Association for Standardisation of
Automation and Measuring Systems

ASAM AE XIL-MA

Generic Simulator Interface for Simulation Model Access

Version 1.0.0

Date: 2014-02-24

Associated Standard

Table of Contents

Foreword

The Generic Simulator Interface for Simulation Model Access (XIL-MA) defines simulator control API commands. It is a subset of the Generic Simulator Interface (XIL), which also supports measuring, calibration, and diagnosis of electronic control units (ECU), as well as network access to e.g. CAN buses. Both, XIL and XIL-MA share the model access port and a couple of general concepts. An simulation tool, which implements the XIL-MA API will be fully compliant to the XIL API, too.
ASAM developed XIL API as a standard for the communication between test automation software and hardware-in-the-loop (HIL) testbenches. HIL API enables users to choose products freely according to their requirements, independent of the vendor. It will support testbenches at all stages of the function software development process – MIL[1], SIL[2], HIL[3], etc. After all, XIL API allows engineers to reuse their existing tests and enables a better know-how transfer from one test bench to the other, resulting in reduced training costs for employees as well.
While ASAM started to specify the XIL API standard, a group of vendors for offline simulation tools developed the Functional Mock-up Interface standard (FMI). FMI has been driven by an EU funded project, called MODELISAR. FMI is a tool independent standard to support both model exchange and co-simulation of dynamic models. In order to accomplish coupling of simulation tools with e.g. test tools, such an API has been also on the agenda with the internal name FMI for Applications. ASAM and MODELISAR decided to join this part of activities in order to develop a single standard, resulting in ASAM XIL.
In order to support offline simulation tool vendors, the XIL-MA standard has been generated, which is freely available, as the FMI standards are. The generic simulator interface for Simulation Model Access offers the possibility that tests written in early simulation environments can be directly reused in HIL environments at a later stage, and vice versa. For the user of this specification exist two packages:

- Standard and
- Implementation Support

In the free public available package standard this specification and an associated UML model are included. The UML model describes all classes and methods with their parameters in detail. Main content of the standard is the description of the MA-port, which shall be used for the remote access to the simulation tools. The port contains data capturing and signal description. The specification document is an excerpt of the full XIL API Programmer's Guide. Due to this fact, it may include some textual references to XIL.
ASAM e.V. additionally offers an implementation support package, which simplifies the implementation inside tools and applications. This package includes technology references of the interfaces for the implementation in python and C#. Also included are templates as schema files for the defining of stimulus descriptions. Factories are

[1] Model-in-the-Loop
[2] Software-in-the-Loop
[3] Hardware-in-the-Loop

distributed for the generic instantiation of one or more testbenches from different vendors.

1 Introduction

1.1 OVERVIEW

ASAM XIL was developed to allow to exchange combinations of test automation software and test hardware. The testbench API separates the test hardware from the test software and allows a standardized access to the hardware. By means of the testbench MA-port access to the simulation model is given, e.g. to read and write parameters, or to capture and generate signals.

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.

1.3 WHAT IS HARDWARE IN THE LOOP SIMULATION

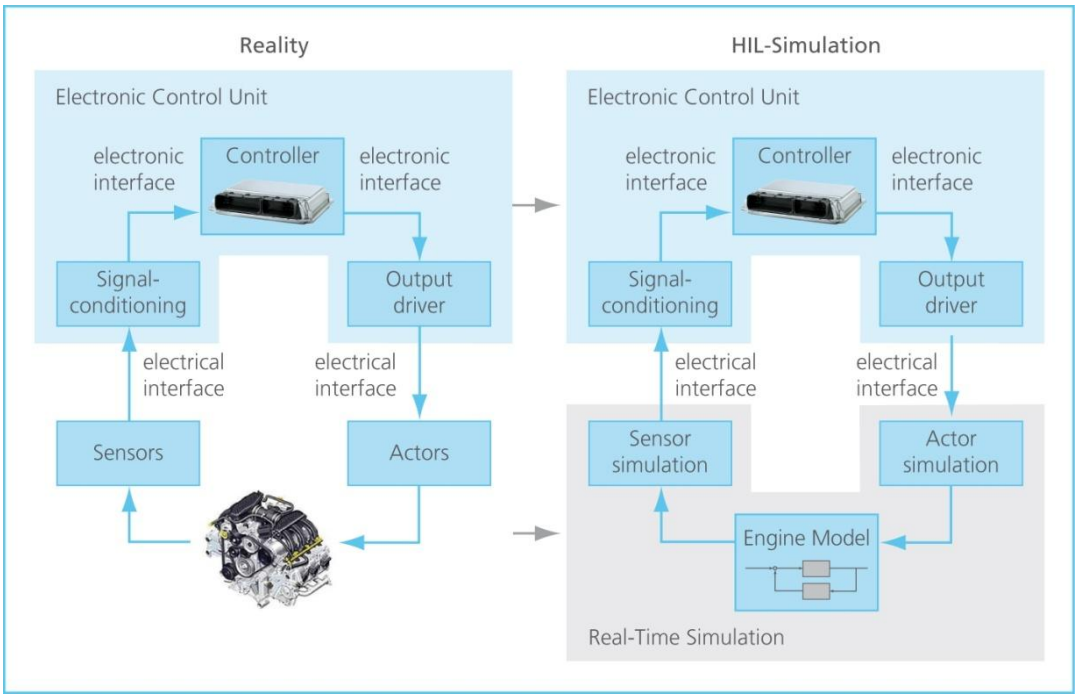


Figure 1: Principle of Hardware-in-the-Loop Simulation

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 factory 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 of provided by vendors in the market and covered by crosstests) 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 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

The specification at hand is the first version of an Generic simulator interface for Simulation Model Access. This specification is an extract of the ASAM AE XIL Version 2.0.0. base standard. [8]

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 in XIL API 2.0 includes numerical data, e. g. within CaptureResult or RecordingResult objects. For streaming this data to file, ASAM XIL 2.0 uses the definition of Measurement Data Format (MDF) [3].

2.3 Versioning

Versioning of the XIL-MA API is done using three numbers: major version, minor version and revision number. The major number is the actual version, the minor number the actual maintenance of the version. The revision number is the revision of the maintenance. These numbers define the interface version of the XIL-MA API.

The version information can be retrieved using the `Testbench` class. The standard implementation of the `Testbench` class returns the following version numbers:

Table 1 Version number

Number	Value
MajorNumber	1
MinorNumber	0
RevisionNumber	0

3 General Concepts

The ASAM XIL-MA standard specifies the Testbench API for usage with MA ports. 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. The shaded areas indicate the ports, which are not part of ASAM XIL-MA.

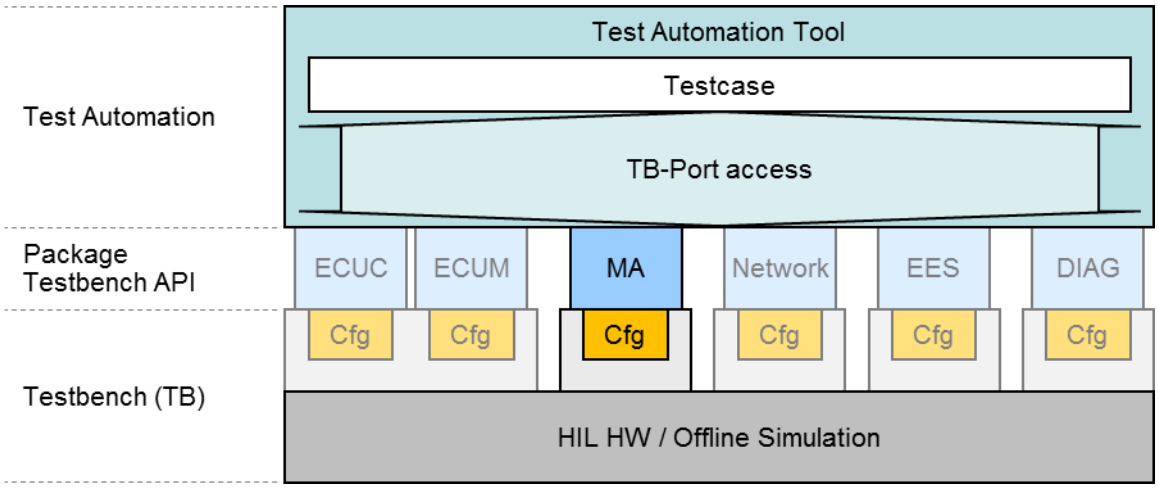


Figure 2: XIL Testbench API with direct port access

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 box. These tools serve different purposes, such as accessing simulation models. 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 block).

The XIL-MA API provides a standardized interface to one or more simulation tools by means of MA Testbench ports, which deal with the simulation model access.

Based on this testbench port interface, a test automation tool could utilize a standardized access to variables and signals on this port without dealing with proprietary details of different vendors and their tools. However, in such a setup the testcase has to implement start and shutdown of the testbench port, 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.

3.2 OVERVIEW TESTBENCH

The XIL Testbench API covers the simulation model access (with their hardware and software components), which is represented by a port (`MAPort`). It is possible to read and to write data and to capture and to generate signals.The port initialization is supported by means of standardized configuration methods and objects.

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.

Note: Testcases directly program initialization sequences.

3.3 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). 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'.

3.4 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` 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 factory `Testbench` 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 is one special interface (`TestbenchFactory`) for utilization of the descriptive information. It is explained in Testbench Factory.

3.4.1 IMPLEMENTATION MANIFEST FILE

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

3.4.1.1 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 the `Testbench` interface. These reference are represented by XML elements. As shown in Figure 3 there are two different reference types corresponding to the interface `Testbench` 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 `PyTestbenchImplementation` element is used for Python classes implementing the `Testbench` interface.

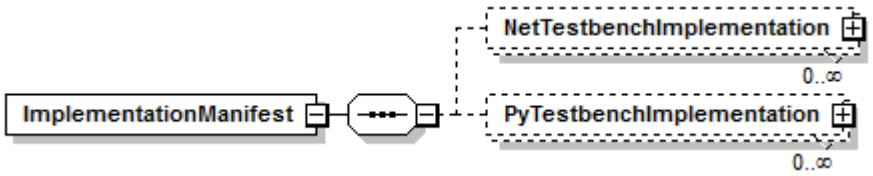


Figure 3: Implementation Manifest files contain a list of elements referring to C# or Python classes that implement the Testbench interface

Basically a manifest file may contain any number of references to different `Testbench` 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 4. References to Python classes have the same attributes (see Figure 5).

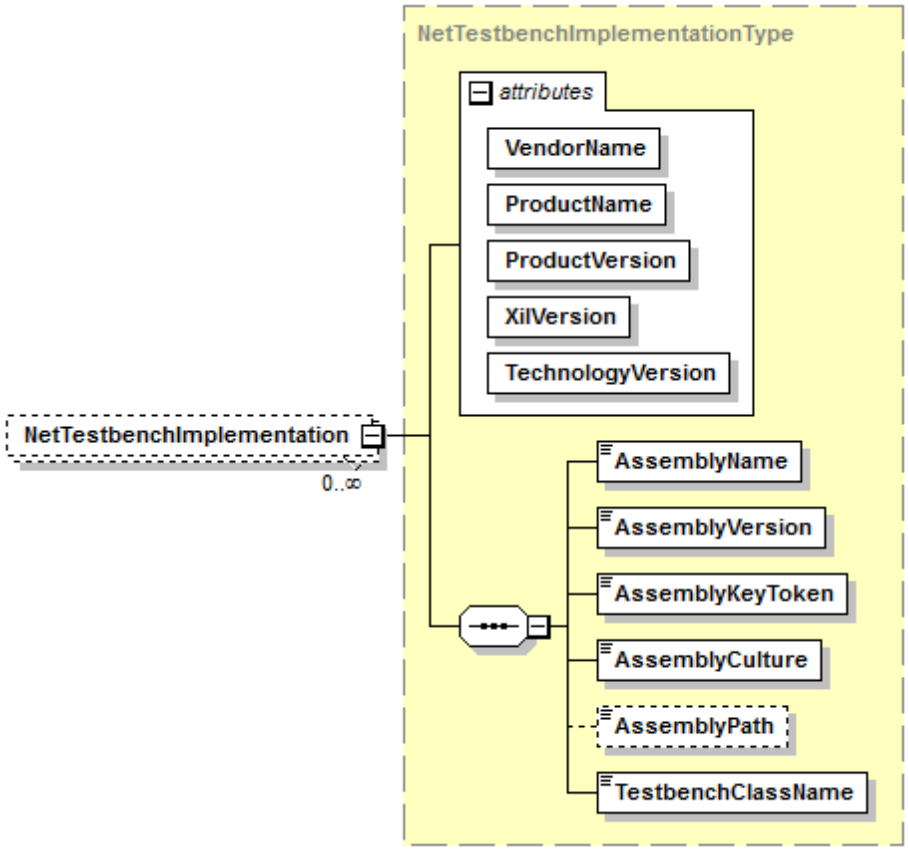


Figure 4: NetTestbenchImplementation element of the Implementation Manifest

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 content of a reference element provides technical information on code module and class name of the respective `Testbench` implementation. This information is required to find and load the module and create `Testbench` 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 4. 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`.

References to `Testbench` classes implemented in Python comprise the following elements as shown in Figure 5. `TestbenchClassModuleName` 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.

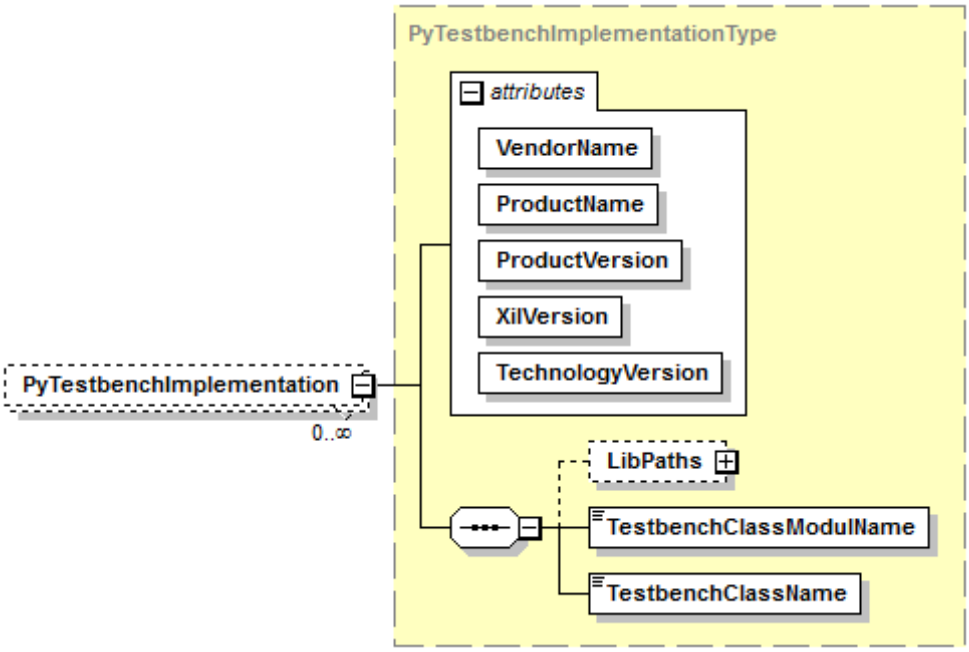


Figure 5: PyTestbenchImplementation element of the Implementation Manifest

3.4.1.2 File Naming convention and storage location

Principally each Implementation Manifest file may contain any number of references to different `Testbench` 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`

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

3.4.2 TESTBENCH FACTORY

The standard defines one interface that serve the creation of `Testbench` instances in a vendor independent manner. `Testbench` instances can be created by calling the method `CreateVendorSpecificTestbench` of the `TestbenchFactory` (see Figure 6). The desired `Testbench` implementation is selected by passing the respective vendor name, product name and product version.

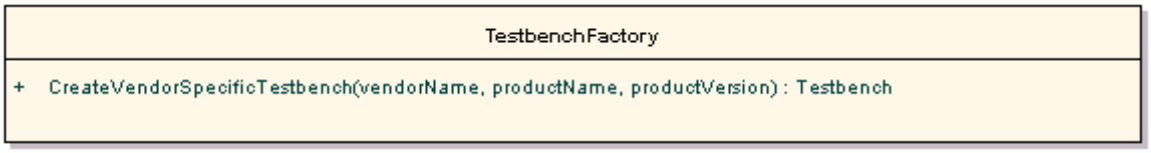


Figure 6: TestbenchFactory class

There is a reference implementation of the `TestbenchFactory` 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` classes. Instance creation of `TestbenchFactory` 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 Testbench

4.1 COMMON FUNCTIONALITIES

This chapter describes functionalities of XIL API Testbench which are not specific for one port. The most important parts of the package Common are described in the following sections.

Table 3 Packages of Common part

Sub package name	Description
Capturing and CaptureResult	Contains classes which do Capturing and handle the result of Capturings.
DocumentHandling	Gives an overview of all classes which are used to read or write content from/to the file system in different file formats.
Error	Contains common classes used for error handling. Error codes are defined in the sub package Error.
Signal and SignalGenerator	Contains classes to describe signal waveforms for general purpose and furthermore for generating signal waveforms stimulation use cases. The symbols which used for symbolic mapping are also described here.
ValueContainer	A set of classes which are designed to store values of different types, e.g. scalar, matrix or map values. Together with the ASAMTypes and the Collection classes, the value container classes are the fundamental type system which is used in the entire UML model.
WatcherHandling	Classes being used by the capturing classes for defining trigger conditions.

4.1.1 VALUECONTAINER

4.1.1.1 Overview

The `ValueContainer` package provides a set of container classes, which are used to store data values. These container classes are divided into three categories:

- The first category comprises all general container classes which are either scalars or which contain elements being accessed by integer based indices, e.g. vectors and matrices. Concrete sub classes are available for the most important data types like Boolean, integer, float and string. Some examples are `ScalarFloatValue`, `StringVectorValue` and `BooleanMatrixValue`.
- In addition, there are more application oriented classes, which are used for calibration access, for capturing or for signal generation. Examples are the classes `CurveValue`, `MapValue` and `SignalGroupValue`.
- The third category consists of named collections. These are explained in more detail in [7].

All container classes are derived from a common base class named `BaseValue`. Its method `Type()` allows to retrieve the concrete data type of a value container instance as specified by the enumeration type `DataTypes`.
It is possible to attach meta information to a value container instance. Examples for such meta information are the name of the variable or the unit of the value. More detailed information about meta data can be found in Attributes.
The `Value` property return copies (not references) of the internal data objects, e.g. a new instance of `VectorValue` is returned when using the `XVector` property on a `MapValue` object. So the value itself cannot be changed by altering the returned instances.
In the following chapters explain the different value container categories and the concept of meta data information in more detail.

4.1.1.2 General Value Container Classes

General `ValueContainer` classes represent scalar, vector and matrix values (Figure 7).

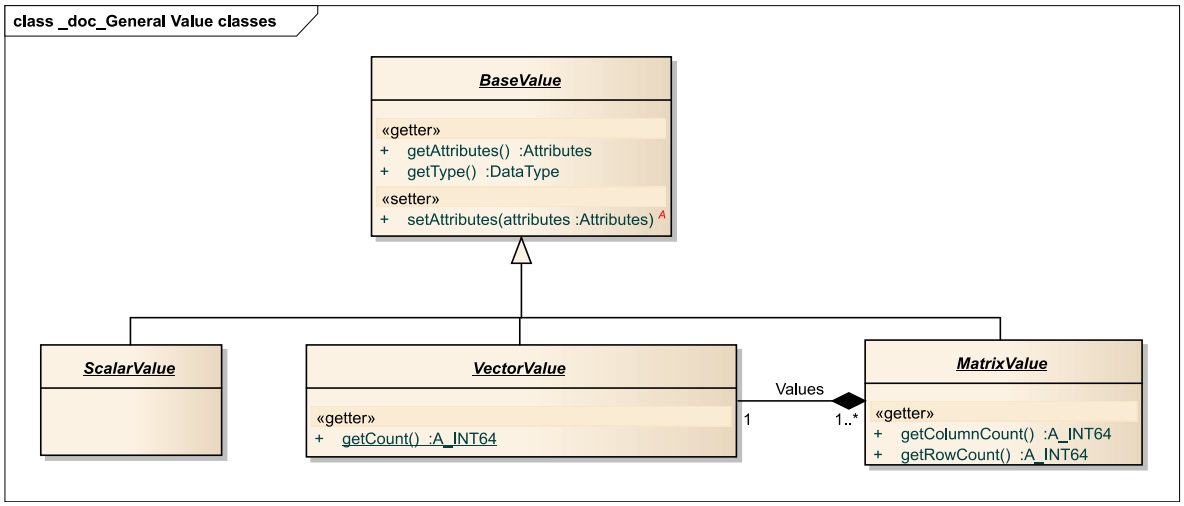


Figure 7: General Value classes

All elements inside a composite `ValueContainer` class (e.g. `VectorValue` or `MatrixValue`) are homogenous, meaning all elements must be of the same type, which is specified by the class. Class names are prefixed with `Int`, `Uint`, `Float`, `String` and `Boolean` corresponding to type of the managed elements (Figure 8).

