

IAV Merida Analyzer

Manual



IAV Merida Analyzer

Version

This manual is for version:

Version: 2.2606.1594033.ca11d580.feature-merida-4551-azure-security

Build Number: 0

Build Date: 2026-05-26 08:44:06

You can find the version in the help menu.

Company

IAV

Carnotstraße 1

10587 Berlin

Tel.: +49 30 3997-80

Internet: www.iav.com

Contact

Merida Support

Tel: +49 5371 80-53518

E-Mail: merida-support@iav.de

Contents

1	Introduction	6
2	Functionality	6
2.1	Preliminary note	6
2.2	Quick start	6
3	Integration into the MessdatenPlattform	8
4	TKO file	8
4.1	Alias	9
4.2	DBCChannelSwap	10
4.3	Graphics generation	10
4.3.1	Event-based	10
4.3.2	Static	11
4.4	Constants	11
4.5	Control commands	12
4.6	Listen	12
4.7	Mathematical operations	13
4.8	Strings	13
5	SCR file	13
6	The context	14
7	Attributes in the measurement data platform	18
7.1	AttributeGetter	18
7.2	AttributeSetter	19
8	File formats	19
8.1	BluePiraT (OTMT/XTMT/TMT)	19
8.2	CSV	20
8.3	CaTe (CLG)	20
8.4	DBCs	21
8.5	Drive Recorder TCS (PST)	21
8.6	Flea-CSV (LOG)	21
8.7	G.i.N. logger (GJF)	21
8.8	G.i.N. logger (ML)	22
8.9	MDF 4.0 (MF4)	23
8.10	Matlab (MAT)	23
8.11	Measurement Data Format (MDF)	23
8.12	PCAPNG	23
8.13	RPC 3	23
8.14	Statistical Data Message (LDC)	24
8.15	UniCAN2/3 (CSM)	24
9	Data and message types	24
9.1	FlexRay messages	24
9.2	Signal name conventions	25
10	Evaluators	25
10.1	ADTFExportEvaluator	26
10.2	ATFXClassificationExportEvaluator	26
10.3	General information	27
10.4	ByteStream	27
10.4.1	ByteStreamMultiplexer	27
10.4.2	ByteStreamToFileEvaluator	30
10.4.3	ByteStreamToStringEvaluator	30
10.4.4	CanToByteStreamEvaluator	31
10.4.5	GNLogGeneratorEvaluator	31
10.5	CAN	32

10.5.1	BAPEvaluator	32
10.5.2	CANBusLoadEvaluator	33
10.5.3	CANCheckEvaluator	34
10.5.4	CANMessageTimeoutEvaluator	35
10.5.5	FrameFilterEvaluator	36
10.6	Diagnosis	36
10.6.1	DiagMessageEvaluator	36
10.6.2	DiagMessageEvaluatorOdx	37
10.6.3	FIBEX AUTOSAR	37
10.6.4	ISOCan	38
10.6.5	IdexExport	39
10.6.6	KWP2000	39
10.6.7	TP16	40
10.6.8	TP20	40
10.6.9	UDS	41
10.6.10	UDS (ODX)	41
10.7	Ethernet	42
10.7.1	DoIPFilterEvaluator	42
10.7.2	EthernetFilterEvaluator	43
10.7.3	EthernetTrafficEvaluator	44
10.7.4	NDPFilterEvaluator	46
10.7.5	PTPFilterEvaluator	46
10.7.6	SomelIPFilterEvaluator	46
10.7.7	TCPCheckEvaluator	48
10.7.8	ViWiFilterEvaluator	49
10.8	Export	49
10.8.1	DataExtractEvaluator	49
10.8.2	DataExtractRPC3Evaluator	51
10.8.3	DataExtractSimpleEvaluator	51
10.8.4	FileExportEvaluator	52
10.8.5	FilePassThroughEvaluator	55
10.8.6	LogbookEvaluator	56
10.8.7	MLClassificationExportEvaluator	58
10.8.8	ReducedSignalExportEvaluator	59
10.8.9	ResultFileListExportEvaluator	59
10.8.10	SignalMetaDataExportEvaluator	60
10.8.11	SignalMonitorEvaluator	60
10.8.12	SignalToTableEvaluator	62
10.9	Extras	68
10.9.1	FrameMultiplexEvaluator	68
10.9.2	HandleSignalDuplicationEvaluator	68
10.9.3	MeasurementInfoEvaluator	69
10.9.4	SignalValidityCheckEvaluator	69
10.9.5	TwoPassEvaluator	70
10.10	FuzzyEvaluator	71
10.11	GPS	74
10.11.1	ReverseGeocodingEvaluator	74
10.11.2	RouteToDBEvaluator	75
10.12	GateEvaluator	75
10.13	GateRealTimeEvaluator	76
10.14	Geometry	77
10.14.1	GeometryCalculationEvaluator	77
10.14.2	GeometryObjectEvaluator	77
10.15	Graphic	78
10.15.1	PreviewGraphicCreator	78
10.15.2	StaticGraphicCreator	79
10.16	MapEvaluator	79
10.17	LIN	81
10.17.1	LINMessageTimeoutEvaluator	81
10.18	LogHandlingEvaluator	82
10.19	Logging	82

10.19.1	BinLoggingEvaluator	82
10.19.2	FrameLoggingEvaluator	83
10.19.3	SignalLoggingEvaluator	85
10.20	MeasurementAnalysisEvaluator	88
10.21	MinMaxEvaluator	89
10.22	PreprocessingEvaluator	90
10.23	Programmable	91
10.23.1	FunctionEvaluator	91
10.23.2	JavaEvaluator	97
10.24	RealTimeEvaluator	100
10.25	Controller	102
10.25.1	Schwingen	102
10.26	Resampling	102
10.26.1	GroupResamplingEvaluator	102
10.26.2	ResampleEvaluator	103
10.26.3	ResampleMeanEvaluator	103
10.27	SignalCleaningEvaluator	104
10.28	Signal processing	105
10.28.1	FourierEvaluator	105
10.28.2	SlidingWindowFunctionEvaluator	106
10.29	Statistics	106
10.29.1	ClassEvaluator	106
10.29.2	CounterBZEvaluator	114
10.29.3	CounterEvaluator	116
10.29.4	GenericClassEvaluator	117
10.29.5	QuantileEvaluator	123
11	Virtual signals	125
11.1	Expressions	125
11.2	Declaration	131
11.3	Special functions	132
11.4	Anonymous functions	134

1 Introduction

IAV AMeDA is a software that can evaluate measurement data. These evaluations are not interactive like the kind completed by programs such as Diadem or FlexPro, as IAV AMeDA cannot be operated in an interactive fashion. Instead, IAV AMeDA has been configured so that it automatically carries out one specific, or even several, evaluations. This configuration can then be used on data sets; an XML file is usually provided as the result of the evaluations. As such, it is possible to automatically evaluate thousands of files without the user having to do anything else. The output file then contains the results of the analyses for all measurement files.

The configuration of IAV AMeDA is carried out by the XML files. These files contain all the information required for the description of a special evaluation task. The evaluation can be both signal-based and message-orientated.

IAV AMeDA can directly input most common file formats used in the world of automobiles. For example, accepted file formats include the formats used by Vector and ETAS *.MDF, *.ASC, *.LOG or the audio format *.WAV. Furthermore, certain data logger specific formats are supported, such as the IAV drive recorder or the G.I.N. data logger from the CANLog and MultiLog series. IAV AMeDA has been designed so that it is easy to expand to include new file formats. The aim of this flexible file access is to prevent the double archiving of data insofar as this is possible: Situations where a file that has been converted to the formatting required for the evaluation has to be stored together with the raw data file that contains the identical information are to be avoided.

The files are not necessarily accessed via a file system. IAV AMeDA does not use simple filenames to access the files, instead it uses a URL. As such, files can also be directly evaluated from an FTP or HTTP server. IAV AMeDA is able to evaluate files directly from a ZIP archive. No manual unzipping of the archived data is required here: The unzipping takes place “on the fly” during the input process.

IAV AMeDA is designed so that measurement files do not have to be inputted as a whole, instead can be analysed point for point. As such, no limits are placed on the size of the file to be analysed. In order to avoid the problem of having to manually analyse or pass on a file containing several gigabytes for a found event, the interesting area of this event can be automatically cut out during the evaluation and written in a separate, small file, which can then be easily handled.

IAV AMeDA has been developed in Java. As such, it can run on all platforms containing a Java virtual machine of version 1.5 or higher. IAV AMeDA can be used on PCs running both Windows and Linux, as well as on most UNIX computers. IAV AMeDA already supports multi-processor computers and there is a load distribution to computer clusters available in connection with the measurement data platform.

2 Functionality

2.1 Preliminary note

This manual describes the creation and functioning of the so-called XML templates. Manually creating this template does offer the greatest flexibility, however it is very time-consuming. For most evaluation tasks it is easy to create them using the MetaTemplateEditor. This editor allows for the automatic creation of all XML files using a convenient graphical interface. If the MetaTemplateEditor is used, it is generally not necessary to have knowledge of the details of the XML templates.

2.2 Quick start

The actual task that can be completed with the help of IAV AMeDA is to search for certain events in a data stream and to create statistics. In order to find an event, IAV AMeDA must identify a description of the events to be found. This description is split into two parts: First there are the so-called evaluators. These are modules

programmed in Java, that can be attached to the actual IAV AMeDA using the PlugIn principle. These evaluators receive all necessary concrete information and parameters that describe a special event from a so-called template. A template is a special XML file with the file ending *.TEM. This XML file is edited with a text editor and, as such, is very easy to modify. Certain information must be contained in every template and is directly evaluated by IAV AMeDA. This includes the information concerning which evaluator is to edit the template. Other information is first evaluated by the evaluator and is therefore different from evaluator to evaluator. Below a minimum example of a template, which searches for a very simple event, is presented. The event is an engine speed of over 6,000 rpm.

```
<\textexclamdown xml version="1.0" encoding="ISO-8859-1"\textexclamdown >
<template>
  <evaluator name="FunctionEvaluator"/>
  <description>Engine speed to high!</description>
  <instance>
    <signal name="M01_speed"/>
  </instance>
  <content>
    <parameter>
      <par name="speed"/>
    </parameter>
    <function>
      <![CDATA[
        if (Drehzahl>6000) result=1;
      ]]>
    </function>
  </content>
</template>
```

The first line is stipulated by the W3C XML standard and shows which XML version and which character encoding are being used. For Windows this is normally ISO-8859-1. Then comes the introduction of the template through <template/>. Under the tag <template/> comes the tag <evaluator/> which contains the information regarding which evaluator is to edit this template. In this case it is the FunctionEvaluator. Then the description of the template is conveyed in the tag <description/>. This description will also appear later in the result file once this event has been found. The tag <instance/> stipulates which signals are consulted for the evaluation. It is assumed that the engine speed is to be found in a signal with the name "M01_Drehzahl". Then comes the tag <content/>. The tags that have been described above must be contained in all templates, as they are directly evaluated by IAV AMeDA. The content of the tags <content/> is however, not directly evaluated by IAV AMeDA; instead by the respective evaluator. As such, this can differ from evaluator to evaluator.

In this example, the FunctionEvaluator is used. This evaluator evaluates one function. In the tag <content/> it expects to see a parameter list for a function and the function itself. The parameter list is defined in the tag <parameter/>. Only the parameter Drehzahl is used here. The function is described in the tag <function/>. The return value of the function is a floating point value between zero and one. It indicates the probability that the event being searched for occurred. A one means "sicher aufgetreten" a zero "sicher nicht aufgetreten". In this case, a one will be returned if the speed is larger than 6,000.

As the text fields in an XML file¹ cannot include any special characters, such as "kleiner" (<) or "groesser" (>), as these are reserved for tags, these must be placed in brackets <![CDATA[und]]>. These brackets inform the XML interpreter that the characters protected in this area are not to be treated as XML elements, as is necessary in this example for the greater-than sign (>).

Beside the templates, which describe a concrete event, a so-called template configuration is required. This *.TKOfile stipulates which templates are required for a certain evaluation task and which should be started. With the prerequisite that the above template was saved using the file name Drehzahl.tem, the template configuration

¹Details about XML format can be found in W3C. (<http://www.w3.org/XML/>)

only comprises one line:

```
use("speed")
```

3 Integration into the MessdatenPlattform

As mentioned above, the idea of IAV AMeDA is that evaluation templates are run automatically. The most common way to do this is to upload templates to the MessdatenPlattform which then distributes corresponding evaluation jobs to IAV AMeDA clients. The results can be viewed as files on the MessdatenPlattform or they are further processed to reports. For the generation of reports several evaluators write results into a database (the so-called meta database). The reporting engine can retrieve the result data from there.

In this context a result template is a ZIP archive which contains the template files described in the following sections as well as an ameda.scr file and a TKO file. Moreover, the archive can contain additional data like database definitions and other files needed for the evaluation.

More details about the upload and activation of evaluation templates can be found in the MessdatenPlattform manual.

