeout.

5.1.9 DURATION

Durations are a concept in XIL to specify the length of time that a process is to be continued or until an event is triggered. XIL allows you to specify durations as number of seconds or based on a time scale that is target system specific, e.g. as number of simulation steps or capture samples etc. Accordingly there are two interfaces (see [Figure 112](#)). For durations in seconds the TimespanDuration interface is used. The CycleNumberDuration interface represents durations on a system dependent time scale. Their meaning is determined by the

usage context. Instances for both interfaces are created via the `DurationFactory` that can be obtained from the `Testbench`.

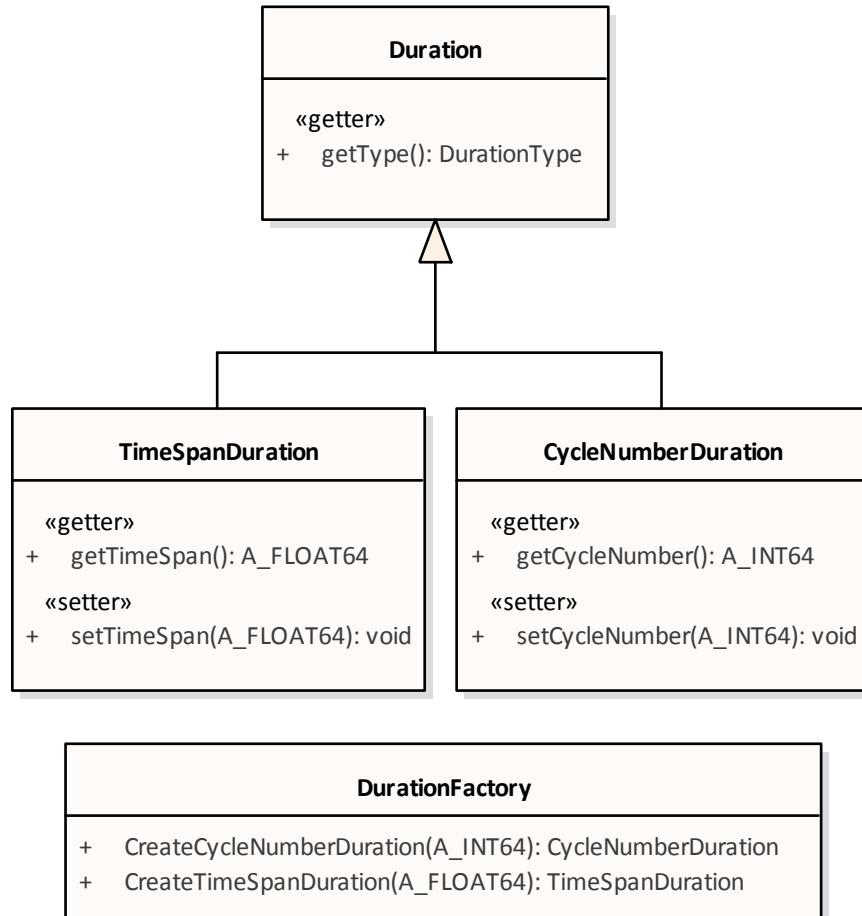


Figure 112: Duration Interfaces

Mostly `Duration` objects are used indirectly as property of `DurationWatcher` objects where they specify the period of time until the watcher triggers. Examples are the stop trigger of `Capture` objects (see also [Triggered Capturing](#)), the stop trigger of `SignalSegments` in `SignalDescriptions` and the `MAPortBreakpoint` (see also [Pausing and stepwise execution of the simulation](#)). The specification of pre trigger interval and post trigger delay for the start and stop trigger in `Capture` objects uses `Duration` objects directly.

5.1.10 META INFO

For many objects such as `Ports`, `Scripts` and `CaptureResults` clients need access to descriptive information about the contained or controlled items. The `Metalinfo` package provides interfaces for this purpose, that are used across the `Testbench API`.

5.1.10.1 METADATA ON VARIABLES

[Figure 113.](#) shows the collection of interfaces used to query meta information on the target system variables accessible through the `Port` objects. Several of these interfaces are also

used for metadata retrieval on variable traces contained in `CaptureResults` or on parameters of `Script` instances.

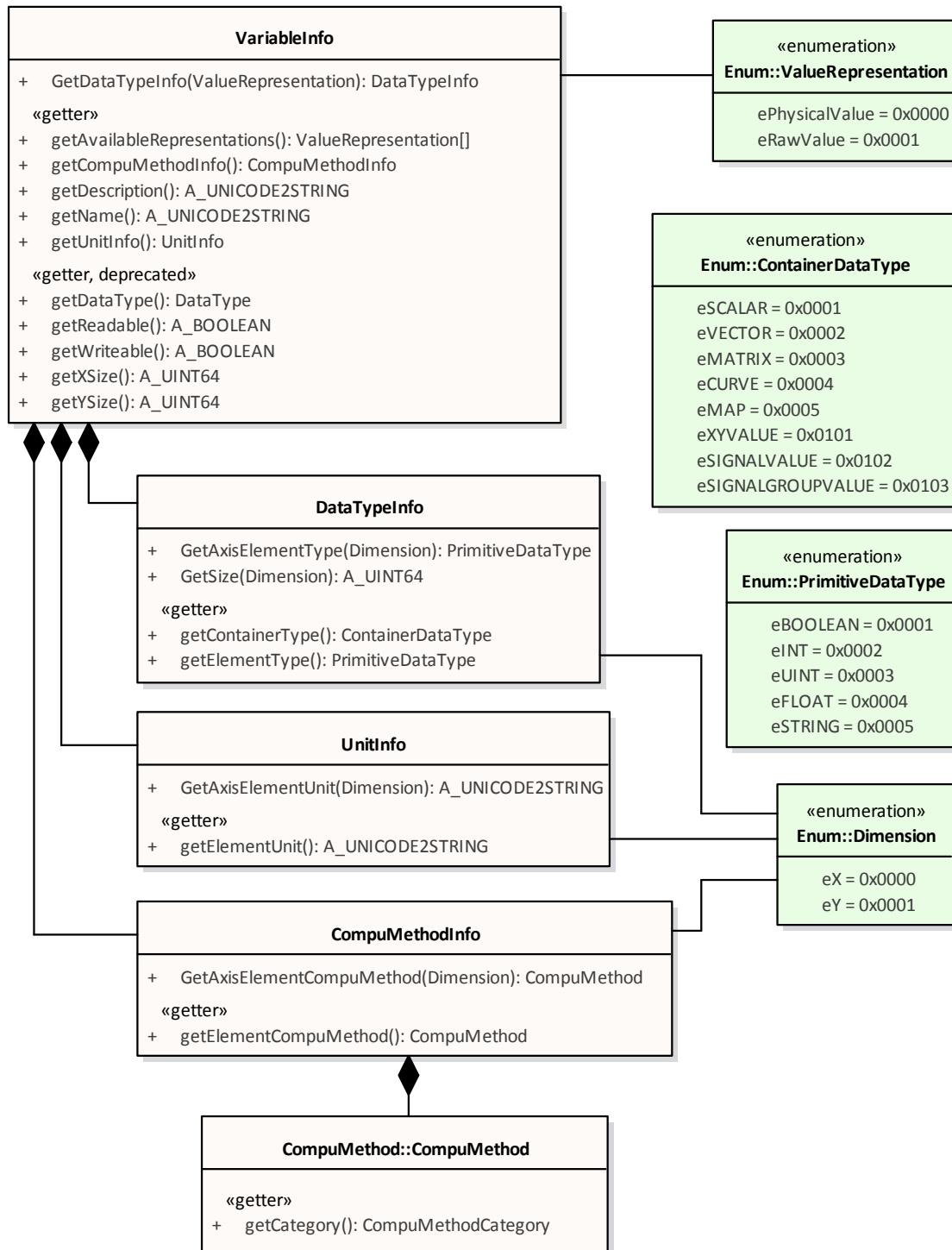


Figure 113: Interfaces for metadata query on variables

The `VariableInfo` interface is the root interface for metadata query on a variable. It provides general information like variable name and description. For the query of unit, data

type and conversion method separate interfaces exist. This segregation enables their independent reuse in other parts of the API (e.g. in `ScriptParameterInfo`). Instances are provided via appropriate named properties or methods of the `VariableInfo` object. The following, indented paragraphs give a short description of these interfaces.

[DataTypeInfo interface](#)

Provides information on the container and element type of a variable's values. For array variables (e.g. vector and matrix variables) their size can be retrieved. In case of variable types with axes (e.g. curve and map variables) also the element type of the axes can be queried.

The data type of the variable values depends on the used representation mode and may differ for the various modes. Hence, there is an individual `DataTypeInfo` object for each available representation mode. Use the property `AvailableRepresentations` to determine the representation modes for which a `DataTypeInfo` can be retrieved. A general explanation of representation modes can be found in section [Variable Ref and Value Representation Mode](#).

[UnitInfo interface](#)

Yields the physical unit that the physical values of a variable refer to. In case of variable types with axes (e.g. curve and map variables) also the unit of the axes can be queried.

[CompuMethodInfo interface](#)

Provides information on the conversion method from raw to physical values. In case of vector or matrix variables this is the same for all elements. For variable types with axes the conversion method of the function values as well as the individual conversion methods for the axes can be obtained. The various types of conversion methods and associated metadata interfaces are explained in [Metadata on Conversion Methods](#).

The `VariableInfo` object for a certain `Port` variable can be obtained via the `GetVariableInfo` method of the `Port` object. The names of all available variables are provided by the `VariableNames` property of the `Port`.

Note: Each of the `Port` interfaces uses its own derivation from the `VariableInfo` interface to provide some additional, port type specific metadata. For example, there is a `MAPortVariableInfo` interface used to describe the variables provided by the `MAPort`. These port type specific derivations of `VariableInfo` are each contained in the package of the respective port type.

5.1.10.2 METADATA ON CONVERSION METHODS

XIL allows a client to operate on both the physical and the raw value, when accessing target system variables (for an explanation of value representations and how to select a specific representation for variable access, see section [Variable Ref and Value Representation Mode](#)). Any necessary conversion between the two representations is performed transparently by the server implementation or the target system itself. Nevertheless, XIL provides the possibility to retrieve information on the conversions applied by the server. [Table 41](#) gives the conversion methods for which information can be retrieved.

Table 41: Conversion methods supported by metadata interfaces (R: Raw value, P: Physical value)

Conversion Method	Description						
Identical	Physical and raw value are equal $P = R$						
Linear	Conversion is defined by a linear function with <i>Factor</i> and <i>Offset</i> as parameters $P = \text{Factor} * R + \text{Offset}$						
Rational Function	Conversion is defined by a rational function with an arbitrary number of terms in the numerator and denominator $P = \frac{N_0 + N_1 * R + N_2 * R^2 + \dots + N_n * R^n}{D_0 + D_1 * R + D_2 * R^2 + \dots + D_m * R^m}$ N ₀ ... N _n : n+1 coefficients of the numerator polynom D ₀ ... D _m : m+1 coefficients of the denominator polynom						
Texttable	Conversion from raw to physical value is defined by a lookup table that maps integer values to character strings. The mapping is one-to-one. Each numerical value is assigned exactly one textual value and vice versa, i.e. there must not be two or more lookup table entries that contain the same numerical or textual value. Example lookup table: <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>R</th> <th>P</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>“OFF”</td> </tr> <tr> <td>1</td> <td>“ON”</td> </tr> </tbody> </table> Note: This conversion method requires the raw values to be integer values.	R	P	0	“OFF”	1	“ON”
R	P						
0	“OFF”						
1	“ON”						

For each of the conversion methods there is a specific CompuMethod interface, that provides the method specific parameters values. The CompuMethod interfaces can be found in the CompuMethod sub package of MetalInfo. [Figure 114](#) gives an overview of this specific kind of metadata interfaces. The CustomCompuMethod interface is used to represent conversions that do not match any method listed in [Table 41](#).

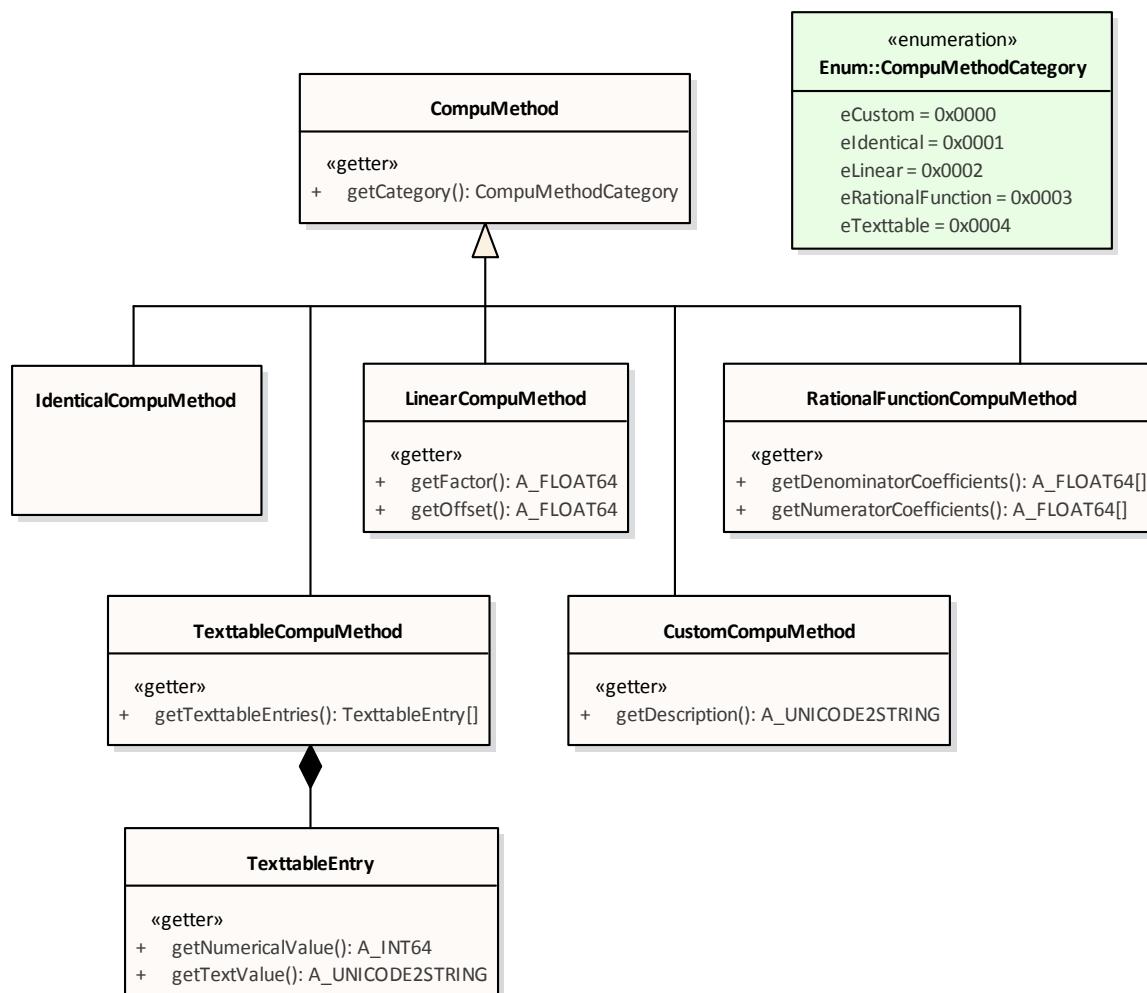


Figure 114: Interfaces for metadata query on conversion methods

The `CompuMethod` object for a certain `Port` variable can be obtained via the `CompuMethodInfo` property of the `VariableInfo` object for that variable (see section [Metadata on Variables](#) for details).

5.1.10.3 METADATA ON ACQUISITION RASTERS / PROCESSOR TASKS

Capturing and simultaneously reading or writing of variables are examples where the client has to specify a processor task to be used. Hence, the tasks available in the target system must be known to the client. For this purpose all `Port` interfaces with tasks provide a `TaskInfos` property that returns a list of `TaskInfo` objects. These `TaskInfo` objects provide information on the available tasks and their properties. [Figure 115](#) shows their interface.

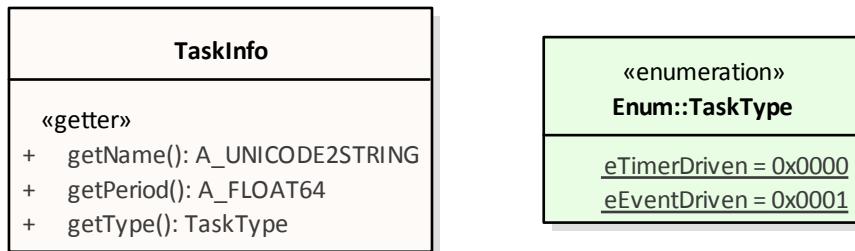


Figure 115: Interface for metadata query on acquisition rasters / processor tasks

The `TaskInfo` interface provides name and type of a task. The type indicates whether it is a timer driven, i.e. periodically executed task, or an event driven task, that is executed upon the occurrence of a certain event only. For timer driven tasks the cycle time can also be retrieved.

5.1.11 VARIABLE REF AND VALUE REPRESENTATION MODE

Wherever the client needs to access or specify variables of the target system, `VariableRef` objects are used. Typical use cases are the reading and writing of target system variables as well as their capturing and stimulation. Instead of plain variable names, `VariableRef` objects are used to specify the variables to operate on.

`VariableRefs` have the advantage that they can refer not only to the target system variable as whole, but - in case of structured variables - also to a single element thereof. With a `VariableRef`, for example, one can specify a single element of a vector variable to be read, written, monitored, captured, stimulated etc.

`VariableRef` instances are created by means of the `VariableRefFactory` that can be obtained from the `Testbench`. Figure 116 shows the various `VariableRef` interfaces. Their `VariableName` property gives the name of the target system variable (e.g. the model path of a simulator variable). If a variable is to be accessed as whole, no matter whether it is a scalar, vector, matrix etc. variable, a `GenericVariableRef` instance must be used. If a single element of a vector or matrix is to be accessed, a `VectorElementRef` resp. `MatrixElementRef` instance must be used.

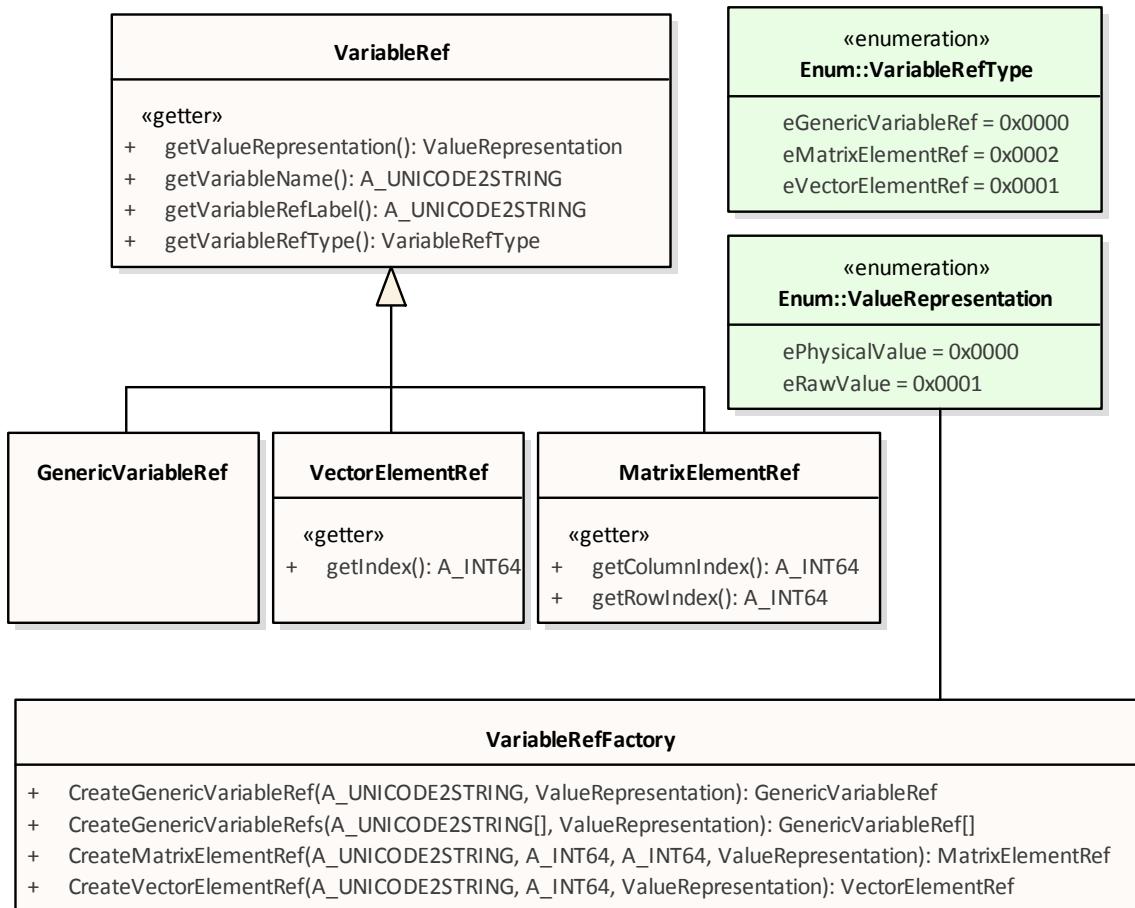


Figure 116: VariableRef interfaces

The second purpose of a **VariableRef** object is to specify the representation mode to be used on access to the variable. The representation mode is determined by the **ValueRepresentation** property which can have one of the following values:

ePhysicalValue

Specifies that operations are to be executed on the physical value of a variable. The physical value denotes the human readable representation. This is either a measured value with a physical unit or a verbal text value. The physical value is also known as application value.

eRawValue

Specifies that operations are to be executed on the raw value of a variable. The raw value denotes a simulator's or ECU's internal representation as well as the encoded signal value when transmitted over a communication network. The raw value is also known as internal or implementation value.

Note: **VariableRef** instances are used as keys in dictionaries (e.g. in the **Assignments2** collection of **SignalGenerator**). That is why they must be testable for equality and provide a hash value. **VariableRef** instances compare equal if and only if they match in all properties. Furthermore, **VariableRef** instances that compare equal have the same hash value.

Note: There are some cases where a unique, human readable name for the object referenced by a `VariableRef` object is required. The creation of names for the variable traces stored in a `CaptureResult` is a typical example for this requirement. A human should be able to draw conclusions about the identity of the captured variables from the trace names. That is why a string representation of the captured `VariableRef` object is used as trace name.

For these use cases the property `VariableRefLabel` was introduced. It yields a human readable, but unique identifier for the referenced variable resp. the referenced variable element. The generation of this name considers all properties of the `VariableRef`. However, the generated name is vendor specific.

5.1.12 DATA CAPTURING

5.1.12.1 INTRODUCTION

Capturing is the continuous data acquisition and its recording. It provides not only the acquired data, but also their chronological order and time of occurrence relative to the clock of the acquisition service. The acquired data can be retrieved after the capturing is completed or while the capturing is in progress.

The following sections describe the interfaces for setup and control of the capture process (sub package `Capturing`) as well as for access to captured data (sub package `CaptureResult`). These interfaces are commonly used by all `Ports` that support capturing (e.g. `MAPort`, `ECUMPort` and `NetworkPort`).

In the closing section [Usage examples for Capture and Capture Result](#) of this chapter a number of related examples can be found that demonstrate how capturing and capture data access can be accomplished.

5.1.12.2 CAPTURE CONFIGURATION AND CONTROL

Configuration and execution control of the acquisition process are performed via the `Capture` interface (see [Figure 117](#)), which can be found in sub package `Capturing` of the `Common` package. `Capture` instances are created via the `CreateCapture` method of the `Port` objects supporting capturing (e.g. `MAPort`).

Capture
<ul style="list-style-type: none">+ ClearConfiguration(): void+ Fetch(whenFinished: A_BOOLEAN): CaptureResult+ SetStartTrigger(watcher: Watcher, delay: Duration): void+ SetStopTrigger(watcher: Watcher, delay: Duration): void+ Start(writer: CaptureResultWriter): void+ Stop(): void+ TriggerClientEvent(eventId: A_UINT64, description: A_UNICODE2STRING): void «getter»+ getCaptureResult(): CaptureResult+ getDiscardFetchedData(): A_BOOLEAN+ getDownsampling(): A_UINT64+ getMinBufferSize(): A_INT64+ getPort(): Port+ getRetriggering(): A_INT64+ getStartTriggerCount(): A_INT64+ getStartTriggerDelay(): Duration+ getStartTriggerWatcher(): Watcher+ getState(): CaptureState+ getStopTriggerDelay(): Duration+ getStopTriggerWatcher(): Watcher+ getVariables2(): VariableRef[] «getter, deprecated»+ getDurationUnit(): DurationUnit+ getVariables(): A_UNICODE2STRING[] «setter»+ setDiscardFetchedData(discardFetchedData: A_BOOLEAN): void+ setDownsampling(downSampling: A_UINT64): void+ setMinBufferSize(minBufferSize: A_INT64): void+ setRetriggering(Retriggering: A_INT64): void+ setVariables2(variableRefs: VariableRef[]): void «setter, deprecated»+ setDurationUnit(durationUnit: DurationUnit): void+ setVariables(variableNames: A_UNICODE2STRING[]): void «deprecated»+ SetStartTriggerCondition(triggerDefinition: Watcher, delay: A_FLOAT64): void+ SetStopTriggerCondition(triggerDefinition: Watcher, delay: A_FLOAT64): void

Figure 117: Capture interface for capture configuration and control

A Capture instance stays always connected to its creating Port object. The Port object determines the variables available to the connected Capture. Variables of other Port objects cannot be recorded. However, there can be multiple Capture instances connected to the same Port object. Each Capture instance represents a separate data capture process that can be independently configured, started and stopped.

A capture process is basically defined by the set of variables to be captured, the acquisition task (aka sampling raster) as well as the capture duration and / or trigger conditions. [Table 42](#) gives the corresponding properties of the `Capture` interface that allow to set or retrieve these configuration options. Note, the name of the acquisition task must be specified on creation of the `Capture` instance and cannot be changed afterwards.

The variables to be captured are specified via the `Variables2` property, which is a collection of `VariableRef` objects (see section [Variable Ref and Value Representation Mode](#) for an explanation of `VariableRef` objects and their creation). For capturing a scalar variable or a complete vector / matrix variable a `GenericVariableRef` must be added to the `Variables2` collection. With vector or matrix variables it is also possible to record only individual elements of them. In this case `VectorElementRefs` resp. `MatrixElementRefs` are added to the `Variables2` collection.

The value representation mode specified via the `VariableRef`'s `ValueRepresentation` property is ignored, if the used capture format (e.g. MDF) can store conversion methods and the data source can provide raw values. In this case the variables' raw values are recorded including their conversion methods to physical values. This allows the `CaptureResult` to provide both physical values and raw values later.

Table 42 Configuration Properties and Methods of Capture

Property /Method	Description
ClearConfiguration	Clears any set configuration and releases all acquired resources and stored data.
DiscardFetchedData	Defines the behavior of <code>Fetch</code> method. If <code>DiscardFetchedData</code> is true, then <code>Fetch</code> discards the return data from memory. <code>getCaptureResult()</code> will not contain discarded data.
Downsampling	Defines the downsampling factor. A downsampling of n specifies that every n th value of the acquired data - based on the specified task (raster) - will be contained in the <code>CaptureResult</code> . The default is 1 (no downsampling). If downsampling is not supported by the XIL server an exception will be thrown (<code>eCOMMON_NOT_SUPPORTED</code>). (optional property)
MinBufferSize	Defines the minimum buffersize [byte] of the acquisition service. The buffer size has to ensure continuous measurement for the specified duration. As a default, the complete available buffer is reserved.
Port	Returns the <code>Port</code> instance the <code>Capture</code> is connected to.
Retriggering	Retriggering means that after receiving a stop trigger the data acquisition is waiting for a start trigger again. The Retriggering

Property /Method	Description
	<p>property is an integer that is interpreted as follows:</p> <p>Retriggering = 0 (default), no retriggering. This means that the given start trigger condition is watched once.</p> <p>Retriggering = N, where N is the (positive) number of consecutive start/stop trigger sequences. (Example: N = 1 means that the start trigger condition is watched two times, first time as normal trigger, second time as a first retrigger.)</p> <p>Retriggering = -1, defines an infinite number of start/stop trigger sequences.</p> <p>If the Retriggering value is not valid, an exception is thrown.</p>
SetStartTrigger	Sets the start trigger in form of a <code>Watcher</code> and optionally a pre-trigger interval or a trigger delay.
SetStopTrigger	Sets the stop trigger in form of a <code>Watcher</code> and optionally a trigger delay.
StartTriggerCount	When a start trigger is defined, this property reflects the number of start triggers that have been detected so far. On the start of the capturing process this property is initialized to a value of zero.
StartTriggerDelay	Returns the pre-trigger interval or trigger delay configured for the start trigger.
StartTriggerWatcher	Returns the configured start trigger in form of a <code>Watcher</code> .
State	Provides the state of the <code>Capture</code> and the capture process.
StopTriggerDelay	Returns the trigger delay configured for the stop trigger.
StopTriggerWatcher	Returns the configured stop trigger in form of a <code>Watcher</code> .
Variables2	Defines the variables to be captured in form of a <code>VariableRef</code> collection. The collection passed to the property's setter must be complete (subsequent additions / removals of collection elements require a re-assignment of the collection for the changes to take effect). Setting the property completely replaces any previously configured variable list.

`Capture` objects have a state that can be retrieved via its `State` property. The state indicates whether the `Capture` is in configuration or in execution mode and it reflects the status of the acquisition / recording process in execution mode. See [Table 43](#) for the possible states and their description.

Table 43 Capture states description

State	Description
eCONFIGURED	After creation, a Capture object is in state eCONFIGURED. In this state, the capturing is defined / configured.
eACTIVATED	In this state, the Capture waits until the start trigger condition becomes true and then switches into eRUNNING. A Stop method call or a start trigger leaves this state.
eRUNNING	This state is reached as soon as the start trigger condition becomes true or in untriggered capturing mode the start method is called. While residing in this state, data is acquired. It remains in this state, until <ul style="list-style-type: none">• the stop method is called or• the stop trigger becomes true and the time delay has expired (in case of no retriggering or in case of retriggering and the retriggering count is reached)• the stop trigger becomes true (in case of retriggering and the retriggering count is not reached).
eFINISHED	This state is reached as soon as the last stop trigger condition becomes true and the stop trigger delay has expired, according to the retriggering definition. It remains in this state, until a Stop method call switches into eCONFIGURED.

State changes and thus the capture process are controlled via the Capture object's Start and Stop methods as well as optionally defined start and stop triggers. Characterized by the use or non-use of start and stop triggers and the Retriggering property, 3 different capturing modes are distinguished. Please refer to sections [Untriggered Capturing](#), [Triggered Capturing](#) and [Re-triggered Capturing](#) for a detailed description including the state transitions, triggers and conditions involved.

Note: A Capture's behavior is usually influenced by the state of the Port object the Capture is connected to. These dependencies are documented in the sections describing the respective Port type. For dependencies on the MAPort state, please refer to section [Relation between MAPort and Capturing / SignalGenerator](#).

The methods and properties of the Capture interface are bound to certain states of a Capture object. Table 44 provides an overview in which states which methods and properties may be called.

Table 44 State dependencies of Capture methods and properties

	eCONFIGURED	eACTIVATED	eRUNNING	eFINISHED
Method ClearConfiguration	x			
Method Fetch		x	x	x
Method Start	x			
Method Stop		x	x	x
Method TriggerClientEvent		x	x	
Method SetStartTrigger	x			
Method SetStartTriggerCondition	x			
Method SetStopTriggerCondition	x			
Method SetStopTrigger	x			
Property getCaptureResult	x			
Property getDiscardFetchedData	x	x	x	x
Property getDurationUnit	x	x	x	x
Property getDownsampling	x	x	x	x
Property getMinBufferSize	x	x	x	x
Property getPort	x	x	x	x
Property getRetriggering	x	x	x	x
Property getState	x	x	x	x
Property getVariables	x	x	x	x
Property getVariables2	x	x	x	x
Property getStartTriggerCount	x	x	x	x
Property getStartTriggerWatcher	x	x	x	x
Property getStopTriggerWatcher	x	x	x	x
Property getStartTriggerDelay	x	x	x	x
Property getStopTriggerDelay	x	x	x	x
Property setDiscardFetchedData	x			
Property setDurationUnit	x			
Property setDownsampling	x			
Property setMinBufferSize	x			
Property setRetriggering	x			
Property setVariables	x			
Property setVariables2	x			

5.1.12.3 UNTRIGGERED CAPTURING

A Capture instance is in untriggered capturing mode if no start trigger and no stop trigger have been defined (i.e. neither `SetStartTrigger()` nor `SetStopTrigger()` have been called). In this case the capturing process is controlled solely by calling the methods `Start` and `Stop` of the `Capture` interface.

The following state diagram illustrates the state changes of a `Capture` instance in untriggered capturing mode.

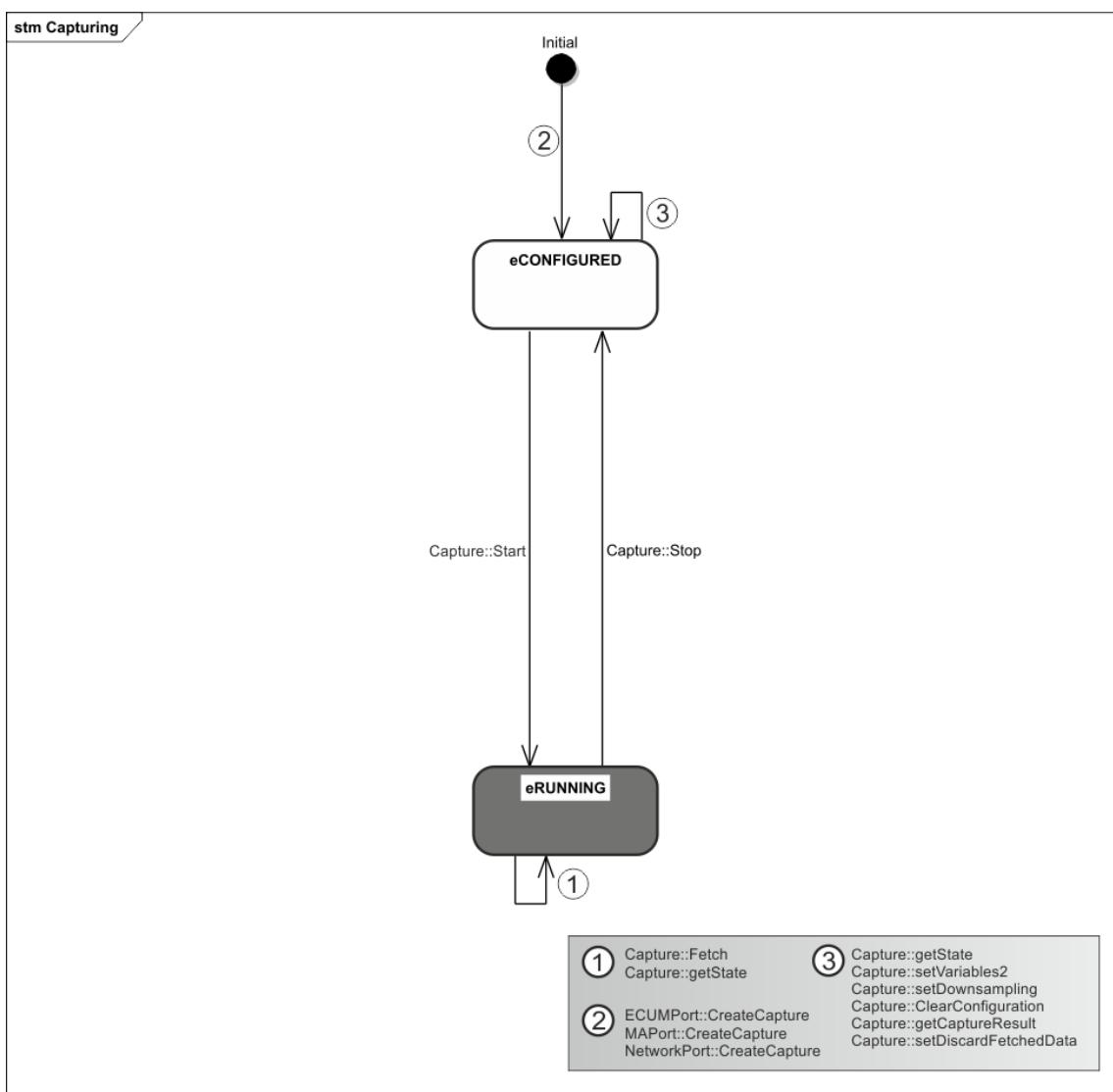


Figure 118: State diagram for untriggered capturing mode

After configuration of the `Capture` object, the method `Start` is called. In untriggered capturing mode the capturing process immediately switches the state to `eRUNNING` and data acquisition starts.

The capturing process is explicitly stopped by calling the method `Stop`.

The method `ClearConfiguration` clears any set configuration and releases all acquired resources and stored data.

5.1.12.4 TRIGGERED CAPTURING

A `Capture` instance is in triggered capturing mode if a start trigger, a stop trigger or both have been defined (i.e. `SetStartTrigger()` and / or `SetStopTrigger()` have been called) and the property `Retriggering` has the default value 0. In this case start resp. end of the capturing process are not only controlled by the methods `Start()` and `Stop()` of `Capture` interface, but also by events fired by the `Watchers` used for the start resp. stop triggers.

The start trigger is defined by a `ConditionWatcher` object. A `DurationWatcher` object is not allowed as start trigger. The stop trigger is defined by either a `ConditionWatcher` or a `DurationWatcher` object. This allows stopping the data acquisition after a certain period of time (`DurationWatcher`) or on the occurrence of a certain condition (`ConditionWatcher`).

The following state diagram illustrates the state changes of a `Capture` instance in triggered capturing mode without retrigerring.

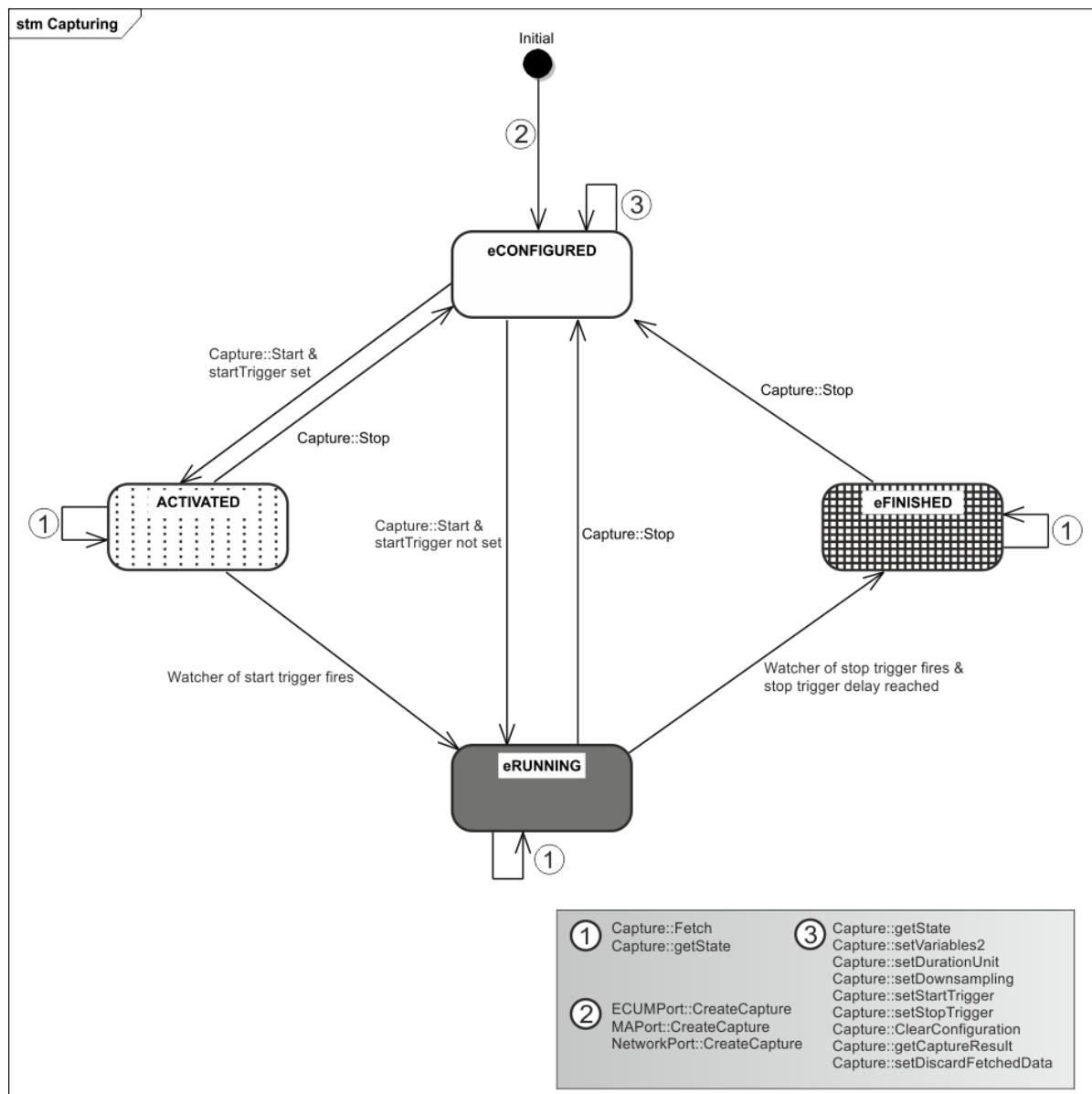


Figure 119: State diagram for triggered capturing mode without retriggering

After configuration of the `Capture` object, the method `Start` is called. If no start trigger has been defined, the capturing process immediately switches the state to `eRUNNING` and data acquisition starts. If a start trigger has been defined, `Start` puts the `Capture` from state `eCONFIGURED` into the state `eACTIVATED`.

If a start trigger has been defined, the capturing process starts to acquire data and internally stores the data into a ring buffer. As soon as the watcher for the start trigger fires its event, the capture state `eRUNNING` is entered.

If a stop trigger has been defined, the capture process stays in state `eRUNNING` until the `Watcher` of the stop trigger fires its event and then the stop trigger delay has expired. It then enters the capture state `eFINISHED`.

If no stop trigger has been defined, the capture process stays in state `eRUNNING` until `Stop()` is called. Then it enters the capture state `eCONFIGURED`.

Any of the states can be left by calling the method `Stop()` at any time.

Note: Only data acquired between the start trigger time minus a negative delay or plus a positive delay and the stop trigger time plus a positive delay is copied to the `CaptureResult`. If no time delay is specified, the capturing starts and ends exactly at the trigger times. If the start or stop trigger is missing, the capturing starts or ends immediately with the call to `Start()` resp. `Stop()`.

Note: A negative stop trigger delay cannot be specified and leads to an exception (`eCOMMON_STOP_TRIGGER_NOT_SET`). Reason: By calling `Fetch()` measured data can be obtained until immediately before the stop trigger fires, and thus data measured before the stop trigger fires cannot be excluded from the obtained `CaptureResult`.

5.1.12.5 RE-TRIGGERED CAPTURING

A `Capture` instance is in retriggered capturing mode if both a start trigger and a stop trigger have been defined and the property `Retriggering` is not zero. In this mode the triggered capturing process is repeated after a data frame is completed. Note, re-triggering only makes sense if a start trigger and also a stop trigger have been defined.

The following state diagram illustrates the state changes of a `Capture` instance in triggered capturing mode with retriggering enabled.

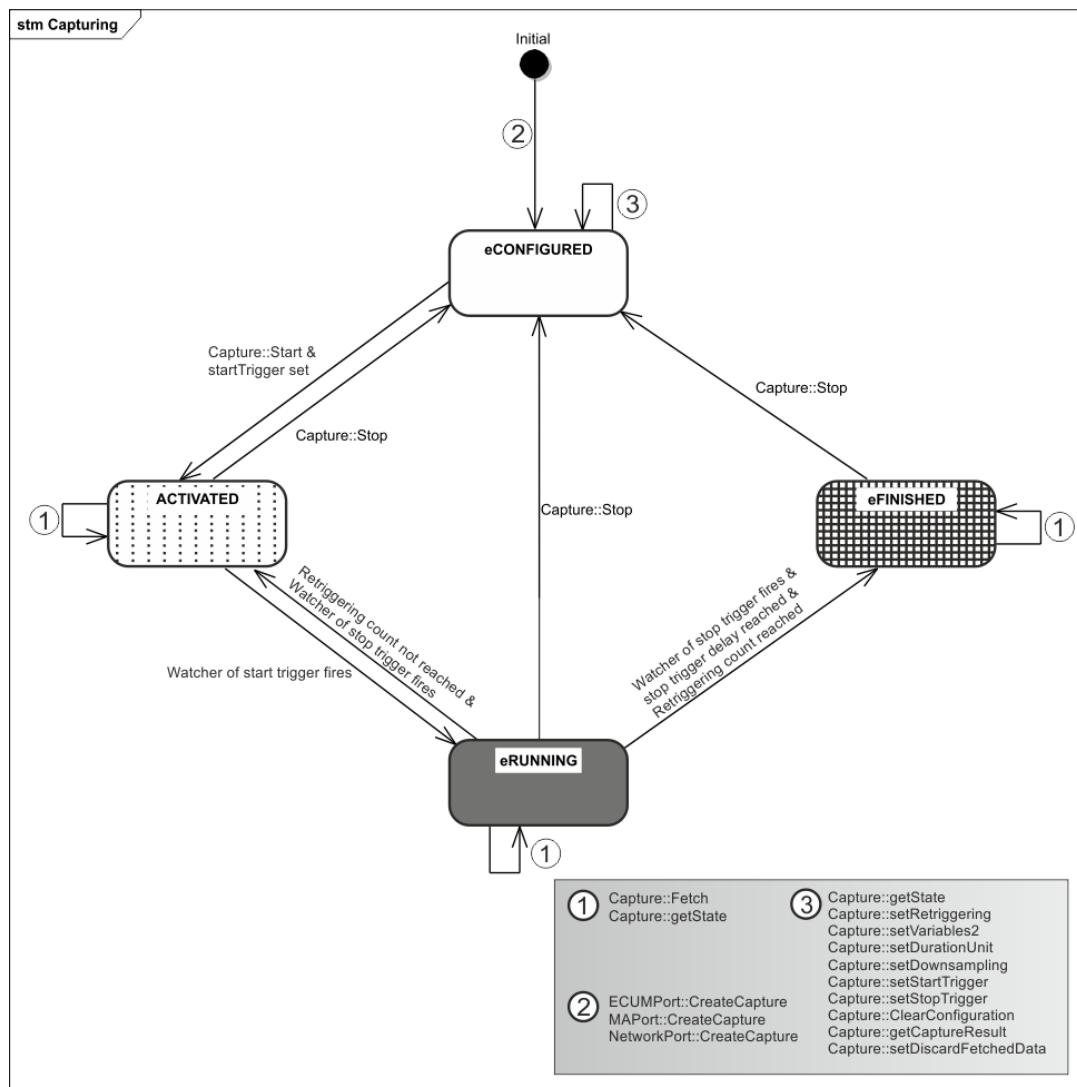


Figure 120: State diagram for retrigged capturing mode

After configuration of the Capture object, the method Start is called to put the Capture from state eCONFIGURED into state eACTIVATED.

In retrigged capturing mode, the capturing process starts to aquire data and internally stores the data into a ring buffer. As soon as the watcher for the start trigger fires its event, the capture state eRUNNING is entered.

The capture process now stays in state eRUNNING until the watcher of the stop trigger fires its event. If the StartTriggerCount is less than or equal to the value of the Retriggering property, the capture state eACTIVATED is entered and the StartTriggerCount is incremented. In this state the capture process waits again for a start trigger. Please note that a stop trigger delay is not considered for a state change as long as the retrigging count (i.e. StartTriggerCount minus 1) has not yet reached the value of the Retriggering property!

If the capture process is triggered again after the StartTriggerCount minus 1 has reached the value of the Retriggering property, the capture process stays in state

eRUNNING until the watcher of the stop trigger fires its event and then the stop trigger delay has expired. It then enters the capture state eFINISHED.

Any of the states can be left by calling the method `Stop()` at any time.

Note: Only data acquired between a start trigger time minus a negative delay or plus a positive delay and the associated stop trigger time plus a positive delay is copied to the `CaptureResult`. If no time delay is specified, each capturing cycle starts and ends exactly at the trigger times.

Note: A negative stop trigger delay cannot be specified and leads to an exception (see corresponding note in [Triggered Capturing](#) for details)

5.1.12.6 OBTAIN CAPTURE RESULTS FROM A CAPTURE

The `Capture` interface offers two ways to obtain the acquired data. The data can be retrieved via the property `CaptureResult` after the capturing is completed and via the method `Fetch` while the capturing is in progress.

Property `CaptureResult`

The purpose of the `CaptureResult` property is to make all acquired data available in a coherent form once the capturing process is complete. Therefore, the property may only be accessed in `Capture` state eCONFIGURED. This means that regardless of whether a stop trigger is used or not, the `CaptureProperty` is only available after the `Stop` method was called.

The `CaptureResult` property returns a `CaptureResult` object that contains all acquired data of the `Capture`, from the beginning of data acquisition until the end of data acquisition. If no data has been acquired (e. g. no start trigger condition has become true), the `CaptureResult` is empty. This means its `CaptureSignalGroup` objects have length 0 and return empty lists of time stamps and measured values.

Note: Depending on the property `DiscardFetchedData` and previous calls to method `Fetch`, there may be an arbitrary amount of data from the beginning of data acquisition missing in the `CaptureResult`. Or the `CaptureResult` may even be empty. See method `Fetch` for details.

Method `Fetch`

The purpose of `Fetch` is to provide online access to the acquired data during the capturing process. Therefore, it may only be called in `Capture` state eACTIVATED, eRUNNING and eFINISHED. Furthermore `Fetch` must only be used if the captured data is stored in memory (i.e. if the `CaptureResultMemoryWriter` is used). If the captured data is stored to a file (e.g. if the `CaptureResultMDFWriter` is used), a call to `Fetch` raises an exception.

The `Fetch` method returns a `CaptureResult` object that contains all acquired data since the last call of `Fetch`. If `Fetch` is called for the first time, it returns a `CaptureResult` object that contains all acquired data of the `Capture` from the beginning of data acquisition. If no data has been acquired meanwhile, the contained `CaptureSignalGroup` objects are empty. That means they have length 0 and return empty lists of time stamps and measured values.

Note: There is a parameter, `whenFinished`, in order to change semantic of the method `Fetch` and provide more convenience to the user.

`whenFinished := FALSE`

The method semantic is as described above.

`whenFinished := TRUE`

The method semantic is as described above, but additionally the call blocks until the Capture state `eFINISHED` has been reached.

Motivation: This provides a convenient way to wait for the end of data acquisition and then fetch the captured data with just one call. However, the user has to ensure that the state `eFINISHED` is reached at all. Otherwise, the call to `Fetch` will block indefinitely. Best practice is, either to provide start and stop trigger conditions, that definitely become true within a certain time, or to set timeouts via the `TimeOut` property of the `ConditionWatcher` objects. The timeouts ensure that the watcher objects start and stop the capturing process after the timeout has expired, even if the trigger conditions have not occurred.

Note: There is a property, `DiscardFetchedData`, that controls behavior of `Fetch` and the possible output of the `CaptureResult` property. It can only be set in `eCONFIGURED` state and its default value is `FALSE`.

`DiscardFetchedData := TRUE`

Method `Fetch` discards the data from memory that it returns. This data is no longer available and it will not be contained in the `CaptureResult` returned by a future call to the Capture's `CaptureResult` property. In other words, the `CaptureResult` returned by the `CaptureResult` property only contains the data after the last call of `Fetch`.

`DiscardFetchedData := FALSE`

All captured data is kept in memory even if `Fetch` is called and can be returned by a call to the Capture's `CaptureResult` property. That means the `CaptureResult` returned by the `CaptureResult` property contains all acquired data from the beginning until the end of acquisition.

Motivation: This provides a convenient way to decrease the memory footprint of the application if the use case does not require all the acquired data to be available at the end of the capturing session via the `CaptureResult` property.

5.1.12.7 CAPTURE RESULT

This section describes the `CaptureResult` interface and associated interfaces that are used to access the signal traces, events and metadata contained in the output of Capture objects as well as in measurement files (e.g. MDF files). See section [Obtain Capture Results from a Capture and Reader and Writer for CaptureResult](#) on how to obtain `CaptureResult` instances from Capture objects and files.

The signal traces contained in a `CaptureResult` object (i.e. the value sequences of the measured variables) are combined in groups. These signal groups are represented by the `CaptureSignalGroup` interface and can be obtained via the `SignalGroups` property of `CaptureResult`. A `CaptureResult` contains at least 1 `CaptureSignalGroup`, but can

contain any number. Figure 121 shows the `CaptureResult` and `CaptureSignalGroup` interfaces and their relation.

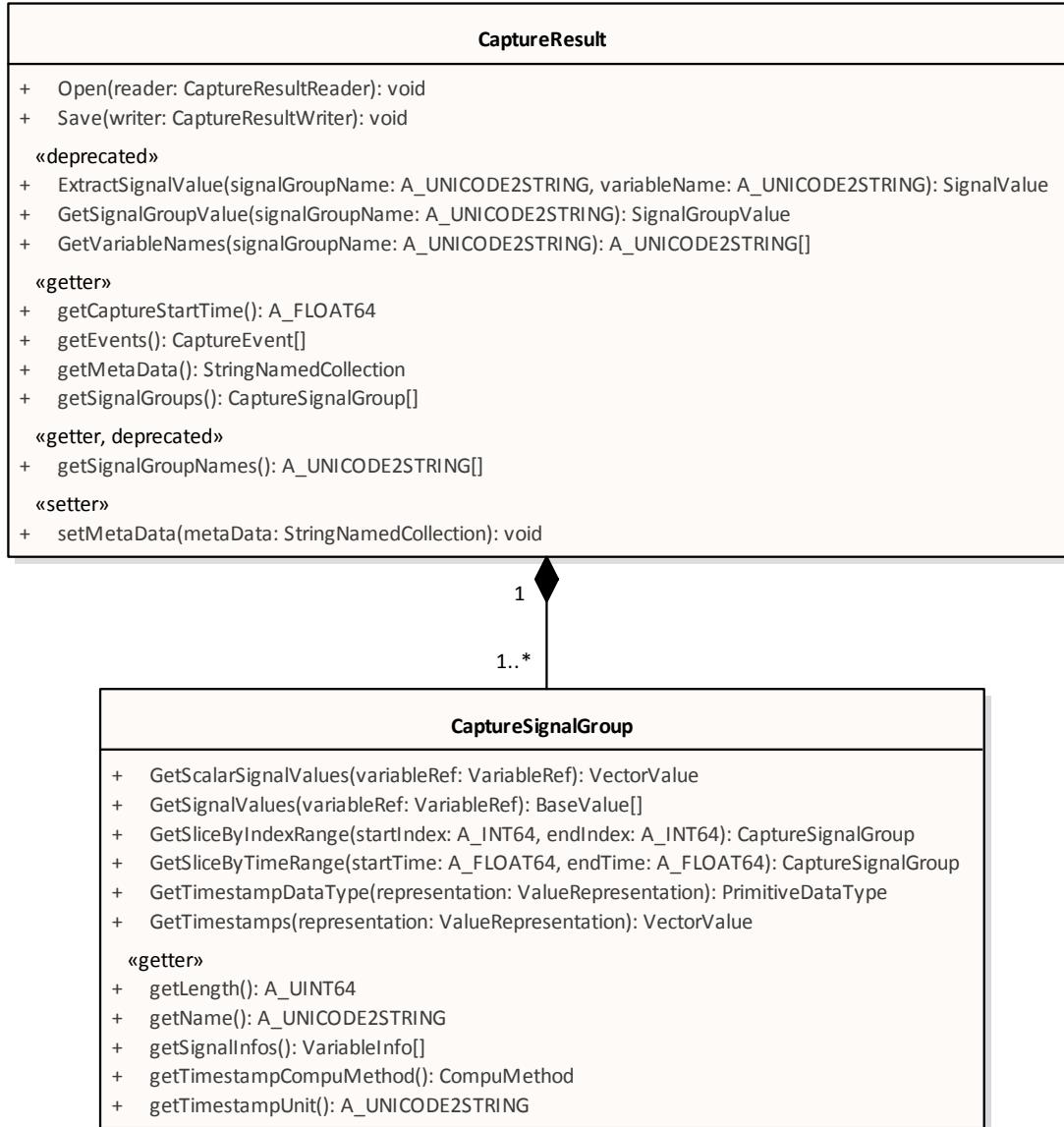


Figure 121: `CaptureResult` and `CaptureSignalGroup`

A signal group combines all signal traces that share a common sequence of timestamps. These are all signal traces that have been acquired and recorded using the same time source and sampling raster. Because of the common timestamp sequence, the number of measured values is the same for each signal trace in a signal group and it equals the number of timestamps in that signal group.

If a `CaptureResult` contains more than one signal group, the timestamps of the individual groups may differ in count and value. However, all of them must refer to the same time source and therefore be comparable. For example, if the capturing is performed on a multiprocessor platform and signals are captured by different processors, each processor may create its own signal group. The start and stop timestamps as well as the length of these signal groups (number of timestamps) may slightly differ. However, all timestamps of

all signal groups refer to the same (global) time. In other words: measured values with the same timestamp, but in different signal groups were acquired at the same (global) time.

Signal values and timestamps

The measured values in a signal trace can be obtained via the `GetSignalValue` method or the `GetScalarSignalValue` method of the `CaptureSignalGroup` interface. The associated timestamps are returned by `GetTimestamps` method. Measured values and associated timestamps are linked by their index values.

Both methods, `GetSignalValue` and `GetScalarSignalValue` use `VariableRef` objects to specify the signal trace (see section [Variable Ref and Value Representation Mode](#) for a description on `VariableRefs` and their creation). With `VectorElementRefs` and `MatrixElementRefs` it is possible to retrieve the value sequence of a single vector / matrix element out of a sequence of vector / matrix values. In contrast to `GetSignalValue`, method `GetScalarSignalValue` is restricted to the query of scalar signal values or values of single vector or matrix elements, but it enables vendors to provide a more efficient implementation.

Measured values as well as timestamps can be retrieved as physical values and as raw values. The desired representation for measured values is selected via the `ValueRepresentation` property of the `VariableRef` objects passed to `GetSignalValue` resp. `GetScalarSignalValue`. The representation of timestamps is selected via the `representation` parameter of the `GetTimestamps` method.

If the values in the `CaptureResult` are not qualified as physical or raw values, the `ValueRepresentation` property resp. the `representation` parameter is ignored. In this case `GetSignalValues`, `GetScalarSignalValues` and `GetTimestamps` return the same values for both representations. If the values are qualified as physical value or raw value, but the requested representation cannot be provided (because the conversion method is unknown or not implemented etc.), an exception is thrown.

MetaData

Name and count of the available signal traces as well as their data type, unit and other descriptive information can be determined using the `SignalInfos` property of the `CaptureSignalGroup` interface. This yields a list with a `VariableInfo` object for each signal trace in the signal group. The `VariableInfo` objects and its associated interfaces provide extensive metadata on the respective signal trace. A description of these metadata interfaces can be found in section [Metadata on Variables](#) and [Metadata on Conversion Methods](#).

Note: The names of the signal traces are not necessarily the same like the names of the captured variables. In case the `CaptureResult` has been created by a `Capture` object, the signal trace names are derived from the names of the captured variables. However, the implementation has some freedom in the assignment of signal trace names. This becomes clear at the latest when a single element of a vector or matrix variable is captured. Its trace name will not only be the name of the vector or matrix variable, but additionally encode the element index in a vendor specific way.

To enable a mapping between the names of the captured variables and the corresponding signal traces, XIL includes the following requirement. The `VariableRefLabels` of the `VariableRef` objects assigned to a `Capture` must be

used as name for the generated signal traces. That means one can determine the resulting signal trace names at Capture configuration time via query of the VariableRefLabels.

Note: If the values in the CaptureResult are not qualified as physical or raw values, the AvailableRepresentation property of the VariableInfo interface returns both representation modes.

Note: The data type of the signal trace differs from the data type of the captured variable if only a single element of a vector or matrix variable is captured. While the VariableInfo interface for the variable returns eVECTOR or eMATRIX as container type, the VariableInfo interface for the signal trace returns eSCALAR.

Additional metadata (that is not specific for a certain signal group but applies to the entire CaptureResult) can be added to the MetaData property of CaptureResult. There are two standardized entries that are automatically added by the Capture to indicate whether a start or stop trigger's time-out has been elapsed. However, these entries only exist if a ConditionWatcher has been used as start and / or stop trigger. See [Appendix C](#) for details.

Note: The trigger related information in the MetaData collection is deprecated and should not be used by clients any longer. Instead the CaptureEvents shall be used. CaptureEvents are not restricted to triggered capturing and provide more detailed information.

CaptureEvents

A CaptureResult object may contain objects of type CaptureEvent. A CaptureEvent marks a point in time within the CaptureResult that is of special interest. CaptureEvents are either automatically generated by the Capture (e.g. start and stop trigger events) or created by the client via call of the Capture interface's TriggerClientEvent method. [Figure 122](#) shows the resulting types of CaptureEvents and their attributes. Each CaptureEvent has a timestamp and a type. Depending on its type it may have further attributes identifying or describing the respective event.

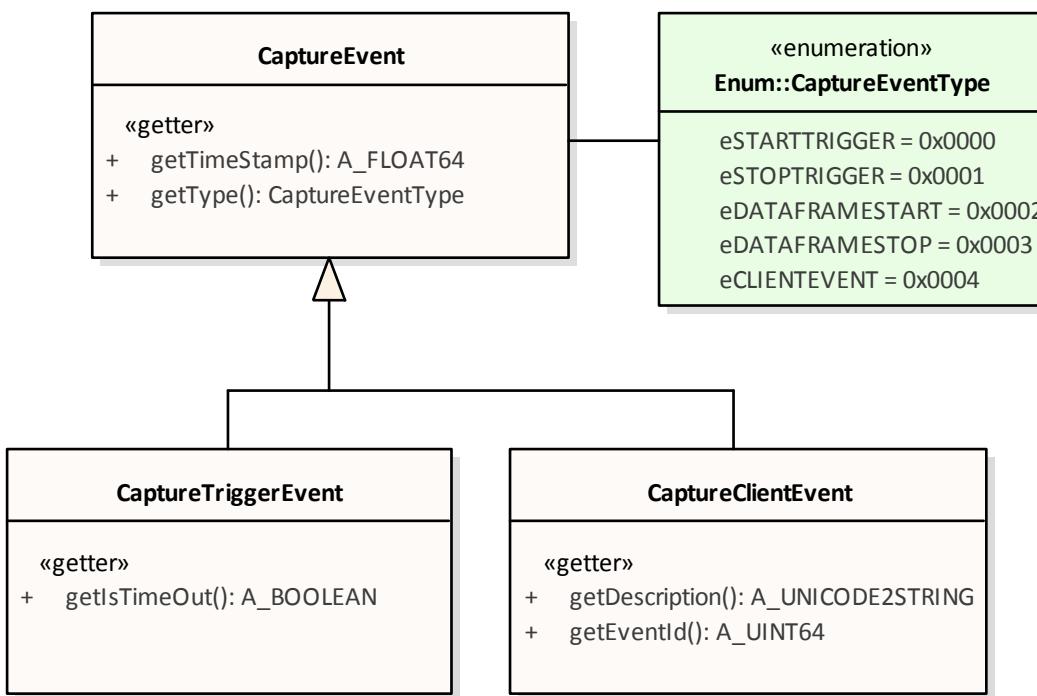


Figure 122: CaptureEvent interfaces

With the **CaptureEvents** automatically generated by the **Capture**, it is possible to retrieve the exact timestamps of trigger events and data frames (start and stop timestamps of the acquired data). See next section for details.

5.1.12.8 RELATION BETWEEN TRIGGERS, CAPTURE STATE, CAPTURE EVENTS AND DATA FRAMES

This section illustrates the relation between triggers, capture state and capture events and their effect on the content of the **CaptureResult**. It is structured according to capturing modes.

Untriggered capturing

If neither a start nor a stop trigger is defined, the capturing starts and stops when the `Start()` resp. `Stop()` method of the corresponding **Capture** instance is called. This case is illustrated in [Figure 123](#). The occurrence of the `Start()` method call indicates the zero-time of the **CaptureResult**.

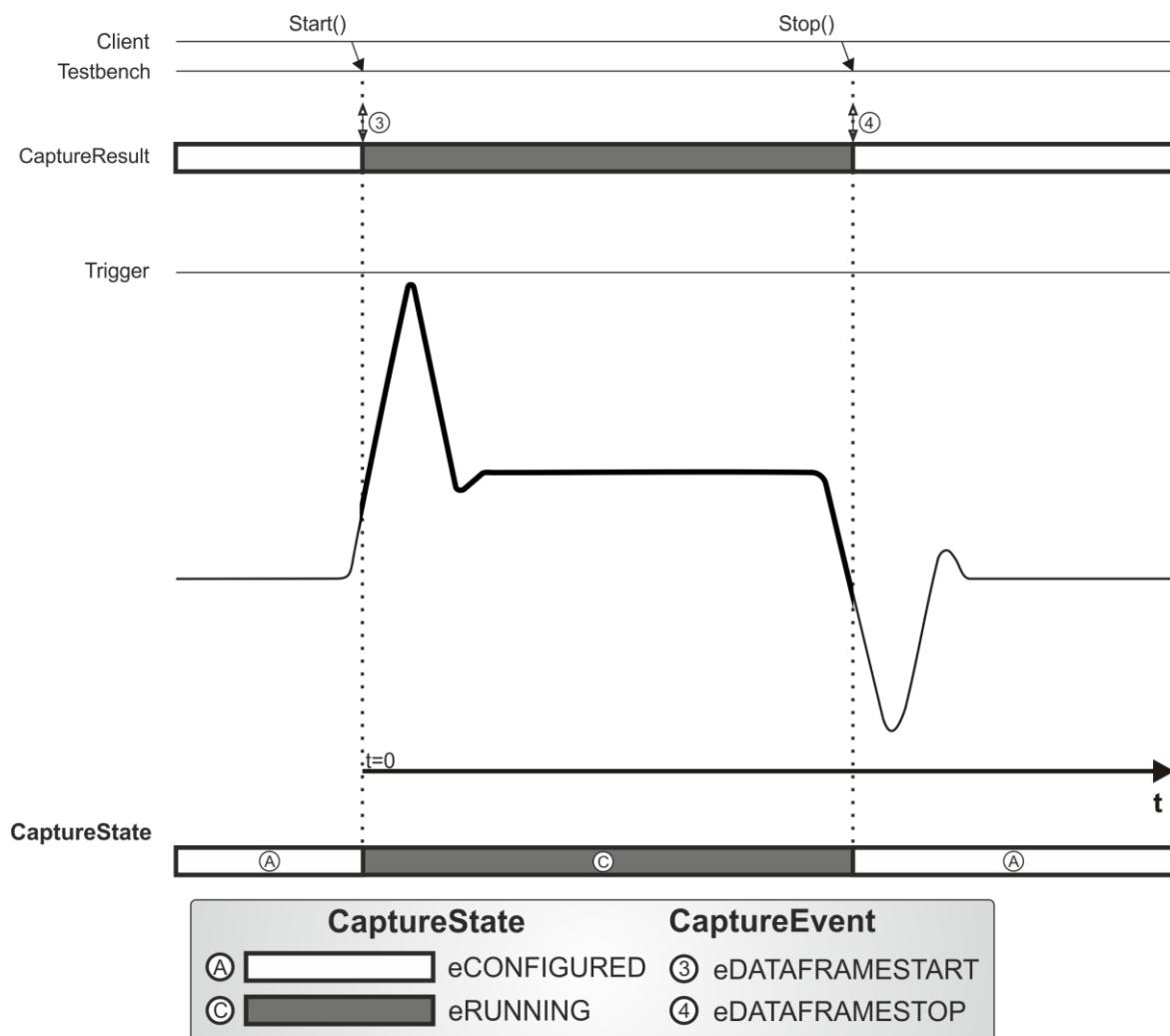


Figure 123: States, events and data frame in untriggered capturing mode

The Events collection of the CaptureResult object only contains a single pair of CaptureEvents which marks the start and end time of the continuous data frame in the CaptureResult. The first event is of CaptureEventType eDATAFRAMESTART and coincides with the zero time. The second event is of CaptureEventType eDATAFRAMESTOP.

At the bottom of Figure 123 the state changes of the Capture instance are shown.

Triggered capturing without trigger delays

Figure 124 illustrates the case where a start trigger and a stop trigger without any delay (delay is zero) are defined. In this case the capturing is started and stopped immediately when the start resp. stop trigger fires. The occurrence of the start trigger event indicates the zero-time of the CaptureResult.

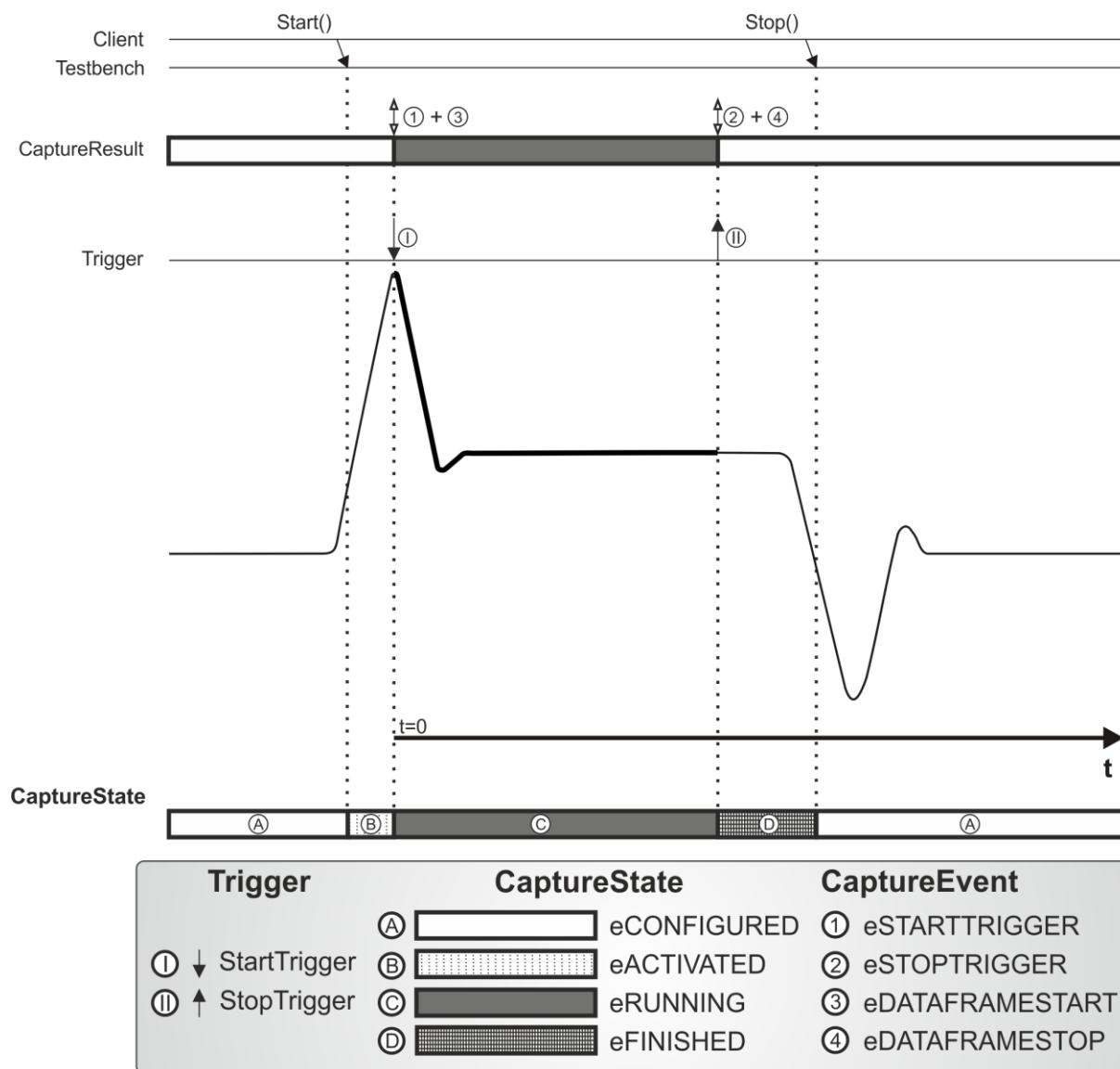


Figure 124: Trigger, states, events and data frame in triggered capturing mode without trigger delays

As with untriggered capturing there is one single, continuous data frame whose start and end time are marked by the `eDATAFRAMESTART` and the `eDATAFRAMESTOP` CaptureEvents in the `CaptureResult`'s Event collection. However, in triggered capturing mode, the Event collection additionally contains an `eSTARTTRIGGER` and / or `eSTOPTRIGGER` CaptureEvent, which marks the time of occurrence of the start resp. stop trigger.