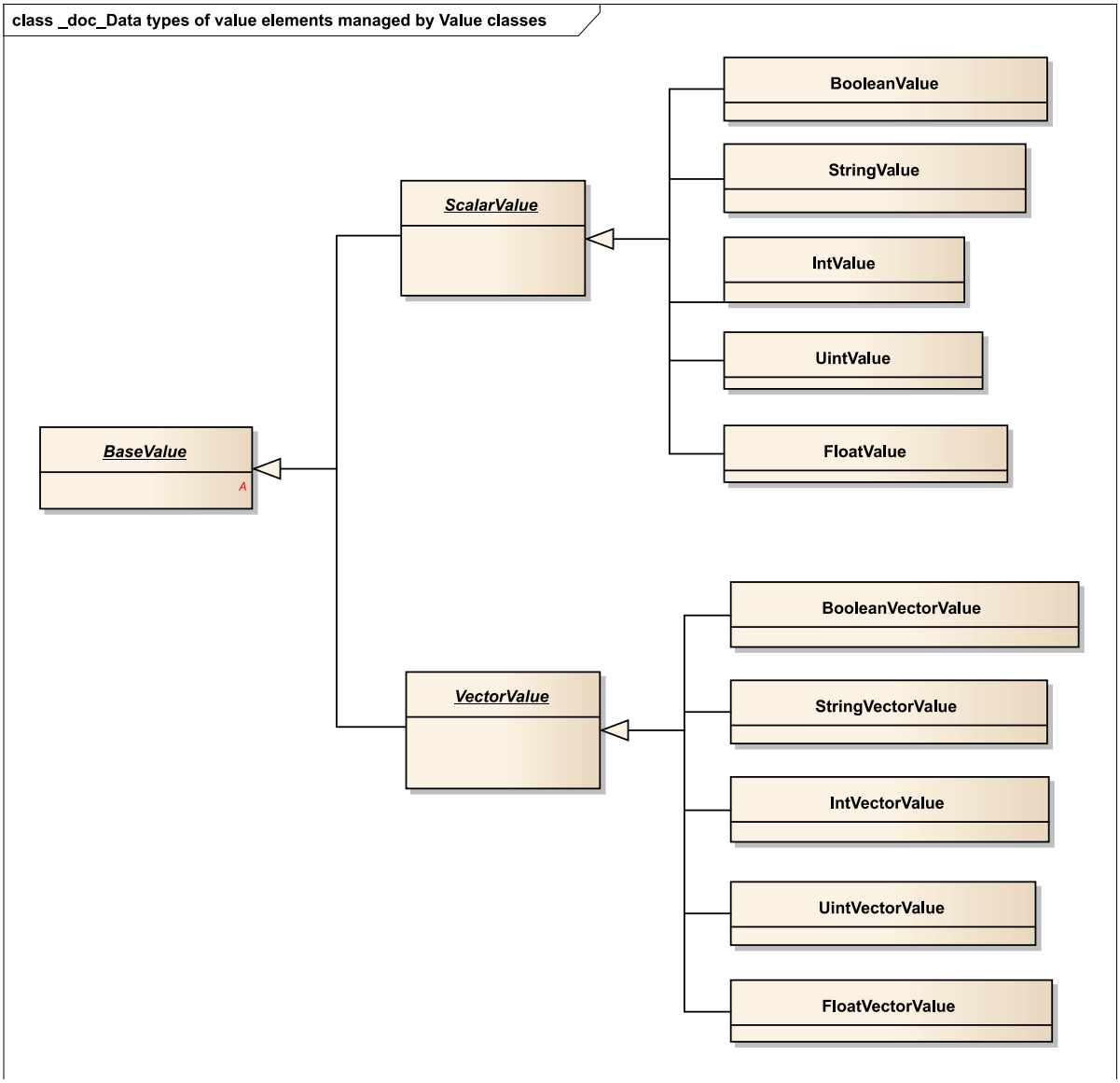


Figure 8: Data types of value elements managed by Value classes ScalarValue and VectorValue

The `ScalarValue` classes `IntValue`, `FloatValue`, `StringValue` and `BooleanValue` represent a single value of the particular data type.
The `VectorValue` classes `IntVectorValue`, `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`, `FloatMatrixValue`, `StringMatrixValue` and BooleanMatrixValue represent a two dimensional array of values. The `ColumnCount` and `RowCount` properties return the number of values of each dimension inside the matrix.

4.1.1.3 Application Oriented Value Container Classes

Application oriented `ValueContainer` classes (Figure 9) are used for more specialized applications like calibration and capturing.

`MapValue` 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 the function values must be equal to the length of the Y / X vector.
`SignalValue` and `SignalGroupValue` are used to represent captured signal data.

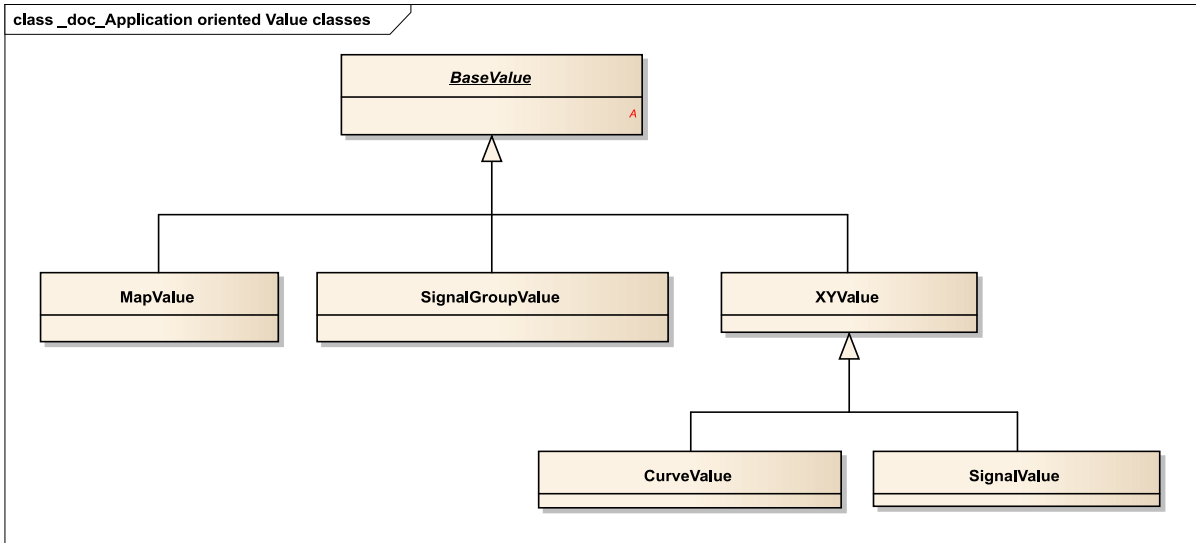


Figure 9: Application oriented Value classes

4.1.1.4 Attributes

Instances of the Attributes class are used to attach meta data to `ValueContainer` objects. The information consists of a list of attribute names and their values. The name and the value of an attribute are strings (`A_UNICODE2STRING`).

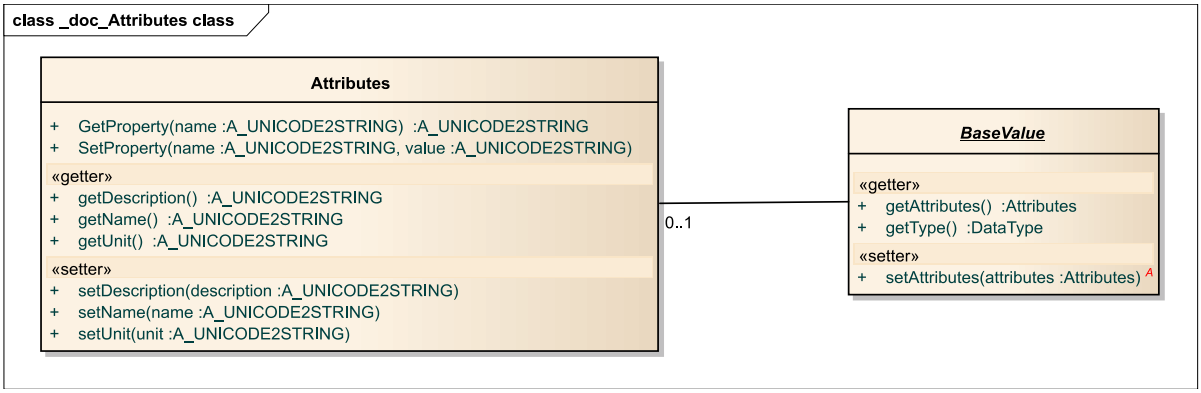


Figure 10: Attributes class

Some commonly used attributes are predefined. These are:

- Name
- Description
- Unit

It is also possible to add user-defined attributes using the property `Property()`.

4.1.2 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, and capture results.

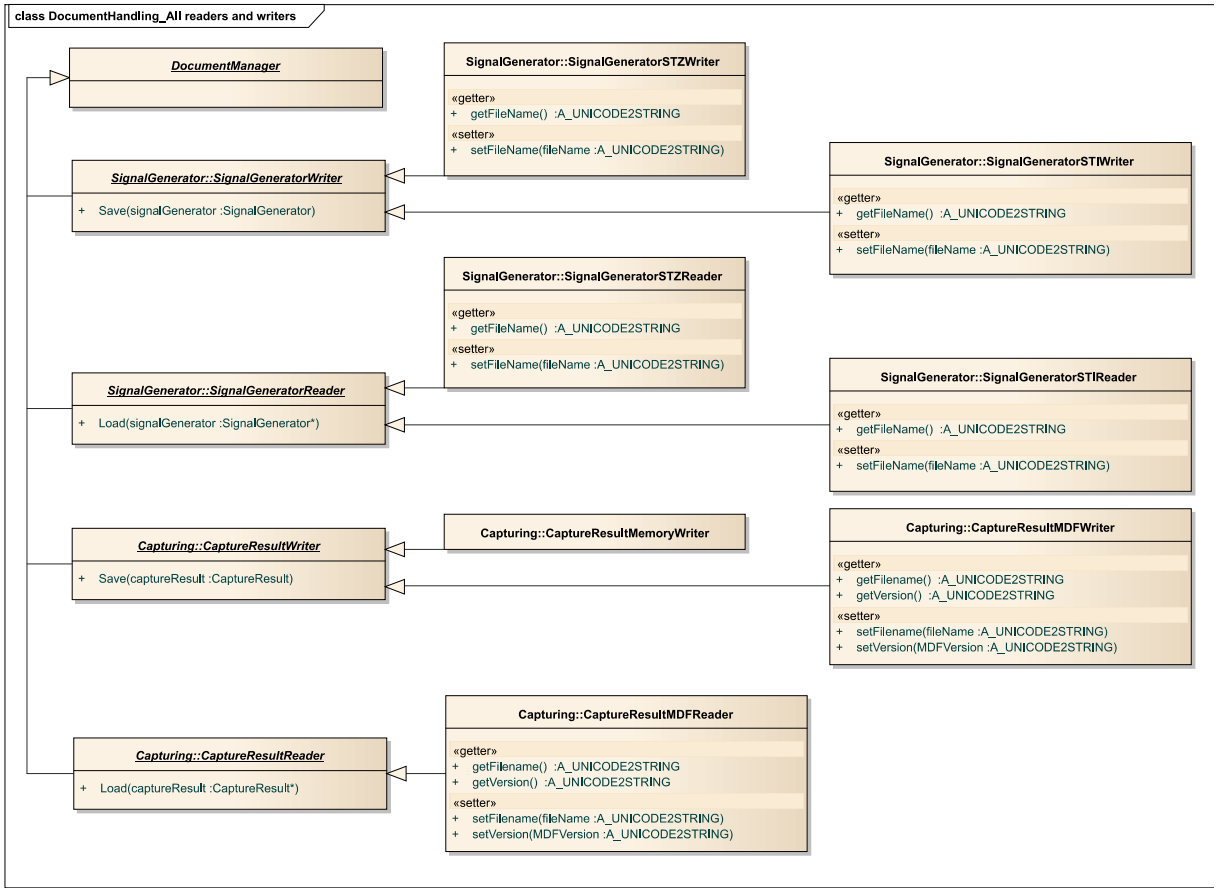


Figure 11: DocumentHandling in XIL

4.1.3 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.

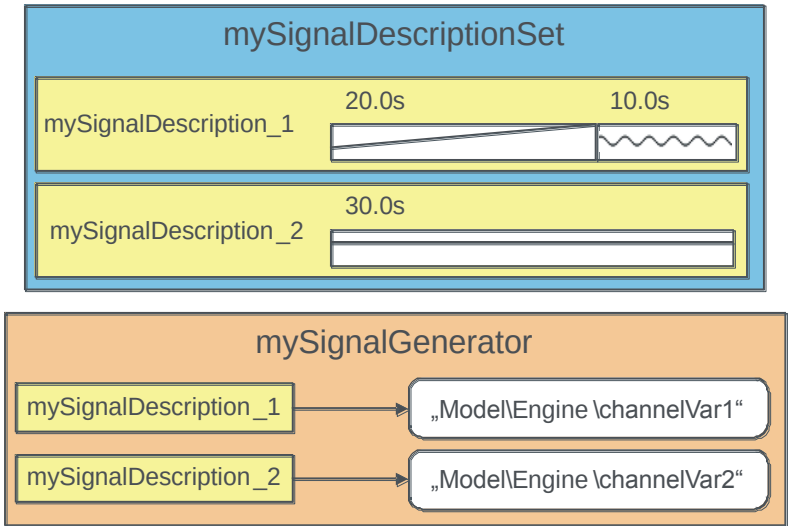


Figure 12: SignalDescriptions and SignalGenerator

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: 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. All parameters of all segment types can be specified by other signals.

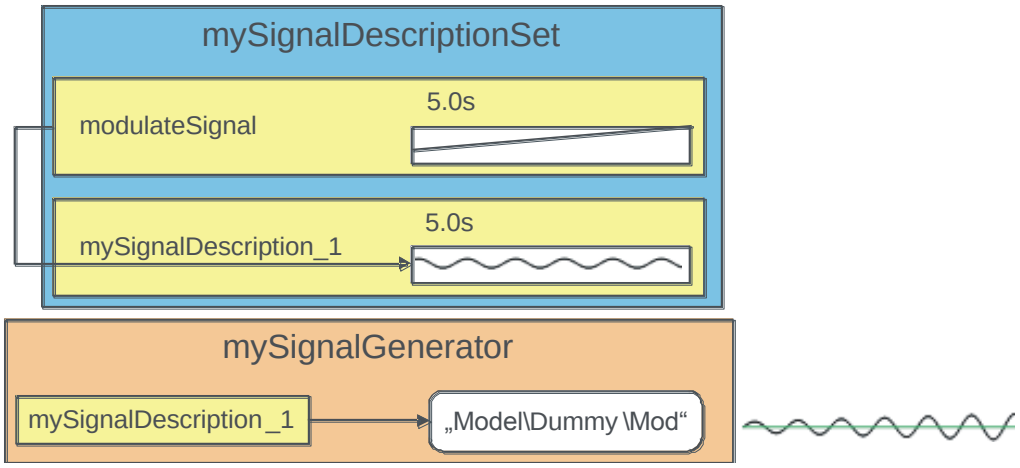


Figure 13: Modulate Signal Parameter by further Signals

Another possibility to describe signals is operational signal descriptions: An operational signal adds or multiplies two signals, as shown in Figure 14.

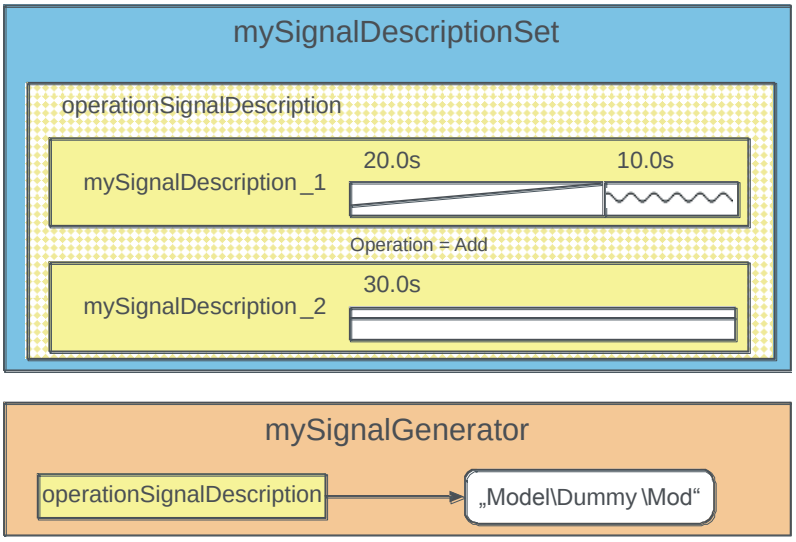


Figure 14: SignalDescriptions and SignalGenerator

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 15). Calling the method `CreateSignalValue()` on a signal description with the sample time as parameter, creates an according signal value (see chapter Valuecontainer). Calling method `CreateSignalGroupValue()` on `SignalDescriptionSet` creates a signal group value. If not all of the Y vectors has the same length, to the shorter ones IDLE values will be added.
IDLE values are the values produced by the IdleSegment as described in chapter

Idlesegment.

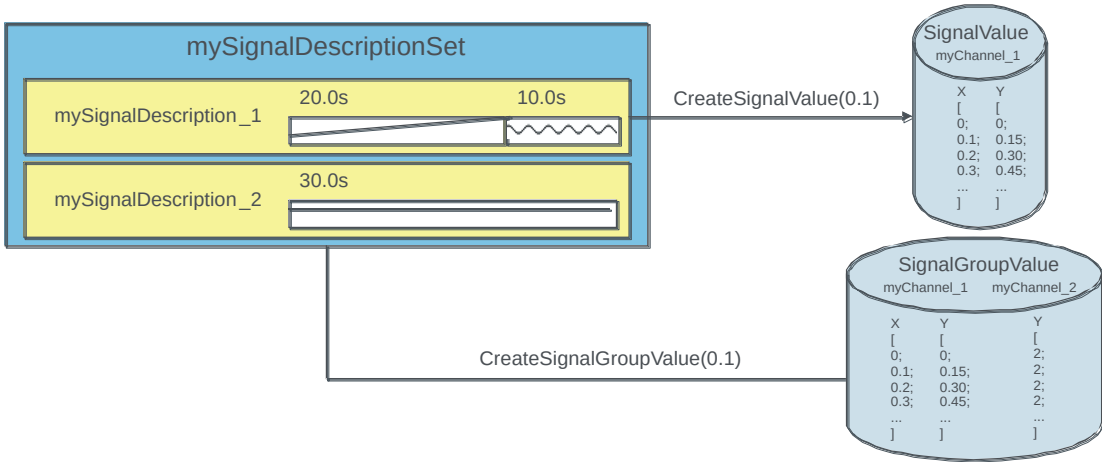


Figure 15: SignalDescriptions and SignalGenerator (data transformation)

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.