4 TKO file

The template configuration controls the execution of the templates depending on different parameters. In most cases, only a set of `use` statements named all templates to be executed. Since the TKO file but is executed once before the evaluation of each measurement file, e.g. depending on the File names different templates can be executed. If in an evaluation the templates a and b are to be executed, a minimal TKO file looks like the following:

```
use("a")
use("b")
```

Listing 1: Minimum TKO file

The `script([name])` statement can be used to run additional files.

```
if (a="test.dat") then
script("testFiles")
endif
```

Listing 2: Example

Variables can be created by a simple assignment.

```
VAR = 1.0
```

Listing 3: Example

If a variable is to be created with a name that is only known as a string variable, you can use the `set` command.

```
varName = 'VAR'
set(varName, 1.0)
```

Listing 4: Example

The `exist` statement can be used to check whether a variable exists.

```

if exist(name) then
out(name)
endif

```

Listing 5: Example

Variables defined in the TKO file can also be used in the attributes of templates. The syntax of templates is described elsewhere. The following example shows how in the instance part of a Templates, two variables are used to attract the signal "signal_name1".

```

SIGNAL="signal_name "
NUMMER=1

```

Listing 6: Definition of variables in the TKO file

```

<?xml version="1.0" encoding="UTF-8"?>
<instance>
  <signal name="{SIGNAL+NUMMER}" />
</instance>

```

Listing 7: Use of a variable in a template

XML files can be read in, and XPath expressions can be used to access these files. Below is first the `test.xml` file, and then an example of the TKO statements.

```

<?xml version="1.0" encoding="UTF-8"?>
<root>
  <key name="test" />
  <key name="test2">hallo</key>
</root>

```

Listing 8: The file `test.xml`

```

xml=loadXML('test.xml')
k=getXPath(xml,'key/@name')           // provides 'test'
k2=getXPath(xml,'key[2]/@name')       // provides 'test2'
k3=getXPath(xml,'key[2]')             // provides 'hallo'
b=testXPath(xml,'key/@name="test"')   // provides true
b2=testXPath(xml,'key/@name="test4"') // provides false
b3=testXPath(xml,'key="hallo"')       // provides true

```

Listing 9: Example

4.1 Alias

The `alias` instruction can be used to assign aliases for signals. The actual Evaluation can then work with the aliases. Thus, the evaluation can be easily adapted to different Signal names can be adjusted. Several signals can be assigned to one alias. The first signal which is of the current measurement to be evaluated is used. In the example, the signal `speed` is generated, and depending on the measurement, the signal `kbi.K01_Rad_kmh` or `ant.BR1_Rad_kmh` used. This function can be used to perform different types of measurements in which the same signals are named differently with one and the same configuration. However, one special feature must be taken into account when evaluating CAN measurements: It is only checked whether the specified Signal is contained in one of the DBCs. Not all CAN messages of the measurement are analyzed to determine if the corresponding CAN message is actually included in the measurement. So it may be that the `alias` statement attracts a CAN signal which is declared in a DBC, but for which there is no matching message in the Measurement there.

```
alias('speed','kbi.K01_Rad_kmh','ant.BR1_Rad_kmh')
```

Listing 10: Example

The above behavior cannot be achieved by two statements, since the names of the aliases are must be unambiguous.

```
alias('speed','kbi.K01_Rad_kmh')  
alias('speed','ant.BR1_Rad_kmh')
```

Listing 11: Example

The prefix (?RE) can be used to specify a pattern (RegEx) for a signal. For example, signals that can occur on different channels (with different prefixes) can be handled in this way. However, which of the signals is actually used is then left to chance. If a particular signal is to be preferred, it must be explicitly specified before the RegEx. In the following example, the speed ("speed") with the prefix "ant." is preferred. If this signal is not found, any speed signal is used instead. A side effect of using a RegEx is that signals can also be used, that have the same name as other signals. However, the random selection principle also applies here.

```
alias('ant.speed','(?RE).*\.speed')
```

Listing 12: Example

4.2 DBCChannelSwap

Via a special instruction in the TKO file, the database assignments of an evaluation can be exchanged. This is achieved by a specification in the dbcChannelSwapList variable. In this variable interchanges can of the DBC assignment can be made. It should be noted that the actual channel assignment is retained! If evaluators are directly conditioned with numeric CAN channels, this will not be changed. Only if an evaluator with a String like ant or kom is set as CAN channel, this assignment is also exchanged.

In the following example, the DBC's of Can channels one and two are swapped.

```
dbcChannelSwapList='1->2;2->1'
```

Listing 13: Example

4.3 Graphics generation

Preview graphics can be created during the evaluation. These are png graphics showing the waveform of selected signals before and after an event represent. This allows a quick preview of the event while still in the browser. The variant described here is an obsolete method of graphics generation. When new evaluations are created, the PreviewGraphicCreator or the StaticGraphicCreator be used. (Siehe auch 10.15.1, 10.15.2)

4.3.1 Event-based

There are two ways to create a graphic. The first is the event-based Variant. A graphic is generated when a event is found. To do this, the graphic generation must be enabled using the .createGraphic=true statement and signal-based logging must be enabled. This can be done e.g. with the logging='MDF,10,10' instruction. By default graphics of the size 640*480 pixels generated.

```
logging='MDF,10,10'  
.createGraphic=true
```

Listing 14: Example

The size of the graphics can be changed with the `.graphicWidth=[width]` and `.graphicHeight=[height]` can be changed.

```
logging='MDF,10,10'  
.createGraphic=true  
.graphicWidth=4000  
.graphicHeight=3000
```

Listing 15: Example

4.3.2 Static

Static graphics can also be generated, which are displayed over the entire measurement and can be generated independently of an event. This is done with the TKO variables `.graphiclist`. This can be passed a list of strings. The first string determines the layout of the graphic, and all further strings each a Signal to be displayed. In the example, a graphic with the name "Graphic1" with 800*200 pixels is generated. In this graphic the signal `s1` is shown. For `s1` a scaling of the y-axis from -1 to 6 is set and the color red.

```
.graphiclist=createList("name='Grafik1' width=800 height=200",  
"name='s1' ymin=-1 ymax=6 color='red'")
```

Listing 16: Example

Several signals can be specified.

```
.graphiclist=createList("name='Grafik1' width=800 height=200",  
"name='s1' ymin=-1 ymax=6 color='red'",  
"name='s2' ymin=-1 ymax=18 color='green'")
```

Listing 17: Example

If a list is specified, which contains further lists, several graphics can also be generated.

```
.graphiclist=createList(  
createList("name='Grafik1' width=800 height=200",  
"name='s1' ymin=-1 ymax=6 color='red'"),  
createList("name='Grafik2' width=800 height=200",  
"name='s2' ymin=-1 ymax=18 color='green'"))
```

Listing 18: Example

4.4 Constants

Constant signals can be generated with the `constSignal` instruction. In the evaluation, these signals can then be used like completely normal signals. There are two ways to declare constant signals, as the following example shows. The first signal is generated without timing. However, this is not how it can be used by certain evaluators. If such a constant signal is to be used in a `FunctionEvaluator`, it must be marked with `call="false"` must be marked. The second signal generates a value at time 0 and has the constant value 1 from then on.

```
constSignal constant1 = 1
constSignal constant2(0) = 1
```

Listing 19: Example

4.5 Control commands

Loop constructs like a `while` loop are possible.

```
a=0
b=0
while a<10 do
a=a+1
b=b+2
endwhile
```

Listing 20: Example

Conditions can be checked with a `if` query.

```
if a<0 then
b=sqrt(-a)
else
b=sqrt(a)
endif
```

Listing 21: Example

A `switch-case` combination can be used.

```
switch a
case 1 : b=5 endcase
case 2 : b=3 endcase
case 3 : b=1 endcase
endswitch
```

Listing 22: Example

4.6 Listen

Lists can be created. The `createList()` statement creates a list, and with `addList(a, [element])` elements can be added. With `getList(a, [Index])` the list can be accessed.

```
a=createList()
addList(a,'Test1')
addList(a,'Test2')
out(getList(a,0))
```

Listing 23: Example

Alternatively, a list can be created directly in the `createList` command with elements be filled.

```
a=createList('Test1','Test2')
out(getList(a,0))           // outputs 'Test1'
```

Listing 24: Example

4.7 Mathematical operations

Within the TKO file simple assignments like `a=10` can be executed. Numeric values, strings and Boolean values (true and false) can be used in the assignments. NaN and Infinity are also valid as numerical values. These can also be used as default values in various evaluators, for example to mark invalid values as inputs of calculations.

It can be calculated `a=2*x+1`.

Boolean expressions like `a=x<3` can be used. In addition, boolean values can be linked with `and`, `or` and `xor` or negated with `not` or `!`.

Mathematical functions such as `sqrt(x)`, `abs(x)`, `sign(x)`, `round(x)`, `floor(x)`, `ceil(x)`, `max(x, y)`, `min(x, y)` can be used.

4.8 Strings

It is possible to calculate with strings. So the following example gives the text `Hello beautiful world off`.

```
x="Hallo"
a=x+" schöne"
a=a+" Welt"
out(a)
```

Listing 25: Example

Special functions are available for editing strings. These are `endsWith`, `startsWith`, `upper`, `parentFile`, `extractFilename`, `subStr`, `strPos`, `strLen`, `getDate`, `format` and `decimalFormat`

```
endsWith('test.dat','.dat') // provides true
endsWith('test.dat','.asc') // provides false
startsWith('test.dat','test.') // provides true
startsWith('test.dat','hallo') // provides false
upper('test.dat') // provides 'TEST.DAT'
parentFile('hallo/test/m.dat') // provides 'hallo/test'
extractFilename('hallo/test/m.dat') // provides 'm.dat'
subStr('hallo/test/m.dat',6,4) // provides 'test'
strPos('test','hallo/test/m.dat') // provides 6
strPos('du','hallo/test/m.dat') // provides -1
strLen('hallo') // provides 5
getDate('yy.MM.dd_HH:mm:ss') // provides date as String
// (in Java SimpleDateFormat Syntax)
format(v, 2) // formats v to Decimal form with 2
// decimal places as a String
decimalFormat(v, '0.00') // formats v to Decimal form with 2
// decimal places as a String
// (in Java DecimalFormat Syntax)
```

Listing 26: Example

5 SCR file

The `ameda.scr` file has the same syntax as the TKO files. The difference is that the TKO files for each Measurement to be executed once, while the SCR file is executed only once, before the start of the evaluation. At least the file pattern used to search for measurement files must be defined in the `ameda.scr` file. It is also necessary to specify the TKO file that will be executed and, if using message-oriented measurements is worked, the data basis must be specified here. When using certain logger raw data formats (e.g. the Telemotive BluePiraT, the file pattern must be specified in a special way. The description of this can be found in the chapter on special file

formats. (Siehe auch 8.1, 8.8)

In the following example, the TKO script Example.tko is executed. The database is defined in the ExampleDatabase.xml file. All .asc files are evaluated from the measurement. The order of the evaluations of the individual .asc files is based on the result the alphanumeric sorting of the file names. The alphanumeric sorting ensures, for example, that the data recorded by the Multilog Data for the fact that a file with the designation Data1F10 is not evaluated directly after a file with the designation Data1F1, but first all files with the designations Data1F2 to Data1F9 are evaluated.

```
database="ExampleDatabase.xml"
pattern="*.asc"
configuration="Example"
```

Listing 27: Example

The database is specified in the form of an XML file. Multiple database descriptions can be specified in the XML file. In the following example, a CAN database in the form of a DBC file (exampleCan.dbc) is specified for CAN channel 1. The signals defined in this DBC are given the prefix "c1". For MOST channel 1, a database is specified in the form of a MOST function catalog (exampleMost.xml). The signals defined in the MOST function catalog are given the prefix "most".

```
<?xml version="1.0" encoding="UTF-8"?>
<plattform name="example">
<CAN-DBC name="exampleCan.dbc" channel="1" prefix="c1" />
<MOST_FC functionCatalog="exampleMost.xml" channel="1" prefix="most"/>
</plattform>
```

Listing 28: Example

The `initDataCollector()` statement activates the DataCollector. This represents the Interface to the MetaDatabase.

```
initDataCollector();
```

Listing 29: Example

When browsing the measurement archive (a measurement folder can also be used for local evaluation) after files to be evaluated, all files matching the file pattern are evaluated one after the other. If several sub-patterns are specified in the file pattern separated by semicolons, the associated files are evaluated in parallel. For this, however, all file patterns must fit together (have the same length in the file part, contain the same number of wildcards and Include wildcard symbols only in the same places). By file part is meant everything after the last "/". Wildcard symbols can be "?" (any symbol) or "*" (no or several arbitrary symbols) can be used. the "*" may be used in the file part only at the beginning or at the end. Furthermore, there are special flags for certain file types that allow special processing. For example, with certain file formats it is possible to read in a whole series of files as a single measurement. These special functions are described with the respective file formats.

6 The context

In AMeDA there is a central data structure, via which different parts of the program can exchange information with each other. This so-called context can be addressed at various points in AMeDA.