Triggered capturing with positive trigger delays

Figure 125 illustrates the triggered capturing scenario where a positive delay is set for both the start and the stop trigger. This causes the capturing to start delayed by the specified period of time after the start trigger is fired. And it causes the capturing to be continued beyond the occurrence of the stop trigger until the specified stop delay has expired. Note that in this scenario also, the occurrence of the start trigger event defines the zero-time of the `CaptureResult`.

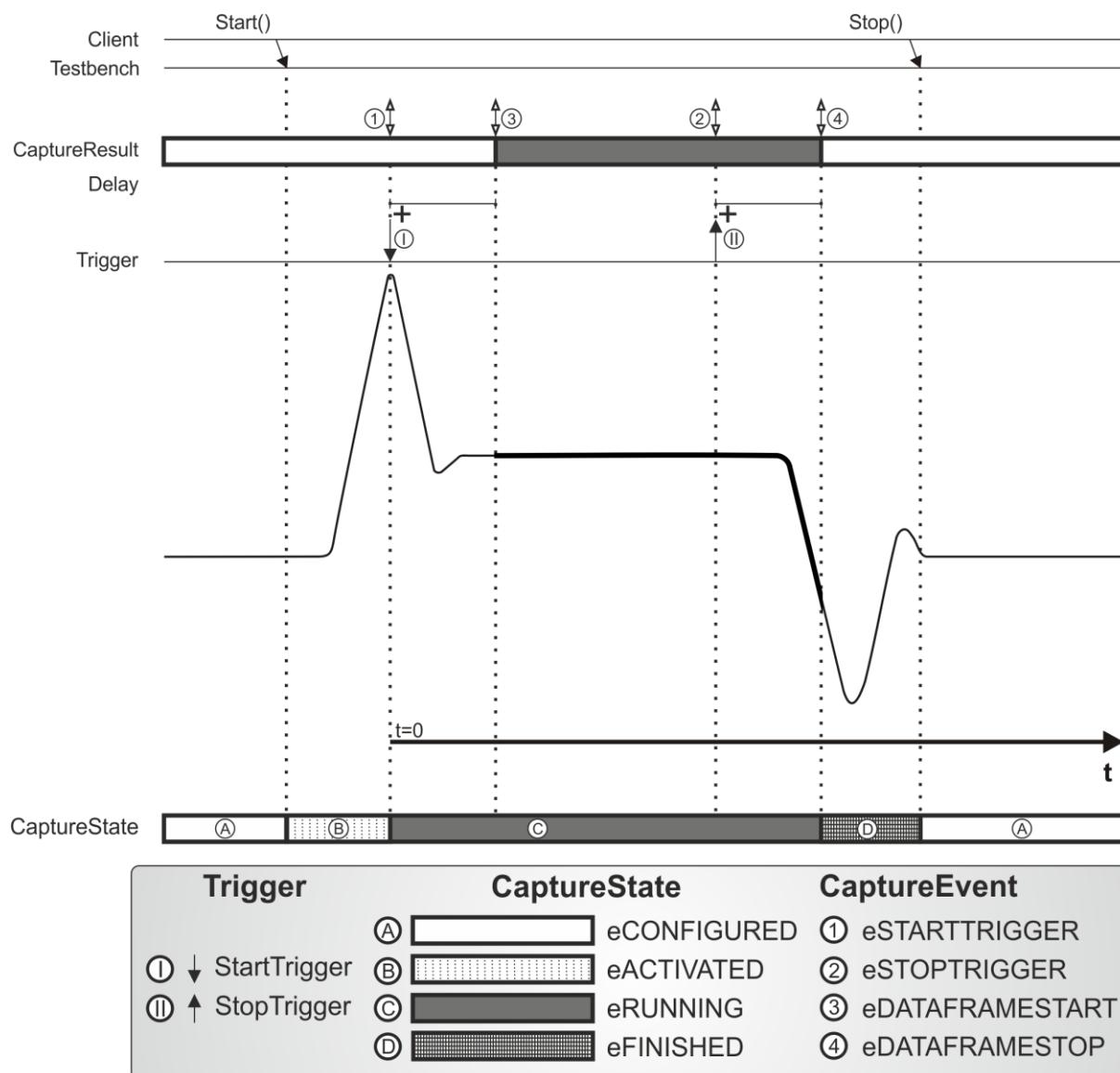


Figure 125: Trigger, states, events and data frame in case of positive start and stop trigger delays

As without trigger delay the `eDATAFRAMESTART` and `eDATAFRAMESTOP` CaptureEvent in the Events collection mark the start and end time of the single, continuous data frame in the `CaptureResult`. However, due to the positive delay, the timestamp of the `eDATAFRAMESTART` event is now greater than the timestamp of the `eSTARTTRIGGER` event, which is also contained in the Events collection. The same applies to the relation of the `eDATAFRAMESTOP` and the `eSTOPTRIGGER` event.

Triggered capturing with negative start trigger delay

[Figure 126](#) illustrates the triggered capturing scenario where a negative delay is set for the start trigger. This causes the capturing to start the specified period of time before the occurrence of the start trigger. That means the `CaptureResult` contains values from the time before the start trigger fired. The timestamps of these values are negative, because the occurrence of the start trigger defines the zero-time of the `CaptureResult`.

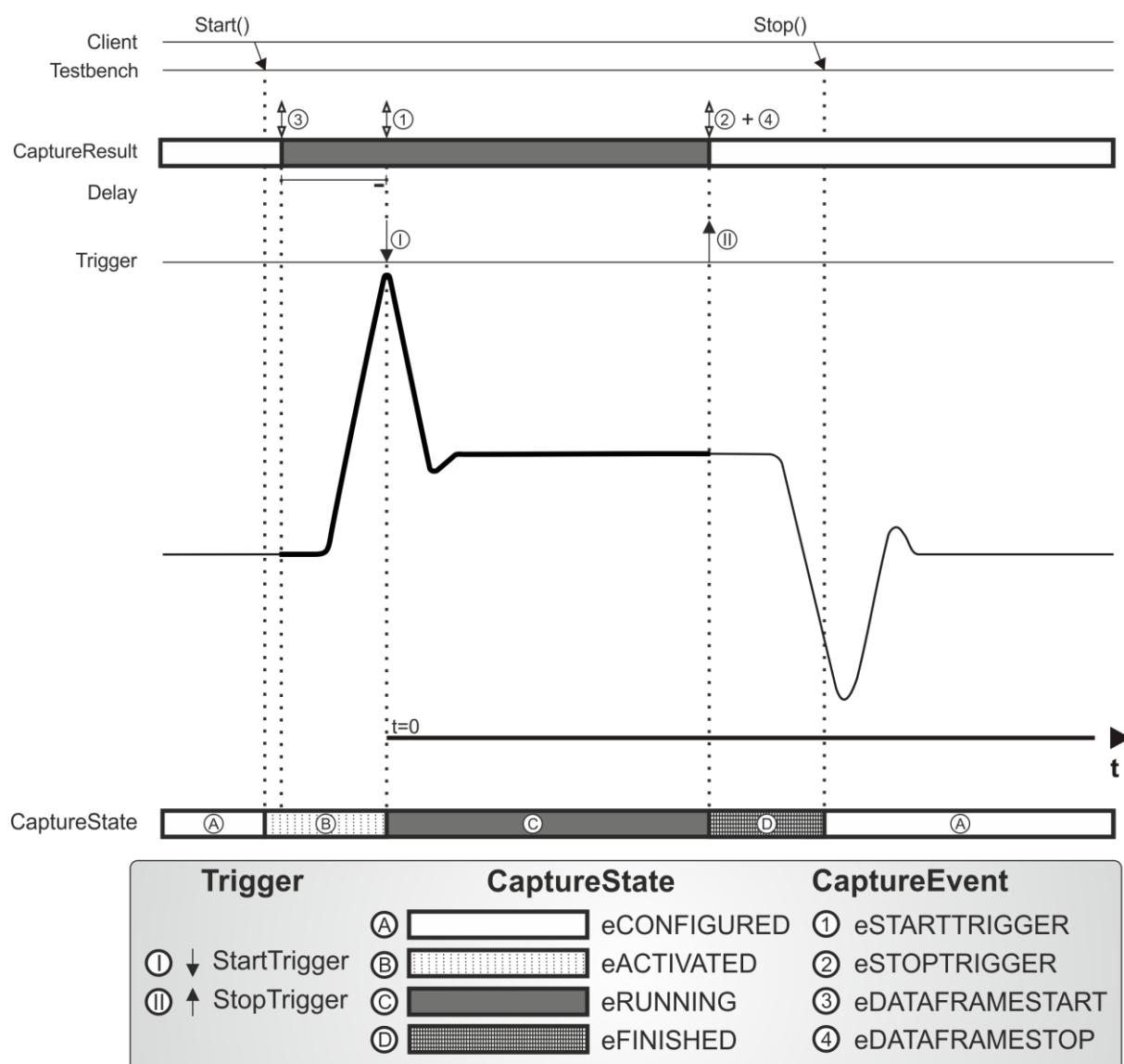


Figure 126: Trigger, states, events and data frame in case of negative start trigger delay

There is one single, continuous data frame contained in the `CaptureResult`. Its start and end time are marked by the `eDATAFRAMESTART` and `eDATAFRAMESTOP` CaptureEvent in the Events collection. However, due to the negative start delay, the timestamp of the `eDATAFRAMESTART` event is now smaller than the timestamp of the `eSTARTTRIGGER` event.

Note: Negative delays are only allowed for the start trigger. It is not possible to set a negative delay for the stop trigger.

Note: Obviously the number of measured values from the time before the start trigger fired is limited. It is not possible to obtain measured values that occurred before the call of the `Start()` method. So, the pre-trigger interval covered by the `CaptureResult` may be smaller than the specified negative delay.

Re-triggered capturing

Figure 127 illustrates a triggered capturing scenario (with stop trigger delay and negative start trigger delay) where the Retriggering property is not zero, but 1. This causes the

capturing to wait for the start trigger a second time after completion of the first capturing cycle and then start again. For Retriggering specifications greater than 1, the number of repetitions would be correspondingly larger.

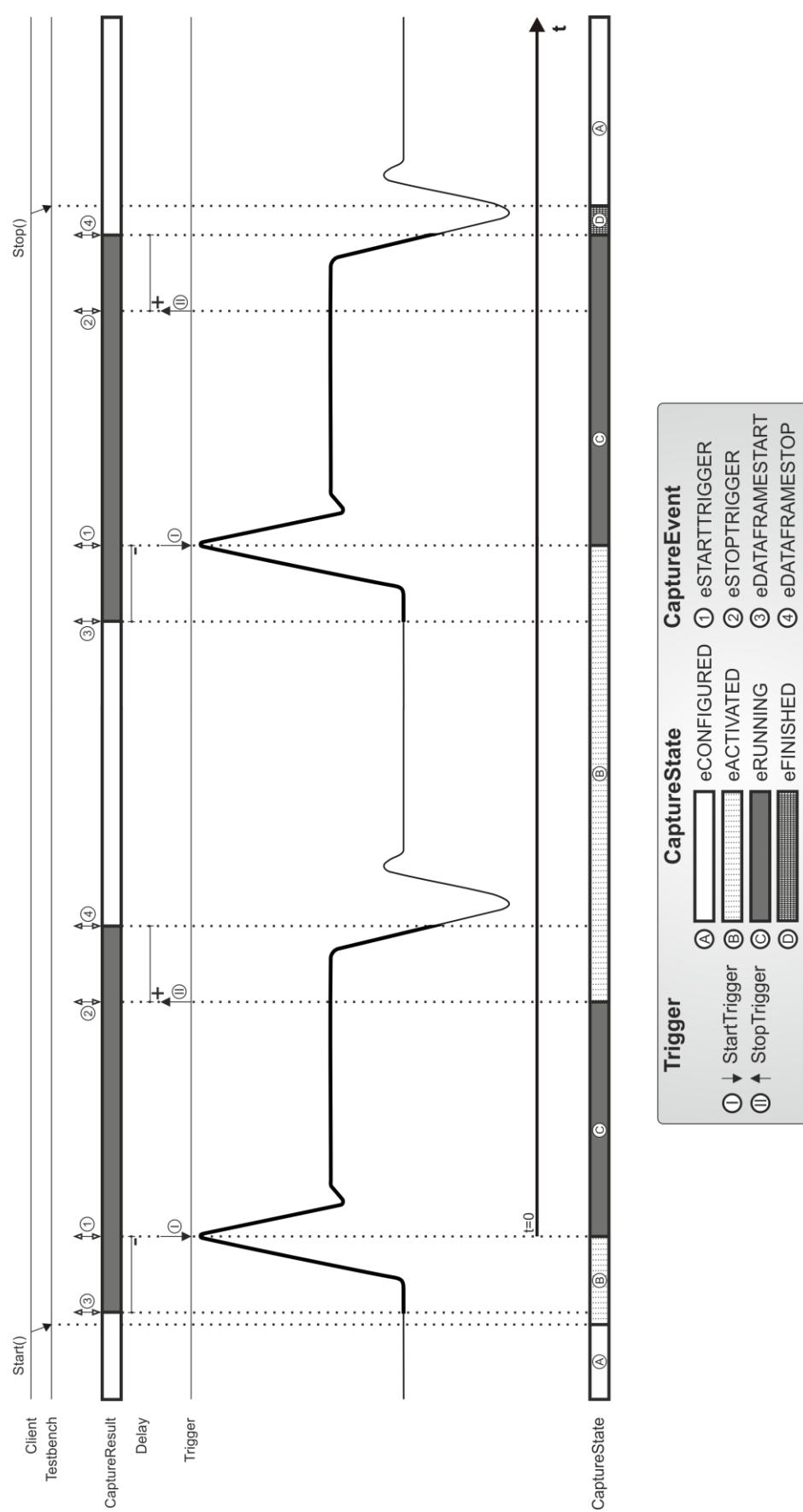


Figure 127: Trigger, states, events and data frames for re-triggered capturing

Since capturing is started twice, the `CaptureResult` contains two complete data frames. That means each data frame includes the pre-trigger interval of the start trigger and the post-trigger interval of the stop trigger. And, for each data frame, there is an `eDATAFRAMESTART` and `eDATAFRAMESTOP` `CaptureEvent` as well as an `eSTARTTRIGGER` and `eSTOPTRIGGER` `CaptureEvent` in the `Events` collection. For Retriggering specifications greater than 1, the number of data frames and `CaptureEvent` would be correspondingly larger.

Please note that in the case depicted in [Figure 127](#), there is a gap in the data contained in the `CaptureResult`. The `CaptureResult` contains no data for the interval between the first `eDATAFRAMESTOP` and the second `eDATAFRAMESTART` event. This is because there is a considerable interval between the occurrence of a stop trigger and the following start trigger, that is not completely covered by the start and stop trigger delays.

Re-triggered capturing with overlapping data frames

If re-triggering is used and the new start trigger occurs only shortly after the preceding stop trigger, the gaps between the individual data frames in the `CaptureResult` might become very small or completely disappear. Depending on the length of the specified start and stop trigger delays, the data frames might even overlap.

The case of overlapping data frames is depicted in [Figure 128](#). It is characterized by the fact that the sum of the stop trigger delay and the (negative) start trigger delay equals or exceeds the interval between the stop trigger of the first data frame and the start trigger of the second one. In this case the `eDATAFRAMESTART` event of the second frame has got a smaller timestamp than the `eDATAFRAMESTOP` event of the first frame.

Note: Even if data frames overlap, the data of the captured signals is contained in the `CaptureResult` only once.

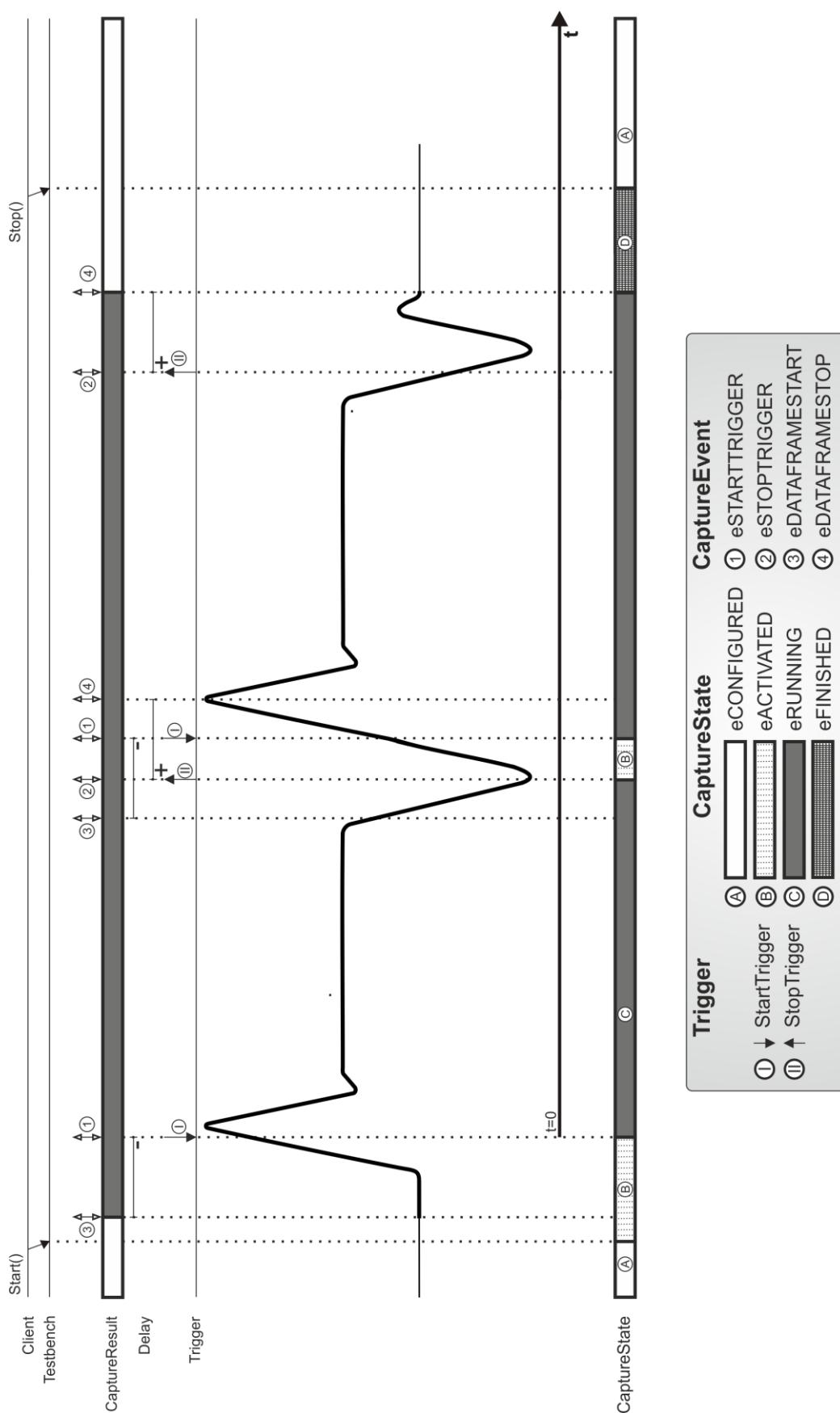


Figure 128: Re-triggered capturing with overlapping data frames

5.1.12.9 READER AND WRITER FOR CAPTURERESULT

This section describes the `CaptureResultWriter` and `CaptureResultReader` interfaces used to save the content of `CaptureResult` objects to persistent storage (e.g. MDF files) and to load capture data from persistent storage.

`CaptureResultWriter` objects are used to specify where and in which format a `Capture` object should store the acquired data. Therefor a `CaptureResultWriter` object is passed to the `Start` method of the `Capture` object.

`CaptureResultReader` objects are used to load capture data stored in a specific format into an existing `CaptureResult` object. The `CaptureResult` object can be created with the `CreateCaptureResult` method of the `CapturingFactory`.

The `CaptureResultWriter` and `CaptureResultReader` interfaces as well as the `CapturingFactory` can be found in the `Common\Capturing` package. [Figure 129](#) gives an overview on the reader and writer interfaces.

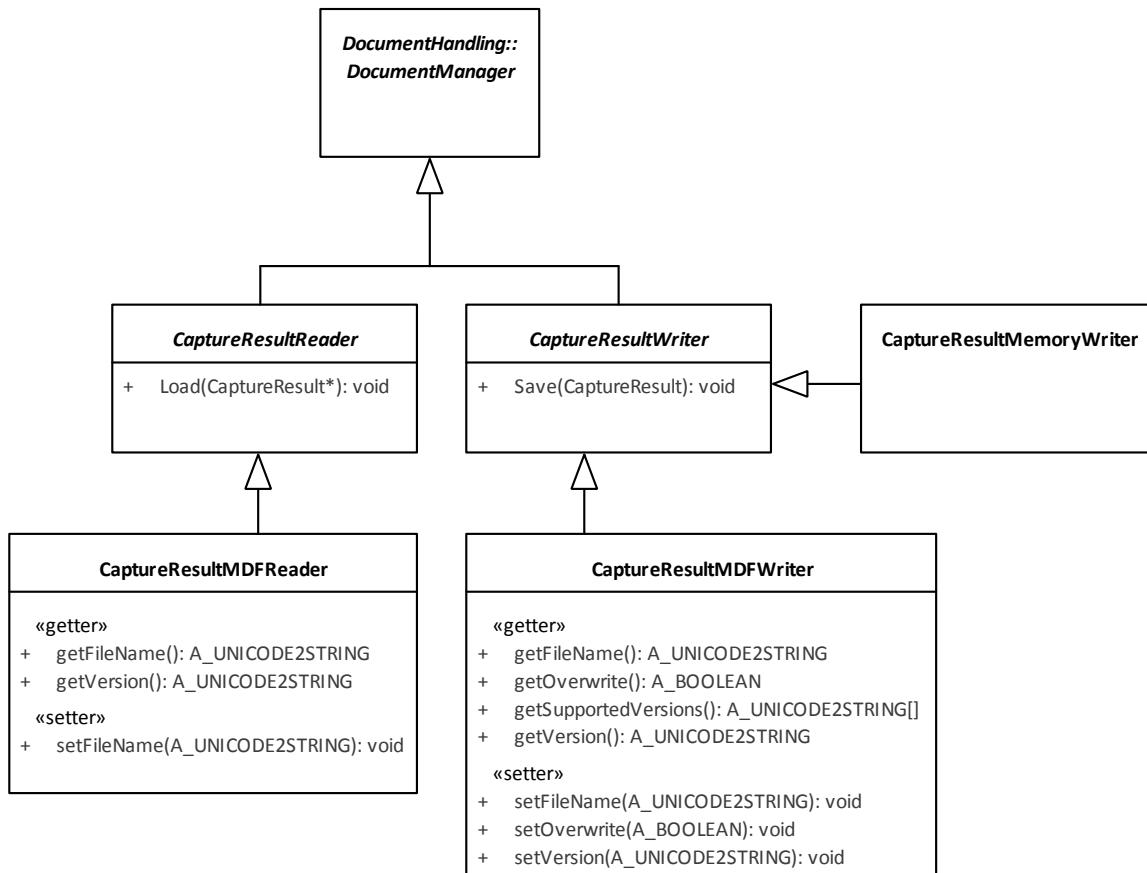


Figure 129: Reader and Writer interfaces for CaptureResult

CaptureResultReader & CaptureResultWriter

These are base interfaces for the concrete reader and writer interfaces. They provide a `Load` or a `Save` method in order to load and to save `CaptureResult` objects.

CaptureResultMDFReader

Instances of this interface handle the loading of MDF files [3]. The loaded data structure is stored in a `CaptureResult` object.

A reader implementation can support multiple versions of the MDF file format. The file's MDF version is automatically detected by the reader and can be retrieved via the `Version` property after loading. If the file's MDF version is not supported by the reader implementation an exception is thrown.

CaptureResultMDFWriter

Instances of this interface handle the saving of `CaptureResult` objects in files that conform to the MDF format.

A writer implementation can support multiple versions of the MDF file format. The version to be used can be specified via the writer's `Version` property. If the specified version is not, an exception is thrown. The list of supported MDF versions can be retrieved from the writer's `SupportedVersions` property. If the MDF version to be used is not specified, the highest version supported by writer is used.

The following syntax applies to the version strings used by the `Version` and `SupportedVersions` properties: `Major.Minor.[Maintenance]`, where each single part is a sequence of digits.

CaptureResultMemoryWriter

If a `CaptureResult` should not be stored in the file system during the capturing process, a `CaptureResultMemoryWriter` object has to be passed to the `Capture` object. In this case the `CaptureResult` is kept in the RAM. Note that `CaptureResultMemoryWriter` can only be used together with a `Capture` object.

5.1.12.10 USAGE EXAMPLES FOR CAPTURE AND CAPTURE RESULT

This section demonstrates how capturing and capture data access are basically accomplished. The simulation model variables, signal traces and the capture task used in the examples are fictitious.

The first step in data capturing is to create a properly configured `Capture` object. It mainly consists of creating the `Capture` object, specifying the variables to be captured and defining the start and stop trigger conditions. This procedure is depicted in [Figure 130](#).

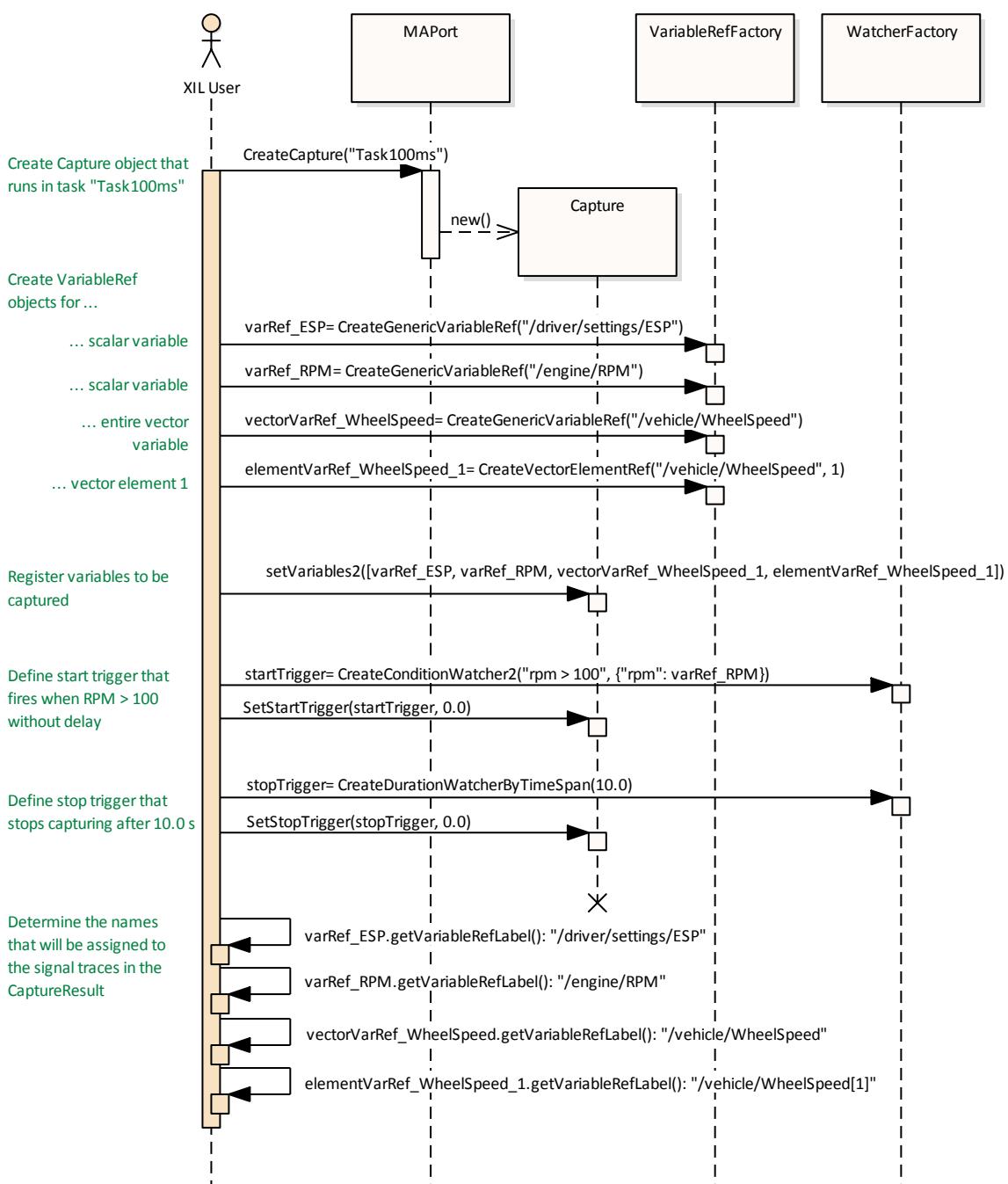


Figure 130: Create and configure a Capture object

The example demonstrates the creation of a `Capture` with the `MAPort`. It uses a 100 ms sampling raster and registers four quantities for capturing: Two simulation variables of scalar type and one vector variable. The fourth registered quantity is a single element (element with index 1) of the vector variable. Note that this is just to illustrate that you can capture vector resp. matrix variables both as whole and elementwise. (In practice, it does not make sense to capture a vector or matrix variable as whole and at the same time individual elements of it, since this would lead to redundant data in the `CaptureResult`.)

In the example, a start trigger definition ensures that the capturing is started as soon as the RPM variable exceeds the value 100 (the creation of the used `Defines` collection is not depicted to save space). A `DurationWatcher` set as stop trigger stops the capturing after a period of 10 s.

Special attention need to be paid to the last 4 lines of the sequence chart. They demonstrate how to obtain the names under which the signal traces will be stored in the `CaptureResult`. Later this will enable the correlation of the signal trace names in the `CaptureResult` with the variables that were registered in the `Capture`. This is important, since the variable names and signal trace names may differ, e.g. if only an element of a vector / matrix variable is captured.

Having the `Capture` object configured to match the intended capturing task, the capturing process must be performed. This procedure is shown in [Figure 131](#) and consists of three steps. First, the `Capture` is started, which activates the start trigger. Then it waits for the actual capturing process to be triggered by the start trigger and stopped again by the stop trigger. Finally, the `CaptureResult` is retrieved.

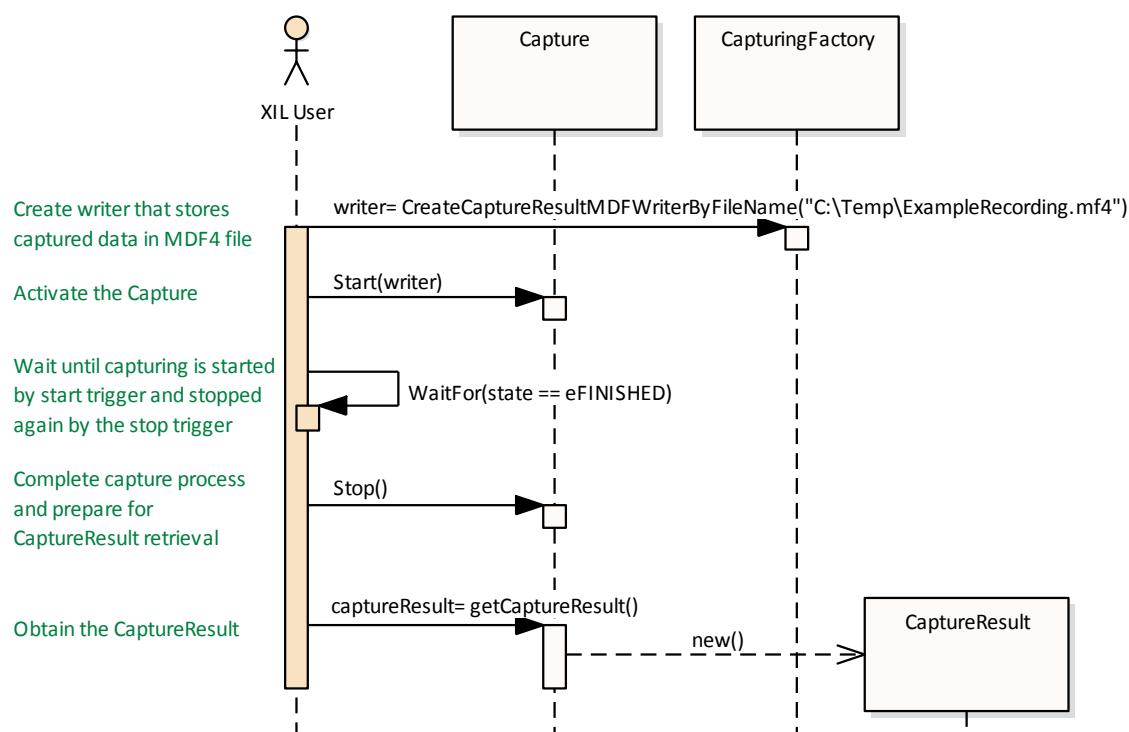


Figure 131: Perform the capturing process

The figure illustrates the use case where the captured data is stored in an MDF file. Therefore a writer for MDF files is created and passed to the `Start` method.

Please also note the call of the `Stop` method before the `CaptureResult` object is retrieved. This is always necessary, even if - as in the example - a stop trigger has already stopped the capturing process.

As an alternative to the depicted scenario, where the `CaptureResult` is obtained from the `Capture` object, it can also be retrieved from the created MDF file. This case is shown in Figure 132.

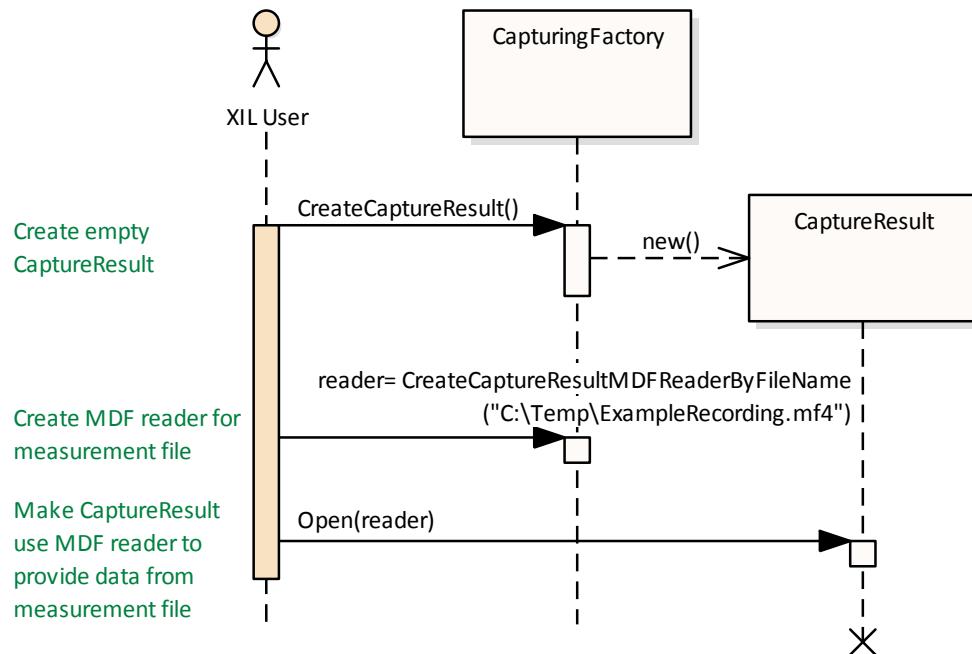


Figure 132: Create a `CaptureResult` from an MDF file

In order to create the `CaptureResult` based on a measurement file, a blank `CaptureResult` must be created and then associated with the measurement file by means of an appropriate writer object. This scenario becomes necessary if capture data processing is performed by another process or at a later time when the `Capture` is not available any more.

Now that the capturing is finished and the `CaptureResult` is available, the captured data can be processed. The following examples demonstrate how to obtain the timestamps and signal traces (sequences of measured values) from the `CaptureResult`. The first step, which is always necessary, is to obtain a handle of the signal group containing the signal traces of interest. However, to save diagram space, this step is only shown simplified in the sequence charts.

The access to scalar traces, which is the most common use case, is depicted in Figure 133. Scalar traces contain measured values of container type `eSCALAR`. They result from capturings of either scalar variables or individual elements of vector resp. matrix variables. Both cases are reflected by the example. This makes it clear that there is no difference in type or structure of scalar traces that would classify them into one or the other category. Hence, the retrieval of the measured values is performed in exactly the same way. In particular, a `GenericVariableRef` object is required in both cases. The trace name is the only attribute that would allow you to infer the container type of the source variable, since it contains an element index if a vector or matrix element has been captured.

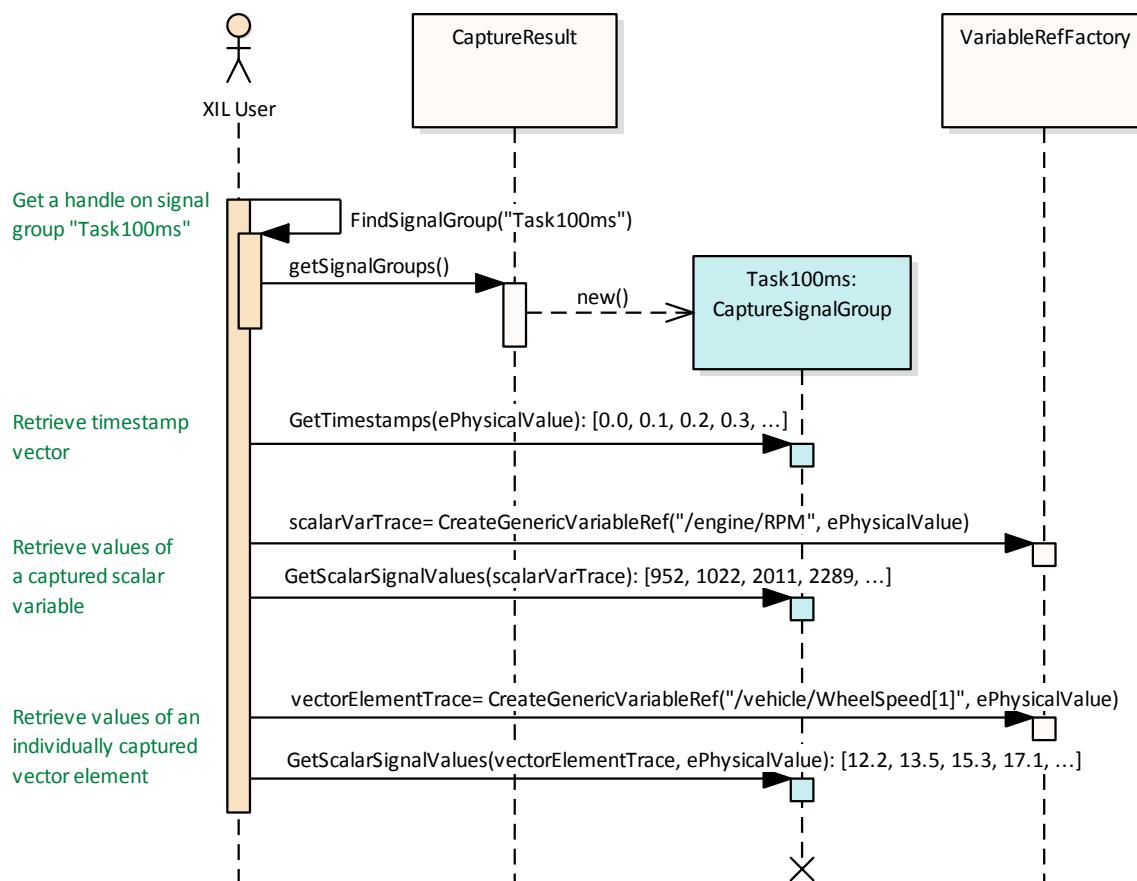


Figure 133: Access to scalar value traces in a CaptureResult

The access to non-scalar traces is depicted in [Figure 134](#). The measured signal values in such traces are structures of container type `eVECTOR` or `eMATRIX`. They result from capturings of entire vector resp. matrix variables. In the example the signal trace of a vector variable is used.

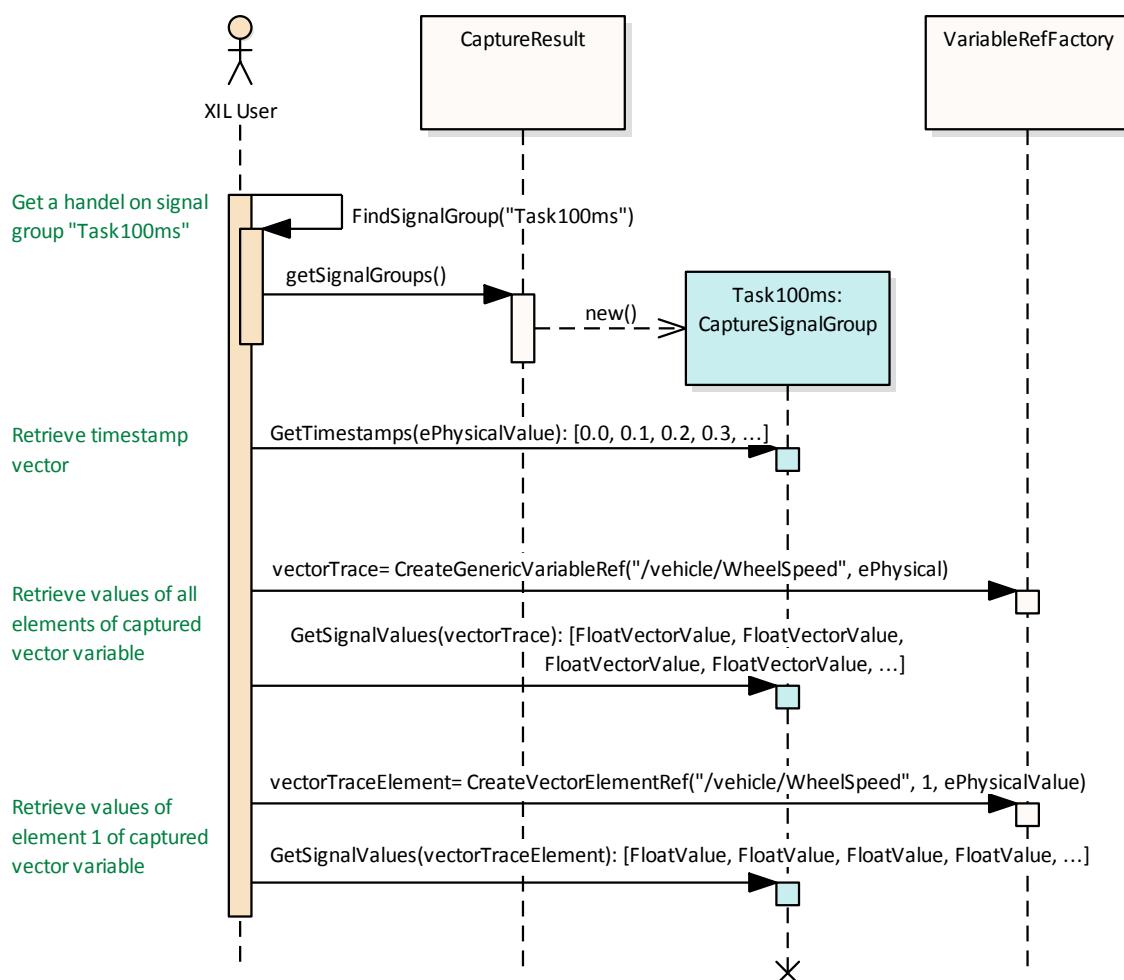


Figure 134: Access to vector value traces in a CaptureResult

The figure depicts the two options that are available when accessing the values of a non-scalar trace. One can either retrieve the sequence of the entire vector or matrix values or the value sequence of a certain element. The difference is that a `VectorElementRef` object must be created in order to obtain the value sequence of just one single vector element. This is because the `VectorElementRef` allows you to additionally specify an element index. In case of a matrix value trace a `MatrixElementRef` would be used for that purpose.

The timestamps are always scalar values. They are therefore retrieved in the same way as with scalar signal traces.

So far, only physical values were accessed in the examples. The sequence in [Figure 135](#) now demonstrates how to select the value representation mode and retrieve raw values.

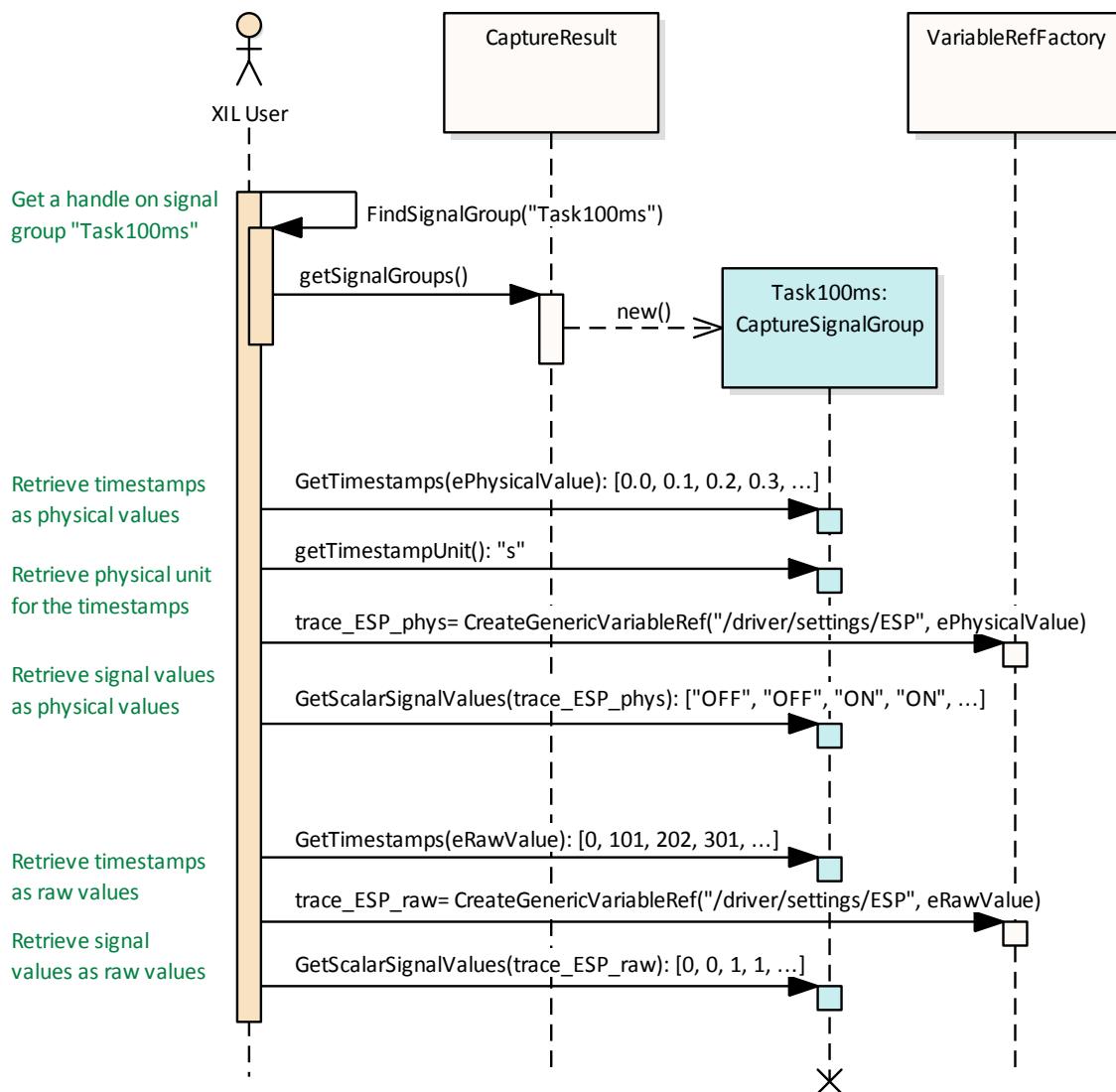


Figure 135: Retrieve physical and raw values from a CaptureResult

As shown in the figure, not only the signal values but also the timestamps can be retrieved both as physical values and as raw values. For the timestamps, the representation mode is selected via a parameter of the `GetTimestamps` method. For the signal values, however, the required representation mode is set as property of the `VariableRef` objects.

The physical timestamp values in this example are float values that represent the time in seconds. The corresponding raw values are integers indicating the tick count of the measuring clock.

The physical values of the signal trace in the example are human readable character strings. The corresponding raw values are integers. Such a signal trace is the result of capturing a kind of enumeration variable, i.e. a variable that uses a `TexttableCompuMethod` for the conversion from raw to physical values. See [Metadata on Conversion Methods](#) for details on the relationship between raw and physical values and conversion methods.

The data type of a signal trace as well as its physical unit, its conversion method and other metadata can be obtained from the `CaptureSignalGroup`. Please find examples on how

to obtain such kind of metadata in chapter [Obtaining Information on Available Model Variables, Tasks and their Properties](#)

The examples in this section demonstrate the usage of the rich metadata retrieval capabilities of the `MAPort`. Note that the available variables and their properties depend on the simulation model. The same also applies to the available tasks. Therefore a loaded simulation model is a prerequisite for metadata retrieval, i.e. the `MAPort` must be at least in `eSIMULATION_STOPPED` state.

The complete list of variable names accessible through the `MAPort` can be obtained from the `VariableNames` property. Next, various information such as data type, array size and physical unit can be individually retrieved for each variable. It is also possible to check whether a variable is readable or writeable within the various `MAPort` states. [Figure 139](#) shows an example sequence for this.

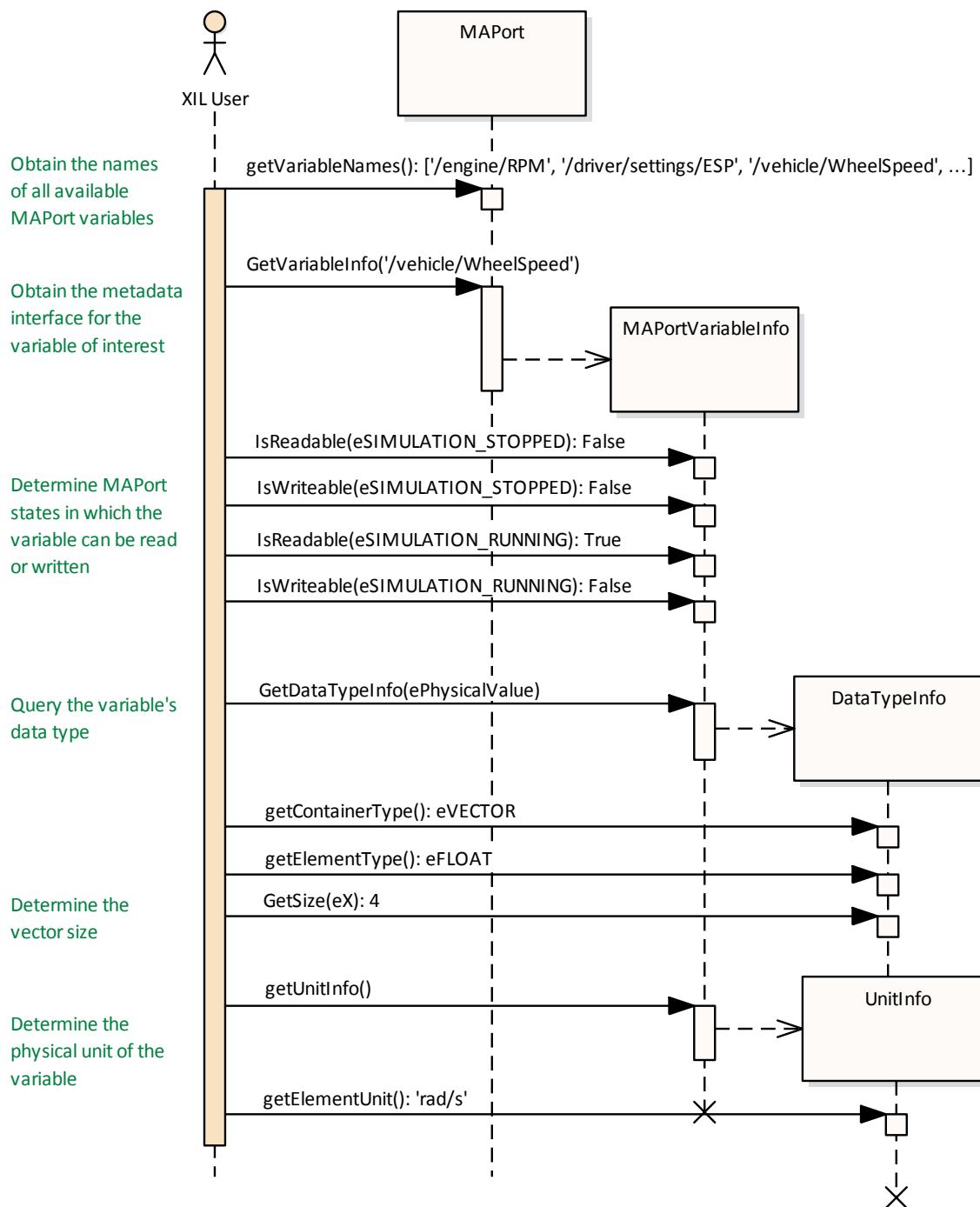


Figure 139: Metadata query on simulation variables

Although the data type query in the example shown is only for the physical variable values, it is also possible for the raw values (see the following example). The depicted array size query only works for non-scalar variables (e.g. vector and matrix variables), it is not possible for scalar variables.

The second example sequence, that is shown in [Figure 140](#), demonstrates the query of a variable's conversion method. This is the calculation rule that the XIL server uses to determine the physical values from the raw values, i.e. the values used internally in the simulator.

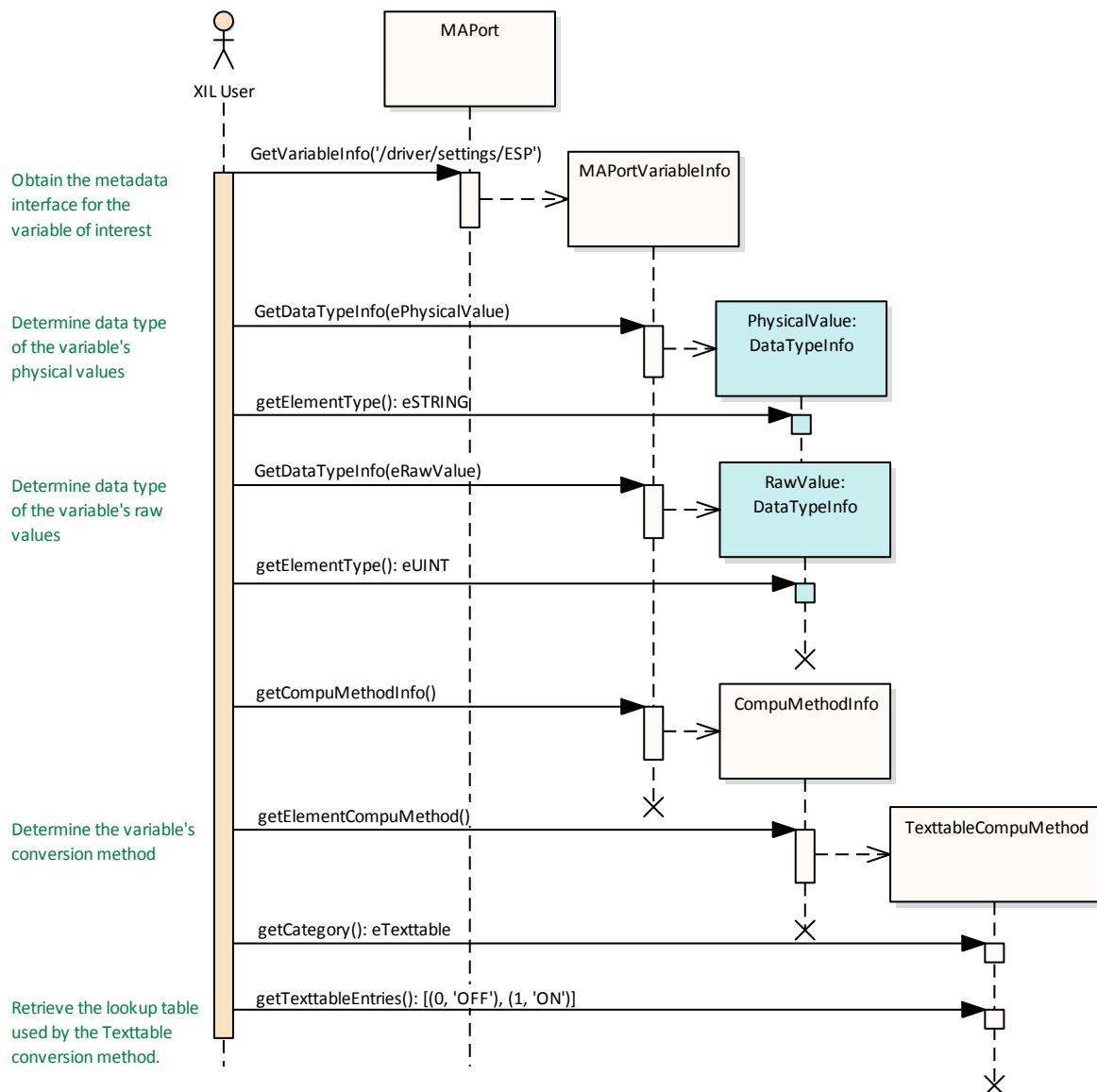


Figure 140: CompuMethod query including value set of TexttableCompuMethods

Depending on the category of the conversion method, various parameters can be retrieved. The depicted example uses a variable with `TexttableCompuMethod`. This conversion method is commonly used to implement enumeration-like variables. Such variables have a very small and discrete value set, where a lookup table maps the human readable character strings (physical values) to integer values (raw values). As shown in the sequence, this mapping and thus the permitted string and integer values can be queried.

The previous examples of metadata queries focus on simulation variables. However, the `MAPort` also allows you to retrieve the available tasks and their properties. This is demonstrated by the sequence in [Figure 141](#). The sequence also shows how to obtain the name and type of a certain task and its cycle time. However, the cycle time can only be queried for timer-driven tasks.

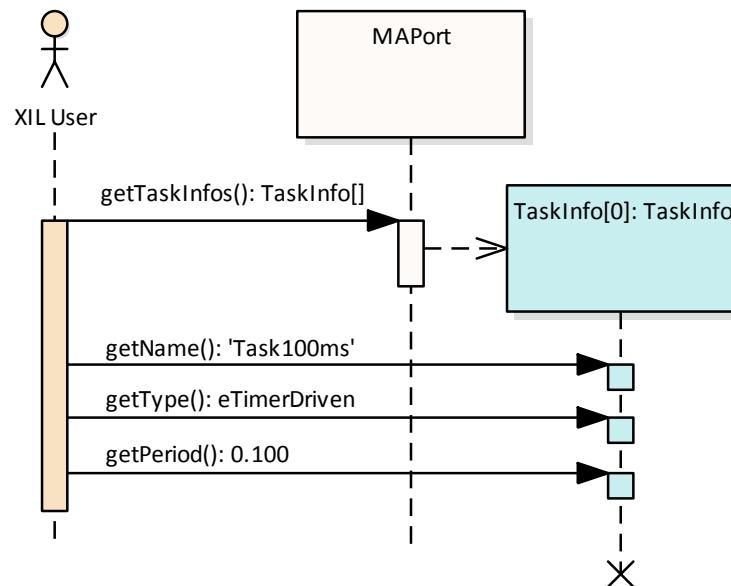


Figure 141: Metadata query on execution tasks in the simulation model

The functionality for metadata retrieval on tasks is very useful, because various methods of the API require a task name as parameter. In particular, a task name is necessary for the creation of Capture objects.

Note: Due to the fact that the adaptation of a simulation model for a specific XIL simulator is vendor specific, the following properties are also vendor specific and may differ on different XIL simulators:

- the list of available variable names
- readability, writability and data type of model variables
- the list of task names
- the timing raster of the tasks (can also depend on the step-size of the simulation model)

Reading & Writing Model Variables. Although the examples there illustrate the metadata query for simulation model variables, they can also be applied to metadata of signal traces in a CaptureSignalGroup.

5.2 MODEL ACCESS PORT

5.2.1 USER CONCEPT

5.2.1.1 GENERAL

The Model Access port is the central interface for managing access to the simulation model that is executed on the XIL simulator. This port provides read and write access to the simulation model, set up of capturing, stimulus generation and target script execution, as well as metadata retrieval for model variables and tasks.

The MAPort package is related to the packages Common:Capturing, Common:SignalGenerator and Common:CaptureResult. They are not sub-packages of MAPort as they are also used by the ECUMPort and the ECUCPort.

5.2.1.2 MAPORT INTERFACE

The class diagram in [Figure 136](#) provides an overview of the MAPort interface, its methods and properties as well as other important interfaces closely related to it.

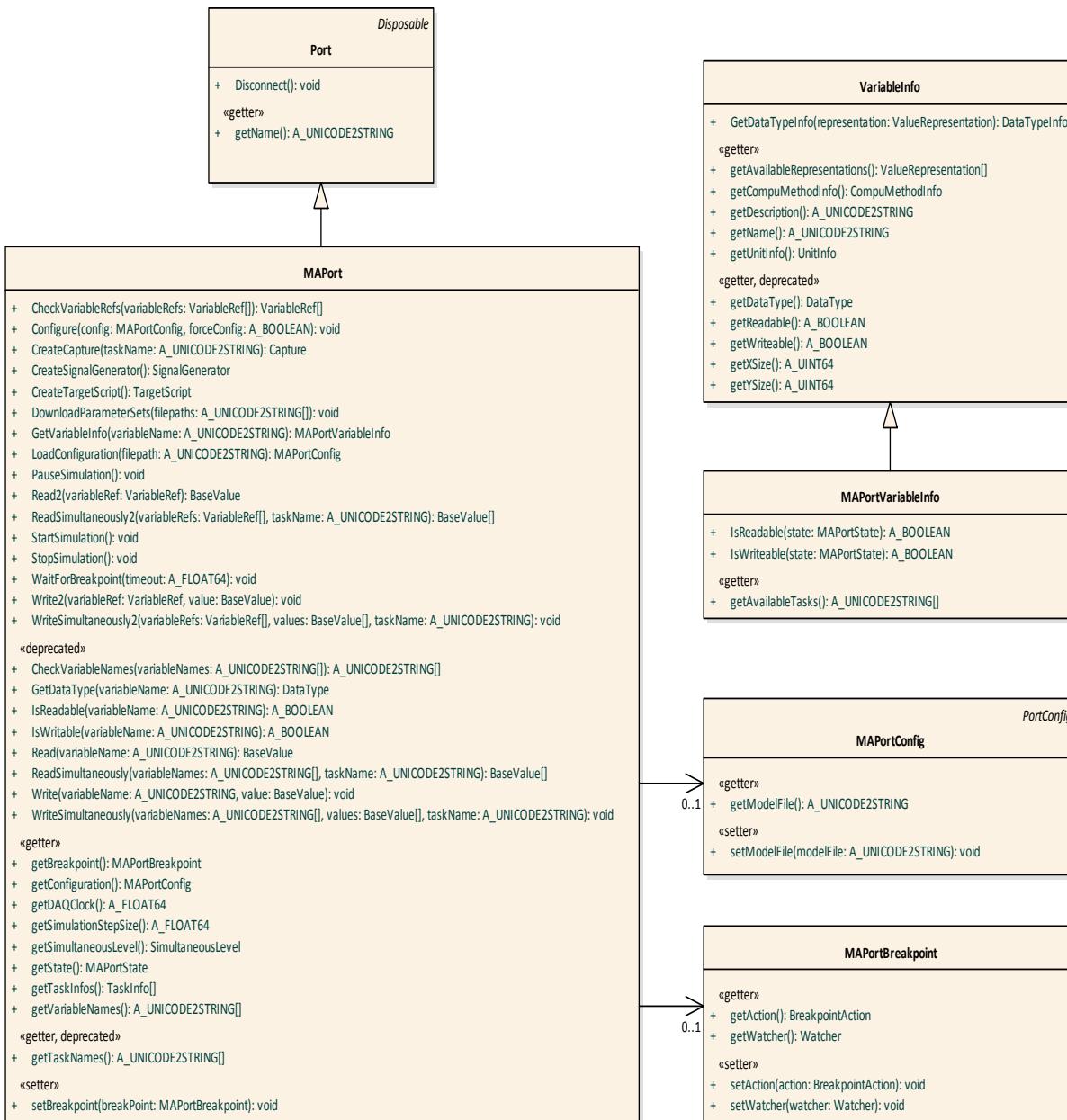


Figure 136: MAPort and associated interfaces

On the one hand, the `MAPort` interface provides general functionality like for example functionality to get information about available model variables, their readability and writability and to read and write model variables. On the other hand, it provides initialization functionality for creating `Capture` and `SignalGenerator` instances as well as `TargetScript` instances.

5.2.1.3 STATES OF THE MAPORT

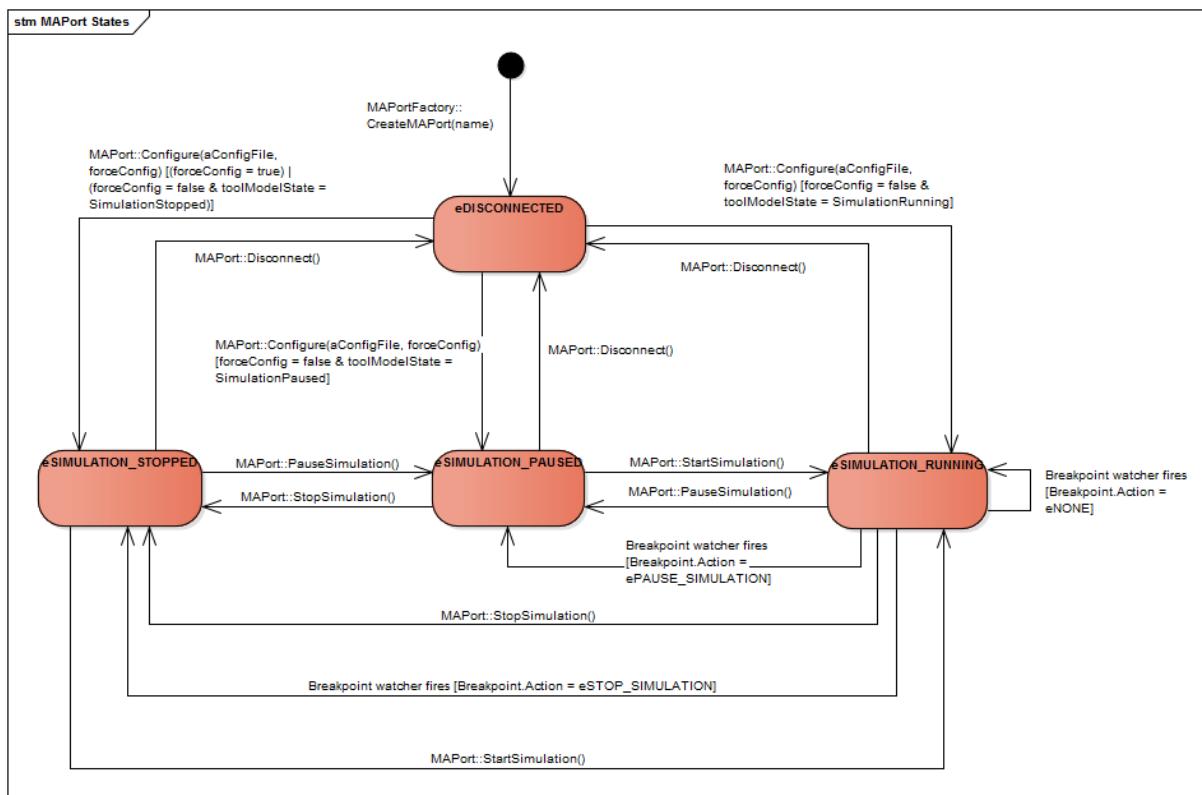


Figure 137: MAPort state diagram

Figure 137 shows the state diagram of the MAPort. There are four states, eSIMULATION_RUNNING, eSIMULATION_PAUSED, eSIMULATION_STOPPED, and eDISCONNECTED. After creation, the MAPort instance is always in state eDISCONNECTED. Table 45 gives an explanation of the states.

Table 45 States of the MAPort

State	Description
eDISCONNECTED	Connection to simulator is not established. Variable access, capturing and signal generation are not possible.
eSIMULATION_RUNNING	Simulator connection established. Execution of the simulation model is running. Captures and SignalGenerators associated with the MAPort are running.
eSIMULATION_PAUSED	Simulator connection established. Execution of the simulation model is paused. Captures and SignalGenerators associated with the MAPort are frozen.
eSIMULATION_STOPPED	Simulator connection established. Simulation is stopped, i.e. execution of the simulation model has not been started yet or has been finished or aborted.

Whether a method of the `MAPort` interface may be called or not depends on the `MAPort` state. [Table 46](#) specifies the states for each method in which it may be called.

Table 46 Allowed states for the MAPort methods

		eDISCONNECTED	eSIMULATION_STOPPED	eSIMULATION_PAUSED	eSIMULATION_RUNNING
Method	CheckVariableNames		x	x	x
Method	Configure	x			
Method	CreateCapture		x	x	x
Method	CreateSignalGenerator		x	x	x
Method	CreateTargetScript		x	x	x
Method	Disconnect		x	x	x
Method	DownloadParameterSets		x		
Property	getBreakpoint		x	x	x
Property	getConfiguration	x	x	x	x
Property	getDAQClock		x	x	x
Method	GetDataType		x	x	x
Property	getSimultaneousLevel		x	x	x
Property	getSimulationStepSize		x	x	x
Property	getState	x	x	x	x
Property	getTaskInfos		x	x	x
Property	getTaskNames		x	x	x
Method	GetVariableInfo		x	x	x
Property	getVariableNames		x	x	x
Method	IsReadable		x	x	x
Method	IsWritable		x	x	x
Method	LoadConfiguration	x	x	x	x
Method	PauseSimulation		x		x
Method	Read		x	x	x
Method	ReadSimultaneously		x	x	x
Property	setBreakpoint		x	x	x
Method	StartSimulation		x	x	

Method	StopSimulation			x	x
Method	WaitForBreakpoint			x	x
Method	Write		x	x	x
Method	WriteSimultaneously		x	x	x

State transitions only take place, if all preconditions are fulfilled and no error occurs during execution of the state changing method. Otherwise the state is not changed, i.e. the state remains the same as before the method was called. Methods that trigger a state change will throw an exception if the state change could not be performed successfully.

5.2.2 USAGE OF MAPORT

In this chapter, the usage of the `MAPort` is described by means of sequence charts. It is explained how to set up the port, how to retrieve metadata on the available model variables and how to read and write model variables.

For examples of how to set up and use a signal generator, please refer to chapter [Usage of SignalGenerator \(Stimulating Model Variables\)](#).

Details on the creation, configuration and usage of `Capture` instances can be found in the chapter [Usage examples for Capture and Capture Result](#).

5.2.2.1 CREATION AND CONFIGURATION

Instances of `MAPort` are created by calls to method `CreateMAPort` of the `MAPortFactory` object. It returns a `MAPort` instance with the name specified as parameter. The `MAPortFactory` object can be obtained from the `Testbench` object that provides a corresponding property. The port creation and the subsequent configuration process are depicted in [Figure 138](#).