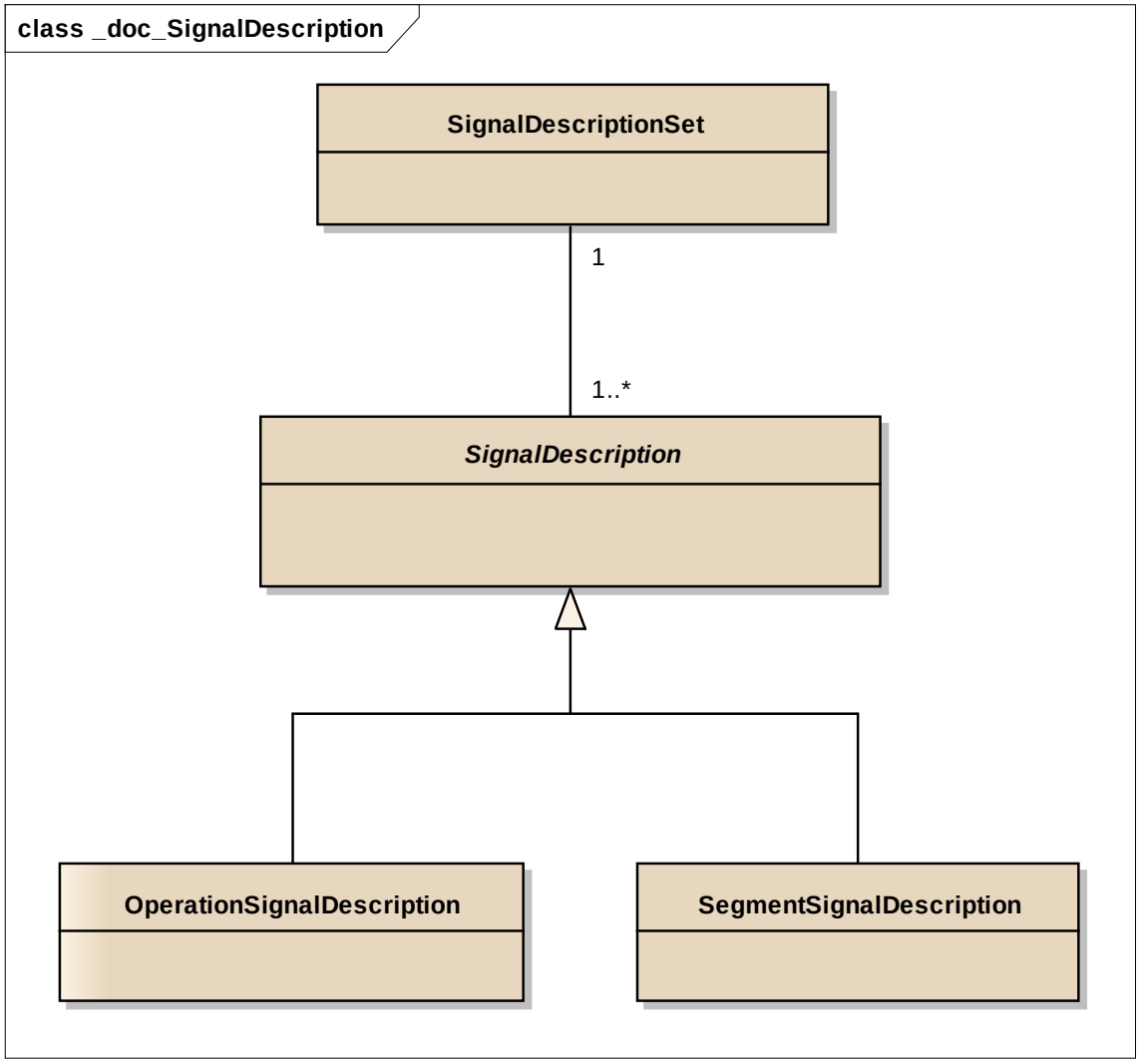


Figure 16: SignalDescription relations

4.1.3.1 Signal File Reading and Writing

To save the whole content of a `SignalDescriptionSet` or to load a complete set of signals into a `SignalDescriptionSet` there are two classes: The `SignalDescriptionWriter` and the SignalDescriptionReader. This concept allows it to load and save data in different formats. The sti and stz format are standardized in XIL.

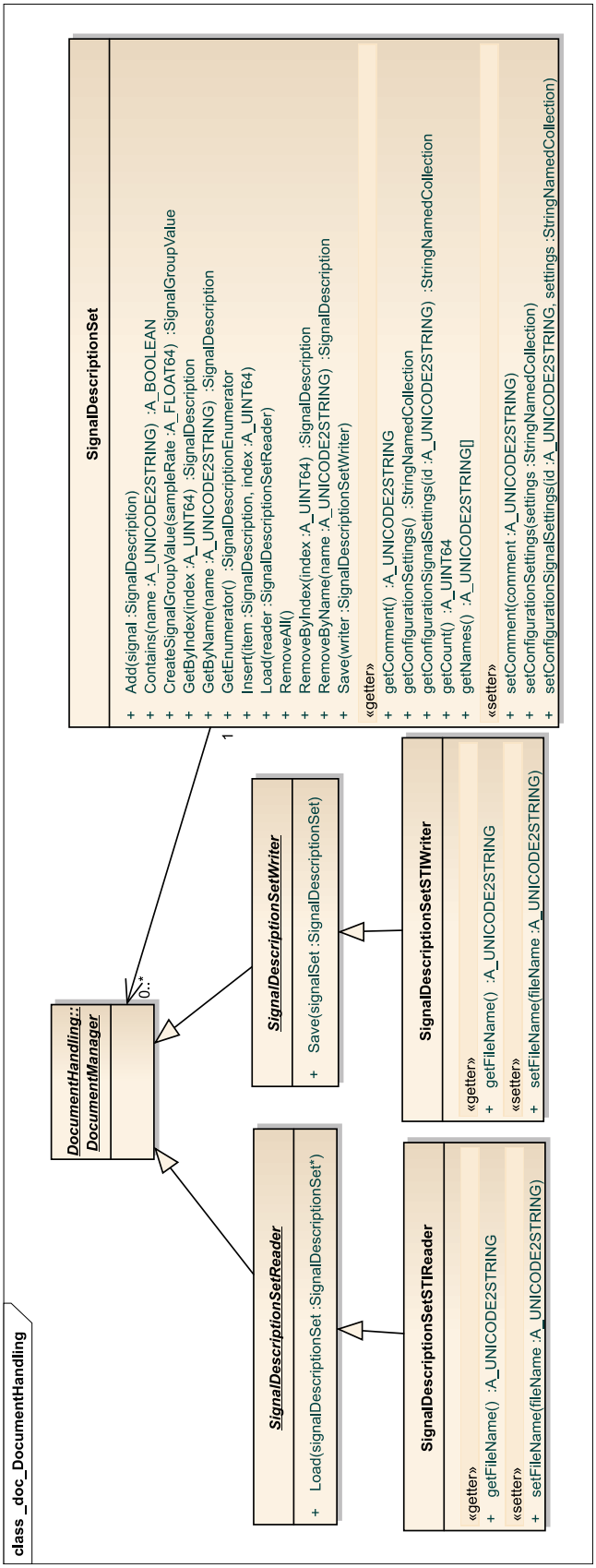


Figure 17: SignalDescription Reader and Writer

4.1.3.2 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 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), the `SignalSymbol` class (the parameter's value is obtained from a SignalDescription) and the `StringSymbol` class (the parameter's value is obtained from another simulation variable).

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`.

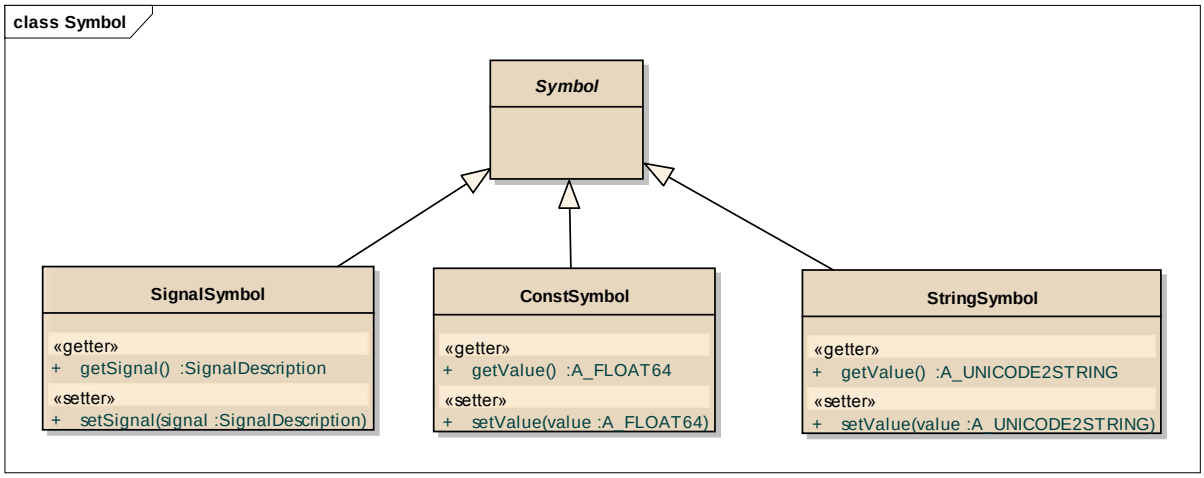


Figure 18: Symbol

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

4.1.3.3 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:

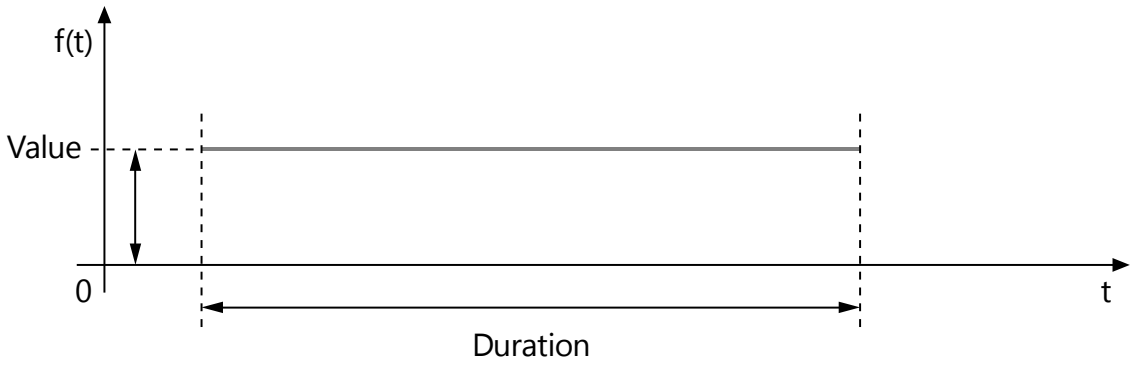


Figure 19: ConstSegment

Table 4 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 bei 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 (Δy/Δx)

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:

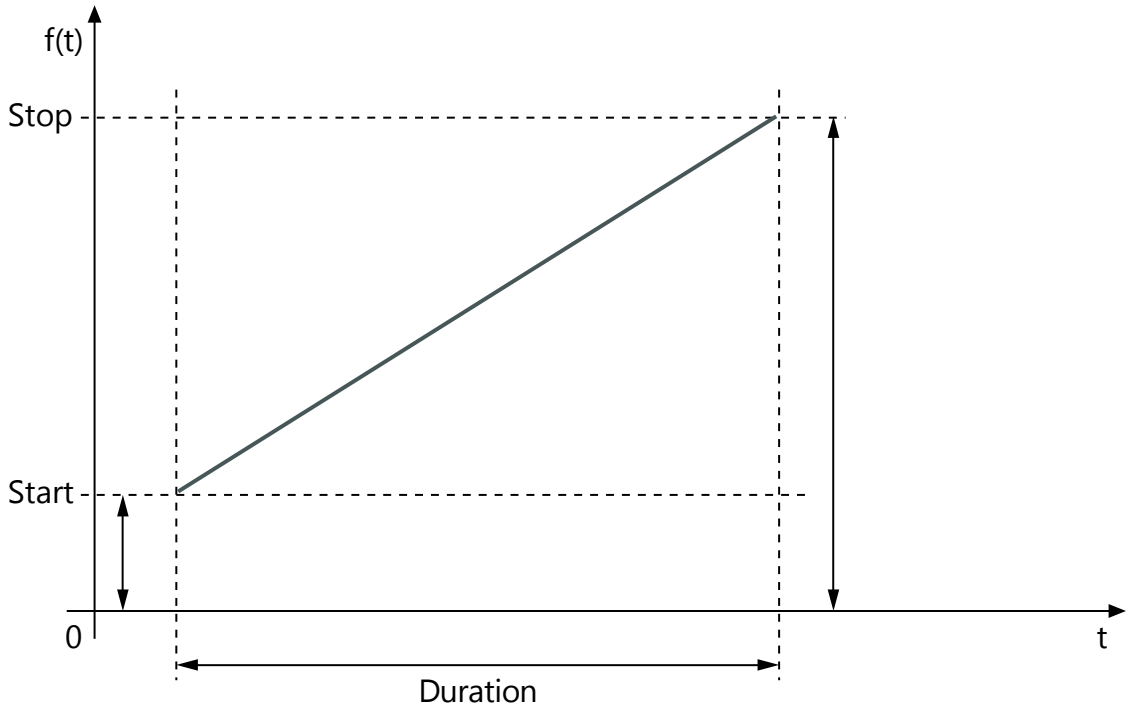


Figure 20: RampSegment

Table 5 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 other parts of the model to write to the variable (eg. model-i/o or user interaction).
If the variable was not written during the idle time by some other parts of the model, the variable is left untouched and will keep its value.
In case of evaluating an `IdleSegment` to produce numerical data e. g. using the method `CreateSignalValue` of the class `SignalDescription`, the value of `NaN` (IEEE 754) must be generated. Please refer to the technology references (see [5] and [6]).

Mathematical description

none

Graphical Representation Example:

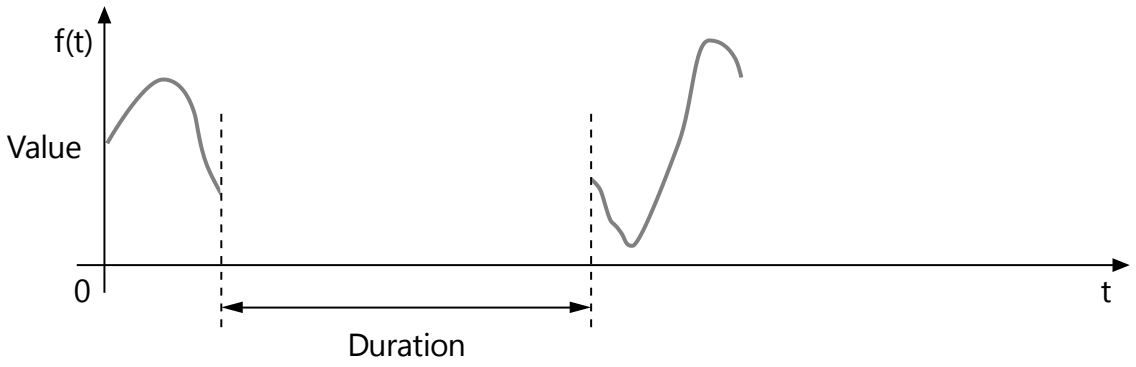


Figure 21: IdleSegment

Table 6 Parameter IdleSegment

Parameter	Description
Duration	Duration / run time of the segment Unit: Seconds [s] Range: [0 < Duration <= MAX(A_FLOAT64)]
StopTrigger	To define a stop trigger bei 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 u1 and u2 in the range 0..1 (e.g. generated via random()) two standard normal-distributed and independent random numbers z1 and z2 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:

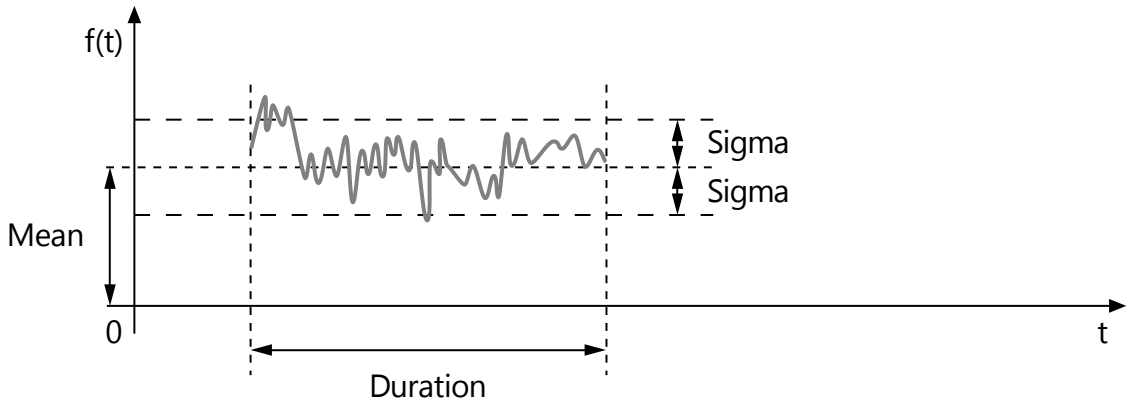


Figure 22: NoiseSegment

Table 7 Parameter NoiseSegment

Parameter	Description
Duration	Duration / run time of the segment Unit: Seconds [s] Range: [0 < Duration <= MAX (A_FLOAT64)]
Mean	Mean value, where the Gaussian distribution is moving Unit: - Range: [MIN (A_FLOAT64) <= Mean <= MAX (A_FLOAT64)]
Sigma	Standard deviation of the signal amplitude against the mean value Unit: - Range: [MIN (A_FLOAT64) <= Sigma <= MAX (A_FLOAT64)]
Seed	Start value of the random generator Unit: - Range: [-2147483646 <= Seed <= +2147483645]
StopTrigger	To define a stop trigger bei 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:

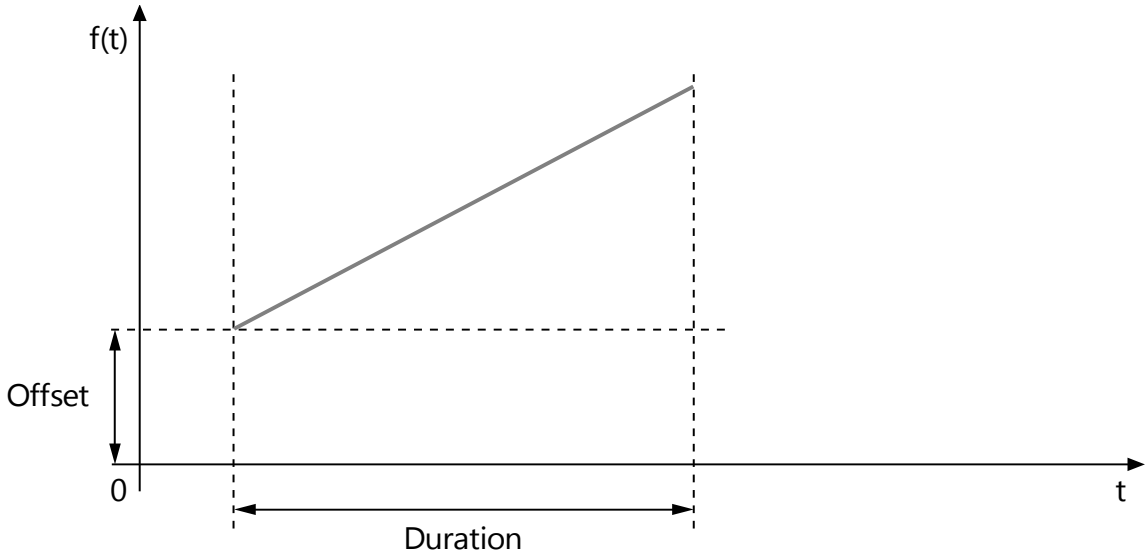


Figure 23: RampSlopeSegment

Table 8 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 bei 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:

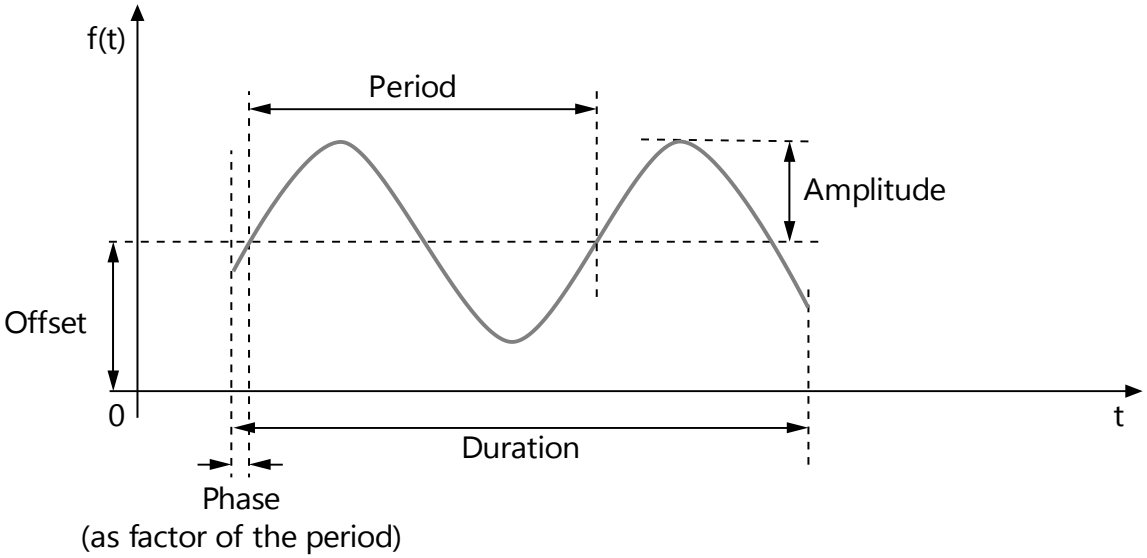


Figure 24: SineSegment

Table 9 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 bei 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:

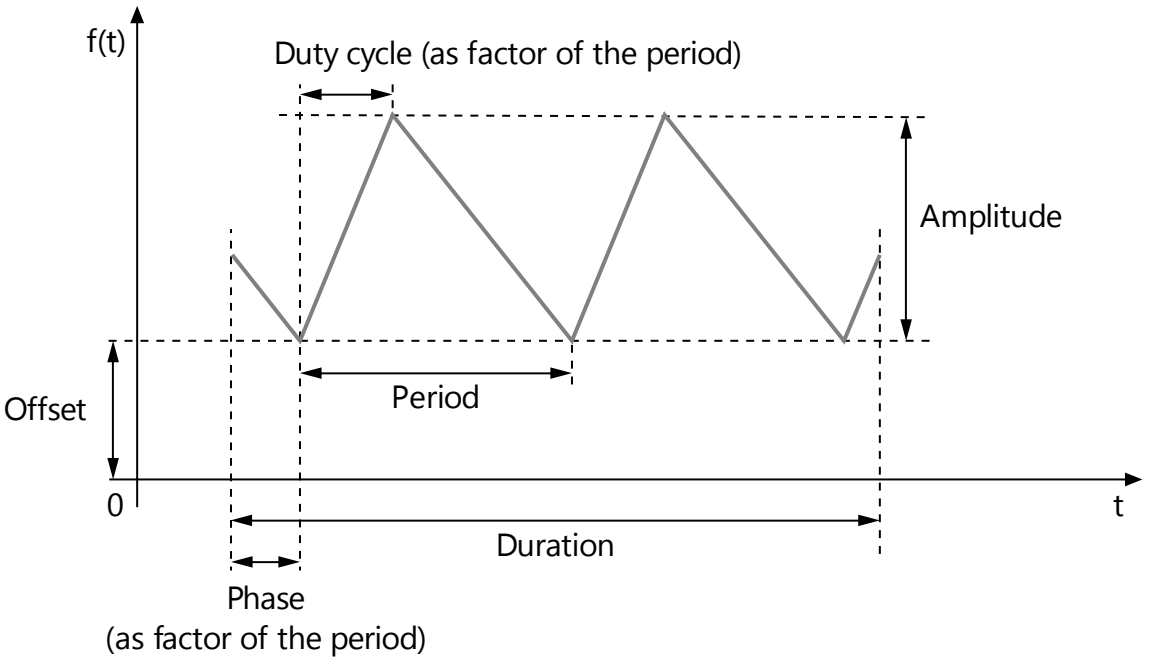


Figure 25: SawSegment

Table 10 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 bei 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:

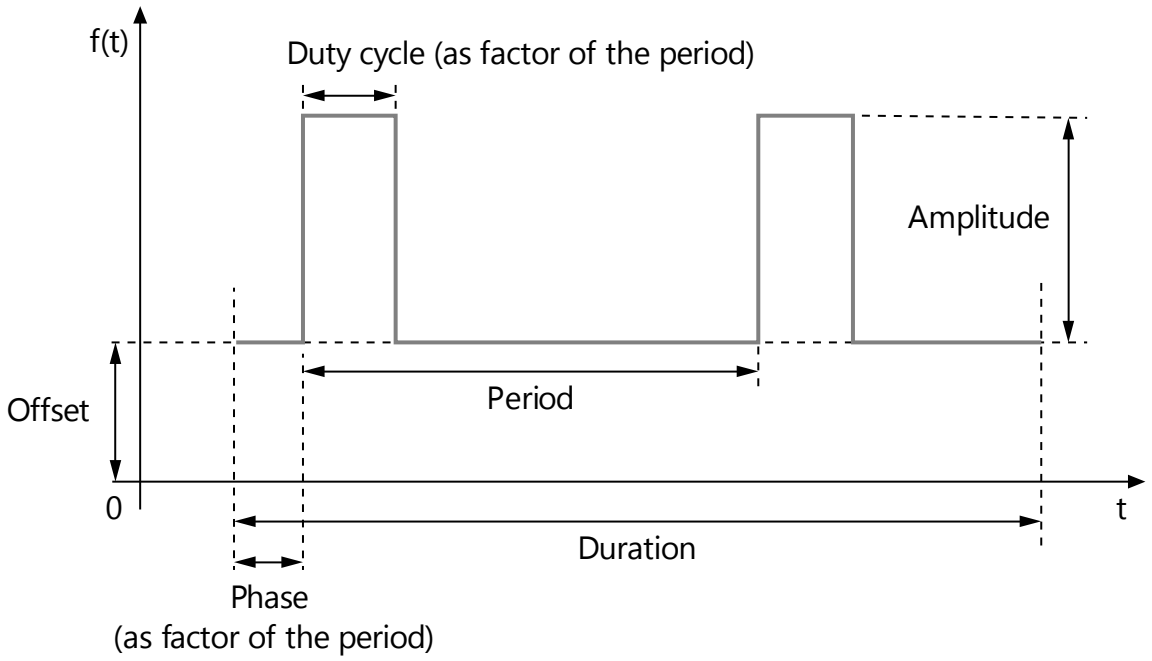


Figure 26: PulseSegment

Table 11 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 bei 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:

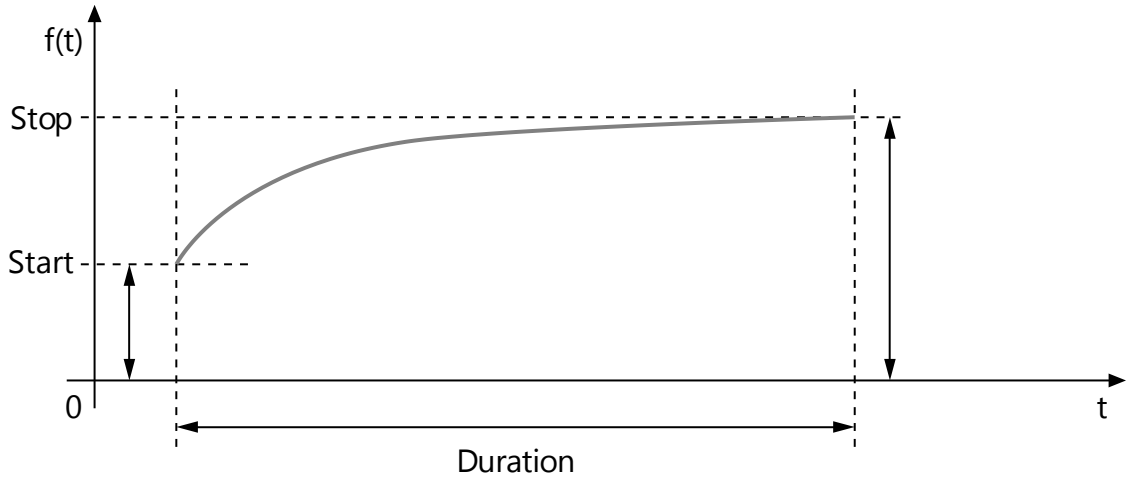


Figure 27: ExpSegment

Table 12 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 bei 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.

Table 13 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 DataFilesSegment 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.

Table 14 Parameter DataFileSegment

Parameter	Description
Duration	Duration / run time of the segment Unit: Seconds [s] Range: [0 < Duration <= MAX (A_FLOAT64)]
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: [0 < Duration <= MAX (A_FLOAT64)]
StopTrigger	To define a stop trigger bei 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 15 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

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

S_1: First segment / first operand

S_2: Second segment / second operand

Table 16 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))

4.1.3.4 Using Signal Descriptions

Each SegmentSignalDescription consists of one or more segments. The sequence diagram show the creation of exemplary signal types.

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

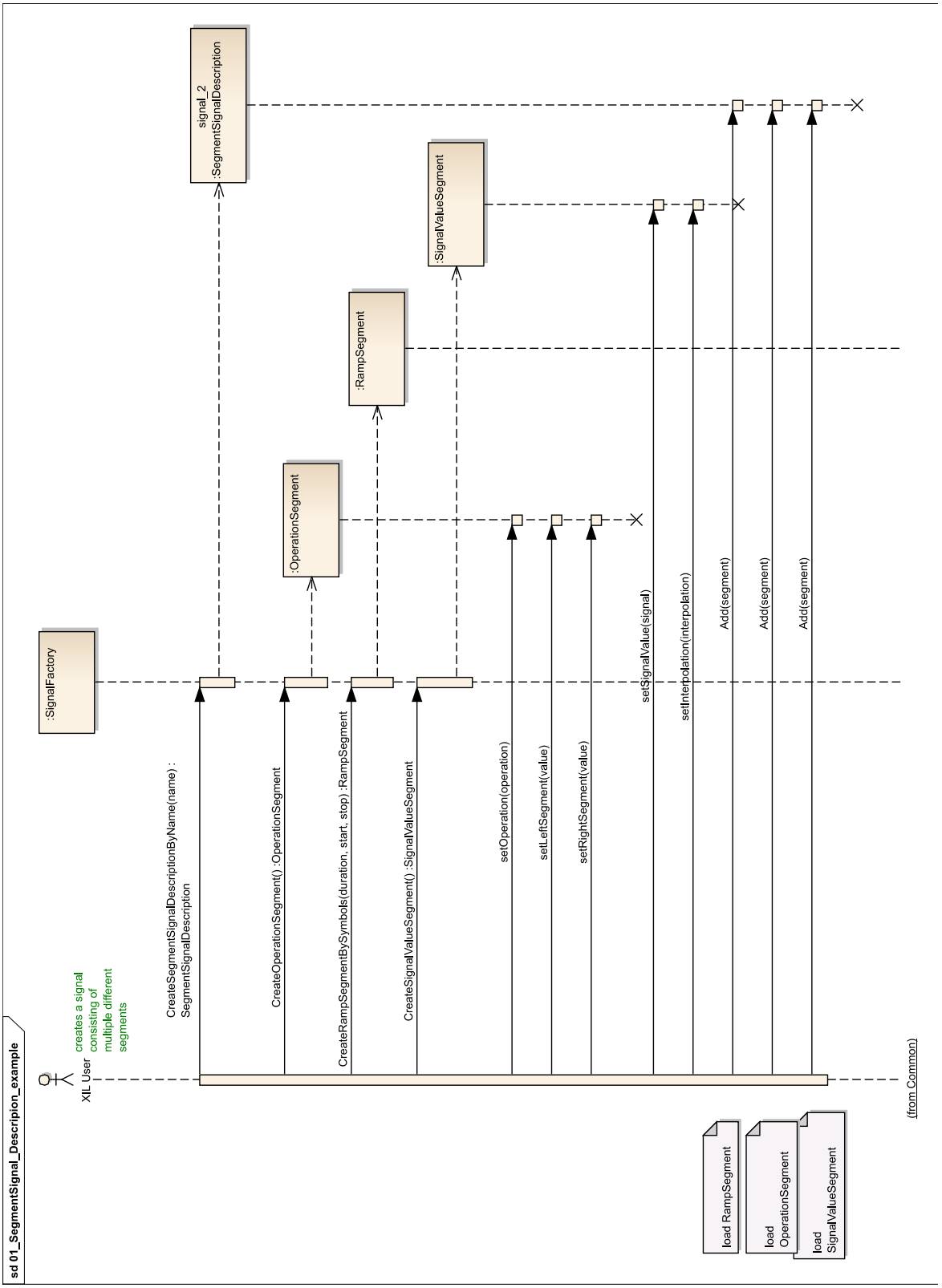


Figure 28: Create Segment Signal Description Example

CREATING AN OPERATION SIGNAL

The sequence diagram Figure 29 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.

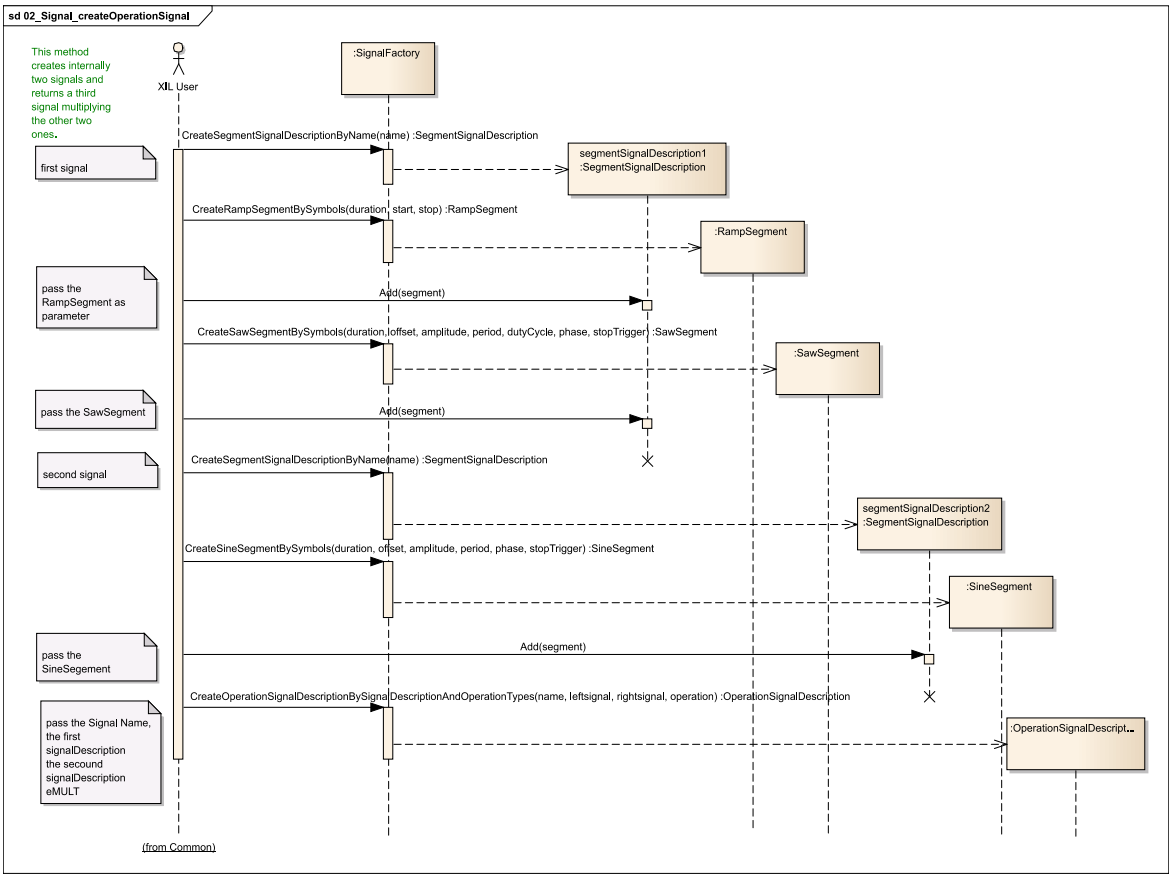


Figure 29: Create OperationSignal

CREATING A WOBBLE SIGNAL

In this example (Figure 30) a periodic signal is created. The frequency property is described by a saw signal, so that the sine signal is wobbling.

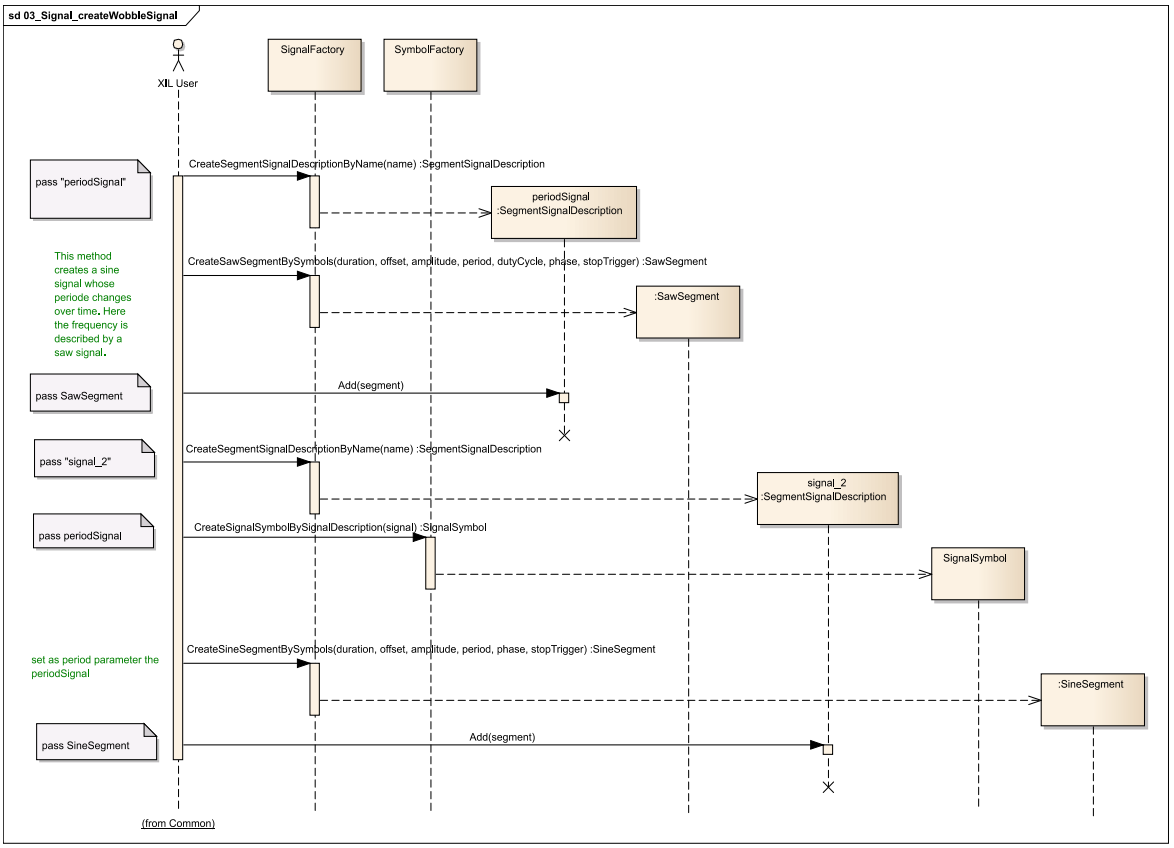


Figure 30: Create a wobbling signal

SIGNAL DESCRIPTION SET

In this example (Figure 31) the access to a signal description set is shown. Two signal descriptions, already created before, are added to the signal description set. Then the set is queried for the names and the contained descriptions. Each of the descriptions is converted into a `SignalValue` object.

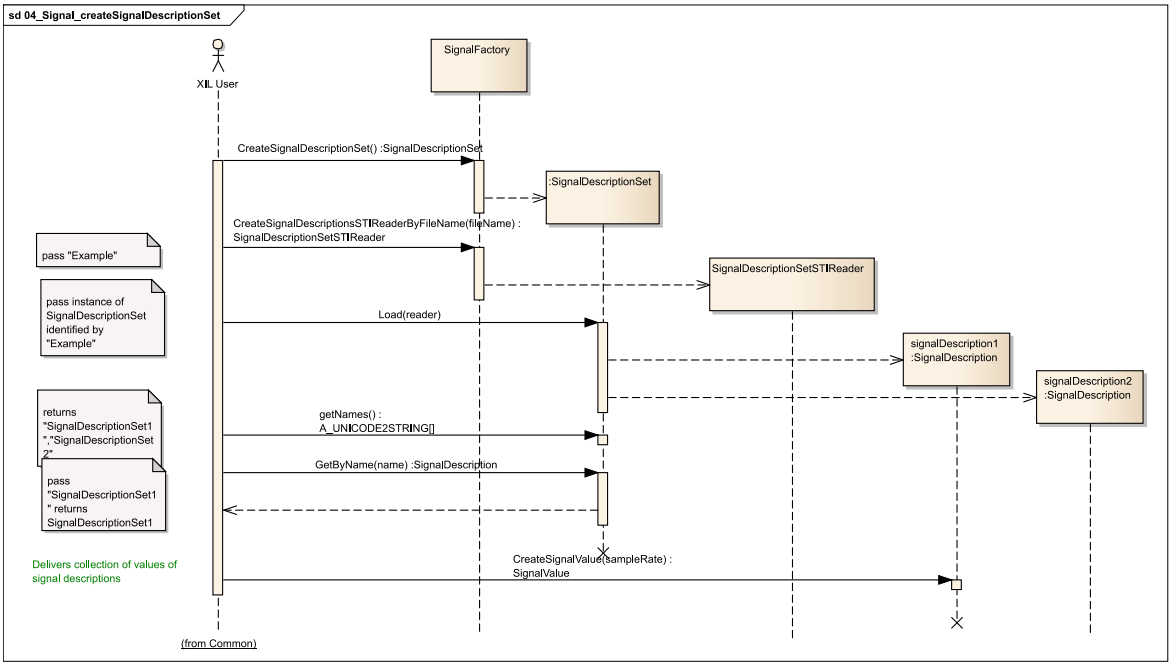


Figure 31: Create SignalDescriptionSet

LOADING THE SIGNAL DESCRIPTION

The Figure 32 shows the alternative way to get a signal description set: via loading an existing set from a STI file.