On the one hand, the context is always used when, when a variable is created in the TKO file or the SCR file. If a variable is created here, it can be read out again from the context at various points. The variables `STRING`, `DOUBLE`, `INTEGER` and `BOOLEAN`, which are created in the following example, can thus be reused in different places. In the example it is also shown how to use variables of different types (String, Double,

Integer and Boolean) can be created.

```
STRING = 'Hallo'
DOUBLE = 1.5
INTEGER = 15
BOOLEAN = true
```

Listing 30: Example

All variables that can be found in the context, can be used in the attributes of XML descriptions of templates. Simple operations can also be applied to the variables here. For this, only the variable name or term to be used must be specified in the attribute in curly brackets are set. For example, if the file name to be used in a signal export, is defined in a variable `FILENAME` via the TKO script, it can be retrieved as follows and extended with the extension MDF.

```
<?xml version="1.0" encoding="UTF-8"?>
<mdf name="{FILENAME+'.MDF'}" />
```

Listing 31: Example

The third way to use context variables is to use them in an evaluator, in which Java code is used (e.g. in the `FunctionEvaluator`). All that is required is to retrieve a reference to the context in the Java code and the `getValue` method are called. In addition, there are special methods for special data types to get back the variables in the corresponding data type. This is shown in the following example, which refers to the example variables defined above. Values can also be set by using the `setValue` method. The validity of the value of the context variable is limited to the evaluation of the current measurement file. If the value of the Context variables should be valid for the evaluations of the subsequent measurement files of a measurement archive, the `setValueParent` method must be used instead. One possible use case is the Evaluation of a measurement in the G.i.N. raw data format and the use of the extension "ML". In the example, the variable `STRING` is extended to the mandatory "Hello World!". The Validity of the value of this variable is limited to the evaluation of the current measurement file.

```
String s = getKontext().getString("STRING");
Double d = getKontext().getDouble("DOUBLE");
Integer d = getKontext().getInteger("INTEGER");
Boolean b = getKontext().getBoolean("BOOLEAN");
getKontext().setValue("STRING", s + " Welt!");
```

Listing 32: Example

Besides self-defined variables, there are also various variables that are set by AMeDA. Some of these variables are only present when AMeDA has been started by a measurement data platform. The following tables show the most important of these variables. The first table lists common variables that AMeDA uses.

Variable	Type	Meaning
sourcepath	java.net.URL	Full path to the measurement (archive or folder) as URL
sourcedata	String	Full path to the current measurement file in the measurement
configuration	string	name of the TKO file to be processed for each measurement
database	string	name of the XML file for the XML database description
pattern	String	The file pattern used for finding the input files
fileFilter	String	File extension to be assumed for opening files. This variable is especially needed if the files to be opened have no extension.
resultCount	Integer	Here you can specify the maximum number of events to be included in a result XML file. If more events occur, a corresponding error message is issued.
sloppySignals	Boolean	If this flag is set, signals that are attracted for the generation of virtual signals are allowed to be missing
mdfComment	String	The content of this variable is written in the comment field of MDF4 files.
mdfCommonProperties	java.util.Map	The contents of this map are written as key value pairs into the common_properties of MDF4 files.

The following table lists variables that can be used to influence the reading of certain file types.

Variable	Type	Meaning
bp_ignore_errors	Boolean	If this flag is set, certain errors are ignored when reading BluePiraT measurements.
continueMultilog	Boolean	If this flag is set and files of type 'MLS' are read, then the program continues with the next file if a read error occurs.
gl_ignore_time_errors	Boolean	If this flag is set, 1,000 errors in real time stamps of G.i.N. measurements are ignored. Only when the limit is exceeded does an error occur.
gl_new_meas_at_reboot	Boolean	If this flag is set, a new evaluation is started for G.i.N. measurements for each reboot of the logger.
gl_start_ts	Double	G.i.N. measurements are read from this timestamp (Unix timestamp or relative).
gl_stop_ts	Double	G.i.N. measurements are read to this timestamp (Unix timestamp or relative).
gl_relative_start_stop_ts	Boolean	If this flag is set, the start and stop timestamps are interpreted as relative (since start of measurement).
lin_ch_map	String	Mapping between CAN messages and LIN messages, if the LIN messages are stored as CAN messages, as it is common for G.i.N. loggers. This variable is normally filled automatically. Otherwise, e.g. 'C16L1:C17L2' maps the CAN channels 16 and 17 to the LIN channels 1 and 2.
cate_sort_buffer_size	String	The messages in CaTe measurements are not stored chronologically. Therefore, they must first be read into a sort buffer whose size can be adjusted by this variable.
csvSeparator	String	Separator symbol to be used when reading CSV files
csvQuote	String	Quote symbol to be used when reading CSV files
csvTimeResolution	Double	Time resolution when reading CSV files
csvReadUnit	Boolean	If this flag is set, the CSV file is expected to contain units for the individual signals (either in square brackets after the signal name, or in an extra line).
csvUnitInExtraLine	Boolean	If this flag is set, the units are assumed to be in an extra line.
csvEncoding	String	Encoding type to use when reading CSV files (e.g. 'utf-8')
csvTimeformat	String	Specifies the date format in which the timestamp column is interpreted. For an empty string, the time stamp is interpreted as a floating point number. (Default: Floating point)
csvTimestampcolumn	String	Specifies in which column the timestamp is found (either as a column name or as a number, default: 0)
csvAbsoluteTimestamp Format	Boolean	Specifies whether the timestamp is to be understood as an absolute timestamp (taken directly) or whether the measurement should start at 0. (Default: true)
osm_interface_address	String	Address of the OSM server used in the RouteEvaluator and ReverseGeocodingEvaluator
breakAtEmptyMeasurement	Boolean	If this flag is set, analyses with SourceEvaluator will abort without error if all file lists are empty

Variable	Typ	Bedeutung
handleIOErrorsAsWarnings	Boolean	If this flag is set, analyzes that fail due to an IO error are still considered successful and only a warning is issued. Results in the results database that were written prior to the cancelation can then still be used for reports, for example.

The following variables are set by or used by the measurement data platform and are only available, if the evaluation was initiated by the measurement data platform.

Variable	Type	Meaning
konfig	String	Name of the measuring device configuration belonging to the measurement to be evaluated
FzgVIN	String	VIN of the vehicle belonging to the measurement to be evaluated
dataVIN	String	A chassis number can be stored here during the evaluation. This is transferred to the measurement data platform at the end of an online evaluation and is used for consistency checks.
FzgInterneNr	String	Vehicle Internal number as stored in the measurement data platform for the vehicle belonging to the measurement
readDate	String	readout date as stored in the measurement data platform for the measurement to be evaluated
fleet	string	name of the fleet belonging to the measurement to be evaluated
readDateTime	String	Readout time as stored in the measurement data platform for the measurement to be evaluated
loggerName	String	Meter type and serial number of the meter used for the measurement to be evaluated
.useLoggerKonfigFiles	Boolean	If AMeDA is started from the measurement data platform, this flag can be used to set that files (e.g. DBCs) can also be used that are located in a ZIP archive that is stored as a measurement device configuration in the measurement data platform

7 Attributes in the measurement data platform

When AMeDA is run with a MeDaPF, information about the state of the data logger, with which the evaluated measurement was created can be exchanged with the MeDaPF. The information is stored as so-called attributes. String, integer, double and Boolean values can be stored. These values can then be accessed in a later processing step (during report generation).

Analogous to the attributes for measuring devices, the attributes for measurements are also available.

7.1 AttributeGetter

Already existing attributes can be read.

For static attributes, a `Object` is returned. When the method is applied to a dynamic attribute, an exception is thrown. If the attribute is not present, `null` is returned.

```
getMeasurementAttributeValue(String label)
getLoggerAttributeValue(String label)
```

Listing 33: Example

For dynamic attributes the return value is `TreeMap<Date, Object>`.

```
getMeasurementAttributeValues(String label)
getLoggerAttributeValues(String label)
```

Listing 34: Example

7.2 AttributeSetter

When setting attributes, a distinction is made between static and dynamic attributes. Static attributes, unlike dynamic ones, have only one value without a date. When setting, already existing value for this attribute will be overwritten. Dynamic attributes can contain several different values with one date. This results in attribute values always being added and never overwritten.

In the following example, values of dynamic measurement attributes are set with the data type `String`, `Double`, `Integer` and `Boolean`.

```
setMeasurementAttribute("String", "Hallo Welt", new java.util.Date ());
setMeasurementAttribute("Double", 1.0, new java.util.Date());
setMeasurementAttribute("Integer", 2, new java.util.Date());
setMeasurementAttribute("Boolean", true, new java.util.Date());
```

Listing 35: Example

The following example shows how to set static measurement attributes with different data types.

```
setMeasurementAttribute("String", "Hallo Welt");
setMeasurementAttribute("Double", 1.0);
setMeasurementAttribute("Integer", 2);
setMeasurementAttribute("Boolean", true);
```

Listing 36: Example

Data logger attributes are set analogously.

```
setLoggerAttribute("String", "Hallo Welt");
setLoggerAttribute("Double", 1.0);
setLoggerAttribute("Integer", 2);
setLoggerAttribute("Boolean", true);
setLoggerAttribute("String", "Hallo Welt", new java.util.Date());
setLoggerAttribute("Double", 1.0, new java.util.Date());
setLoggerAttribute("Integer", 2, new java.util.Date());
setLoggerAttribute("Boolean", true, new java.util.Date());
```

Listing 37: Example

8 File formats

8.1 BluePiraT (OTMT/XTMT/TMT)

The raw data format of the BluePiraT data logger from Telemotive is directly supported. A measurement here can consist of several files. The files of the BluePiraT have the extension `.otmt`, with the BluePiraT 2 there are also the extensions `.tmt` and

.xmt. Also supported are the .tmt.gz and .xmt.gz formats as they can be output by the BluePiraT 2 software. In these cases, the files are additionally compressed. Since a file does not necessarily correspond to a measurement, attaching “&bluepirat;” to the file pattern in the ameda.scr file, it is possible to achieve that an ignition cycle (which can consist of several files) is treated like a single file. To prevent entire measurements from becoming unusable because individual data blocks are faulty, there is a option, to control the handling of file errors via the “bp_ignore_errors” flag. If the flag is defined in the ameda.scr file and set to true, reading will simply continued at the next data block file in case of a read error. Is it set to false (default), the current evaluation will be aborted. This flag should be used with caution, as inconsistent measurement data may be evaluated, if set to true. <p> The message types CAN, LIN, FlexRay and MOST150 are supported. In addition, the signals of the GPIO interfaces, the serial interface and Ethernet frames can be read.

8.2 CSV

CSV files can be read. Since there are a number of different CSV formats, the importer can be configured in different ways. With the context variable `csvSeparator` the delimiter can be configured. By default, a comma is set, with `csvSeparator=“;“` it is changed to semicolon.

Additional context variables to customize the behavior of the evaluator can be found in the Kontext section.

It is assumed that in the first line of a CSV file the signal names, separated by the separator separately, are listed. The first column must contain the time if no further settings have been made.

```
t,U[V],I[A]
0,0,1
1,1,2
2,2,1
3,3,2
4,4,1
```

Listing 38: Example

Empty lines in the CSV file are read over.

With the variable `csvCmlTime=“true“` the interpretation of the timestamp in the format “HH:MM:SS,SSS” can be set be switched on. The timestamp must then be enclosed in quotation marks.

If an overflow occurs in the date values - from 23:59:59 to 00:00:00 - the relative Timestamp continues without jump.

CSV files can be read and written. When writing, a semicolon is used as the separator symbol. The decimal separator is a comma. This allows the files to be opened directly in MS Excel. When reading in, the separator symbol can be freely selected. The context variable `csvSeparator` determines the separator symbol. The decimal separator may be a point or comma.

8.3 CaTe (CLG)

The Cate-Logger can perform frame-based CAN traces and signal-based recordings. The data is stored in separate files (either frame-based or signal-based). CanError-Frames can also appear in the signal-based format (only ErrorFrames, no correct frames). Therefore, when opened with MDFView, the platform dialog opens. However, this can usually be ignored (canceled). In addition to error frames, error messages may also appear. These are output in the signal errorStrings as key-value pairs. MDFView is used to open a measurement by placing it (with all associated files) in a folder and then opening one of the .clf files with MDFView is opened. In AMeDA the measurement archive can be edited by using “*;&cate” as pattern.

When evaluating data, the entire folder with clg measurements can be activated by specifying `"*;&cate"`. In this case, all clg files are read in as one large measurement, and considered as one file. This also works, if the measurement is available as an archive.

8.4 DBCs

If the attribute `SystemSignalLongSymbol` is present for a signal, the name specified there is taken over as signal name.

If the `SignalLongName` attribute is present for a signal, the name specified there can be used as signal name.

DBC's can be used for the interpretation of CAN messages.

If the DLC of a message is smaller than assumed in a DBC, the missing part of the message will be padded with zeros for the interpretation of the signals in the DBC.

If a signal is defined outside a CAN frame, an error message is generated when the signal is used.

8.5 Drive Recorder TCS (PST)

The raw data format of the TCS data logger from IAV is directly supported. A measurement here can consist of several files. The files have the extensions `.pst` and `pr#`, where `#` stands for a number. All files that start with four identical numbers belong to one measurement series and are treated like a file during evaluation. For this, one of the associated PST files must be opened. The file name can also be used to distinguish between message-based and signal-based measurements. Message-based measurement files end in `lg##.pst` or `lg##.pr#`, signal-based in `ch##.pst` or `ch##.pr#`. For the evaluation of all signal-based measurement series, for example, the pattern in the `ameda-scr` file must be set to `"*/????ch01.pst"`. Of course, the path must be specified even more precisely if multiple signal-based measurement kingdoms are included in the measurement. In rare cases, evaluations with TCS measurements may result in an error message stating, that the sorting buffer is too small. In this case the variable `tcs_sort_buffer` be set in the `ameda.scr` script or in the TKO file. The default value is 1200 CAN packets.