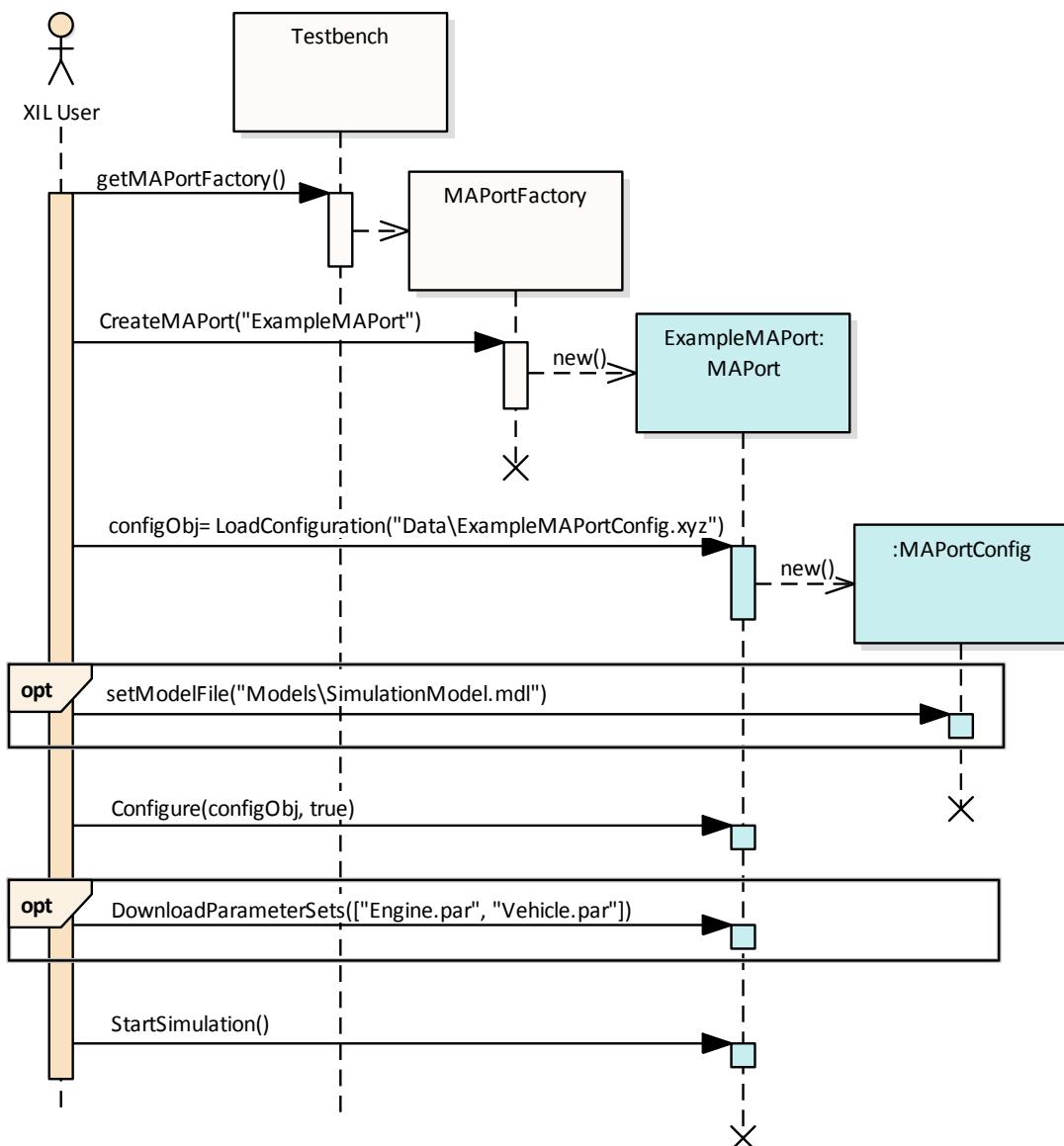


Figure 138: Process of MAPort creation and configuration

Configuration of the new port instance is a two-step process. First a vendor specific configuration file has to be loaded via method `LoadConfiguration`. This creates a `MAPortConfig` object from the content of the configuration file. Besides other, vendor specific port settings this file particularly specifies the model file to be executed on the simulator. The `MAPortConfig` object that is returned by `LoadConfiguration` provides a property for the model file path. This allows you to optionally change the reference to the model file before the configuration is loaded into the simulator.

The second step of configuration is calling the `Configure` method and passing the `MAPortConfig` object created in step one. This establishes a connection to the simulator and checks whether a simulation model has already been loaded or even started. If no model is loaded, the simulation model passed in `MAPortConfig` is loaded and the state is set to `eSIMULATION_STOPPED`. This behavior can also be forced by setting the `Configure` method's parameter `forceConfig` to `true`, as shown in the figure. In this case a running simulation model will be stopped and replaced by the specified one.

Note: If a simulation has already been configured on the simulation tool or hardware when the `Configure` method is called and the `forceConfig` parameter has the value `false`, the current simulation, especially the simulation state, is not affected in any way. This also applies if the passed configuration differs from the currently active one. However, `Configure` yields an exception in the latter case. On successful execution the `MAPort` reflects the current simulation state that can be `eSIMULATION_STOPPED`, `eSIMULATION_RUNNING` or `eSIMULATION_PAUSED`.

Having successfully completed the configuration steps described, the simulation model is now ready for execution. But before starting the simulation one can download parameter sets to the simulator using the method `DownloadParameterSets()`. This step is also included in [Figure 138](#). However, it is optional. It depends on the simulation model and use case.

Parameter sets are stored in parameter set files and contain a list of parameter names and values to be assigned to the corresponding simulation variables. Parameter sets facilitate management and adjustment of dependent or related simulation parameters. They are commonly used to parametrize a generic simulation model for a specific purpose or to initialize the model with specific, but consistent start values.

The method `DownloadParameterSets()` frees the user from the need to adjust a large set of related parameters individually and from the need to know about the internal format details of vendor specific parameter set files. Note that this method may be called in simulation state `eSIMULATION_STOPPED` only.

Finally, when all configuration work is completed, the simulation is started with a call of the `StartSimulation` method.

5.2.2.2 OBTAINING INFORMATION ON AVAILABLE MODEL VARIABLES, TASKS AND THEIR PROPERTIES

The examples in this section demonstrate the usage of the rich metadata retrieval capabilities of the `MAPort`. Note that the available variables and their properties depend on the simulation model. The same also applies to the available tasks. Therefore a loaded simulation model is a prerequisite for metadata retrieval, i.e. the `MAPort` must be at least in `eSIMULATION_STOPPED` state.

The complete list of variable names accessible through the `MAPort` can be obtained from the `VariableNames` property. Next, various information such as data type, array size and physical unit can be individually retrieved for each variable. It is also possible to check whether a variable is readable or writeable within the various `MAPort` states. [Figure 139](#) shows an example sequence for this.

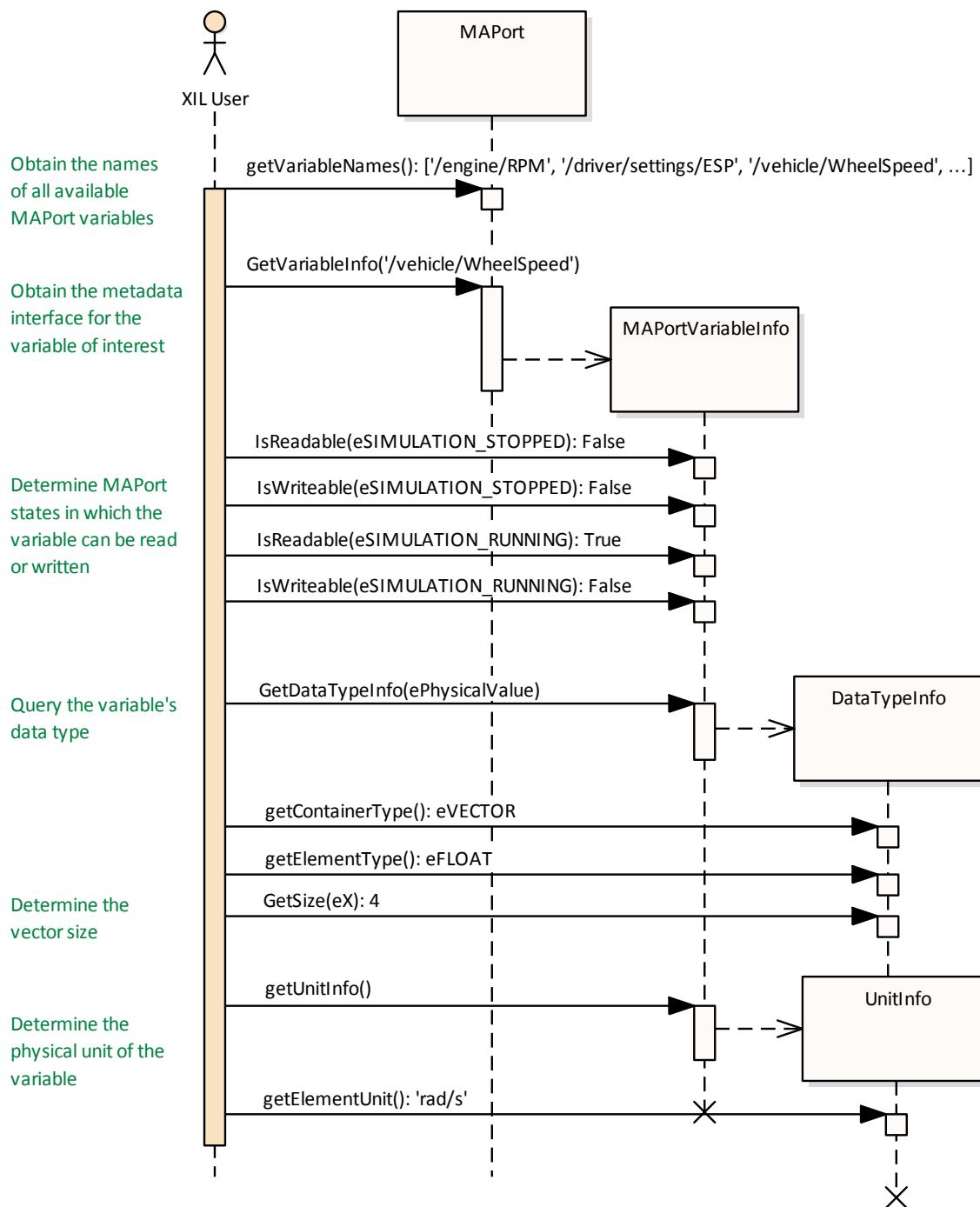


Figure 139: Metadata query on simulation variables

Although the data type query in the example shown is only for the physical variable values, it is also possible for the raw values (see the following example). The depicted array size query only works for non-scalar variables (e.g. vector and matrix variables), it is not possible for scalar variables.

The second example sequence, that is shown in [Figure 140](#), demonstrates the query of a variable's conversion method. This is the calculation rule that the XIL server uses to determine the physical values from the raw values, i.e. the values used internally in the simulator.

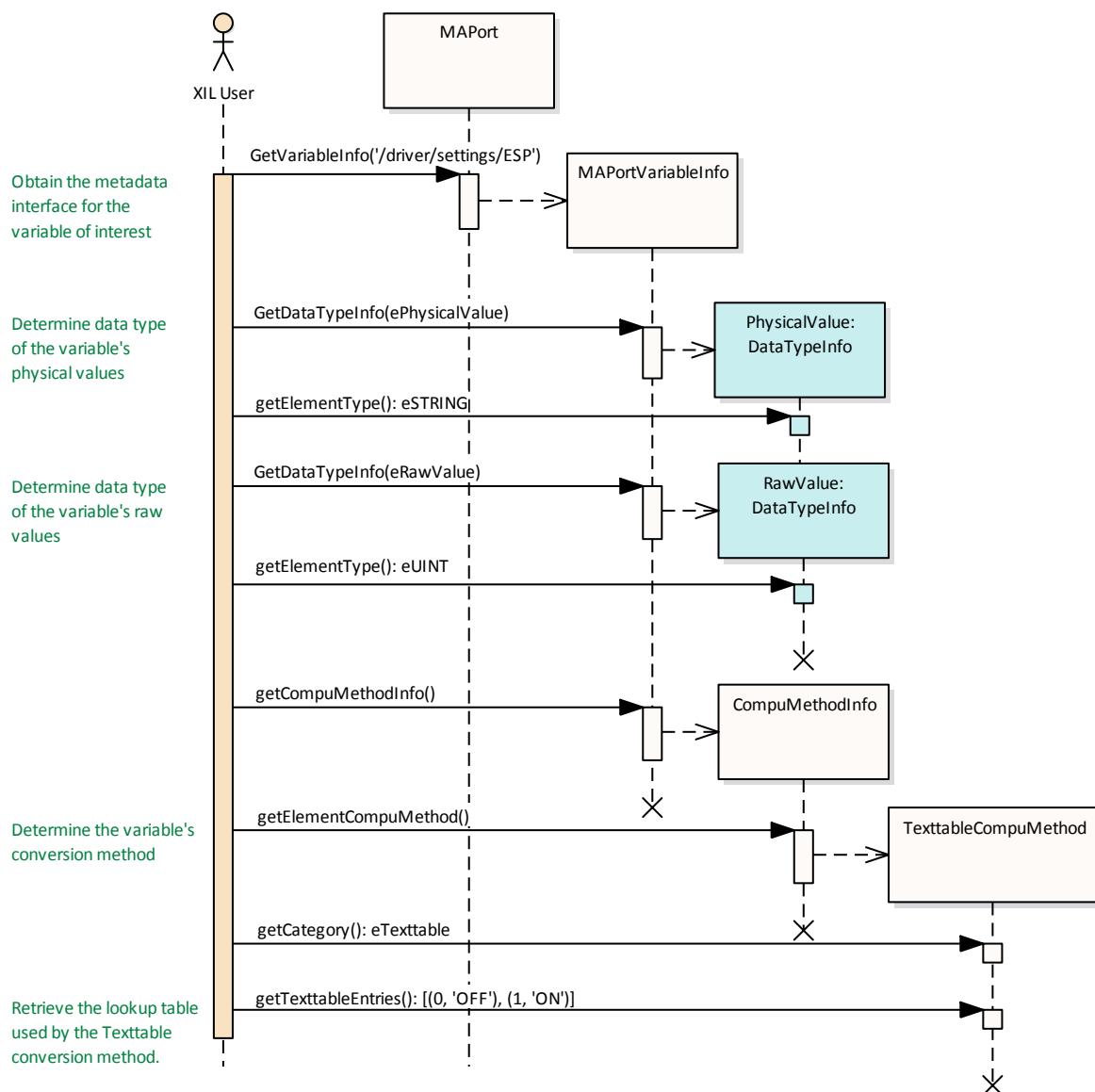


Figure 140: CompuMethod query including value set of TexttableCompuMethods

Depending on the category of the conversion method, various parameters can be retrieved. The depicted example uses a variable with `TexttableCompuMethod`. This conversion method is commonly used to implement enumeration-like variables. Such variables have a very small and discrete value set, where a lookup table maps the human readable character strings (physical values) to integer values (raw values). As shown in the sequence, this mapping and thus the permitted string and integer values can be queried.

The previous examples of metadata queries focus on simulation variables. However, the `MAPort` also allows you to retrieve the available tasks and their properties. This is demonstrated by the sequence in [Figure 141](#). The sequence also shows how to obtain the name and type of a certain task and its cycle time. However, the cycle time can only be queried for timer-driven tasks.

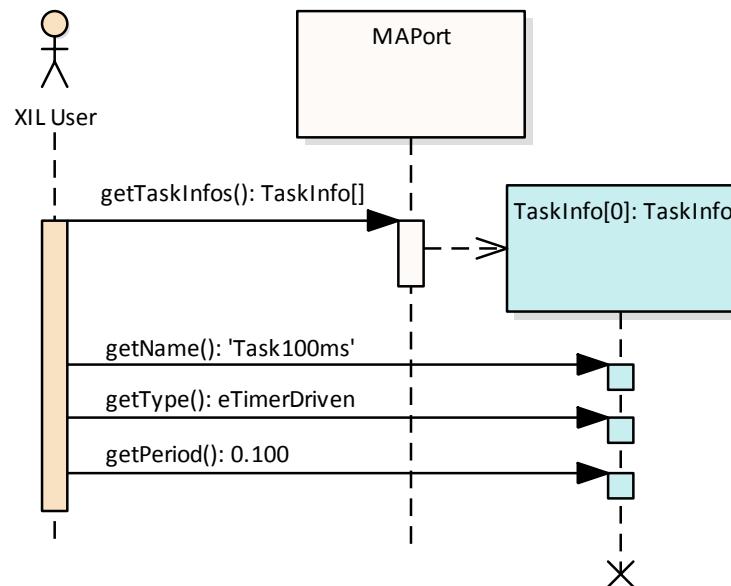


Figure 141: Metadata query on execution tasks in the simulation model

The functionality for metadata retrieval on tasks is very useful, because various methods of the API require a task name as parameter. In particular, a task name is necessary for the creation of Capture objects.

Note: Due to the fact that the adaptation of a simulation model for a specific XIL simulator is vendor specific, the following properties are also vendor specific and may differ on different XIL simulators:

- the list of available variable names
- readability, writability and data type of model variables
- the list of task names
- the timing raster of the tasks (can also depend on the step-size of the simulation model)

5.2.2.3 READING & WRITING MODEL VARIABLES

The examples in this section illustrate the general approach and some specific aspects of reading and writing variables of the simulation model. It is assumed that the MAPort has already been configured as explained in [Creation and Configuration](#) and that a simulation model is running.

The sequence in [Figure 142](#) demonstrates the general procedure for reading and writing model variables. As one can see, using a GenericVariableRef object to identify the affected model variable is a generic approach. It works for both scalar and non-scalar variables.

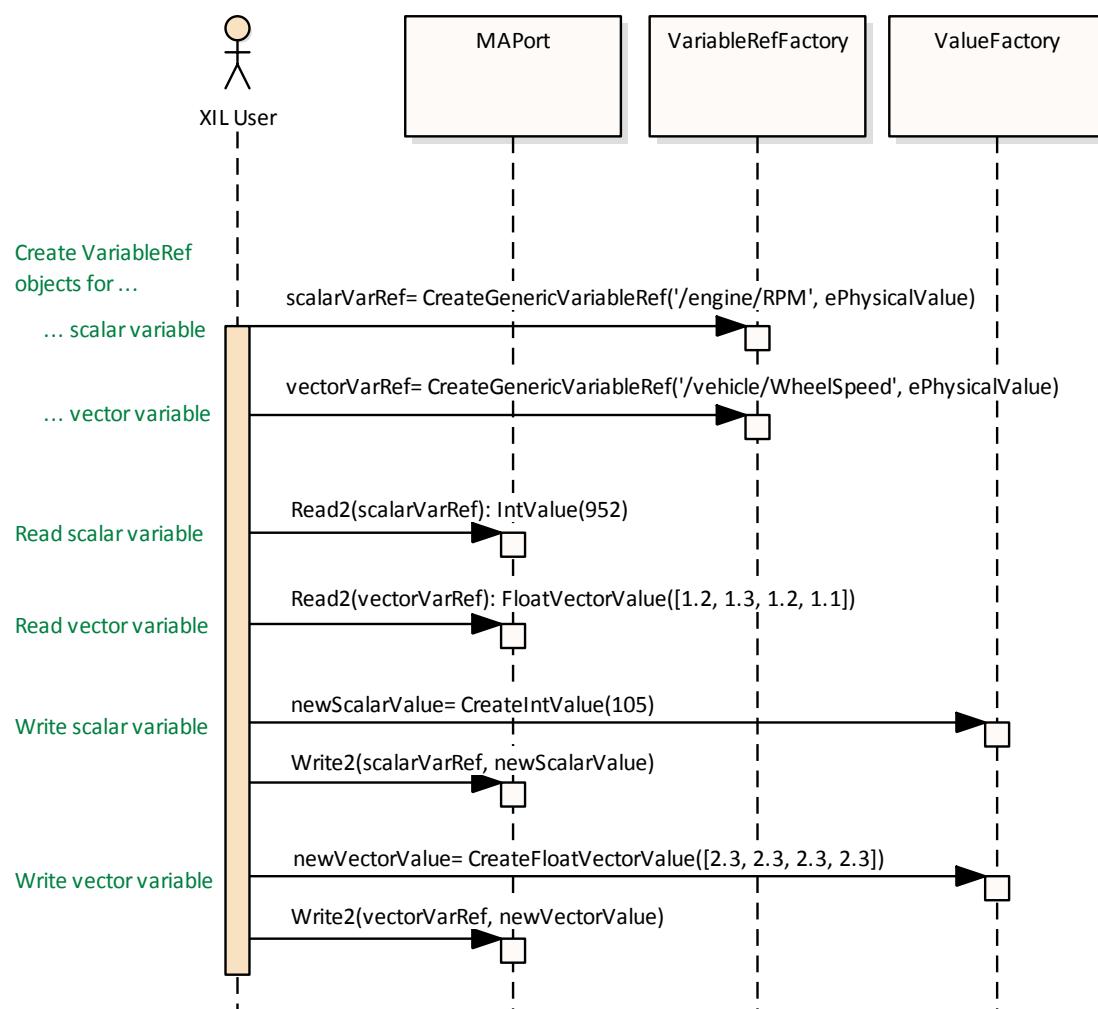


Figure 142: Reading and writing model variables

The VariableRefFactory and the ValueFactory shown in the sequence chart can be obtained from the correspondingly named properties of the Testbench interface.

When using GenericVariableRef objects to access non-scalar variables, the complete value structure (e.g. all vector elements) is always read or written. If only a single element of a vector variable shall be read or written, a VectorElementRef must be used instead, which additionally specifies the element index. This use case is depicted in [Figure 143](#).

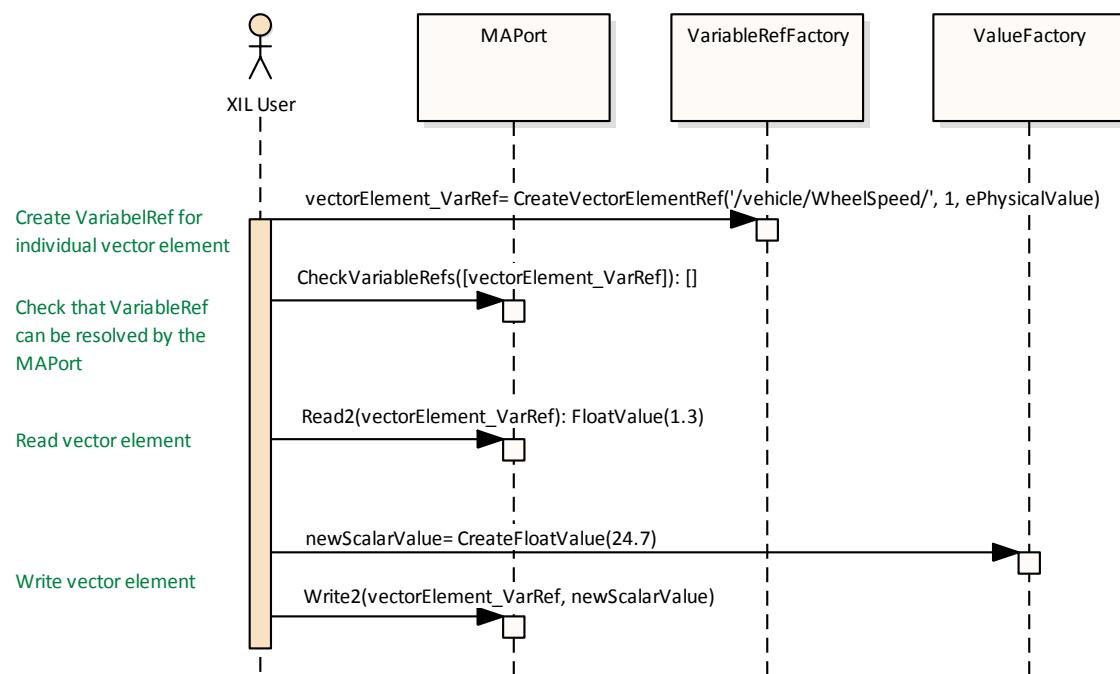


Figure 143: Reading and writing an individual vector element

The example in the previous figure demonstrates the access to a vector element. If a matrix element is to be accessed instead, a `MatrixElementRef` object would have to be used.

Note the usage of method `CheckVariableRefs`. It checks whether the passed `VariabelRef` objects can be resolved by the `MAPort`. If the returned list is empty, no invalid `VariabelRef` objects have been found. This way it can be used to avoid failing read and write calls later on. The check includes the index specification of `VectorElementRef` and `MatrixElementRef` objects. Thus it is particularly useful for this use case.

In all previous examples, only the physical value of the model variables was read or written. However, the `MAPort` also allows you to read and write the value representation that is used internally by the simulator. This value representation, referred to as raw value, might differ from the representation as physical value. The sequence in [Figure 144](#) focuses on the selection of the required value representation and demonstrates how to read raw values.

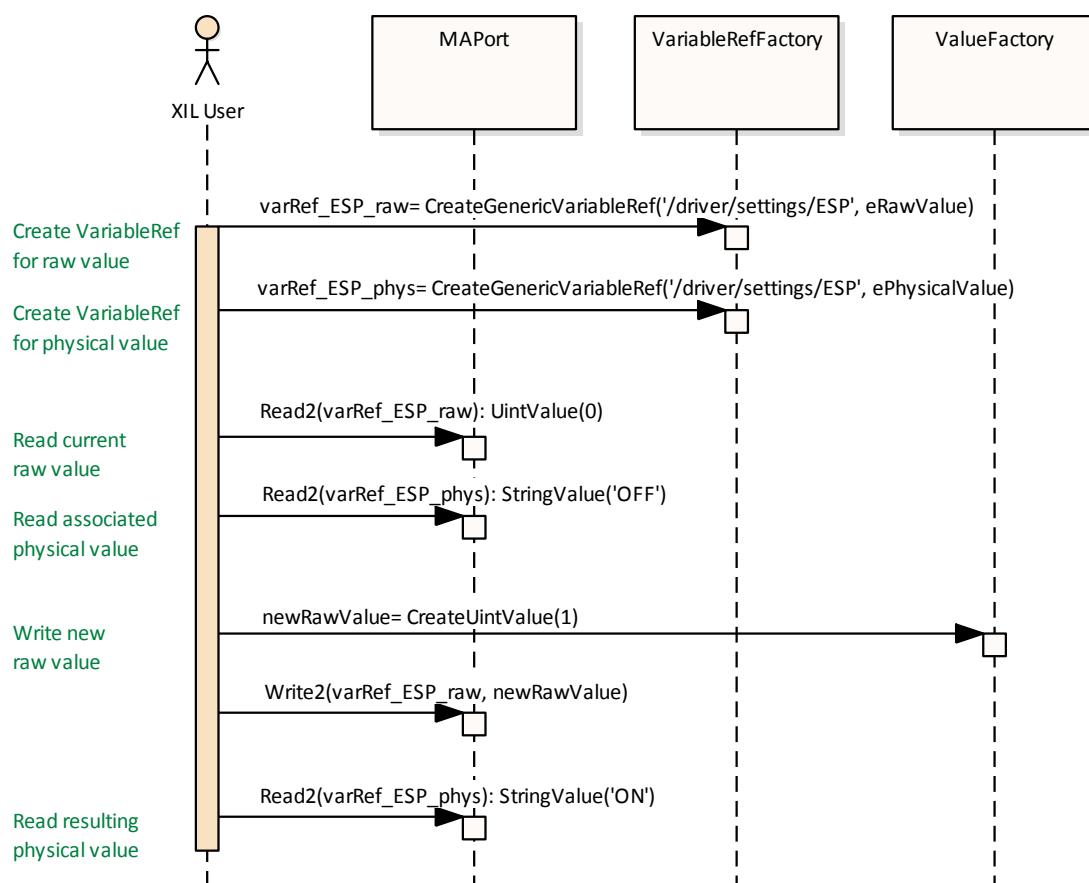


Figure 144: Reading and writing a variable's raw value

As shown in the figure, the required value representation is specified via a corresponding property of the `VariableRef` objects. Note that the data type of the value passed to the `Write` method depends on the selected representation mode and might differ for physical and raw value access.

The used example variable has human readable character strings as physical values and integers as raw values. Such variables are commonly used to implement enumeration-like variables. (see the example in [Figure 140](#) for the retrieval of the mapping between raw and physical values).

The `Read2()` and the `Write2()` methods are executed synchronously. The standard does not define any timing constraints for the read and write access to model variables. However, the `MAPort` interface provides two methods, `ReadSimultaneously2()` and `WriteSimultaneously2()`. These methods operate on a list of variables whereby they ensure that reading and writing of the variables take place in the same simulation step or in the same cycle of a given task. The level of simultaneously reading or writing is vendor and system specific and can be retrieved by the `SimultaneousLevel` property of the `MAPort`.

5.2.2.4 STATE DEPENDENCY OF VARIABLES' WRITEABILITY AND READABILITY

Not all variables of a MAPort must be readable or writeable. That is why the user can determine the variable's readability or writeability by the MAPort's MAPortVariableInfo interface (see [Figure 145](#)).

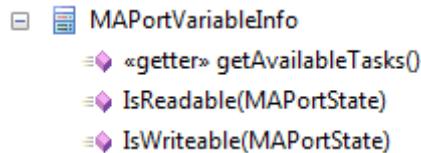


Figure 145: MAPortVariableInfo methods

Please note that a variable's readability or writeability might also depend on the MAPort state. If a variable is not writeable in a certain state, an eMA_COULD_NOT_WRITE_VARIABLE exception is thrown by the MAPort.Write() method when it is called in that state.

The dependency of the writeability on the simulation state is useful for the following reasons:

- Some variables influence a model only during initialization, e. g. initial values for discrete or continuous state variables.
- Variables can be used for computation of other variables, e. g. the vehicle's total mass m as the sum of the vehicle mass m_v and the additional load m_l . Thus, m needs to be calculated only once during initialization, and not again and again in every simulation step.

In the FMI standard [8] this point is addressed by a concept called variability. This is a classification of variables based, among other aspects, on the writeability of variables and their dependency on the simulation status.

The categories "fixed" and "tunable" are two examples for a variable's variability. The value of "fixed" variables cannot be changed after initialization of the simulation whereas the value of "tunable" variables can be modified during and after initialization.

Thus, the variability of an FMI variable determines whether or not the variable is writable in a given MAPort state and therefore whether the MAPortVariableInfo.IsWriteable() method returns TRUE or FALSE. See the following table for the writeability of a variable with respect to their FMI variability and the MAPort state.

Table 47 Writeability of FMI variables with respect to their variability and the MAPort state

Variability according to FMI	eDISCONNECTED	eSIMULATION_STOPPED	eSIMULATION_PAUSED	eSIMULATION_RUNNING
constant	False	False	False	False
fixed	False	True	False	False
tunable	False	True	True	True

Please note that in the context of FMI, only tunable variables can be stimulated using a SignalGenerator.

5.2.2.5 RELATION BETWEEN MAPORT AND CAPTURING / SIGNALGENERATOR

The run of the simulation (MAPort state) affects the behavior of capturing and the signal generator. This principle is shown in [Figure 146](#).

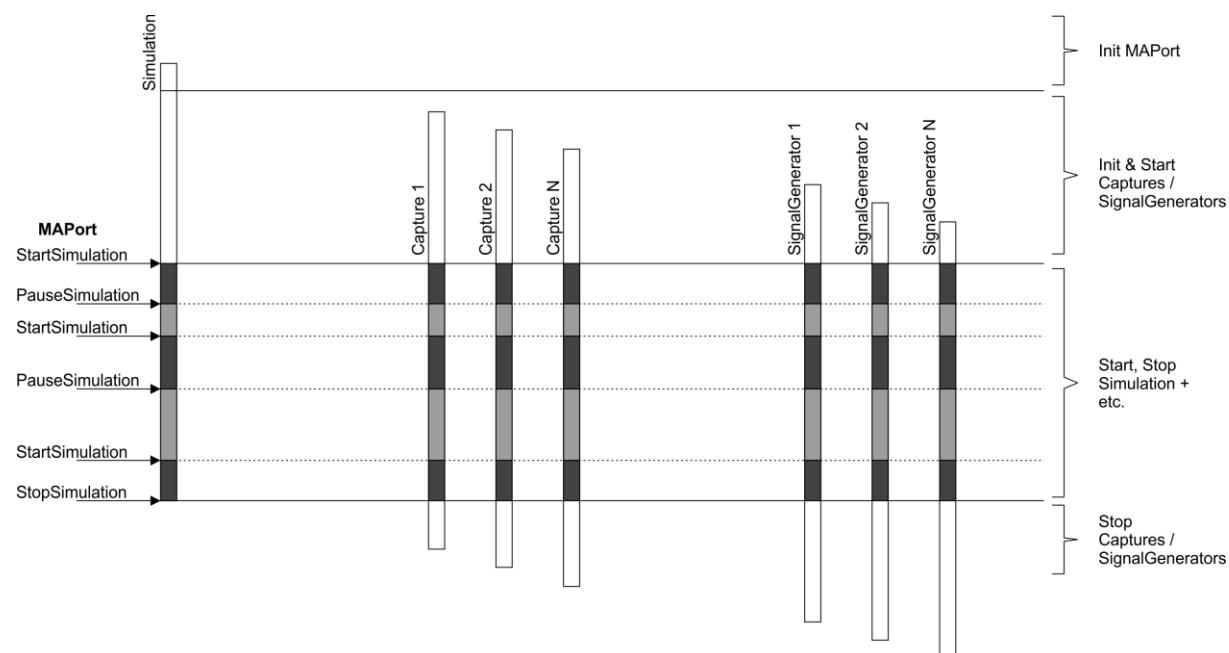


Figure 146: Relation between simulation and capturing / stimulation

If a simulation pauses at the MAPort, then the simulation time freezes and so does the time in the corresponding Capture and SignalGenerator objects. The Capture objects are "frozen" (grey illustrated bars in the figure) and do not record data. They remain in their current state (immediately before the break). And the SignalGenerator objects are also

"frozen" (grey illustrated bars in the figure) and do not stimulate data. They remain in their current state (immediately before the break).

The behavior after the break is as before the break, when the simulation is started again (black illustrated bars in the figure). The time then continues without interruption in the simulation, in the measurement recording and in the stimulation.

Capture and SignalGenerator objects can be initialized and started before StartSimulation of the MAPort is called. However, in this case no capturing or stimulation takes place yet. With StartSimulation, these capture and signal generator objects run synchronously (simultaneously) and record or stimulate data (from the first simulation step).

Trigger conditions can only be evaluated while the simulation is running. This means, for example, with triggered capturing (and not yet started simulation), the state eACTIVATED is always taken after Capture.Start (), even if the trigger condition had already been formally fulfilled.

Note: If MAPort.StartSimulation is called again after the simulation has been stopped (i.e. MAPort.StopSimulation has been called) and any of the SignalGenerator and Capture objects used in the previous simulation run has not been stopped, the behavior is undefined. For a defined behavior it is necessary to explicitly ensure that all Capture and SignalGenerator objects have been stopped before calling MAPort.StartSimulation again.

5.2.2.6 PAUSING AND STEPWISE EXECUTION OF THE SIMULATION

There are use cases especially with MIL and SIL scenarios that require the simulation to be interrupted at a certain point and resumed from that point later on. A similar and very common requirement of MIL and SIL simulators is the possibility of client controlled stepwise execution of simulations.

Both requirements are addressed by the MAPort whose state chart provides the eSIMULATION_PAUSED state for this purpose (see chapter 5.2.1.3). Pausing the simulation, i.e. entering the eSIMULATION_PAUSED state, can be triggered in two different ways. Either the client directly calls the PauseSimulation method of the MAPort or it sets a breakpoint.

A breakpoint automatically interrupts the simulation after a specified duration or when certain simulation variables meet a specified condition. The breakpoint is set via the MAPort's Breakpoint property. When using a breakpoint, the client can wait for the simulation to be paused by using the WaitForBreakpoint method of the MAPort. In order to resume a paused simulation, the StartSimulation method must be called.

Stepwise execution of a simulation requires the simulator to automatically pause the simulation each time the specified amount of simulation time has elapsed or alternatively the specified number of simulation steps has been executed. Therefore, a breakpoint containing a DurationWatcher must be set before starting or resuming the simulation. The DurationWatcher determines the duration of the next simulation interval. As already indicated, this duration can be specified in seconds of simulation time (passing a TimeSpanDuration) or as number of simulation steps (passing a CycleNumberDuration). The system specific duration of a single simulation step (i.e. the

minimal increment of the discrete simulation time), can be retrieved via the MAPort's SimulationStepSize property, if the simulation uses a fixed step size solver.

In case of stepwise execution of a simulation it might be necessary to pause the simulation just before executing the very first simulation step, e.g. to read or modify the initial values of simulation variables. This can be realized by calling PauseSimulation in state eSIMULATION_STOPPED, which triggers a direct state transition to eSIMULATION_PAUSED. That means, upon return of PauseSimulation the MAPort is in state eSIMULATION_PAUSED while ensuring that no simulation step has been executed so far. However, the simulation is completely initialized and simulation time is 0 as if method StartSimulation would have been called.

The following figure illustrates the general approach for two use cases. It shows how the simulation is started and automatically paused when the point of operation is reached (e.g. when the initial transient oscillation is completed). From then on, the simulation is stepwise executed which enables the client to precisely place a stimulation and to inspect the simulation's response after each simulation step.

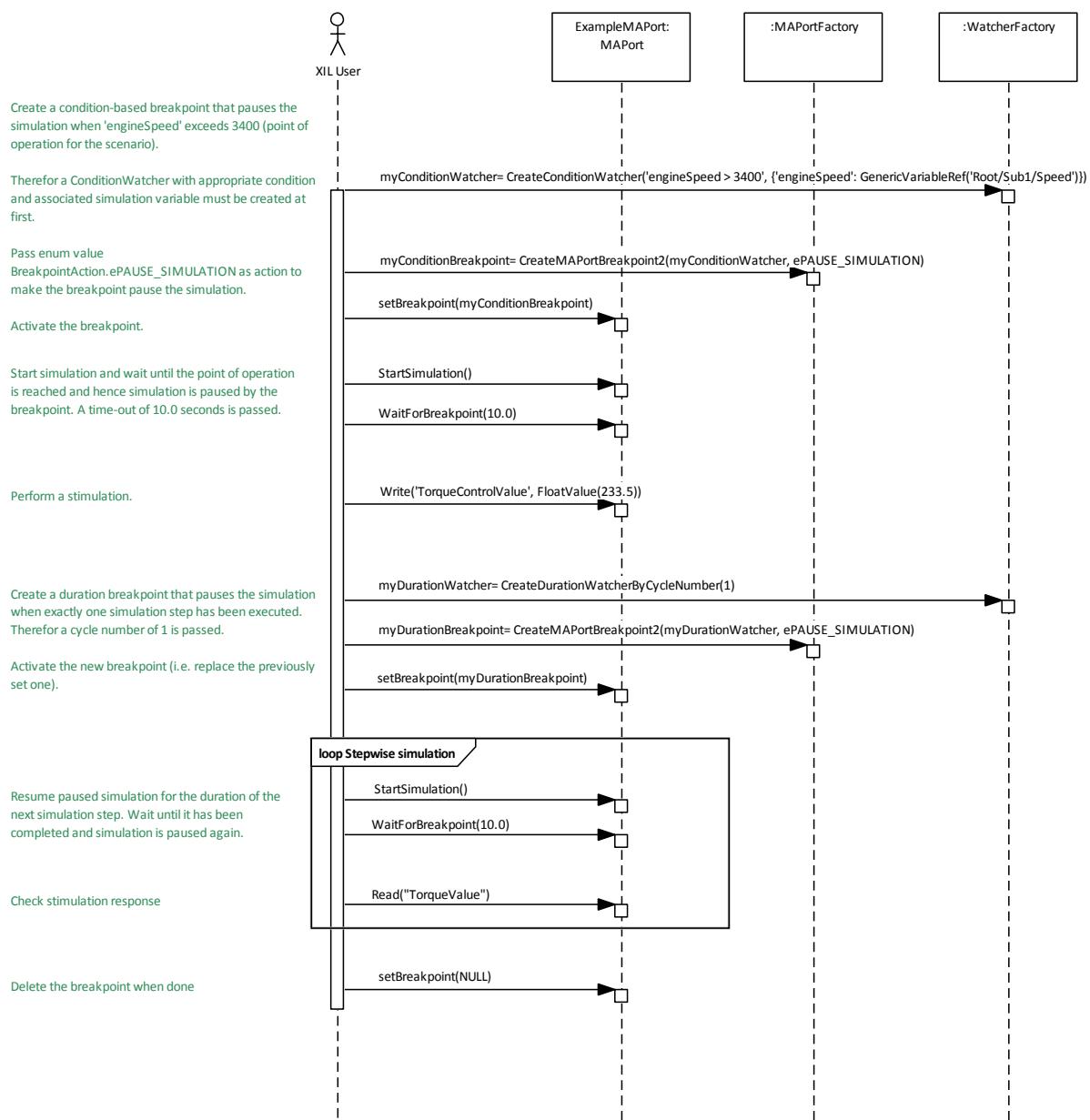


Figure 147: Example for pausing and stepwise executing a simulation

For the depicted example scenario it is assumed that the MAPort has already been successfully configured but the simulation has not been started. In order to get notified when the point of operation is reached the client sets and waits for a breakpoint that contains a ConditionWatcher whose condition represents the point of operation. (Note: Instead of using a breakpoint it is also possible to just start the simulation and then call PauseSimulation after a reasonable time.)

Then, for stepwise simulation, the client replaces the breakpoint by another one containing a DurationWatcher with a CycleNumberDuration of 1. This makes the simulation pause each time a simulation step has been executed. Note that the breakpoint can be reused. There is no need to renew the breakpoint for each resume cycle since breakpoints stay in operation until explicitly removed by the client.

A breakpoint with `DurationWatcher` needs a reference time, i.e. a point in time when the specified duration starts to elapse. This reference time is initially set to the current simulation time when the breakpoint is assigned to the `MAPort`. However, it is updated each time the simulation state changes from `eSIMULATION_STOPPED` or `eSIMULATION_PAUSED` to state `eSIMULATION_RUNNING`. The updated reference time then equals the first timestamp in status `eSIMULATION_RUNNING`.

The update of the breakpoint's reference time is necessary for two reasons. Firstly, the state change may reset the simulation time to 0. Secondly, the same breakpoint may be reused, i.e. trigger multiple times without being reassigned to the `MAPort` each time.

If a duration defined breakpoint has further been assigned `BreakpointAction.eNONE`, its reference time must additionally be updated each time the breakpoint triggers. Thereby the trigger time is set as new reference time. This is necessary to enable the mentioned reuse scenario despite these breakpoints do not cause any simulation state change (they only notify the client of the elapsed duration).

Note: Pausing the simulation might not be supported by real-time simulations (e.g. HIL scenarios). However, the `MAPort`'s breakpoint feature supports two further use cases that are also applicable in real-time scenarios: It can stop the simulation (instead of pausing it) when a specific condition (e.g. error condition) is met. And it allows a client to just wait for (i.e. synchronize on) a specific simulation condition without affecting the simulation state.

The action to be performed by a breakpoint (pause simulation / stop simulation / only client notification) is set via the `Action` property of the `MAPortBreakpoint` interface. There is an enumeration `BreakpointAction` representing the possible actions.

5.3 DIAGNOSTIC PORT

5.3.1 OVERVIEW

The ECU access part of the XIL API accesses an ECU via a D-Server. A diagnostic tool may consist of hardware, software or both that allow for ECU diagnostics. It provides the following functionality:

Diagnostic Tool

- Configuring the diagnostic tool
- Setting the communication mode

ECU

- Starting and stopping the communication explicitly
- Reading and clearing the fault memory
- Reading identification data
- Reading measurement data by short names
- Reading and writing variant coding data
- Reading and writing data from and to the EEPROM by address
- Reading and writing data from and to the EEPROM by alias names
- Executing diagnostic jobs
- Sending hex services

Functional Group

- Starting and stopping the communication explicitly
- Reading and clearing the fault memory
- Reading measurement data by short names
- Sending hex services

The XIL API only communicates with the D-Server, it does not communicate directly with the ECU. It is the task of the D-Server to handle the communication with the ECU and to execute the methods defined by the XIL API. Therefore, the XIL API does not need any knowledge about the interface being used for communication with the ECU (e.g. a CAN interface or a KWP2000 interface), and consequently this does not influence the code accessing the XIL API.

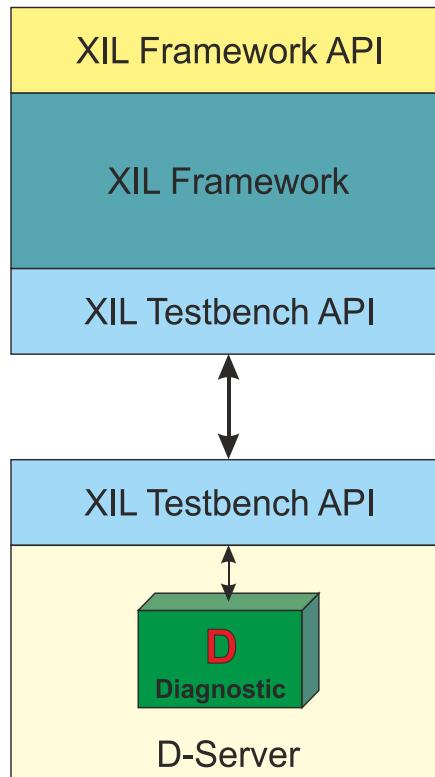


Figure 148: Accessing ECUs via the XIL API for diagnostic

5.3.2 API

The Diagnostic Port API consists of several classes that represent logical components used in the diagnostic use cases—such as the diagnostic tool, the ECU, the fault memory etc.—in an abstract manner. The following chapters describe the classes of the Diagnostic Port API and the relations between them. Furthermore, the context and the usage of these classes is illustrated.

The access to ECUs via the XIL API for Diagnostic is done by the `DiagPort`, which is derived from the class `Port` (see [Figure 149](#)).



Figure 149: ECU DiagPort Class diagram

5.3.2.1 COMMUNICATION MODES

The Diagnostic Port communicates directly with the diagnostic tool and indirectly with one or more ECUs. The client has to consider that there is hardware underneath the diagnostic tool that - depending upon the state of development of the hardware - is more or less robust and reliable. That is why the client may anticipate hardware and communication failures underneath the diagnostic tool. The system and communication failures that are detected by the diagnostic tool are delivered to the Diagnostic Port's client by exceptions.

The Diagnostic Port API allows for setting the communication mode between the diagnostic tool and the ECU. The communication mode is either automatic or explicit. In automatic communication mode the client does not need to call the methods for starting and stopping the communication with the ECU explicitly as this is handled by the diagnostic tool. In explicit communication mode however, the client has to call the methods for starting and stopping the communication explicitly. In automatic communication mode the diagnostic tool has to check whether the communication with the ECU has already been started whenever a client calls a method of the ECU class, the FunctionalGroup class or one of the BaseController classes. If the communication has not been already started, the diagnostic tool has to start and stop the communication on its own responsibility so that the communication status is preserved regardless of the invoked method. The automatic communication mode is the default communication mode. Chapter [Sending Hex Services with Explicit Communication](#) shows how to send a HEX service with explicit communication.

5.3.2.2 ECU

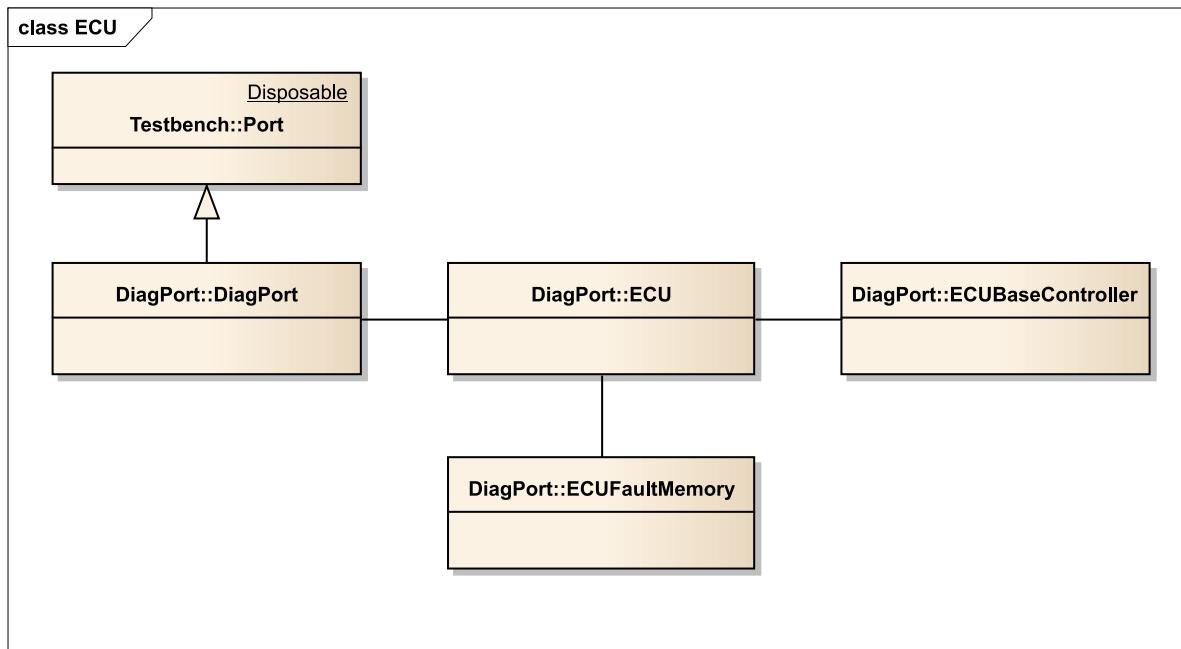


Figure 150: ECU classes

The main classes of the Diagnostic Port API are outlined in the class diagram in [Figure 150](#). The `DiagPort` class is a subclass of the generic `Port` class. A `DiagPort` object can be used to obtain an `ECU` object that is a representative of a real ECU. Therefore, the `ECU` object contains methods for high level use cases such as executing diagnostic jobs, reading variant data or the fault memory for example. The `ECU` object can also be used to receive an `ECUBaseController` object for more sophisticated tasks such as sending HEX services or reading from the EEPROM.

5.3.2.3 FUNCTIONAL GROUPS

A functional group is a group of ECUs that have equal or similar functionality. Mostly, these ECUs can be addressed by a functional address, i.e. a symbolic name for the functional group. For example, all door controlling ECUs of a vehicle may form a functional group. [Figure 151](#) shows the classes that are provided for functional groups by the Diagnostic Port API.

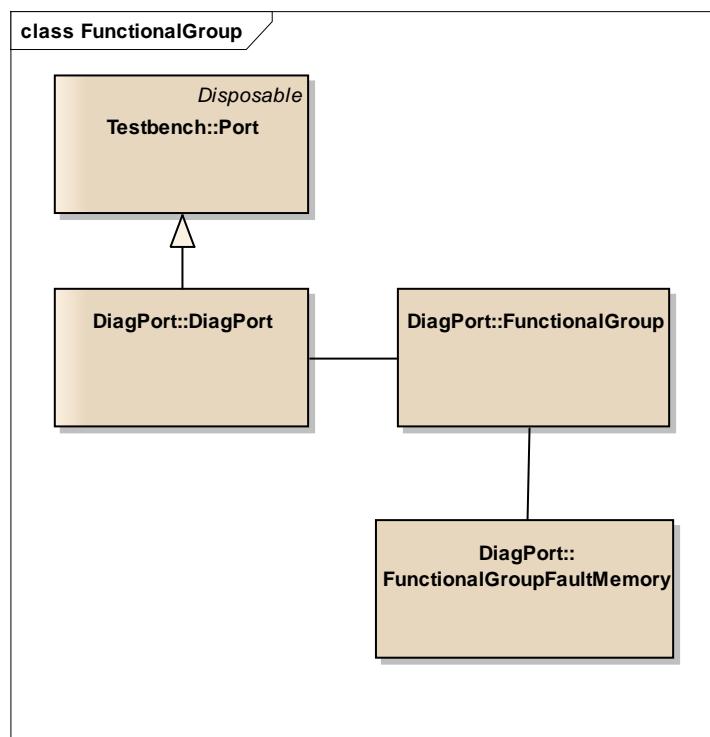


Figure 151: Functional group classes

The Diagnostic Port API contains several classes in order to meet the demands for functional groups. A `DiagPort` object can be used to obtain a `FunctionalGroup` object that is a representative of a functional group of real ECUs. The `FunctionalGroup` object contains methods for high level use cases such as reading measurement data or the fault memory for example. In order to read or clear the fault memory a `FunctionalGroupFaultMemory` object is used. The `FunctionalGroup` object can also be used to receive a `FunctionalGroupBaseController` object for more sophisticated tasks such as sending HEX services.

5.3.3 STATES OF THE DIAGPORT

[Figure 152](#) shows the state diagram of the `DiagPort`. There are two states, `eCONNECTED`, and `eDISCONNECTED`. After creation, the `DiagPort` instance is always in state `eDISCONNECTED`. Following the states are explained:

Table 48 States of the DiagPort

State	Description
eDISCONNECTED	Initial state with no connection to the D-Server. Thus no diagnostic operations are possible in this state.
eCONNECTED	The D-Server is connected and a project and a vehicle information table is selected. In this state ECUs or function groups can be accessed to perform diagnostic operations on them.

The `Configure(aConfigFile)` method switches from the `eDISCONNECTED` state to the `eCONNECTED` state.

The `Disconnect()` method switches from `eCONNECTED` state back to the `eDISCONNECTED` state.

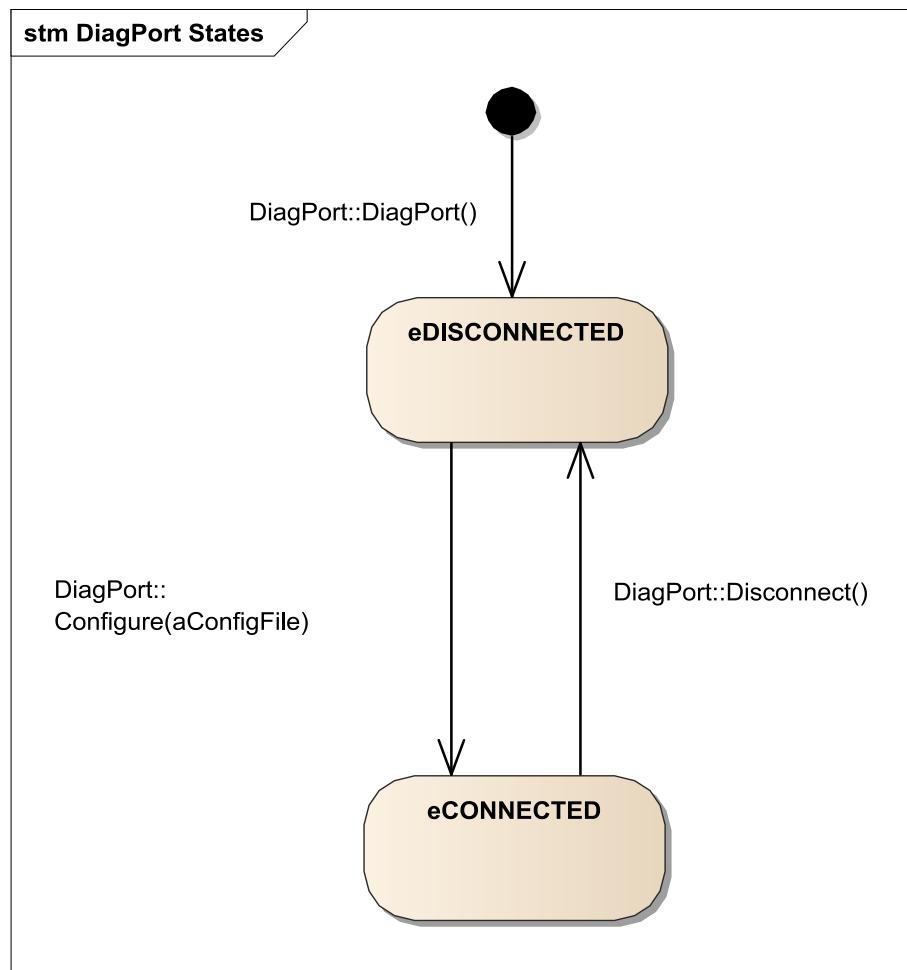


Figure 152: States of the Diag port

Table 49 DiagPort states

	eDISCONNECTED	eCONNECTED
Method Configure	x	
Method Disconnect		x
Method GetECU		x
Method GetFunctionalGroup		x
Property getProjectName		x
Property getState	x	x
Property getVehicleInfoTable		x

State transitions only take place, if all preconditions are fulfilled and no error occur during execution of the state changing method. Otherwise the state is not changed, i.e. the state remains the same as before the method was called. Methods, which trigger a state change, will throw an exception if the state change could not be processed successfully.

5.3.4 USAGE OF THIS PORT

This chapter depicts the usage of the Diagnostic Port API. The example use cases are based on popular tasks in the context of hardware-in-the-loop test automation.

5.3.4.1 CREATION AND CONFIGURATION

Instances of `DiagPort` are created by calls to method `CreateDiagPort` of a `DiagPortFactory` object. It returns a `DiagPort` instance with the name given as parameter. The `DiagPortFactory` object can be obtained from the `Testbench` object that provides a corresponding property.

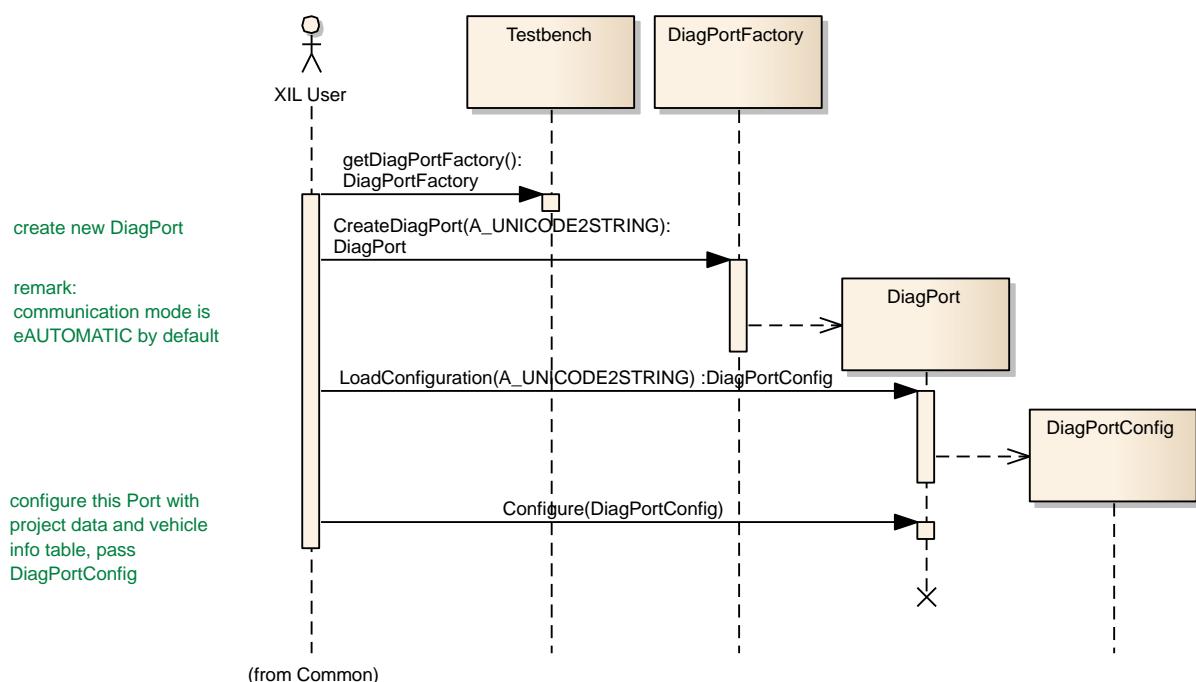


Figure 153: Process of DiagPort creation and configuration

Configuration of the new port instance is a two-step process (see [Figure 153](#)). First a vendor specific configuration file has to be loaded via method `LoadConfiguration`. Besides other vendor specific port settings this file particularly specifies the diagnostic project and the vehicle information table to be used by the port. Both settings can be queried from or changed by corresponding properties of the `DiagPortConfig` object that is returned by `LoadConfiguration`.

The second step of configuration is calling the `Configure` method and passing the `DiagPortConfig` object created in step one. This establishes a connection to the diagnostic tool and activates the specified diagnostic project and vehicle information table. On successful configuration the port's state is set to `eConnected`.

5.3.4.2 GETTING THE ECU OBJECT

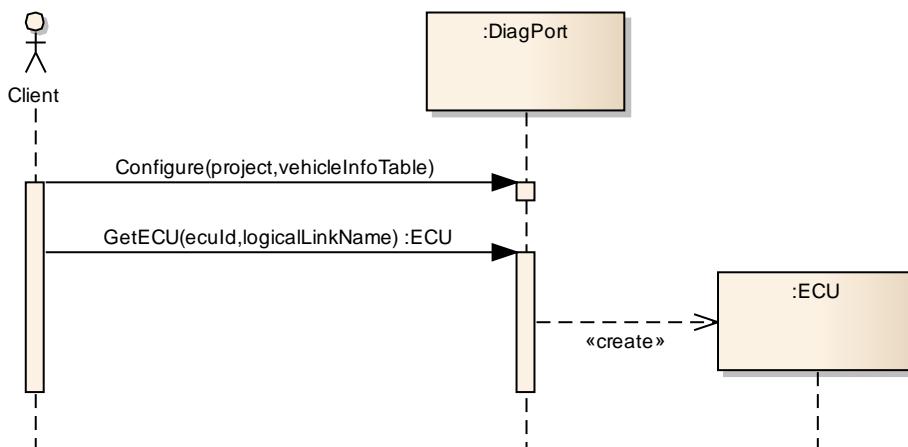


Figure 154: Getting the ECU Object

An ECU object can be obtained by invoking the `GetECU()` method of the `DiagPort` object with parameters for the ECU's ID and the name of the logical link the ECU is connected to. The sequence diagram in [Figure 154](#) depicts the necessary steps. It is assumed that the `DiagPort` instance has been created and configured previously.

5.3.4.3 READING AND CLEARING THE FAULT MEMORY

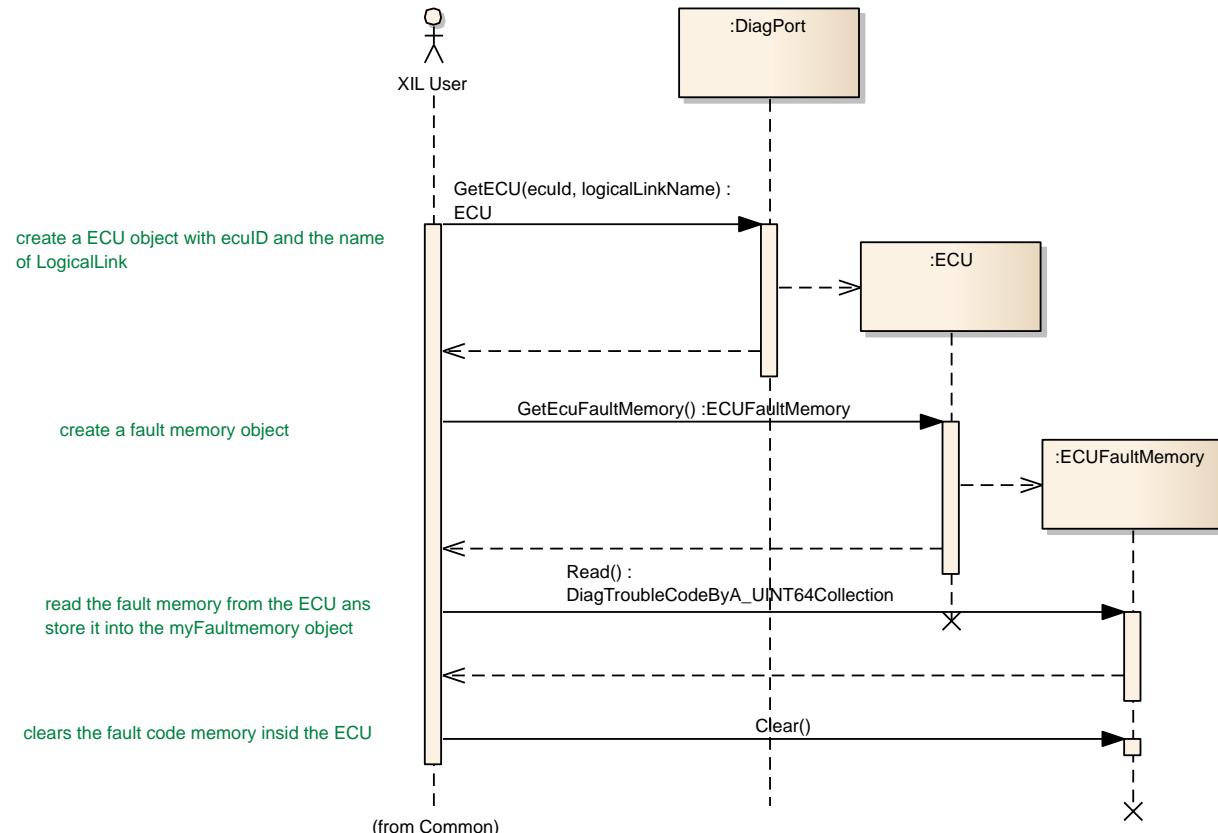


Figure 155: Reading and clearing the fault memory

When the client holds a reference to an ECU object (see chapter [Getting The Ecu Object](#)), reading the fault memory of the ECU is performed in several steps. The client invokes the `GetEcuFaultMemory()` method of the ECU object and gets an `ECUFaultMemory` object as return value. The `ECUFaultMemory` object is used to receive a dictionary of diagnostic trouble codes. In order to receive this dictionary the client of the diagnostic port has to invoke the `ECUFaultMemory` object's `Read()` method. The `DiagTroubleCodeByA_UINT64Collection` object that is returned provides methods for getting diagnostic trouble codes by DTC value or listing all available trouble code entries. See the API documentation of the `DiagTroubleCode_ByA_UINT64Collection` class for a full overview of available methods. [Figure 155](#) depicts the steps needed to read a diagnostic trouble code by DTC value. The last step clears the fault memory by invoking the `Clear()` method of the `ECUFaultMemory` object. The `DiagTroubleCode` object contains getters for short name, long name, description and value of the DTC entry.

5.3.4.4 READING THE VARIANT CODING DATA

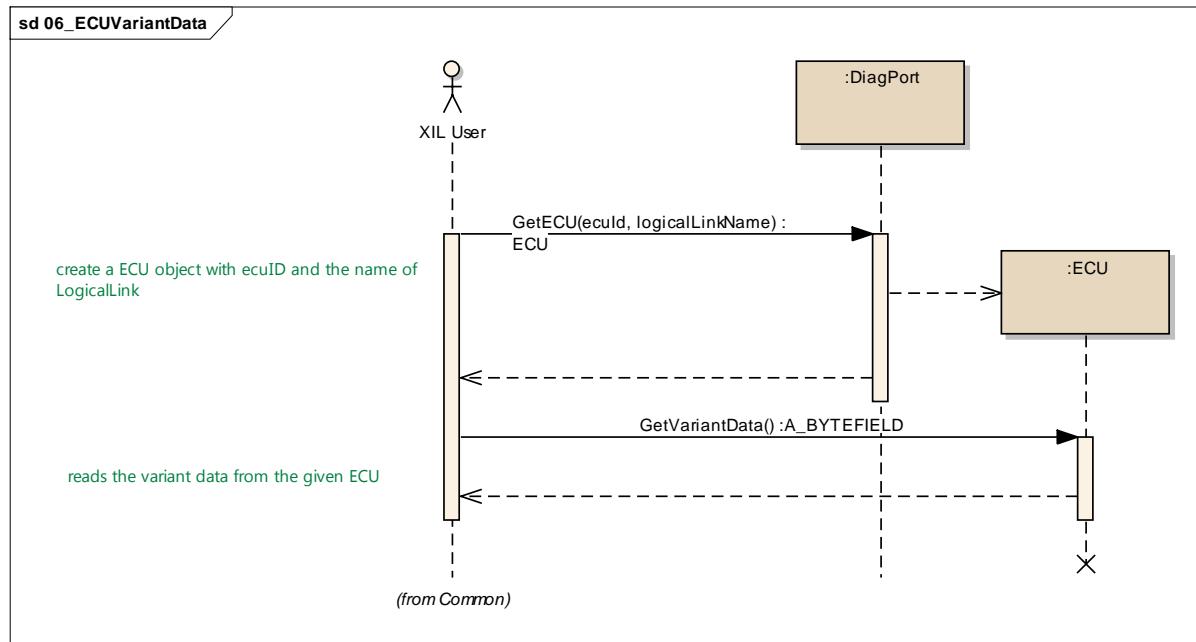


Figure 156: Reading the variant coding data

Variant Coding is used to adapt the ECU's software to operating conditions. Typically, this is performed during the production of the vehicle. For example toggling between left-hand drive and right-hand drive according to the sales country is performed during variant coding. Figure 156 illustrates the steps needed to read the variant coding of an ECU. After having received the ECU object from the DiagPort object (see chapter [Getting The Ecu Object](#)) the GetVariantData() method is invoked to read the variant coding data.

5.3.4.5 READING IDENTIFICATION DATA

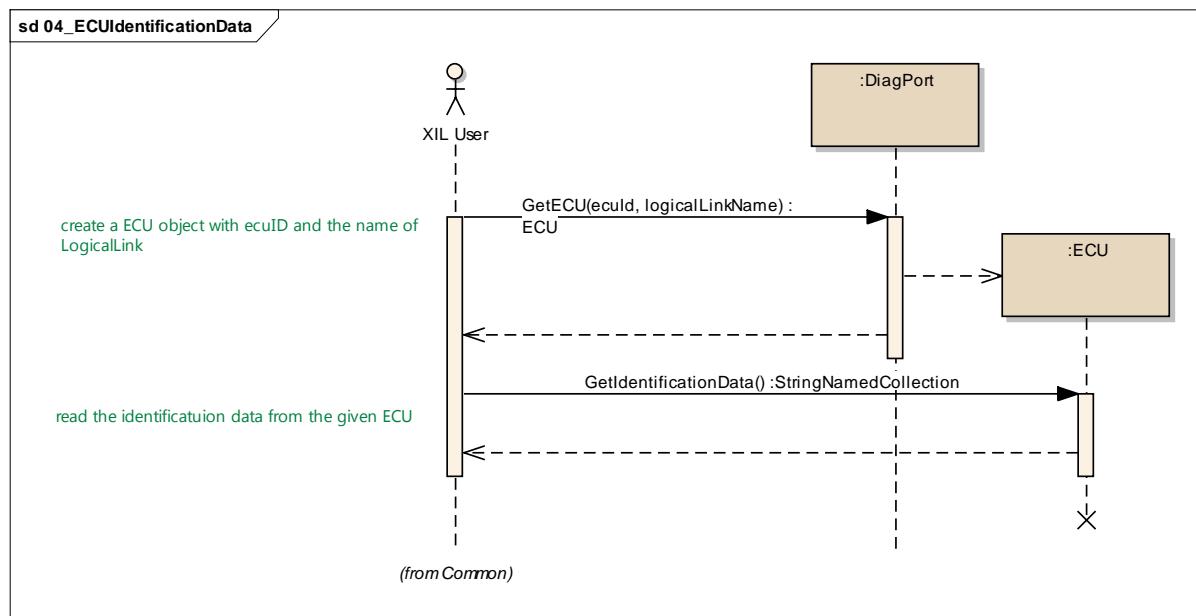


Figure 157: Reading identification data

Identification data is used to identify a given ECU. Identification data may comprise the vehicle manufacturer's part number, hardware part number, hardware version, software version etc. The Diagnostic Port API client can read identification data by means of the ECU object's `GetIdentificationData()` method. See chapter [Getting The Ecu Object](#) how to receive an ECU object. The result of an invocation of the `GetIdentificationData()` method is an DID array.

5.3.4.6 READING AND WRITING VALUES FROM AND TO THE EEPROM BY ALIAS NAMES

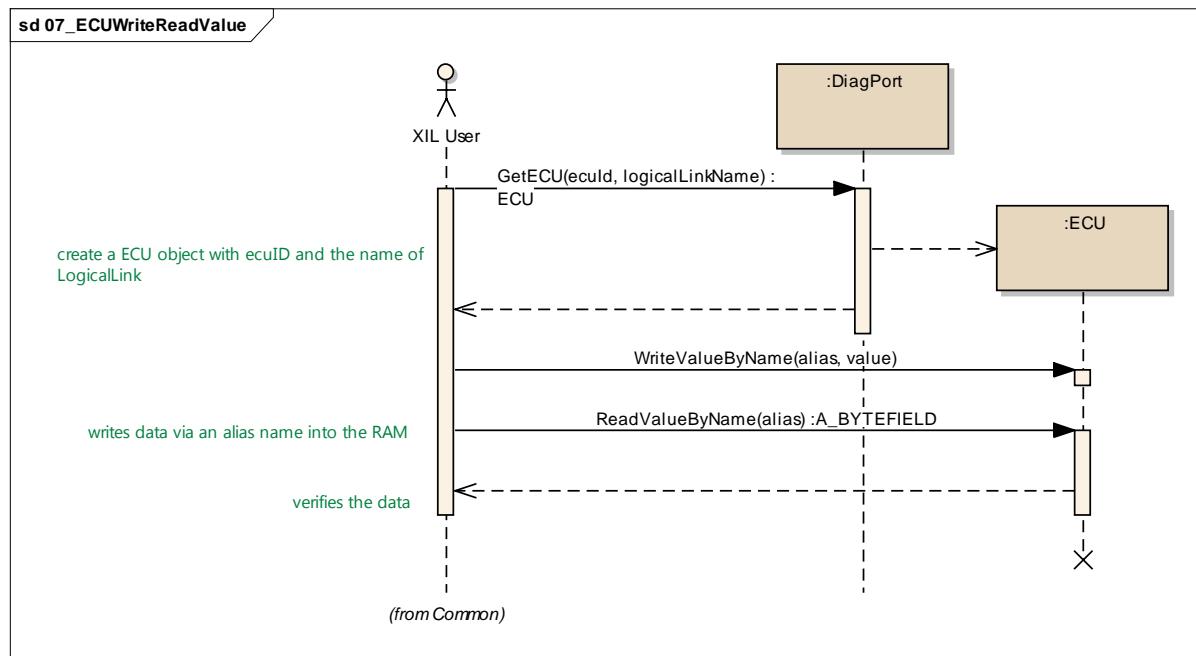


Figure 158: Reading and writing values from and to the EEPROM by an alias name

The client of the Diagnostic Port API can read and write data from and to the ECU's EEPROM. For the value access DID objects are used. For working with a DID object one possibility is using symbolic (alias) names for the EEPROM address. The diagnostic tool is responsible for resolving the symbolic name to a specific address value. The corresponding methods are located in the ECU class of the Diagnostic Port API. The method for writing is named `WriteDIDValues()` and the method for reading is named `ReadDIDValue()`. The diagnostic tool has to know how to map this DID objects. [Figure 158](#) depicts the steps needed to read and write from and to the ECU's EEPROM by an alias name.