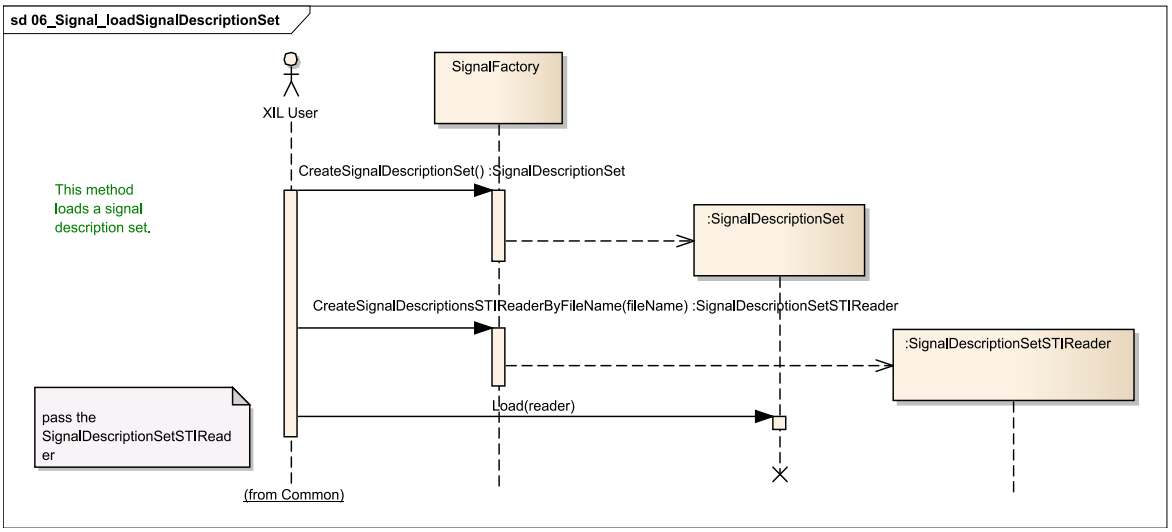


Figure 32: Load a SignalDescriptionSet

SAVING THE SIGNAL DESCRIPTION

Figure 33 shows how to save a signal description set to a file for further reuse.

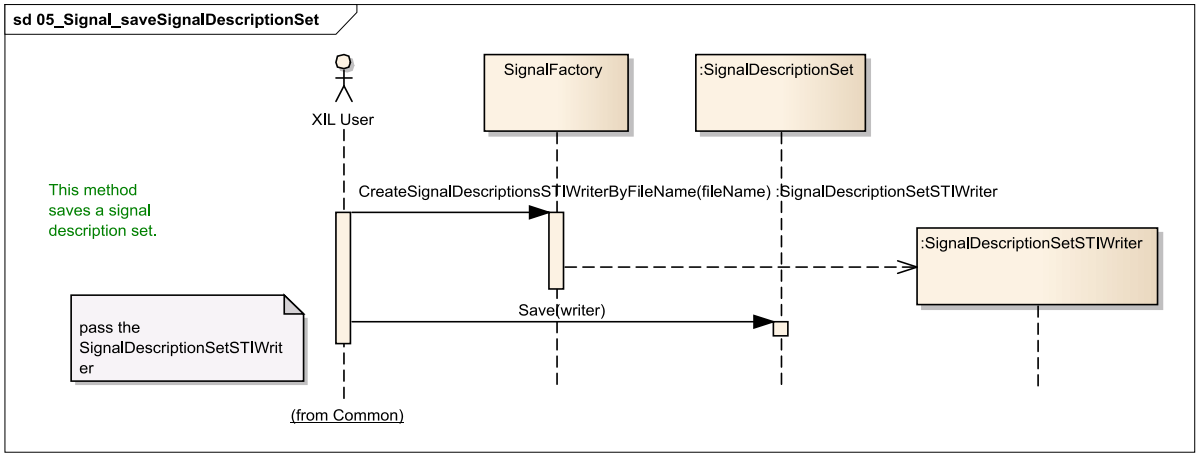


Figure 33: Save SignalDescriptionSet

4.1.3.5 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.

4.1.4 WATCHER

4.1.4.1 General

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

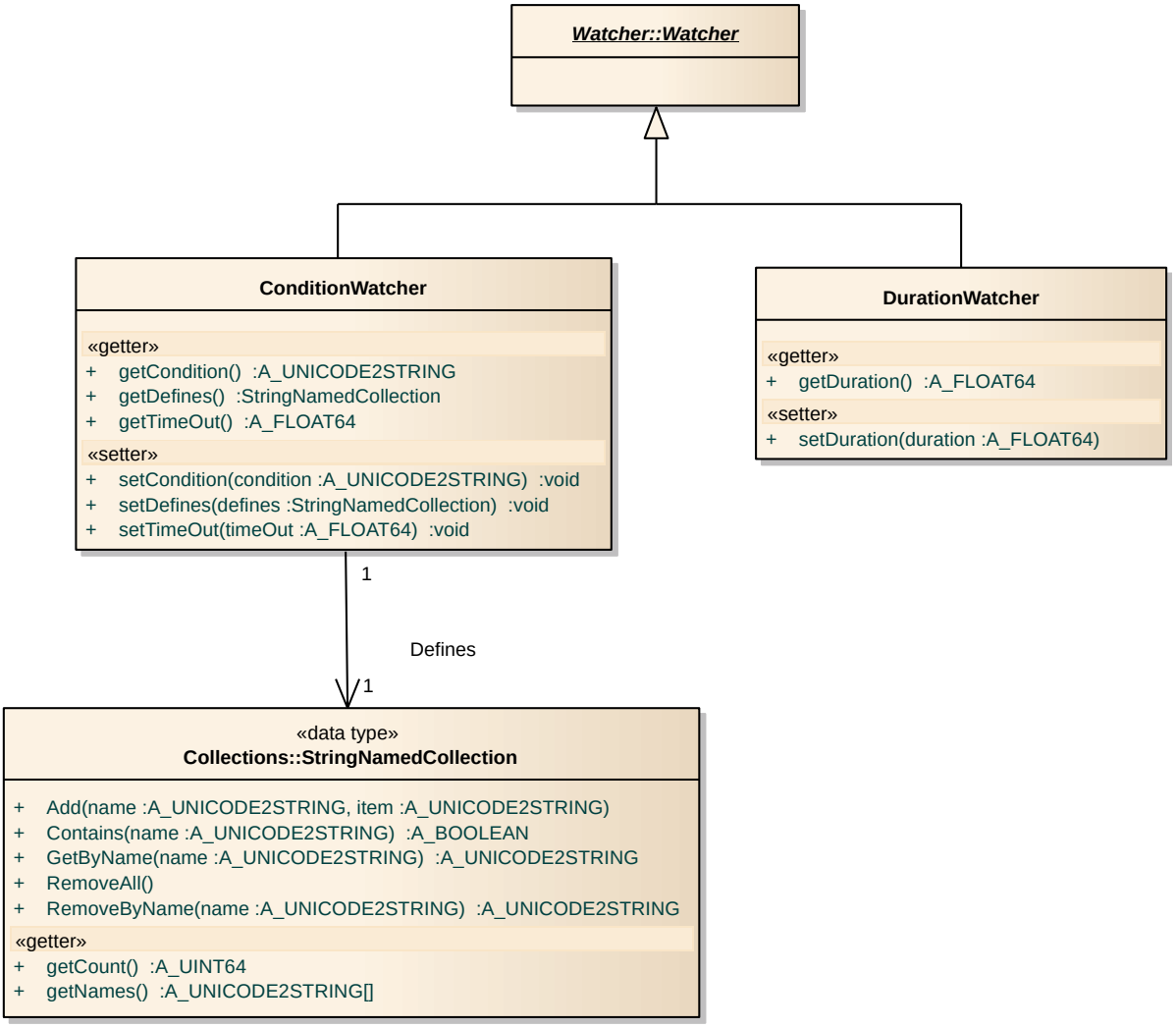


Figure 34: Testbench Watcher

XIL API distinguishes between two watcher types:

1. Watchers that get activated if a specified condition becomes true (i.e. a simulation variable takes on a specific value)
2. Watchers that get activated after a specific amount of time

The first type of watchers is implemented in the `ConditionWatcher` class. The second type is implemented in the `DurationWatcher` class.

DURATIONWATCHER

The `DurationWatcher` fires after a specified duration (given by the property `Duration)` relative to the watcher activation. For details on the behaviour of the `DurationWatcher` in the context of a capture see section Capturing.
See also property `DurationUnit` of `Capture` for the interpretation of the property `Duration` of the `DurationWatcher`. This is different for the values `eSECONDS` and `eSAMPLES` of the property `DurationUnit` of class `Capture`.

CONDITIONWATCHER

The `ConditionWatcher` fires when the condition specified by the property `Condition` becomes true after its activation. The syntax of the condition is defined in Syntax of Watcher Conditions. The condition syntax is validated when the property `Condition` is set. As soon as the specified condition becomes true, the `ConditionWatcher` fires.
The `ConditionWatcher`'s condition may contain signals or parameters of the simulation model. To shorten the condition expression, aliases for the simulation variables are used which are mapped to model paths. The mapping can be defined via the property 'Defines'. The use of aliases makes a condition more readable for humans and leads to a decoupling of the test case description from the simulation model. All mappings of aliases used in the condition expression must be added to the StringNamedCollection accessed by the Defines property. The condition and both sides of the 'Defines' mapping are of type `A_UNICODE2STRING`, e.g. "`velocity`" is mapped to "`Model Root/Subsystem/Vel/Value`".

4.1.4.2 Using the TimeOut

Some method calls may be blocking and insist on a watcher condition to become true, in order to end properly. As an example, the `Capture.Fetch` method with the parameter `whenFinished := TRUE` insists on a watcher condition to become true to end and return. In order to prevent the program of blocking endlessly, the property `TimeOut` of the `ConditionWatcher` should be set. This ensures, that the `ConditionWatcher` stops evaluating its condition and fires its event as soon as the duration given by `TimeOut` is elapsed.
If a timeout value of zero is given, the `ConditionWatcher` fires its event immediately and the specified condition is ignored. The default value of `TimeOut` is -1.0 for an infinite timeout.

Note: The interpretation of the property `TimeOut` of a watcher used in the context of `Capture` depends on the value of the property `DurationUnit` of `Capture`; it may be interpreted as number of samples or duration in seconds!

See also property `DurationUnit` of the Capture for the interpretation of the parameter `delay`. This is different for the values `eSECONDS` and `eSAMPLES` of `DurationUnit`.

4.1.5 DATA CAPTURING

4.1.5.1 Introduction

Capturing is a process of acquiring data in a continuous data stream. It guarantees that all process data can be retrieved as they occur related to the real-time service respective to the capture service task. The data aquired can be retrieved after completion of this process or even while it is still in progress.

The classes in the Capturing and in the Capture Result package are used to define captures, to control the execution of capturing and to obtain the measured data as results. They are located in the Testbench Common package since they are used for MAPort, ECUMPort and NetworkPort.

4.1.5.2 Capturing

The main class of the Capturing package is the class Capture (Figure 35). It is used to define captures and to control the execution of capturing.

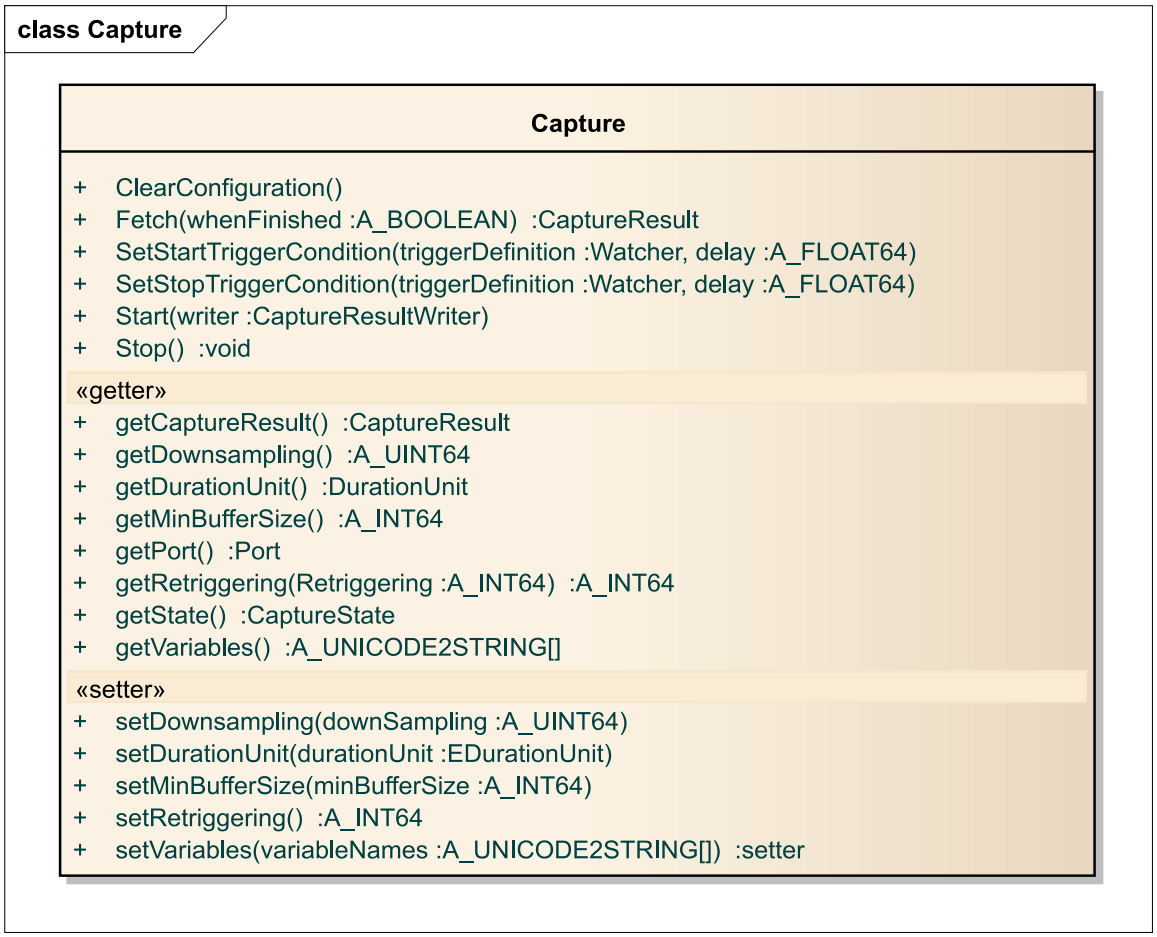


Figure 35: The class Capture

CAPTURE

An instance of class Capture represents a capture definition. A capture is created by the port for which a capture shall be defined (`MAPort`, `ECUMPort` or `NetworkPort`). The capture is configured by setting the properties or calling the methods shown the table below:

Table 17 Configuration Properties and Methods of Capture

Property /Method	Description
Variables	Defines the variables for capturing. Only complete sets of variables can be set. Each new property assignment will clear and reset the variable list.
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 CaptureResult. The default is 1 (no downsampling). If downsampling is not supported by the XIL Server an exception will be thrown (eCOMMON_NOT_SUPPORTED). (optional property)
DurationUnit	Specifies wether the unit used for the x axes of CaptureResults is timebased (in seconds) or sample-based (in integer values, 0, 1, 2 etc.). The enumeration values eSECONDS and eSAMPLES of DurationUnit can be assigned. The default is eSECONDS. If `eSAMPLES` is set, the parameter delay of the methods „`SetStartTriggerCondition`“ and „`SetStopTriggerCondition`“,the duration of the `DurationWatcher` and the `TimeOut` of the `ConditionWatcher` are interpreted as `A_INT32`. If `eSECONDS` is set, these values are are interpreted as `A_FLOAT64`.
MinBufferSize	Defines the minimum buffersize [byte] of the real-time service. The buffer size has to ensure continuous measurement for the specified duration. As a default, the complete available buffer is reserved.
Retriggering	Retriggering means that after receiving a stop trigger the data acquisition is waiting for a start trigger again. The Retriggering property is an integer that is interpreted as follows: Retriggering = 0, no retriggering. This means that the given start trigger condition is watched once.

Property /Method	Description
	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.) Retriggering = -1, defines an infinite number of start/stop trigger sequences. If the Retriggering value is not valid, an exception is thrown.
SetStartTriggerCondition	Sets the start trigger condition See also `DurationUnit` for the interpretation of the parameter `delay`. This is different for the values `eSECONDS` and `eSAMPLES` of `DurationUnit`.
SetStopTriggerCondition	Sets the stop trigger condition. See also `DurationUnit` for the interpretation of the parameter `delay`. This is different for the values `eSECONDS` and `eSAMPLES` of `DurationUnit`.

After configuration of the Capture object, the method `Start` is called to put the Capture from state eCONFIGURED into the state eACTIVATED. In eACTIVATED, data acquisition starts if the start trigger condition becomes true. If no start trigger is set, data acquisition starts immediately after calling method `Start`.

Note: The Method `Start` is non-blocking. It returns immediately after being called – even if the capturing has not been started yet.

The capturing is stopped either if the stop trigger condition becomes true or by calling the method `Stop` explicitly.

Note: The start trigger is defined by a ConditionWatcher object, the stop trigger is defined by either a ConditionWatcher or a DurationWatcher object.

This allows stopping the data acquisition after a specific amount of time (DurationWatcher) or according to a specific condition (ConditionWatcher). If no stop trigger is set, the data acquisition runs until the method `Stop` is called.

The method ClearConfiguration clears any set configuration, releases all ressources and stored data.

Accessing the property `State` allows to observe the current capture state, e.g. to check if the capture state is equal to `eRUNNING` indicating that the start trigger occurred already.

Different ways to access the acquired data of a Capture object

Property `CaptureResult`
Returns a CaptureResult object that contains all acquired data of the Capture, from beginning of the data acquisition til the end of data acquisition.
If no data has been acquired (e. g. no start trigger condition has become true), the corresponding SignalValue objects within the CaptureResult have the length of 0, which means they are empty.
Access to this property is allowed in Capture state eCONFIGURED only.

Method `Fetch`
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 beginning of the data acquisition til the call to `Fetch` or since data acquisition has started, if `Fetch` is called for the first time.
If no data has been acquired meanwhile, (e. g. no start trigger condition has become true), the corresponding `SignalValue` objects within the `CaptureResult` have the length of 0, which means they are empty.

If the captured data is stored to file (e.g. use of `CaptureResultMDFWriter`), a call to `Fetch` raises an exception.

The call of this method is allowed in Capture state `eACTIVATED`, `eRUNNING`, `eFINISHED` only.

Note: There is a parameter, `whenFinished`, in order to change the method's semantic und 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 is blocked until the Capture state `eFINISHED` has been reached.
Motivation: this is a convenient way to wait until the Capture's data acquisition has finished within 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 become true within an expected time, or by defining a timeout via the `TimeOut` property of the `ConditionWatcher` object in order to force the corresponding watcher to fire its event after the configured timeout has elapsed.

Special cases: Delayed Triggering

When setting a start or a stop trigger for a Capture object, it is possible to set a delay. In case the delay is not zero, this leads to the following behaviour, as depicted in the following Figure 36:

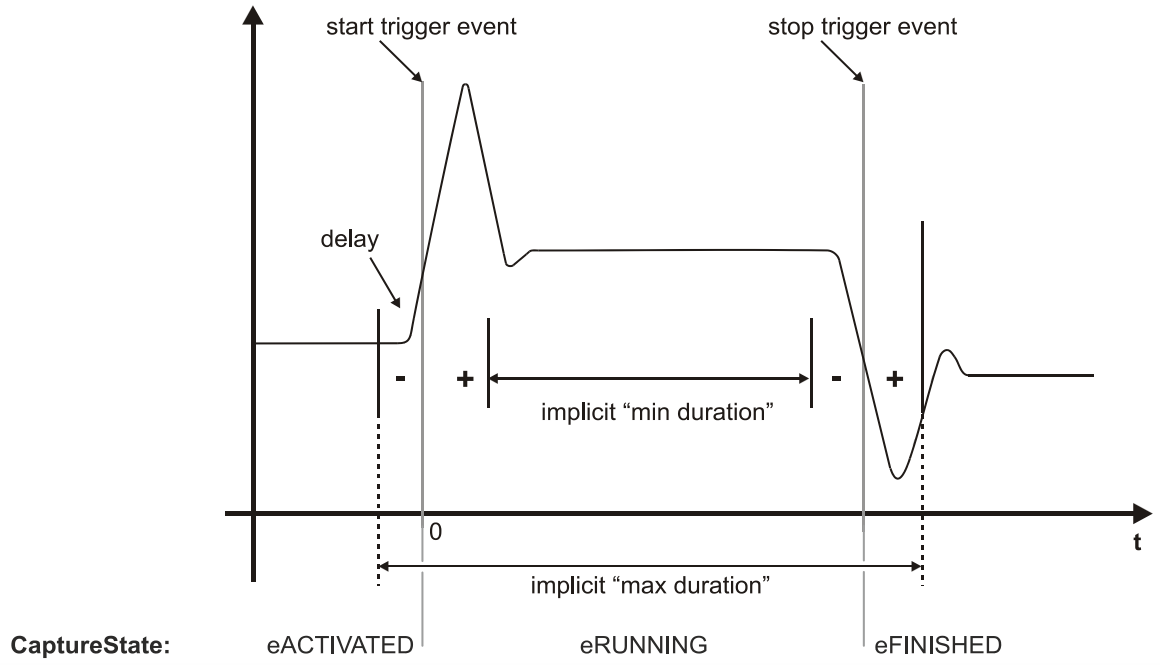


Figure 36: Start and Stop Trigger used

If the delay for the start trigger is positive, the capturing starts at the specified amount of time after the start trigger became true – or in case no start trigger has been specified, the capturing starts the specified amount of time after the `Start()` method has been called. If the delay for the start trigger is negative, the capturing starts the specified amount of time before the start trigger occurred, i.e. the capture result will contain even values before the start trigger became true. Obviously, this case is limited: It is not possible to obtain measured values which occurred before the call of the `Start()` method.
If the delay for the stop trigger is positive, the capturing stops the specified amount of time after the stop trigger occurred (or `Stop()` is called resp.). If it is negative, it stops the specified amount of time before, i.e. the capture result will not contain the measured values that occurred during the delay time before the stop trigger occurred (or `Stop()` is called resp.). This results into the “implicit min duration” or “implicit max duration” of the capturing (see Figure 36).