8.6 Flea-CSV (LOG)

Data written with the Flea-Box have a special CSV format. This differs from the simple CSV format in that each file is preceded by a header. In addition, several files can belong to one measurement. by appending `"&flea"` to the file pattern in the `ameda.scr` file can be achieved, that the files are treated as one file during evaluation. When opening a file from a measurement in MDFView, the entire measurement is automatically read in. Make sure that the folder contains only data that belong to a measurement. Otherwise, error messages will occur.

8.7 G.i.N. logger (GJF)

The G.i.N. data loggers use a new file format in newer versions for recording Ethernet data. This format has the GJF extension. If corresponding files are to be evaluated, the file pattern must be selected like this, that any file from the corresponding set of measurement files is opened. AMeDA/MDFView then internally creates a list of all related files (same file name except sequence number) and opens them together, starting with the selected file.

In addition to ethernet data, ethernet statistics frames may also be included. They are output in a separate frame signal.

Similar to CAN mapping, Ethernet channels can also be assigned differently in the logger configuration. Information about this is only available in the master measure-

ment data. In order to ensure a correct allocation of channels, the data must be evaluated as a compound with master data.

Usually the corresponding data loggers are used in the network and synchronize with each other over the network. Various flags can be used to control what should happen to data blocks, that do not contain synchronization information. These data blocks then do not contain reliable timestamps. The flags described here should therefore only be adjusted with great care, if the user knows exactly what he is doing.

Parameter	Default	Meaning
gl_skip_unsynced	true	Data blocks without synchronization information are skipped when read. If this flag is not set, the previously read information will be reused.
gl_allow_time_sync_from_successor	false	If a data block itself does not contain synchronization information, and no synchronization could take place before, the information of the following block is used if necessary.
gl_allow_unsynced	false	If there is no following block with synchronization information, the data will be read anyway (without absolute timestamp).
gl_check_timestamps	true	It will be checked if all timestamps are chronologically correct. Otherwise an error is output.
continueGJF	false	If an error occurs, a new analysis is started with the next data block. There are then several independent analysis results.

8.8 G.i.N. logger (ML)

The raw data format of the MultiLog and GinLog data loggers from G.i.N. is directly supported. A measurement here can consist of several files. Although the files have no extension, AMeDA has the pseudo extensions ML and MLS for this raw data format. With the extension ML it is achieved that a single file is read from the measurement. With MLS all files are read from a row. Appending "&mldrstart;" to the file pattern in the ameda.scr file opens the first file in a series. If you want to evaluate all files from a row, you have to use for example the pattern "Data0F*;&mldrstart;" and the FileFilter MLS. If the optimized folder format was used when recording the data, the pattern must be of the form "**X/D0F*;&mldrstart;". So the complete row will be opened even if e.g. Data0F1 does not exist. Starts the series at Data0F1, you can also use "Data0F1" as pattern. <p> In G.i.N. raw data, there are two different types of pointer stamps. The relative timestamp counts seconds from the start of measurement. This timestamp may not be reliable, because it starts again at 0 during a logger reboot within one measurement. In AMeDA/MDFView, a time gap of about one second is inserted at such a point. Therefore, if a logger reboot takes place, this must be taken into account accordingly in the evaluation. It can be recognized by the fact that the "LoggerReboot" signal outputs a value of 1. Measurements recorded with current loggers contain additional information that allows reconstruction of the time stamps. AMeDA/MDFView uses this information to calculate continuous time stamps. In this case, a value of the signal UTCTime is output at the beginning of the measurement, which outputs a Unix time stamp in ms. This signal can be used e.g. for the RealTimeEvaluator. The second type of time stamps is written into the measurement in the form of signals containing day, month, year, hour, minute and second. This function must be activated in the configuration of the data logger. These timestamps can also be used for the RealTimeEvaluator. <p> Depending on the measurement type and firmware version with which the measurement was created, the time stamps have different accuracy. In Data0 blocks the absolute timestamp is basically only to the second, Data1 and Data2 contain, at least in more recent firmware versions,

absolute time stamps accurate to microseconds. If different data blocks are to be compared with each other, the absolute time stamps should therefore be treated with caution.

8.9 MDF 4.0 (MF4)

The MF4 format is a signal-based format specified by ASAM. It can be read by AMeDA and MDFView. The most important features from the specification are supported. Certain conversions ("Text To Text", "Text To Value" and "Algebraic") are not supported. "Arrays" and "Structures" are not fully supported.

8.10 Matlab (MAT)

The MAT format does not generally represent a measurement data format. Therefore, only MAT files that contain signals in some form are supported in AMeDA, as it can be generated e.g. with IAV MDFView (the common, not the CANoe form).

In the files there are signal groups, where a signal group just corresponds to a group of signals, contain the values at the same points in time.

For each signal group there must be a descriptive cell array. This consists of two columns, the first of which contains signal names and the second units as a character array. Each line corresponds to a signal, with the first column being for the sampling times (in seconds).

In addition, for each signal group there is a two-dimensional matrix in which the columns contain the signal values. Here, too, the first column is reserved for the sampling times. The order corresponds to the order in the descriptive cell array. The two data structures can be clearly assigned by their names. The cell array must be named according to the `groupn_name` scheme. The `n` stands for a whole number. The associated matrix with the signal values must then be called `groupn_signal`.

8.11 Measurement Data Format (MDF)

The MDF format can be read and written. Various evaluators allow the generation of MDF files. Among these evaluators is the `SignalLoggingEvaluator` and the `DataExtractEvaluator`. In addition, MDFView is able to export signals to MDF files. When writing, make sure that the files do not exceed a size of 2 GB. In such a case, the program terminates with an error message. If signal names are duplicated in a file, the string "`_CGn`" is appended to each of the affected signal names. Here `n` corresponds to the number of the channel group to which the signal belongs. In a measurement created from a CAN measurement, a channel group usually corresponds to the group of signals originating from a CAN message.

8.12 PCAPNG

The PCAPNG format is used by tools such as Wireshark to save Ethernet frames. The format can be both read and written. When writing, the context variable "`pcap-MaxFramesPerSection`" can be used to set the maximum size of a section in the output file. Smaller section sizes require less memory. However, WireShark only allows segmented files to a limited extent due to a bug (version 4.2.6), which means that sections that are too small can lead to errors.

8.13 RPC 3

RPC 3 is a data format of the company MTS. Please note that RPC files do not contain an explicit time channel, but only an implicit time channel consisting of the specification of the start time and time interval of the points. In addition, all signals must have the same Time channel and therefore all contain the same number of points. In addition, when converting raw values to physical values only one scaling

factor can be specified. Even the specification of an offset is not possible. This may reduce the resolution of the stored signals. In the RPC 3 format so called half frames can be stored. Half frames can be stored. These are currently not supported by the importer.

Supported are RPC 3 files which contain IEEE 754 little endian 32-bit float values.

Likewise little endian 16-bit integer values are supported.

8.14 Statistical Data Message (LDC)

The TDBox LDC file format for Statistic Data Messages is supported. All the Signals for which signal curves are stored are displayed. Signals for which cumulative metrics are stored are identified by one of the following suffixes: `internal_f1` (First or Last), `internal_avg` (Average) and `internal_max` (Max). It is also possible that empty signal traces are stored. In this case, only the cumulative metrics are displayed.

8.15 UniCAN2/3 (CSM)

The raw data format of the UniCAN2 data logger of the company CSM is directly supported. A measurement here consists of multiple files. The files have the extensions `.dat` and `.csm`. If a measurement is to be opened, this is done by opening the File `CFG_....CSM`. All other files are then read out in the correct order.

The newer version of the logger, UniCAN3, supports not only the logging format of the UniCAN2, but also recording in MF4 files. As in the old format, multiple files are created, by multiple parallel data streams as well as by multiple time snippets per data stream. If the MF4 files are recognized as UniCAN3 recordings when opened, all files are opened together and treated like one measurement file when selecting any of the files for opening (similar to the old format). This requires that the files are in a common folder or archive.

In addition to the usual signals and CAN messages, this file format can also contain the so-called Logbook entries are read. These are output as a special signal of type `DataRecord` and can be evaluated with the (generic) `JavaEvaluator` be evaluated. Details about the logbook entries can be found in the corresponding specification of CSM.

9 Data and message types

AMeDA reads both signal-based and message-based file formats. In particular, the message-based formats differ in the types of messages supported and in the way the messages are stored. However, so that the evaluations can be performed regardless of the file format, there are certain conventions on how messages are to be interpreted. This can lead in individual cases to the fact that information contained in a file format, not be read out as well. On the other hand, it can also happen that information, that are not saved in a file format are filled with default values. For example, depending on the file format, CRC checksums are in the corresponding data structure, sometimes filled and sometimes simply set to 0. The following are special features that occur with different message types.

9.1 FlexRay messages

Unlike CAN messages, FlexRay messages have redundant channels. So, for example, there is not only channel 1, but channels 1A and 1B. Different file formats differ slightly in the way, how the channels are interpreted. In AMeDA, the channels are counted alternately, regardless of the file format. Channel 1A is thus read in as channel 1, channel 1B as channel 2, and so on. In this way, the channels can also be viewed separately. This is particularly important if the B channel is not used merely

as a redundant channel, but is used for bandwidth expansion. The FlexRay standard also allows this.

9.2 Signal name conventions

Depending on which data source signals come from, different information besides the pure name can be of interest for signals. For signals taken from CAN frames, for example, the CAN ID and the channel may be of interest. AMeDA reads - depending on the file format - corresponding additional information and makes it available. In MDFView, the corresponding information can then be viewed via the Signal Info dialog. AMeDA makes available for analysis each signal under several names. This serves in particular the possibility of signals whose signal names are not unique in themselves, to be able to reference them unambiguously nevertheless. Therefore CHANNEL.SIGNAL and CHANNEL.MESSAGE.SIGNAL are provided as alternative signal names. CHANNEL stands for the channel name, or if this is not known, the channel number. MESSAGE stands for the message name, or if this is not known, the message ID in hexadecimal notation. SIGNAL stands for the actual signal name. If the CHANNEL or MESSAGE information cannot be determined, because the source data format does not provide the information, the corresponding part of the signal name is omitted or the signal name is not generated at all. In FIBEX definitions it is in principle possible that one and the same signal is referenced in several messages. In this case the name of the last message occurring in the FIBEX file is used as the message name.

10 Evaluators

Most evaluators require input signals to be specified for processing. This is done using `<signal/>` tags within the `<instance/>` tags if necessary. Usually it is sufficient here to reference the signals by their signal name. In some cases, however, this is not clear. Then it is possible to specify additional attributes for finding the signals as well.

The following are available as attributes for filtering signal names - depending on the file format.

attribute	meaning
channel	name of the channel (e.g. CAN)
channel_id	ID of the channel (e.g. CAN)
message	name of the message
message_id	ID of the embassy
source_info	information from SI block (only for MDF files)
description	description of the signal from the meta information
pdu	PDU name from Ethernet data

In the following example the signal `s1` is used, which originates from the CAN message `CAN_MSG`.

```
<?xml version="1.0" encoding="UTF-8"?>
<instance>
  <signal name="s1 with(message=CAN_MSG)" />
</instance>
```

Listing 39: Example

Due to the structure of with expressions, the special characters `"\`", `"("` and `")"` must be escaped. To do this, these characters must be preceded by a backslash. The with expressions can also be used in aliases. If you want to filter for multiple attributes, you can simply use multiple with expressions, as in the following example, can be attached to each other.

```
s1 with(message=CAN_MSG) with(channel=CAN1)
```

Listing 40: Example

10.1 ADTFExportEvaluator

The ADTFExportEvaluator exports measurements to the DAT format, which is supported by ADTF, with the help of the ADTF streaming library. So far, CAN, FlexRay, Ethernet and UART are supported. To ensure that no frames got lost at the beginning of the measurement, the template "Milliseconds UTC" with input signal "UTCTime" should be used.

In the example, several signals are read in: flexRayFrame (FlexRay), ser5ext_string (UART), EthernetRx1, EthernetRx2 (Ethernet) and canMessage (CAN). In the <content/> tag, the out attribute specifies that the export is to the out.dat file. And since the zip attribute is true, this file will be packed as a zip. Via the attributes streamNameCan, streamNameFlexray, streamNameEthernet and streamNameUart the respective streams are assigned their names. The CAN stream gets the designation CAN here, the FlexRay stream gets the name Flexray, the Ethernet stream gets the name Ethernet and the UART stream gets the name UART. Since the ADTFExportEvaluator expects strings as input, the CanToByteStreamEvaluator is first used to create a ByteStream from the incoming CAN messages. Then these ByteStream will be converted to a string with the help of the ByteStreamToStringEvaluator. A detailed description of these evaluators can be found in the respective chapter.

```
<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>ADTFExportEvaluator.ADTFExportEvaluator</evaluator>
    <description short="Dat ADTF" />
    <instance>
      <signal name="flexRayFrame" />
      <signal name="ser5ext_string" />
      <signal name="EthernetRx1" />
      <signal name="EthernetRx2" />
      <signal name="canMessage" />
    </instance>
    <content out="output.dat" zip="true" streamNameCan="CAN" streamNameFlexray="Flexray"
streamNameEthernet="Ethernet" streamNameUart="UART" />
  </template>
</templates>
```

Listing 41: File datExport.tem

10.2 ATFXClassificationExportEvaluator

The ATFXClassificationExportEvaluator writes data from ATFX measurements directly into the meta database. The histograms are assumed to follow the nomenclature as found in CAETEC measurements.