5.3.4.7 READING FROM THE EEPROM

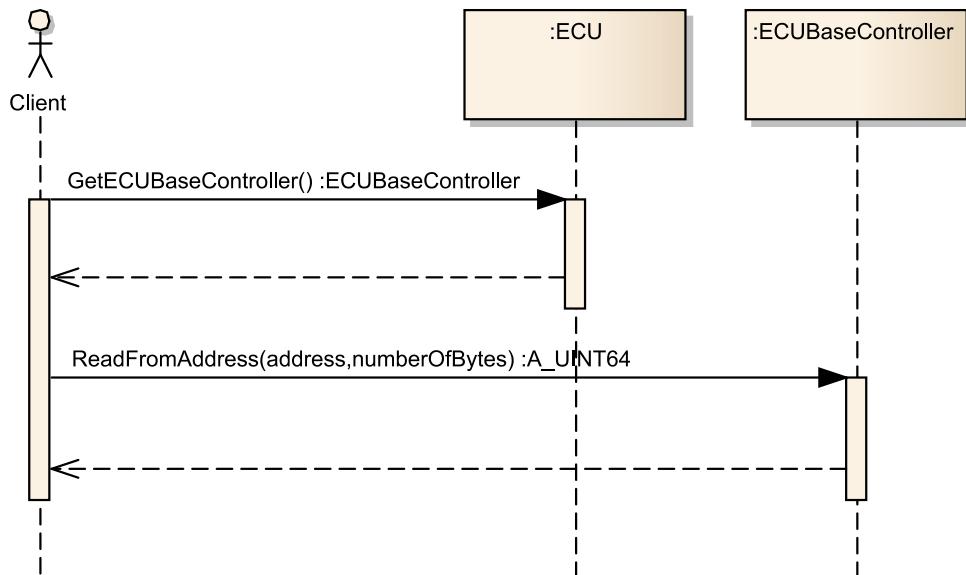


Figure 159: Reading from the EEPROM

Reading data from an address in the EEPROM of the ECU is performed by using the ECUBaseController object. A ECUBaseController object can be received from the ECU object by invoking its `GetECUBaseController()` method. See chapter [Getting The Ecu Object](#) how to obtain an ECU object. With the ECUBaseController's `ReadFromAddress()` method the client can read a bunch of bytes from the ECU's EEPROM. The `ReadFromAddress()` takes two parameters: one for the address in the EEPROM and one for the number of bytes to read.

5.3.4.8 WRITING TO THE EEPROM

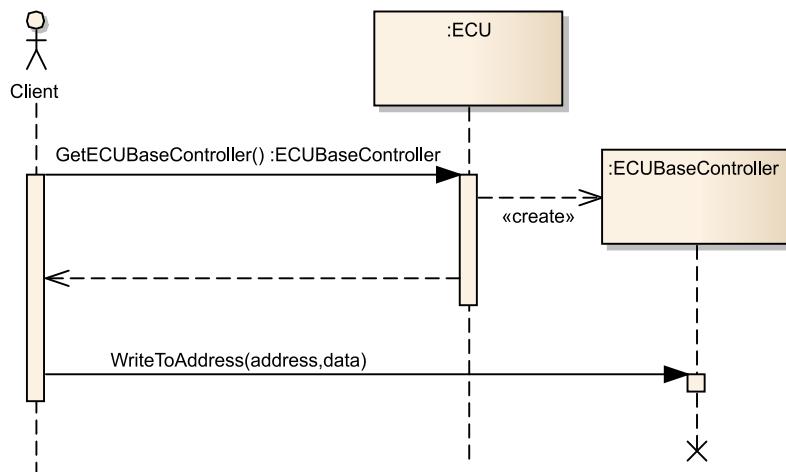


Figure 160: Writing to the EEPROM

Writing data to an address in the EEPROM is performed by using the ECUBaseController object. A client receives an ECUBaseController object by invoking the ECU object's `GetECUBaseController()` method. See chapter [Getting The Ecu Object](#) how to obtain an ECU object. The ECUBaseController's `WriteToAddress()` method writes

the data specified with the data parameter to the EEPROM at the address specified with the address parameter.

5.3.4.9 IMPLICIT AND EXPLICIT COMMUNICATION

The chapters above did all use the automatic (implicit) communication mode. In this case the client of the Diagnostic Port must not take care of creating and destroying communication channels in order to communicate with the ECU. In automatic communication mode it's up to the diagnostic tool—i.e. the XIL API Diagnostic Port server—to handle this task. The automatic communication mode is the default communication mode.

But there may also be use cases when a more sophisticated control over the communication is needed. That is why there is the explicit communication mode which is explained in the following chapter. In explicit communication mode the client is responsible for starting the communication with the `StartCommunication()` method before methods that need to communicate with the ECU can be invoked, e.g. `SendHexService()`. Also the client is responsible for stopping the communication with the `StopCommunication()` method after the mentioned method has been called.

5.3.4.10 SENDING HEX SERVICES WITH EXPLICIT COMMUNICATION

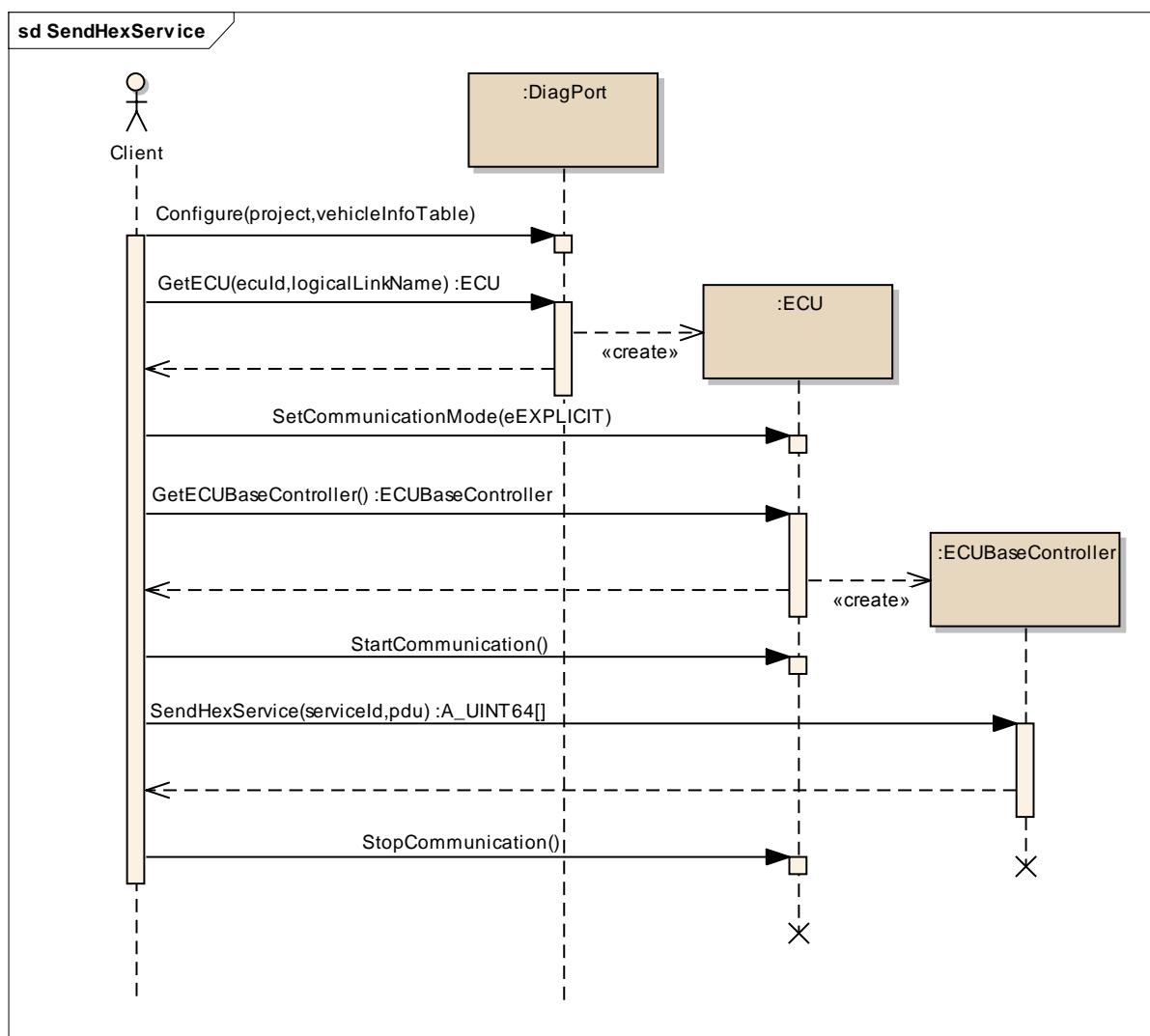


Figure 161: Sending HEX service with explicit communication

This chapter illustrates the usage of the explicit communication mode. In explicit communication mode the client has to start and stop the communication with the ECU explicitly. After the project is configured the client receives the ECU object by invoking the `GetECU()` method. Then the communication mode is set to explicit communication mode (`eEXPLICIT`) with the `CommunicationMode()` property. Before calling an method that communicates with the ECU—e.g. the `SendHexService()` method—the `StartCommunication()` method of the ECU class has to be called. Finally the `StopCommunication()` method has to be called in order to stop the communication with the ECU. [Figure 161](#) shows the steps needed to send an HEX service with explicit communication.

Since the Diagnostic Port only defines the two mentioned communication modes as state that can be changed via the `CommunicationMode()` property no state diagrams are provided here.

5.3.4.11 EXECUTING JOBS

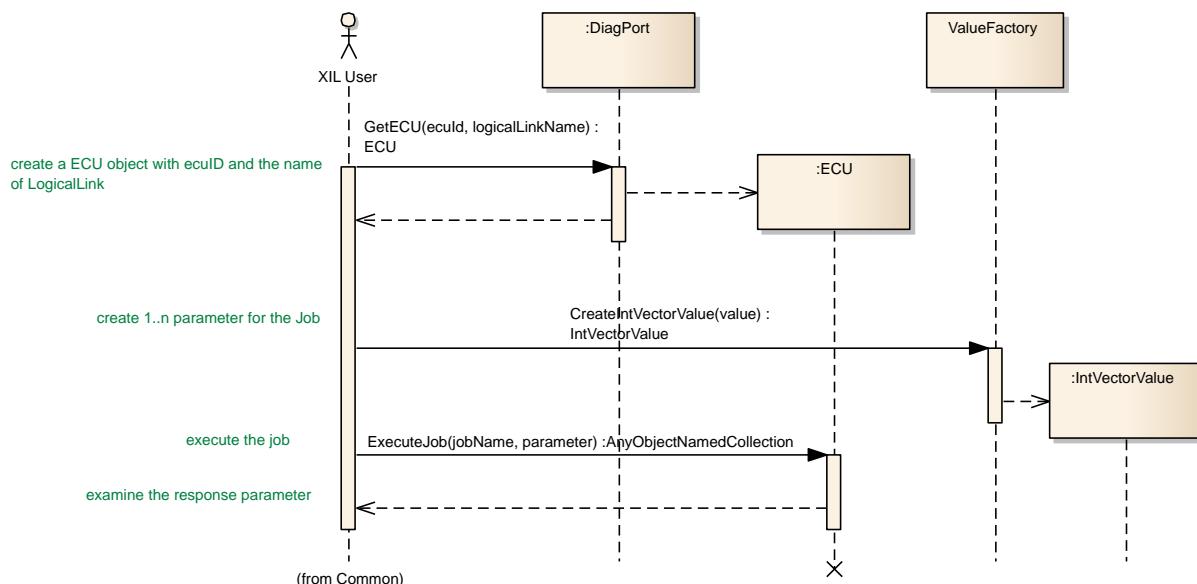


Figure 162: Executing a Job

The Diagnostic Port API provides methods for the execution of diagnostic jobs. [Figure 162](#) shows the steps needed to execute a diagnostic job. After configuration of the Diagnostic Port and receiving of the ECU object, the client creates the job parameters and invokes the `ExecuteJob()` method of the ECU class.

5.3.4.12 READING DATA FROM A FUNCTIONAL GROUP

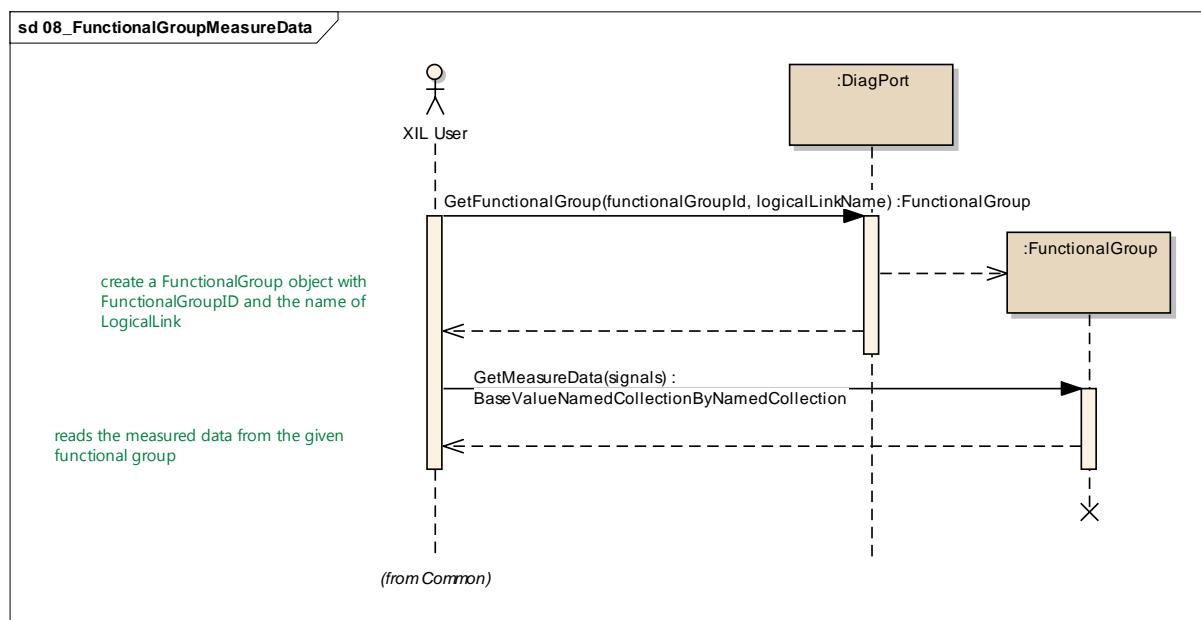


Figure 163: Reading measurement data from a functional group

Since the Diagnostic Port API supports functional groups, the client can read data from a functional group, i.e. from a group of ECUs. [Figure 163](#) depicts the steps needed to perform this task. After configuration of the DiagPort object and receiving of the FunctionalGroup object, the client invokes the Functional Group object's ReadDIDValues() method with the respective DID objects as parameter.

5.3.4.13 USING THE BASECONTROLLER

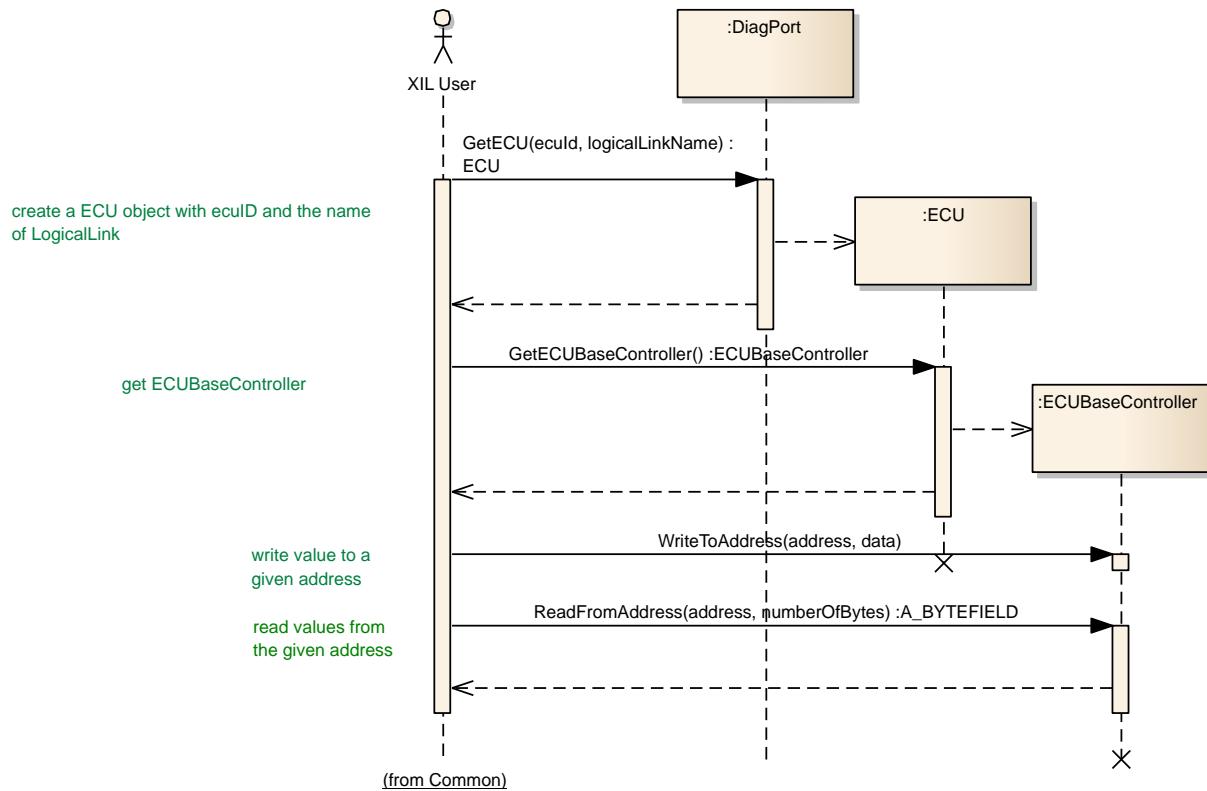


Figure 164: Using the BaseController

The BaseController is used in cases a more sophisticated access to the diagnostic tool is needed. With the BaseController's methods the client is able to read and write values directly to addresses of the ECU's memory. Figure 164 outlines the steps needed to read and write a number of bytes from and to the ECU's memory. The client receives the BaseController object directly from the ECU object as a return value of the GetECUBaseController() method.

5.3.5 SPECIAL HINTS

5.3.5.1 STRUCTURE OF RETURNED COLLECTIONS

The structures of the return collections of methods of the Diagnostic Port's classes depend on the ECU that is connected to the diagnostic tool. Therefore the Diagnostic Port API does not specify any return structures.

5.3.5.2 STATES IN THE DIAGNOSTIC TOOL

There are no restrictions on the invocation of methods that affect the communication mode or the communication status respectively.

5.4 ECUMPORT

5.4.1 OVERVIEW

The ECUMPort of the XIL API accesses an ECU via an MC server. It provides the functionalities of measurement and capturing

The XIL API only communicates with the MC server, it does not communicate directly with the ECU. It is the task of the MC server to handle the communication with the ECU and to execute the methods defined by the XIL API. Therefore the XIL API does not need any knowledge about the interface being used for communication with the ECU (e.g. a CAN interface or a KWP2000 interface), and consequently this does not influence the code accessing the XIL API.

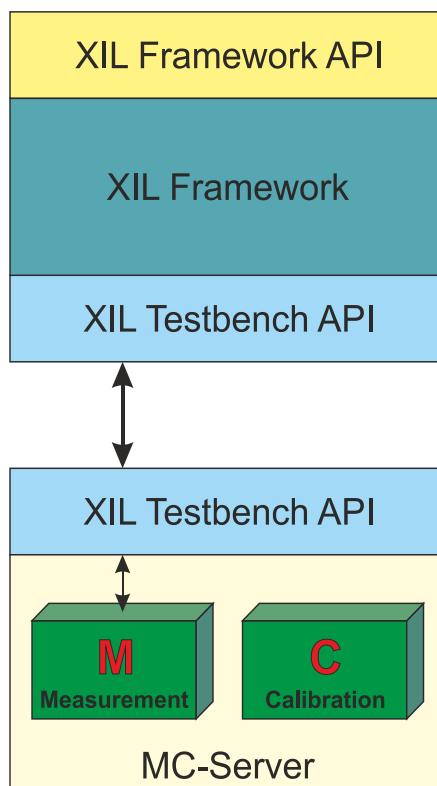


Figure 165: Accessing ECUs via the XIL API for measurement

The access to ECUs via the XIL API for Measurement is done by the ECUMPort, which is derived from the class Port (see [Figure 166](#)).

The ECUMPort is used for measuring and capturing variables of an ECU. The measuring can be done

- As a snapshot using the Read() method,
- By measuring of different variables in a specified raster (using Capture objects, see chapter Variable Ref and Value Representation Mode)

Wherever the client needs to access or specify variables of the target system, VariableRef objects are used. Typical use cases are the reading and writing of target system variables as

well as their capturing and stimulation. Instead of plain variable names, VariableRef objects are used to specify the variables to operate on.

VariableRefs have the advantage that they can refer not only to the target system variable as whole, but - in case of structured variables - also to a single element thereof. With a VariableRef, for example, one can specify a single element of a vector variable to be read, written, monitored, captured, stimulated etc.

VariableRef instances are created by means of the VariableRefFactory that can be obtained from the Testbench. Figure 116 shows the various VariableRef interfaces. Their VariableName property gives the name of the target system variable (e.g. the model path of a simulator variable). If a variable is to be accessed as whole, no matter whether it is a scalar, vector, matrix etc. variable, a GenericVariableRef instance must be used. If a single element of a vector or matrix is to be accessed, a VectorElementRef resp. MatrixElementRef instance must be used.

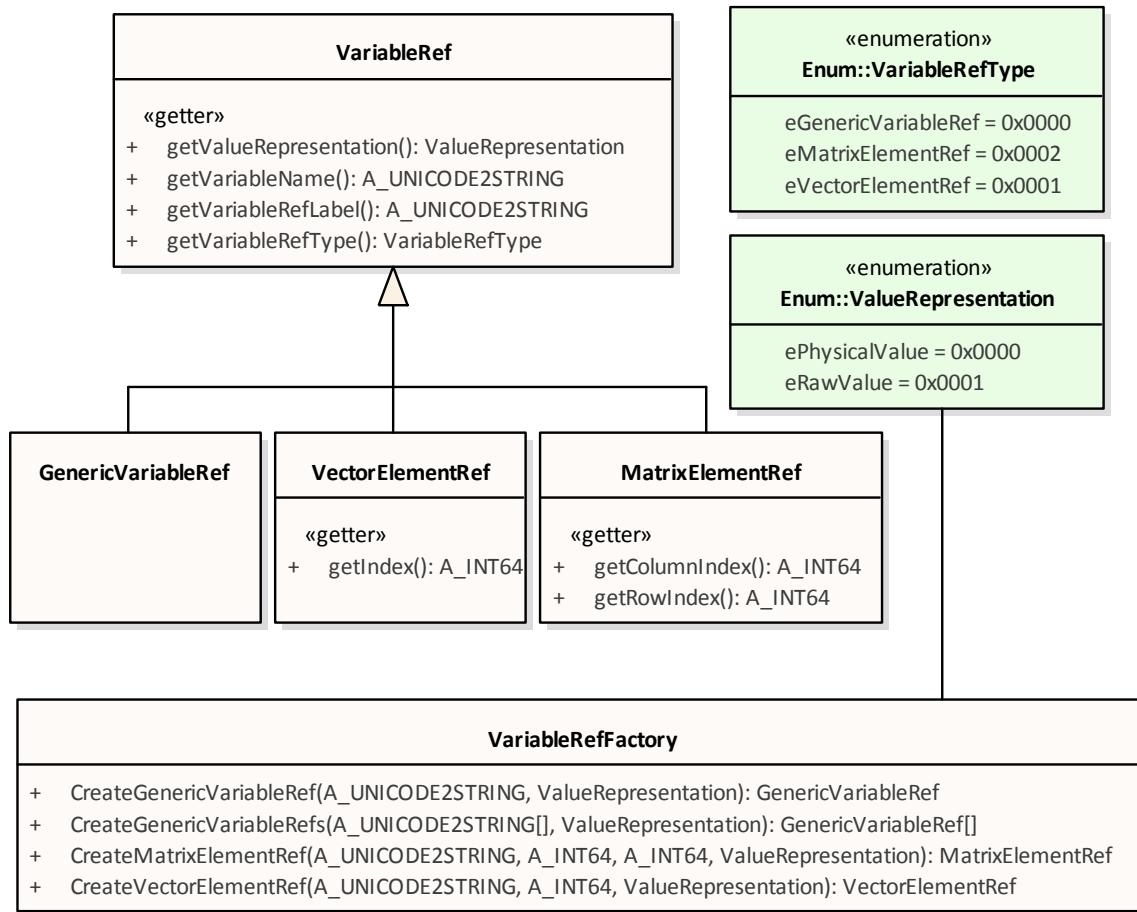


Figure 116: VariableRef interfaces

The second purpose of a VariableRef object is to specify the representation mode to be used on access to the variable. The representation mode is determined by the ValueRepresentation property which can have one of the following values:

ePhysicalValue

Specifies that operations are to be executed on the physical value of a variable. The physical value denotes the human readable representation. This is either a measured value with a physical unit or a verbal text value. The physical value is also known as application value.

eRawValue

Specifies that operations are to be executed on the raw value of a variable. The raw value denotes a simulator's or ECU's internal representation as well as the encoded signal value when transmitted over a communication network. The raw value is also known as internal or implementation value.

Note: VariableRef instances are used as keys in dictionaries (e.g. in the Assignments2 collection of SignalGenerator). That is why they must be testable for equality and provide a hash value. VariableRef instances compare equal if and only if they match in all properties. Furthermore, VariableRef instances that compare equal have the same hash value.

Note: There are some cases where a unique, human readable name for the object referenced by a VariableRef object is required. The creation of names for the variable traces stored in a CaptureResult is a typical example for this requirement. A human should be able to draw conclusions about the identity of the captured variables from the trace names. That is why a string representation of the captured VariableRef object is used as trace name.

For these use cases the property VariableRefLabel was introduced. It yields a human readable, but unique identifier for the referenced variable resp. the referenced variable element. The generation of this name considers all properties of the VariableRef. However, the generated name is vendor specific.

- Data Capturing). Use the CreateCapture() method to create capture objects.

The VariableNames() property returns a list of all available variable names of the ECUMPort instance.

The TaskNames() property returns a list of all available raster names of the ECUMPort instance.

Note: If the user wants to read variables from the ECUMPort it is necessary that these variables are announced to the port as precondition. The announcement is realized with the method ECUMPort.setMeasuringVariables.

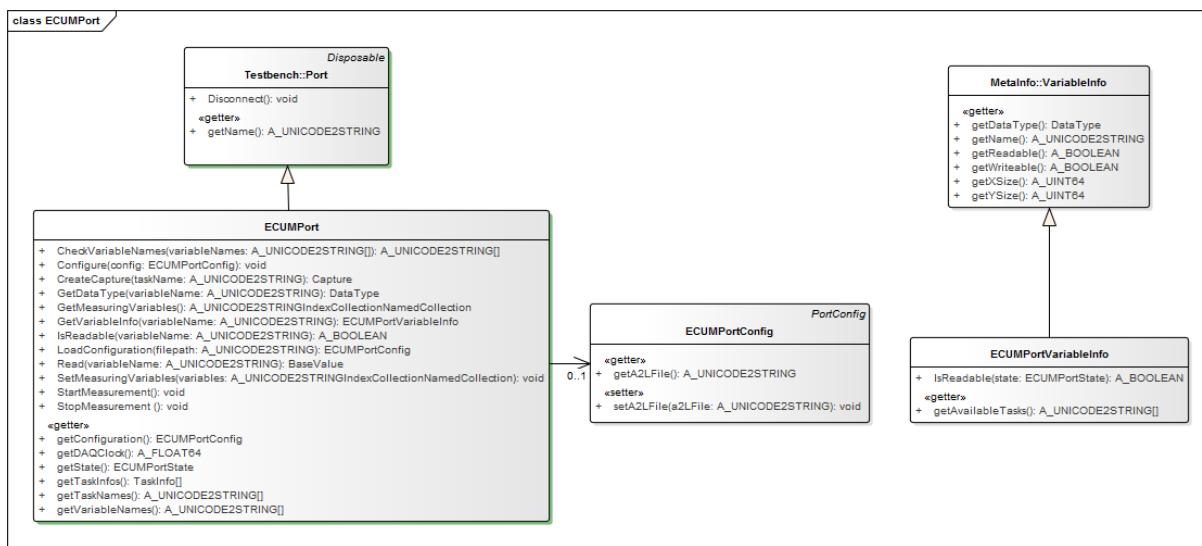


Figure 166: ECUMPort Classdiagram

5.4.2 STATES OF THE ECUMPORT

Figure 167 shows the state diagram of the ECUMPort.

There are three states, eMEASUREMENT_STOPPED, eMEASUREMENT_RUNNING and eDISCONNECTED. After creation, the ECUMPort instance is always in state eDISCONNECTED. The states in Table 50 are explained:

Table 50 States of the ECUMPort

State	Description
eDISCONNECTED	Measurements are not possible, because there is no connection to the MC-Server.
eMEASUREMENT_STOPPED	The MC-Server is connected and configured but there is no connection to the ECU. Capturing and reading ECU variable values is not possible. However, it is possible to create capture instances.
eMEASUREMENT_RUNNING	A connection to the ECU has been established. Reading ECU variable values is possible. Capturing can be started and stopped, without any influence on the ECUM port's state.

The StartMeasurement() method switches from the eMEASUREMENT_STOPPED state to the eMEASUREMENT_RUNNING state and is only allowed in state eMEASUREMENT_STOPPED. With this method the alignment between MCServer and ECU takes place.

The StopMeasurement() method switches from eMEASUREMENT_RUNNING state back to the eMEASUREMENT_STOPPED state.

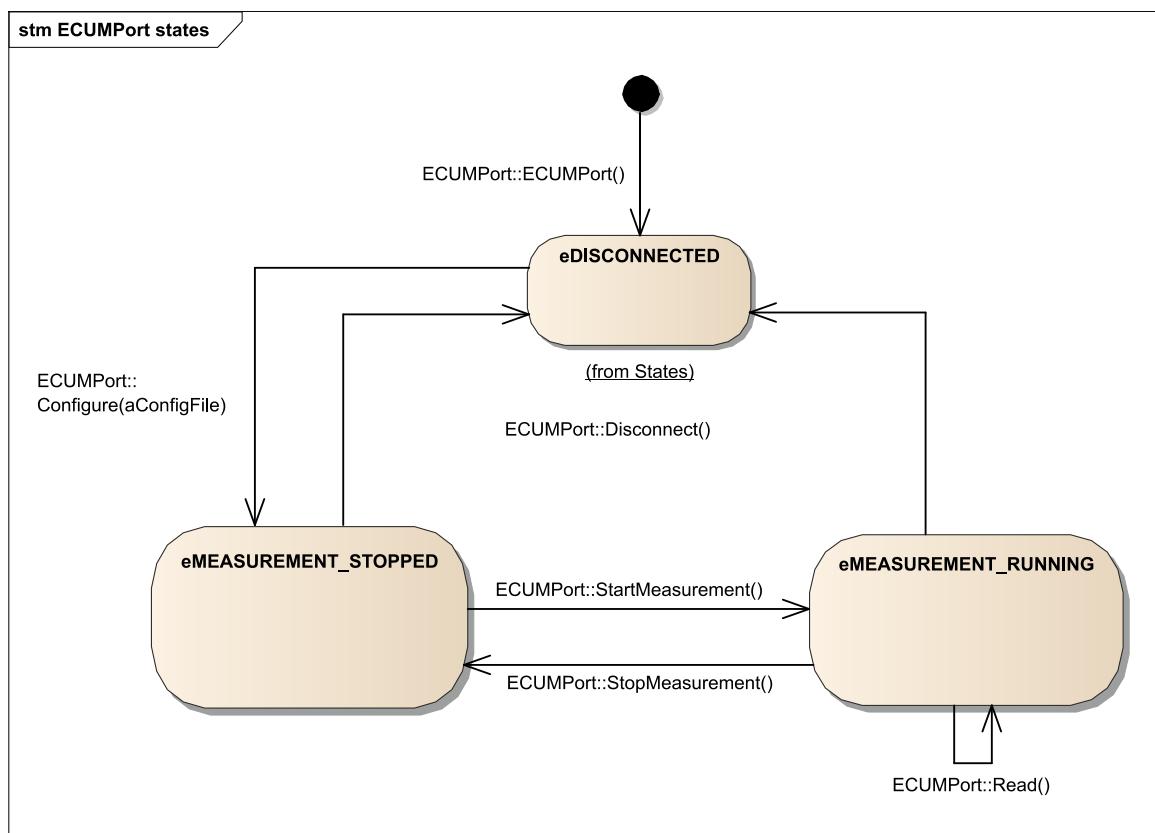


Figure 167: States of the ECUM port

Table 51 ECUMPort states

	eDISCONNECTED	eMEASUREMENT_STOPPED	eMEASUREMENT_RUNNING
Method CheckVariableNames		x	x
Method Configure	x		
Method CreateCapture		x	
Method Disconnect		x	x
Method GetDataType		x	x
Method GetMeasuringVariable		x	x
Property getState	x	x	x
Property getTaskInfos		x	x
Property getTaskNames		x	x
Method GetVariableInfo		x	x
Property getVariableNames		x	x
Method IsReadable		x	x
Method Read			x
Method SetMeasuringVariable		x	
Method StartMeasurement		x	
Method StopMeasurement			x

State transitions only take place, if all preconditions are fulfilled and no error occur during execution of the state changing method. Otherwise the state is not changed, i.e. the state remains the same as before the method was called. Methods, which trigger a state change, will throw an exception if the state change could not be processed successfully.

5.4.3 USAGE OF ECUMPORT

5.4.3.1 CREATION AND CONFIGURATION

Instances of `ECUMPort` are created by calls to method `CreateECUMPort` of an `ECUMPortFactory` object. It returns a `ECUMPort` instance with the name given as parameter. The `ECUMPortFactory` object can be obtained from the `Testbench` object that provides a corresponding property.

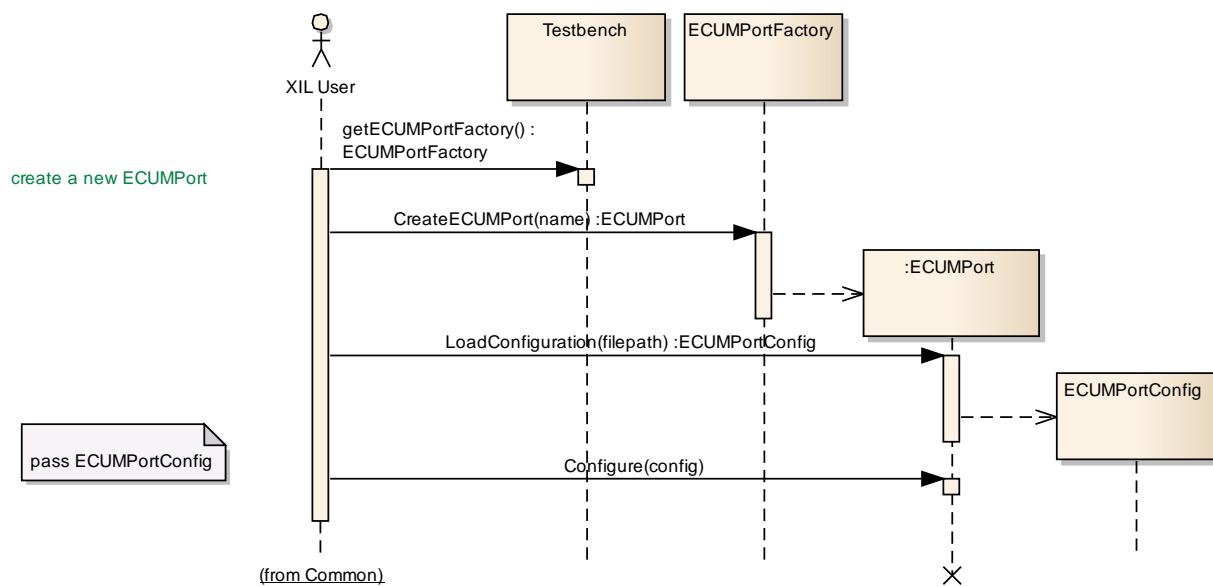


Figure 168: Process of ECUMPort creation and configuration

Configuration of the new port instance is a two-step process (see [Figure 168](#)). First a vendor specific configuration file has to be loaded via method `LoadConfiguration`. Besides other vendor specific port settings this file particularly specifies the A2L file to be used by the port. The A2L file path can be queried from or changed by a corresponding property of the `ECUMPortConfig` object that is returned by `LoadConfiguration`.

The second step of configuration is calling the `Configure` method and passing the `ECUMPortConfig` object created in step one. This establishes a connection to the measuring tool or hardware and activates the passed configuration. On successful configuration the port's state is set to `eMEASUREMENT_STOPPED`.

5.4.3.2 GETTING LISTS OF VARIABLE AND TASK NAMES

The first example of the `ECUMPort` describes how a user can get the names of all variables and all tasks supported by this `ECUMPort` instance. Precondition is that the `ECUMPort` has been successfully created and configured.

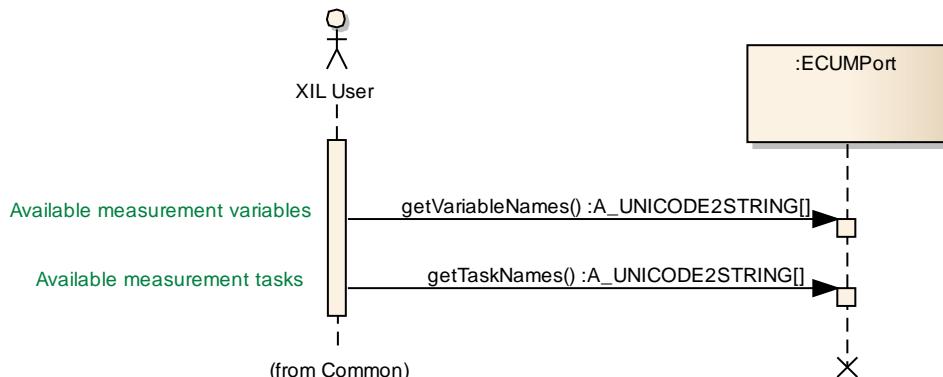


Figure 169: Get lists of variable and task names

Variable names can be obtained from the port's property `VariableNames`, task names are available via property `TaskNames` (see [Figure 166](#)).

5.4.3.3 READ A SCALAR VARIABLE VALUE AND ITS PROPERTIES

This example shows how to read the value of a scalar ECU variable.

The `ECUMPort` instance is switched to the `eONLINE` mode after creation. Then the measure value is fetched using the `Read()` method. After switching back to the `eOFFLINE` mode, the properties (e.g. name, unit, etc.) of the variable value are examined.

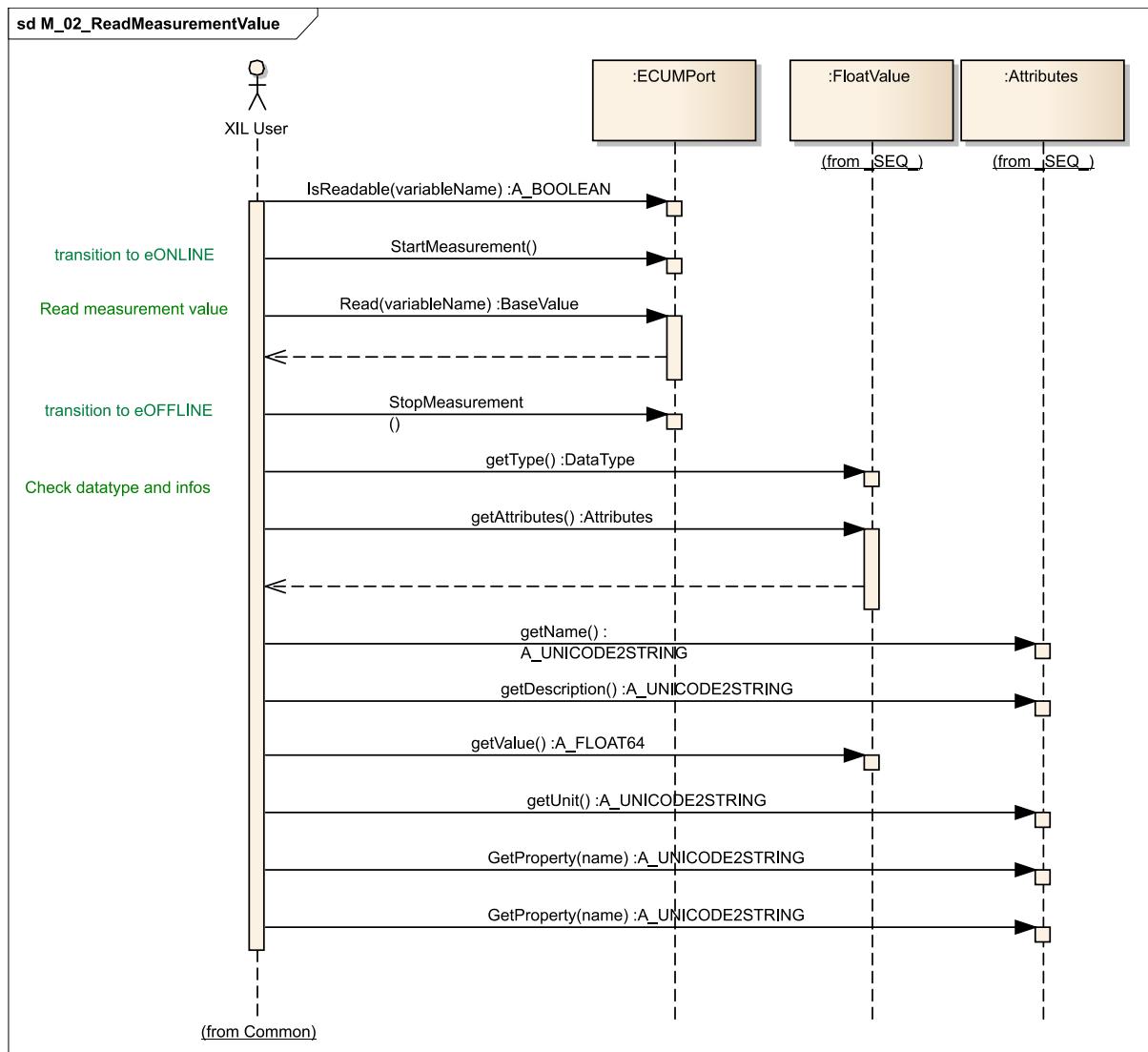


Figure 170: Read a scalar variable value and examine its properties

5.4.3.4 READ AN ARRAY VARIABLE VALUE AND ITS PROPERTIES

Reading an array value is nearly the same as reading a scalar value. The value returned by the `Read()` method now is a `VectorValue` which can be read out index by index.

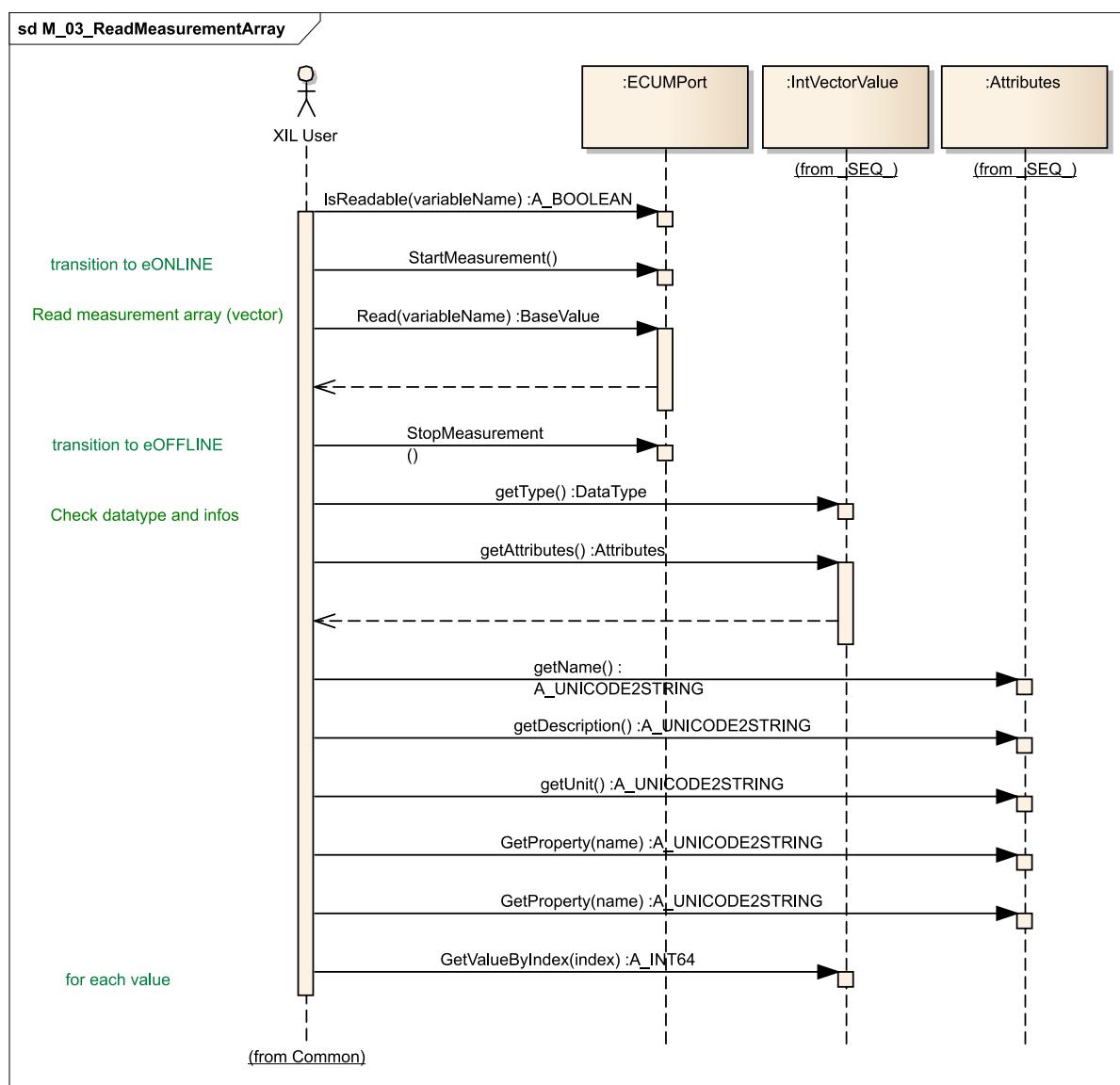


Figure 171: Read an array variable value and examine its properties

5.4.3.5 READ A MATRIX VARIABLE VALUE AND ITS PROPERTIES

Reading a matrix value is nearly the same as reading an array value. The value returned by the `Read()` method now is a `MatrixValue` which can be read out using row and column indices.

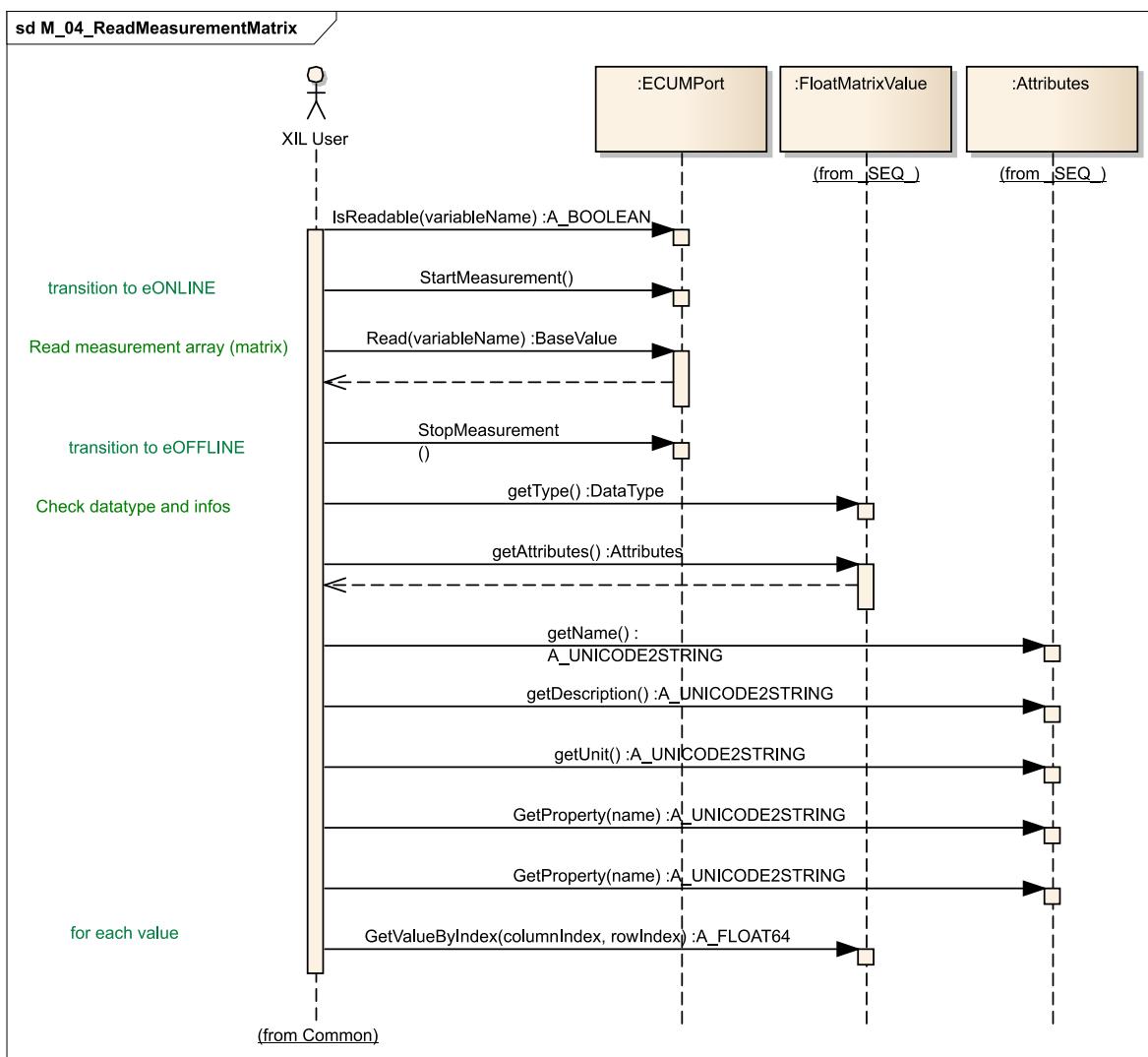


Figure 172: Read a matrix variable value and examine its properties

5.4.3.6 CAPTURING ECU VARIABLES

For reading more than one variable value, including its timestamps, a capture object must be used. The following example illustrates this.

A capture object always does the data acquisition using a specified raster. This raster is defined at creation time of the capture object using the `CreateCapture()` method. The subsequent call to `Variables()` property defines which ECU variables shall be captured by this capture instance.

Capturing can only be done in the `eONLINE` state, which is initiated by a call of the `Start()` method. Now the capturing can be started and stopped by the `Capture.Start()` and `Capture.Stop()` methods. When capturing has been finished, the `ECUMPort` instance can be switched back to the `eOFFLINE` state. A call to the `Capture.getCaptureResult()` function returns all captured variable values.

See chapter [Variable Ref](#) and Value Representation Mode

Wherever the client needs to access or specify variables of the target system, `VariableRef` objects are used. Typical use cases are the reading and writing of target system variables as well as their capturing and stimulation. Instead of plain variable names, `VariableRef` objects are used to specify the variables to operate on.

VariableRefs have the advantage that they can refer not only to the target system variable as whole, but - in case of structured variables - also to a single element thereof. With a VariableRef, for example, one can specify a single element of a vector variable to be read, written, monitored, captured, stimulated etc.

VariableRef instances are created by means of the VariableRefFactory that can be obtained from the Testbench. Figure 116 shows the various VariableRef interfaces. Their VariableName property gives the name of the target system variable (e.g. the model path of a simulator variable). If a variable is to be accessed as whole, no matter whether it is a scalar, vector, matrix etc. variable, a GenericVariableRef instance must be used. If a single element of a vector or matrix is to be accessed, a VectorElementRef resp. MatrixElementRef instance must be used.

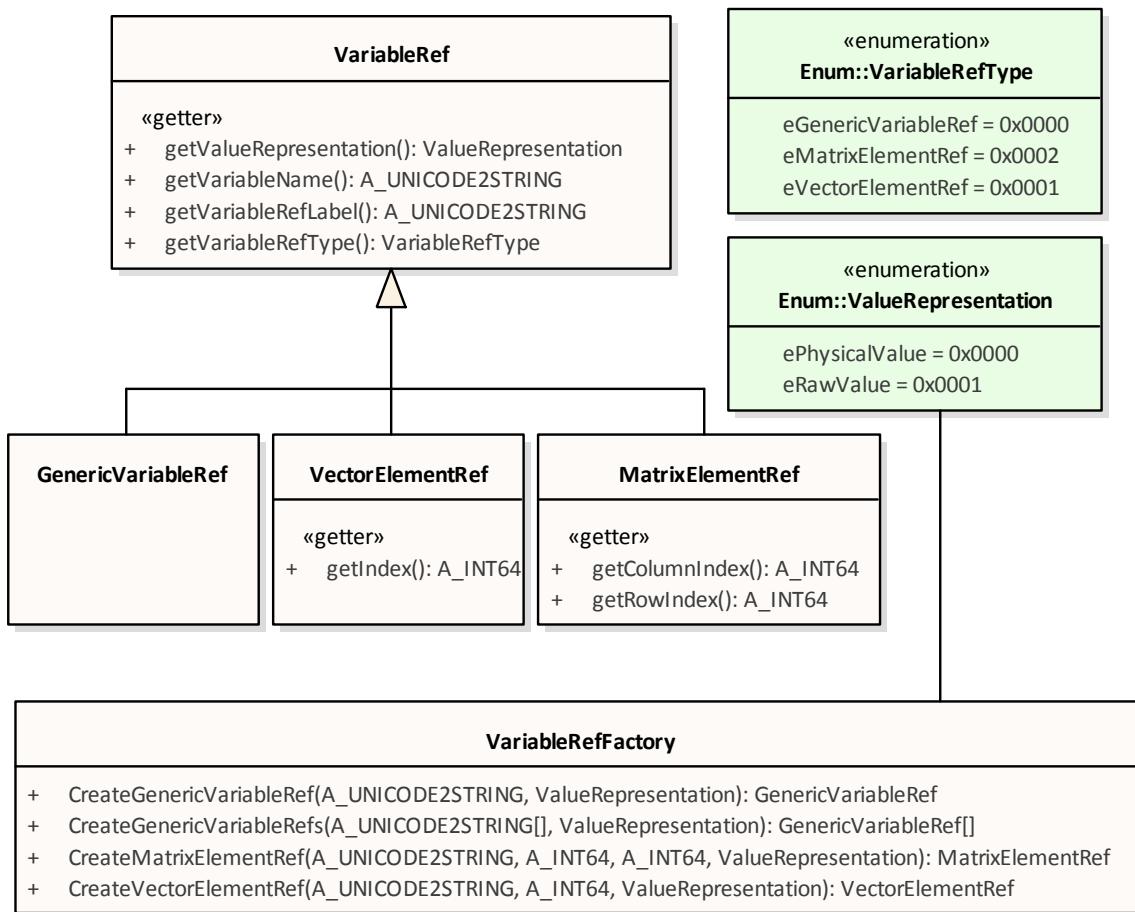


Figure 116: VariableRef interfaces

The second purpose of a VariableRef object is to specify the representation mode to be used on access to the variable. The representation mode is determined by the ValueRepresentation property which can have one of the following values:

ePhysicalValue

Specifies that operations are to be executed on the physical value of a variable. The physical value denotes the human readable representation. This is either a measured value with a physical unit or a verbal text value. The physical value is also known as application value.

eRawValue

Specifies that operations are to be executed on the raw value of a variable. The raw value denotes a simulator's or ECU's internal representation as well as the encoded signal value when transmitted over a communication network. The raw value is also known as internal or implementation value.

Note: VariableRef instances are used as keys in dictionaries (e.g. in the Assignments2 collection of SignalGenerator). That is why they must be testable for equality and provide a hash value. VariableRef instances compare equal if and only if they match in all properties. Furthermore, VariableRef instances that compare equal have the same hash value.

Note: There are some cases where a unique, human readable name for the object referenced by a VariableRef object is required. The creation of names for the variable traces stored in a CaptureResult is a typical example for this requirement. A human should be able to draw conclusions about the identity of the captured variables from the trace names. That is why a string representation of the captured VariableRef object is used as trace name.

For these use cases the property VariableRefLabel was introduced. It yields a human readable, but unique identifier for the referenced variable resp. the referenced variable element. The generation of this name considers all properties of the VariableRef. However, the generated name is vendor specific.

Data Capturing for more information about the Capture and CaptureResult classes.

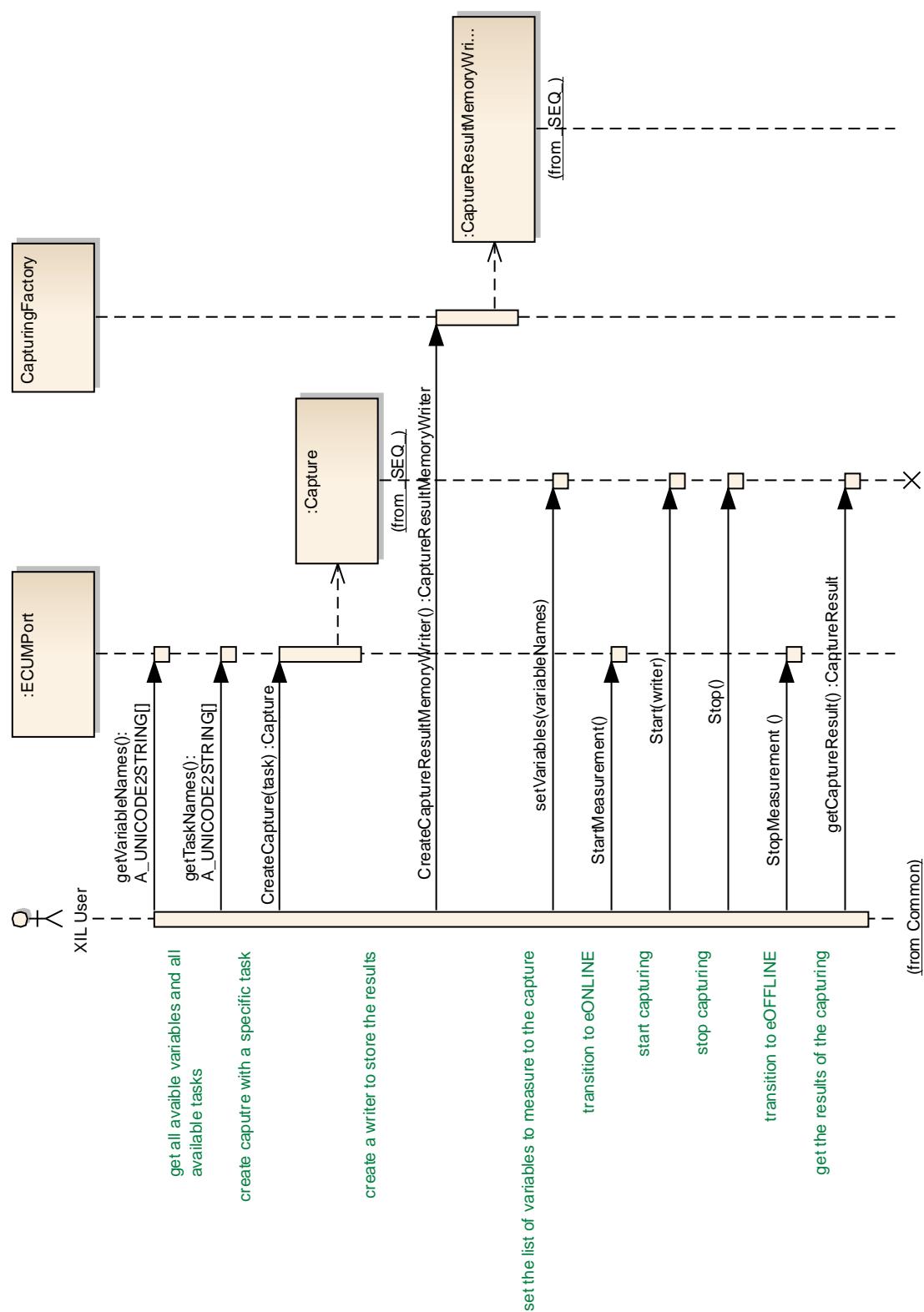


Figure 173: Capturing ECU variables

5.5 ECUCPort

5.5.1 OVERVIEW

The ECUCPort of the XIL API accesses an ECU via an MC server. It provides the following functionality:

- Calibration
- Management of ECU memory pages

The XIL API only communicates with the MC server, it does not communicate directly with the ECU. It is the task of the MC server to handle the communication with the ECU and to execute the methods defined by the XIL API. Therefore the XIL API does not need any knowledge about the interface being used for communication with the ECU (e.g. a CAN interface or a KWP2000 interface), and consequently this does not influence the code accessing the XIL API. [Figure 174](#) illustrates this:

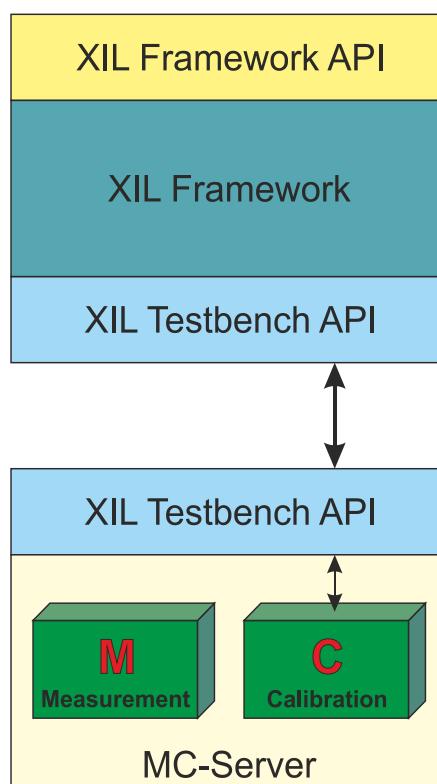


Figure 174: Accessing ECUs via the XIL API for Calibration

The access to ECUs via the XIL API for Calibration is done by the `ECUCPort`, which is derived from the class `Port` (see [Figure 175](#)).

The `ECUCPort` class provides functionality for reading and writing parameters of an ECU. It can also handle memory pages.

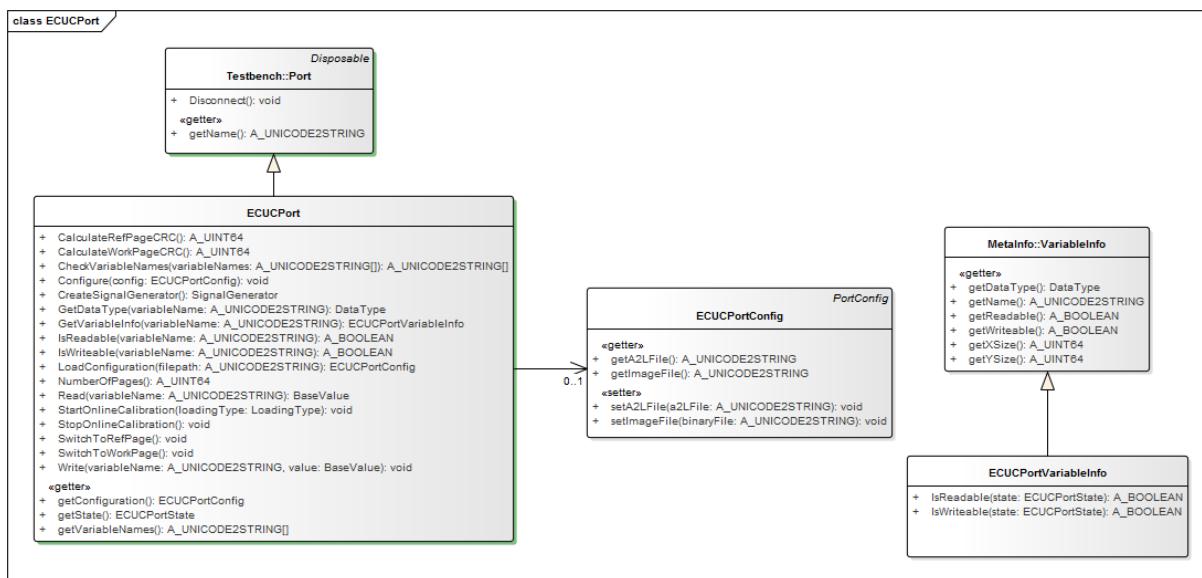


Figure 175: ECUCPort Classdiagram

5.5.2 STATES OF THE ECUCPORT

Figure 176 illustrates the state diagram of the ECUCPort.

There are three states, eOFFLINE, eONLINE and eDISCONNECTED. After creation, the ECUCPort instance is always in state eDISCONNECTED. Following the states are explained:

Table 52 States of the ECUCPort

State	Description
eDISCONNECTED	Calibrations are not possible, because there is no connection to the MC-Server.
eOFFLINE	The MC-Server is connected and configured but there is no connection to the ECU. Reading parameter values returns the values currently being stored in the server only, not those from the ECU. Writing parameter values changes the values currently being stored on the server only, they are not written to the ECU.
eONLINE	A connection to the ECU has been established. Reading parameter values returns the current values from the ECU. Writing parameter values changes the current values on the ECU.

The StartOnlineCalibration(LoadingType) method switches from the eOFFLINE state to the eONLINE state (see also chapter [Manage ECU Memory Pages](#) for details) and is only allowed in state eOFFLINE. With the method the alignment between MCServer and ECU takes place.

The StopOnlineCalibration() method switches from eONLINE state back to the eOFFLINE state.

The Configure() method allows a connection to a running or a not running environment. Dependent from this the state eOFFLINE or eONLINE is occupied.

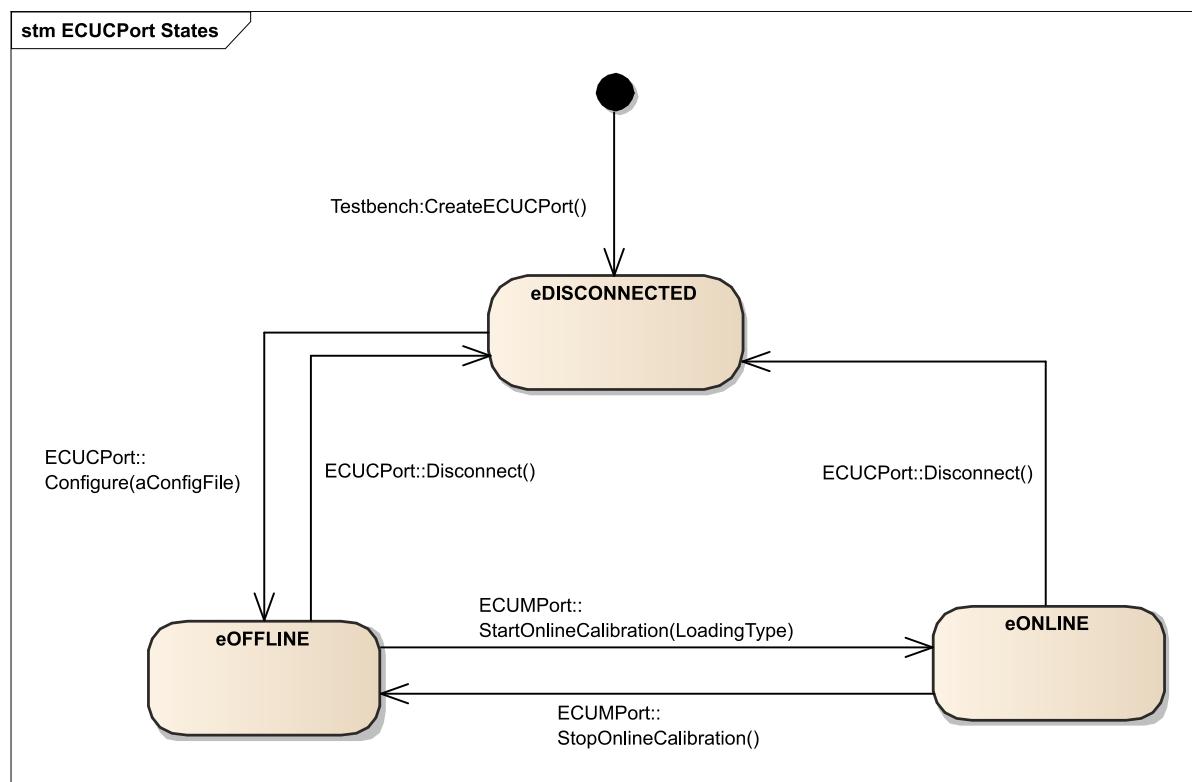


Figure 176: States of the ECUC port

Table 53 ECUCPort states

	eDISCONNECTED	eOFFLINE	eONLINE
Method CalculateRefPageCRC		x	x
Method CalculateWorkPageCRC		x	x
Method CheckVariableNames		x	x
Method Configure	x		
Method Disconnect		x	x
Method GetDataType		x	x
Property getState	x	x	x
Method GetVariableInfo		x	x
Property getVariableNames		x	x
Method IsReadable		x	x
Method IsWriteable		x	x
Method NumberOfPages		x	x
Method Read		x	x
Method StartOnlineCalibration		x	
Method StopOnlineCalibration			x
Method SwitchToRefPage		x	x
Method SwitchToWorkPage		x	x
Method Write		x	x

State transitions only take place, if all preconditions are fulfilled and no error occur during execution of the state changing method. Otherwise the state is not changed, i.e. the state remains the same as before the method was called. Methods, which trigger a state change, will throw an exception if the state change could not be processed successfully.

5.5.3 USAGE OF ECUCPORT

5.5.3.1 CREATION AND CONFIGURATION

Instances of `ECUCPort` are created by calls to method `CreateECUCPort` of an `ECUCPortFactory` object. It returns an `ECUCPort` instance with the name given as parameter. The `ECUCPortFactory` object can be obtained from the `Testbench` object that provides a corresponding property.

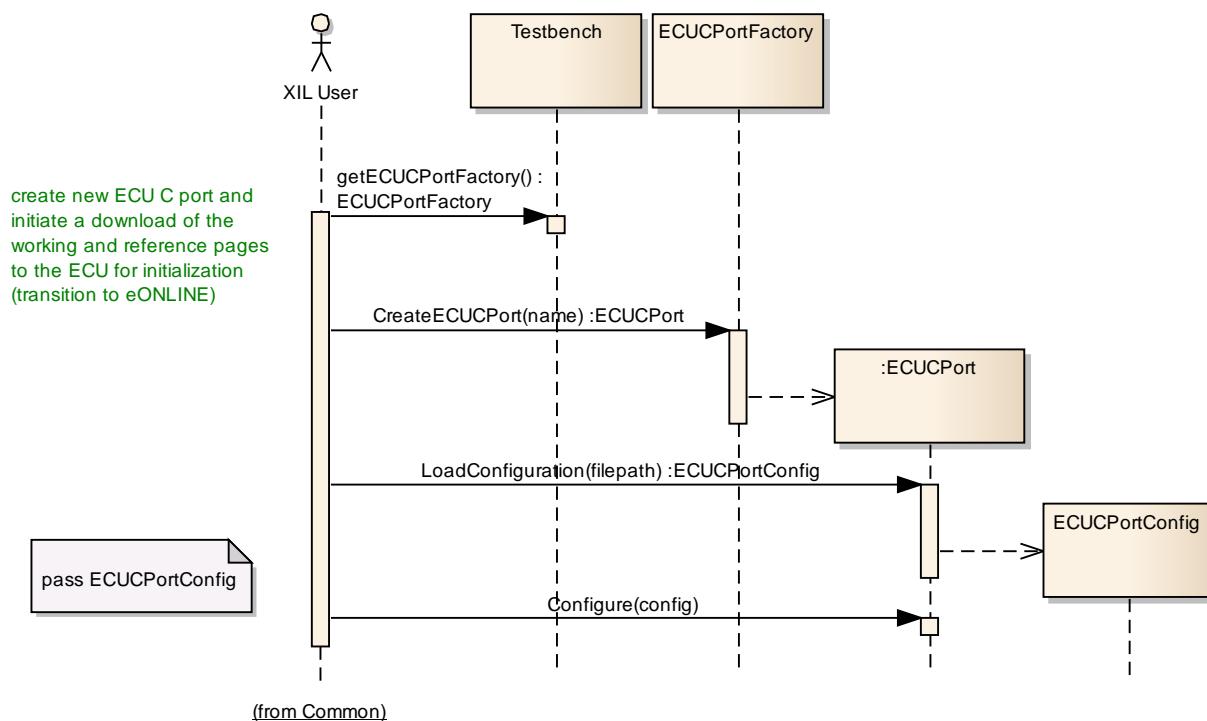


Figure 177 Process of ECUCPort creation and configuration

Configuration of the new port instance is a two-step process (see [Figure 177](#)). First a vendor specific configuration file has to be loaded via method `LoadConfiguration`. Besides other vendor specific port settings this file particularly specifies the A2L file and the ECU image file to be used by the port. Both settings can be queried from or changed by corresponding properties of the `ECUCPortConfig` object that is returned by `LoadConfiguration`.