Note: The occurrence of the start trigger event indicates the zero-time of the capture result. If no start trigger is given, the occurrence of the Capture.Start() method call indicates the zero-time of the capture result.

4.1.5.3 State Diagram of Capturing

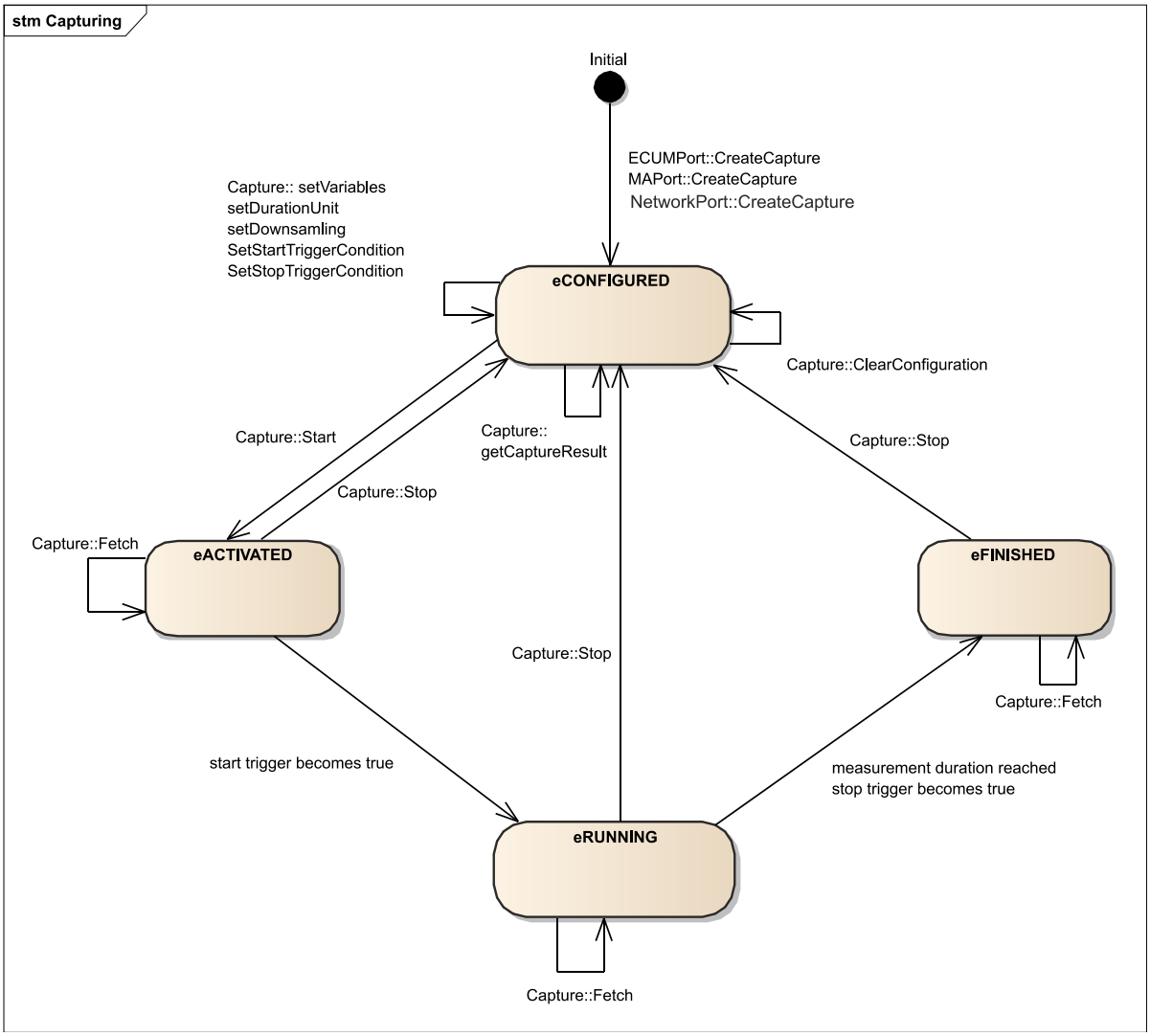


Figure 37: Capturing state diagram

eCONFIGURED

After creation, a Capture object is in state `eCONFIGURED`. In this state, the capturing is defined / configured.
It remains in this state, until the `Start` method is called, which changes the state to `eACTIVATED`.

eACTIVATED

In this state, the Capture waits until the start trigger condition becomes true and then switches into `eRUNNING`. If no start trigger is set, it swichtes immediately into `eRUNNING` right after entering `eACTIVATED`.
A `Stop` method call switches into `eCONFIGURED`.

eRUNNING

While residing in this state, data is acquired. It remains in this state, until

- a `Stop` method call switches into `eCONFIGURED`
- the stop trigger becomes true and `Retriggering` == 0:
  it switches to eFINISHED
- the stop trigger becomes true and `Retriggering` <> 0:
  it switches to eACTIVATED and is again waiting for the next start trigger

eFINISHED

This state is reached as soon as the last stop trigger condition becomes true according to the retriggering definition.
It remains in this state, until a `Stop` method call switches into `eCONFIGURED`.

In case of Retriggering == -1, this state cannot be reached, since this changes between eACTIVATED and eRUNNING.

4.1.5.4 Capture Result

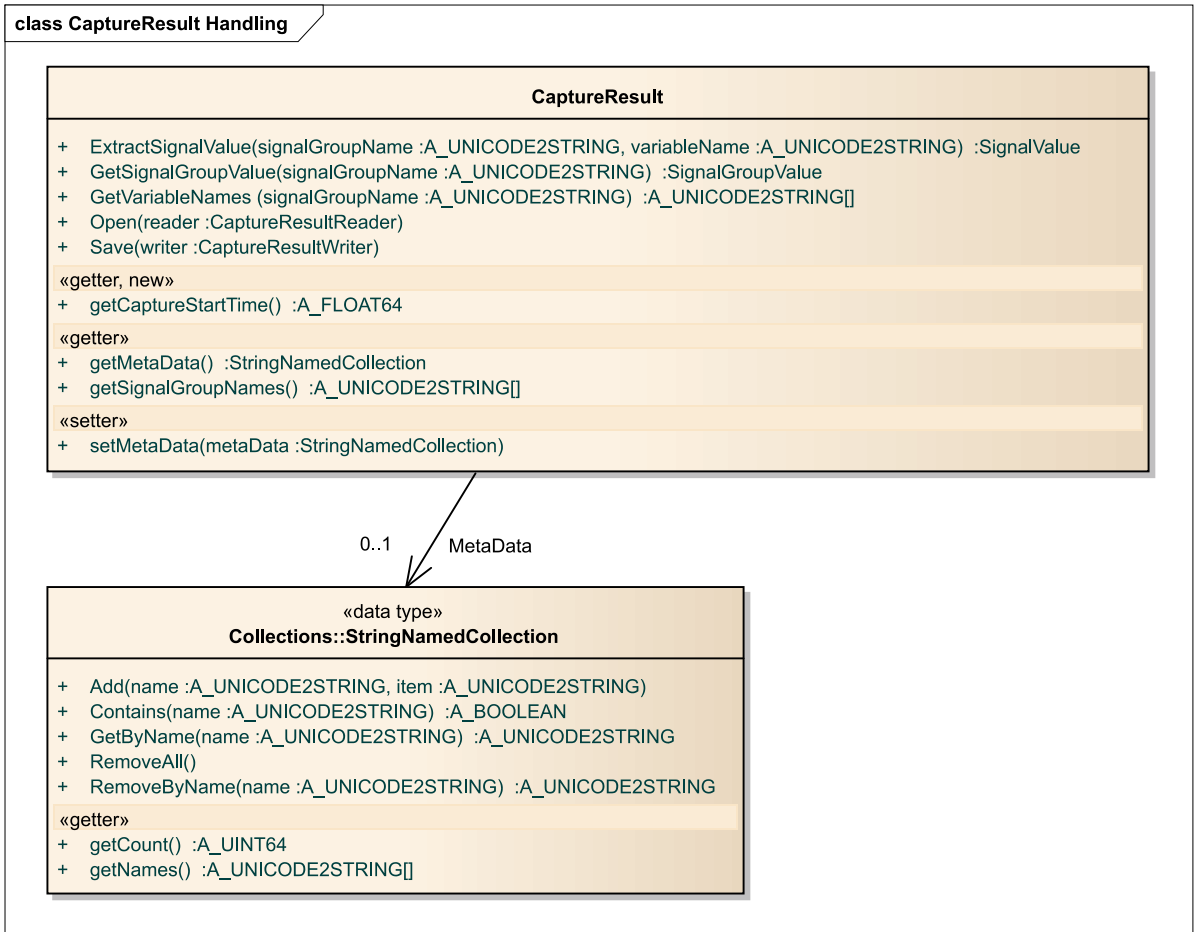


Figure 38 Capture results

CaptureResult

A `CaptureResult` object holds the data acquired by a Capture object. It provides access to objects of type `ValueContainer::SignalGroupValue` which holds the sampled data.
Capture results can contain 1 to n Signal Groups. This is possible, because Capture results can be created by Capturing itself or reading MDF files. Another possibility is the capturing on multi processor platforms. In such a case for each processor an own value acquisition takes place. This will be mapped on separate signal groups. Each signal group have independent start and stop time points.
SignalValue and SignalGroupValue contain time stamp / axis (time lane) and separately the function value(s) [for each variable an own data lane]. Each Signal Group value has its own time axis.

MetaData

Via the MetaData association, additional information can be added to the `CaptureResult`.

4.1.5.5 Document Handling for Capture Data

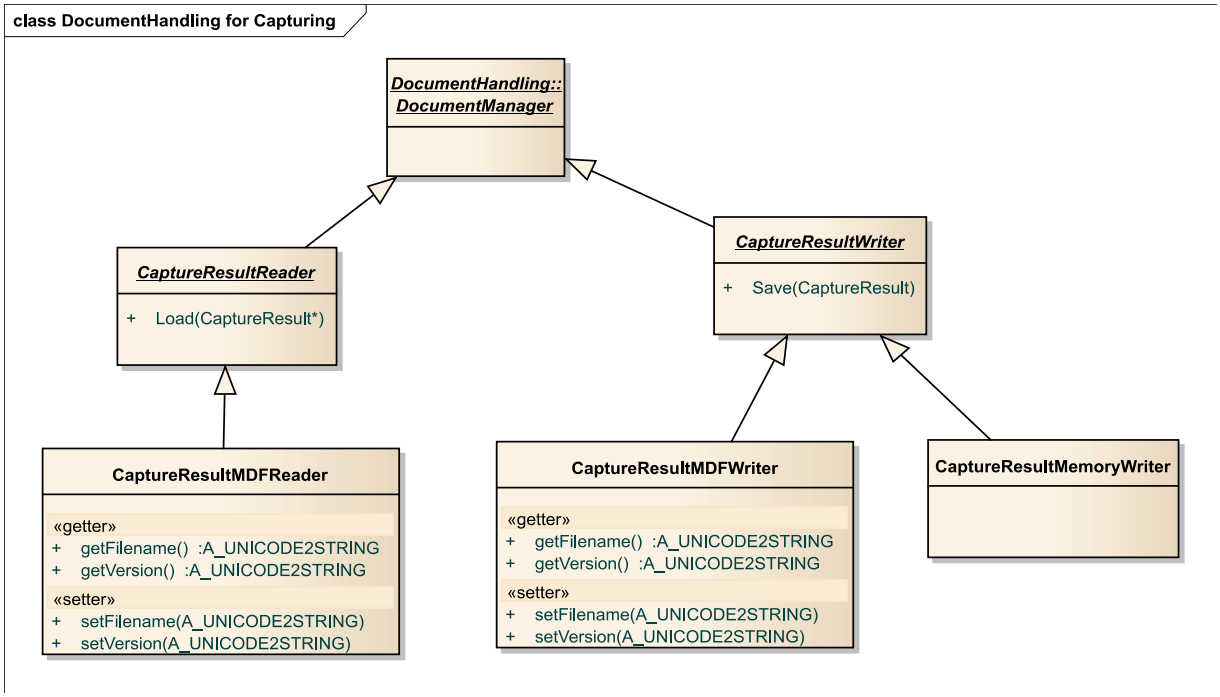


Figure 39: Document Handling

CaptureResultReader & CaptureResultWriter

These classes represent abstract super classes for the concrete reader and writer classes. They provide a Load or a Save method in order to load and to save `CaptureResult` objects.

CaptureResultMDFReader

This class handles the loading of MDF files [3]. The loaded data structure is stored in a `CaptureResult` object.
The used MDF format is identified by a `Version` property. If the version is not selected, the version of the file which shall be load is determined and shall be used. The determined version is set into the version property. In case the determined version is not supported by the user an exception will be thrown.

CaptureResultMDFWriter

This class handles the saving of `CaptureResult` objects compliant to the MDF format.
The used MDF format is identified by a version property. If the version is not selected, the highest by vendor supported version will be used. If selected version is supported by the vendor, captured data will be stored in the specified format, otherwise an exception will be thrown.

CaptureResultMemoryWriter

In case a `CaptureResult` is not stored in the file system during the capturing process, an object of the `CaptureResultMemoryWriter` class is used as writer instance. Instead of streaming the capture to disk, the `CaptureResult` is held in the RAM.

Note: `CaptureResultMDFWriter` and `CaptureResultMDFReader` uses a version property for the identification of the used MDF format. The following syntax is used for version identification:
`Major.Minor.[Maintenance]`
For the different items a sequence of numbers is used. Each reader and writer can support different versions of the MDF file format.

4.1.5.6 Usage of Capturing

Capturing with Watcher

The following sequence diagrams Figure 40, and Figure 41 show how to use capturing. First, a Capture object has to be configured: Here, the instance of the capture object is created by an instance of the `MAPort`. Then, all variables that shall be captured are added to the capture's list of variables. To define the beginning and the end of the capture, two watcher objects are created. For a simple human understanding of the trigger conditions, defines are created. A define relates a name to the path of a model variable. These names are used in the conditions of the watcher objects. Finally, the watcher objects are set as start and stop triggers for the Capture object during configuration.

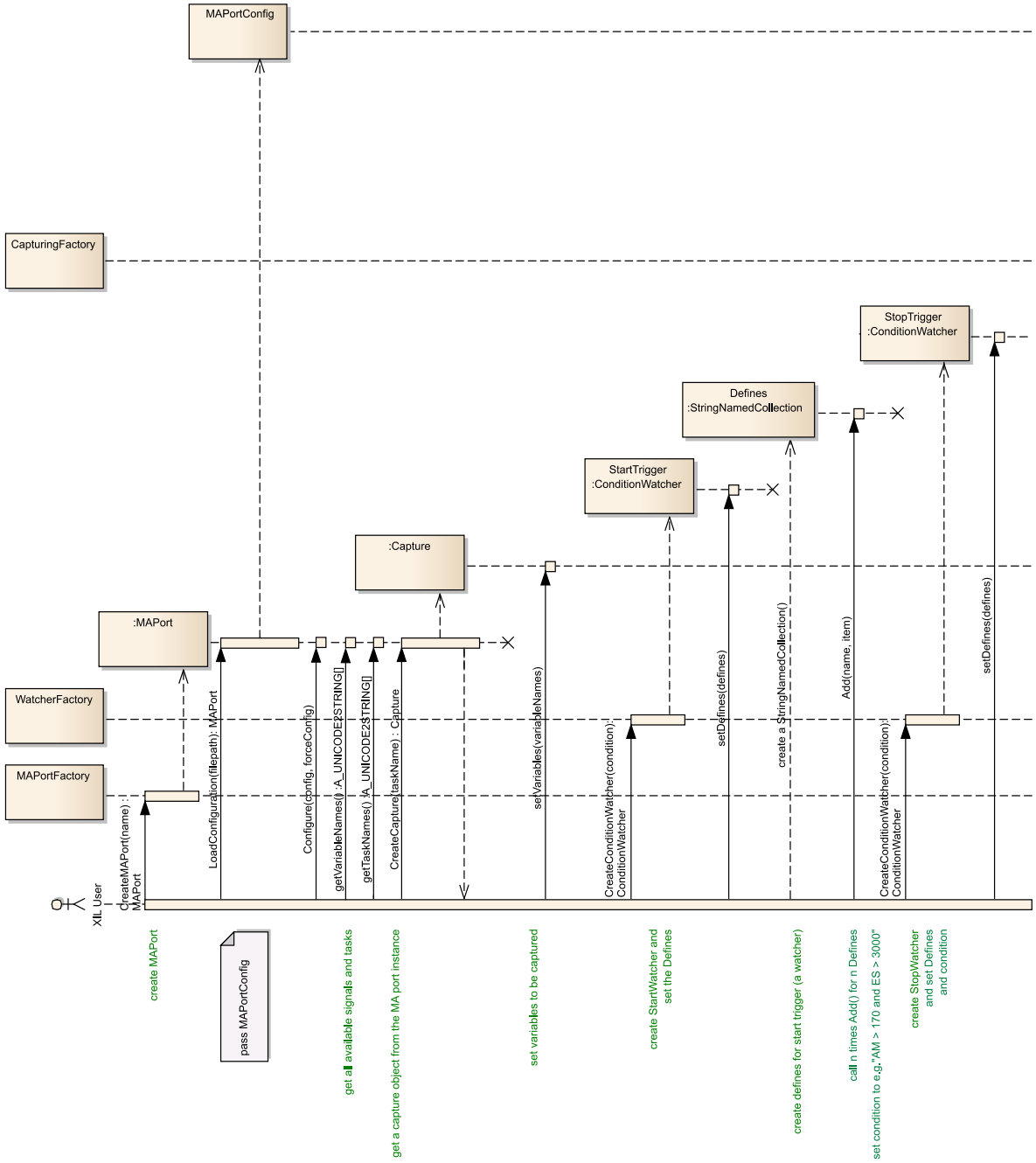


Figure 40: Usage of capturing with Watcher (Part 1)

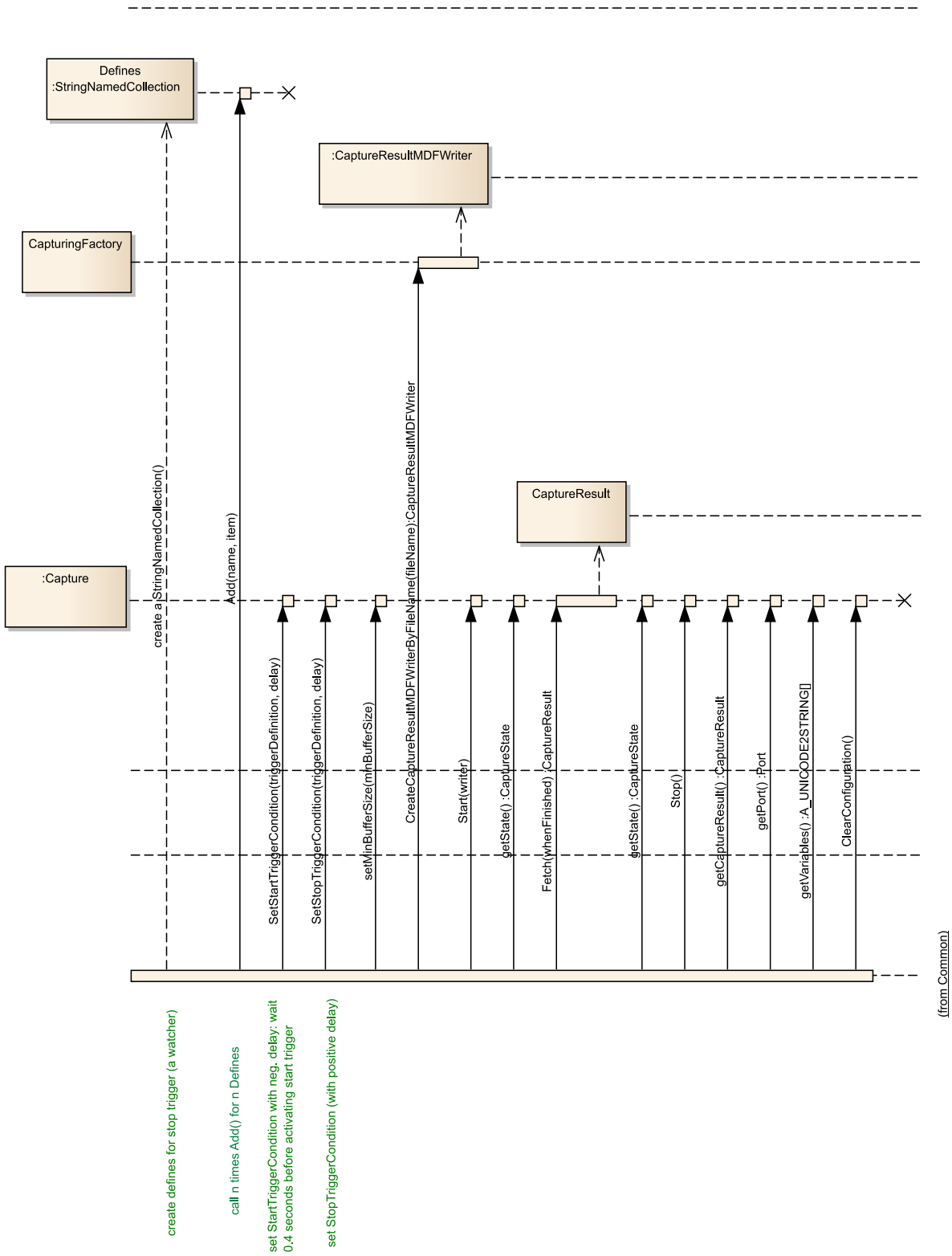


Figure 41: Usage of capturing with Watcher (Part 2)

4.2 MODEL ACCESS PORT

4.2.1 USER CONCEPT

4.2.1.1 General

The Model Access port is the central point for managing access to the model, simulated on the XIL simulator. This port provides functionality for read- and write-access to the model, to set up capturing and stimuli, and to manage model variables.
When using this port, it is required that all initialization of the XIL simulator, like download and start of the model, has been done previously.
The ModelAccessPort-package is related to to the packages "Common:Capturing", "Common:SignalGenerator" and "Common:CaptureResult". The two latter ones are not sub-packages of `ModelAccess` as they are also used by the `ECUMPort and` the `ECUCPort`.