As parameter fileName a file pattern must be specified which determines, which ATFX files should be searched from the measurement. The parameter tableName specifies the table into which the histogram is to be written. The signalAlias parameter specifies an identifier for the analyzed signal, which is used in the table for the signal column label, if simpleTable is set and thus the table is generated in the simplified form as with the ClassEvaluator. With the parameters start, classWidth and numClasses can be specified, how the histogram was divided in bins. These parameters must, of course, fit the parameters, used when recording the data.

```

<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>AtfxClassificationExportEvaluator</evaluator>
    <description short="neuer Name" />
    <content fileName="*.atfx" measurementName="SAMPLECOUNTING_(UL_12200210)" signalAlias="a1"
simpleTable="false" tableName="class_UL_12200210" start="0" classWidth="1" numClasses="150" />
  </template>
</templates>

```

Listing 42: File atfxClasses.tem

10.3 General information

For each evaluator, signals can be listed under the `<signalValueToGet/>` tag, whose values are to be stored in the event of an event, similar to the Environmental conditions during a fault memory entry. In the `<signal/>` tag, in the attribute name the name of the signal which will be stored in case of the event should be. In the example this is the value of the signal `s2`. In the tag attribute can be used to specify the name of a tag under which the signal value will be included in the XML result of the evaluation is to be taken over. If an event has a duration greater than zero, the Signal values at the beginning of the event taken over.

```

...
</content>
<signalValueToGet>
<signal name="s2" tag="sig"/>
</signalValueToGet>
</template>

```

Listing 43: Example

If the values are to be saved at the end of an event, the `valueAtEnd` attribute must be set to `true`.

```

...
</content>
<signalValueToGet>
<signal name="s2" tag="sig" valueAtEnd="true"/>
</signalValueToGet>
</template>

```

Listing 44: Example

10.4 ByteStream

10.4.1 ByteStreamMultiplexer

The `ByteStreamMultiplexer` is used to combine different binary data streams into one data stream. Different multiplexers can be used, which make it possible to separate the different sources in the common data stream.

In the following example, consisting of four templates, the first two templates each convert one CAN message into a binary stream (signals `data1` and `data2`). The template `bsm.tem` then combines them into a data stream named `muxData`. The `mux="no"` attribute is used to simply Copying the data together. Finally, the last template `bstf.tem` copies the generated Data stream `muxData` into a file.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>CanToByteStreamEvaluator</evaluator>
  <description>Test</description>
  <content>
    <message id="16" can="1" out="data1" />
  </content>
</template>

```

Listing 45: File ctbs1.tem

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>CanToByteStreamEvaluator</evaluator>
  <description>Test</description>
  <content>
    <message id="17" can="1" out="data2" />
  </content>
</template>

```

Listing 46: File ctbs2.tem

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>ByteStreamMultiplexer</evaluator>
  <description>Test</description>
  <instance><signal name="data1"/><signal name="data2"/></instance>
  <content>
    <data mux="no" out="muxData"/>
  </content>
</template>

```

Listing 47: File bsm.tem

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>ByteStreamToFileEvaluator</evaluator>
  <description>Test</description>
  <instance><signal name="muxData"/></instance>
  <content>
    <data zip="false"/>
  </content>
</template>

```

Listing 48: File bstf.tem

The following is an example using the Panasonic multiplexer, which has a special header before each data packet. The header consists of the 4 bytes 0xBF,0xC[mux],[len low],[len high].

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>ByteStreamMultiplexer</evaluator>
  <description>Test</description>
  <instance><signal name="data1"/><signal name="data2"/></instance>
  <content>
    <data mux="panasonic" out="muxData"/>
  </content>
</template>

```

Listing 49: File bsmPanasonic.tem

In the following an example is shown where the GNLog multiplexer is used, which organises the data in such a way that it can be read by the Harman/Becker tools. The GNLog functionality is activated via the mux type gnlog in the data tag. If this

mux type is selected, for each signal that is fed into the multiplexer, a channel must be assigned in the channelMap. Channel 0 always contains the received data (RX) for GNLogging, Channel 3 contains the transmitted data (TX). Channel 1 contains additional information such as time stamps and markers. If the data is to be output as a file, the output signal (in the example GNLog_mux) must be assigned to a ByteStreamToFileEvaluator. This must then be configured accordingly via the filetype in the instance tag. Accordingly, the BinLoggingEvaluator can also be used to output triggered data. The source signals that enter the multiplexer depend on which logger is used to process the GNLog data were recorded. Loggers of the company G.i.N. provide all data as pseudo CAN frames. These can be used with The source signals that enter the multiplexer depend on which logger was used to record the GNLog data. Loggers from G.i.N. provide all data as pseudo CAN frames. These can be read with a CANToByteStreamEvaluator into ByteStreams and then transferred to the multiplexer. The CAN IDs can be taken from the DBC provided by the G.i.N. software. With BluePiraT loggers, the signals are directly available. EthernetRxn and EthernetTxn contain the RX and TX data. TX data (n stands for the corresponding Ethernet channel configured in the logger). The signal SystemGNLog signal contains additional information for GNLog channel 1 and the time stamp must be generated with the GNLogGeneratorEvaluator.

```
<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>ByteStreamMultiplexer</evaluator>
    <description short="GNLog_mux" />
    <noLog />
    <instance>
      <signal name="GNLog_bytes_rx" />
      <signal name="GNLog_bytes_tx" />
      <signal name="GNLog_LoggerInfo" />
      <signal name="GNLog_Marker2" />
    </instance>
    <content>
      <data out="GNLog_stream" mux="gnlog" />
      <channelMap>
        <channel to="0"/>
        <channel to="3"/>
        <channel to="1"/>
        <channel to="1"/>
      </channelMap>
    </content>
  </template>
</templates>
```

Listing 50: File GNLog.tem

```
<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>ByteStreamToFileEvaluator</evaluator>
    <description short="File" />
    <noLog />
    <instance filetype="gnlog">
      <signal name="GNLog_stream" />
    </instance>
    <content>
      <data out="gnlog.txt" zip="false" />
    </content>
  </template>
</templates>
```

Listing 51: File FileWriter.tem

```

<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>LoggingEvaluator.BinLoggingEvaluator</evaluator>
    <description short="FileWriter" />
    <instance filetype="gnlog"/>
    <content signalName="GNLog_stream" preBytes="100000" postBytes="100000"
filename="gnlog.txt" zip="false" />
  </template>
</templates>

```

Listing 52: File FileWriterTriggered.tem

10.4.2 ByteStreamToFileEvaluator

The ByteStreamToFileEvaluator is used to write binary data streams to a file.

The zip attribute can be used to specify whether the output file should be compressed or not. The file extension of the created file can be specified in the out attribute of the <data/> tag. (Siehe auch 10.4.4)

10.4.3 ByteStreamToStringEvaluator

The ByteStreamToStringEvaluator can be used to convert the data from a binary data stream to individual lines of text. The line breaks are recognised. Time stamps can be added and various output formats are supported.

In the following example, the format attribute sets the output format to plain. This means that the text is converted unmodified into strings. Only one line separation takes place. The characters CR (#13) and LF (#10) and combinations of them are recognized as line breaks. If the hex attribute is set to true, a hex string is generated. The attribute out can be used to stipulate the name of the output signal. If this specification is omitted, log is used as the name.

```

<?xml version="1.0" encoding="UTF-8"?><template>
  <evaluator>ByteStreamToStringEvaluator</evaluator>
  <description>Test</description>
  <content>
    <data format="plain"/>
  </content>
</template>

```

Listing 53: File bst.tem

If the attribute lineSep is set to sinn then the line marking of the Sinn company will be used. 0xD5, 0xE6 are identified as line starts and 0xE5, 0xF6 as line ends.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>ByteStreamToStringEvaluator</evaluator>
  <description>Test</description>
  <content>
    <data format="plain" lineSep="sinn" hex="true"/>
  </content>
</template>

```

Listing 54: File bst.tem

10.4.4 CanToByteStreamEvaluator

The CanToByteStreamEvaluator can be used to evaluate the data that is transmitted via CAN, into a byte stream. This is required e.g. if a TextTrace is recorded via CAN to search the text for it or similar.

In this example the data of CAN 1 and the Id 16 are used. The data stream is provided in the data signal. The second template then writes the data to a file. For possible preprocessing of the considered messages, the optional inSignal a message-based input signal can be defined. In the absence of a definition, canMessage is used. (Siehe auch 10.4.2)

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>CanToByteStreamEvaluator</evaluator>
  <description>Test</description>
  <content>
    <message id="16" can="1" out="data" inSignal="canMessage" />
  </content>
</template>
```

Listing 55: File ctbs.tem

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>ByteStreamToFileEvaluator</evaluator>
  <description>Test</description>
  <instance><signal name="data"/></instance>
  <content>
    <data zip="false"/>
  </content>
</template>
```

Listing 56: File bstf.tem

10.4.5 GNLogGeneratorEvaluator

The GNLogGeneratorEvaluator is used to generate special signals which are related to the GNLog protocol of the company Harman/Becker. Depending on the logger that has recorded the GNLog data, these signals may not have to be generated separately.

If the GNLog data was recorded with a logger from the company G.i.N., virtually all the signals in the data stream are delivered in the form of pseudo CAN messages. Only the marker signal still has to be adapted in such a way so that it can be written directly into the output bytestream. In principle, any signal can be used to generate the marker (in the example GNLog_Marker). At each point where this signal delivers a value, a GNLog marker is then triggered. With loggers from the company G.i.N., a certain pseudo-CAN message is provided, which should be used for this purpose. The CAN-ID can be taken from the DBC, which is generated by the G.i.N. tools. The signal GNLog_Marker2 can then be taken over into a ByteStreamMultiplexer which takes the GNLog data (see ByteStreamMultiplexer documentation). For data generated with a BluePiraT, a time stamp must also be generated. The timer tag causes a corresponding time stamp to be generated every second.

```

<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>GNLogGeneratorEvaluator</evaluator>
    <description short="GNLog_Generator" />
    <noLog />
    <content>
      <marker name="GNLog_Marker" out="GNLog_Marker2"/>
      <timer out="GNLog_Timer"/>
    </content>
  </template>
</templates>

```

Listing 57: File GNLogGen.tem

10.5 CAN

10.5.1 BAPEvaluator

The BAPEvaluator is used to process the information transmitted via BAP as signals and make it available for subsequent operations. The BAP protocol itself can also be monitored with regard to timings, time-out, etc.

In the example, the number of the CAN channel to be analyzed is specified in the `<bapAddrParam/>`. Optionally the input signal can be defined in the attribute `inSignal`. If no definition is given, `canMessage` is used. In the tag `<bapError/>` the attribute `all="off"` or `all="on"` switches the protocol monitoring on or off. If this is switched on, the protocol itself is monitored, and if errors occur, an event is generated. Each error has a number and with the attribute `e7="off"` can e.g. the recognition of the error with the number 7 can be switched off. In this way, known errors that occur repeatedly and have been known for a long time can be hidden from the logs. The function catalogues are specified in the tag `<functionCatalog/>`. Here, for example, `bap-fc.xml`. The function catalogues contain the description of the BAP messages related to the BAP-specific logical control unit ID. This includes, for example, the information from which logical control unit ID a display signal is sent and at which bit position it is transmitted. Several function catalogues can be specified. In the tag `<functionCatalog/>`, the generation of signals can be switched on with the attribute `makeSignals="true"`. The function catalogues are essentially identical to the function catalogues used with MOST. BAP can also be understood as a mapping from MOST to the physical layer CAN bus. The basic software description files are specified in the tag `<bapXML/>`. They represent the connection between the logical ECU IDs and the names of the CAN messages via which the information is transmitted via CAN. It should be noted that a suitable CAN DBC must also be connected, since only the names of the CAN messages are actually stored in the basic software description files. of the CAN messages is contained in the basic software description files. The concrete ID of the CAN messages must be determined via a DBC.

```

<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>BAPEvaluator.CANBAPEvaluator</evaluator>
    <noLog />
    <content>
      <bapAddrParam can="8" inSignal="canMessage" />
      <bapError all="off" />
      <bapXML>
        <file name="bap_sw.xml" stgName="HCA" />
      </bapXML>
      <functionCatalog makeSignals="true">
        <fc name="bap-fc.xml" />
      </functionCatalog>
    </content>
  </template>
</templates>

```

Listing 58: File bap.tem

In the previous example, a BAPEvaluator is configured for CAN. If BAP is to be used in IP, an EthBAPEvaluator must be used instead of the CANBAPEEvaluator.

```

<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>BAPEvaluator.EthBAPEvaluator</evaluator>
    <description short="bap" />
    <noLog />
    <content>
      <bapAddrParam />
      <bapLog mode="off" interactive="false" />
      <bapError all="off" />
      <bapXML>
        <file name="BAP_SW18_MIB3_OI_MQB37W_AP_V1_11_F.xml" stgName="" />
      </bapXML>
      <arxml file="MQB2020.MQB_W_V9.07F_AR421_20190529_ZR_MIB_OPTION_ab_Gen3.arxml" />
      <functionCatalog>
        <fc name="BAP-FC_Audio_P30DF50_v3.02_F.nxml" />
      </functionCatalog>
      <output bapSignalOut="false" bapMessageOutName="bapMessages" />
    </content>
  </template>
</templates>

```

Listing 59: File bap_eth.tem

10.5.2 CANBusLoadEvaluator

This is an evaluator that calculates the bus load on a CAN based on the messages sent via this bus. Stuffing bits are taken into account. However, data frames are always assumed, even if there are should actually be a remote frame. This may result in a slight deviation from the actual Number of stuffing bits lead.