The second step of configuration is calling the `Configure` method and passing the `ECUCPortConfig` object created in step one. This establishes a connection to the calibration tool or hardware and activates the passed configuration. On successful configuration the port's state is set to `eOFFLINE`.

5.5.3.2 ACCESSING ECU PARAMETERS

It is assumed that the client has a valid `ECUCPort` instance. The `ECUCPort` allows reading and writing of parameter values of the ECU. If a parameter value is readable can be determined by using the `isReadable` function. If a parameter also can be modified can be determined using the `isWritable` function.

The following example shows how to read and write a scalar float ECU parameter value (see [Figure 178](#)). First the connection to the ECU is set up using the `Start()` method. The value of the `LoadingType` parameter is not important for this example, any value can be used. Then the data type of the variable is fetched using the `GetDataType()` method. The following assumes that the chosen parameter value is readable and writable.

Then the new parameter value is written to the ECU. After writing, the current value of the parameter on the ECU is read back in order to check if it is the same value as the one which has been written. After that the connection to the ECU is stopped and the `ECUCPort` goes offline.

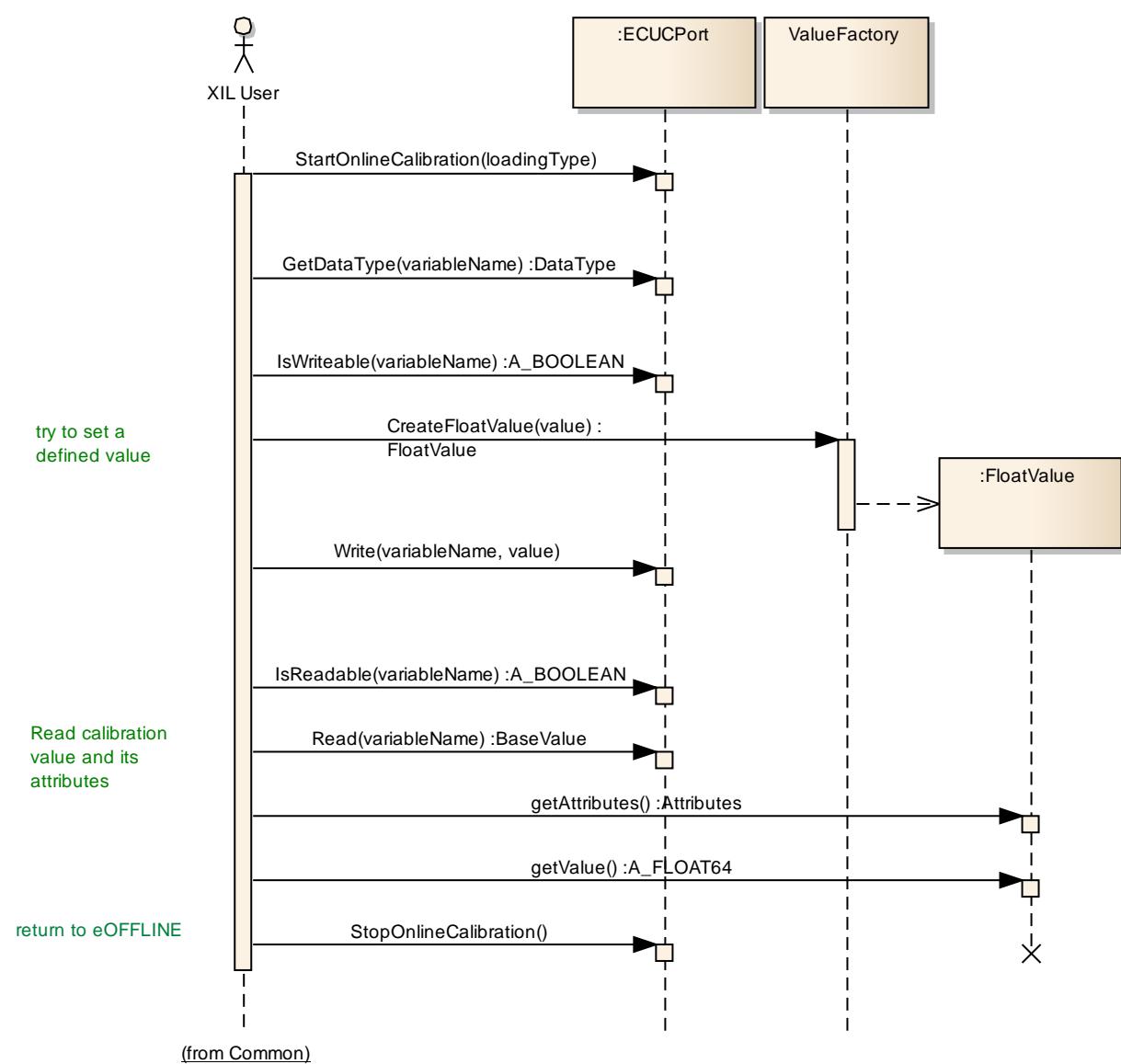


Figure 178: Read and write a scalar float ECU parameter

5.5.3.3 GETTING THE LIST OF VARIABLES OF THE ECUCPORT

This example describes how a user can get the names of all parameters supported by this ECUCPort instance. A call of the `VariableNames()` property returns a list of all parameter names.

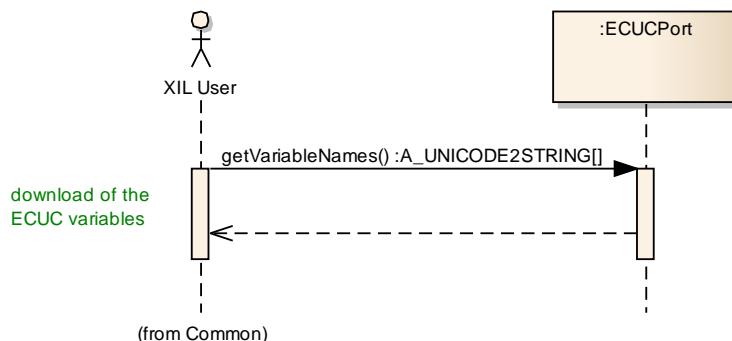


Figure 179: Get the list of parameters of an ECUCPort instance

5.5.3.4 MANAGE ECU MEMORY PAGES

Another block of functionality of the `ECUCPort` is the management of ECU memory pages. Most MC tools support the handling of several memory pages. Each of these memory pages is able hold all parameter values of the ECU. Two of the most popular memory pages are the working page and a read-only reference page. The following list gives a short overview of the memory page management functions of the `ECUCPort`:

- The `NumberOfPages()` method returns the number of pages which the current MC system supports. Most MC systems support 2 memory pages, a reference and a working page.
- If more than 1 memory page is supported by the MC system, the `SwitchToRefPage()` method allows to switch from the working page to the reference page, and the `SwitchToWorkPage()` method allows to switch from the reference page to the working page.
- To check if the contents of two memory pages are equal, the `CalculateRefPageCRC()` and `CalculateWorkPageCRC()` methods can be used.
- In order to set up a connection with the ECU, the `Start()` method must be used. The `LoadingType` parameter defines if the content of the current memory page is downloaded to the ECU (`eDOWNLOAD`) or if the content of the actual page is filled with the data coming from the ECU memory (`eUPLOAD`).

Setting a variable value with the `Write` method in state `eOFFLINE` sets the value on the actual page. This value can be written to ECU with the `Start(eDOWNLOAD)` method.

Setting the value of the same variable in state `eONLINE` changes the value on the ECU directly.

The following example shows how to handle memory pages.

After creation of the `ECUCPort` the transition to `eONLINE` is performed. Then the number of pages is fetched. In this case a value of 2 is expected, to make sure that a working and a reference page exist. Then the reference page is made the current memory page and the checksum is calculated. Then a parameter value from the ECU is read. After a switch to the working page, the checksum of the working page is calculated. The value of the same parameter is read again – this time coming from the working page. The 2 values of the same variable can differ because they are coming from different memory pages. Then a new value is written to this variable and the checksum is calculated again. Now the value of the parameter on the ECU should be different to the checksum of the same page before. Finally, a call of the `Stop()` method executes a transition to the ECU port state to `eOFFLINE`.

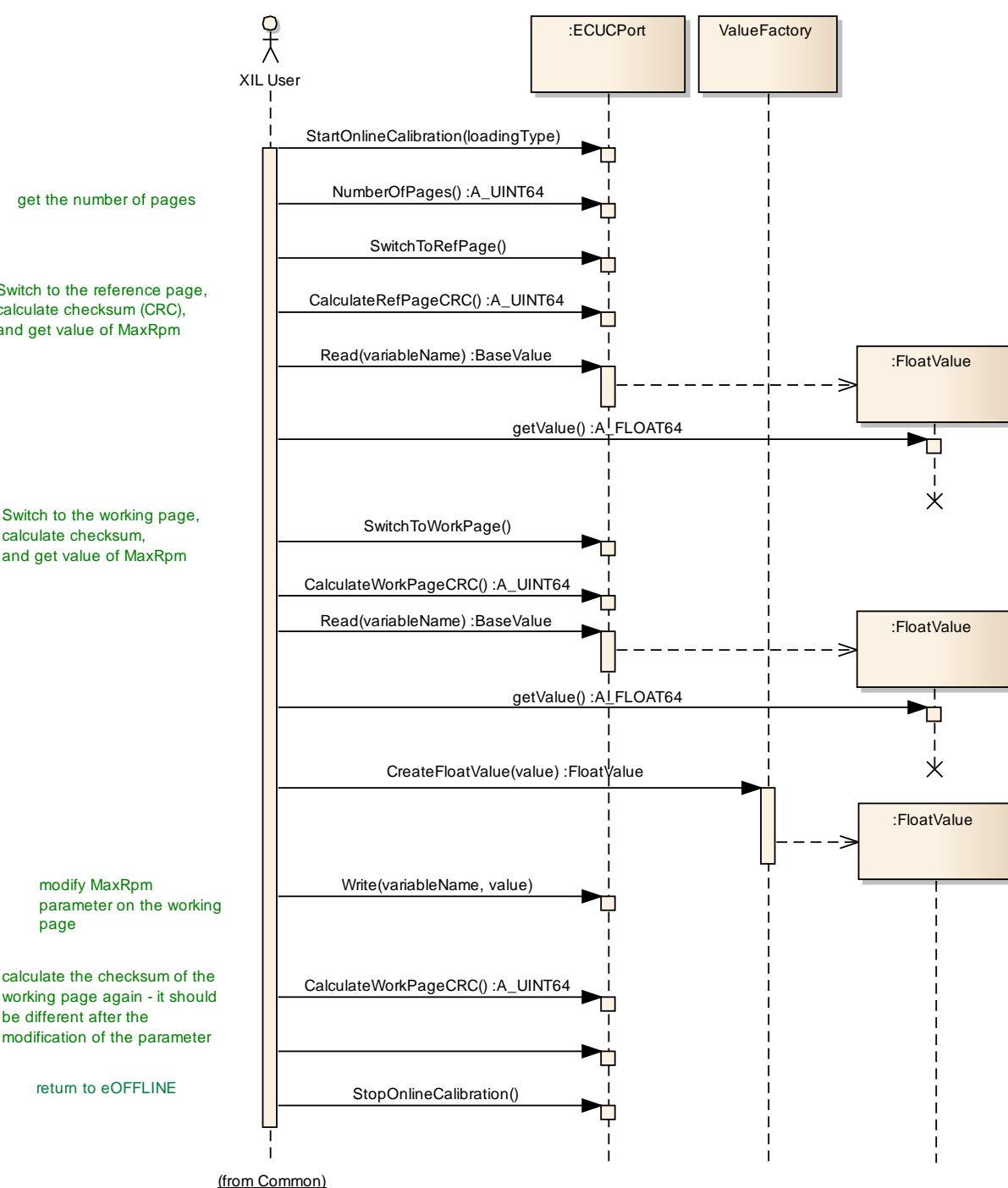


Figure 180: Handling of memory pages

5.6 EES PORT

5.6.1 USER CONCEPT

5.6.1.1 GENERAL

Electrical Error Simulation on the XIL System

The pins of the system under test (SUT) are connected to the XIL system that provides power supply lines, communication busses like CAN, and simulated sensors and actuators. But test cases may not only comprise checking the behavior of the SUT in a fully functional environment. It is also important to check the SUT in case of electrical errors on the input and output pins. The question is: How does the SUT behave if the sensors and actuators are not working correctly or if they are not connected correctly?

Typical errors that have to be tested are electrical problems, mainly caused by wiring. To generate this class of errors the connections between the SUT and its environment (sensors, actuators, power supply, busses) have to be disturbed by an appropriate hardware system. This is the task of the so-called electrical error simulator (EES, see [Figure 181](#)).

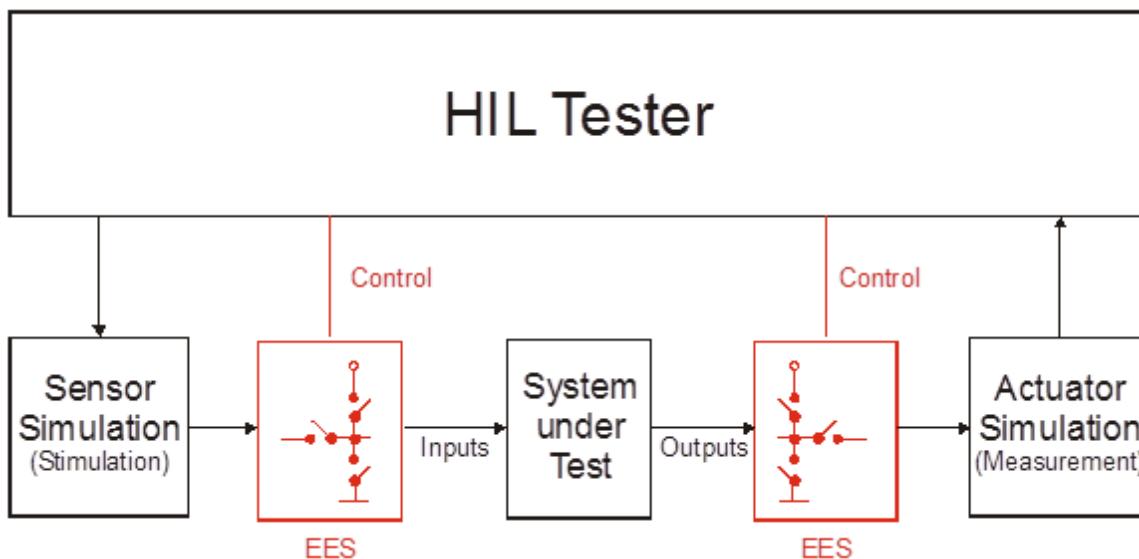


Figure 181: Electrical Error Simulation is used to disturb the signals between the XIL system and the SUT

The electrical error simulator creates typical wiring errors like loose contacts, broken cables, short-circuits to neighboring pins, to ground (chassis) or to battery voltage. EES is provided by a special hardware in the XIL tester. But EES has not to be a separate component in general. It may be also integrated in a comprehensive XIL hardware.

The EES hardware is controlled by the test cases during the test, not by the real-time model. The XIL API EES port provides a general API for electrical error simulation hardware. The API hides the specific API of the used hardware, its driver software, and the communication between the machine the test is running on and the EES hardware. The EES port provides a defined set of functionality in an abstract manner. It is designed from the test case writer's

point of view. Thus, the test case writer deals with some abstract error functionality. It is not necessary to know the technical details of the EES hardware.

Functional Principle of the EES Port

The general functional principle of the EES port is: A sequence of errors is defined using the XIL API by the test script. This is called the error configuration and may be stored in an XML file. The error configuration is downloaded to the specific EES hardware or software. The execution of the error sequence is completely transparent for the EES port user. It is done by the vendor-specific hardware, software, or driver.

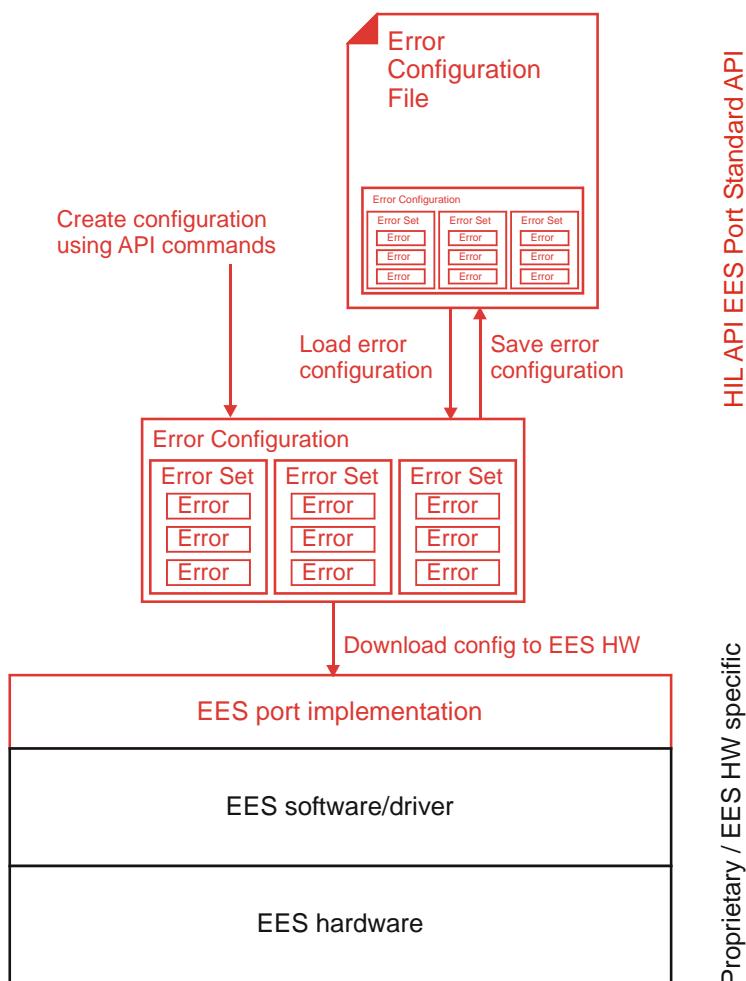


Figure 182: Error configurations are defined by EES ports and downloaded for execution to the vendor-specific EES hardware respective software

XIL API EES port mainly deals with the configuration of errors and starting a formerly configured error sequence by downloading (see [Figure 182](#)). Therefore the XIL API EES port is independent from a concrete EES implementation.

5.6.1.2 CONFIGURATION AND EXECUTION OF ELECTRICAL ERRORS

Error Configuration

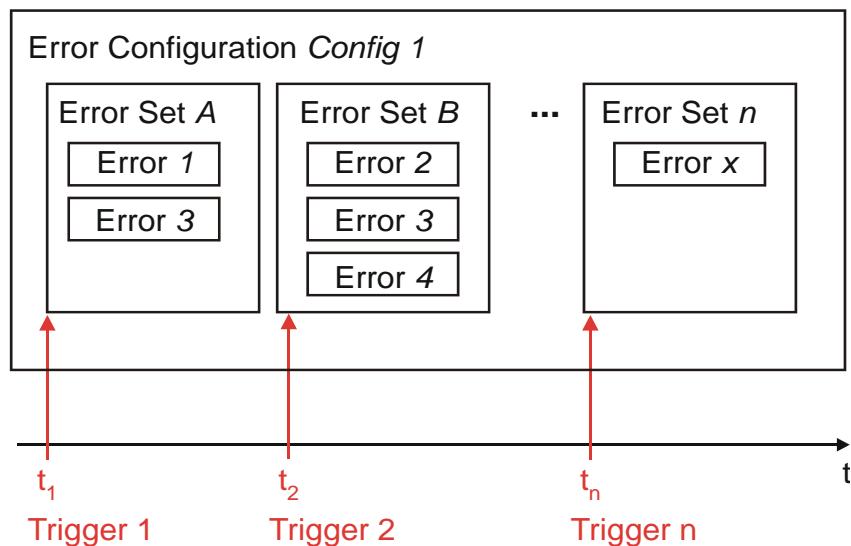


Figure 183: An error configuration comprises a sequence of error sets with several errors

An error is a defined disturbance of one or more electrical signals, typically pins of the SUT. E.g. an error may disturb a signal by interrupting the line and replacing it with a resistor. More than one error for different signals may be in effect at the same time, but each signal may only be used once in an error set. All errors that start at the same time are put together in an error set. An EES configuration comprises of a sequence of error sets (see [Figure 183](#)).

The EES Port can be used in two logical modes:

1. In the static mode the error sets defined in the error configuration are triggered in the defined sequence. This mode is to prefer if the timing is most important.
2. In the dynamic mode error sets can be added dynamically to the error configuration while the port is in the states eDOWNLOADED or eACTIVATED. This mode is to prefer if flexibility - i.e. to create variations of test cases during the test execution – is most important. This mode was introduced with XIL 2.0.

Error configuration instances (and the error sets they contain) must not be changed, as long as they are assigned to an instance of the class `EESPort`. Although, new `ErrorSets` can be dynamically added to the `ErrorConfiguration` after it is downloaded to the hardware. The assignment of an error configuration to an `EES` port is not possible during the execution of an error configuration, but `Update` of an error configuration is possible. The assignment of an error configuration to an `EESPort` is possible in state `eCONNECTED` only and `Update` is possible in states `eDOWNLOADED` and `eACTIVATED` only.

Execution of an Error Configuration

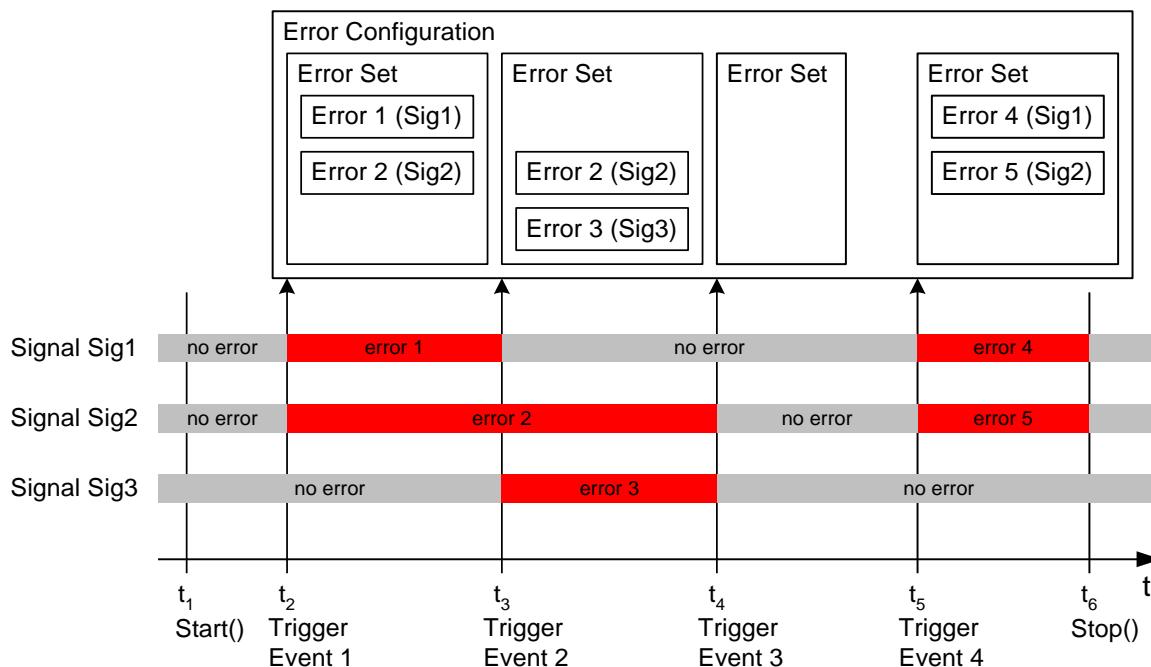


Figure 184: Example for the execution of an error configuration

To execute an error configuration, it has to be downloaded and started by the test case. When an error configuration is executed, the error sets are executed in the defined sequence (see [Figure 184](#)). That means the first error set is activated when the trigger for the first error set becomes true. All other triggers are not considered so far. The errors in the error set stay active as far as the trigger of the next error set becomes true.

The sequence of error sets is defined by the error configuration. It does not depend on the sequence the triggers of the error sets are fired. The triggers only determine when the next error set replaces the currently active error set.

The error configuration has to be stopped by the test case using the XIL API. The last error set remains active as long as the error configuration is not stopped. To get a defined end of signal disturbance, an empty error set can be used as the last error set in the configuration. Empty error sets can also be used to create error-free phases (refer to [Figure 184](#) for an example).

If an error is defined in the same way in two consecutive error sets, the error will stay in action. There is no restart of the error or any other kind of influence when one error set is replaced by the consecutive error set containing the same error for the same signal.

Download of an Error Configuration

The EES error configuration has to be downloaded before it can be executed. Download means that the configuration has to be completely passed by the EES port to the specific EES driver, software, and hardware system. Typically the configuration will be physically downloaded to the hardware and executed there. But in general, this is not required by the XIL API definition. It is also possible that an error configuration is executed by the driver or another software system on the same PC.

Therefore, the conceptual sequence to create and use an error configuration is:

1. An error configuration is created by means of the XIL API EES port (construction by API calls or loading from a file).

2. Now the error configuration is stored on the computer the XIL API is running on and may be changed.
3. Then the error configuration is downloaded to the EES system, a specific software/hardware system. The error configuration can be dynamically extended after the download.
4. After starting the error configuration in the test case (using XIL API), the defined sequence of errors is executed. Execution is independent from the test case and the XIL API.
5. When the XIL API stops the execution, all kinds of disturbance immediately stop. Possibly the last error set of the error configuration remains active until the execution is stopped.

A downloaded error configuration can be used several times.

5.6.1.3 TRIGGERS IN EES

The EES system uses trigger events to switch from one error set to another error set. These triggers are handled by the EES hardware and software. They are not defined by the XIL API. And there are also no means to define trigger conditions for EES error sets in the EES port. In an EES error configuration only the type of the awaited trigger is defined. The type of a trigger can be thought as the trigger input connection of an appropriate hardware. But in fact, the trigger may be controlled by software also.

From EES port's point of view an EES system has three possible trigger inputs:

- Manual trigger: This trigger is fired by the controlling test script. The EES port offers a method to fire this trigger.
- Hardware trigger: The hardware trigger reacts on some kind of electrical trigger line of the EES hardware. Further details are not defined by the XIL API. It is just expected that the EES hardware has some kind of a hardware trigger input.
- Software trigger: The software trigger reacts on a trigger signal defined in the model or another software part of the XIL system. It is not defined by XIL API or the EES error configuration how the EES system is associated with the software system.

A manual trigger is fired by calling the method `Trigger` of the `EESPort`. This leads to leaving the actual error set and activation of the next error set. When a manual trigger is fired and there are no more error sets left to activate, or the trigger type of the next error set is not manual trigger, an exception is thrown.

In case of dynamic errors (with configured duration) the fired trigger leads to leaving the actual error set and activation of the next error set, independent of the configured duration of the actual error set, which means possibly before the duration is reached.

If an additional configuration of triggers is needed by the EES system, this has to be done using the EES specific software interface. There are no means in XIL API so far to define additional options for the triggers.

5.6.1.4 ELECTRICAL ERRORS

An error is defined by several independent aspects: the error category, the error type, and the option to disturb with or without load.

Error Category

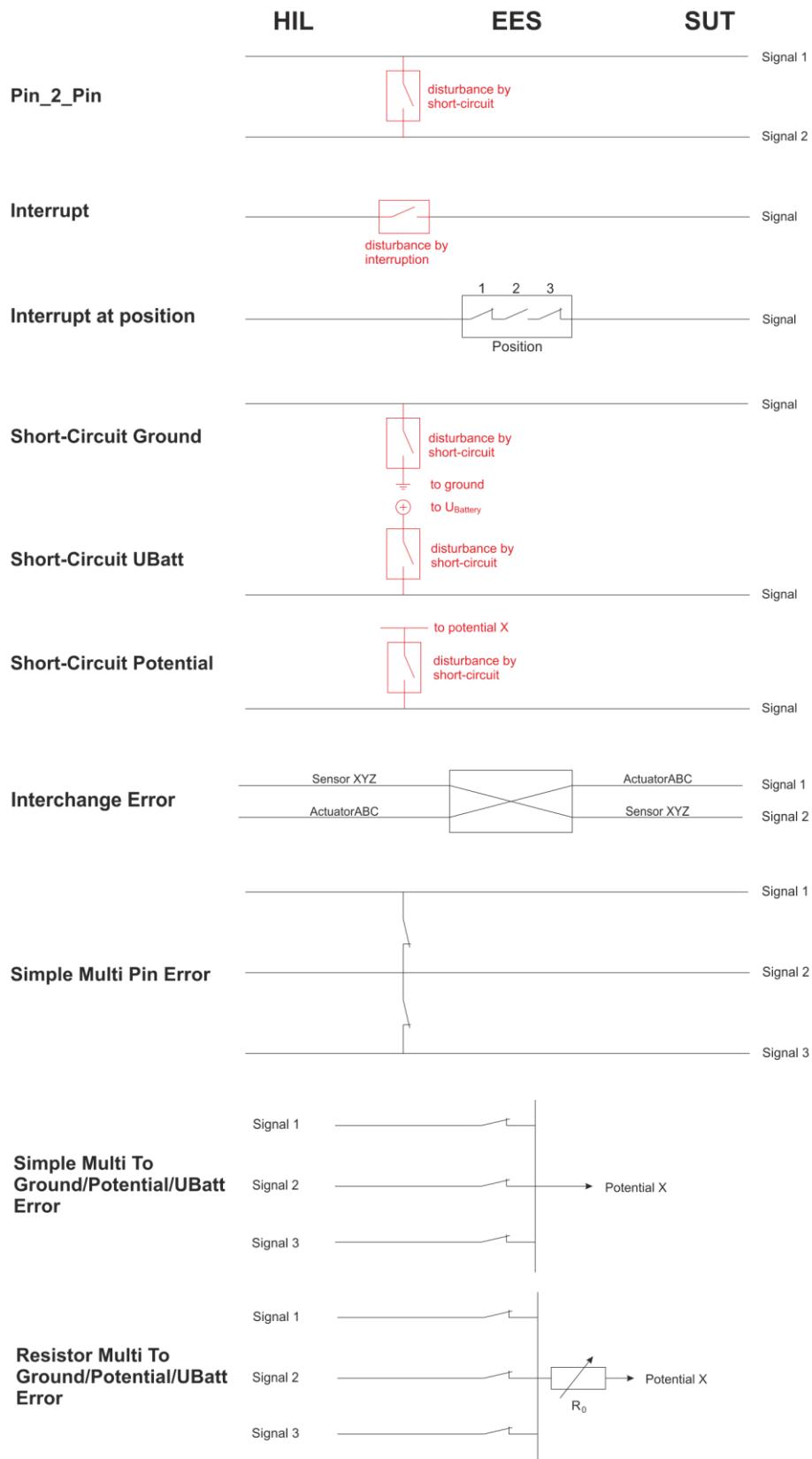


Figure 185: Illustration of the error categories defined by EES port

The error category defines how a signal should be disturbed. A signal is interrupted or connected to other signals or a potential (see [Figure 185](#)). The way the interruption or short-circuit is provided is not defined by the error category (but by the error type, see section [Error Type](#)).

Typically the error category affects one signal. Only in case of a pin to pin, interchanged or multi pin error two or more signals are affected.

The error short-circuit to potential is the generalized form of a short-circuit error. For this category of errors the EES hardware has to provide additional potentials beside Ubattery and ground. Multiple potentials identified by unique numbers may be supported. Ubattery and ground are covered by separate categories because of their importance.

Error Type

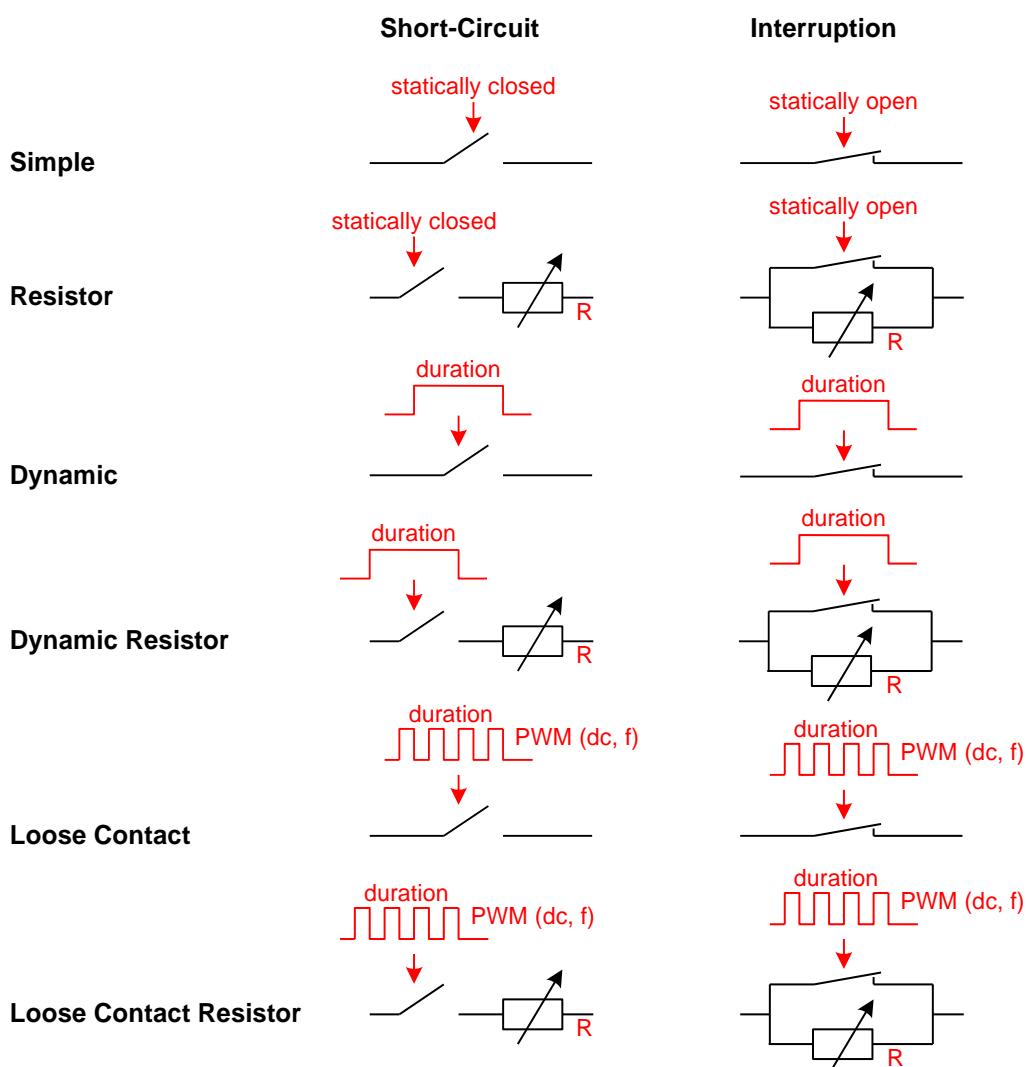


Figure 186: Illustration of the error types defined by EES port

The error type defines the disturbance itself. There are several possibilities that differ in the dynamic of the disturbance (static, for a defined duration, controlled by a PWM signal) and the resistance in case of the error (defined resistance or completely open/closed).

The concrete circuit also differs between short-circuit errors and interrupt errors, because in one case the error is caused by closing a connection, in the other by opening the connection. Nevertheless, the idea of an error of the same error type is the same in both cases.

[Figure 186](#) shows the available error types and the principal circuits used for short-circuits and interrupts.

With or without Load

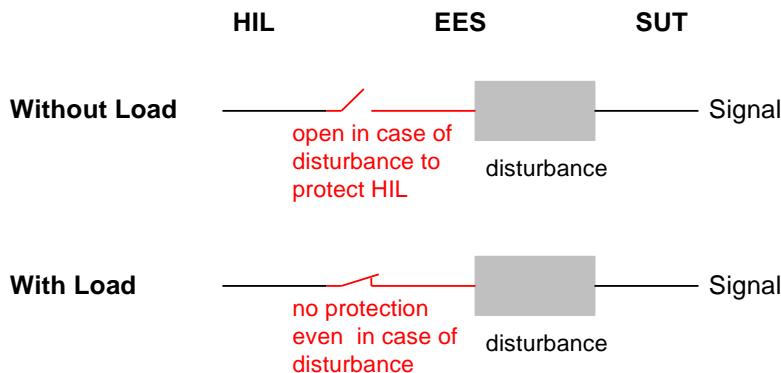


Figure 187: Illustration of the option with load and without load

The option “with load” or “without load” is an additional aspect of a signal. This aspect is orthogonal to error category and error type and can be freely chosen for almost every kind of error. Only interruptions do not provide this option because technically it does not make any sense in this case.

Background: If a signal between the XIL and the SUT is disturbed by the EES hardware, not only the SUT has to deal with the disturbance. A short-circuit for example effects the XIL hardware, too. To protect the XIL hardware the EES can open the connection between XIL and EES. Thus, the disturbance has an effect on the SUT but cannot damage the XIL.

[Figure 187](#) shows the principal circuit of the with/without load protection in the EES hardware.

5.6.1.5 API

The EES port API consists of several classes to control the EES system and to create, store, load, and represent error configurations. In the following chapters the most important classes of the EES port and the relations between them are described.

EES Port

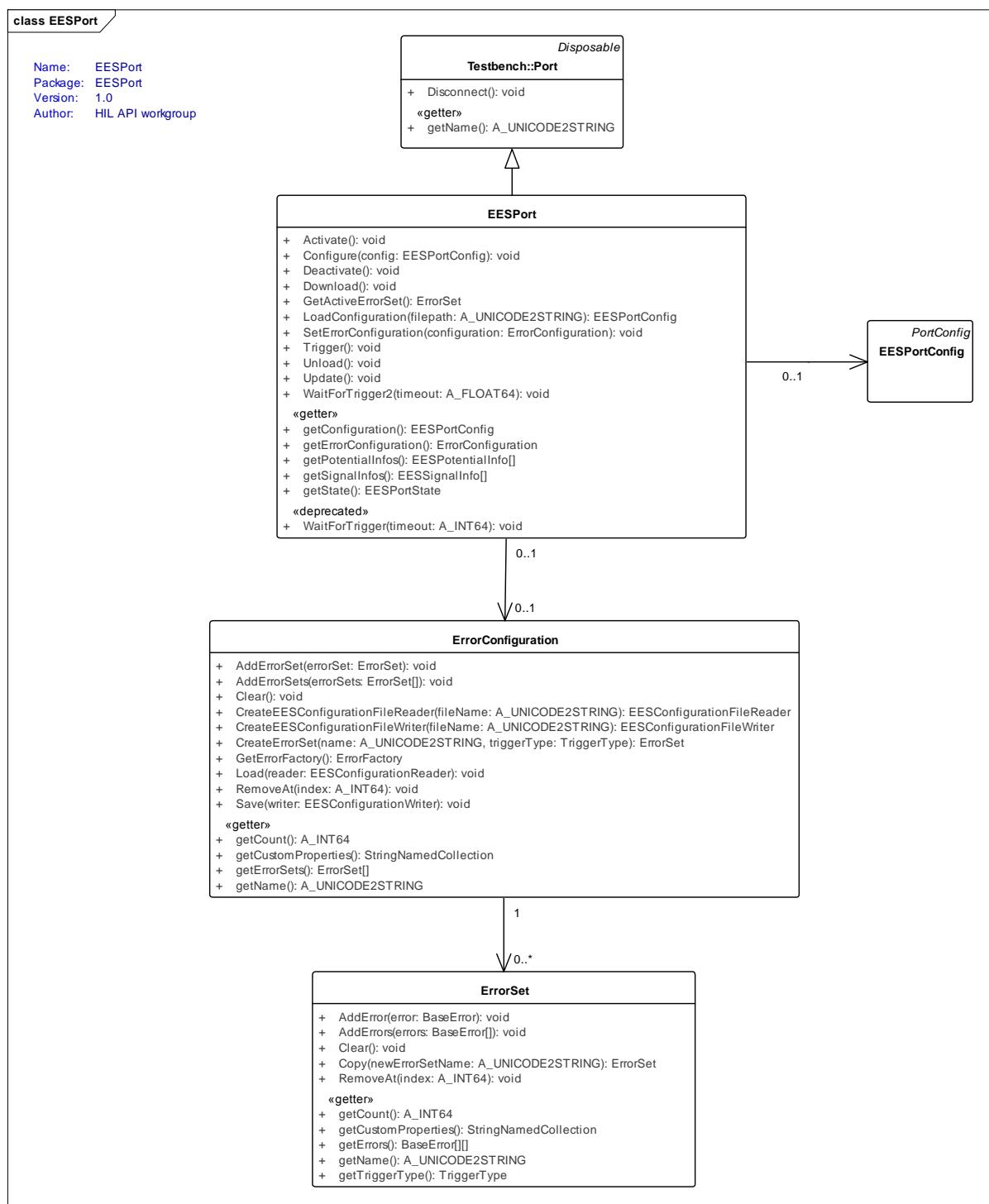


Figure 188: EES port classes

The EES port itself is represented by the class **EESPort**. This class is derived from the general XIL API Port class. **EESPort** provides methods to download an error configuration, to start and stop the execution of a downloaded error configuration on the EES system, and to trigger the EES system manually. This is the way a trigger of type "MANUAL" is fired.

To synchronize the test run of the test script with the execution of the error configuration, the synchronous method `WaitForTrigger2` can be used. This method waits until the next trigger event defined in the error configuration. Alternatively the method returns with an error when the timeout time is reached.

An error configuration can be created using the constructor of the class `ErrorConfiguration`. It is possible to create several error configurations. But only one error configuration can be assigned to the `EESPort` instance. The assignment overrides an older assignment. Assigned error configurations can be changed. But after downloading the error configuration to the EES system no further changes are considered. Nevertheless, the error configuration may be downloaded again.

Error Configuration

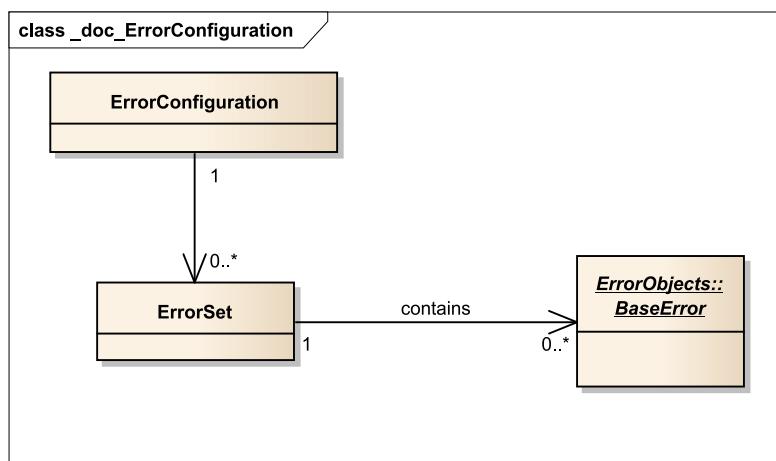


Figure 189: Classes used to represent an error configuration

According to the structure of an error configuration there exists one class for the error configuration itself (`ErrorConfiguration`), one for the error sets (`ErrorSet`), and several classes for errors of the different error types (`SimpleError`, ...).

Only the error configuration class can be instantiated by the constructor. All other classes are constructed using the factory method respective the error factory class of the error configuration object. Therefore error sets and error objects exist only in the context of an error configuration and will be destroyed automatically when the error configuration is destroyed by the user (using the destructor of `ErrorConfiguration`).

Errors (`SimpleError`, ...) are created using the factory provided by the error configuration instance and then assigned to one or more error sets. It is not allowed to assign an error to another error configuration.

Some properties of error objects, error configuration and error sets (like name) cannot be changed after creation. These properties are set at creation time by the factory class. In this sense, these objects are read-only, but `ErrorConfiguration`, `ErrorSet` and `BaseError` Classes provide a property to add vendor specific properties. These properties are saved as `StringNamedCollection`.

Error Objects

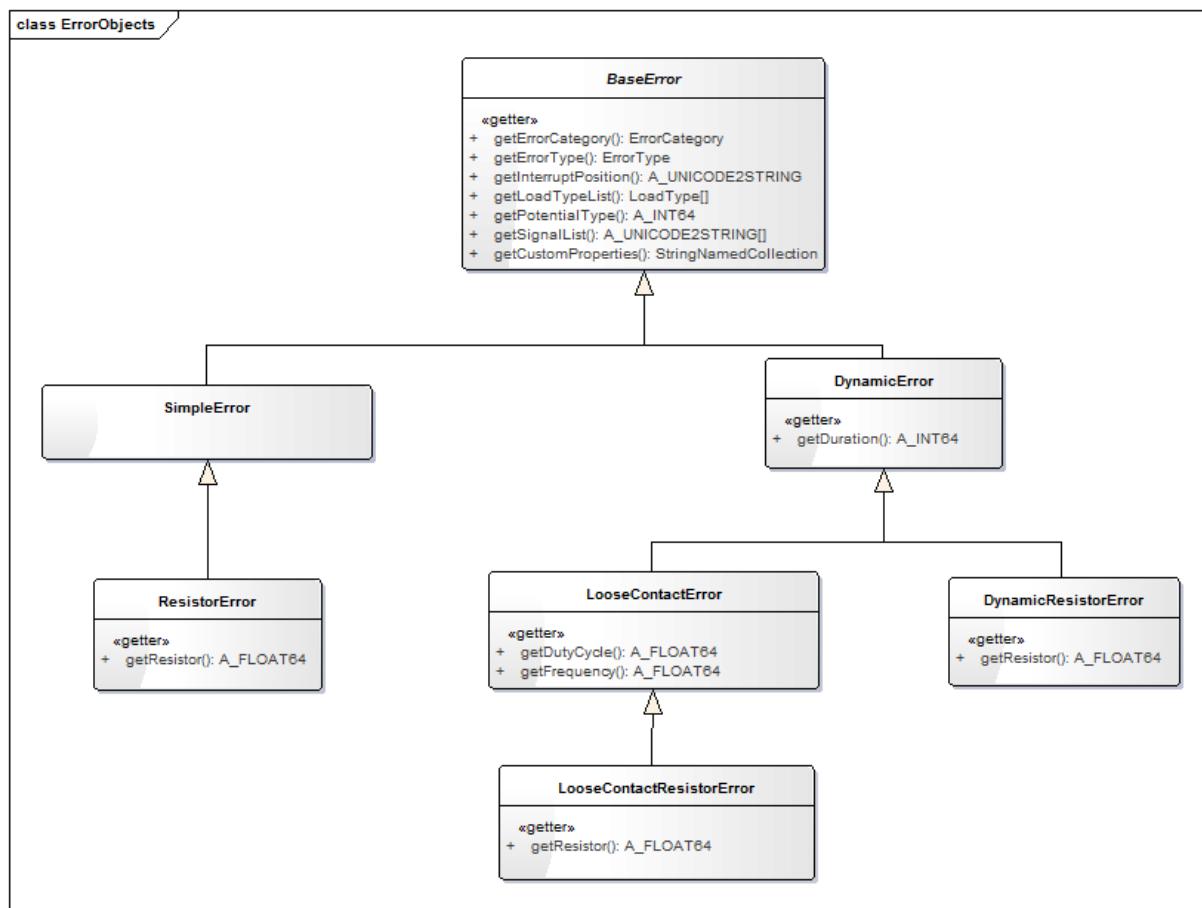


Figure 190: Class hierarchy of error objects

The six error classes represent the six error types. Other differentiation characteristics of errors like error category and option with/without load are stored as attributes in these classes.

The error classes are hierarchically organized with a common base class. The base class BaseError is abstract and cannot be instantiated. Attributes and variables of this type store an instance of an arbitrary error.

Creation of Error Objects

Table 54 Error Builder

ErrorBuilder	Methode		ErrorObject received from ToBaseError
SimpleErrorBuilder			SimpleError
	WithResistor		ResistorError
DynamicErrorBuilder			DynamicError
	WithResistor		DynamicResistorError
	WithFrequency		LooseContactError
		ResistorErrorBuilder : WithResistor	LooseContactResistorError
ResistorErrorBuilder			SimpleError
	WithResistor		ResistorError
DynamicErrorBuilder2			DynamicError
	WithFrequency		LooseContactError

Factory and builder classes are used to create a specific error object. The result of the builder is an instance of one error class (see section [Error Objects](#)).

In principle, error objects are created by the following sequence:

1. Fetch factory `ErrorFactory` from error configuration. Use the error configuration the new error should belong to.
2. Choose the error category and define the affected signal or signals. For each signal the option with or without load is defined, too. The `ErrorFactory` will return a `SpecificErrorFactory`.
3. Choose whether you want to create a simple or a dynamic error. This is one aspect of the error type. The `SpecificErrorFactory` will return an object of class `SimpleErrorBuilder` or class `DynamicErrorBuilder`.
4. Configure now the error type using the error builder object. Possibly the error builder returns another error builder so that the building process may comprise several levels.
5. The method `ToBaseError()` returns the configured error object. `ToBaseError()` is available in all error builder classes.

Factories and builder objects cannot be created or destroyed by the user. They must be fetched again from the according father object each time a new error should be created.

This multi-level structure of factory/builder objects is designed to be used in a one line statement to create an error object with all its characteristics. The example code demonstrates how this works in the supported programming languages.

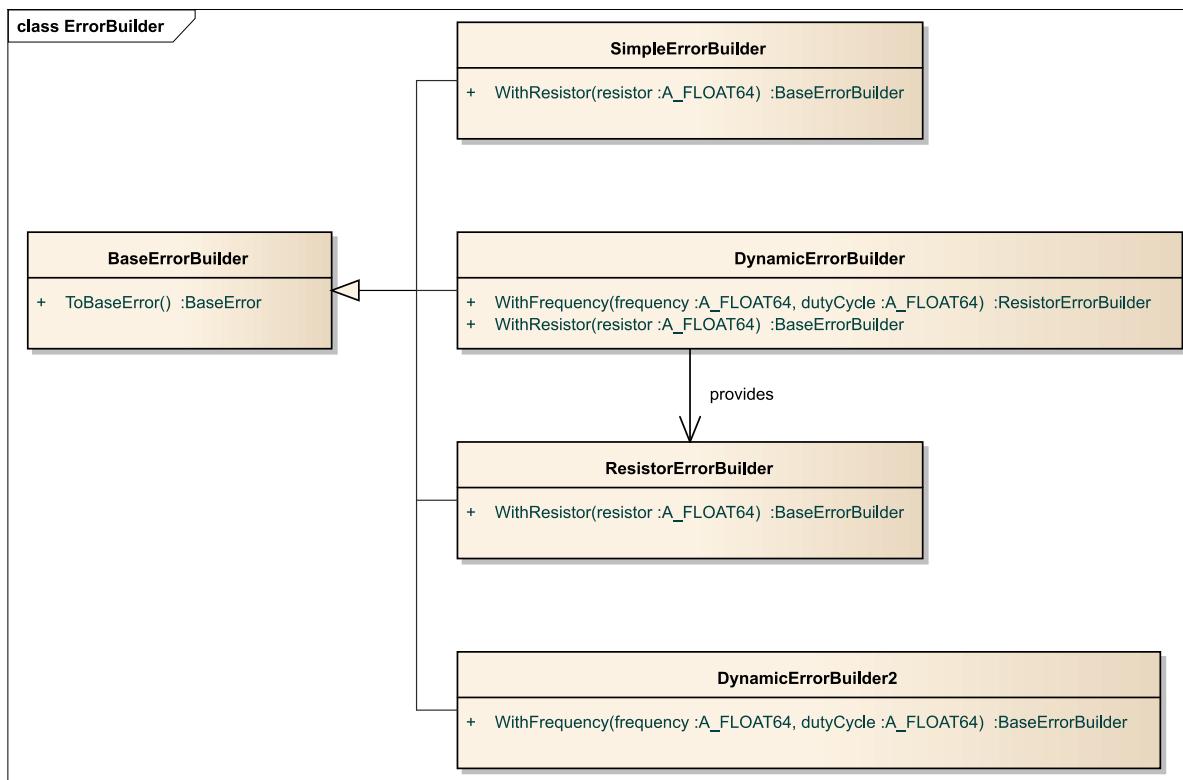


Figure 191: Error Builder

Reader and Writer for EES Configuration Files

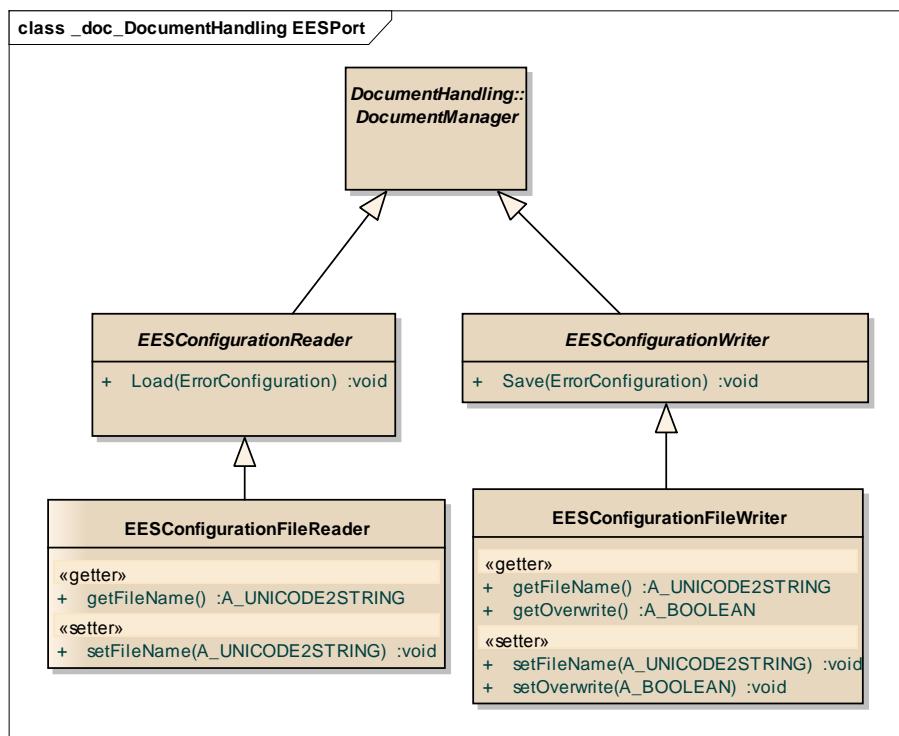


Figure 192: Reader and writer classes for error configuration files

Error configurations can be externally stored, normally in files. To allow different storage types and formats, reading and writing is handled by abstract reader and writer classes. For the EES error configuration these are the abstract classes `EESConfigurationReader` and `EESConfigurationWriter`. Both classes are derived from the common XIL API document handler class `DocumentManager`.

XIL API supports an XML format to store complete error configuration in a file. The schema definition of this XML file is part of the XIL API standard (`EESConfiguration.xsd`). The specific reader and writer classes are `EESConfiguration.FileReader` and `EESConfiguration.FileWriter`.

5.6.1.6 STATES OF THE EES PORT

[Figure 193](#) shows the state diagram of the EES port.

There are four states - `eDISCONNECTED`, `eCONNECTED`, `eDOWNLOADED` and `eACTIVATED`. After creation, the EES Port instance is always in state `eDISCONNECTED`. Following the states are explained:

Table 55 States of the EESPort

State	Description
<code>eDISCONNECTED</code>	The initial state of EES port. The port is disconnected from the hardware. In this state it is possible to configure the port. Electrical error simulation is not possible in this state, because the EES hardware is not connected.
<code>eCONNECTED</code>	The port is connected with the configured hardware. In this state it is possible to assign an error configuration.
<code>eDOWNLOADED</code>	An error configuration is downloaded. In this state it is possible to start the error configuration or dynamically add error sets to the configuration. It is not possible to edit or remove error sets from the downloaded error configuration.
<code>eACTIVATED</code>	The downloaded error configuration is started. The EES system waits for the first defined trigger event to execute the first error set in the error configuration. This may be also a manual trigger, fired by using the method <code>Trigger</code> of the EES port object. The error sets in the error configuration can be triggered in the given sequence. New error sets can be added.

The `Configure(aConfigFile)` method switches from the `eDISCONNECTED` state to the `eCONNECTED` state.

The `Disconnect()` method switches from `eCONNECTED` state back to the `eDISCONNECTED` state.

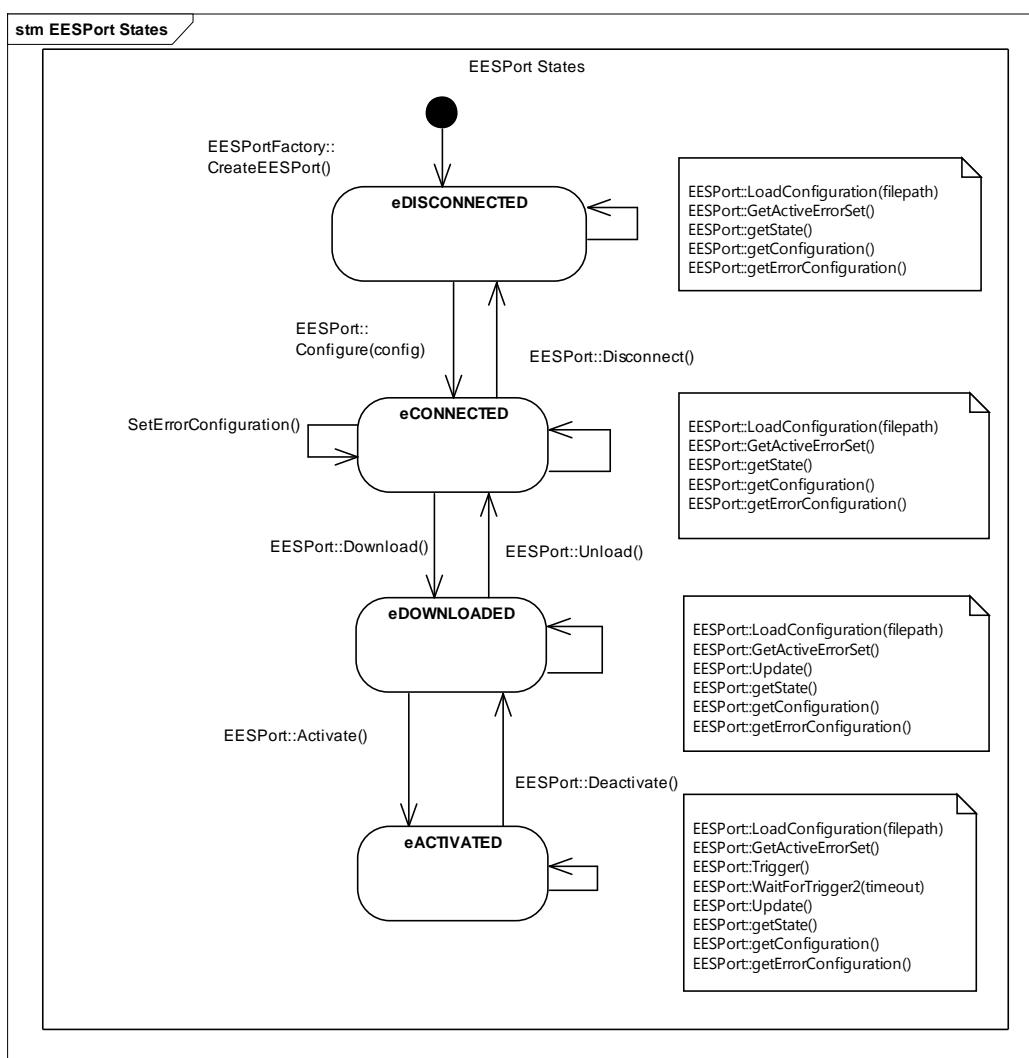


Figure 193: EESPort States

Table 56 EESPort states

	eDISCONNECTED	eCONNECTED	eDOWNLOADED	eACTIVATED
Method Activate			x	
Method Configure	x			
Method Deactivate				x
Method Disconnect		x		
Method Download		x		
Method GetActiveErrorSet	x	x	x	x
Property getConfiguration	x	x	x	x
Property getErrorConfiguration	x	x	x	x
Property getPotentialInfos		x	x	x
Property getSignalInfos		x	x	x
Method SetErrorConfiguration		x		
Method LoadConfiguration	x	x	x	x
Method Trigger				x
Method Unload			x	
Method Update			x	x
Method WaitForTrigger2				x

State transitions only take place, if all preconditions are fulfilled and no error occur during execution of the state changing method. Otherwise the state is not changed, i.e. the state remains the same as before the method was called. Methods, which trigger a state change, will throw an exception if the state change could not be processed successfully.

Download() and Update() calls shall check that the errors from the error configuration can be applied by the underlying EES system and throw an exception in case an unsupported feature or parametrization is used.

- If Download() fails, the port state remains eCONNECTED
- If Update() fails, the port state remains unchanged, i.e. eDOWNLOADED or eACTIVATED, and the downloaded error configuration is not modified.

The Trigger method returns only when the new ErrorSet is also electrically active. Likewise, the Deactivate method only returns when all errors have also been electrically deactivated. Their implementation either performs an electrical measurement of when the errors were actually electrically activated or deactivated, or it waits until the operate time of the relay has elapsed, in accordance with the operate time specified by the relay manufacturer. If several relays are operated, the maximum operate time is taken into account.

The user

- creates a port
- connects the port to a hardware
- sets, downloads and activates an error configuration
- triggers the error sets and/or waits for next trigger to synchronize the execution of the test script
- deactivates the error set, unloads the error configuration
- Disconnects from the hardware.

Configuration (port configuration) of the EES port is not possible, not necessary, and not reasonable during execution of an error configuration. The assignment of an error configuration to an EES port is only possible in the state eCONNECTED. The user can dynamically add error sets to the error configuration in the states eDOWNLOADED and eACTIVATED. Adding error sets does not interrupt or affect the errors currently active in the EES hardware.

5.6.2 USAGE OF THIS PORT

5.6.2.1 PORT CREATION AND CONFIGURATION

Instances of `EESPort` are created by calls to method `CreateEESPort` of an `EESPortFactory` object. It returns an `EESPort` instance with the name given as parameter. The `EESPortFactory` object can be obtained from the `Testbench` object that provides a corresponding property.

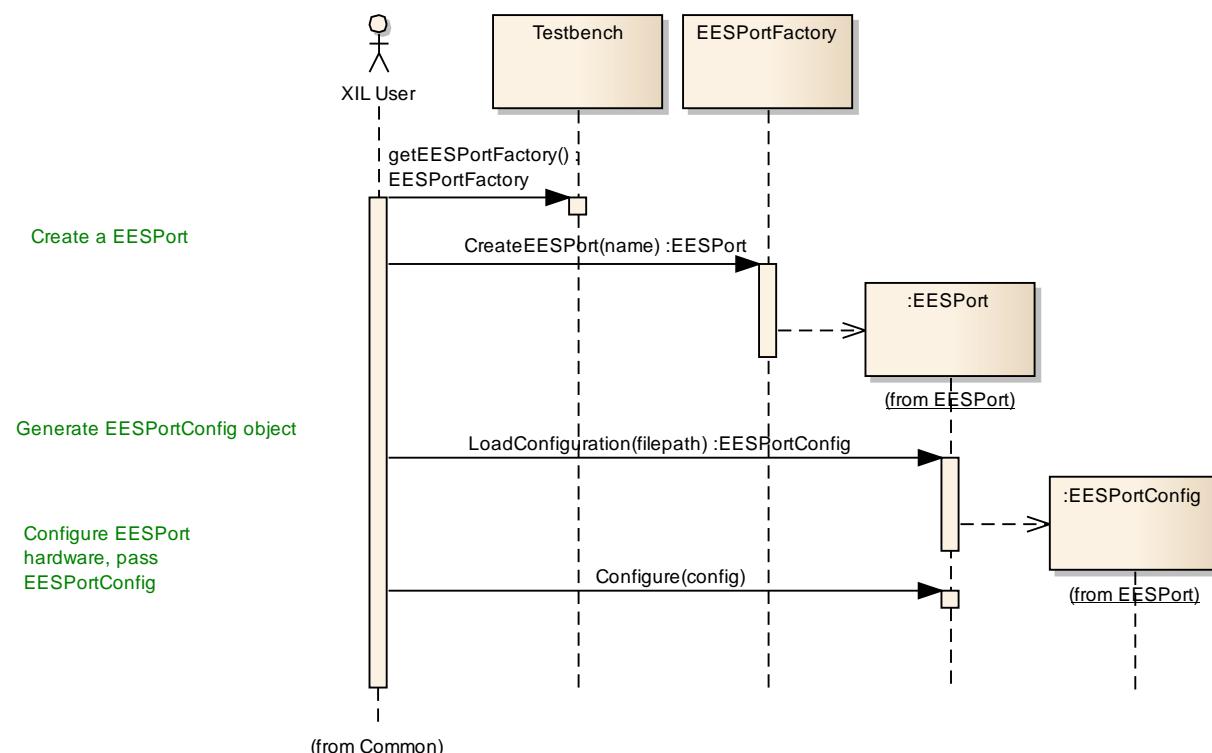


Figure 194 Process of EESPort creation and configuration

Configuration of the new port instance is a two-step process (see [Figure 194](#)). First a vendor specific configuration file has to be loaded via method `LoadConfiguration` that returns an `EESPortConfig` object.

The second step of configuration is calling the `Configure` method and passing the `EESPortConfig` object created in step one. This establishes a connection to the EES hardware and activates the passed configuration. On successful configuration the port's state is set to `eCONNECTED`.

In this state, the signal collection can be obtained from the respective `EESPort` object through `getSignalInfos` property. This property returns a collection of `EESSignalInfo` Objects. This property returns an Empty read-only List in state `eDISCONNECTED`.

EESSignalInfo

The `EESSignalInfo` class represents the Signal information. This class contains read-only properties to get the ECU Name, Pin Name, Signal Name, Vendor-Specific properties and allowed ErrorCategories of a signal. The property `AllowedErrorCategory` returns a collection (Dictionary) of ErrorCategories that can be simulated on this signal with `ErrorCategory` enum as key and List of `LoadType` enum as value. If specific `ErrorCategory` and `LoadType` combination is not found, then that `ErrorCategory` cannot be simulated on the signal. The property `AllowedErrorTypes` returns a collection (List) of `ErrorTypes` that can be simulated on this signal. If specific `ErrorTypes` is not found, then that `ErrorType` cannot be simulated on the signal.

5.6.2.2 CREATING ERROR CONFIGURATIONS

The example shows how an error configuration is created. First a new error configuration is created. This is independent from the EES port instance. In a second step, a specific error is created using the factory and builder objects of the configuration. In the third step, a new error set is requested from the error configuration and the error is assigned to this error set. The error configuration is assigned to the EES port instance and stored in a file.

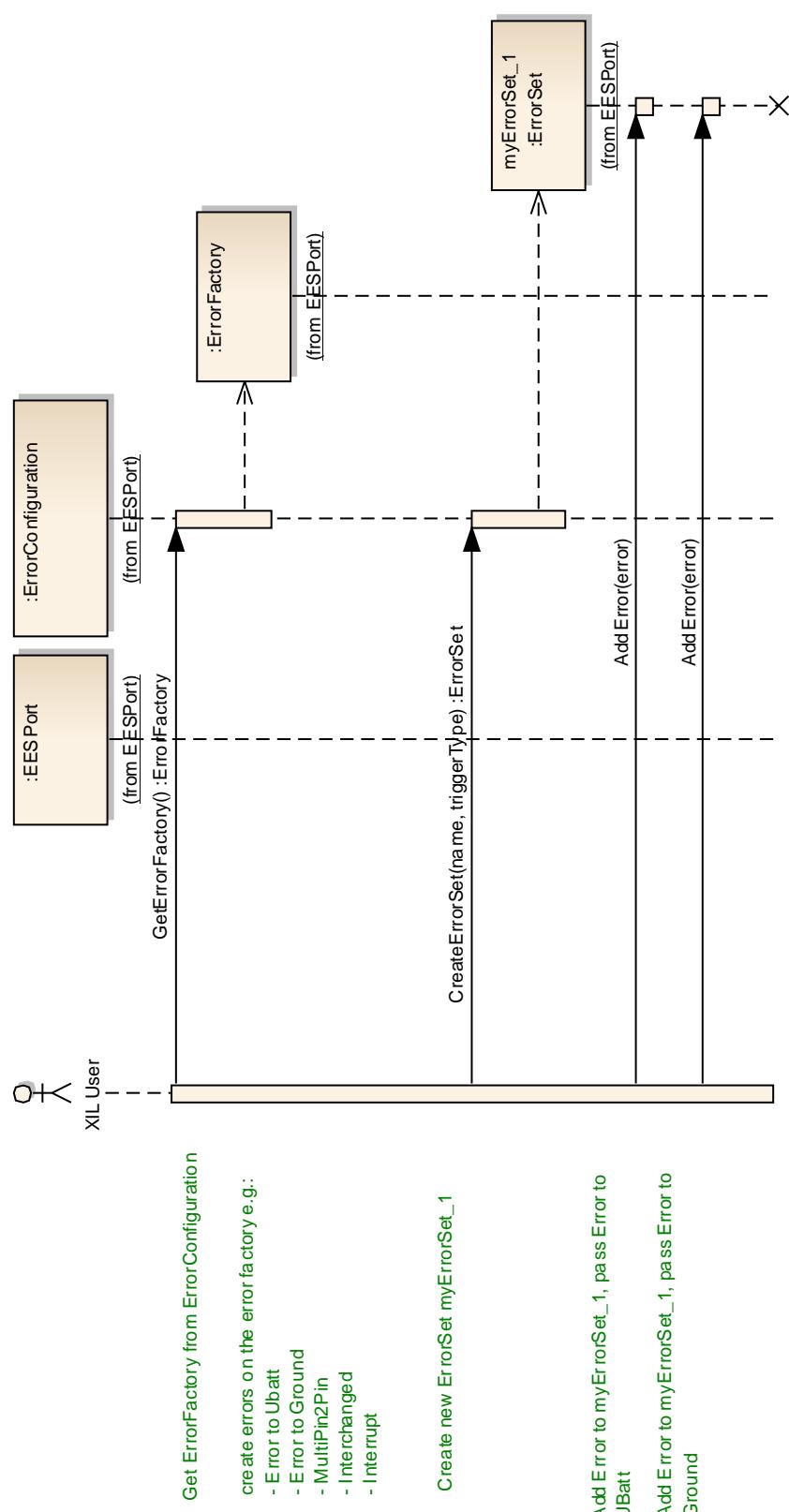


Figure 195: Error Configuration Part 1

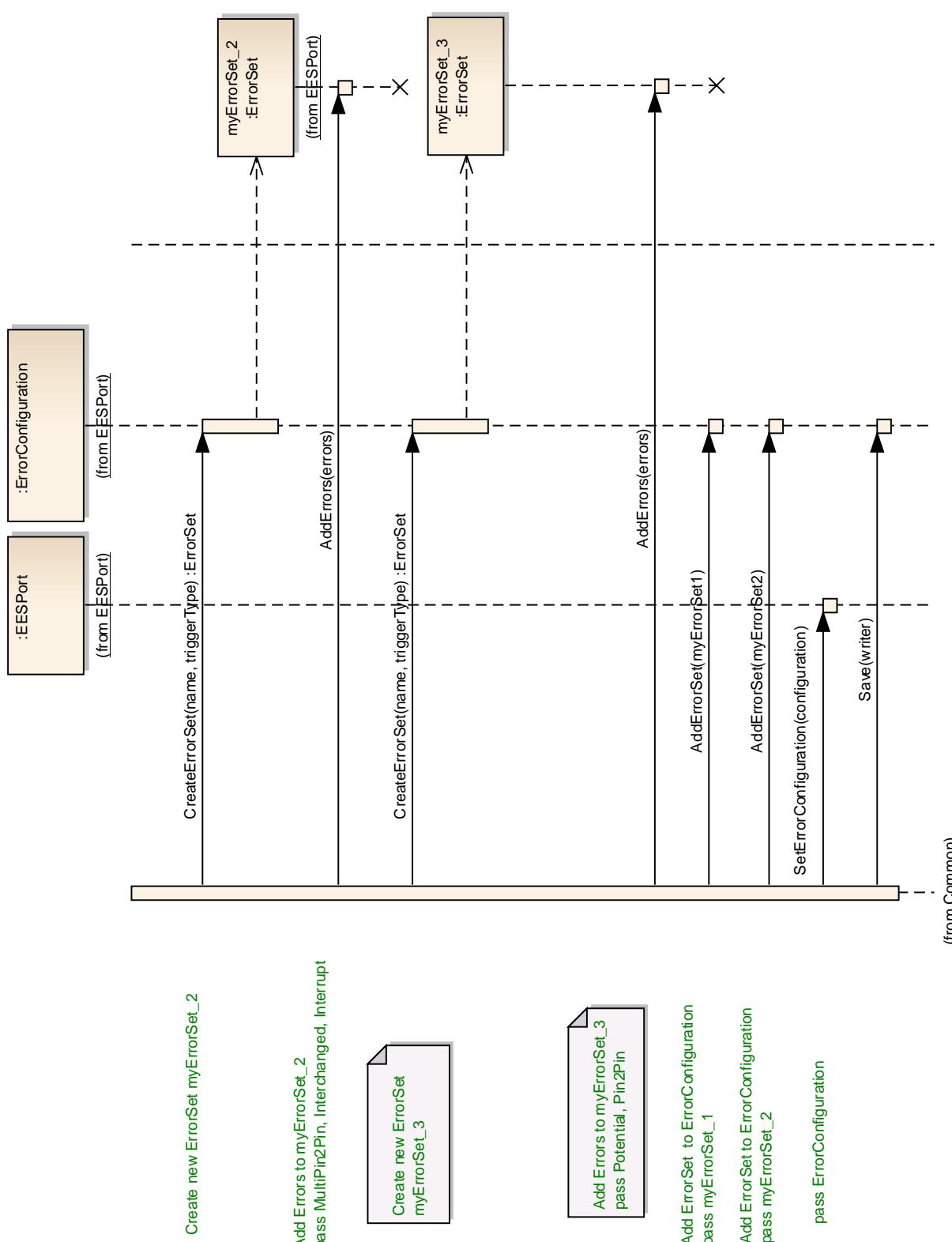


Figure 196: Error Configuration Part 2

5.6.2.3 ERROR STIMULATION

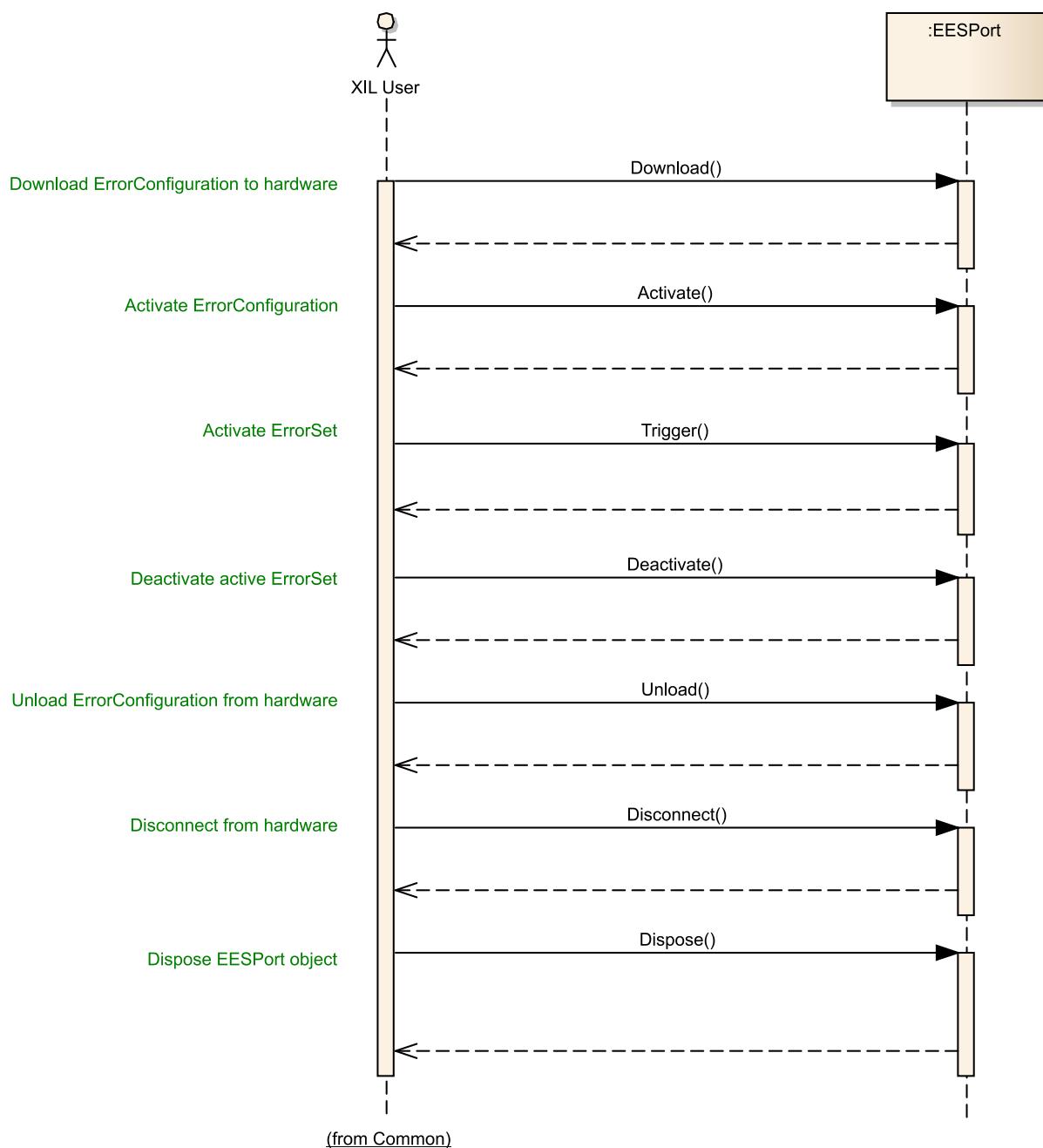


Figure 197: Error Stimulation

The created error configuration is executed in this sequence diagram. This is straight forward: the error configuration is assigned to the EES port, it is downloaded, and activated. Then a manual trigger is fired by the method Trigger() and the next trigger is awaited (a hardware trigger as defined in the second error set). At the end, the execution of the ErrorConfiguration is stopped.