4.2.1.2 Model Access Port

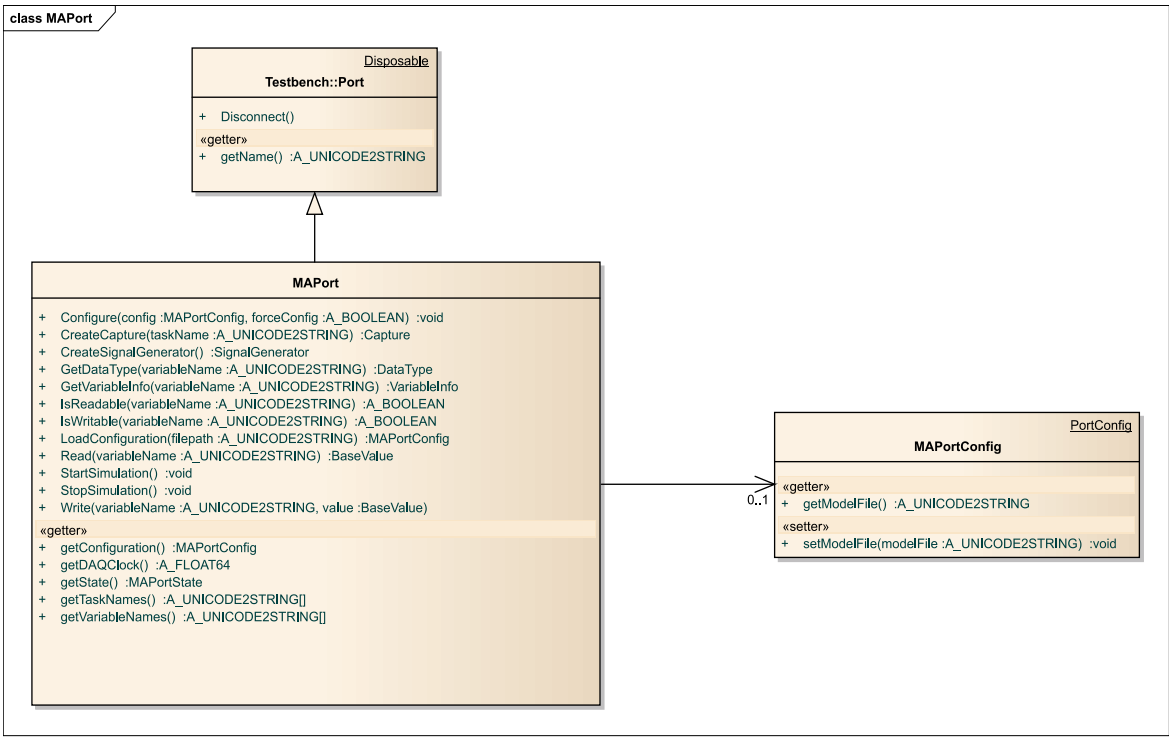


Figure 42: Model Access Port

Class MAPort

On the one hand, this class provides general functionality like for example functionality to get information about available model variables, their readability and writeability and to read and write model variables. On the other hand, it provides initialization functionality to `CreateCapture` and `SignalGenerator` instances.

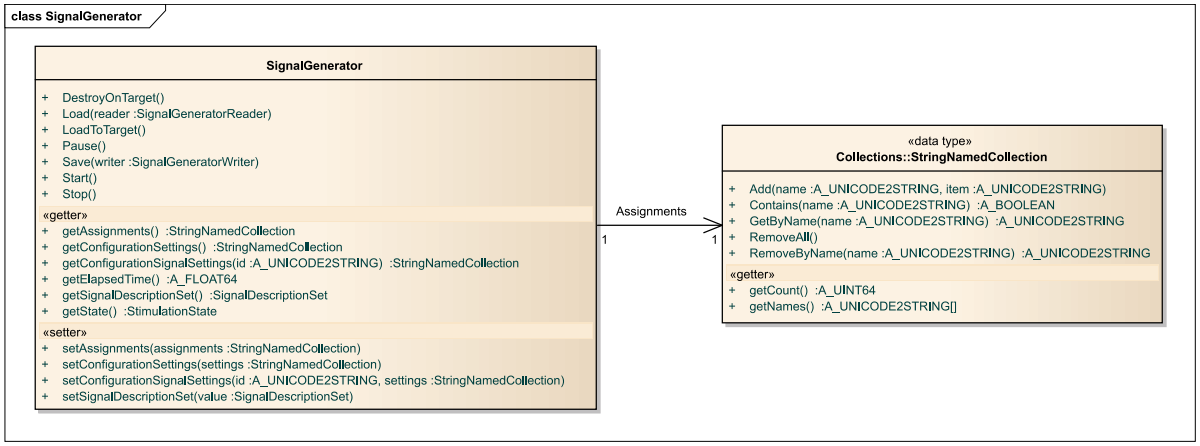


Figure 43: Signal Generator

Class SignalGenerator

A `SignalGenerator` defines stimuli and manages their execution. For the definition of a stimulus, a `SignalDescriptionSet` is referenced by the `SignalGenerator`. The signals from the `SignalDescriptionSet` are assigned with model variables in the "Assignments" collection. For the management of the stimulus, functionality is provided for downloading the stimulus to the XIL simulator, for starting, stopping, and pausing it and for observing its current state.

4.2.1.3 Document Handling

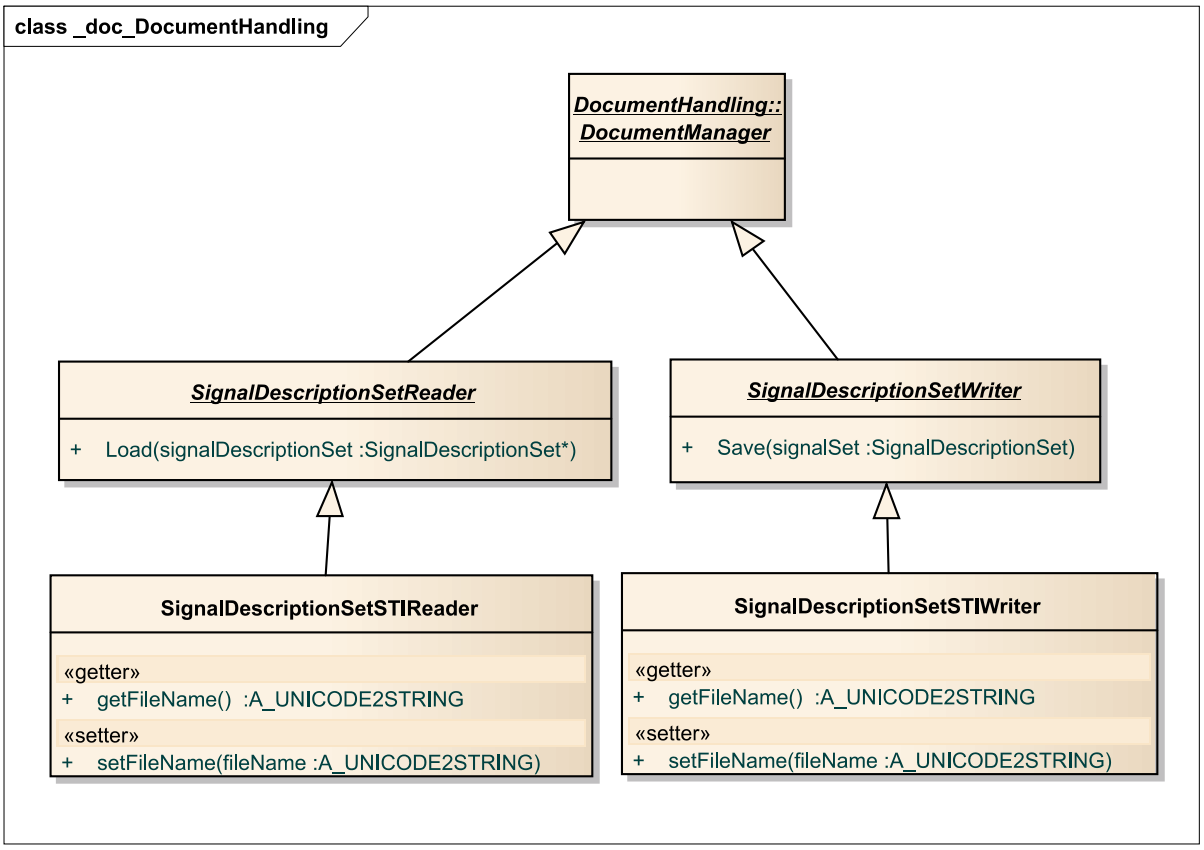


Figure 44: Document Handling

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

This class handles the loading of a `SignalGenerator` object stored in a STI files. STI is a file format for `SignalGenerator` objects which is also part of the XIL API standard. The loaded data structure is stored in a `SignalGenerator` object.

SignalGeneratorSTIWriter

This class handles the saving of `SignalGenerator` objects in STI format. STI is a file format for `SignalGenerator` objects which is also part of the XIL API standard.

4.2.1.4 States of the MAPort

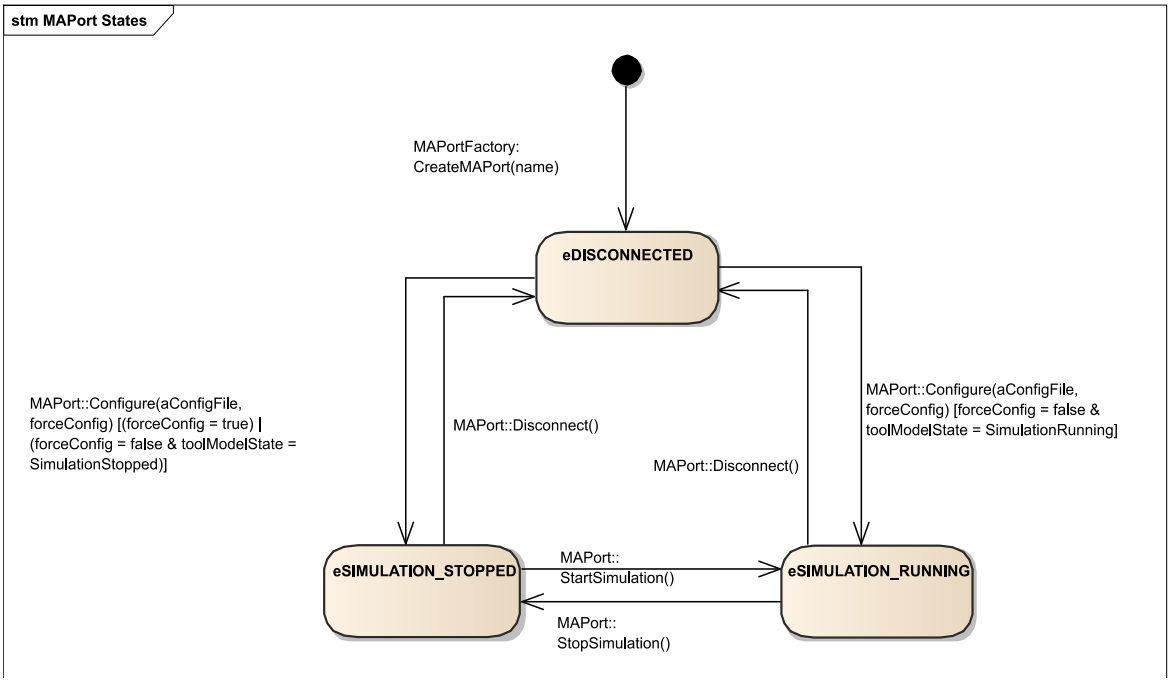


Figure 45: MAPort state diagram

Figure 45 shows the state diagram of the `MAPort`. There are three states, `eSIMULATION_RUNNING`, `eSIMULATION_STOPPED`, and `eDISCONNECTED`. After creation, the `MAPort` instance is always in state `eDISCONNECTED`. Following the states are explained:

Table 18 State of the MAPort

State	Description
`eDISCONNECTED`	The port cannot be used for simulation purposes because there is no connection to the hardware

eSIMULATION_RUNNING	Calculation of simulation model is running, Capturing and Signal Generation is possible
eSIMULATION_STOPPED	Calculation of simulation model is not running.

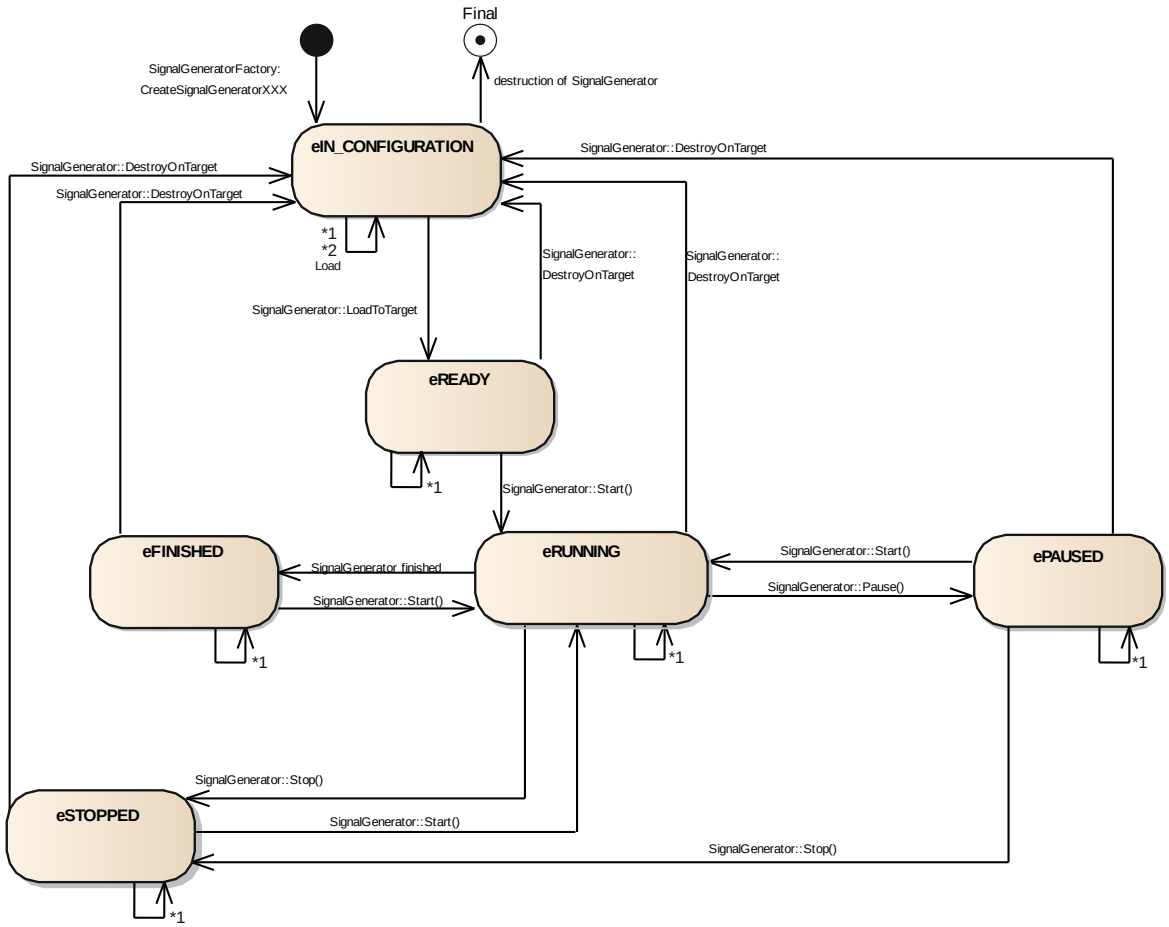
Table 19 MAPort states

	eDISCONNECTED	eSIMULATION_STOPPED	eSIMULATION_RUNNING
Method Configure	x		
Method CreateCapture		x	x
Method CreateSignalGenerator		x	x
Method Disconnect		x	x
Method Dispose	x		
Method GetDataType		x	x
Property getState	x	x	x
Property getTaskNames		x	x
Property getVariableNames		x	x
Method IsReadable		x	x
Method IsWritable		x	x
Method Read			x
Method StartSimulation		x	
Method StopSimulation			x
Method Write			X

State transitions are only successful, if all preconditions are fulfilled and no error occurs during the transition. Otherwise the previous state is not changed.
Methods, which trigger a state change, will throw an exception if the state change could not be processed successfully.

4.2.1.5 States of the SignalGenerator

The state of the `SignalGenerator` class can be queried at any time by the method `SignalGenerator.getState()`.



*1:
get Assignments
get ConfigurationSettings
get ConfigurationSignalSettings
get Elapsed Time
get SignalDescriptionSet
get state
Save

*2:
set Assignments
set ConfigurationSettings
set ConfigurationSignalSettings
set SignalDescriptionSet

Figure 46: Signal generator state diagram

Table 20 SignalGenerator states

	eIN_CONFIGURATION	eREADY	eFINISHED	eRUNNING	ePAUSED	eSTOPPED
Method DestroyOnTarget		X	X	X	X	X
Property getAssignments	X	X	X	X	X	X
Property getConfigurationSettings	X	X	X	X	X	X
Property getConfigurationSignalSettings	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 LoadToTarget	X					
Method Pause				X		
Method Save	X	X	X	X	X	X
Property setAssignments	X					
Property setConfigurationSettings	X					
Property setConfigurationSignalSettings	X					
Property setSignalDescriptionSet	X					
Method Start		X	X		X	X
Method Stop				X	X	

State transitions are only successful, if all preconditions are fulfilled and no error occurs during the transition. Otherwise the previous state is not changed.
Methods, which trigger a state change, will throw an exception if the state change could not be processed successfully.

eIN_CONFIGURATION

After creation, a `SignalGenerator` object is in state `eIN_CONFIGURATION`. In this state, the signal generation is defined / configured. Usually, it is loaded to the XIL simulator target, when configuration has been done.

eREADY

After loading a defined signal description set to the XIL Simulator target, the `SignalGenerator` object is in state `eREADY`. In this state, it waits for being started.

eRUNNING

After starting the signal generation, the `SignalGenerator` object is in state `eRUNNING`. In this state, the model variables are stimulated by the actual signals as defined.

eFINISHED

If signal generation is finished, this state is entered.

ePAUSED

Signal generation can be paused. In this case, state `ePAUSED` is entered. Leaving this state the signal generation resumes, and does not start at beginning.

eSTOPPED

If the signal generation is stopped, this state is entered.