In the example, the `interval` attribute in the `<can/>` tag specifies the period over which the Bus load is to be averaged. With a value of 1.0 the bus load is determined once per second. In the example, this value is output in the `out` signal, which is specified in the `<result/>` tag in the `name` attribute. In the tag `<can/>` in the attribute `num` the CAN number is specified and in the attribute `baud` the baud rate is specified on this CAN (here 500k). For possible preprocessing of the considered messages, the optional attribute `inSignal` in the tag `<can/>` a message based input signal can be defined, over which the bus load is determined. For example, the output signal of the `FrameFilterEvaluator`, which offers extensive filter options, can be selected. (Siehe auch 10.5.5)

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>CANBusLoadEvaluator</evaluator>
  <result name="out"/>
  <content>
    <can num="1" intervall="1.0" baud="500000"/>
  </content>
</template>
```

Listing 60: File test.tem

In order to also take the baud rate of CAN FD into account in the bus load calculation, the Parameter "baudRateFD" can be used.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>CANBusLoadEvaluator</evaluator>
  <result name="out"/>
  <content>
    <can num="3" intervall="1.0" baud="500000" baudRateFD="1000000"/>
  </content>
</template>
```

Listing 61: File test.tem

10.5.3 CANCheckEvaluator

This is an evaluator that checks whether the actual CAN assignment in the measurement matches that determined by the DBCs used and their channel assignment. For this purpose, the messages from the DBCs are used that are unique with regard to the channel assignment. If these messages appear on other CANs, a warning message is generated. Since all the necessary informations are generated from the used DBCs, no configuration of the evaluator is necessary.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<template>
  <evaluator>CanCheckEvaluator</evaluator>
  <description>CanCheck</description>
  <content/>
</template>
```

Listing 62: Example

If all messages are sent on the expected CANs, no error message will be generated.

If messages are sent on a wrong CAN, a corresponding event will be generated

```
Es kommen auf CAN 2 viele Botschaften die auf CAN 1 erwartet wurden.
```

Listing 63: Example

If two CAN channels are swapped, two error messages are generated accordingly.

```
Es kommen auf CAN 2 viele Botschaften die auf CAN 1 erwartet wurden.
Es kommen auf CAN 1 viele Botschaften die auf CAN 2 erwartet wurden.
```

Listing 64: Example

10.5.4 CANMessageTimeoutEvaluator

This is a special evaluator that can monitor CAN messages for TimeOuts. Via the `cycles` attribute in the Tag `<dbc/>` the timeout time can be made dependent on the cycle time stored in the DBC. Here the timeout time has been set to 11 cycles. It is also possible to specify here in the attribute `min` a minimum Timeout time can be set, which must not be fallen short of. If, for example, messages with very short Cycle time e.g. 7ms should be monitored with only 500ms, this can be done with the attribute `min="0.5"` can be done. Likewise, a maximum timeout time of 2 seconds can be set with `max="2.0"` can be set. Furthermore, there is the possibility, in the `factor` attribute to multiply a factor onto the specified timeout time. `factor="2"`) In the `<message/>` tag, the `name` attribute contains the name specified in the message, which is to be monitored.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<template>
  <evaluator>CANMessageTimeoutEvaluator</evaluator>
  <description>TestTemplate</description>
  <content>
    <dbc cycles="11"/>
    <message name="ant.TestMessage1"/>
  </content>
</template>
```

Listing 65: Simple example

If a DataCollector is present, the events are also written to the DataCollector.

If a shaft fails for a time greater than the set timeout time, an error message is generated.

If a message fails completely, usually no error message is generated. On this In this way, different types of shoring can be taken into account. Misses a message completely, it is assumed that the sending control unit is not present.

This behavior can be changed if in the `<message/>` tag the `mandatory` attribute is set to `true` is set. Then an error message is also generated in case of a total failure. This attribute would be set for control units that are installed in any case.

```
<?xml version="1.0" encoding="UTF-8"?>
<message name="ant.TestMessage1" mandatory="true" />
```

Listing 66: Example

In addition, a signal can be specified in the `enable` attribute, which the monitoring is activated. This could be e.g. a KL15 signal, if the Control unit sends only at KL15-On.

```
<?xml version="1.0" encoding="UTF-8"?>
<message name="ant.TestMessage1" enable="ant.enable" />
```

Listing 67: Example

If a message fails completely, and the `mandatory` attribute is set to `true` is set and an `enable` signal is specified, for each `enable` on phase an error is generated.

The following special case should be noted: If there are failures between the last `enable-on` and the current `enable-off`, no errors will be because in this case it is not sure when exactly the `enable` went out. An analog KL15-out may reach an ECU long before the corresponding CAN-signal is sent. The control unit will then stop transmitting even though the last `Enable On` is still valid on the CAN. This should not result in an error message.

In the `<errors/>` tag, an upper limit for the number of errors can be specified in the `limit` attribute. In this way, it is possible to prevent a control unit from flooding the

error report with error messages. In the example, the maximum number of errors to be displayed is limited to 5.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<template>
  <evaluator>CANMessageTimeoutEvaluator</evaluator>
  <description>TestTemplate</description>
  <content>
    <errors limit="5"/>
    <dbc cycles="9"/>
    <message name="ant.TestMessage1" enable="ant.enable"/>
  </content>
</template>
```

Listing 68: Example

10.5.5 FrameFilterEvaluator

The `FrameFilterEvaluator` filters messages from a message-based input signal (CAN or FlexRay) according to defined filter pattern. The usage is analogous to the filters in the `FrameLoggingEvaluator`.

A filter is defined by a range of message IDs and a channel. In the following example, only messages with IDs from 200 to 400 and from 700 to 800 (inclusive) are written. If in the `<filter/>` tag additionally the `exclude` attribute is set to `true`, the filter is inverted. Thus, all messages to which the filter parameters do not apply are written. Basically, if multiple filters are specified, all messages to which a filter condition applies are written. If further in the `<filters/>` tag the `exclude` attribute is set to `true`, the total filter resulting from the combination of all individual filters is inverted.

The `name` attribute of the `<output/>` tag is used to specify the name of the generated output signal. The `FrameFilterEvaluator` is applicable to CAN, FlexRay and Ethernet messages. This must be done in the `<output/>` tag the attribute `type` can be set to the value "CAN", "FR" or "ETH". Ethernet frames can only be filtered by channel. The `from` and `to` attributes remain here without effect.

For possible preprocessing of the messages to be filtered, the optional `<signal/>` with the `name` attribute a message-based input signal can be defined. For this for example, the output signal of another `FrameFilterEvaluator` can be selected.

```
<instance><description name="FrameFilterEvaluatorTest"/>
  <signal name="flexRayFrame"/>
</instance>
<content>
  <output name="out_signal" type="FR"/>
  <filters exclude="false">
    <filter from="200" to="400" channel="1"/>
    <filter from="700" to="800" channel="1"/>
  </filters>
</content>
```

Listing 69: Use of frame filters

10.6 Diagnosis

10.6.1 DiagMessageEvaluator

The `DiagMessageEvaluator` is used to enter diagnostic information into the Result of the evaluation and to be transferred to the metadata database. As Input signals usually serve the output signals of the Diagnostic Protocol Evaluators. (Siehe auch 10.6.1, 10.6.10, 10.6.1)

As you can see in the example, there is no further need to consult the evaluator required. Several input signals can be specified if multiple diagnostic protocol evaluators are used.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Diagnose.DiagMessageEvaluator</evaluator>
  <instance><signal name="in"/></instance>
  <description>Beispiel</description>
  <content />
</template>
```

Listing 70: File test.tem

By setting the “writeXml” attribute to false, writing the interpreted Diagnostic data into the main AMeDA results document can be disabled. At the same time, no events are generated.

10.6.2 DiagMessageEvaluatorOdx

The DiagMessageEvaluatorODX is used to enter diagnostic information into the Main results document of the evaluation, to be saved in the metadatabase or as separate logs. Only the values interpreted by the evaluator “UDSEvaluator (ODX)” can be processed.

The simple example shows the output of diagnostic data to the metadata database and main results document. In addition to the specification of one (possibly several) input signals, the specification of the Table name, which is used for creating the tables in the database, is required. By setting the “writeXml” attribute, the writing of the interpreted Diagnostic data into the main AMeDA results document can be controlled. The names of the tables created by this evaluator consist of the specified table name and a suffix together. For the diagnostic responses table, the suffix is “responses”, for the table with the diagnostic requests it is “requests” and for the table with the interpreted diagnostic parameters “params”. If no table name is specified, the tables are “diagresponses”, “diagrequests” and “diagparams”.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Diagnose.DiagMessageEvaluatorOdx</evaluator>
  <instance><signal name="in"/></instance>
  <description>Beispiel</description>
  <content tableName="diagtable" writeXml="true"/>
</template>
```

Listing 71: File test.tem

The output of separate diagnostic logs in an ODIS format is to be defined in the content via tag <protocol/>. In addition to the file name, the Timeout value must be taken into account. Suspends the input signal for the defined time, a new protocol is started. For the output of the error memory entries the behavior can be configured. When deletion requests/responses are actively taken into account, all errors are deleted in the log every time and thus only the errors after the last deletion are output. If this option is deactivated, the errors in the log will never be deleted and thus all errors read out will be output. However, the individual errors are listed only once, despite the fact that they can occur several times in the data. Furthermore, a signal can be defined to control the output of multiple protocols.

10.6.3 FIBEX AUTOSAR

This evaluator decodes diagnostics between a tester and an ECU communicating via FlexRay. The output is an object of type TPMessage. A subset of the AUTOSAR protocol is supported. In particular, only the variant is supported, where the source and

destination addresses are each one byte long. To find the positions of the diagnostic messages within the FlexRay messages, a FIBEX file is required. To ensure backward compatibility with existing diagnostic evaluators, a file with additional information can be specified (the well-known diaginfo.xml). Using the information from this file, the addresses are mapped to the K-Line 5 baud addresses, where possible, and the ECU names are matched to those in the file.

The following example shows how to condition the evaluator.

```
<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>Diagnose.fibex.FibexAutosarTpEvaluator</evaluator>
    <description short="neuer Name" />
    <noLog />
    <content channel="1" fibexFile="fibex.xml" fibexChannel="MLB_FlexRay_Kanal_A" diag
out="tpMessage" />
  </template>
</templates>
```

Listing 72: File test.tem

10.6.4 ISOCAN

To establish the ISOCAN communication between two ECUs or between an ECU and a diagnostic tester analyze, the ISOCAN evaluator can be used. In the process, the ISOCAN messages are analyzed and turned into TPMessages. The output is an object of type TPMessage. Thereby single frames are immediately is output as a TPMessage, and first frames and the following consecutive frames are combined into a TPMessage and forwarded as soon as the TMessage is complete.

In the following example, the required template is shown first. This essentially contains in the can attribute the number of the can channel to be analyzed. Optionally also in the attribute inSignal the input signal can be defined. In the absence of a definition, canMessage is used. The out attribute specifies the name of the signal used to output the TP messages. In the diagInfo attribute, a description file is specified that contains the required information to the individual ISOCAN channels. In the example, the attribute name specifies the name of the ISOCAN channel, and the attributes UDSRequest and UDSResponse the two CAN identifiers which are used for the communication.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Diagnose.ISOCAN.ISOCANEvaluator</evaluator>
  <description>Test</description>
  <content can="1" diagInfo="diaginfo.xml" out="out" inSignal="canMessage" />
</template>
```

Listing 73: File test.tem

```
<?xml version="1.0" encoding="UTF-8"?>
<diagInfo>
  <ecu name="Motorelektronik" UDSRequest="0x10" UDSResponse="0x11" />
</diagInfo>
```

Listing 74: File diaginfo.xml

When messages are transmitted unsegmented, this must be additionally marked in the diaginfo.xml file. Since request and response can be transmitted independently of each other in unsegmented form, both can be configured separately via requestUdt and responseUdt.

```
<?xml version="1.0" encoding="UTF-8"?>
<diagInfo>
  <ecu name="Motorelektronik" UDSRequest="0x7E0" UDSResponse="0x5E8"
    requestUdt="true" responseUdt="true"/>
</diagInfo>
```