5.6.2.4 EXTENSION OF ERROR STIMULATION

In contrast to the previous sequence diagram, the following figure shows that the created error configuration is extended while it is being executed.

The error configuration is assigned to the EES port, downloaded, activated and triggered. While an ErrorSet is active, a new errorset is created and added to the existing error configuration. Then the error set is downloaded using the method Update() and triggered. At the end, the execution of the ErrorConfiguration is stopped.

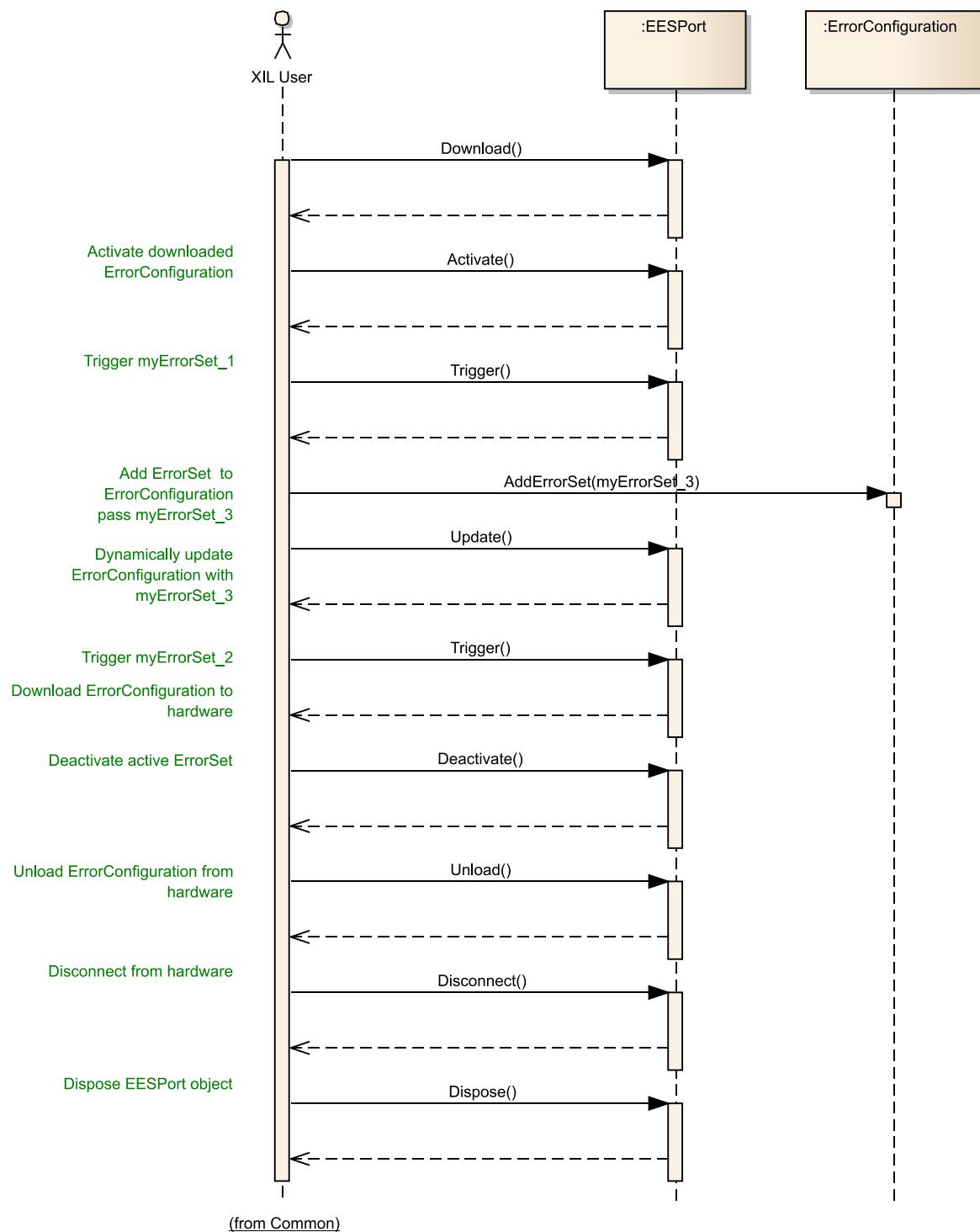


Figure 198: Extension of Error Stimulation

5.6.2.5 CREATING ERROR OBJECTS

The error objects stored in the error configuration are created using the error factory and error builder classes. The following sequence diagrams show how to create the different errors exemplary.

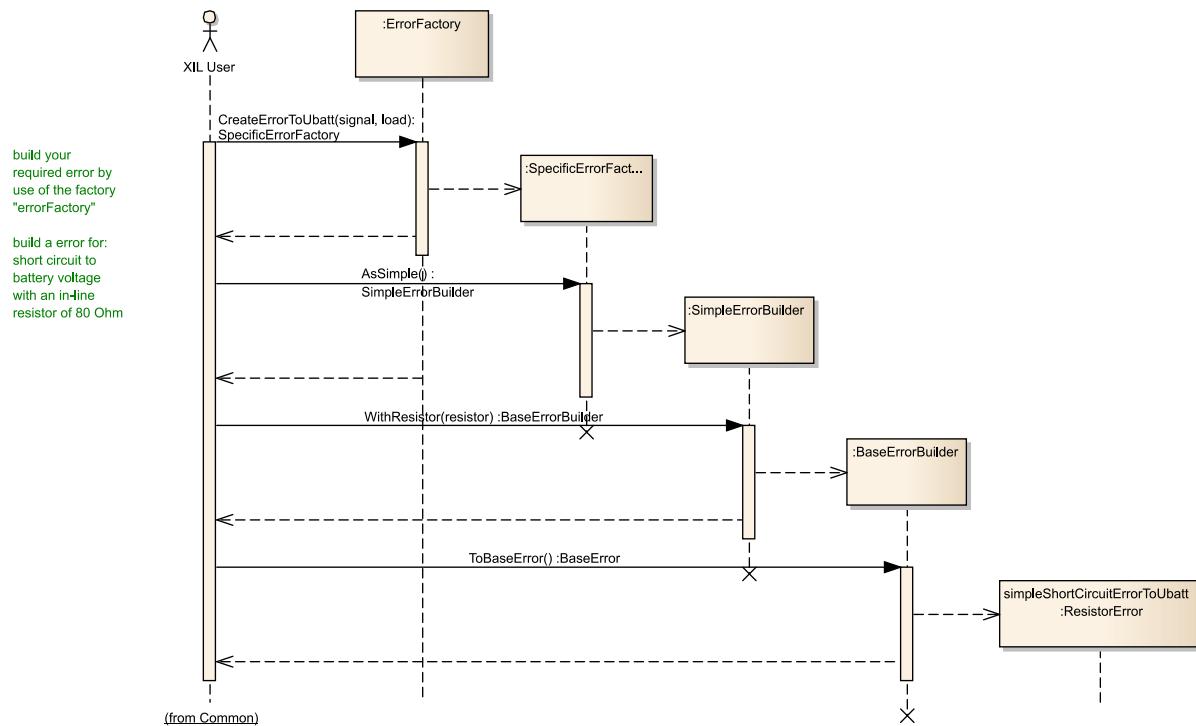


Figure 199: create a short-circuit to U_{battery} error object

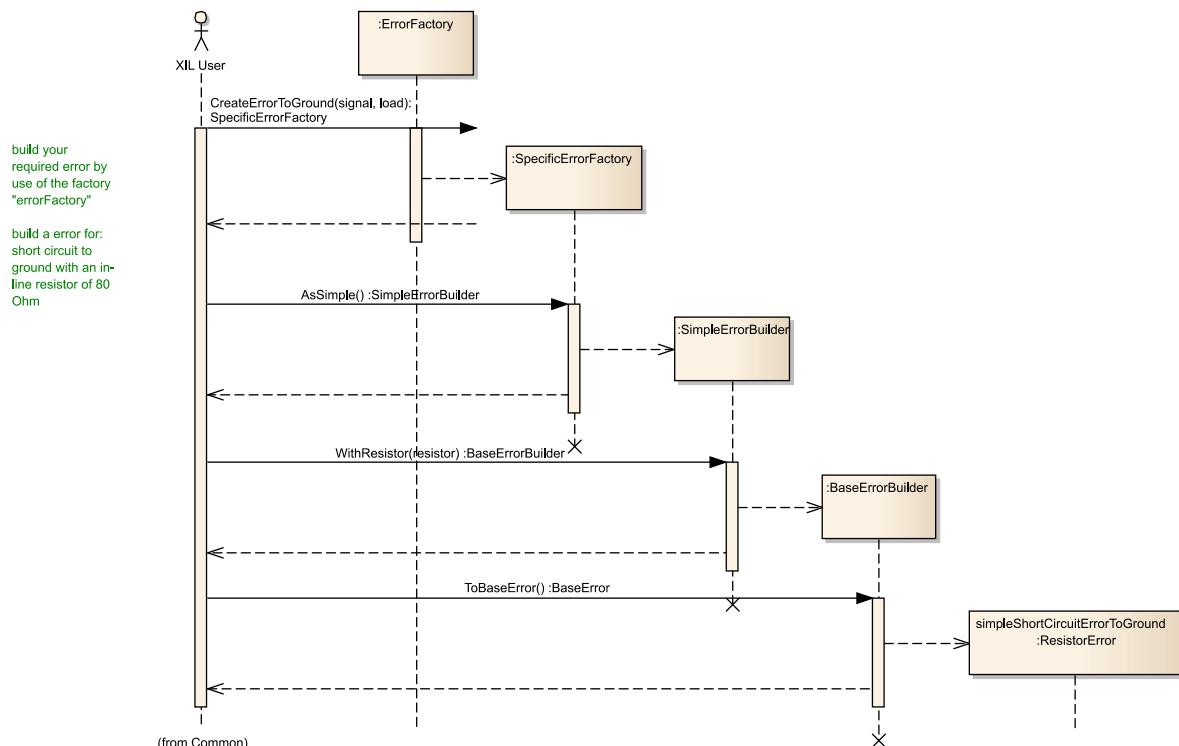


Figure 200: create a short-circuit Ground error object

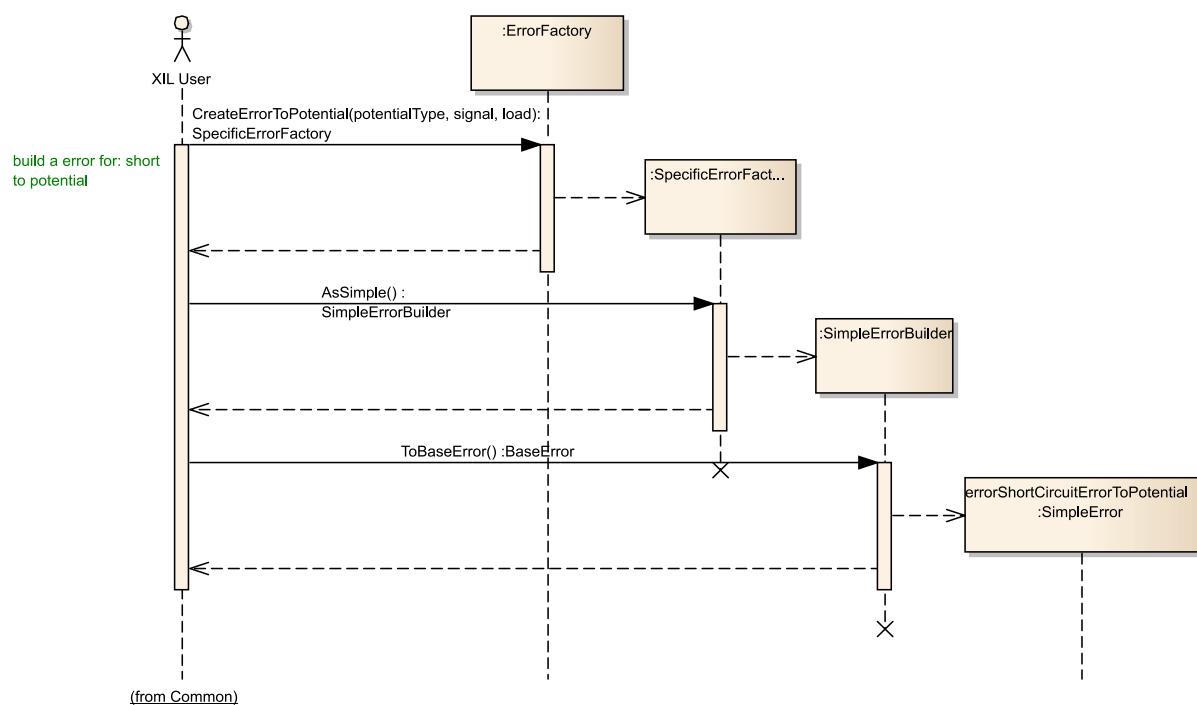


Figure 201: create a short-circuit potential error object

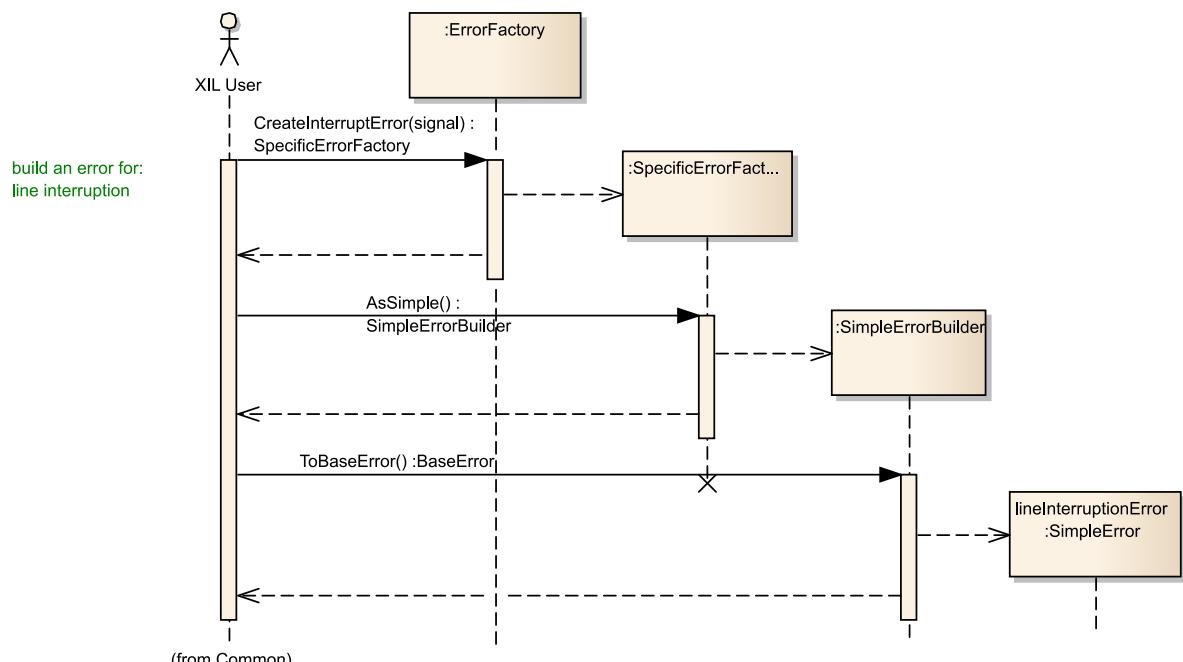


Figure 202: create a line interruption error object

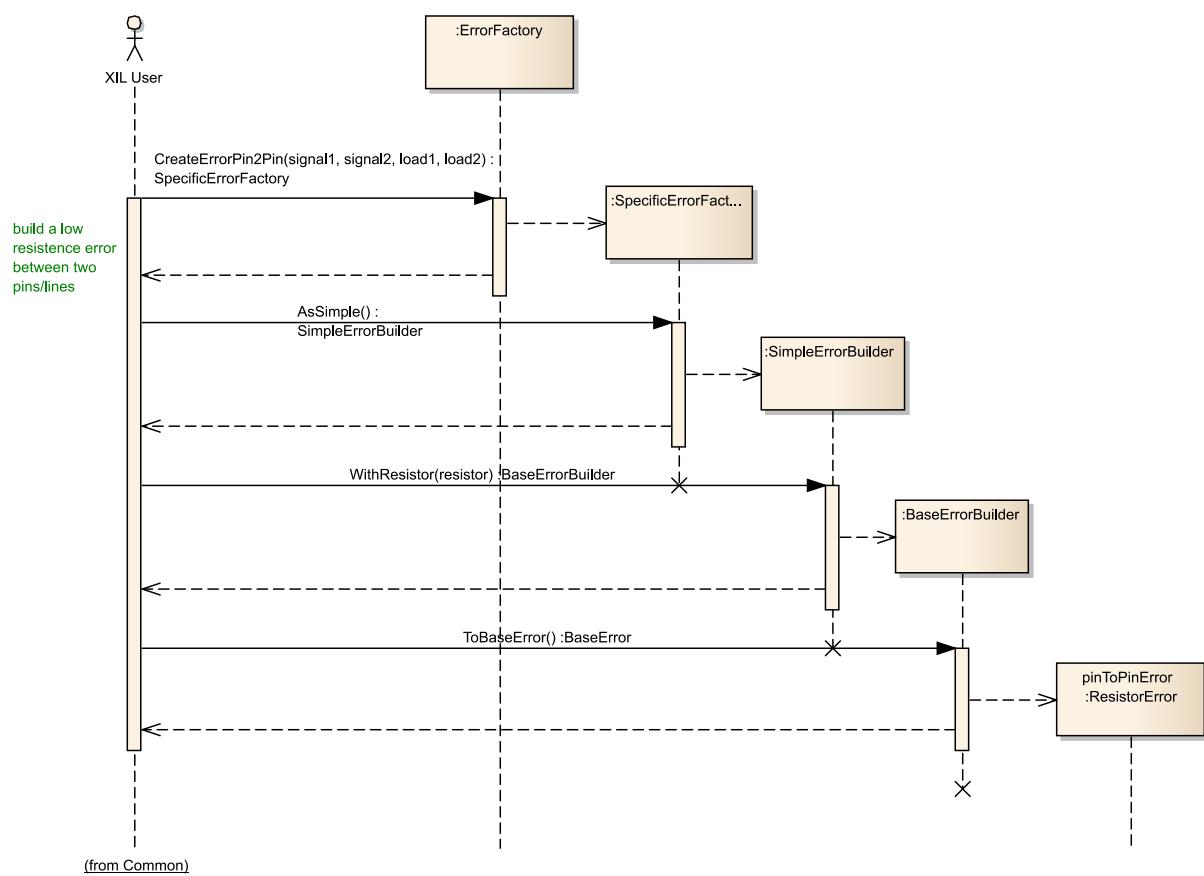


Figure 203: create a pin-to-pin error object

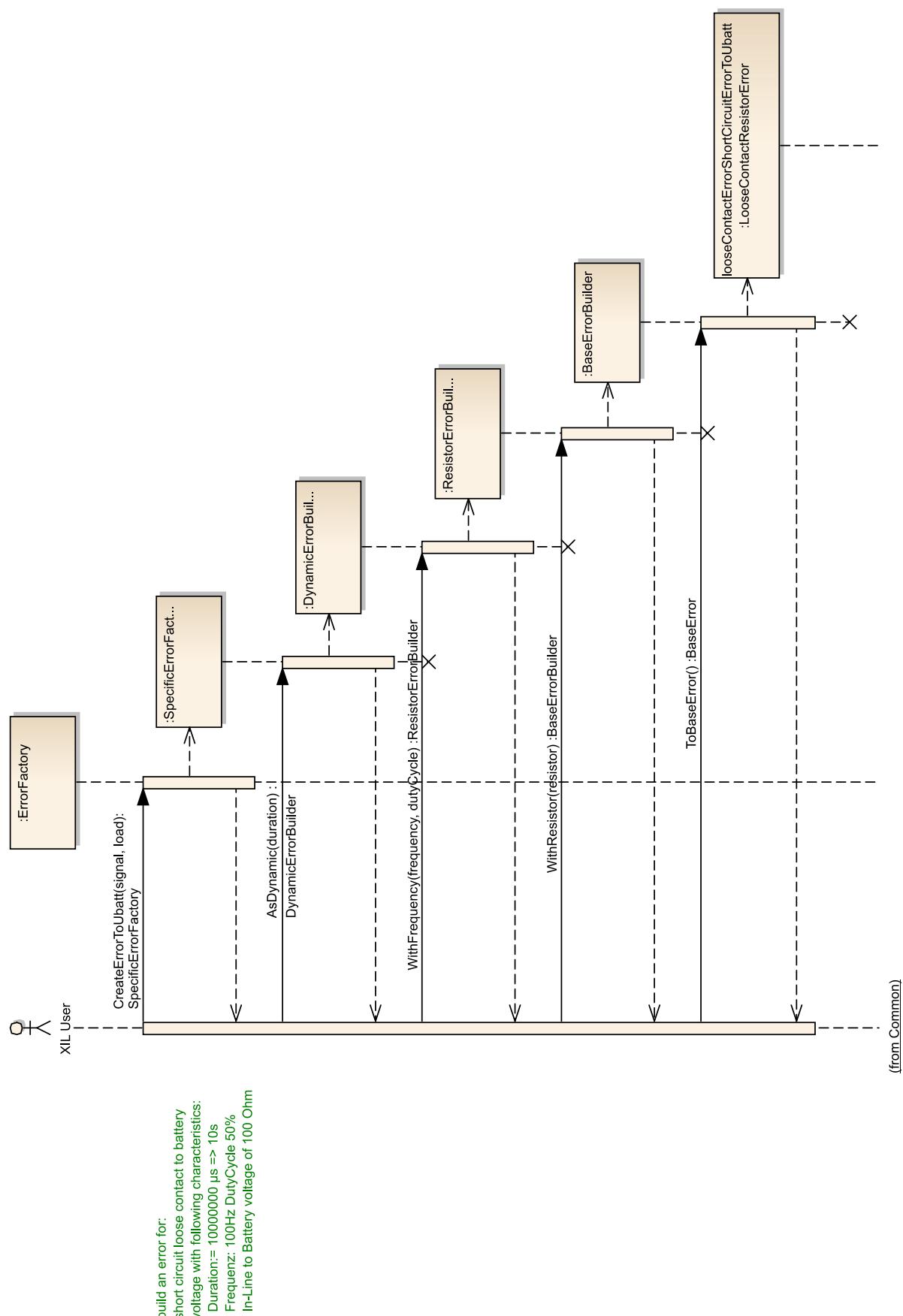


Figure 204: create a loose contact error object

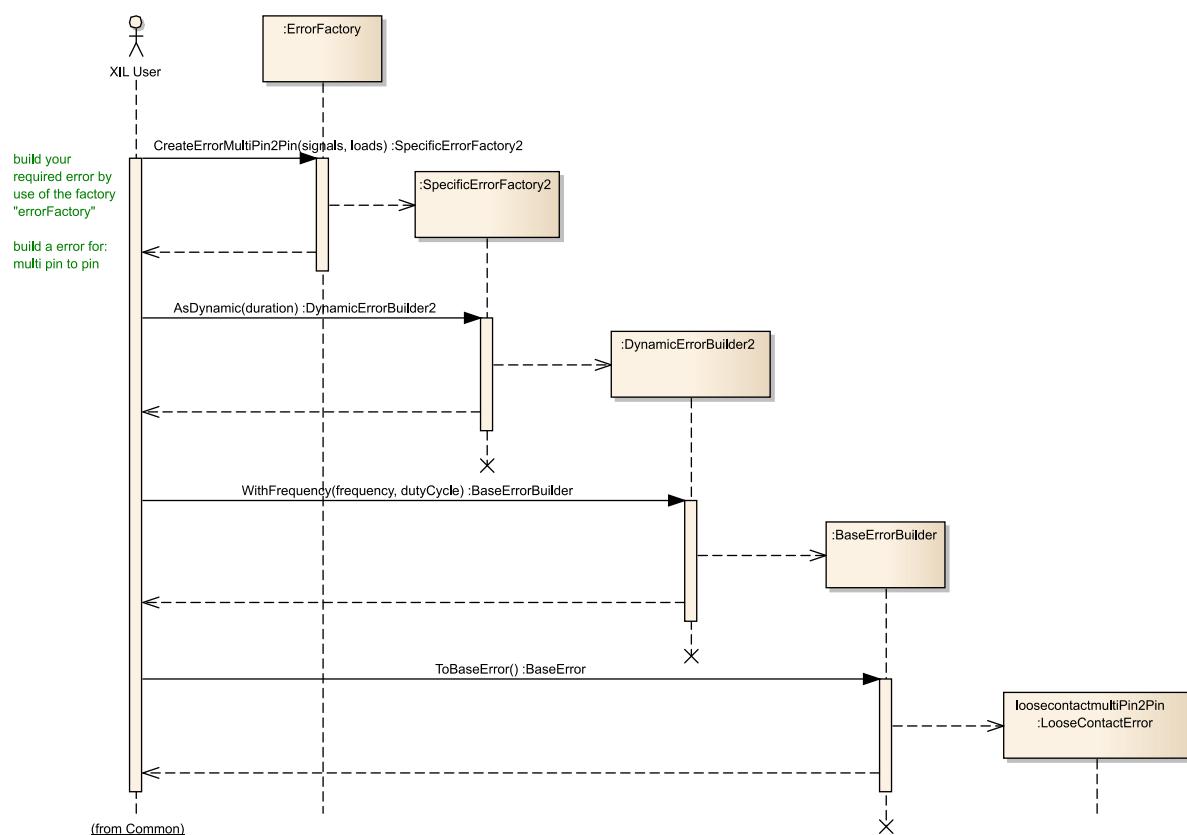


Figure 205: create a loose contact error object (MultiPin2Pin)

These error creation sequences are part of the creation of an error configuration as shown in Creating Error Configurations. For the not shown errors the creation sequence is similar.

5.6.2.6 LOADING ERROR CONFIGURATIONS FROM FILE

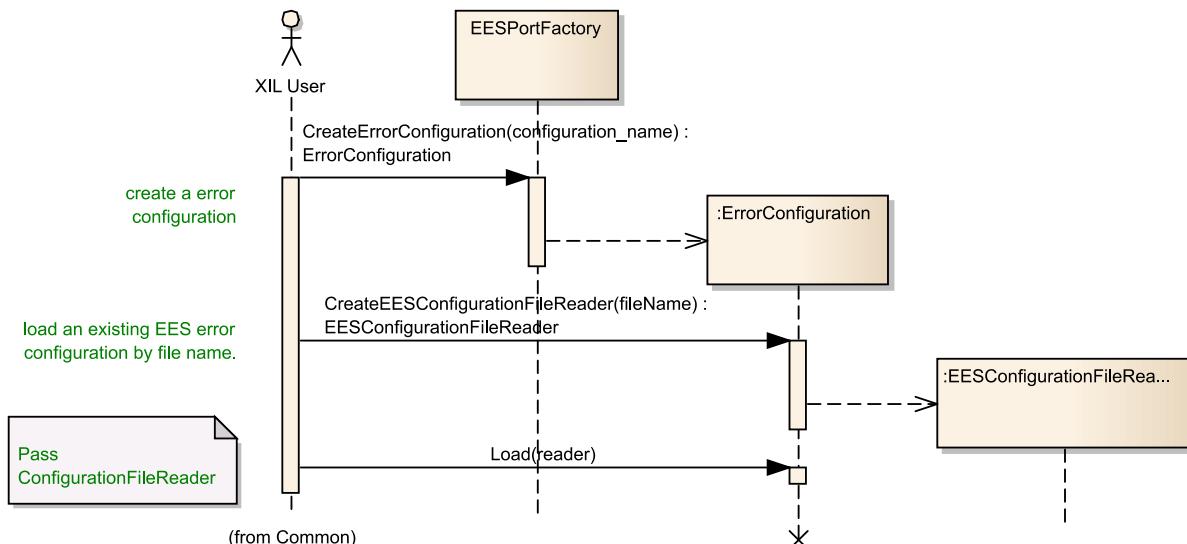


Figure 206: load error configuration from file

In this sequence the error configuration is loaded from a file. The sequence shows how this is done using the error configuration file reader object. Other parts of usage, especially assignment of the error configuration to the EES port, downloading, starting, and stopping

the configuration is independent from the creation. Therefore it is in principle the same as in the example above (see chapter [Creating Error Configurations](#)).

When an error configuration is loaded, all existing data will be overwritten with the data in the file. Already existing errors and error sets are removed from the configuration.

5.6.3 SPECIAL HINTS

5.6.3.1 EES HARDWARE LIMITATIONS AND EXTENSIONS

The EES port handles the EES hardware in an abstract manner. Therefore it is limited to a defined set of functionality. Possibly the hardware supports additional functions that are not supported by the XIL API EES port. These functions are not accessible with means of XIL API. Additional functionality may be used within a test case using additional APIs of the EES implementation. But strict compatibility to XIL API is lost for test cases that make use of such extended functionality, of course.

On the other hand, a specific EES hardware may lack some functionality that is defined by the XIL API. This is also a valid use case. The EES hardware respective port implementation is still XIL API compliant. In this case, the EES port returns an error when not supported features or parametrizations are used (see also section [5.6.1.6](#) for associated checks to be implemented by certain port methods). XIL API compatible test cases can be executed but return an error due to lack of functionality of the underneath hardware ECU Port.

5.7 NETWORK PORT

5.7.1 USER CONCEPT

5.7.1.1 GENERAL

The Network Port is designed to provide access to bus communication in a standardized way, allowing measurement and transmission of bus data. The port in general can handle different types of bus systems, such as CAN, LIN and FlexRay but this version solely provides functionality for CAN communication. Later compatible extensions for other bus systems will follow.

This port provides functionality for read- and write-access for CAN frames, to set up captures of CAN frames and the replay of CAN frames. The Network Port is designed mainly to operate in parallel to a HIL system, ECU or any other system (see [Figure 207](#)).

When using this port, it is assumed that a communication matrix (such as a DBC or FIBEX file) and vendor specific configuration files are prepared in advance. The whole Network Port object model will be created automatically based on this information. Individual Network Nodes (e. g. ECUs) are not part of the object model since the port was designed to support use cases which do not require specific information about the individual network node topology.

The Network Port -package is related to the packages "Common:Capturing" and "Common:CaptureResult". These sub-packages are not part of Network Port as they are also used in other ports like ECUPort and MAPort.

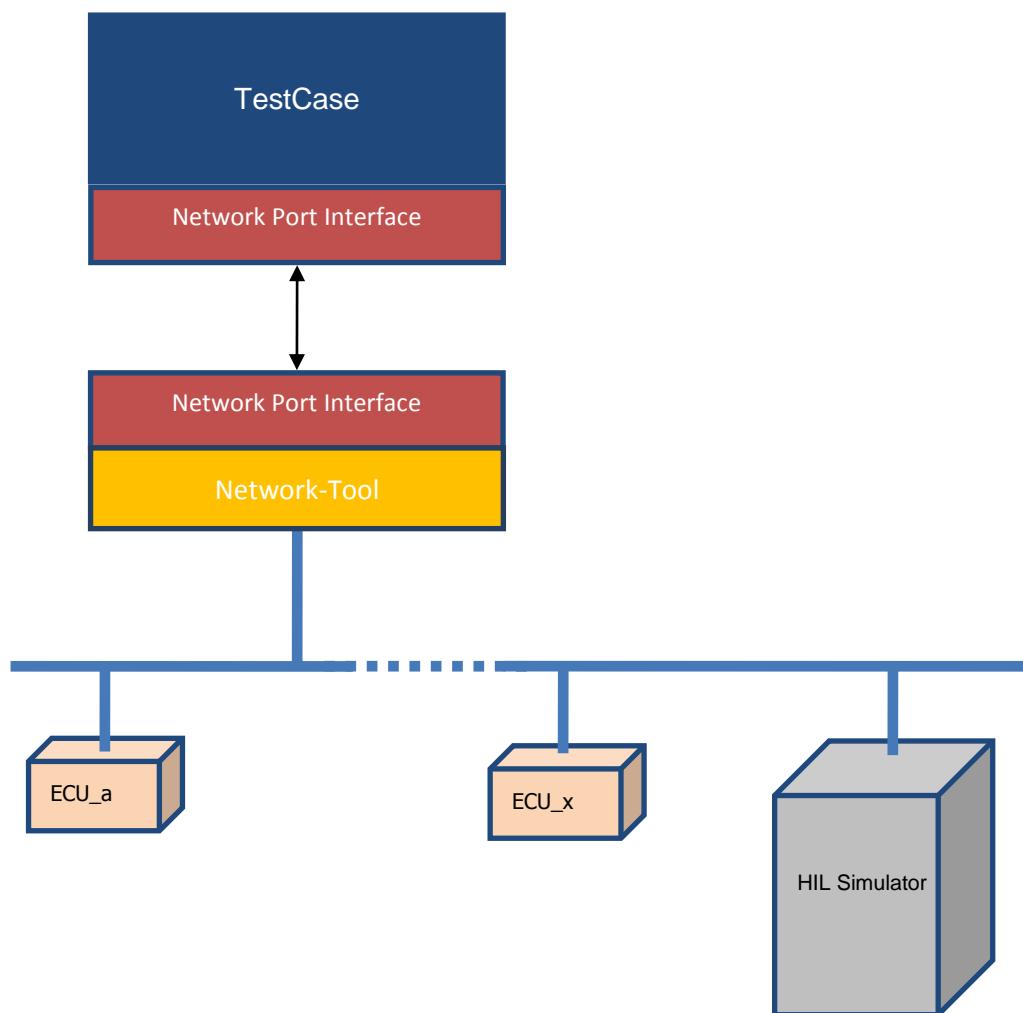


Figure 207: Network Port setup

5.7.1.2 NETWORK PORT

The Network Port can be configured by using predefined configuration files or just a channel configuration. Below a channel all objects can be created manually starting with the creation of new frames.

The Network Port organizes its object classes in a tree-like hierarchy. This hierarchy is based on the Cluster-, Channel-, Frame-, PDU-, Signal- objects.

The Network Port offers methods from reading a single frame from a bus channel up to reading a single signal from the bus. Furthermore the port provides methods for sending frames which can be created manually. This frame can be sent in different ways:

- Cyclic (continuous sending of a frame after starting till stop),
- Repetition x times (frame is sent only for a given number of frames after start),
- sequence sending; sending of different frame contents (Frame is sent with different content sequentially)

The Network Port provides also the functionality to capture frames and even signals from a frame (see `CreateFrameCapture()` and `CreateSignalCapture()` in [Figure 208](#)). For this purpose the Network Port uses the comprehensive capturing functionality of the XIL standard, see chapter [Capture Configuration and Control](#) and [Capture Result](#).

To replay captured frames, the Network Port provides functionalities in the class ReplayCan. The reply of frames can be done on the level of the channel hierarchy.

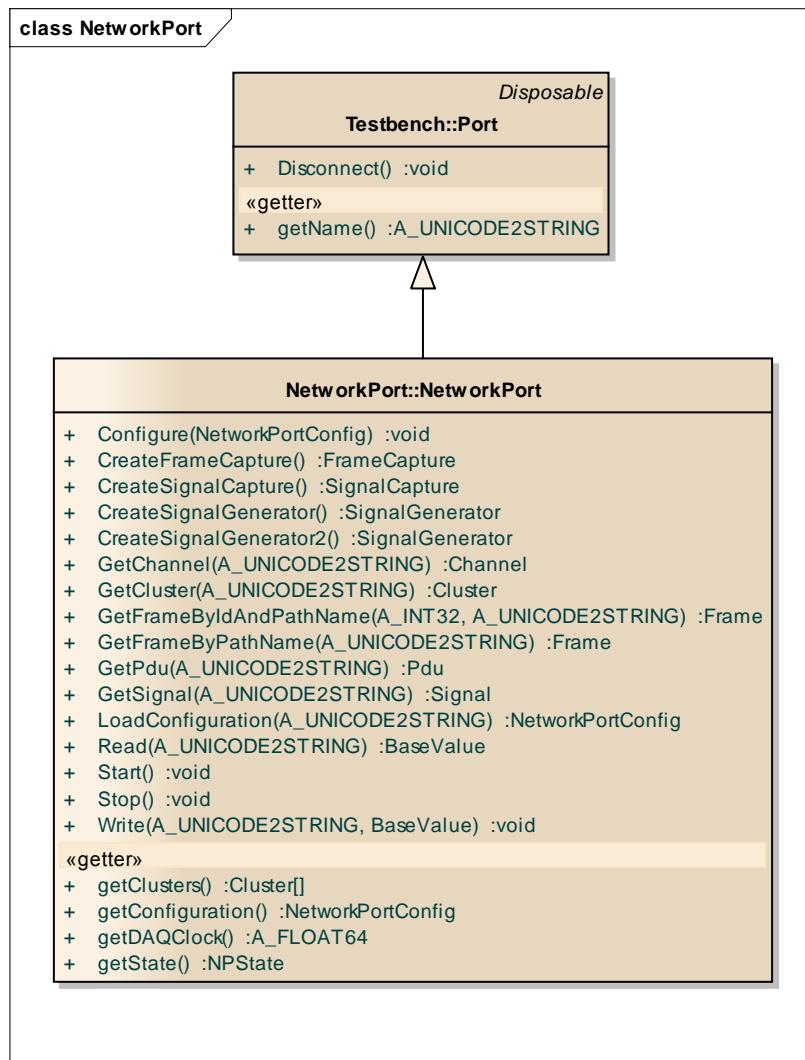


Figure 208: Network Port interface

5.7.1.3 STATES OF THE NETWORK PORT

The Network Port has three states which are called eDISCONNECTED, eSTOPPED and eRUNNING.

In the initial DISCONNECTED state the port is not connected to a bus system and not configured. By calling the `Configure()` method, which is common for all Testbench Ports the STOPPED state can be entered. In this state the port is configured but not connected to a bus system.

Calling the `Start()` method of the Network Port leads to the RUNNING state where the port is configured and connected to a bus system. Receiving and transmitting bus frames is possible now. All Sender and Receiver instances of the Network Port which are activated (Sender default: false, Receiver default: true) are now have been started automatically. By calling the `Stop()` method all Sender and Receiver instances are stopped automatically and the port falls back to the STOPPED state.

The initial DISCONNECTED state can be entered from both STOPPED and RUNNING by calling the Disconnect() method.

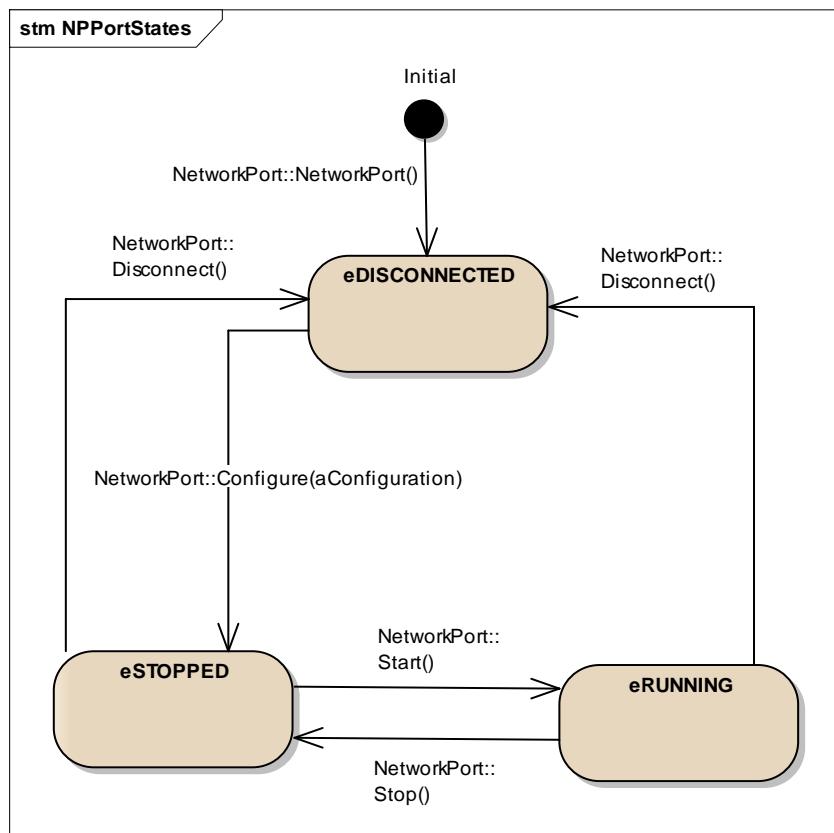


Figure 209: States of the Network Port

State transitions only take place, if all preconditions are fulfilled and no error occur during execution of the state changing method. Otherwise the state is not changed, i.e. the state remains the same as before the method was called. Methods, which trigger a state change, will throw an exception if the state change could not be processed successfully.

Methods, which require a certain state, will throw an exception if the current state does not match the required state.

Table 57 Network Port states

	eDISCONNECTED	eSTOPPED	eRUNNING
Method LoadConfiguration	X		
Method Configure	X		
Method Disconnect		X	X
Method Start		X	
Method Stop			X
Method Read			X
Method Write			X
All other methods and properties	X	X	X

5.7.2 USAGE OF THE NETWORK PORT

5.7.2.1 PORT CREATION AND CONFIGURATION

Instances of `NetworkPort` are created by calls to method `CreateNetworkPort` of a `NetworkPortFactory` object. It returns a `NetworkPort` instance with the name given as parameter. The `NetworkPortFactory` object can be obtained from the `Testbench` object that provides a corresponding property.

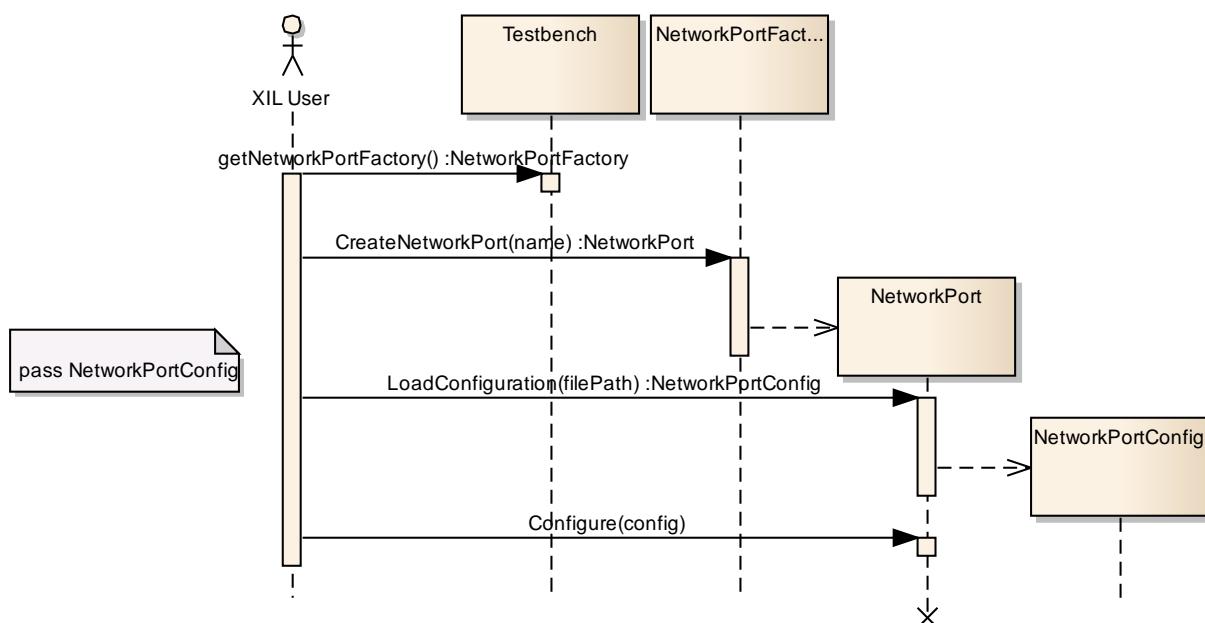


Figure 210 Process of NetworkPort creation and configuration

Configuration of the new port instance is a two-step process (see [Figure 210](#)). First a vendor specific configuration file has to be loaded via method `LoadConfiguration` that returns a `NetworkPortConfig` object.

The second step of configuration is calling the `Configure` method and passing the `NetworkPortConfig` object created in step one. This establishes a connection to the bus communication hardware and activates the passed configuration. On successful configuration the port's state is set to `eSTOPPED`.

5.7.2.2 OBJECT MODEL

The Network Port is organized as hierarchical object model. The basic elements are Cluster, Channel, Frame, Pdu (Protocol Data Unit) and Signal. Individual Network Nodes (e. g. ECUs) are not part of the object model since the port was designed to support use cases which do not require specific information about the individual network node topology.

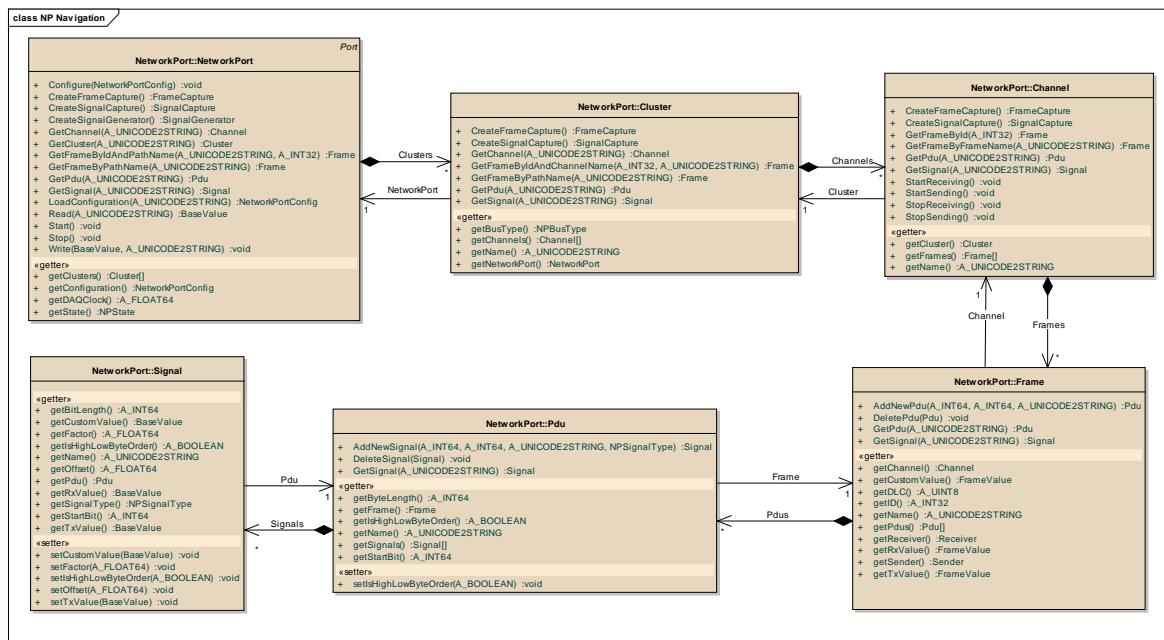


Figure 211: Class diagram of the hierarchical Network Port structure

Network Port Cluster

A Network Port Cluster represents all channels which are linked in an arbitrary topology (bus, star, ring) by one communication medium such as CAN, LIN, FlexRay, etc. A Network Port Cluster contains an array of Channels.

Network Port Channel

A Network Port Channel entity represents a single physical access to the communication medium of a cluster. A Network Port Channel contains an array of Frames.

Network Port Frame and FrameValue

A Network Port `Frame` contains the definition of the Frame such as the data length code (DLC) of the payload, the Name, the Identifier and the contained Pdus as defined in the communication matrix. Since individual network nodes are not modeled a `Frame` does not need to have a direction property (Receive, Transmit).

A Network Port `FrameValue` contains a snapshot of a Frame payload (`Byte[]`) plus a timestamp and the Frame Identifier. This allows to handle many FrameValues (e. g. created during Frame-Capturing) referring to a single Frame definition.

Network Port Pdu

A Network Port `Pdu` (Protocol Data Unit) consists of one or more Signals and is assigned via Startbit and Length to partitions of the Frame payload (`FrameValue::DataBytes`). For CAN always only one Pdu is used which covers the complete Frame payload.

Network Port Signal

A Network Port `Signal` contains the definition of the Signal such as the Startbit, BitLength, Byteorder, Factor and Offset. The Signals numerical physical value is represented as a `BaseValue`.

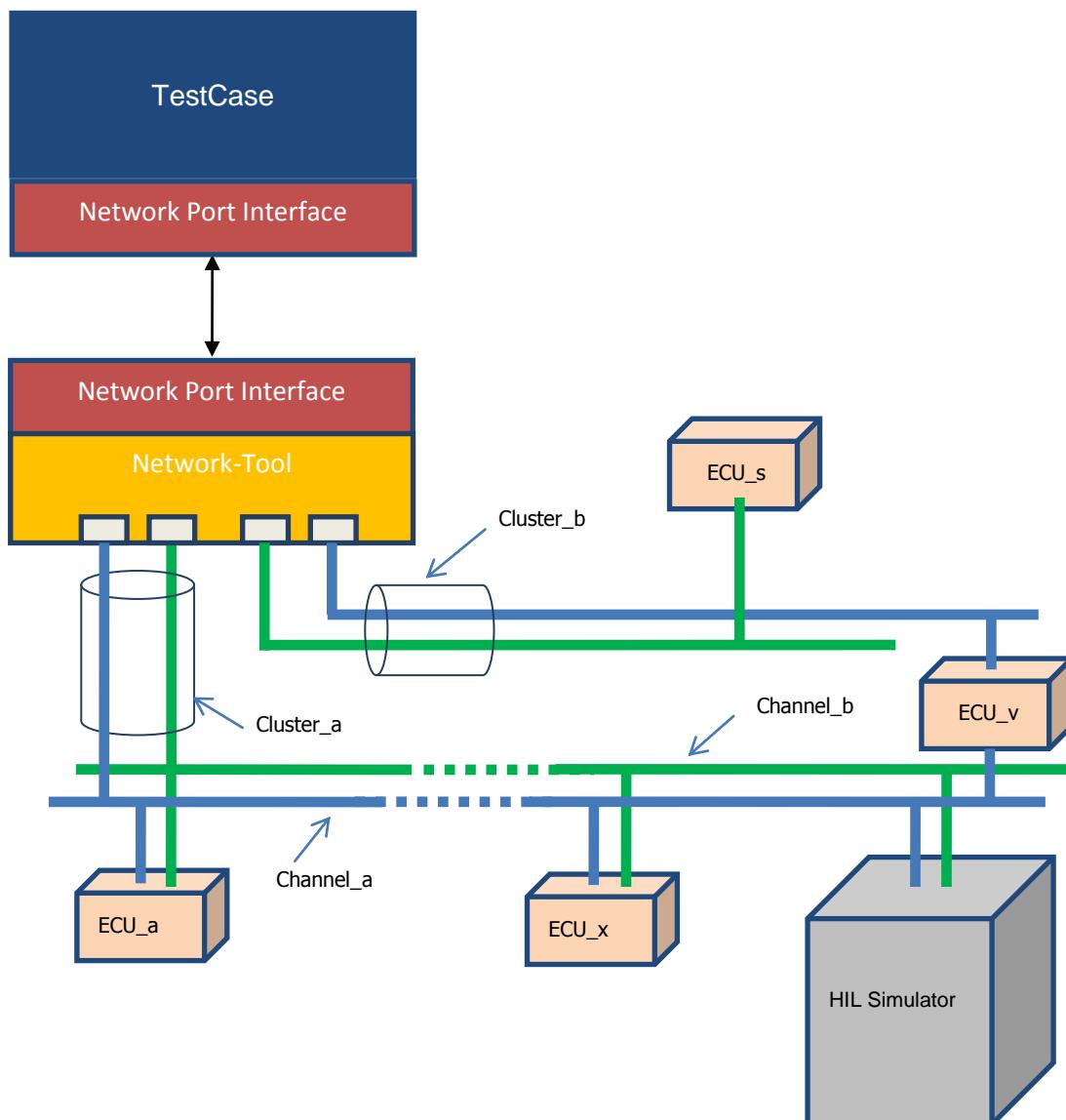


Figure 212: Network Port configuration

5.7.2.3 MAPPING TO DBC AND FIBEX

The description of CAN networks is often stored in CANdb databases (DBC files). This storage format is a de facto standard for CAN and a lot of tools can read DBC files. One CANdb database describes the communication of a single CAN network. It contains the signals and messages transmitted on the bus and the network nodes connected to the bus. The description of FlexRay networks can be stored by using the FIBEX (ASAM MCD-2 NET) format. The Network Port was designed to be compatible to FIBEX and CANdb. The following table shows the mapping of CANdb, FIBEX 3.1 and Network Port object types:

Table 58 Mapping between Network Port, CANdb and FIBEX

Network Port	CANdb	Fibex 3.1
Cluster	Network	CLUSTER
Channel	Network	CHANNEL

N.A. (no network node objects)	Network Node	ECU, CONTROLLER, CONNECTOR
Frame	Message	FRAME, FRAME-Triggering
Pdu	N.A. (no PDU objects)	PDU, PDU-INSTANCE
Signal	Message Signals	SIGNAL, SIGNAL-INSTANCE

5.7.2.4 NAVIGATING THE OBJECT MODEL

Each object is named and can be accessed by name or by a path (fully qualified name) for hierarchical access. Inside the Cluster, Channel, Frame, Pdu and Signal arrays each object name is unique.

For top-down navigation (e. g. from the Network Port down to a Frame) the methods GetCluster(...), GetChannel(...), etc. are available. A path is constructed by using “::” as delimiter for each level of hierarchy.

Examples:

```
myClusterA = myNetworkPort.GetCluster("ClusterA");
myFrame4Path = "ClusterA::Channel1::Frame4";
myFrame4 = myNetworkPort.GetFrame(myFrame4Path);
```

For bottom-up navigation (e. g. from a signal up to the Network Port) each object has a reference to his parent object.

Example:

Get a reference to the Network Port instance starting from a reference to a signal:

```
myNetworkPort = mySignal.Pdu.Frame.Channel.Cluster.NetworkPort;
```

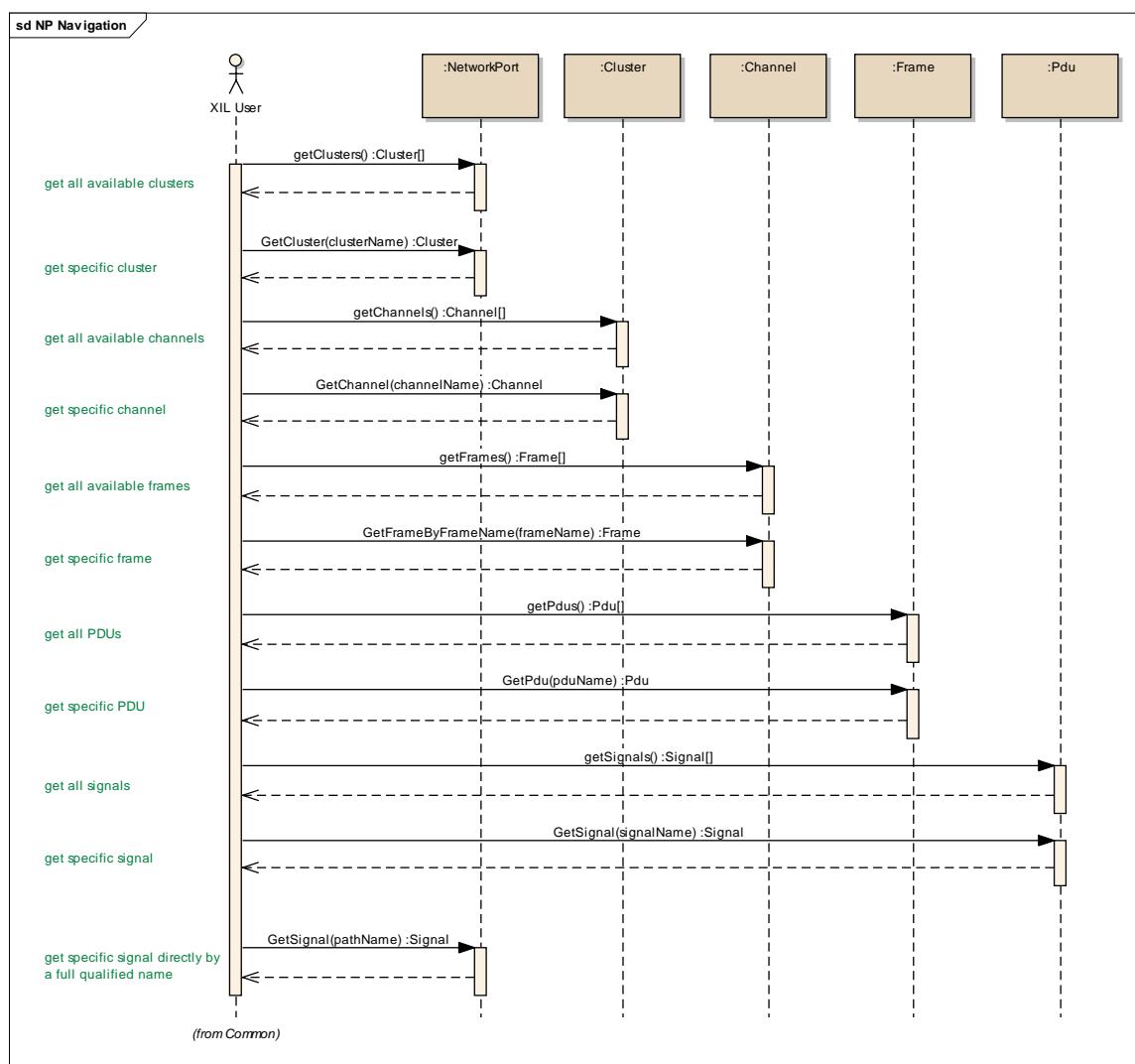


Figure 213: Sequence diagram for Network Port top-down navigation

5.7.2.5 SEND AND RECEIVE FRAMES

The NetworkPort Frame owns three predefined buffers (TxValue, RxValue, CustomValue) of type FrameValue which are used by the frame's Sender and Receiver objects. A FrameValue is derived from BaseValue and serves as a container to store the raw bytes as used by the send and receive operations together with a timestamp and frame identification data.

Each signal of a frame contains properties which are named TxValue, RxValue and CustomValue accordingly. These properties represent the numerical (BaseValue) physical values of the signal. The port implementation is supposed to keep each FrameValue of the frame synchronized to the according BaseValue of each signal.

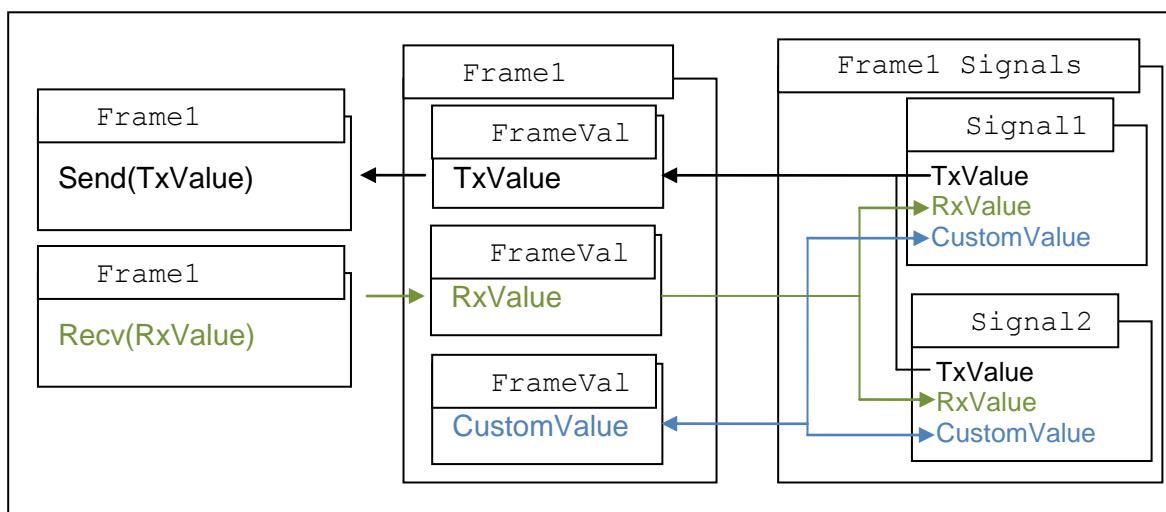


Figure 214: Frame and Signal buffer synchronization

Each time the property `Signal::TxValue` is modified by the user the `Frame::TxValue` is updated automatically by the Network Port implementation. By this a permanent synchronization between all Signal TX values and transmitted frame payload is ensured.

After receiving new `Frame::RxValue` data each `Signal::RxValue` is updated automatically by the Network Port implementation. By this a permanent synchronization between all Signal RX values and received frame payload is ensured.

The properties `Frame::CustomValue` and `Signal::CustomValue` can be set by the user and are synchronized in both directions. By this Signal value can be determined from a captured `FrameValue` or frame values can be constructed out of signal values. The Network Port implementation internal performs the necessary bit masking and byte swapping operations to convert between frame payload and signal values.

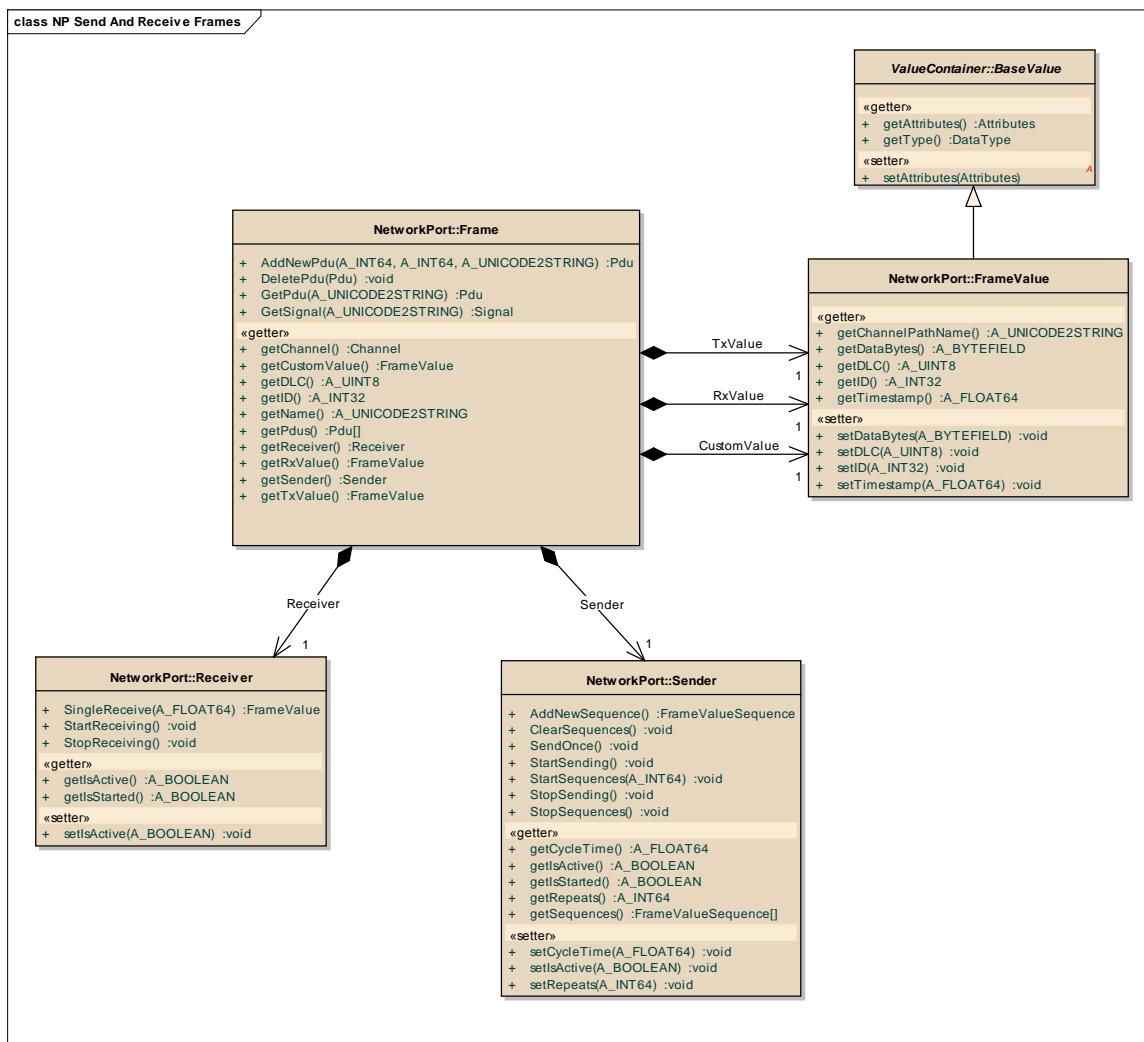


Figure 215: Class diagram for Frame, FrameValue, Sender and Receiver

If a Sender is activated and started (or a `SendOnce()` call is made) it transmits the payload as provided by `Frame::TxValue`.

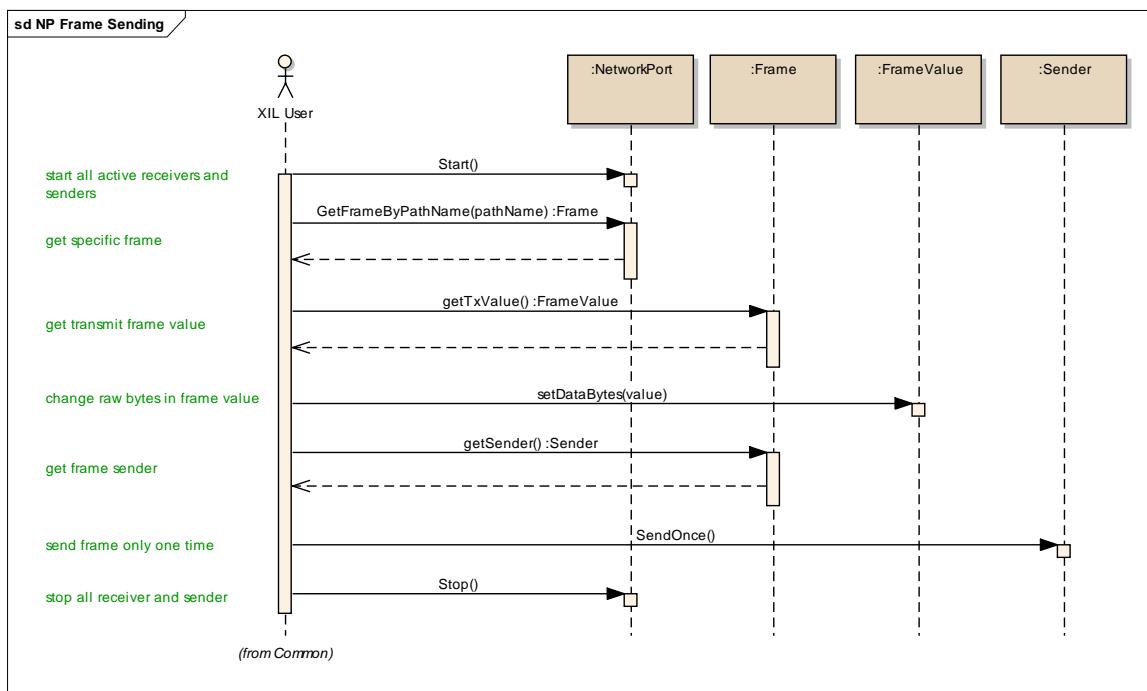


Figure 216: Sequence diagram for frame data transmission

If a Receiver is activated and started (or a SingleReceive() call is made) it updates the Frame::RxValue with the payload received on the bus.