4.2.2 USAGE OF THIS PORT

In this chapter, the usage of this port is described by means of some examples. It is shown how to set up the port, how to read and write model variables and how to set up a signal generator or a capture. How to use captures is shown in the chapter Data Capturing.

4.2.2.1 Creation and Configuration

Instances of `MAPort` are created by calls to method `CreateMAPort` of a `MAPortFactory` object. It returns a `MAPort` instance with the name given as parameter. The `MAPortFactory` object can be obtained from the `Testbench` object that provides a corresponding property.

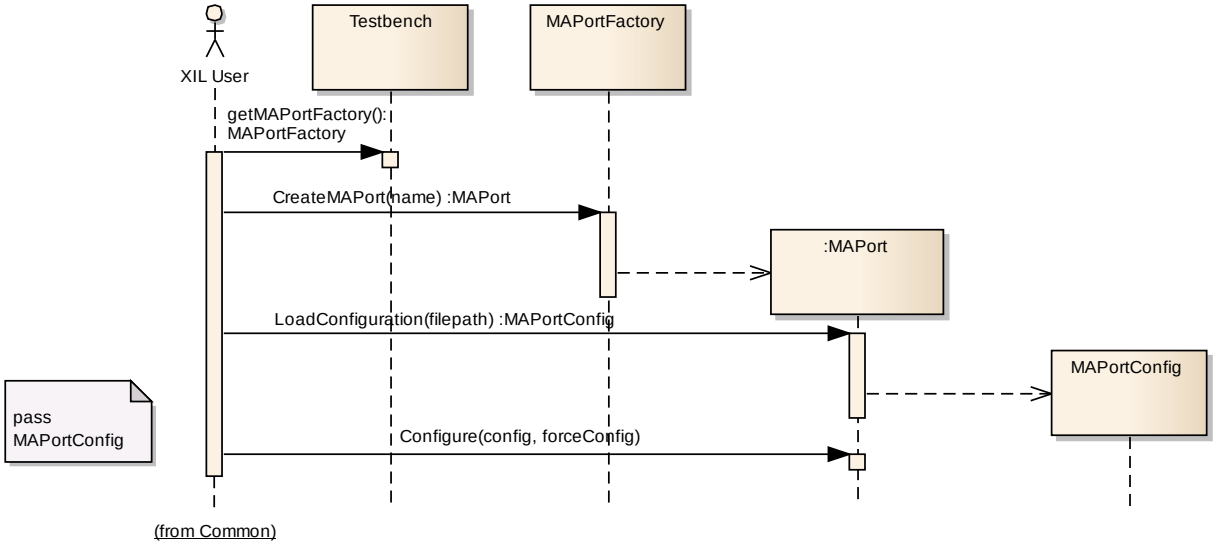


Figure 47: Process of MAPort creation and configuration

Configuration of the new port instance is a two-step process (see Figure 47). First a vendor specific configuration file has to be loaded via method `LoadConfiguration`. Besides other vendor specific port settings this file particularly specifies the model file to be executed on the simulator. The model file path can be queried from or changed by a corresponding property of the `MAPortConfig` object that is returned by `LoadConfiguration`.

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 hardware 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`. Afterwards the simulation model is ready for execution. This behaviour can also be forced by setting the `Configure` method's parameter `forceConfig` to true. In this case a running simulation model will be stopped and replaced by the specified one. If the `forceConfig` parameter is false an already loaded simulation model will not be replaced even if different from the specified one. Furthermore a running simulation will not be stopped, so simulation state will remain `eSIMULATION_RUNNING` in this case.

4.2.2.2 Reading & Writing Model Variables

The sequence diagram in Figure 48 depicts how to handle and how to access model variables: First, an instance of the model access port is created. When such an instance has been created, it is assumed that the XIL simulator has been initialized and a simulation model is running. The instance of the `MAPort` is used to request all available model variables and all tasks (timing raster) existing in the simulation. A Capture object is created by the `MAPort` instance with a raster specified by one of the existing tasks.

Note: Due to the fact that the adaptation of a simulation model for a specific XIL simulator is vendor specific:

- the list of available variable names of an initialized MAPort is vendor specific (see `VariableNames` property)
- readability and writability of model variables can be different on different XIL simulators (see `IsReadable` and `IsWritable` methods)
- data type of model variables can be different on different XIL simulators (see `GetDataType` method)
- the list of task names of an initialized MAPort is vendor specific (see `TaskNames` property)
- the timing raster of the tasks provided by an initialized MAPort is vendor specific and can be independent of the step-size of the simulation model.

The standard doesn't define any timing constraints for the read and write accesses of model variables. It is assumed that the read and write accesses are executed synchronously.

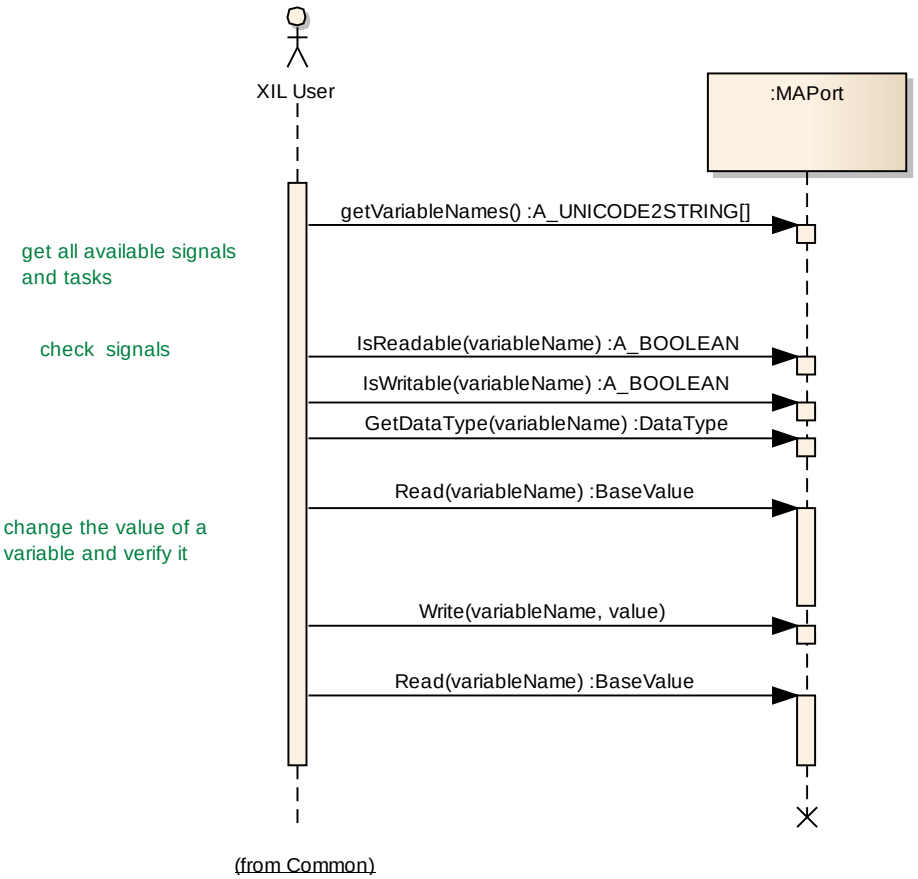


Figure 48: Model AccessPort example

In order to stimulate model variables by signals, a `SignalGenerator` instance is required that is also constructed by the `MAPort` object. Existing signal descriptions can be loaded (see chapter Stimulating Model Variables for details). The usage of the `Capture` and the `SignalGenerator` instances is described in chapter Data Capturing and in chapter Stimulating Model Variables.
Before accessing a model variable, the `MAPort` instance can check if the variable is readable or writeable (or both) and of which data type it is. Finally the variable is accessed by the `Read()` and the `Write()` method of the `MAPort` object.

4.2.2.3 Stimulating Model Variables

How to stimulate model variables by signals is depicted in Figure 49 and Figure 50. As described in the previous chapter, instances of class `MAPort` and of class `SignalGenerator` need to be created beforehand. Using a `SignalGeneratorSTIReader` object, existing signals are loaded as described in chapter Document Handling. An example for such signals is presented in chapter Signal Description.

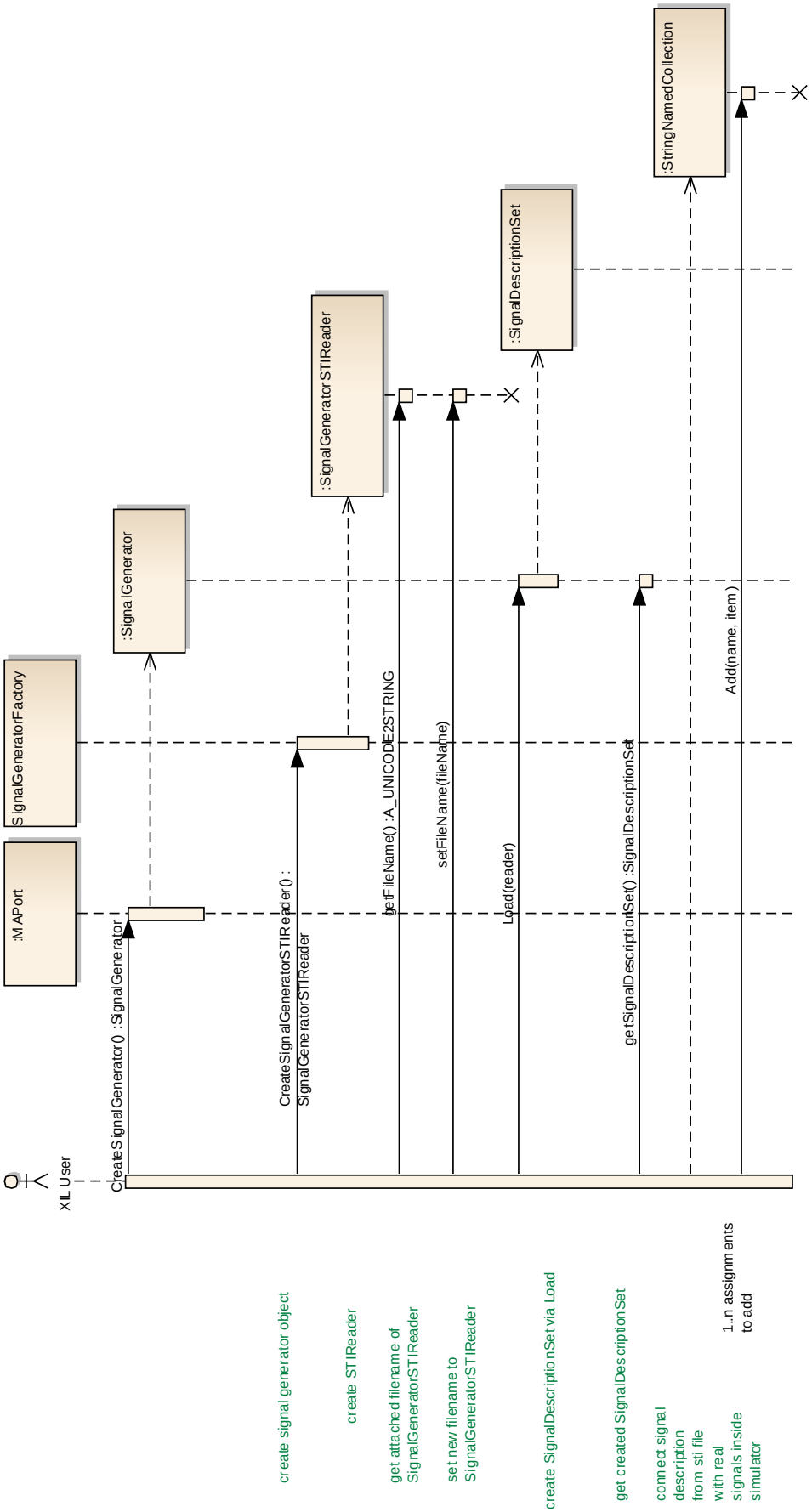


Figure 49: SignalGenerator example (part 1)

After executing the `Load()` method, signal descriptions are referenced by the `SignalGenerator` via a `SignalDescriptionSet` object. It may be that the file contains already information that assigns the signals to model variables. In this case the signal generator is configured after loading. Otherwise, these assignments are specified by adding name-item pairs to the Assignments-Collection of the `SignalGenerator` object. The name is one of the signal names, the item is the model path of the variable to be stimulated.

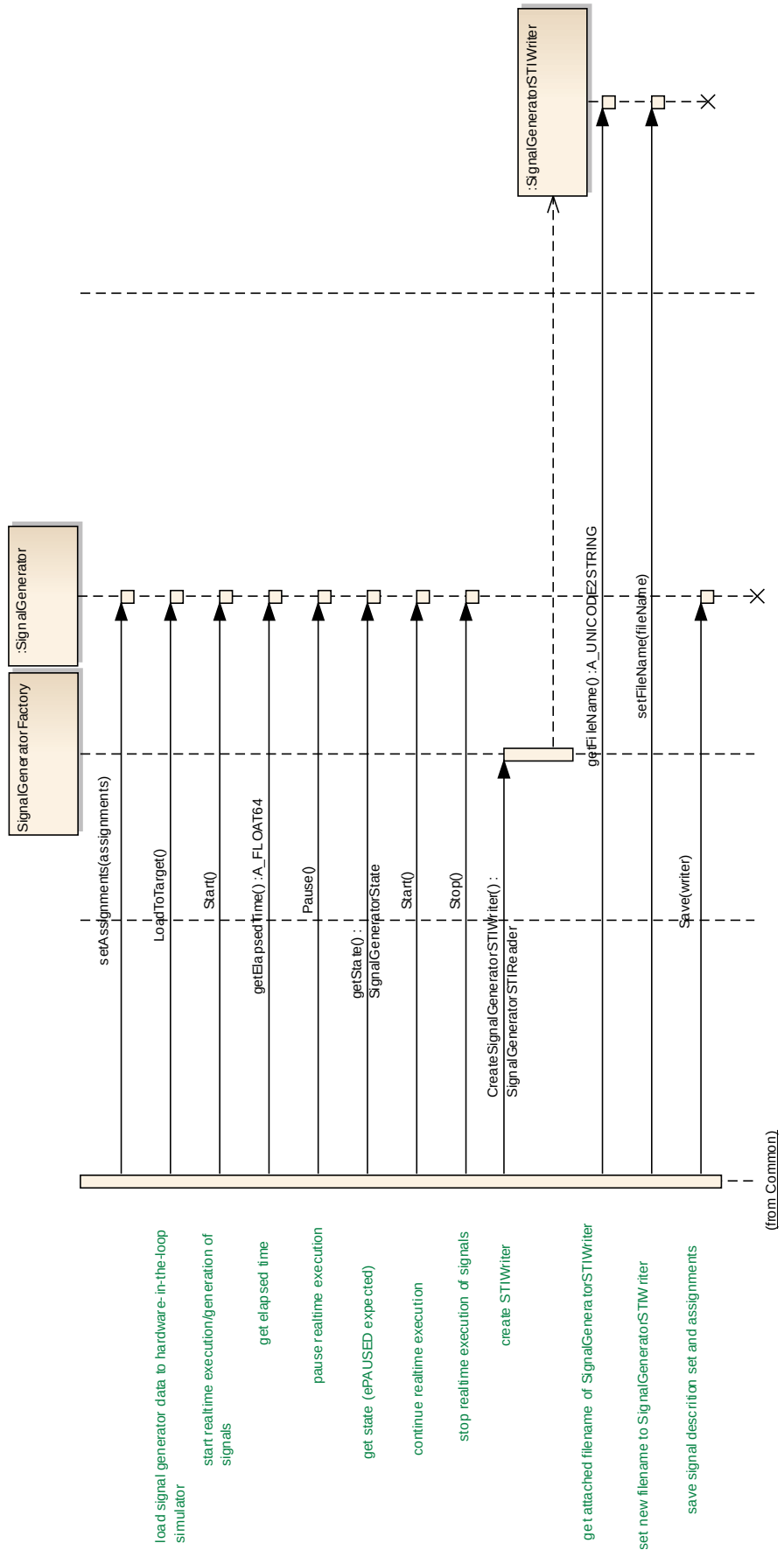


Figure 50: SignalGenerator example (part 2)

After setting these assignments, the stimulus is configured. The next step is to load the stimulus down to the target which is usually the XIL simulator. Then, the stimulus can be started, paused, and stopped by calling the corresponding methods. Further, the user can ask for the current state of the signal generator using the `State()` property. Finally, the signal generator object can be saved including the new assignments, using a `SignalGeneratorSTIWriter` as described in Stimulating Model Variables.

5 Symbols and Abbreviated Terms

(D)COM	(Distributed) Component Object Model
AE	Automotive Electronics
API	Application Programming Interface
ASAM	Association for Standardisation of Automation and Measuring Systems
CAN	Controller Area Network
DLL	Dynamic Link Library
DTC	Diagnostic Trouble Code
ECU	Electronic Control Unit
EEPROM	Electrically Erasable Programmable Read Only Memory
EES	Electrical Error Simulation
FIU	Failure Injection Unit
FMI	Functional Mock-up Interface
GES	General Expression syntax
GPIB	General Purpose Interface Bus
HIL	Hardware In the Loop
LIN	Local Interconnect Network
PC	Personal Computer
RS232	Recommended Standard 232 (standard for serial binary data signals)
sti	stimulus description file (XML format)
stz	stimulus description file (zip archive)
SUT	System Under Test
TA	Test Automation
TCP/IP	Transmission Control Protocol/Internet Protocol
UML2	Unified Modeling Language Version 2
XIL	Generic Simulator Interface
XML	Extensible Markup Language
XSD	XSD XML Schema Definition

6 Bibliography

[1] ASAM e.V.: ASAM Data Types; 2005

[2] ASAM e.V.: ASAM General Expression; 2011

[3] ASAM e.V.: ASAM Measurement Data Format; 2012

[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; 2013;

[6] ASAM e.V.: Python API Technology Reference Mapping Rules; 2013;

[7] ASAM e.V.: Modeling Guidelines

[8] ASAM e.V.: ASAM XIL; 2013

Appendix A. SYNTAX OF WATCHER CONDITIONS

In the ASAM XIL API the General Expression Syntax is used for defining watcher conditions. Not all possible functions and operators of the ASAM General Expression [2] are required. This document defines the subset of ASAM Expressions supported by the XIL API V2.0.0.

Table 21 Operators and Functions supported by XILAPI V 2.0.0

Semantic	Syntax and Arguments	XILAPI V2.0.0
Sequential evaluation of several trigger conditions: when the left hand side condition evaluates to true, the evaluation of the right hand side condition starts (and continues even if the left hand side condition 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 (exclusice 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 or equal	expr1 <= expr2	✓
Greater or equal	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	✓

Semantic	Syntax and Arguments	XILAPI V2.0.0
Positive sign; has no effect, just to show a positive number like in C	+ expr	✓
Logical not	! expr	✓
Bitwise not	~ 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 (pow(a,b) → a^b)	pow(expr1, expr2)	✓
Power operator (a**b) → 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 the signal defined by the variable changes from a value lower than threshold to a value greater or equal than threshold	posedge(expr1, expr2Threshold)	✓
Detection of negative edge: returns true when the value of the signal defined by the variable changes from a value higher than threshold to a value lesser or equal than threshold	negedge(expr1, expr2Threshold)	✓

Semantic	Syntax and Arguments	XILAPI V2.0.0
Detection of positive edge: returns true when the value of the signal defined by the variable changes from a value lower than threshold to a value greater than threshold	strictposedge(expr1, expr2Threshold)	-
Detection of negative edge: returns true when the value of the signal defined by the variable changes from a value higher than threshold to a value lesser than threshold	strictnegedge(expr1, expr2Threshold)	-
Detection of value change; change is detected when difference between current number and its direct successor (number in the last evaluation step) is greater or equal than given delta	changed(expr1, expr2Delta)	✓
Detection of hardware trigger	hwtrigger()	-
Detection of manual trigger	mantrigger()	-

A.1. OTHER RESTRICTIONS

- Bit operations like shift are not defined.
- The first parameter of the functions posedge, negedge, and changed must be only variables, not expressions.
- The number of significant initial characters of an identifier is 32.

A.2. SYNTAX OVERVIEW

The following syntax defines the subset of the ASAM GES that can be used in XIL API watcher conditions:

```
XIL-API-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 (exclusice 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

primary-expression:
        identifier
        constant
        ( conditional-expression )
```

Appendix B. KEY VALUE PAIRS

Inside the Meta Data Key Value Pairs can be used. An overview about currently defined Keys / Value pairs is shown in Table 22.

Table 22 Overview about reserved Key Value Pairs

Key	Usage area	Value	Description
StartTriggerTimeOut	Meta data of capture results	TRUE FALSE	activation by Time Out activation by Trigger Condition
StopTriggerTimeOut	Meta data of capture results	TRUE FALSE	activation by Time Out activation by Trigger Condition

Figure Directory

Table Directory

Association for Standardisation of
Automation and Measuring Systems

E-mail: support@asam.net
Web: www.asam.net