Listing 75: File diaginfo.xml

In OBD, functional IDs are used for broadcast messages. For AMeDA to take these into account, they must be listed in the description file. Broadcast messages are characterized by the fact that the indication of UDSResponse is missing. The following example shows a description file for OBD with two Control units listed.

```
<?xml version="1.0" encoding="UTF-8"?>
<diagInfo>
  <ecu name="OBD" UDSRequest="0x7DF"/>
  <ecu name="ECU1" UDSRequest="0x7E0" UDSResponse="0x7E8"/>
  <ecu name="ECU2" UDSRequest="0x7E1" UDSResponse="0x7E9"/>
</diagInfo>
```

Listing 76: File diaginfo.xml

10.6.5 IdexExport

The IdexExportEvaluator is used to convert diagnostic information into the format used by the iDEX is used. The output signals of the diagnostic protocol evaluators are usually used as input signals. (Siehe auch 10.6.5, 10.6.10, 10.6.5)

In the <content/> tag, the name of the stylesheet must be specified in the stylesheet attribute, which is to be used for the visualization of the XML report.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Diagnose.diagExport.IdexExport</evaluator>
  <instance><signal name="in"/></instance>
  <description>Test</description>
  <content stylesheet="stylesheet.xsl"/>
</template>
```

Listing 77: File test.tem

If there is more than 60 sec. between two diagnostic events, this is assumed that a new diagnostic extract has been triggered and it is another log is generated.

10.6.6 KWP2000

The KWP2000Evaluator is used to analyze KWP2000 messages exchanged via a transport protocol. The input signal is usually the output of a transport protocol evaluator. (Siehe auch 10.6.6, 10.6.6)

In the following example, the required template is shown first. In the instance the name of the signal is specified in the <signal/> tag in the name attribute the messages of the transport protocol are received. In the <content/> tag, the pdx attribute specifies the name of the PDX file to be used. In the attribute out the name of the signal is specified over which the decoded diagnostic information is output. Finally, the DTCTableT attribute specifies the name of the table from which the information about the TroubleCodes is taken. If the error memory is read out in SAE mode (0x18 00 ff 00) the table is used for decoding, which is specified in the DTCTableS attribute.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Diagnose.KWP2000.KWP2000Evaluator</evaluator>
  <instance><signal name="in"/></instance>
  <description>Test</description>
  <content DTCTableS="dtcTableS.xml" DTCTableT="dtcTableT.xml" out="out"/>
</template>
```

Listing 78: File test.tem

If errors occur during the interpretation of diagnostic messages, these are recorded as errors in the log saved, but the evaluation is not marked as defective on the MeDaPF.

10.6.7 TP16

To establish the TP16 communication between two ECUs or between an ECU and a diagnostic tester analyze, the TP16 evaluator can be used. First of all, the opening identifiers are used to determine the Message IDs of the actual transport channels negotiated. After that, the transfer of the data takes place via the Transport channels take place. It should be noted that the opening communication, which was used on the K-Line and also is sent at KWP on TP 1.6 is filtered. This is therefore not output on the output signal. With this the output signal can be used directly to analyze the diagnostic communication.

In the following example, the required template is shown first. This contains essentially in the attribute `can` the number of the can channel which is to be analyzed. In the `out` attribute the Name of the signal via which the TP messages are output. The opening identifier of the Control unit is obtained by adding the TP16 address. In the `diagInfo` attribute a description file is specified which contains the required information about the individual TP1.6 channels. In the example, the attribute `name` contains the name of the TP1.6 channel, and the attributes `tp16adr` the the TP1.6 address of the participant. In the attribute `speed` the speed of the CAN bus is specified (high or low). This information is needed to determine the address of the transport channel, because this is not completely transmitted via the bus with TP1.6.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Diagnose.TP16.TP16Evaluator</evaluator>
  <description>Test</description>
  <content can="2" diagInfo="diaginfo.xml" out="out" speed="low"/>
</template>
```

Listing 79: File test.tem

```
<?xml version="1.0" encoding="UTF-8"?>
<diagInfo>
  <ecu name="Klimabedienteil hinten" tp16adr="30" />
</diagInfo>
```

Listing 80: File diaginfo.xml

10.6.8 TP20

To establish TP20 communication between two control units or between a control unit and a diagnostic tester analyze, the TP20 evaluator can be used. First of all, the opening identifiers are used to determine the Message IDs of the actual transport channels negotiated. After that, the transfer of the data takes place via the Transport channels take place.

In the following example, the required template is shown first. This contains essentially in the attribute `can` the number of the can channel which is to be analyzed. Optionally also in the attribute `inSignal` the input signal can be defined. In the

absence of a definition, `canMessage` is used. The `out` attribute specifies the name of the signal used to output the TP messages. The `startID` attribute specifies the tester's opening identifier. The opening identifier of the Control unit is obtained by adding the TP20 address. In the `diagInfo` attribute a description file is specified, which contains the required information about the individual TP20 channels. In the example, the `name` attribute specifies the name of the TP20 channel, and the `tp20adr` attributes specify the the TP20 address of the subscriber.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Diagnose.TP20.TP20Evaluator</evaluator>
  <description>Test</description>
  <content can="1" startID="0x200" diagInfo="diaginfo.xml" out="out" inSignal="canMessage" />
</template>
```

Listing 81: File test.tem

```
<?xml version="1.0" encoding="UTF-8"?>
<diagInfo>
  <ecu name="Motorelektronik" tp20adr="0x01" />
</diagInfo>
```

Listing 82: File diaginfo.xml

10.6.9 UDS

The `UDSEvaluator` is used to analyze UDS messages that are exchanged via a transport protocol. The input signal is usually the output of a transport protocol evaluator. (Siehe auch 10.6.4, 10.6.8)

In the following example, the required template is shown first. In the instance, the `name` attribute in the `<signal/>` tag specifies the name of the signal over which the messages of the transport protocol are received. In the `<content/>` tag, the `pdx` attribute specifies the name of the PDX file to be used. In the attribute `out` the name of the signal is specified over which the decoded diagnostic information is output. Finally, the `DTCTableT` attribute specifies the name of the table from which localized information about the `DisplayTroubleCodes` is taken. (In the ODX file you will find only the symptom code incl. a description and the `DisplayTroubleCode`. The description of the `DisplayTroubleCode` is taken from a table that is available on the Tester is available in the respective national language)

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Diagnose.UDS.UDSEvaluator</evaluator>
  <instance><signal name="in"/></instance>
  <description>Test</description>
  <content pdx="pdx.pdx" DTCTableT="dtcTable.xml" out="out"/>
</template>
```

Listing 83: File test.tem

If errors occur during the interpretation of diagnostic messages, these are recorded as errors in the log saved, but the evaluation is not marked as defective on the MeDaPF.

10.6.10 UDS (ODX)

The “`ODX.UDSEvaluator`” can be used to analyze UDS messages similar to the “`UDSEvaluator`”. Both evaluators differ in the way the UDS messages are interpreted by The “`ODX.UDSEvaluator`” uses the ODX standard and is therefore more flexible with respect to on the interpretation of the various UDS diagnostic services. (Siehe auch 10.6.9)

The following example shows the use of the evaluator. In the <signal/> tag of the <instance/> subtag, the name of the signal over which the transport protocol messages are received is specified. In the <content/> tag, the pdx attribute specifies the name of the PDX file to be used. In the out attribute, the name of the signal is specified via which the decoded diagnostic information is to be output.

For the interpretation of the diagnostic data, the appropriate ECU variant is automatically selected from the PDX file. This works only with existing queries of ODXFileIdentifier (0xF19E) and ODXFileVersion (0xF1A2). The name of an ECU variant can be specified in the variant attribute. If this is used, there is no automatic selection of control unit variant, the specified variant is always used. In the usingBV attribute, the use of base variants can be allowed if the required EV variant is not present in the specified PDX file.

In the tag <sids/> one or more start bytes can be specified. The specification is expected as hex code. The hex codes can optionally be separated with spaces. The evaluator interprets only all responses from queries that start with one of the specified start bytes. In the example the start byte 22 is specified, which according to the protocol corresponds to the service identifier 22 (ReadDataByIdentifier). The evaluator thus ignores all diagnostic responses that do not originate from requests with this start byte or service identifier. If no start bytes are specified, all diagnostic responses are interpreted.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Diagnose.UDS.ODX.UDSEvaluator</evaluator>
  <instance><signal name="in"/></instance>
  <description>Test</description>
  <content pdx="file.pdx" out="out" variant="FG_WWH_OBD">
    <sids>
      <sid>22</sid>
    </sids>
  </content>
</template>
```

Listing 84: File test.tem

There may be several appropriate interpretations to a diagnostic answer. When using the evaluator "DiagMessageEvaluatorOdx" the short names of the diagnostic services used for the interpretation will be issued. The services can be used to decide in the report template which interpretations should be included in the Report should be displayed.

10.7 Ethernet

10.7.1 DoIPFilterEvaluator

The DoIPFilterEvaluator examines Ethernet frames for the DoIP protocol. If an Ethernet frame represents itself as DoIP frame, this is converted accordingly and output in a new signal.

To use the template, an input signal containing Ethernet frames must be selected. About the channelNum attribute, the channel number to be examined is defined.

In the following example a new signal with the name "doip" is output.

```

<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>Diagnose.DoIP.DoIPFilterEvaluator</evaluator>
    <description />
    <instance>
      <signal name="ethernetFrame" />
    </instance>
    <content channelNum="1">
      <ports/>
      <doip name="doip"/>
    </content>
  </template>
</templates>

```

Listing 85: File test.tem

In order to use the DoIP messages with the UDSEvaluator, the diagnostic messages must be extracted. If the <tpmessage/> tag is set, the UDS messages are pulled from the DoIP messages.

The name attribute can be used to define the name of the output signal.

To assign the node addresses to the ECU's, an XML file (diagnostic info) is required. In the XML document the node addresses can be set in the <ecu/> tag via the nodeAddr attribute. It is used by the hexadecimal notation. The diagnostic info XML has the same structure as in the other diagnostic evaluators.

Below is an example of a template where the diagnostic messages are output in the "tpout" signal.

```

<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>Diagnose.DoIP.DoIPFilterEvaluator</evaluator>
    <description />
    <instance>
      <signal name="ethernetFrame" />
    </instance>
    <content channelNum="2">
      <ports/>
      <doip name="doip"/>
      <tpmessage name="tpout" diagInfo="diaginfo.xml"/>
    </content>
  </template>
</templates>

```

Listing 86: File test.tem

10.7.2 EthernetFilterEvaluator

The EthernetFilterEvaluator allows the filtering of Ethernet frames. Several filter conditions can be specified. Frames are passed if at least one of them is satisfied. VLAN IDs or network address pairs (source and destination) can be specified as conditions. A network address is a pair of IP address and TCP/UDP port. If the complete communication between two participants is to be analyzed, the bidirectional flag can be set. Otherwise, only communication in one direction is allowed through.

In the following example, only frames transmitted on the VLAN with ID 2 are allowed through. Furthermore, a filter for network addresses is set up so that all communication between the appropriate participants is allowed through. For IP addresses, asterisks can be used as wildcards. If you do not want to filter by a specific port, this parameter can be omitted (or as an empty string). If the <exclude/> tag is set to true, the entire filter will be inverted.

```

<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>EthernetFilterEvaluator</evaluator>
    <description />
    <instance />
    <content outputSignal="filteredEthFrames" exclude="false">
      <vlan id="2" />
      <ipFilter bidirectional="true">
        <source address="*:*:*:*:*:*:004f" port="" />
        <dest address="*:*:*:*:*:*:000a" port="" />
      </ipFilter>
    </content>
  </template>
</templates>

```

Listing 87: File filter.tem

The following example filters by MAC address, with all outgoing communication of the corresponding network subscriber being allowed through. The tag `<etherFilter/>` can also be used to filter for “EtherType”. This is a 2-byte value, which denotes the protocol contained in the frame (e.g. 0x0800 for IPv4). As with the IP filter, the `bidirectional` flag can be set if both communication directions are to be filtered. If a parameter is left empty, it will be ignored during filtering.

```

<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>EthernetFilterEvaluator</evaluator>
    <description />
    <instance />
    <content outputSignal="filteredEthFrames" exclude="false">
      <etherFilter bidirectional="false" source="02:7D:FA:01:06:00"
        dest="" etherType="" />
    </content>
  </template>
</templates>

```

Listing 88: File filter.tem

10.7.3 EthernetTrafficEvaluator

The `EthernetTrafficEvaluator` can measure the Ethernet traffic for individual transmitters and receivers. Here you can choose between Frames and frame contents can be parameterized. Additionally, the evaluator offers the possibility to filter packages.