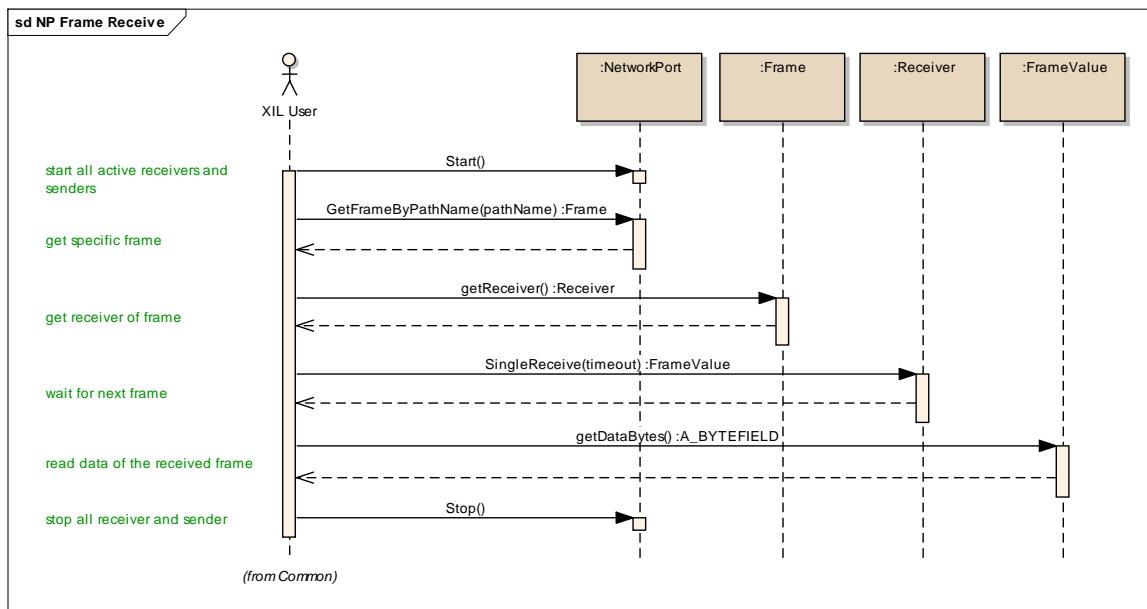


Figure 217: Sequence diagram for frame data reception

5.7.2.6 SEND AND RECEIVE SIGNALS

Sending and receiving of signals basically works identically to sending and receiving at frame level. The only difference is the usage of signal values (BaseValue) instead of frame values (FrameValue). Signal values are physical values (converted values) which are internally

converted to raw values (coded values) by using the linear conversion provided by the Factor and Offset properties of the Signal.

Each time the property `Signal::TxValue` is modified by the user the `Frame::TxValue` is updated automatically by the Network Port implementation. By this a permanent synchronization between all Signal TX values and transmitted frame payload is ensured.

The sequence diagram below additionally shows how a Sender is activated, started and configured for cyclic transmission:

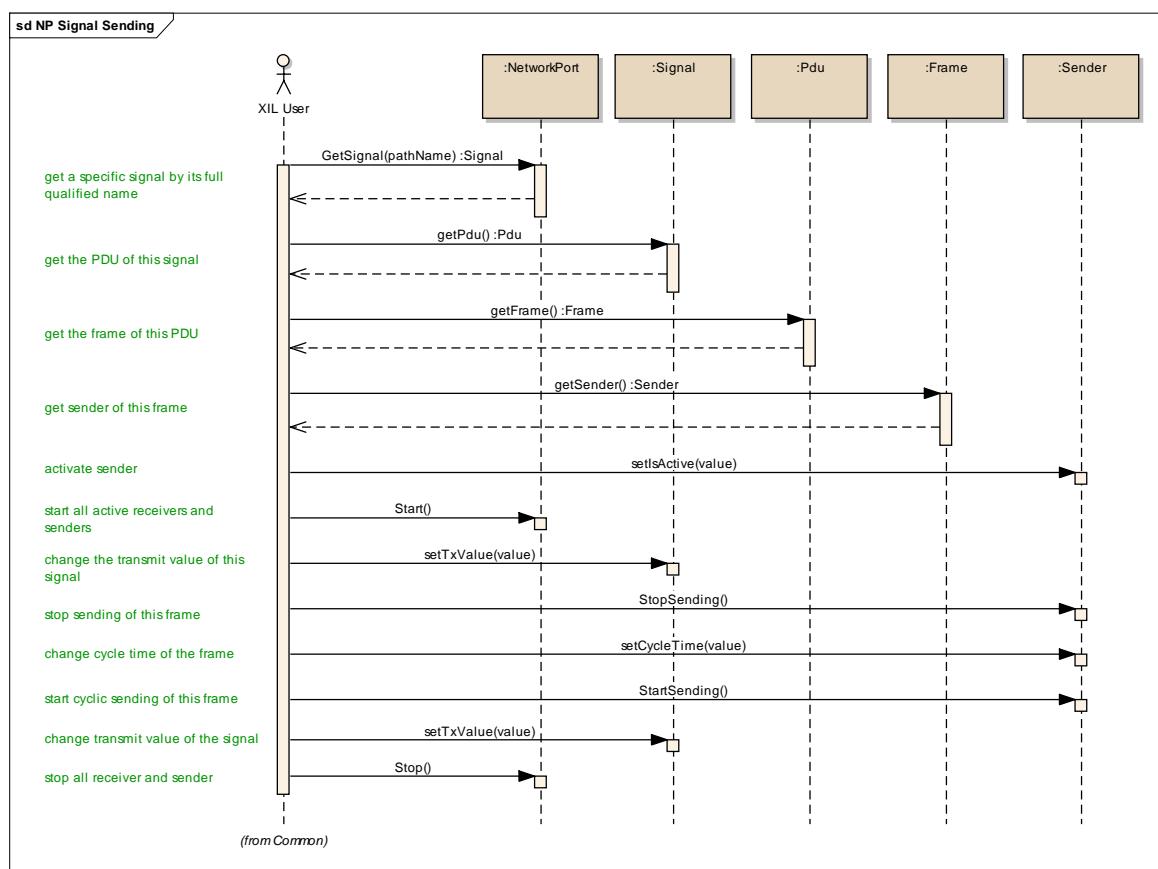


Figure 218: Sequence diagram for signal data transmission

The sequence diagram below shows the minimal number of method calls needed to receive a signal value from the bus. Since all `Receiver` objects are activated by default, they will start to receive bus data as soon as the `Start()` call at port level was successful. After receiving new `Frame::RxValue` data each `Signal::RxValue` is updated automatically by the Network Port implementation. By this a permanent synchronization between all Signal RX values and received frame payload is ensured. So every call of `Signal::getRxValue()` will return a value taken from the latest `Frame::RxValue` received from the bus.

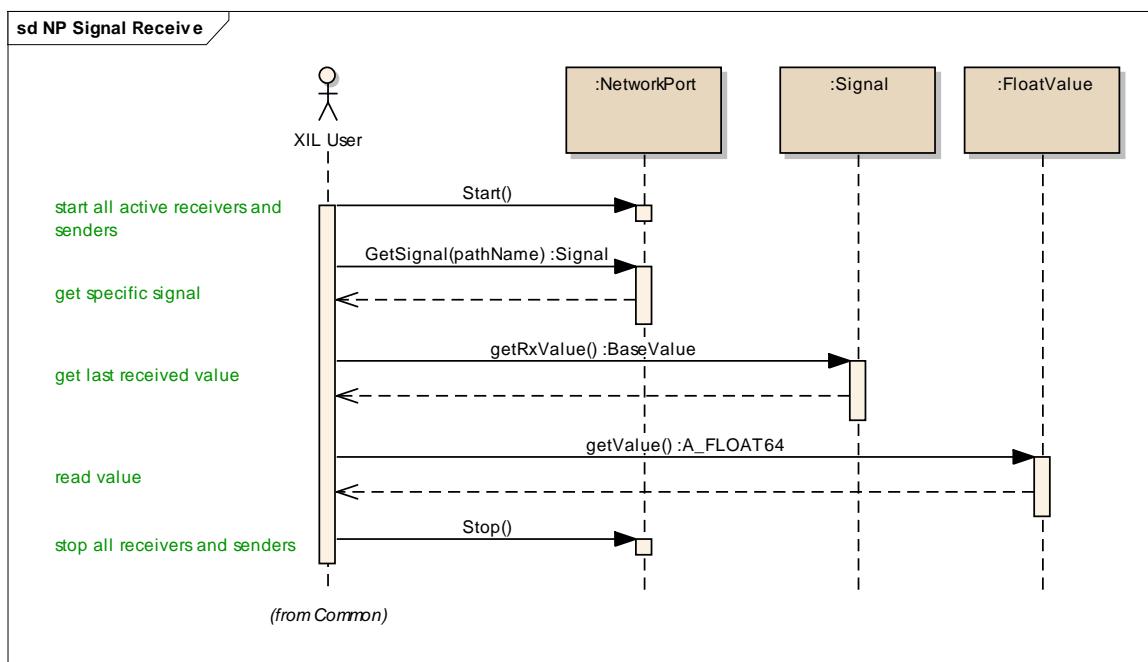


Figure 219: Sequence diagram for signal data reception

5.7.2.7 CAPTURING OF BUS SIGNALS

Network Port signals have a close relation to variables of the MA Port and the common capturing interfaces for variables are used in the Network Port as well. For this the Network Port object model defines an object called `SignalCapture`. It is derived from the common interface `Capture` and adds a `SetSignals(Signal[])` method which allows to configure a `SignalCapture` by providing an array of Signals. To create a `SignalCapture` instance the Network Port Channel object provides the according factory method `CreateSignalCapture()`.

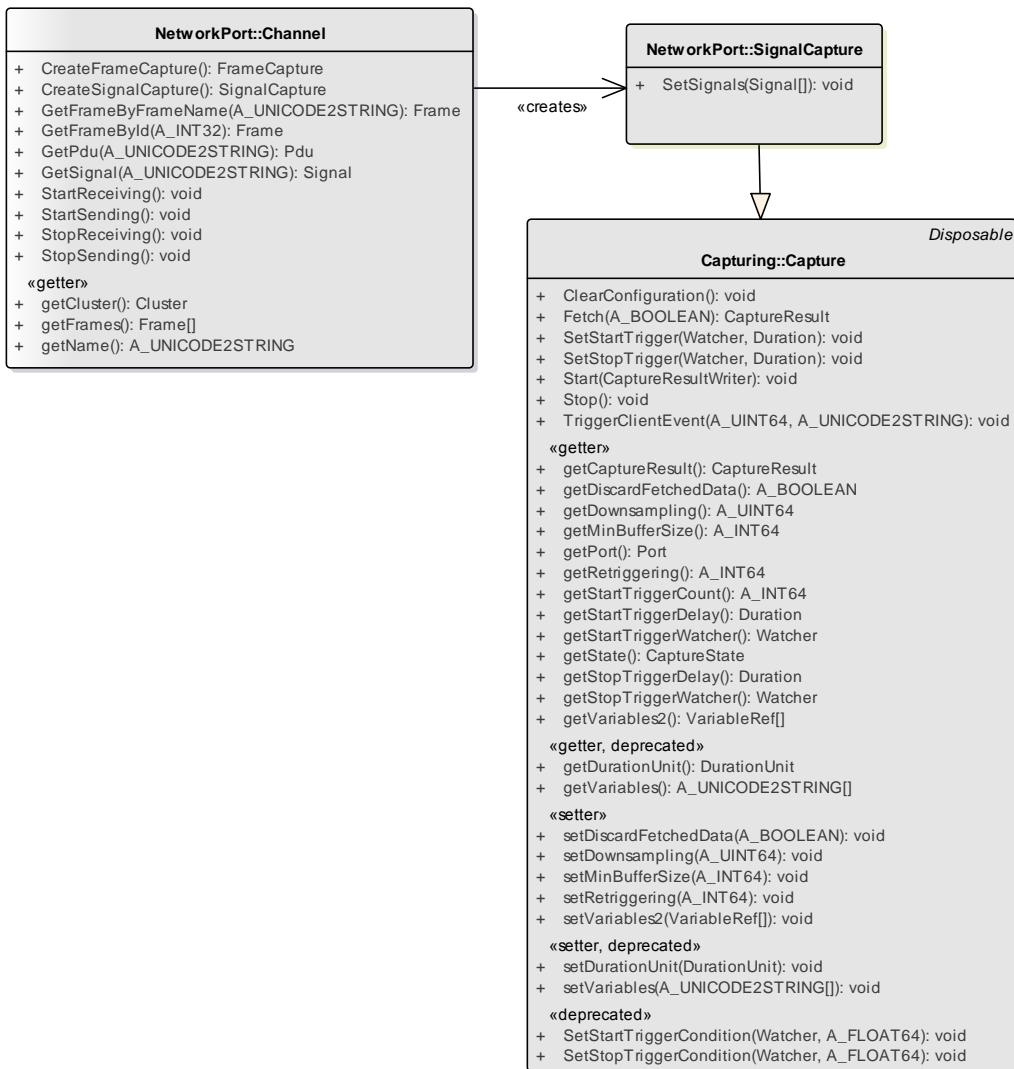


Figure 220: Class diagram for Network Port signal capturing

The recorded signals are accessible through the `CaptureResult` class. The `CaptureResult` can be analyzed directly or can be written into a file for a later use. For writing it into a file a `CaptureResultWriter` (e. g. `CaptureResultMDFWriter`) can be used.

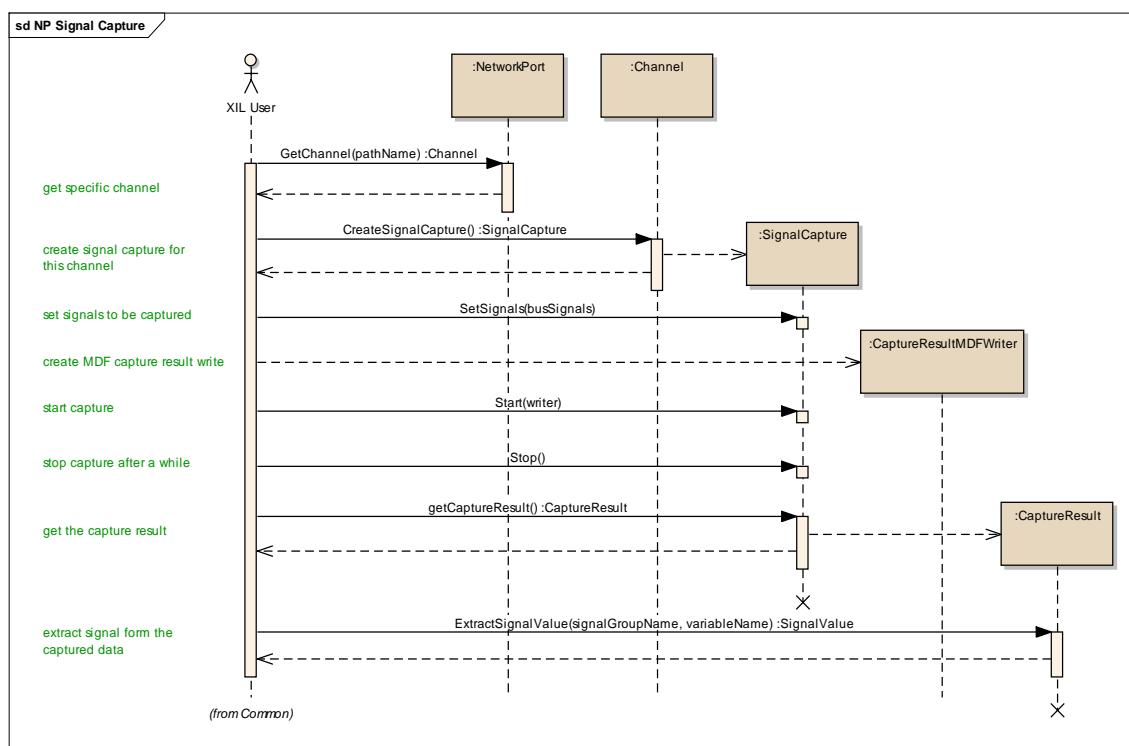


Figure 221: Sequence diagram for Network Port signal capturing

5.7.2.8 CAPTURING OF BUS FRAMES

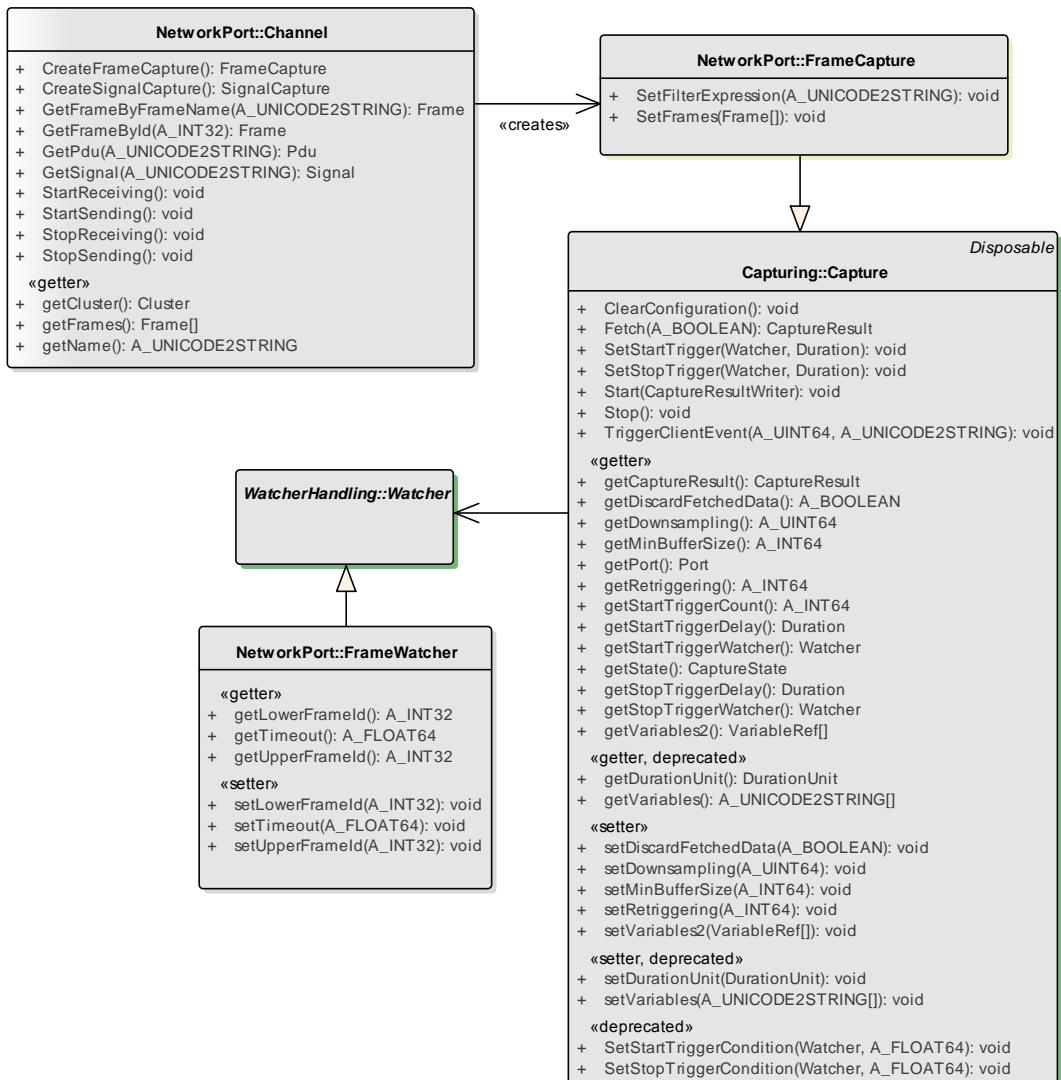


Figure 222: Frame Capture

With the bus frame capture it is possible to record the whole content of bus frames. Capturing is possible on three different levels (Port, Cluster, Channel) in the Network Port hierarchy. At Network Port level frames from all clusters are recorded whereas at channel level only frames from one channel are recorded.

The frame capture will be configured with the `FrameCapture` class. This class can be created with the method `CreateFrameCapture` on a `NetworPort`, `Cluster` or `Channel` class instance. The `FrameCapture` class is derived from the `Capture` class. Therefore a frame capture can be used similar to a signal or a data capture with all its further functionality such as triggering.

Frame Capture

In addition to the normal capture the `FrameCapture` class has two further methods to specific which frames should be recorded. The method `SetFrames` sets the list of frames which should be captured. With the method `SetFilterExpression` a string based filter expression can be specified for the frame filtering.

Note: It is intended to use `SetFrames` or `SetFilterExpression` but not both methods to avoid conflicting configurations.

Filter Expression

A filter expression contains of different in- or excluded terms for the frames which should be captured or not. An include term of a single frame looks like this “[FrameId]”. With expression “[LowerFrameId, UpperFrameId]” a range of frames from LowerFrameId to UpperFrameId can be included. The exclude terms have a similar syntax. Expression “[FrameId]” excludes a single frame Id from capturing. To exclude a range of frames following expression can be used “[LowerFrameId, UpperFrameId]”. Different terms can be combined with “;” to a complex filter expression. But it is only possible to use include or exclude terms in an expression not a combination of both otherwise an exception will be thrown. If no filter expression or frames are set then all frames will be captured. This is also the default setting of the `FrameCapture` class. The frame Id will be interpreted as decimal value or as hex value if the Id starts with “0x”.

Following examples should demonstrate the usage of filter expressions.

Example 1:

Expression: “[0x100];+[0x110, 0x120]”

Captured Frames: 0x100, 0x110-0x120

Example 2:

Expression: “[0x200];-[256]”

Captured Frames: All frames except frame 0x200 and 0x100

Example 3:

Expression: “[0x200];+[0x210, 0x220]”

Captured Frames: 0x200, 0x210-0x220

Frame Watcher

The NetworkPort introduces a new watcher class the `FrameWatcher` (see [Figure 222](#)). With this new class it is possible to start or stop the frame capture when a dedicated frame is received. The `FrameWatcher` can be used in the `SetStartTriggerCondition` method to specify which frames could start the frame capture. In the `SetStopTriggerCondition` method the `FrameWatcher` sets the frames which stop the capture.

To set the frames for the trigger condition the `FrameWatcher` has two properties. With `LowerFrameId` the lower Id of the frame range can be set. The property `UpperFrameId` sets the upper Id of the frame range. If lower frame Id and upper frame Id have the same value then only one frame with the specified Id can activate the trigger.

Capture Result

The recorded frames are accessible through the `CaptureResult` class. To access the data of the captured frames method `ExtractSignalValue` is available. This method returns an instance of a `SignalValue` class. The `XVector` property of the `SignalValue` returns a `FloatVectorValue` with the timestamps of the captured frames. The `FncValues` property returns a `FrameVectorValue` with the content of the recorded frames. The `CaptureResult` can be analyzed directly or can be written into a file for a later use. For writing it into a file a `CaptureResultWriter` (e.g. `CaptureResultMDFWriter`) can be used.

Example

Figure 223 demonstrates the usage of a frame capture.

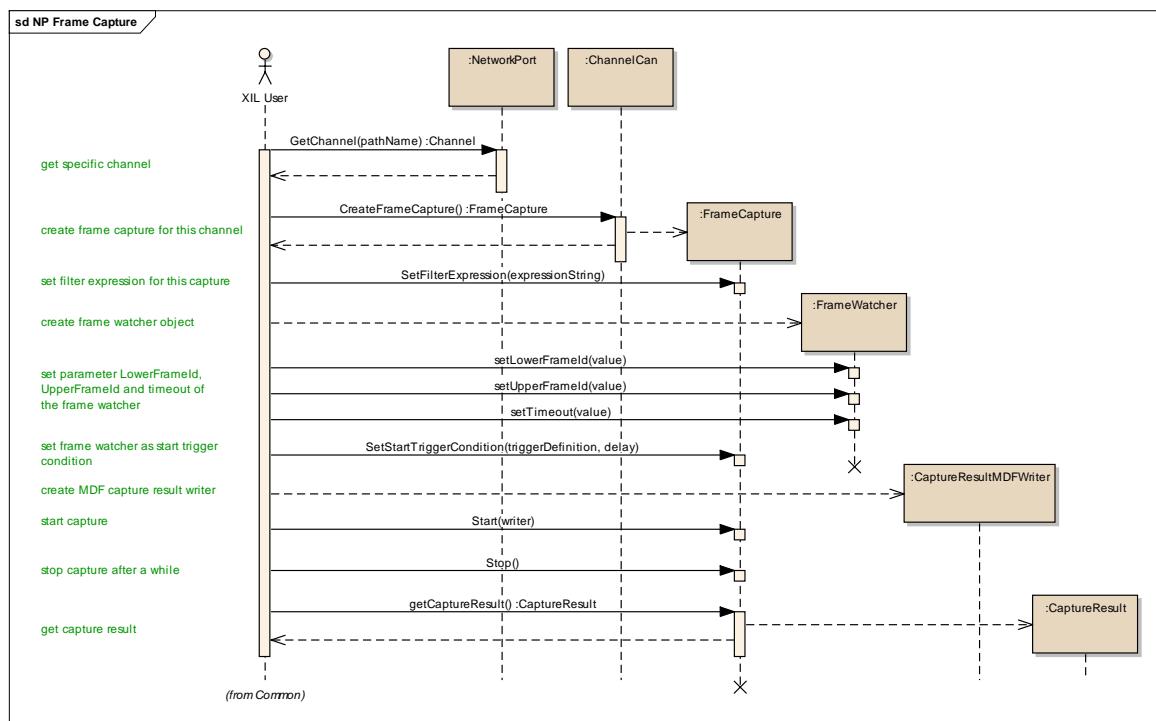


Figure 223: Usage of frame capturing

5.7.2.9 REPLAY OF BUS FRAMES

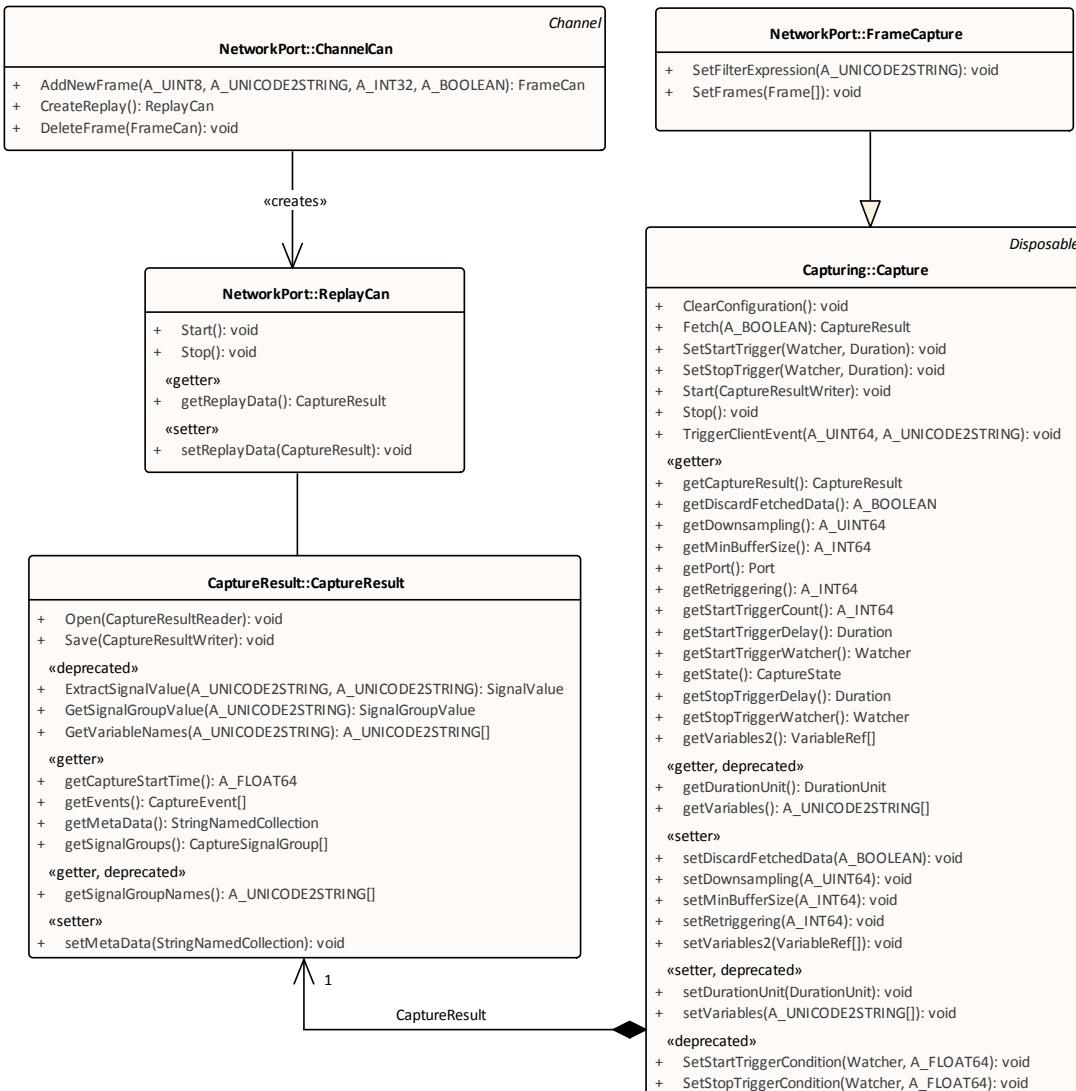


Figure 224: Frame Replay

Besides the frame capturing it is also possible to make a replay of captured frames. But a frame replay is only possible on the CAN channel level in the Network Port hierarchy. To make a frame replay the class `ReplayCan` can be created with the method `CreateReplay` on the `ChannelCan` class instance.

On the `ReplayCan` object the data to replay will be set with the property `ReplayData`. The replay data comes from a previously recorded `CaptureResult` which contains CAN Frames. The `CaptureResult` data can be loaded from file with a `CaptureResultReader` (e.g. `CaptureResultMDFReader`) or used directly from memory from a previously frame capture. The frame replay can be controlled with method `Start` and `Stop` to start and stop the replay. Figure 225 shows an example usage of a frame replay.

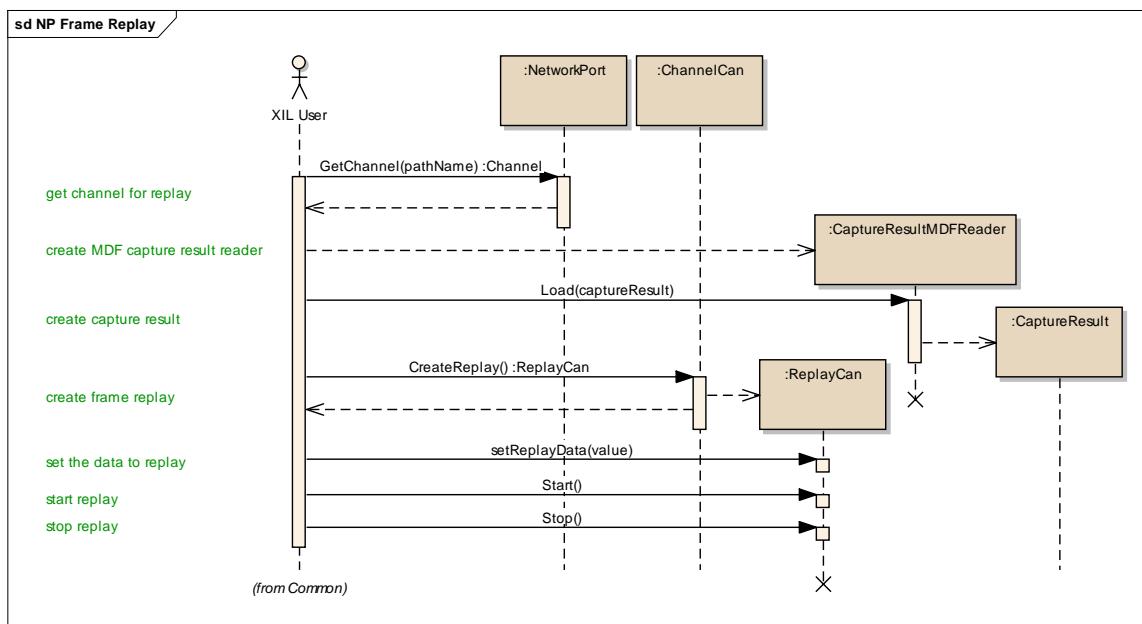


Figure 225: Usage of frame replay

5.7.2.10 EXTENDING THE CAN OBJECT MODEL

The Network Port allows changes to the CAN object tree after creation by adding or deleting frames, PDUs and signals to and from the model. But modifications are only allowed in state eSTOPPED. To add and delete frames the class ChannelCan has the methods AddNewFrame and DeleteFrame. PDUs can be added or deleted with the method call AddNewPdu and DeletePdu on a Frame class instance.

Signals will be created by the method call AddNewSignal and deleted by the call DeleteSignal on a Pdu class instance. There are several predefined values available for the creation of different signal types. Following signal types are defined (see also [Figure 226](#)):

- StandardSignal
- CounterSignal
- ParitySignal
- CRCSignal
- MultiplexedSignal
- MultiplexorSignal
- ToggleSignal

The creation of these types can be managed by the parameter signalType in the method AddNewSignal.

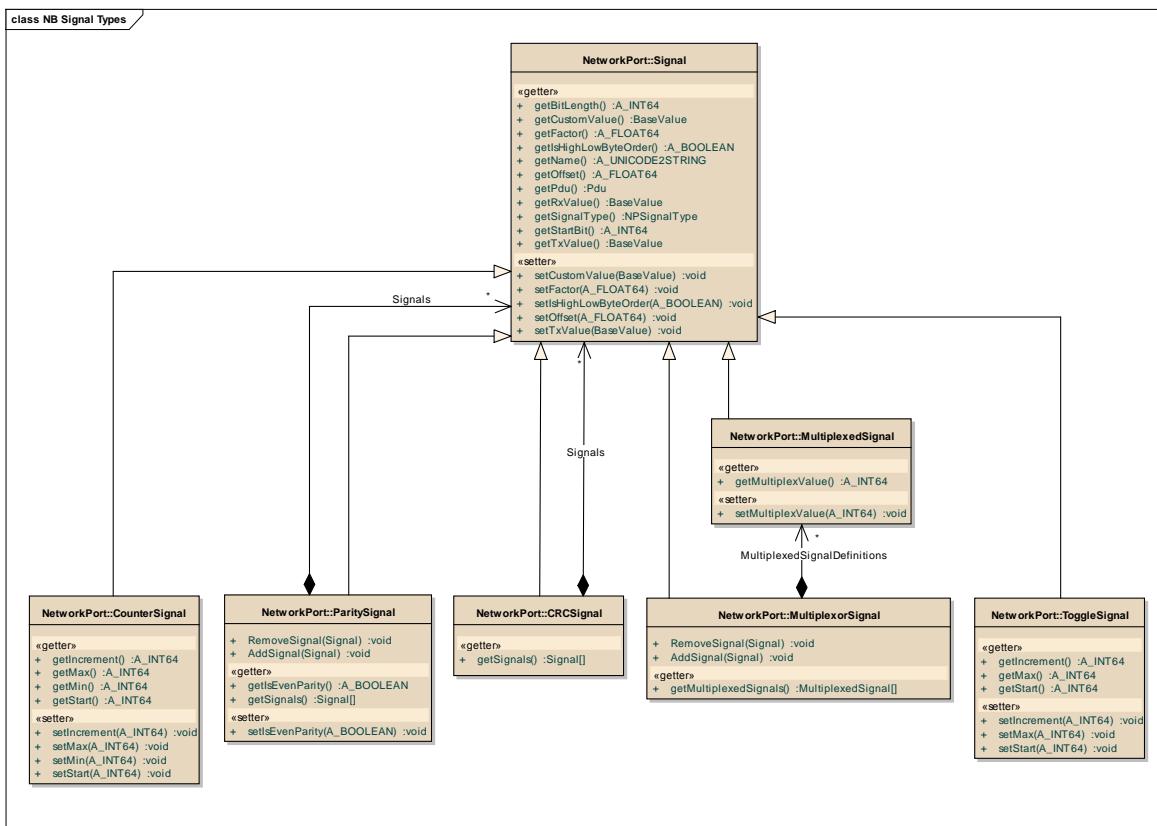


Figure 226: Different signal types

CAN Counter Signal Example

The following example demonstrates the creation of a new CAN counter signal in a new CAN frame. A counter signal is a simple counter value which will be incremented each time it will be sent. The increment of the counter can be set with property Increment. To set the possible value range of this signal the properties Min and Max are available (see also [Figure 227](#)).

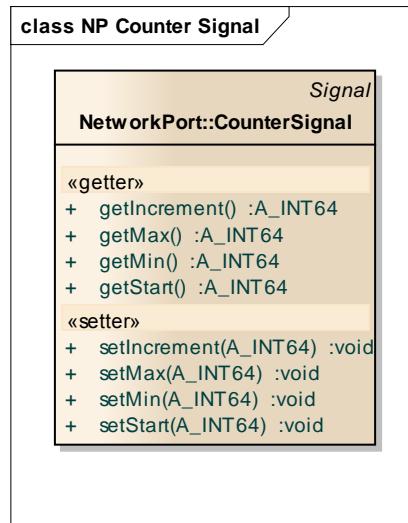


Figure 227: Counter Signal Class

[Figure 228](#) shows a sequence diagram for the creation of CAN counter signal.

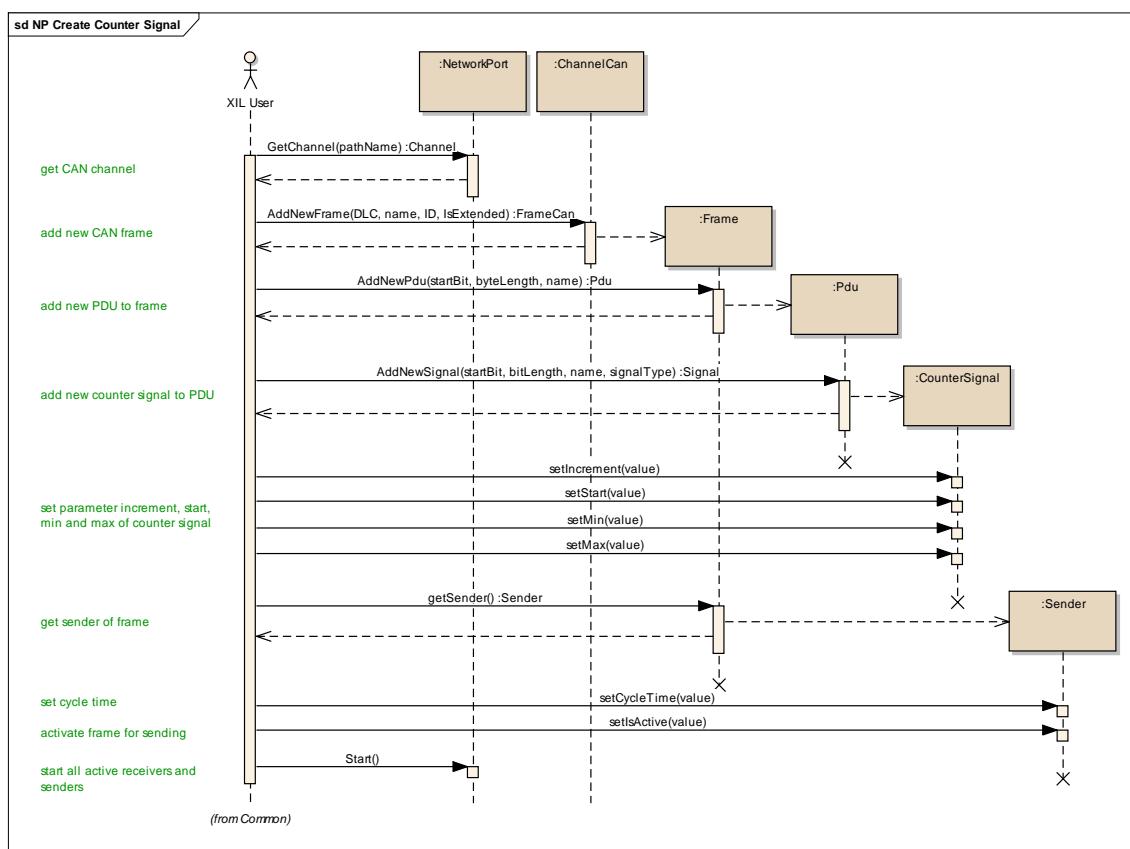


Figure 228: Add counter signal to model

First of all the CAN channel is needed on which the new CAN frame should be added. The new CAN frame will be added to the channel by calling method `AddNewFrame`. Now a new PDU has to be added to the frame by calling `AddNewPdu`. At next the counter signal can be created with the method `AddNewSignal`. On the new created counter signal the parameter for increment, min, max and start can be set. Now the sender for this frame can be configured and finally started.

6 Symbols and Abbreviated Terms

<i>API</i>	Application Programming Interface
<i>ASAM</i>	Association for Standardisation of Automation and Measuring Systems
<i>CAN</i>	Controller Area Network
<i>DTC</i>	Diagnostic Trouble Code
<i>ECU</i>	Electronic Control Unit
<i>EEPROM</i>	Electrically Erasable Programmable Read Only Memory
<i>EES</i>	Electrical Error Simulation
<i>FIU</i>	Failure Injection Unit
<i>GES</i>	General Expression syntax
<i>HIL</i>	Hardware In the Loop
<i>LIN</i>	Local Interconnect Network
<i>MIL</i>	Model In the Loop
<i>PC</i>	Personal Computer
<i>RS232</i>	Recommended Standard 232 (standard for serial binary data signals)
<i>SIL</i>	Software In the Loop
<i>sti</i>	stimulus description file (XML format)
<i>stz</i>	stimulus description file (zip archive)
<i>SUT</i>	System Under Test
<i>TCP/IP</i>	Transmission Control Protocol/Internet Protocol
<i>UML</i>	Unified Modeling Language
<i>XIL</i>	Generic Simulator Interface
<i>XML</i>	Extensible Markup Language
<i>XSD</i>	XSD XML Schema Definition

7 Bibliography

- [1] ASAM e.V.: ASAM Data Types; 2005
- [2] ASAM e.V.: ASAM General Expression; 2017
- [3] ASAM e.V.: ASAM Measurement Data Format; 2014
- [4] The MathWorks Inc.: MAT-File Format; 2013;
http://www.mathworks.com/help/pdf_doc/matlab/matfile_format.pdf
- [5] ASAM e.V.: C# API Technology Reference Mapping Rules; 2017
- [6] ASAM e.V.: Python API Technology Reference Mapping Rules; 2017
- [7] ASAM e.V.: Modeling Guidelines
- [8] FMI Functional Mock-up Interface for Model Exchange and Co-Simulation
https://svn.modelica.org/fmi/branches/public/specifications/v2.0/FMI_for_ModelExchange_and_CoSimulation_v2.0.pdf
- [9] ANSI: IEEE Standard for Floating-Point Arithmetic; 2008

Appendix A. SYNTAX OF WATCHER CONDITIONS

In ASAM XIL the General Expression Syntax is used for defining watcher conditions. Not all functions and operators of the ASAM General Expression Syntax [2] are allowed. This appendix defines the subset of the ASAM General Expression Syntax supported by watcher conditions in ASAM XIL.

Table 59 Operators and Functions supported by ASAM XIL ConditionWatcher

Semantic	Syntax and Arguments	Condition Watcher
Sequential evaluation of sub-expressions: when expr1 evaluates to true, the evaluation of expr2 starts (and continues even if expr1 does not remain true)	expr1 &> expr2	✓
Conditional operator	expr1 ? expr2 : expr3	-
Logical or	expr1 expr2	✓
Logical xor	expr1 ^^ expr2	✓
Logical and	expr1 && expr2	✓
Bitwise or (inclusive or)	expr1 expr2	-
Bitwise xor (exclusive or)	expr1 ^ expr2	-
Bitwise and	expr1 & expr2	-
Equality; implementation of comparison of floating-point numbers is implementation specific	expr1 == expr2	✓
Non-equality	expr1 != expr2	✓
Less than	expr1 < expr2	✓
Greater than	expr1 > expr2	✓
Less than or equal to	expr1 <= expr2	✓
Greater than or equal to	expr1 >= expr2	✓
Bitwise shift left, 0 is added at LSB	expr1 << expr2	-
Bitwise shift right, 0 is added at the MSB if MSB was 0 else 1 is added	expr1 >> expr2	-
Addition	expr1 + expr2	✓
Subtraction	expr1 - expr2	✓
Multiplication	expr1 * expr2	✓
Division	expr1 / expr2	✓
Modulo operation	expr1 % expr2	-
Negation	- expr	✓
Positive sign; has no effect, just to show a positive number like in C	+ expr	✓

Semantic	Syntax and Arguments	Condition Watcher
Logical negation	! expr	✓
Bitwise complement	~ expr	-
Postfix operator .	identifier.identifier	-
Array element access	identifier[decimal-constant]	-
Sine (argument in radians)	sin(expr)	✓
Cosine (argument in radians)	cos(expr)	✓
Tangent (argument in radians)	tan(expr)	-
Arc sine (return value in radians)	asin(expr)	-
Arc cosine (return value in radians)	acos(expr)	-
Arc tangent (return value in radians)	atan(expr)	-
Hyperbolic sine	sinh(expr)	-
Hyperbolic cosine	cosh(expr)	-
Hyperbolic tangent	tanh(expr)	-
Natural logarithm (base e)	log(expr)	-
Common logarithm (base 10)	log10(expr)	-
Exponential function, returns e^{Number}	exp(expr)	-
Power ($\text{pow}(a,b) \rightarrow a^b$)	pow(expr1, expr2)	✓
Power operator ($a^{**b} \rightarrow a^b$)	expr ** expr	✓
Square root	sqrt(expr)	-
Absolute value	abs(expr)	✓
Sign (returns -1 for negative number, 0 if zero, +1 for positive number)	sgn(expr)	-
Returns the nearest integer of the given number	round(expr)	-
Returns smallest integer that is greater than or equal to the given number	ceil(expr)	-
Returns largest integer that is less than or equal to the given number	floor(expr)	-
Minimum	min(expr1, expr2)	✓
Maximum	max(expr1, expr2)	✓
Detection of positive edge: returns true when the value of expr1 changes from a value lower than threshold expr2Threshold to a value greater than or equal to expr2Threshold	posedge(expr1, expr2Threshold)	✓
Detection of negative edge: returns true when the value of expr1 changes from a value greater than threshold expr2Threshold to a value less than or	negedge(expr1, expr2Threshold)	✓

Semantic	Syntax and Arguments	Condition Watcher
equal to <i>expr2Threshold</i>		-
Detection of positive edge: returns true when the value of <i>expr1</i> changes from a value lower than threshold <i>expr2Threshold</i> to a value greater than <i>expr2Threshold</i>	strictposedge(expr1, <i>expr2Threshold</i>)	-
Detection of negative edge: returns true when the value of <i>expr1</i> changes from a value greater than threshold <i>expr2Threshold</i> to a value lower than <i>expr2Threshold</i>	strictnegedge(expr1, <i>expr2Threshold</i>)	-
Detection of value change: returns true when the value of <i>expr1</i> is increased or decreased with respect to the previous evaluation step and the absolute value change is greater than or equal to <i>expr2Delta</i>	changed(expr1, <i>expr2Delta</i>)	✓
Detection of a positive value change: returns true when the value of <i>expr1</i> is increased with respect to the previous evaluation step and the increase is greater than or equal to <i>expr2Delta</i>	changedpos(expr1, <i>expr2Delta</i>)	✓
Detection of a negative value change: returns true when the value of <i>expr1</i> is decreased with respect to the previous evaluation step and the decrease is greater than or equal to <i>expr2Delta</i>	changedneg(expr1, <i>expr2Delta</i>)	✓
Detection of hardware trigger	hwtrigger()	-
Detection of manual trigger	mantrigger()	-

A.1. OTHER RESTRICTIONS

- The first parameter of the functions posedge, negedge, changed, changedpos and changedneg must be only identifiers, not expressions.
- The number of significant initial characters of an identifier is 32.
- The predefined constants INF and NaN of the GES shall not be used.
- The predefined constant epsilon may only be used as expression for the parameter expr2Delta of changed, changedpos and changedneg to designate an arbitrary small value unequal to zero.

A.2. SYNTAX OVERVIEW

The following syntax defines the subset of the ASAM GES that can be used in XIL watcher conditions:

XIL-Watcher-Condition:

and-then-expression

and-then-expression:

conditional-expression

and-then-expression &> conditional-expression

conditional-expression:

// GES conditional operator omitted here

logical-OR-expression

logical-OR-expression:

logical-XOR-expression

logical-OR-expression || logical-XOR-expression

logical-XOR-expression:

logical-AND-expression

logical-XOR-expression ^^ logical-AND-expression

logical-AND-expression:

inclusive-OR-expression

logical-AND-expression && inclusive-OR-expression

inclusive-OR-expression:

// GES bitwise or (inclusive or) omitted here

exclusive-OR-expression

exclusive-OR-expression:

// GES bitwise xor (exclusive or) omitted here

AND-expression

AND-expression:

// GES bitwise and omitted here

equality-expression

equality-expression:

relational-expression

equality-expression == relational-expression

equality-expression != relational-expression

relational-expression:

shift-expression

relational-expression < shift-expression

relational-expression > shift-expression

relational-expression <= shift-expression

relational-expression >= shift-expression

shift-expression:

// GES shift operators omitted here

additive-expression

additive-expression:

multiplicative-expression

additive-expression + multiplicative-expression

additive-expression - multiplicative-expression

multiplicative-expression:

// GES modulo operation omitted here

```
power-expression
multiplicative-expression * power-expression
multiplicative-expression / power-expression

power-expression:
    unary-expression
    power-expression ** unary-expression

unary-expression:
    postfix-expression
    function-call-expression
    unary-operator unary-expression

unary-operator: one of           // GES bitwise not operator omitted here
    + - !

postfix-expression:             // GES postfix operators . and [ ] omitted here
    primary-expression

function-call-expression:       // GES nullary-function-call-expression omitted here
    unary-function-call-expression
    binary-function-call-expression

unary-function-call-expression:
    unary-built-in-function ( conditional-expression )

binary-function-call-expression:
    binary-built-in-function ( conditional-expression , conditional-expression )

unary-built-in-function: one of // some GES built-in functions omitted here
    sin cos abs

binary-built-in-function: one of // some GES built-in functions omitted here
    pow min max
    posedge negedge
    changed changedpos changedneg

primary-expression:
    identifier
    constant
    ( conditional-expression )
```

Appendix B. SYNTAX OF CONSTSYMBOL EXPRESSIONS

In ASAM XIL the General Expression Syntax is used for defining ConstSymbol expressions. Not all functions and operators of the ASAM General Expression [2] are allowed. This appendix defines the subset of ASAM General Expression Syntax supported by ConstSymbol expressions in ASAM XIL.

Table 60 Operators and Functions supported by ConstSymbol.

Semantic	Syntax and Arguments	ConstSymbol
Sequential evaluation of sub-expressions: when expr1 evaluates to true, the evaluation of expr2 starts (and continues even if expr1 does not remain true)	expr1 &> expr2	-
Conditional operator	expr1 ? expr2 : expr3	-
Logical or	expr1 expr2	-
Logical xor	expr1 ^^ expr2	-
Logical and	expr1 && expr2	-
Bitwise or (inclusive or)	expr1 expr2	-
Bitwise xor (exclusive or)	expr1 ^ expr2	-
Bitwise and	expr1 & expr2	-
Equality; implementation of comparison of floating-point numbers is implementation specific	expr1 == expr2	-
Non-equality	expr1 != expr2	-
Less than	expr1 < expr2	-
Greater than	expr1 > expr2	-
Less than or equal to	expr1 <= expr2	-
Greater than or equal to	expr1 >= expr2	-
Bitwise shift left, 0 is added at LSB	expr1 << expr2	-
Bitwise shift right, 0 is added at the MSB if MSB was 0 else 1 is added	expr1 >> expr2	-
Addition	expr1 + expr2	✓
Subtraction	expr1 - expr2	✓
Multiplication	expr1 * expr2	✓
Division	expr1 / expr2	✓
Modulo operation	expr1 % expr2	-
Negation	- expr	✓
Positive sign; has no effect, just to show a positive number like in C	+ expr	✓
Logical negation	! expr	-

Semantic	Syntax and Arguments	ConstSymbol
Bitwise complement	$\sim \text{expr}$	-
Postfix operator .	identifier.identifier	-
Array element access	identifier[decimal-constant]	-
Sine (argument in radians)	sin(expr)	✓
Cosine (argument in radians)	cos(expr)	✓
Tangent (argument in radians)	tan(expr)	-
Arc sine (return value in radians)	asin(expr)	✓
Arc cosine (return value in radians)	acos(expr)	✓
Arc tangent (return value in radians)	atan(expr)	-
Hyperbolic sine	sinh(expr)	-
Hyperbolic cosine	cosh(expr)	-
Hyperbolic tangent	tanh(expr)	-
Natural logarithm (base e)	log(expr)	✓
Common logarithm (base 10)	log10(expr)	-
Exponential function, returns e^{Number}	exp(expr)	✓
Power ($\text{pow}(a,b) \rightarrow a^b$)	pow(expr1, expr2)	✓
Power operator ($a^{**b} \rightarrow a^b$)	expr ** expr	✓
Square root	sqrt(expr)	-
Absolute value	abs(expr)	✓
Sign (returns -1 for negative number, 0 if zero, +1 for positive number)	sgn(expr)	-
Returns the nearest integer of the given number	round(expr)	-
Returns smallest integer that is greater than or equal to the given number	ceil(expr)	-
Returns largest integer that is less than or equal to the given number	floor(expr)	-
Minimum	min(expr1, expr2)	✓
Maximum	max(expr1, expr2)	✓
Detection of positive edge: returns true when the value of expr1 changes from a value lower than threshold expr2Threshold to a value greater than or equal to expr2Threshold	posedge(expr1, expr2Threshold)	-
Detection of negative edge: returns true when the value of expr1 changes from a value greater than threshold expr2Threshold to a value less than or equal to expr2Threshold	negedge(expr1, expr2Threshold)	-

Semantic	Syntax and Arguments	ConstSymbol
Detection of positive edge: returns true when the value of <i>expr1</i> changes from a value lower than threshold <i>expr2Threshold</i> to a value greater than <i>expr2Threshold</i>	strictposedge(expr1, expr2Threshold)	-
Detection of negative edge: returns true when the value of <i>expr1</i> changes from a value greater than threshold <i>expr2Threshold</i> to a value lower than <i>expr2Threshold</i>	strictnegedge(expr1, expr2Threshold)	-
Detection of value change: returns true when the value of <i>expr1</i> is increased or decreased with respect to the previous evaluation step and the absolute value change is greater than or equal to <i>expr2Delta</i>	changed(expr1, expr2Delta)	-
Detection of a positive value change: returns true when the value of <i>expr1</i> is increased with respect to the previous evaluation step and the increase is greater than or equal to <i>expr2Delta</i>	changedpos(expr1, expr2Delta)	-
Detection of a negative value change: returns true when the value of <i>expr1</i> is decreased with respect to the previous evaluation step and the decrease is greater than or equal to <i>expr2Delta</i>	changedneg(expr1, expr2Delta)	-
Detection of hardware trigger	hwtrigger()	-
Detection of manual trigger	mantrigger()	-

B.1. OTHER RESTRICTIONS

- The number of significant initial characters of an identifier is 32.
- The predefined constants `epsilon` and `Nan` of the GES shall not be used.
- The predefined constant `INF` may only be used as expression to designate the range of data to be used in a DataFileSegment.
- Hexadecimal constants of the GES shall not be used.

B.2. SYNTAX OVERVIEW

The following syntax defines the subset of the ASAM GES that can be used in expressions of ConstSymbol:

```
cs-expression:           // main entry point of the ConstSymbol expression syntax
                    multiplicative-expression
                    cs-expression + multiplicative-expression
                    cs-expression - multiplicative-expression
```

multiplicative-expression:

- power-expression
- multiplicative-expression * power-expression
- multiplicative-expression / power-expression
- multiplicative-expression % power-expression

power-expression:

- unary-expression
- power-expression ** unary-expression

unary-expression:

- postfix-expression
- function-call-expression
- unary-operator unary-expression

unary-operator: one of // GES bitwise not and logical not operators omitted here
+ -

postfix-expression: // GES postfix operators . and [] omitted here
primary-expression

function-call-expression: // GES nullary-function-call-expression omitted here
unary-function-call-expression
binary-function-call-expression

unary-function-call-expression:
unary-built-in-function (cs-expression)

binary-function-call-expression:
binary-built-in-function (cs-expression , cs-expression)

unary-built-in-function: one of // some GES built-in functions omitted here
sin cos asin acos abs log exp

binary-built-in-function: one of // some GES built-in functions omitted here
pow min max

primary-expression:
identifier
constant
(cs-expression)

Appendix C. KEY VALUE PAIRS IN CAPTURERESULT META DATA

Inside the Meta Data Key Value Pairs can be used. An overview about standardized Keys / Value pairs is shown in [Table 61](#). Independent from these standardized Key value pairs user / implementation specific pairs can be used.

Table 61 Overview about reserved Key Value Pairs

Key	Usage area	Value	Description
DEPRECATED ⁶ StartTriggerTimeOut	Meta data of capture results	TRUE FALSE	activation by Time Out activation by Trigger Condition
DEPRECATED StopTriggerTimeOut	Meta data of capture results	TRUE FALSE	activation by Time Out activation by Trigger Condition

⁶ Instead MetaData the CaptureResult's contains CaptureEvent's. The CaptureTriggerEvent has an entry for a time out.

Appendix D. STORAGE OF DATA IN MDF4

For a XIL client exchangeability of MDF4 files between XIL servers of different vendors is as crucial as a common programming interface to read and write these files. If MDF4 files written by a particular XIL server could not be read by other XIL servers, exchange of XIL servers or cooperation of different XIL servers in the same tool chain would be restricted.

Addressing this issue the appendix gives details on the storage of capture data in MDF4 files to ensure exchangeability of these files between XIL servers of different vendors. The following explanations refer to terms and definitions of ASAM MDF (see [3]).

D.1. FRAMEWORK MEASUREMENT DATA

Measurement data in XIL API 2.0 includes numerical data and a corresponding unit for each measured signal. When storing measurement data to a file, additionally all information required to access individual signals unambiguously also has to be included in the file. This includes five pieces of information:

1. The name of the port instance
(abstract port identifier = PortDef.PortInstanceName)
2. The name of the measurement task (concrete task name)
3. In case of real-time applications consisting of several sub-applications
(multi-processor systems), the name of the sub-application
4. The name of the measured variable (concrete variable name; variable path)
5. The name of a corresponding AcquisitionConfiguration (only required when performing a measurement via the Framework)

To identify a signal unambiguously within an MDF4 file, this information is stored within different blocks of the file. In this document, we follow the conventions described in section 4.1 of the ASAM document “MDF - Naming of channels and channel groups”. According to the scheme described there, the above information is stored in the following places:

1. Port identifier: “channel path” and “group path”
2. Task name: “group name”
3. Sub-application name: “channel source” and “group source”
4. Variable name: “channel name”
5. AcquisitionConfiguration name:
This name is stored in the cn_md_comment block of the corresponding channel. Specifically, it is stored in the <description> tag which itself is a sub-tag of the <names> tag (see section 6.4 ‘Alternative names’ of the MDF4 specification).

For an explanation of these terms, please refer to chapter 5.4.3 of the ASAM “MDF Programmer’s Guide” (see [3]).

Units used in XIL API 2.0 are stored in MDF4 by employing the meta data section of the channel block (CNBLOCK). Within the section “cn_md_unit”, XML Code based on the ASAM Harmonized Data Objects standard (A-HDO) is employed. This standard is described in the schema file „harmonizedObjects.xsd“. Generally, the Unit class in XIL API 2.0 has been designed along the lines of the A-HDO units.

Units referred to in a channel block are first defined in the MDF header in section “hd_md_comment”, using the XML tag <ho:UNIT-SPEC>. Individual unit definitions are

enclosed in the tag <ho:UNIT>. Units defined in this way can be referenced from the channel blocks of individual signals.

The properties of the class Unit are encoded in the following way:

- DisplayName => <ho:DISPLAY-NAME>
- Factor => <ho:FACTOR-SI-TO-UNIT>
- Offset => <ho:OFFSET-SI-TO-UNIT>
- PhysicalDimension => <ho:PHYSICAL-DIMENSION-REF>
- Name => <ho:SHORT-NAME>
- IsAbsolute => <ho:DESC>ISABSOLUTE=#</ho:DESC>

Here, the property ‘IsAbsolute’ requires a special treatment, since it is not part of the A-HDO physical dimensions. To encode this property, the tag <ho:DESC> containing a special string is used. This string is „ISABSOLUTE=#“, where the hash is either ‘TRUE’ or ‘FALSE’.

Properties of the class PhysicalDimension are encoded via the XML tag <ho:PHYSICAL-DIMENSION>:

- Current => <ho:CURRENT-EXP>
- Length => <ho:LENGTH-EXP>
- LuminousIntensity => <ho:LUMINOUS-INTENSITY-EXP>
- Mass => <ho:MASS-EXP>
- MolarAmount => <ho:MOLAR-AMOUNT-EXP>
- Name => <ho:SHORT-NAME>
- Temperature => <ho:TEMPERATURE-EXP>
- Time => <ho:TIME-EXP>
- Angle => <ho:DESC>ANGLE-EXP=#</ho:DESC>

To encode the Angle exponent, the tag <ho:DESC> containing a special string is used. This string is „ANGLE-EXP=#“, where the hash denotes the exponent of the angle.

Units defined in this way can be referenced from the section “cn_md_unit” of a signal’s channel block. There, individual units are referenced via the tag <ho_unit unit_ref=""\>, where the attribute “unit_ref” corresponds to the “ID” attribute of the referenced signal’s <ho:UNIT> tag.

One thing regarding the storage of units has to be kept in mind: After loading a RecorderResult into memory (via a RecorderResultReader), measurement data of individual signals is obtained as an instance of the CurveData class. With this instance, the time and signal axes can be accessed as an array of Quantity instances (e.g., FloatQuantity[]). However, this is not the way units are stored within an MDF4 file. There, as described above, a single unit is used for each measured signal. When saving a RecorderResult, the units of individual elements of the measurement data are not considered and the default (Framework) unit for this signal is used.

D.1.1. RETRIGGERED MEASUREMENTS

When using the Retriggering feature of either the Acquisition or the Recorder class, a measurement is not finished when its corresponding stop trigger condition has been fired. Instead, the system again waits for the start trigger condition to occur and thus the measurement cycle starts again, until the number of cycles as specified by the Retriggering property has been recorded.

With regard to the measurement data which are actually stored in a RecorderResult, data is supposed to be recorded only between the activation of the start and stop triggers. This usually leads to a situation where gaps occur in the time axes of individual recorded signals. Thus, this case also has to be considered when writing measurement data to MDF4 files.

To describe these gaps in the measurement data, MDF4 event blocks are used as specified in chapter 5.12 of the ASAM MDF 4.1 Programmer's Guide (see [3]). Specifically, events of the type 'Acquisition Interrupt' (`ev_type = 2`) are employed here. The beginning and ending of gaps is marked by two separate events, each having their property '`ev_range_type`' set to the corresponding value (=1 for the beginning of a gap and =2 for the end). The synchronization type is time in seconds (`ev_sync_type = 1`). The time stamp of an event is expressed by a base value (`ev_sync_base_value`) which is multiplied by a factor (i.e. step size, `ev_sync_factor`).

The following figure shows the MDF4 blocks involved in storing gaps. Each stop trigger event in XIL API has a corresponding event block that marks the beginning of a range. This block also references one or more channel groups that contain the corresponding measurement data (the number of referenced channel groups must be stored in the `ev_scope_count` property).

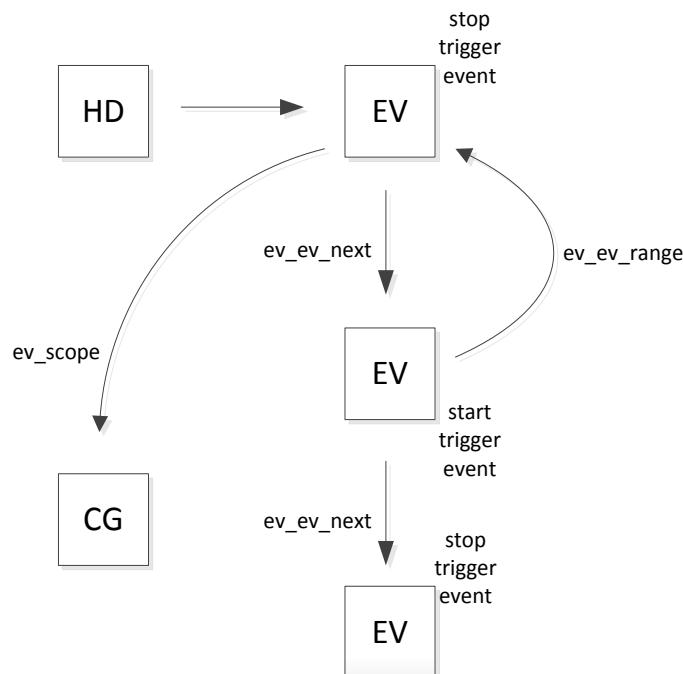


Figure 229: Storage of gaps in the measurement data in MDF4

D.2. TESTBENCH CAPTURE DATA

Special interfaces are provided to store and read capture results to and from MDF4 files (see chapter 5.1.12.7). This section describes how particular data items of these capture results are stored in MDF4 files by the server.

D.2.1. STORAGE OF CLIENT EVENTS

Each CaptureEvent of type `CaptureEventType.eCLIENTEVENT` (aka `CaptureClientEvent`) is stored as event block (EVBLOCK) of type Marker. The `CaptureClientEvent`'s id and



description are stored as individual text blocks (TXBLOCK) and referenced by the EVBLOCK's ev_tx_name and ev_md_comment members. The CaptureClientEvent's id, an integer value of type A_UINT64 (see method TriggerClientEvent of interface Capture) is converted to a character string for storage in a TXBLOCK. This string matches the pattern "XIL_CLIENT_EVENT_<id>", where <id> is the event's id in decimal representation without sign and without leading zeros.

A detailed specification for the assignment of the EVBLOCK's members is given in [Table 62](#).

Note: Only members are listed theirs assignment is restricted to specific values compared to ASAM MDF.

Table 62 EVBLOCK members

Member name	Type	Description
Link section		
ev_ev_parent	LINK	NIL (event is not part of an event hierarchy)
ev_ev_range	LINK	NIL (event refers to a point, not a range)
ev_tx_name	LINK	Pointer to TXBLOCK containing the CaptureClientEvent's id in the form "XIL_CLIENT_EVENT_<id>", where <id> is the Event's id in decimal representation without sign and without leading zeros.
ev_md_comment	LINK	Pointer to TXBLOCK containing the CaptureClientEvent's description or NIL if there is no description, i.e. CaptureClient-Event description is the empty string.
ev_scope	LINK	NIL (event has a global scope, i.e. the event applies to the whole file)
ev_at_reference	LINK	NIL (event does not have any attachments)
Data section		
ev_type	UINT8	6 (Marker)
ev_sync_type	UINT8	1 (calculated synchronization value represents time in seconds)
ev_range_type	UINT8	0 (point)
ev_cause	UINT8	3 (SCRIPT, i.e. event was caused by a scripting command)
ev_flags	UINT8	Bit 0 (post processing flag) is 0
ev_scope_count	UINT32	0 (see ev_scope)
ev_attachment_count	UINT16	0 (see ev_at_reference)
ev_creator_index	UINT16	0 (reference to the FHBLOCK of the application creating the file)

Appendix E. ERROR OVERVIEW

The error codes used by a method are defined in the UML model. They are specified as postconditions in the property dialog of the respective method. A general error condition leading to an error is an invalid object state. The object states in which a method can be called are specified as preconditions in the method's property dialog. Further constraints that lead to the specified errors if they are not met, can be found in the method description.

If a method's postcondition is missing an appropriate error code for a specific error condition, the following error codes shall be used:

- for TestbenchPortExceptions eCOMMON_OTHER_FAIL and
- for FrameworkException eFW_OTHER_FAIL.

The following table contains an overview of all defined error codes and their usage by different methods.

Table 63 Error Overview

Error (Errorcode)	
Error-Description	
Class	Method
eCOMMON_ARGUMENT_NULL (1059)	
The argument {0} must not be null.	
LoopSegment	Add
	Insert
SegmentSignalDescription	Add
	Insert
SignalDescriptionSet	Add
	Load
	Save
SignalDescriptionSetReader	Load
SignalDescriptionSetWriter	Save
SignalSymbol	setSignal
SignalValueSegment	setSignalValue
SignalFactory	CreateSignalDescriptionSetByReader
	CreateSignalValueSegmentByValueAndInterpolation
SymbolFactory	CreateSignalSymbol
	CreateSignalSymbolBySignalDescription
SignalGeneratorReader	Load
SignalGeneratorWriter	Save
eCOMMON_ARGUMENT_OUT_OF_RANGE (1000)	

Error (Errorcode)	
Error-Description	
Class	Method
The argument {0} is out of range. Valid values are: {1}.	
ConstSegment	setStopTrigger
DataFileSegment	setStopTrigger
DynamicErrorBuilder	WithFrequency
	WithResistor
ErrorFactory	CreateErrorPin2Pin
	CreateErrorToGround
	CreateErrorToPotential
	CreateErrorToUbatt
EESPort	WaitForTrigger2
ExpSegment	setTau
	setStopTrigger
IdleSegment	setStopTrigger
MAPort	WaitForBreakpoint
NoiseSegment	setSeed
	setStopTrigger
PulseSegment	setDutyCycle
	setPeriod
	setPhase
	setStopTrigger
RampSlopeSegment	setStopTrigger
ResistorErrorBuilder	WithResistor
SawSegment	setDutyCycle
	setPeriod
	setPhase
	setStopTrigger
Script	WaitForState
SignalDescription	CreateSignalValue
SignalDescriptionSet	CreateSignalGroupValue
SineSegment	setStopTrigger
SegmentSignalDescription	setLoopCount
SignalFactory	CreateConstSegmentBySymbols
	CreateExpSegmentBySymbols
	CreateIdleSegmentByDuration
	CreateNoiseSegmentBySymbols

Error (Errorcode)	
Error-Description	
Class	Method
	CreateDataFileSegmentByParameters
	CreatePulseSegmentBySymbols
	CreateRampSegmentBySymbols
	CreateRampSlopeSegmentBySymbols
	CreateSawSegmentBySymbols
	CreateSineSegmentBySymbols
SimpleErrorBuilder	WithResistor
SineSegment	setPeriod
	setPhase
SpecificErrorFactory	AsDynamic
WatcherFactory	CreateDurationWatcher
	CreateDurationWatcherByCycleNumber
	CreateDurationWatcherByTimeSpan
eCOMMON_ARGUMENT_SIZE_MISMATCH (1093)	
The argument {0} does not match the expected array size.	
MAPort	WriteSimultaneously
	WriteSimultaneously2
eCOMMON_ARGUMENT_TYPE_MISMATCH (1094)	
The argument {0} or one of its elements does not match the expected data type.	
MAPort	WriteSimultaneously
	WriteSimultaneously2
eCOMMON_CAPTURE_CLEAR_CONFIGURATION_FAILED (1074)	
Failed to clear the capture configuration.	
Capture	ClearConfiguration
eCOMMON_CAPTURE_COULD_NOT_FETCH_DATA (1073)	
Failed to fetch capture data.	
Capture	Fetch
eCOMMON_CAPTURE_COULD_NOT_SET_VARIABLES (1079)	
Failed to set the list of capture variables.	
Capture	setVariables
	setVariables2
eCOMMON_CAPTURE_COULD_NOT_START_CAPTURE (1078)	
Failed to start capturing.	
Capture	Start
eCOMMON_CAPTURE_COULD_NOT_STOP_CAPTURE (1082)	

Error (Errorcode)	
Error-Description	
Class	Method
Failed to stop capturing.	
Capture	Stop
eCOMMON_CAPTURE_GET_DOWNSAMPLING_FAILED (1075)	
Failed to get the downsampling factor.	
Capture	getDownsampling
eCOMMON_CAPTURE_GET_DURATION_UNIT_FAILED (1077)	
Failed to get the duration unit.	
Capture	getDurationUnit
eCOMMON_CAPTURE_GET_STATE_FAILED (1081)	
Failed to get the current capture state.	
Capture	getState
eCOMMON_CAPTURE_SET_DOWNSAMPLING_FAILED (1088)	
Failed to set the downsampling factor.	
Capture	setDownsampling
eCOMMON_CAPTURE_SET_DURATION_UNIT_FAILED (1076)	
Failed to set the duration unit.	
Capture	setDurationUnit
eCOMMON_CAPTURERESULT_READ_FAILED (1044)	
Could not read CaptureResult. Reason: {0}.	
CaptureResult	Open
CaptureResultReader	Load
eCOMMON_CAPTURERESULT_SAVE_FAILED (1045)	
Could not save CaptureResult. Reason: {0}.	
Capture	Start
CaptureResult	Save
CaptureResultWriter	Save
eCOMMON_CONSISTENCY_CHECK_FAILED (1097)	
Constraint violated: {0}.	
Script	LoadToTarget
SignalDescriptionSet	CheckConsistency
SignalGenerator	CheckConsistency
eCOMMON_COULD_NOT_CREATE_CAPTURE (1062)	
Could not create an instance of class Capture.	
MAPort	CreateCapture

Error (Errorcode)	
Error-Description	
Class	Method
eCOMMON_COULD_NOT_EXTRACT_SIGNALVALUE (1072)	
Could not extract the signal value for variable '{0}' and signal group '{1}'.	
CaptureResult	ExtractSignalValue
CaptureSignalGroup	GetScalarSignalValues
	GetSignalValues
eCOMMON_COULD_NOT_GET_RETRIGGERING (1086)	
Could not get retrigerring.	
Capture	getRetrigerring
eCOMMON_COULD_NOT_OPEN_FILE (1001)	
Could not read CaptureResult. Reason: {0}.	
eCOMMON_COULD_NOT_SAVE_FILE (1002)	
Could not save CaptureResult. Reason: {0}.	
eCOMMON_COULD_NOT_SET_DOWNSAMPLING (1034)	
Could not set downsampling {0}.	
eCOMMON_COULD_NOT_SET_RETRIGGERING (1087)	
Could not set retrigerring.	
Capture	setRetrigerring
eCOMMON_COULD_NOT_SET_TASK_NAME (1063)	
Could not set the capture task name.	
MAPort	CreateCapture
	ReadSimultaneously
	ReadSimultaneously2
	WriteSimultaneously
	WriteSimultaneously2
eCOMMON_COULD_NOT_SET_TRIGGERCONDITION (1080)	
Failed to set the trigger condition.	
Capture	SetStartTrigger
	SetStartTriggerCondition
	SetStopTriggerCondition
	SetStopTrigger
eCOMMON_CREATE_SIGNALGROUPVALUE_FAILED (1061)	
Could not create numeric signal data from a SignalDescriptionSet.	

Error (Errorcode)	
Error-Description	
Class	Method
SignalDescriptionSet	CreateSignalGroupValue
eCOMMON_CREATE_SIGNALVALUE_FAILED (1060)	
Could not create numeric signal data from a SignalDescription.	
SignalDescription	CreateSignalValue
eCOMMON_CREATE_TESTBENCH_FAILED (1058)	
Error during loading of the requested Testbench library occurred.	
TestbenchFactory	CreateVendorSpecificTestbench
TestbenchFactory	CreateVendorSpecificTestbench2
eCOMMON_DEFINE_NOT_AVAILABLE (1003)	
Define {0} not available.	
ConditionWatcher	setCondition
	setDefines
	setDefines2
WatcherFactory	CreateConditionWatcher
	CreateConditionWatcher2
eCOMMON_DESTROYONTARGET_FAILED (1057)	
An error occurred in destroy the signal generator. Reason: {0}.	
Script	DestroyOnTarget
eCOMMON_ELAPSEDTIME_FAILED (1085)	
Failed to get the elapsed time of the signal generator.	
SignalGenerator	getElapsedTime
eCOMMON_ELEMENT_NOT_FOUND (1005)	
The specified alias name does not exist.	
Attributes	GetProperty
eCOMMON_EXPRESSION_SYNTAX_ERROR (1101)	
Syntax error in expression {0}.	
SymbolFactory	CreateConstSymbolByExpression
ConstSymbol	setExpression
eCOMMON_FILE_ACCESS_ERROR (1006)	
The file {0} cannot be accessed.	
DiagPort	LoadConfiguration
ECUCPort	LoadConfiguration
ECUMPort	LoadConfiguration
EESPort	LoadConfiguration

Error (Errorcode)	
Error-Description	
Class	Method
ErrorConfiguration	CreateEESConfigurationFileReader
	CreateEESConfigurationFileWriter
MAPort	LoadConfiguration
	DownloadParameterSets
NetworkPort	LoadConfiguration
eCOMMON_FILE_NOT_FOUND (1037)	
A file with the name {0} is not found.	
DiagPort	LoadConfiguration
ECUCPort	LoadConfiguration
ECUMPort	LoadConfiguration
EESPort	LoadConfiguration
MAPort	LoadConfiguration
	DownloadParameterSets
NetworkPort	LoadConfiguration
eCOMMON_IDENTIFIER_NOT_FOUND (1007)	
Identifier {0} not found.	
CaptureResult	ExtractSignalValue
	GetSignalGroupValue
	GetVariableNames
SignalGroupValue	ExtractSignalValue
eCOMMON_IDENTIFIER_SYNTAX_ERROR (1096)	
Identifier {0} does not comply with the required syntax.	
SignalFactory	CreateSignalDescriptionParameter
eCOMMON_INCONSISTENCE_PARAMETER (1089)	
The number of values is inconsistent with number of names or types fit not to variables.	
eCOMMON_INCONSISTENT_VECTOR_SIZES (1008)	
X vector and function vector do not have same size.	
ValueFactory	CreateCurveValue
	CreateSignalValue
	CreateXYValue
XYValue	SetValues
eCOMMON_INCONSISTENT_X_Y_SIGNALGROUP_SIZES (1028)	
Length of X vector the length of the Y vectors are not the same.	
SignalGroupValue	setXVector

Error (Errorcode)	
Error-Description	
Class	Method
	setYVectors
ValueFactory	CreateSignalGroupValue
eCOMMON_INCONSISTENT_XVECTOR_SIZES (1009)	
Length of X vector and column count of function values are not the same.	
MapValue	setFcnValues
	setXVector
ValueFactory	CreateMapView
eCOMMON_INCONSISTENT_Y_SIGNALGROUP_SIZES (1029)	
The Y vectors do not have the same length.	
SignalGroupValue	setYVectors
ValueFactory	CreateSignalGroupValue
eCOMMON_INCONSISTENT_YVECTOR_SIZES (1010)	
Length of Y vector and row count of function values are not the same.	
MapValue	setFcnValues
	setYVector
ValueFactory	CreateMapView
eCOMMON_INDEX_OF_END_VALUE_NOT_FOUND (1031)	
The end value {0} has not been found.	
CaptureSignalGroup	GetSliceByTimeRange
SignalGroupValue	GetSliceByXRange
eCOMMON_INDEX_OF_START_VALUE_NOT_FOUND (1030)	
The start value {0} has not been found.	
CaptureSignalGroup	GetSliceByTimeRange
SignalGroupValue	GetSliceByXRange
eCOMMON_INDEX_OUT_OF_RANGE (1011)	
The element with index {0} is not accessible. Index range is: 0 .. {1} .	
BooleanMatrixValue	GetColumnValues
	GetRowValues
	GetValueByIndex
	SetColumnValues
	SetRowValues
	SetValueByIndex
BooleanVectorValue	GetValueByIndex
	SetValueByIndex
CaptureSignalGroup	GetSliceByIndexRange

Error (Errorcode)	
Error-Description	
Class	Method
DiagTroubleCodeByA_UINT64Collection	GetByKey
ErrorConfiguration	RemoveAt
ErrorSet	RemoveAt
FloatMatrixValue	GetColumnValues
	GetRowValues
	GetValueByIndex
	SetColumnValues
	SetRowValues
	SetValueByIndex
FloatVectorValue	GetValueByIndex
	SetValueByIndex
IntMatrixValue	GetColumnValues
	GetRowValues
	GetValueByIndex
	SetColumnValues
	SetRowValues
	SetValueByIndex
IntVectorValue	GetValueByIndex
	SetValueByIndex
LoopSegment	GetByIndex
	Insert
	RemoveByIndex
SegmentSignalDescription	GetByIndex
	Insert
	RemoveByIndex
SignalDescriptionSet	GetByIndex
	Insert
	RemoveByIndex
SignalGroupValue	ExtractSignalValueByIndex
	GetSliceByIndex
	GetYVectorByIndex
StringMatrixValue	GetColumnValues
	GetRowValues
	GetValueByIndex
	SetColumnValues
	SetRowValues

Error (Errorcode)	
Error-Description	
Class	Method
StringVectorValue	SetValueByIndex
	GetValueByIndex
	SetValueByIndex
UintMatrixValue	GetColumnValues
	GetRowValues
	GetValueByIndex
	SetColumnValues
	SetRowValues
	SetValueByIndex
UintVectorValue	GetValueByIndex
	SetValueByIndex
eCOMMON_INFO_NOT_AVAILABLE (1012)	
The requested information {0} is not available.	
Capture	getVariables
CaptureSignalGroup	GetScalarSignalValues
	GetSignalValues
	GetTimestamps
CompuMethodInfo	GetAxisElementCompuMethod
ConditionWatcher	getDefines
ConstSymbol	getValue
DataTypeInfo	GetAxisElementType
	GetSize
DurationWatcher	getDuration
Framework	getBuildNumber
	getMajorNumber
	getMinorNumber
	getProductVersion
	getRevisionNumber
	getVendorName
Testbench	getBuildNumber
	getMajorNumber
	getMinorNumber
	getProductName
	getProductVersion
	getRevisionNumber

Error (Errorcode)	
Error-Description	
Class	Method
	getVendorName
UnitInfo	GetAxisElementUnit
VariableInfo	GetDataTypeInfo
	getCompuMethodInfo
eCOMMON_INSTANCE_ERROR (1050)	
Object creation or retrieval failed	
DiagPortFactory	CreateDiagPort
ECUCPortFactory	CreateECUCPort
ECUMPortFactory	CreateECUMPort
NetworkPortFactory	CreateNetworkPort
TestbenchFactory	CreateVendorSpecificTestbench
TestbenchFactory	CreateVendorSpecificTestbench2
eCOMMON_INVALID_DURATION (1013)	
Invalid duration value. Duration has to be a positive floating point number.	
ConstSegment	setDuration
DurationWatcher	setDuration
ExpSegment	setDuration
FWDurationWatcher	setDuration
IdleSegment	setDuration
NoiseSegment	setDuration
PulseSegment	setDuration
RampSegment	setDuration
RampSlopeSegment	setDuration
SawSegment	setDuration
SineSegment	setDuration
eCOMMON_INVALID_METHODCALL_FETCH (1039)	
It is not possible to call the method Fetch() while storing capture data to file.	
Capture	Fetch
eCOMMON_INVALID_PORT_SETTING (1046)	
Setting value out of range or unknown setting option	
DiagPort	LoadConfiguration
DiagPortConfig	setProject
	setVehicleInfoTable
ECUCPort	LoadConfiguration
ECUCPortConfig	setA2LFile

Error (Errorcode)	
Error-Description	
Class	Method
	setImageFile
ECUMPort	LoadConfiguration
ECUMPortConfig	setA2LFile
EESPort	LoadConfiguration
MAPort	LoadConfiguration
MAPortConfig	setModelFile
NetworkPort	LoadConfiguration
PortConfig	setVendorSpecificConfiguration
eCOMMON_INVALID_STATE (1041)	
This method can not be called in this state.	
Capture	ClearConfiguration
	Fetch
	getCaptureResult
	setDiscardFetchedData
	setDownsampling
	setDurationUnit
	setMinBufferSize
	SetStartTrigger
	SetStartTriggerCondition
	SetStopTrigger
	SetStopTriggerCondition
	setVariables
	setVariables2
	setRetriggering
	Start
Channel	Stop
	TriggerClientEvent
ChannelCan	StartReceiving
	StartSending
DiagPort	AddNewFrame
	DeleteFrame
Disposable	Configure
	GetECU
	GetFunctionalGroup
Disposable	Dispose

Error (Errorcode)	
Error-Description	
Class	Method
ECUCPort	CheckVariableNames
	CheckVariableRefs
	CalculateRefPageCRC
	CalculateWorkPageCRC
	Configure
	GetDataType
	getVariableNames
	IsReadable
	IsWriteable
	NumberOfPages
	Read
	Read2
	StartOnlineCalibration
	StopOnlineCalibration
	SwitchToRefPage
	SwitchToWorkPage
	Write
	Write2
ECUMPort	CheckVariableNames
	CheckVariableRefs
	Configure
	CreateCapture
	GetDataType
	GetMeasuringVariables
	getMeasuringVariables2
	getTaskNames
	getTaskInfos
	getVariableNames
	IsReadable
	Read
	Read2
	SetMeasuringVariables
	setMeasuringVariables2
	StartMeasurement
	StopMeasurement
EESPort	Activate

Error (Errorcode)	
Error-Description	
Class	Method
	Configure
	Deactivate
	Download
	getPotentialInfos
	getSignalInfos
	SetErrorConfiguration
	Trigger
	Unload
	Update
	WaitForTrigger
	WaitForTrigger2
Frame	AddNewPdu
Frame	DeletePdu
MAPort	CheckVariableNames
MAPort	CheckVariableRefs
MAPort	Configure
MAPort	CreateCapture
MAPort	CreateSignalGenerator
MAPort	DownloadParameterSets
MAPort	getDAQClock
MAPort	GetDataType
MAPort	getTaskNames
MAPort	GetVariableInfo
MAPort	getVariableNames
MAPort	IsReadable
MAPort	IsWritable
MAPort	Read
MAPort	Read2
MAPort	StartSimulation
MAPort	StopSimulation
MAPort	Write
MAPort	Write2
MAPort	ReadSimultaneously
MAPort	ReadSimultaneously2
MAPort	WriteSimultaneously
MAPort	WriteSimultaneously2

Error (Errorcode)	
Error-Description	
Class	Method
	getSimultaneousLevel
	CreateTargetScript
	PauseSimulation
	WaitForBreakpoint
	setBreakpoint
	getBreakpoint
	getSimulationStepSize
	getTaskInfos
MultiplexorSignal	AddSignal
	RemoveSignal
NetworkPort	Configure
	Read
	Start
	Stop
	Write
ParitySignal	AddSignal
	RemoveSignal
Pdu	AddNewSignal
	DeleteSignal
Port	Disconnect
Receiver	SingleReceive
	StartReceiving
ReplayCan	Start
Script	DestroyOnTarget
	getStopInfo
	LoadToTarget
	Pause
	SetParameterValue
	Start
	Stop
Sender	SendOnce
	StartSending
	StartSequences
SignalGenerator	CheckConsistency
	Load

Error (Errorcode)	
Error-Description	
Class	Method
	setAliasDefinitions
	setAssignments
	setAssignments2
	setSignalDescriptionSet
TargetScript	Load
eCOMMON_INVALID_VARIABLE_REFERENCE (1102)	
Reference to test environment variable {0} cannot be resolved or variable type not permitted.	
Capture	setVariables2
	SetStartTrigger
	SetStopTrigger
CaptureSignalGroup	GetScalarSignalValues
	GetSignalValues
ECUCPort	Read2
	Write2
ECUMPort	Read2
	setMeasuringVariables2
MAPort	setBreakpoint
	Read2
	ReadSimultaneously2
	Write2
	WriteSimultaneously2
eCOMMON_IRREGULAR_OPEN_CALL (1025)	
Open can be called on a empty CaptureResult, only.	
CaptureResult	Open
eCOMMON_LOAD_FAILED (1038)	
Loading failed. Reason: {0}.	
Player	Load
SignalDescriptionSet	Load
SignalDescriptionSetReader	Load
SignalGenerator	Load
SignalGeneratorReader	Load
TargetScript	Load
TargetScriptFileReader	Load
EESConfigurationReader	Load

Error (Errorcode)	
Error-Description	
Class	Method
ErrorConfiguration	Load
eCOMMON_LOADTOTARGET_FAILED (1051)	
Could not load to target. Reason: {0}.	
Script	LoadToTarget
eCOMMON_MDF_VERSION_INVALID_FORMAT (1043)	
Invalid format of specified version. Use "MAJOR.MINOR[.MAINTENANCE]".	
CaptureResultMDFWriter	setVersion
eCOMMON_MDF_VERSION_NOT_SUPPORTED (1042)	
Version "{0}" not supported.	
CaptureResultMDFWriter	setVersion
eCOMMON_MEM_BUFSIZE (1017)	
Could not set buffersize.	
Capture	setMinBufferSize
eCOMMON_NAME_ALREADY_EXISTS (1036)	
An element with the name '{0}' already exists.	
SignalDescription	setName
SignalDescriptionSet	Add
	Insert
	setParameters
eCOMMON_NAME_NOT_FOUND (1018)	
Name {0} not found.	
SignalDescriptionSet	GetByName
	RemoveByName
SignalGenerator	setAssignments2
Script	GetParameterValue
	SetParameterValue
eCOMMON_NO_PORT (1019)	
No port available.	
Capture	Start
eCOMMON_NO_VARIABLE (1020)	
No variable for logging set.	
Capture	Start
eCOMMON_NOT_IMPLEMENTED (1048)	
The requested feature is not supported by this vendor.	
DiagPortFactory	CreateDiagPort

Error (Errorcode)	
Error-Description	
Class	Method
ECUCPortFactory	CreateECUCPort
ECUMPortFactory	CreateECUMPort
NetworkPortFactory	CreateNetworkPort
Testbench	getCapturingFactory
	getDiagPortFactory
	getECUCPortFactory
	getECUMPortFactory
	getEESPortFactory
	getMAPortFactory
	getNetworkPortFactory
	getSignalFactory
	getSignalGeneratorFactory
	getSymbolFactory
	getValueFactory
	getWatcherFactory
	getTargetScriptFactory
eCOMMON_NOT_SUPPORTED (1033)	
Operation {0} is not supported.	
ConditionWatcher	getTimeOut
	setTimeOut
FWConditionWatcher	getTimeOut
	setTimeOut
DiagTroubleCode	GetEnvironmentDataCollection
eCOMMON_PAUSE_FAILED (1055)	
An error occurred when trying to pause execution. Reason: {0}.	
Player	Pause
Script	Pause
eCOMMON_PORT_CONFIGURATION_FAILED (1047)	
Port configuration could not be completed successfully	
DiagPort	Configure
ECUCPort	Configure
ECUMPort	Configure
EESPort	Configure
MAPort	Configure
NetworkPort	Configure

Error (Errorcode)	
Error-Description	
Class	Method
eCOMMON_PORT_NOT_AVAILABLE (1026)	
Port not available.	
ECUMPort	CreateCapture
MAPort	CreateCapture
	Read
	Read2
	Write
	Write2
	ReadSimultaneously
	ReadSimultaneously2
	WriteSimultaneously
	WriteSimultaneously2
eCOMMON_SAVE_FAILED (1056)	
Saving failed. Reason: {0}.	
EESConfigurationWriter	Save
ErrorConfiguration	Save
Player	Save
SignalDescriptionSet	Save
SignalDescriptionSetWriter	Save
SignalGenerator	Save
SignalGeneratorWriter	Save
eCOMMON_SCRIPT_CLIENT_TRIGGERED_STOP (1091)	
Running script was stopped by Client.	
Script	getStopInfo
eCOMMON_SCRIPT_PARAMETER_TYPE_NOT_WRITEABLE (1099)	
Parameter {0} has a type that is not writeable.	
Script	SetParameterValues
eCOMMON_SCRIPT_READ_PARAMETER_FAILED (1098)	
Parameter {0} could not be read.	
Script	GetParameterValue
eCOMMON_SCRIPT_RUNTIME_ERROR (1092)	
Running script was stopped by script runtime error.	
Script	getStopInfo
eCOMMON_SCRIPT_WRITE_PARAMETER_FAILED (1100)	
Parameter {0} could not be written. Reason : {1}	

Error (Errorcode)	
Error-Description	
Class	Method
Script	SetParameterValues
eCOMMON_START_FAILED (1052)	
An error occurred when trying to start execution. Reason: {0}.	
Player	Start
Script	Start
eCOMMON_STARTTRIGGER_NOT_SET (1021)	
Start trigger not set.	
eCOMMON_STARTINDEX_GREATER_THAN_ENDINDEX (1032)	
The start index {0} is greater than the end index {1}.	
CaptureSignalGroup	GetSliceByIndexRange
SignalGroupValue	GetSliceByIndex
eCOMMON_STATEREQUEST_FAILED (1054)	
An error occurred when requesting execution state. Reason: {0}.	
Player	getState
Script	getScriptState
SignalGenerator	getState
eCOMMON_STOP_FAILED (1053)	
An error occurred when trying to stop execution. Reason: {0}.	
Player	Stop
Script	Stop
eCOMMON_STOP_TRIGGER_NOT_SET (1022)	
Stop trigger not set.	
Capture	SetStopTriggerCondition
eCOMMON_TASK_NOT_AVAILABLE (1027)	
Task {0} not available.	
ECUMPort	CreateCapture
	SetMeasuringVariables
	setMeasuringVariables2
MAPort	CreateCapture
	ReadSimultaneously
	ReadSimultaneously2
	WriteSimultaneously
	WriteSimultaneously2
eCOMMON_TIMEOUT_OCCURRED (1095)	

Error (Errorcode)	
Error-Description	
Class	Method
A timeout occurred.	
EESPort	WaitForTrigger2
MAPort	WaitForBreakpoint
Script	WaitForState
eCOMMON_TRIGGER_SYNTAX_ERROR (1023)	
Syntax error in trigger condition {0}.	
ConditionWatcher	setCondition
FWConditionWatcher	setCondition
WatcherFactory	CreateConditionWatcher
	CreateConditionWatcher2
eCOMMON_VARIABLE_NOT_FOUND (1024)	
Variable {0} not found.	
Capture	SetStartTriggerCondition
	SetStopTriggerCondition
	setVariables
CaptureResult	ExtractSignalValue
ECUCPort	GetDataType
	IsReadable
	IsWriteable
ECUMPort	SetMeasuringVariables
	Get(DataType
	IsReadable
MAPort	Get(DataType
	IsReadable
	IsWritable
	Read
	Write
	GetVariableInfo
	ReadSimultaneously
	WriteSimultaneously
eCOMMON_WRONG_VECTOR_TYPE (1040)	
ValueContainer does not accept boolean type as range for a slice.	
SignalGroupValue	GetSliceByXRange
eCONTROL_INIT_FAILED (12004)	
Init was not successfully completed.	

Error (Errorcode)	
Error-Description	
Class	Method
Framework	Init
eCONTROL_LOAD_CONFIGURATION_FAILED (12003)	
Framework configuration file or referenced mapping file could not be loaded successfully.	
Framework	LoadConfiguration
eCONTROL_NO_CONFIGURATION (12000)	
Configuration was not successfully completed.	
Framework	CreateVariable
eCONTROL_NO_INIT (12001)	
Init was not successfully completed.	
Framework	CreateVariable
eCONTROL_SHUTDOWN_FAILED (12005)	
Shutdown was not successfully completed.	
Framework	Shutdown
eCONTROL_UNKNOWN_PORT_NAME (12006)	
Port name {0} not found.	
Framework	GetPort
FrameworkConfig	GetPortDef
eDIAG_COMMUNICATION_ERROR (2000)	
Communication could not be stopped or started.	
ECU	StartCommunication
	StopCommunication
FunctionalGroup	StartCommunication
	StopCommunication
eDIAG_DEVICE_NOT_CONNECTED (2001)	
HW device to access diagnostic bus (i.e. CAN,...) is not connected to PC.	
ECU	ExecuteJob
	GetIdentificationData
	GetVariantData
	SetVariantData
ECUBaseController	ReadFromAddress
	SendHexService
	WriteToAddress
ECUFaultMemory	Clear
	SendHexService
FunctionalGroupFaultMemory	Clear

Error (Errorcode)	
Error-Description	
Class	Method
	Read
eDIAG_DTC_STATUS_NOT_SUPPORTED (2016)	
The requested type is not supported by the database or runtime.	
ECUFaultMemory	Read
FunctionalGroupFaultMemory	Read
eDIAG_INVALID_PARAMETERS_FOR_JOB (2002)	
The parameters {0} are not valid for job {1}.	
ECU	ExecuteJob
eDIAG_NEG_ACKNOWLEDGE (2003)	
Diagnostic service was successfully sent to ECU but negative acknowledge was received from ECU.	
ECUBaseController	ReadFromAddress
	SendHexService
	WriteToAddress
ECUFaultMemory	Clear
FunctionalGroupFaultMemory	Clear
	Read
eDIAG_NO_SERVICE_PROVIDED (2015)	
The service is not provided in the database.	
DiagTroubleCode	GetEnvironmentDataCollection
eDIAG_STATUS_BYTE_NOT_SUPPORTED (2017)	
DTC status byte not supported by protocol.	
DiagTroubleCode	GetStatus
eDIAG_TIMEOUT (2005)	
No response from ECU within required time.	
ECU	ExecuteJob
	GetIdentificationData
	GetVariantData
	SetVariantData
ECUBaseController	ReadFromAddress
	SendHexService
	WriteToAddress
ECUFaultMemory	Clear
	SendHexService
FunctionalGroupFaultMemory	Clear

Error (Errorcode)	
Error-Description	
Class	Method
	Read
eDIAG_UNKNOWN_ALIAS_NAME (2006)	
ALIAS Name {0} not known.	
eDIAG_UNKNOWN_DID (2014)	
The DID does not exist in the database.	
ECU	CreateDIDByName
	CreateDIDByShortIdentifier
	CreateDIDByIdentifier
FunctionalGroup	CreateDIDByName
	CreateDIDByShortIdentifier
	CreateDIDByIdentifier
eDIAG_UNKNOWN_ECU_ID (2007)	
ECU ID {0} not known.	
DiagPort	GetECU
eDIAG_UNKNOWN_JOB (2009)	
Job {0} not known.	
ECU	ExecuteJob
eDIAG_UNKNOWN_LINK_NAME (2010)	
Logical Link Name {0} not known.	
DiagPort	GetECU
	GetFunctionalGroup
eDIAG_UNKNOWN_MEASUREMENT_SIGNAL (2011)	
Measurement signal {0} not known.	
	SendHexService
eDIAG_UNKNOWN_PROJECT (2012)	
Project {0} not known.	
DiagPort	Configure
eDIAG_UNKNOWN_VEHICLE_INFO_TABLE (2013)	
Vehicle Info Table {0} not known.	
DiagPort	Configure
eEES_ARRAY_LENGTH_NOTEQUAL (3006)	
The lengths of the provided arrays are not equal.	
ErrorFactory	CreateErrorMultiPin2Pin

Error (Errorcode)	
Error-Description	
Class	Method
eEES_ERROR_ALREADY_EXISTS (3001)	
An Error with the name {0} already exists in the ErrorSet {1}.	
ErrorSet	AddError
	AddErrors
eEES_ERRORSET_ACTIVATION_FAILED (3009)	
Could not activate the next error set. Reason: {0}	
EESPort	Trigger
eEES_ERRORSET_ALREADY_EXISTS (3000)	
An ErrorSet with the name {0} already exists in the ErrorConfiguration {1}.	
ErrorConfiguration	AddErrorSet
	AddErrorSets
	CreateErrorSet
eEES_HARDWARE_NOT_ACCESSIBLE (3002)	
Operation {0} failed because the EES hardware is not accessible.	
EESPort	Activate
	Deactivate
	Download
	GetActiveErrorSet
	Trigger
	Unload
	Update
	WaitForTrigger2
eEES_INVALID_CONFIGURATION (3003)	
Configuration is invalid. The following ErrorSets and Errors are not valid: {0}.	
EESPort	Download
	Update
eEES_INVALID_PORT_CONFIGURATION (3008)	
Invalid port configuration	
EESPort	Update
eEES_TIMEOUT_OCCURRED (3005)	
A timeout occurred.	
eEES_UPDATE_FAILED (3007)	
Operation failed because the EES hardware is not accessible.	
EESPort	Update

Error (Errorcode)	
Error-Description	
Class	Method
eFW_ARGUMENT_NULL (13001)	
The argument {0} must not be null.	
FWSignalSymbol	setSignal
FWSymbolFactory	CreateSignalSymbol
	CreateSignalSymbolBySignalDescription
eFW_CREATE_FRAMEWORK_FAILED (13000)	
Error during loading of the requested Framework library occurred.	
FrameworkFactory	CreateVendorSpecificFramework
FrameworkFactory	CreateVendorSpecificFramework2
eFW_UNAVAILABLE_PROPERTY (13002)	
The property is not available in this state.	
MAPortDef	getForceConfig
EESPortDef	getErrorConfigurationFile
ECUCPortDef	getLoadingType
eMA_COULD_NOT_CREATE_SIGNALGENERATOR (8064)	
Could not create an instance of class SignalGenerator.	
MAPort	CreateSignalGenerator
eMA_COULD_NOT_CREATE_TARGETSCRIPT (8072)	
Could not create an instance of class TargetScript.	
MAPort	CreateTargetScript
eMA_COULD_NOT_GET_AVAILABLE_TASKS (8067)	
Could not get the list of available tasks.	
MAPort	getTaskNames
eMA_COULD_NOT_GET_AVAILABLE_VARIABLES (8068)	
Could not get the list of available variables	
eMA_COULD_NOT_GET_DATA_TYPE (8065)	
Could not get the data type of the variable '{0}'.	
MAPort	GetDataType
eMA_COULD_NOT_READ_VARIABLE (8066)	
Could not read the variable '{0}'.	
MAPort	Read
	Read2
	ReadSimultaneously

Error (Errorcode)	
Error-Description	
Class	Method
	ReadSimultaneously2
eMA_COULD_NOT_WRITE_VARIABLE (8069)	
Could not write the variable '{0}'.	
MAPort	Write
	Write2
	WriteSimultaneously
	WriteSimultaneously2
eMA_DOWNLOAD_PARAMETER_SETS_FAILED (8072)	
Download of parameter set files failed. Reason: {0}.	
MAPort	DownloadParameterSets
eMA_PAUSE_SIMULATION_FAILED (8074)	
MAPort	PauseSimulation
eMA_SIMULTANEOUS_READ_WRITE_VAR_COUNT_LIMIT_EXCEEDED (8073)	
Number of variables to be simultaneously read or written exceeds system limit.	
MAPort	ReadSimultaneously
	ReadSimultaneously2
	WriteSimultaneously
	WriteSimultaneously2
eMA_START_SIMULATION_FAILED (8070)	
Failed to start the simulation.	
MAPort	StartSimulation
eMA_STOP_SIMULATION_FAILED (8071)	
Failed to stop the simulation.	
MAPort	StopSimulation
eMAPPING_INVALID (11000)	
Mapping error occurred, reason: {0}	
ConversionTable	GetFrameworkStringByTestbenchString
	GetTestbenchStringByFrameworkString
FrameworkMapping	GetConversionTableByName
	GetFrameworkMappingInfoByLabelName
	GetStringMappingInfosByPortInstanceName
	GetStringMappingValue
	GetTestbenchMappingInfosByPortInstanceName
	GetUnitByName

Error (Errorcode)	
Error-Description	
Class	Method
FrameworkMappingInfo	getFcnAxisUnit
	getXAxisUnit
	getYAxisUnit
FrameworkMetaData	getIsReadable
	getIsWriteable
TestbenchMappingInfo	getFcnAxisMetaData
	getFcnAxisUnit
	getXAxisMetaData
	getXAxisType
	getXAxisUnit
	getYAxisMetaData
	getYAxisType
	getYAxisUnit
XMLMetaData	getIsReadable
	getIsWriteable
	getMax
	getMin
eMEASURING_ACQUISITION_NOT_CONFIGURED (9000)	
To start the Acquisition, it must have been configured via the method SetConfigurations.	
Acquisition	Start
eMEASURING_CONFIG_NAME_NOT_FOUND (9001)	
A configuration named '{0}' could not be found.	
RecorderResult	GetVariablesByAcquisitionConfigurationName
eMEASURING_IDENTIFIER_NOT_FOUND (9002)	
A variable with the abstract identifier '{0}' could not be found.	
RecorderResult	GetSignalByAbstractIdentifier
	GetVariableByAbstractIdentifier
eMEASURING_INVALID_ACQUISITIONSTATE (9015)	
The Acquisition is in wrong state.	
Acquisition	Start
	Stop
	setConfigurations
	setResynchronization
	ClearConfiguration
eMEASURING_INVALID_CONFIG_NAME (9004)	

Error (Errorcode)	
Error-Description	
Class	Method
The name of a configuration must not be an empty string. The name of a configuration must be unique among all configurations in the Acquisition.	
AcquisitionConfiguration	setName
Acquisition	setConfigurations
eMEASURING_INVALID_CONFIGURATIONS (9003)	
The specified list of configurations must not be null or empty.	
Acquisition	setConfigurations
eMEASURING_INVALID_DELAY (9014)	
Stop delay must be a non-negative number.	
AcquisitionConfiguration	setStopDelay
eMEASURING_INVALID_DOWNSAMPLING (9005)	
The downsampling must be a positive integer.	
AcquisitionConfiguration	setDownsampling
AcquisitionRecorderConfiguration	setDownsampling
RecorderConfiguration	setDownsampling
eMEASURING_INVALID_FILE_NAME (9006)	
The file name must not be empty or null.	
RecorderResultMemoryWriter	setFileName
RecorderResultReader	setFileName
RecorderResultWriter	setFileName
eMEASURING_INVALID_OPERATION (9017)	
The operation is not valid.	
FWConditionWatcher	setCondition
eMEASURING_INVALID_OPERATOR (9018)	
The operator is not valid.	
FWConditionWatcher	setCondition
eMEASURING_INVALID_RETRIGGERING (9007)	
Retriggering must be a non-negative integer or -1.	
AcquisitionConfiguration	setRetriggering
RecorderConfiguration	setRetriggering
eMEASURING_LOAD_FAILED (9008)	
Could not load the RecorderResult from file "{0}". Reason: {1}	
RecorderResultReader	Load
eMEASURING_MODIFY_CONFIGURATION_FAILED (9016)	
Used Framework Variable inside condition is not part of acquisition	

Error (Errorcode)	
Error-Description	
Class	Method
FWConditionWatcher	setCondition
eMEASURING_NO_VARIABLES_SPECIFIED (9009)	
The list of variable objects must not be empty.	
AcquisitionConfiguration	setVariables
AcquisitionRecorderConfiguration	setVariables
RecorderConfiguration	setVariables
eMEASURING_PORT_NOT_AVAILABLE (9010)	
One or more of the ports configured for Acquisition is not available.	
Acquisition	Start
eMEASURING_RECORDER_NOT_CONFIGURED (9011)	
To start the Recorder, it must have been configured via a ResultRecorderConfiguration.	
Recorder	Start
eMEASURING_SAVE_FAILED (9012)	
Could not save the RecorderResult to file "{0}". Reason: {1}	
RecorderResult	Save
eMEASURING_UNKNOWN_TASK_NAME (9013)	
The specified abstract task identifier '{0}' is unknown.	
AcquisitionConfiguration	setTaskName
eNW_CAN_NOT MODIFY_RUNNING_CAPTURE (7003)	
The configuration of a running capture can not be modified.	
FrameCapture	SetFilterExpression
	SetFrames
SignalCapture	SetSignals
eNW_CAN_NOT_MODIFY_STARTED_SEQUENCES (7011)	
Running sequences can not be modified.	
Sender	AddNewSequence
	ClearSequences
eNW_DUPLICATE_FRAME_ID (7010)	
A frame with the provided id {0} already exists.	
ChannelCan	AddNewFrame
eNW_DUPLICATE_NAME (7008)	
An item with the provided name {0} already exists.	
ChannelCan	AddNewFrame
Frame	AddNewPdu

Error (Errorcode)	
Error-Description	
Class	Method
MultiplexorSignal	AddSignal
ParitySignal	AddSignal
Pdu	AddNewSignal
eNW_FRAME_FILTEREXPRESSION_SYNTAX_ERROR (7004)	
The provided frame filter expression contains one or more syntax errors.	
FrameCapture	SetFilterExpression
eNW_UNKNOWN_FRAME_ID (7000)	
A frame with the provided frame id {0} does not exist.	
Channel	GetFrameById
Cluster	GetFrameByIdAndChannelName
NetworkPort	GetFrameByIdAndPathName
eNW_UNKNOWN_ITEM (7006)	
The provided item does not exist.	
ChannelCan	DeleteFrame
Frame	DeletePdu
MultiplexorSignal	RemoveSignal
ParitySignal	RemoveSignal
Pdu	DeleteSignal
eNW_UNKNOWN_NAME (7002)	
An item with the provided name {0} does not exist.	
Channel	GetFrameByFrameName
Cluster	GetChannel
	GetFrameByIdAndChannelName
Frame	GetPdu
NetworkPort	GetCluster
Pdu	GetSignal
eNW_UNKNOWN_PATH (7001)	
An item with the provided path {0} does not exist.	
Channel	GetPdu
	GetSignal
Cluster	GetFrameByPathName
	GetPdu
	GetSignal
Frame	GetSignal
NetworkPort	GetChannel

Error (Errorcode)	
Error-Description	
Class	Method
	GetFrameByIdAndPathName
	GetFrameByPathName
	GetPdu
	GetSignal
	Read
	Write
eSTIMULUS_ARGUMENT_NULL (15002)	
The argument {0} must not be null.	
FWLoopSegment	Add
	Insert
FWSegmentSignalDescription	Add
	Insert
FWSignalDescriptionSet	Insert
	Load
	Save
FWSignalDescriptionSetReader	Load
FWSignalDescriptionSetWriter	Save
FWSignalFactory	CreateSignalDescriptionSetByReader
eSTIMULUS_ARGUMENT_OUT_OF_RANGE (15000)	
The argument {0} is out of range. Valid values are: {1}.	
FWExpSegment	setTau
FWNoiseSegment	setSeed
FWPulseSegment	setDutyCycle
	setPeriod
	setPhase
FWSawSegment	setDutyCycle
	setPeriod
	setPhase
FWSignalFactory	CreateConstSegmentBySymbols
	CreateExpSegmentBySymbols
	CreateIdleSegmentByDuration
	CreateNoiseSegmentBySymbols
	CreatePulseSegmentBySymbols
	CreateRampSegmentBySymbols
	CreateRampSlopeSegmentBySymbols

Error (Errorcode)	
Error-Description	
Class	Method
	CreateSawSegmentBySymbols
	CreateSineSegmentBySymbols
FWSineSegment	setPeriod
	setPhase
eSTIMULUS_FILE_NOT_FOUND (15001)	
A file with the name {0} is not found.	
FWSignalDescriptionSet	Load
eSTIMULUS_INDEX_OUT_OF_RANGE (15003)	
The element with index {0} is not accessible. Index range is: 0 .. {1} .	
FWLoopSegment	GetByIndex
	Insert
	RemoveByIndex
FWSegmentSignalDescription	GetByIndex
	Insert
	RemoveByIndex
FWSignalDescriptionSet	GetByIndex
	Insert
	RemoveByIndex
eSTIMULUS_INVALID_DURATION (15004)	
Invalid duration value. Duration has to be a positive floating point number.	
FWConstSegment	setDuration
FWExpSegment	setDuration
FWIdleSegment	setDuration
FWNoiseSegment	setDuration
FWPulseSegment	setDuration
FWRampSegment	setDuration
FWRampSlopeSegment	setDuration
FWSawSegment	setDuration
FWSineSegment	setDuration
eSTIMULUS_LOAD_FAILED (15005)	
An error occurred in loading the signal generator. Reason: {0}.	
FWSignalDescriptionSet	Load
eSTIMULUS_NAME_NOT_FOUND (15006)	
Name {0} not found.	
FWSignalDescriptionSet	GetByName

Error (Errorcode)	
Error-Description	
Class	Method
	RemoveByName
eSTIMULUS_SAVE_FAILED (15007)	
An error occurred in saving the signal generator. Reason: {0}.	
FWSignalDescriptionSet	Save
eVARIABLES_ABSOLUTE_STATES (10006)	
The Operation is not allowed with different isAbsolute states.	
FloatBoolCurveVariable	Read
	SetXUnit
	Write
FloatFloatBoolMapVariable	Read
	SetXUnit
	SetYUnit
	Write
FloatFloatCurveVariable	Read
	SetFcnUnit
	SetXUnit
	Write
FloatFloatFloatMapVariable	SetFcnUnit
	SetXUnit
	SetYUnit
	Write
FloatFloatIntMapVariable	Read
	SetFcnUnit
	SetXUnit
	SetYUnit
	Write
FloatFloatStringMapVariable	Read
	SetXUnit
	SetYUnit
	Write
FloatIntCurveVariable	Read
	SetFcnUnit
	SetXUnit
	Write
FloatMatrixVariable	Read

Error (Errorcode)	
Error-Description	
Class	Method
	SetUnit
FloatQuantity	ConvertToUnit
	Add
FloatStringBoolMapVariable	Read
	SetXUnit
	Write
FloatStringCurveVariable	Read
	SetXUnit
	Write
FloatStringFloatMapVariable	Read
	SetFcnUnit
	SetXUnit
	Write
FloatStringIntMapVariable	Read
	SetFcnUnit
	SetXUnit
	Write
FloatStringStringMapVariable	Read
	SetXUnit
	Write
FloatVariable	Read
	SetUnit
	Write
FloatVectorVariable	Read
	SetUnit
	Write
IntMatrixVariable	Read
	SetUnit
	Write
IntQuantity	ConvertToUnit
	Add
IntVariable	SetUnit
	Write
IntVectorVariable	Read
	SetUnit
	Write

Error (Errorcode)	
Error-Description	
Class	Method
StringFloatBoolMapVariable	Read
	SetYUnit
	Write
StringFloatCurveVariable	Read
	SetFcnUnit
	Write
StringFloatFloatMapVariable	Read
	SetFcnUnit
	SetYUnit
	Write
StringFloatIntMapVariable	Read
	SetFcnUnit
	SetYUnit
	Write
StringFloatStringMapVariable	Read
	SetYUnit
	Write
StringIntCurveVariable	Read
	SetFcnUnit
	Write
StringStringFloatMapVariable	Read
	SetFcnUnit
	Write
StringStringIntMapVariable	Read
	SetFcnUnit
	Write
UIntMatrixVariable	Read
	SetUnit
	Write
UIntQuantity	ConvertToUnit
	Add
UIntVariable	Read
	SetUnit
	Write
UIntVectorVariable	Read
	SetUnit

Error (Errorcode)	
Error-Description	
Class	Method
	Write
eVARIABLES_AXIS_LENGTH_MISMATCH (10004)	
Axis length have not the same size.	
BoolMatrixVariable	Write
BoolVectorVariable	Write
FloatBoolCurveVariable	Write
FloatFloatBoolMapVariable	Write
FloatFloatCurveVariable	Write
FloatFloatFloatMapVariable	Write
FloatFloatIntMapVariable	Write
FloatFloatStringMapVariable	Write
FloatIntCurveVariable	Write
FloatMatrixVariable	Write
FloatStringBoolMapVariable	Write
FloatStringCurveVariable	Write
FloatStringFloatMapVariable	Write
FloatStringIntMapVariable	Write
FloatStringStringMapVariable	Write
FloatVectorVariable	Write
IntMatrixVariable	Write
IntVectorVariable	Write
StringBoolCurveVariable	Write
StringFloatBoolMapVariable	Write
StringFloatCurveVariable	Write
StringFloatFloatMapVariable	Write
StringFloatIntMapVariable	Write
StringFloatStringMapVariable	Write
StringIntCurveVariable	Write
StringMatrixVariable	Write
StringStringBoolMapVariable	Write
StringStringCurveVariable	Write
StringStringFloatMapVariable	Write
StringStringIntMapVariable	Write
StringStringStringMapVariable	Write
StringVectorVariable	Write

Error (Errorcode)	
Error-Description	
Class	Method
UIntMatrixVariable	Write
UIntVectorVariable	Write
eVARIABLES_DIVISION_BY_ZERO (10007)	
Division by zero is not allowed.	
IntQuantity	Divide
UIntQuantity	Divide
eVARIABLES_FORMULA_UNKNOWN (10000)	
The required formula is not defined in tableau.	
FloatQuantity	Divide
	Multiply
IntQuantity	Divide
	Multiply
UIntQuantity	Divide
	Multiply
eVARIABLES_INCOMPATIBLE_PHYSICAL_DIMENSION (10003)	
Procedure not allowed with different Physical Dimensions.	
FloatBoolCurveVariable	SetXUnit
FloatFloatBoolMapVariable	SetXUnit
	SetYUnit
FloatFloatCurveVariable	SetFcnUnit
	SetXUnit
FloatFloatFloatMapVariable	SetFcnUnit
	SetXUnit
	SetYUnit
FloatFloatIntMapVariable	SetFcnUnit
	SetXUnit
	SetYUnit
FloatFloatStringMapVariable	SetXUnit
	SetYUnit
FloatIntCurveVariable	SetFcnUnit
	SetXUnit
FloatMatrixVariable	SetUnit
FloatQuantity	ConvertToUnit
FloatStringBoolMapVariable	SetXUnit
FloatStringCurveVariable	SetXUnit

Error (Errorcode)	
Error-Description	
Class	Method
FloatStringFloatMapVariable	SetFcnUnit
	SetXUnit
FloatStringIntMapVariable	SetFcnUnit
	SetXUnit
FloatStringStringMapVariable	SetXUnit
FloatVariable	SetUnit
FloatVectorVariable	SetUnit
IntMatrixVariable	SetUnit
IntQuantity	ConvertToUnit
IntVariable	SetUnit
IntVectorVariable	SetUnit
StringFloatBoolMapVariable	SetYUnit
StringFloatCurveVariable	SetFcnUnit
StringFloatFloatMapVariable	SetFcnUnit
	SetYUnit
StringFloatIntMapVariable	SetFcnUnit
	SetYUnit
StringFloatStringMapVariable	SetYUnit
StringIntCurveVariable	SetFcnUnit
StringStringFloatMapVariable	SetFcnUnit
StringStringIntMapVariable	SetFcnUnit
UIntMatrixVariable	SetUnit
UIntQuantity	ConvertToUnit
UIntVariable	SetUnit
UIntVectorVariable	SetUnit
eVARIABLES_NO_UNIT_OPERATION (10010)	
This operation is not allowed, cause the quantity is based on Unit "NO_UNIT".	
FloatQuantity	Add
	Subtract
IntQuantity	Add
	Subtract
UIntQuantity	Add
	Subtract
eVARIABLES_NOT_A_NUMBER_RESULT (10002)	
The result of calculation is not a valid value e.g. 0.0D / 0.0D.	

Error (Errorcode)	
Error-Description	
Class	Method
FloatQuantity	ConvertToUnit
	Divide
	Multiply
IntQuantity	ConvertToUnit
UIntQuantity	ConvertToUnit
eVARIABLES_NOT_READABLE (10008)	
The Framework variable {0} is not readable.	
BoolMatrixVariable	Read
BoolVariable	Read
BoolVectorVariable	Read
FloatBoolCurveVariable	Read
FloatFloatBoolMapVariable	Read
FloatFloatCurveVariable	Read
FloatFloatFloatMapVariable	Read
FloatFloatIntMapVariable	Read
FloatFloatStringMapVariable	Read
FloatIntCurveVariable	Read
FloatMatrixVariable	Read
FloatStringBoolMapVariable	Read
FloatStringCurveVariable	Read
FloatStringFloatMapVariable	Read
FloatStringIntMapVariable	Read
FloatStringStringMapVariable	Read
FloatVariable	Read
FloatVectorVariable	Read
IntMatrixVariable	Read
IntVariable	Read
IntVectorVariable	Read
StringBoolCurveVariable	Read
StringFloatBoolMapVariable	Read
StringFloatCurveVariable	Read
StringFloatFloatMapVariable	Read
StringFloatIntMapVariable	Read
StringFloatStringMapVariable	Read
StringIntCurveVariable	Read

Error (Errorcode)	
Error-Description	
Class	Method
StringMatrixVariable	Read
StringStringBoolMapVariable	Read
StringStringCurveVariable	Read
StringStringFloatMapVariable	Read
StringStringIntMapVariable	Read
StringStringStringMapVariable	Read
StringVariable	Read
StringVectorVariable	Read
UIntMatrixVariable	Read
UIntVariable	Read
UIntVectorVariable	Read
eVARIABLES_NOT_WRITEABLE (10009)	
The Framework variable {0} is not writeable.	
BoolMatrixVariable	Write
BoolVariable	Write
BoolVectorVariable	Write
FloatBoolCurveVariable	Write
FloatFloatBoolMapVariable	Write
FloatFloatCurveVariable	Write
FloatFloatFloatMapVariable	Write
FloatFloatIntMapVariable	Write
FloatFloatStringMapVariable	Write
FloatIntCurveVariable	Write
FloatMatrixVariable	Write
FloatStringBoolMapVariable	Write
FloatStringCurveVariable	Write
FloatStringFloatMapVariable	Write
FloatStringIntMapVariable	Write
FloatStringStringMapVariable	Write
FloatVariable	Write
FloatVectorVariable	Write
IntMatrixVariable	Write
IntVariable	Write
IntVectorVariable	Write
StringBoolCurveVariable	Write

Error (Errorcode)	
Error-Description	
Class	Method
StringFloatBoolMapVariable	Write
StringFloatCurveVariable	Write
StringFloatFloatMapVariable	Write
StringFloatIntMapVariable	Write
StringFloatStringMapVariable	Write
StringIntCurveVariable	Write
StringMatrixVariable	Write
StringStringBoolMapVariable	Write
StringStringCurveVariable	Write
StringStringFloatMapVariable	Write
StringStringIntMapVariable	Write
StringStringStringMapVariable	Write
StringVariable	Write
StringVectorVariable	Write
UIntMatrixVariable	Write
UIntVariable	Write
UIntVectorVariable	Write
eVARIABLES_OUT_OF_RANGE (10001)	
The result of operation exceeds the range Min: {0}, Max: {1}.	
IntQuantity	Add
	ConvertToUnit
	Multiply
	Subtract
UIntQuantity	Add
	ConvertToUnit
	Multiply
	Subtract

Appendix F. DEPRECATED ELEMENTS

The API elements listed in the second column of the following table are deprecated and might be removed in a future version of the standard. So it is recommended to use the replacement depicted in the third column.

Table 64 Deprecated elements and their replacements

Affected Component	Deprecated Element	Replacement
Attributes (Package Testbench.Common.Value Container)	Complete interface	Interface VariableInfo and its Port specific derivations (e.g. MAPortVariableInfo from package Testbench.MAPort) for predefined Attributes “Name”, “Description” and “Unit”. No replacement for user defined Attributes.
BaseValue (Package Testbench.Common.Value Container)	Property Attributes	Method GetVariableInfo of the Port interfaces for predefined Attributes “Name”, “Description” and “Unit”. No replacement for user defined Attributes.
	Property Type	Properties ContainerType and ElementType
Capture (Package Testbench.Common.Capturing)	Property DurationUnit	Implicitly set by passing a TimeSpanDuration or CycleNumberDuration object (from package Testbench.Common.Duration) to SetStartTrigger, SetStopTrigger and the DurationWatcher factory methods.
	Method SetStartTriggerCondition	Method SetStartTrigger
	Method SetStopTriggerCondition	Method SetStopTrigger
	Property Variables	Property Variables2
	Method ExtractSignalValue	Methods GetTimestamps , GetSignalValues and GetScalarSignalValues of the CaptureSignalGroup interface obtainable via the SignalGroups property
CaptureResult (Package Testbench.Common.CaptureResult)	Method GetSignalGroupValue	Property SignalGroups
	Method GetVariableNames	Property Name of the VariableInfo interface obtainable via method GetVariableInfos
	Property SignalGroupNames	Property Name of the CaptureSignalGroup objects obtainable via the SignalGroup

		property
ConditionWatcher (Package Testbench.Common.Watch erHandling)	Property Defines	Property Defines2
	Property setter setCondition	The Condition property becomes read-only. Create a new ConditionWatcher to change the condition expression.
ConstSymbol (Testbench.Common.Symb ol)	Property Value	Property Expression
DataType (Package Testbench.Common.Value Container.Enum)	Enumeration DataType	Enumerations ContainerDataType and PrimitiveDataType in package Testbench.Common.ValueContain er.Enum
DurationUnit (Testbench.Common.Captu ring.Enum)	Enumeration DurationUnit	Implicit representation by TimeSpanDuration and CycleNumberDuration in package Testbench.Common.Duration
DurationWatcher (Package Testbench.Common.Watch erHandling)	Property Duration	Property Duration2 . (This is a read-only property. Create a new DurationWatcher to change the duration value.)
ECUCPort (Package Testbench.ECUCPort)	Method CheckVariableNames	Method CheckVariableRefs
	Method GetDataType	Properties ContainerType and ElementType of the DataTypeInfo interface obtainable via method GetVariableInfo and then property DataTypeInfo
	Method IsReadable	Method IsReadable of the ECUCPortVariableInfo interface obtainable via method GetVariableInfo
	Method IsWriteable	Method IsWriteable of the ECUCPortVariableInfo interface obtainable via method GetVariableInfo
	Method Read	Method Read2
ECUMPort (Package Testbench.ECUMPort)	Method Write	Method Write2
	Method CheckVariableNames	Method CheckVariableRefs
	Method GetDataType	Properties ContainerType and ElementType of the DataTypeInfo interface obtainable via method GetVariableInfo and then property DataTypeInfo
	Method GetMeasuringVariables	Property MeasuringVariables2
	Method IsReadable	Method IsReadable of the ECUMPortVariableInfo interface obtainable via method GetVariableInfo

	Method Read	Method Read2
	Method SetMeasuringVariables	Property MeasuringVariables2
	Property TaskNames	Property Name of interface TaskInfo obtainable via property TaskInfos
EESPort (Package Testbench.EESPort)	Method WaitForTrigger	Method WaitForTrigger2
FWSymbolFactory (Package Framework.Common.Symbol)	Method CreateScalarVariableSymbolByValue	Method CreateScalarVariableSymbolByValue2
MAPort (Package Testbench.MAPort)	Method CheckVariableNames	Method CheckVariableRefs
	Method GetDataType	Properties ContainerType and ElementType of the DataTypeInfo interface obtainable via method GetVariableInfo and then property DataTypeInfo
	Method IsReadable	Method IsReadable of the MAPortVariableInfo interface obtainable via method GetVariableInfo
	Method IsWritable	Method IsWriteable of the MAPortVariableInfo interface obtainable via method GetVariableInfo
	Method Read	Method Read2
	Method ReadSimultaneously	Method ReadSimultaneously2
	Property TaskNames	Property Name of interface TaskInfo obtainable via property TaskInfos
	Method Write	Method Write2
	Method WriteSimultaneously2	Method WriteSimultaneously2
MAPortFactory	Method CreateMAPortBreakpoint	Method CreateMAPortBreakpoint2
Player (Package Framework.Stimulation)	Property State	Property State2
RecorderResultMemoryWriter (Package Framework.Measuring)	Complete interface	Interface RecorderResultWriter is sufficient.
ScriptParameterInfo (Package Testbench.Common.Script)	Property DataType	Properties ContainerType and ElementType of the DataTypeInfo interface obtainable via property DataTypeInfo
	Property XSize	Method GetSize of the DataTypeInfo interface obtainable via property DataTypeInfo

	Property YSize	Method GetSize of the DataTypeInfo interface obtainable via method GetDataTypeInfo
SignalGenerator (Package Testbench.Common.Signal Generator)	Property Assignments	Property Assignments2 and property AliasDefinitions
	Property State	Property ScriptState inherited from the base interface Script in package Testbench.Common (SignalGenerator is derived from Script)
SignalGeneratorState (Package Testbench.Common.Signal Generator.Enum)	Enumeration SignalGeneratorState	Enumeration ScriptState in package Testbench.Common.Script.Enum
SignalGroupValue (Package Testbench.Common.Value Container)	Method ExtractSignalValue	Method ExtractSignalValueByIndex
	Property VariableNames	No replacement. Use the y vectors' index numbers instead of names.
	Property setter setXVector	The property becomes read-only. Create a new SignalGroupValue to have a different XVector.
	Property setter setYVectors	The property becomes read-only. Create a new SignalGroupValue to have different YVectors.
SymbolFactory (Testbench.Common.Symb ol)	Method CreateConstSymbolByValue	Method CreateConstSymbolByExpressio n
ValueFactory (Package Testbench.Common.Value Container)	Method CreateAttributes	No replacement, because the Attributes interface itself is deprecated.
	Method CreateXYValue	No replacement, because the XYValue interface itself is deprecated.
VariableInfo (Package Testbench.Common.MetalIn fo)	Property DataType	Properties ContainerType and ElementType of the DataTypeInfo interface obtainable via method GetDataTypeInfo
	Property Readable	Method IsReadable of the derived, port specific VariableInfo interfaces (e.g. MAPortVariableInfo from package Testbench.MAPort)
	Property Writable	Method IsWriteable of the derived, port specific VariableInfo interfaces (e.g. MAPortVariableInfo from package Testbench.MAPort)
	Property XSize	Method GetSize of the DataTypeInfo interface obtainable via method GetDataTypeInfo
	Property YSize	Method GetSize of the DataTypeInfo interface obtainable via method GetDataTypeInfo

WatcherFactory (Package Testbench.Common.Watch erHandling)	Method CreateConditionWatcher	Method CreateConditionWatcher2
	Method CreateDurationWatcher	Methods CreateDurationWatcherByTimeS pan und CreateDurationWatcherByCycle Number
XYValue (Package Testbench.Common.Value Container)	Complete interface	No replacement

Figure Directory

Figure 1:	Principle of Hardware-in-the-Loop Simulation	11
Figure 2:	XIL Testbench API with direct port access	16
Figure 3:	Test system with XIL Framework API and XIL Testbench API	17
Figure 4:	The Framework class	18
Figure 5:	The FrameworkVariable class	19
Figure 6:	The Acquisition class	20
Figure 7:	The Stimulation class	21
Figure 8:	XIL Ports	22
Figure 9:	Implementation Manifest files contain a list of elements referring to C# or Python classes that implement the Testbench or Framework interface	24
Figure 10:	NetTestbenchImplementation element of the Implementation Manifest	25
Figure 11:	NetFrameworkImplementation element of the Implementation Manifest	26
Figure 12:	PyTestbenchImplementation element of the Implementation Manifest	27
Figure 13:	PyFrameworkImplementation element of the Implementation Manifest	28
Figure 14:	TestbenchFactory class	29
Figure 15:	FrameworkFactory class	29
Figure 16:	Framework configuration files	31
Figure 17:	Top level elements of framework configuration file	32
Figure 18:	Structure of mapping file references in the framework configuration file	32
Figure 19:	Specific port definition types in the framework configuration file	32
Figure 20:	Structure of MAPortDefinition elements in the framework configuration file	33
Figure 21:	Structure of the TargetState element of the MAPortDefinition	33
Figure 22:	Framework configuration process example (* for descriptive purpose only, not part of the interface)	35
Figure 23:	Example for port creation and setup process by the framework (* for descriptive purpose only, not part of the interface)	36
Figure 24:	Framework shutdown process example (* for descriptive purpose only, not part of the interface)	37
Figure 25:	Example for customized Framework configuration process (* for descriptive purpose only, not part of the interface)	39
Figure 26:	Framework configuration object model	40
Figure 27:	Port definition interfaces of the framework configuration object model	41
Figure 28:	Top level entities of the mapping XSD	43
Figure 29:	XSD structure of a framework label	45
Figure 30:	XSD structure of a testbench label	46
Figure 31:	XSD structure of the testbench metadata	47
Figure 32:	XSD structure of the label mapping	49
Figure 33:	XSD structure of the FromCurve mapping selector	49
Figure 34:	XSD structure of the string mapping	50
Figure 35:	XSD structure of the raster mapping	51
Figure 36:	XSD structure of the conversion tables	52
Figure 37:	XSD structure of the units	53
Figure 38:	XSD structure of the units	54
Figure 39:	Framework Mapping Info API	56
Figure 40:	Access via MappingInfo API	59
Figure 41:	Data access using Framework variables	62
Figure 42:	Main framework variable types	64

Figure 43:	Main framework variable classes	65
Figure 44:	Scalar variable classes	66
Figure 45:	Vector variable classes	67
Figure 46:	Matrix variable classes	68
Figure 47:	Curve variable classes	70
Figure 48:	Map variable classes	72
Figure 49:	Scalar Quantity classes	75
Figure 50:	Complex Curve Quantity classes	77
Figure 51:	Complex Map Quantity classes	78
Figure 52:	QuantityFactory class	79
Figure 53:	Using quantities with Framework variables	81
Figure 54:	The Unit and PhysicalDimension classes	84
Figure 55:	The MetaData classes	85
Figure 56:	Acquisition and AcquisitionConfiguration	94
Figure 57:	Framework Watcher	99
Figure 58:	Example of different AcquisitionConfiguration Objects	101
Figure 59:	Acquisition with longer interval	102
Figure 60:	Acquisition data flow	102
Figure 61:	Effects of offsets and drifting timers (without resynchronization)	103
Figure 62:	Effects of offsets and drifting timers corrected with resynchronization	104
Figure 63:	AcquisitionRecorder and Recorder	105
Figure 64:	AcquisitionRecorder and its Configuration	106
Figure 65:	AcquisitionRecorder and RecorderConfiguration	107
Figure 66:	RecorderResultReader and RecorderResultWriter	108
Figure 67:	AcquisitionRecorder and Access of the Measured Data	110
Figure 68:	Sequence of Acquisition and Recorder Commands	113
Figure 69:	Player and relations to SignalDescriptionSet and Assignments	118
Figure 70:	General Value classes	122
Figure 71:	Element type specific sub classes of ScalarValue and VectorValue	123
Figure 72:	Specific Value classes	124
Figure 73:	DocumentHandling in XIL	125
Figure 74:	Script and relations to TargetScript and SignalGenerator	126
Figure 75:	States and state transitions of Script objects	127
Figure 76:	ScriptParameterInfo interface	131
Figure 77:	TargetScript interface	132
Figure 78:	Example for executing a TargetScript	133
Figure 79:	SignalDescriptions and SignalGenerator	134
Figure 80:	Modulate Signal Parameter by further Signals	135
Figure 81:	SignalDescriptions and SignalGenerator	135
Figure 82:	SignalDescriptions and SignalGenerator (data transformation)	136
Figure 83:	SignalDescription relations	137
Figure 84:	Symbol	138
Figure 85:	ConstSegment	140
Figure 86:	RampSegment	141
Figure 87:	IdleSegment	143
Figure 88:	NoiseSegment	145
Figure 89:	RampSlopeSegment	146
Figure 90:	SineSegment	148
Figure 91:	SawSegment	150
Figure 92:	PulseSegment	152
Figure 93:	ExpSegment	154
Figure 94:	Create Segment Signal Description Example	162
Figure 95:	Create OperationSignal	163
Figure 96:	Create a wobbling signal	164

Figure 97:	Create SignalDescriptionSet	165
Figure 98:	Load a SignalDescriptionSet	166
Figure 99:	Save SignalDescriptionSet	167
Figure 100:	Usage of constants in SegmentSignalDescription	168
Figure 101:	Signal createParameterizedSignalDescription	170
Figure 102:	Signal Generator	171
Figure 103:	Initial stimulation values and zero point of elapsed stimulation time when stimulating two variables with equidistant but different stimulation rasters	172
Figure 104:	Course of stimulation at discontinuous segment transition (with equidistant stimulation raster)	173
Figure 105:	Course of stimulation during IDLESegment in absence of additional manipulations from outside the SignalGenerator and with equidistant stimulation raster used	174
Figure 106:	End of stimulation with final stimulation values and their persistence in absence of other signal manipulations (Equidistant but different stimulation rasters are used. SignalDescriptions have different lengths.)	175
Figure 107:	SignalGenerator example (part 1)	177
Figure 108:	SignalGenerator example (part 2)	178
Figure 109:	Signal Description Document Handling	179
Figure 110:	Signal Generator Document Handling	180
Figure 111:	Testbench Watcher	181
Figure 112:	Duration Interfaces	183
Figure 113:	Interfaces for metadata query on variables	184
Figure 114:	Interfaces for metadata query on conversion methods	187
Figure 115:	Interface for metadata query on acquisition rasters / processor tasks	188
Figure 116:	VariableRef interfaces	189
Figure 117:	Capture interface for capture configuration and control	191
Figure 118:	State diagram for untriggered capturing mode	196
Figure 119:	State diagram for triggered capturing mode without retriggering	198
Figure 120:	State diagram for retriggered capturing mode	200
Figure 121:	CaptureResult and CaptureSignalGroup	203
Figure 122:	CaptureEvent interfaces	206
Figure 123:	States, events and data frame in untriggered capturing mode	207
Figure 124:	Trigger, states, events and data frame in triggered capturing mode without trigger delays	208
Figure 125:	Trigger, states, events and data frame in case of positive start and stop trigger delays	209
Figure 126:	Trigger, states, events and data frame in case of negative start trigger delay	210
Figure 127:	Trigger, states, events and data frames for re-triggered capturing	212
Figure 128:	Re-triggered capturing with overlapping data frames	214
Figure 129:	Reader and Writer interfaces for CaptureResult	215
Figure 130:	Create and configure a Capture object	217
Figure 131:	Perform the capturing process	218
Figure 132:	Create a CaptureResult from an MDF file	219
Figure 133:	Access to scalar value traces in a CaptureResult	220
Figure 134:	Access to vector value traces in a CaptureResult	221
Figure 135:	Retrieve physical and raw values from a CaptureResult	222
Figure 136:	MAPort and associated interfaces	228
Figure 137:	MAPort state diagram	229
Figure 138:	Process of MAPort creation and configuration	232
Figure 139:	Metadata query on simulation variables	234

Figure 140:	CompuMethod query including value set of TexttableCompuMethods	235
Figure 141:	Metadata query on execution tasks in the simulation model	236
Figure 142:	Reading and writing model variables	237
Figure 143:	Reading and writing an individual vector element	238
Figure 144:	Reading and writing a variable's raw value	239
Figure 145:	MAPortVariableInfo methods	240
Figure 146:	Relation between simulation and capturing / stimulation	241
Figure 147:	Example for pausing and stepwise executing a simulation	244
Figure 148:	Accessing ECUs via the XIL API for diagnostic	247
Figure 149:	ECU DiagPort Class diagram	247
Figure 150:	ECU classes	248
Figure 151:	Functional group classes	249
Figure 152:	States of the Diag port	250
Figure 153:	Process of DiagPort creation and configuration	252
Figure 154:	Getting the ECU Object	252
Figure 155:	Reading and clearing the fault memory	253
Figure 156:	Reading the variant coding data	254
Figure 157:	Reading identification data	254
Figure 158:	Reading and writing values from and to the EEPROM by an alias name	255
Figure 159:	Reading from the EEPROM	256
Figure 160:	Writing to the EEPROM	256
Figure 161:	Sending HEX service with explicit communication	257
Figure 162:	Executing a Job	258
Figure 163:	Reading measurement data from a functional group	259
Figure 164:	Using the BaseController	260
Figure 165:	Accessing ECUs via the XIL API for measurement	261
Figure 166:	ECUMPort Classdiagram	264
Figure 167:	States of the ECUM port	265
Figure 168:	Process of ECUMPort creation and configuration	267
Figure 169:	Get lists of variable and task names	267
Figure 170:	Read a scalar variable value and examine its properties	268
Figure 171:	Read an array variable value and examine its properties	269
Figure 172:	Read a matrix variable value and examine its properties	270
Figure 173:	Capturing ECU variables	273
Figure 174:	Accessing ECUs via the XIL API for Calibration	274
Figure 175:	ECUCPort Classdiagram	275
Figure 176:	States of the ECUC port	276
Figure 177:	Process of ECUCPort creation and configuration	278
Figure 178:	Read and write a scalar float ECU parameter	279
Figure 179:	Get the list of parameters of an ECUCPort instance	279
Figure 180:	Handling of memory pages	281
Figure 181:	Electrical Error Simulation is used to disturb the signals between the XIL system and the SUT	282
Figure 182:	Error configurations are defined by EES ports and downloaded for execution to the vendor-specific EES hardware respective software	283
Figure 183:	An error configuration comprises a sequence of error sets with several errors	284
Figure 184:	Example for the execution of an error configuration	285
Figure 185:	Illustration of the error categories defined by EES port	287
Figure 186:	Illustration of the error types defined by EES port	288
Figure 187:	Illustration of the option with load and without load	289
Figure 188:	EES port classes	290
Figure 189:	Classes used to represent an error configuration	291
ASAM XIL Generic Simulator Interface Version 2.2.0		400

Figure 190:	Class hierarchy of error objects	292
Figure 191:	Error Builder	294
Figure 192:	Reader and writer classes for error configuration files	294
Figure 193:	EESPort States	296
Figure 194	Process of EESPort creation and configuration	298
Figure 195:	Error Configuration Part 1	300
Figure 196:	Error Configuration Part 2	301
Figure 197:	Error Stimulation	302
Figure 198:	Extension of Error Stimulation	303
Figure 199:	create a short-circuit to U _{battery} error object	304
Figure 200:	create a short-circuit Ground error object	304
Figure 201:	create a short-circuit potential error object	305
Figure 202:	create a line interruption error object	305
Figure 203:	create a pin-to-pin error object	306
Figure 204:	create a loose contact error object	307
Figure 205:	create a loose contact error object (MultiPin2Pin)	308
Figure 206:	load error configuration from file	308
Figure 207:	Network Port setup	311
Figure 208:	Network Port interface	312
Figure 209:	States of the Network Port	313
Figure 210	Process of NetworkPort creation and configuration	314
Figure 211:	Class diagram of the hierarchical Network Port structure	315
Figure 212:	Network Port configuration	317
Figure 213:	Sequence diagram for Network Port top-down navigation	319
Figure 214:	Frame and Signal buffer synchronization	320
Figure 215:	Class diagram for Frame, FrameValue, Sender and Receiver	321
Figure 216:	Sequence diagram for frame data transmission	322
Figure 217:	Sequence diagram for frame data reception	322
Figure 218:	Sequence diagram for signal data transmission	323
Figure 219:	Sequence diagram for signal data reception	324
Figure 220:	Class diagram for Network Port signal capturing	325
Figure 221:	Sequence diagram for Network Port signal capturing	326
Figure 222:	Frame Capture	327
Figure 223:	Usage of frame capturing	329
Figure 224:	Frame Replay	330
Figure 225:	Usage of frame replay	331
Figure 226:	Different signal types	332
Figure 227:	Counter Signal Class	332
Figure 228:	Add counter signal to model	333
Figure 229:	Storage of gaps in the measurement data in MDF4	348

Table Directory

Table 1	Version Number and Version Properties	15
Table 2	Storage locations for Implementation Manifest files (environment variables are enclosed by % signs)	28
Table 3	TargetState elements and contained parameters depending on the PortDefinition type in the framework configuration file	34
Table 4	Physical Dimension Example	54
Table 5	Consistency rules	60
Table 6	setUnit() Methods of the Curve Framework variables	71
Table 7	setUnit() Methods of the Map Framework variables	73
Table 8	Rules for the quantity conversion methods	76
Table 9	Definition of Units and Physical Dimensions	83
Table 10	Mathematical Operations	85
Table 11	Dimension definitions with common physical identifiers	88
Table 12	Computation table with common physical identifiers	88
Table 13	Dimension definitions with user defined identifiers (example German)	89
Table 14	Computation table with user defined identifiers	90
Table 15	Comparison	90
Table 16	Data Type Conversions on Quantities	90
Table 17:	Acquisition States	96
Table 18	AcquisitionConfiguration	97
Table 19	Attributes of RecorderResultReader and RecorderResultWriter	108
Table 20:	Recorder States	112
Table 21:	Stimulation States	116
Table 22	Namespaces in ASAM.XIL.Interfaces	117
Table 23:	Player states	119
Table 24	Packages of Common part	120
Table 25	Script states	127
Table 26	Script, TargetScript and SignalGenerator methods in script object states	129
Table 27	Parameters ConstSegment	140
Table 28	Parameter RampSegment	142
Table 29	Parameter IdleSegment	143
Table 30	Parameter NoiseSegment	145
Table 31	Parameter RampSlopeSegment	147
Table 32	Parameter SineSegment	149
Table 33	Parameter SawSegment	151
Table 34	Parameter PulseSegment	153
Table 35	Parameter ExpSegment	155
Table 36	Parameter SignalValueSegment	156
Table 37	Start and Duration Parameter Values of of DataFileSegment	157
Table 38	Parameter DataFileSegment	158
Table 39	Parameter LoopSegment	159
Table 40	Parameter OperationSegment	160
Table 41:	Conversion methods supported by metadata interfaces (R: Raw value, P: Physical value)	186
Table 42	Configuration Properties and Methods of Capture	192
Table 43	Capture states description	194
Table 44	State dependencies of Capture methods and properties	195
Table 45	States of the MAPort	229
Table 46	Allowed states for the MAPort methods	230
Table 47	Writeability of FMI variables with respect to their variability and the MAPort state	241



Table 48	States of the DiagPort	250
Table 49	DiagPort states	251
Table 50	States of the ECUMPort	264
Table 51	ECUMPort states	266
Table 52	States of the ECUCPort	275
Table 53	ECUCPort states	277
Table 54	Error Builder	293
Table 55	States of the EESPort	295
Table 56	EESPort states	297
Table 57	Network Port states	314
Table 58	Mapping between Network Port, CANdb and FIBEX	317
Table 59	Operators and Functions supported by ASAM XIL ConditionWatcher	336
Table 60	Operators and Functions supported by ConstSymbol.	341
Table 61	Overview about reserved Key Value Pairs	345
Table 62	EVBLOCK members	349
Table 63	Error Overview	350
Table 64	Deprecated elements and their replacements	392



Association for Standardization of
Automation and Measuring Systems

E-mail: support@asam.net

Web: www.asam.net

© by ASAM e.V., 2020