A `<ethernet/>` tag must be present for Ethernet traffic calculation. In this tag you can set a time window in which the traffic (load) should be calculated. In the example below, a window size of 1 second is set in the `windowSize` attribute. This value corresponds to the default value. To adjust the sampling rate for the time windows the `period` attribute is used. This is shown in the following Example set to 0.5 seconds. The default value is 1 second. This means that in the example, every 0.5 seconds a time window of 1.0 second is considered in which the Load of all frames present in it is calculated. In the output signal, at the time of the end of the time window, the load calculated in it is output.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>EthernetTrafficEvaluator</evaluator>
  <result name="out"/>
  <content>
    <ethernet windowsize="1.0" period="0.5"/>
  </content>
</template>
```

Listing 89: File test.tem

Besides the traffic calculation based on the number of frames, there is the possibility to view the frame contents. If the `usedata` attribute is set to `true`, all bytes of the Ethernet packets in the Traffic calculation used. In the output signal, the number of bytes is then divided by the size of the time window to obtain the Number of bytes per second.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>EthernetTrafficEvaluator</evaluator>
  <result name="out"/>
  <content>
    <ethernet windowsize="1.00" period="1.0" usedata="true"/>
  </content>
</template>
```

Listing 90: File test.tem

In order to filter certain transmitters and receivers during the load calculation, there is the possibility to define filters. The filters are collected under the `<filters/>` tag. Each `<filter/>` tag that is entered there represents represents a filter of its own. The attributes of the `<filter/>` tag can be used to set where to filter for shall. In addition, there is the possibility to specify via the `operation` attribute whether the filters are linked with AND should be combined (`operation=and`) or with OR (`operation=or`). The default value is OR.

Currently the following filters are available:

attribute	function	example
source-mac	Filters MAC address of the sender	11:22:33:44:55:66
dest-mac	Filters MAC address of the receiver	11:22:33:44:55:66
source-ip	Filters IP address of the sender	127.0.0.1
dest-ip	Filters IP address of the receiver	127.0.0.1
vlan-id	Filter by VLAN ID	3
source-port	filters by port of the sender	1337
dest-port	filters by port of the receiver	1337
chan-num	filters by channel number	42

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>EthernetTrafficEvaluator</evaluator>
  <result name="out"/>
  <content>
    <ethernet windowsize="1.00" period="1.0" usedata="true"/>
    <filters operation="and">
      <filter source-mac="11:22:33:44:55:66"/>
      <filter dest-mac="AA:BB:CC:DD:EE:FF"/>
    </filters>
  </content>
</template>
```

Listing 91: File test.tem

10.7.4 NDPFilterEvaluator

The NDPFilterEvaluator examines Ethernet frames for the NDP protocol. If an Ethernet frame represents itself as NDP frame, this is converted accordingly and output in a new signal. To use the template, an input signal containing Ethernet frames must be selected.

In the following example a new signal with the name "ndpOut" is output.

```
<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>NDPFilterEvaluator</evaluator>
    <description />
    <instance>
      <signal name="ethernetFrame" />
    </instance>
    <content>
      <ndp name="ndpOut"/>
    </content>
  </template>
</templates>
```

Listing 92: File test.tem

10.7.5 PTPFilterEvaluator

The PTPFilterEvaluator examines Ethernet frames for the PTP protocol. If an Ethernet frame represents itself as PTP frame, it is converted accordingly and output in a new signal. To use the template, an input signal containing Ethernet frames must be selected.

In the following example a new signal with the name "ptpOut" is output.

```
<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>PTPFilterEvaluator</evaluator>
    <description />
    <instance>
      <signal name="ethernetFrame" />
    </instance>
    <content>
      <ptp name="ptpOut"/>
    </content>
  </template>
</templates>
```

Listing 93: File test.tem

10.7.6 SomeIPFilterEvaluator

The SomeIPFilterEvaluator examines Ethernet frames for the SOME/IP protocol. If an Ethernet frame represents itself as SOME/IP frame, this is converted accordingly and output in a new signal. To use the template, an input signal containing Ethernet frames must be selected.

In the following example, a new signal with the name "someip" is output.

```

<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>SomeIPFilterEvaluator</evaluator>
    <description />
    <instance>
      <signal name="ethernetFrame" />
    </instance>
    <content useStandardPorts="true">          <filters/>
      <ports>
        </ports>
        <someip name="someip"/>
      </content>
    </template>
  </templates>

```

Listing 94: File test.tem

If you know the sender and receiver ports which should be used for SOME/IP filtering, you can these are defined in a list. To do this, each port must be entered under the `<ports/>` tag. A port is defined with the `<port/>` tag and the `number` attribute sets the number of the port. Below is an example where all frames are viewed with port 30490.

```

<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>SomeIPFilterEvaluator</evaluator>
    <description />
    <instance>
      <signal name="ethernetFrame" />
    </instance>
    <content useStandardPorts="false">          <filters/>
      <ports>
        <port number="30490"/>
      </ports>
      <someip name="someip"/>
    </content>
  </template>
</templates>

```

Listing 95: File test.tem

Besides the output of the SOME/IP frames there is the possibility to pre-filter the frames. For this purpose, under the `filters` attribute defines a filter. Each filter has an `attribute` element to specify the type of the filter and define an `attribute value` around the value to be filtered. Besides, there is a possibility to determine the linkage of the filters. The `operation-` attribute for this on "and" or "or" can be set.

Currently, "serviceid" and "servicetype" are implemented as filter types. In the following example a filter which filters all SOME/IP frames with the service ID 0x601b.

```

<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>SomeIPFilterEvaluator</evaluator>
    <description />
    <instance>
      <signal name="ethernetFrame" />
    </instance>
    <content useStandardPorts="false">
      <filters filterFile="" useFile="false" operation="and">
        <filter element="serviceid" value="601b"/>
      </filters>
      <ports>
        <port number="30490"/>
        <port number="30491"/>
      </ports>
      <someip name="someip"/>
    </content>
  </template>
</templates>

```

Listing 96: File test.tem

10.7.7 TCPCheckEvaluator

The TCPCheckEvaluator analyzes TCP data streams and outputs events via a special event signal, if something special happens in a TCP session. In addition, the TCP frames can be output as a frame signal. The resulting signal then contains the frames in correct order and cleaned of duplicates.

In the following example, a TCPCheckEvaluator is initialized. The events are output to the tcpEvents signal. This signal can be viewed in MDFView or further analyzed with a JavaEvaluator. The TCP frames are output on the ipFrames signal. The tcpTimeout parameter is optional. When it is set, a connection is considered broken, if a message is not acknowledged by ACK within the corresponding time. Otherwise, connections are considered broken if outstanding ACKs remain until the end of measurement. The TCP events are output as a signal of type TcpEvent. Should they be used in a JavaEvaluator, the signal values must be cast accordingly. A TcpEvent basically has a CommunicationPath consisting of source and destination address (IP and port). It also has an EventType and different parameters depending on the EventType.

EventType	Meaning	Parameter
NO_CONNECTION	A message was received without a connection being opened.	none
CONNECTION_ESTABLISHED	The connection has been opened.	none
CONNECTION_CLOSED_BY_PARTICIPANT	The connection was closed by one of the participants.	none
CONNECTION_CLOSED	The connection was closed on both sides.	none
RST	A TCP connection was rejected (port not open).	none
REPETITION	A frame has been transmitted multiple times.	COUNTER: Number of repetitions
WINDOW_OVERFLOW	More data was transmitted without ACK than the receiver allows.	BYTES: Bytes transferred without ACK, WINDOW_SIZE: Allowed bytes
FRAME_LOST	An ACK was sent, but a frame that should be confirmed by it is missing.	none
CONNECTION_LOST	A connection failure has occurred (frame without ACK).	none

```

<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>TCPCheckEvaluator</evaluator>
    <description />
    <instance>
      <signal name="ethernetFrame" />
    </instance>
    <content eventSignal="tcpEvents" ipOutput="ipFrames" tcpTimeout="60.0" />
  </template>
</templates>

```

Listing 97: File tcpCheck.tem

10.7.8 ViWiFilterEvaluator

The ViWiFilterEvaluator examines Ethernet frames for ViWi messages. The evaluator outputs three signals in the process. One signal with HTTP frames, one signal with WebSocket frames and a signal with ViWi messages, which are derived from the HTTP and WebSocket frames result. The user must use as input signal a signal with Ethernet frames specify

In the following example the signal ethernetFrame is examined for ViWi messages.

Furthermore, there is the possibility, in the event-based communication, to consider only frames where previously an HTTP connection has been established. This is done by the requireWsProtocolSwitch attribute in the <content/> tag set. If it is set to false, all WebSocket messages with ViWi content will be interpreted. If set to true, frames will be interpreted only if the HTTP handshake has been done before. This option has the advantage that the analysis performance is significantly improved, since not all frames have to be analyzed.

10.8 Export

10.8.1 DataExtractEvaluator

The DataExtractEvaluator is used to convert measurements completely into another format.

This is a simple example that creates an MDF file. The desired file type is specified in the tag <save/> specified in the ext attribute. The signals s1 and s2 are extracted.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>DataextractEvaluator</evaluator>
  <description>Test</description>
  <instance>
    <signal name="s1"/>
    <signal name="s2"/>
  </instance>
  <content>
    <save ext="mdf"/>
  </content>
</template>

```

Listing 98: File test.tem

The boolean attribute startMdfAtZero in the <content/> tag can be used to control, whether the output file should start with a timestamp of 0s, so a corresponding offset is subtracted from all time stamps. The default value is false. This means that the file retains the original timestamps.

The name attribute can be used to specify a desired file name, where the name is extended by the name of the source file is supplemented.

If this addition is to be omitted, this can be switched off with the attribute `plain="true"`.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>DataextractEvaluator</evaluator>
  <description>Test</description>
  <instance>
    <signal name="s1"/>
    <signal name="s2"/>
  </instance>
  <content>
    <save name="hallo.mdf" plain="true" ext="mdf"/>
  </content>
</template>
```

Listing 99: File test.tem

With the attribute `ext="xls"` an Excel file can be created.

With the attribute `ext="csv"` a CSV file can be created. In this case, the other attributes `decimalSeparator`, `columnSeparator`, `completeTable` and `defaultValue` must be specified in the `<save/>` tag. With `decimalSeparator` the decimal separator for the values in the table is set,

with `columnSeparator` the separator for the table columns.

With `completeTable="true"` it is ensured that the table contains a value in each cell. Signal values that do not occur in a line (at a point in time) are filled with their predecessor value. If there is no predecessor, the value specified by `defaultValue` is used.

If no signals are specified, all existing signals will be exported. The optional `exclude` attribute can be used to specify, that all signals except the selected ones should be exported.

With the specification of `ext="MAT"` the Matlab format can be generated, which is also used by the vector tools is generated. This is done for each Signal generates a $1 \times n$ matrix, and the respective name of the matrix ends with a `t[m]`. This `t[m]` specifies another $1 \times n$ matrix containing the time points belonging to the signal values.

With the specification of `ext="MAT2"` a Matlab file can be generated in a slightly different format. Here, all signals of a group are written into a matrix together with the corresponding time points. Signals form a group if the respective times of all signal values are identical.

When an MDF file is written, the corresponding tags can be used to fill the fields of the MDF file header.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>DataextractEvaluator</evaluator>
  <description>Test</description>
  <instance>
    <signal name="s1"/>
    <signal name="s2"/>
  </instance>
  <content>
    <author>Author</author>
    <comment>Comment</comment>
    <organization>Organization</organization>
    <identification>Identification</identification>
    <project>Project</project>
    <save ext="mdf"/>
  </content>
</template>
```

Listing 100: File test.tem

10.8.2 DataExtractRPC3Evaluator

The DataExtractRPC3Evaluator is used to completely convert measurements into the Export RPC3 format. The RPC3 format supports only signals of the same Type and same sampling rate within one file. As type is always FLOATING_POINT selected. Not all software that is capable of running RPC3 is read supports this format. To set the sampling rate, the time interval between the individual samples (deltaT). When not all signals start at the same time, the signals that start later are Signals padded with values from defaultValue. When the number of sampled measured values is not a multiple of 256, at the end of the file so many measuring points with the value defaultValue added, that a Multiple of 256 is reached.

In this example, a file is written that contains all signals. The signals are sampled with an interval of 0.1 seconds and missing values filled with 0.0. The zip attribute can be used to specify, whether the RPC 3 file to be created should be packed into a ZIP archive or not.

```
<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>DataextractEvaluator.DataExtractRPC3Evaluator</evaluator>
    <description short="Extract" />
    <instance />
    <content>
      <save zip="false" name="out" exclude="false" deltaT="0.1" defaultValue="0.0" />
    </content>
  </template>
</templates>
```

Listing 101: File rpc3Extract.tem

10.8.3 DataExtractSimpleEvaluator

The DataExtractSimpleEvaluator is used to convert measurements into a special binary format. Similar to TDM or ATF/XML, a binary file is created that contains only double values, and an additional XML file containing the corresponding description information. Thereby for each signal group the offset is stored in the binary file, where the measured values of group start

Here is a simple template which exports the signals s1, s2, s5 and s6.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>DataextractEvaluator.DataExtractSimpleEvaluator</evaluator>
  <description>Test</description>
  <instance>
    <signal name="ant.s1"/>
    <signal name="ant.s2"/>
    <signal name="ant.s5"/>
    <signal name="ant.s6"/>
  </instance>
  <content>
    <save name="test"/>
  </content>
</template>
```

Listing 102: File test.tem

With the attribute zip="true" in the <save/> tag, the binary file can be compressed with GZIP. Since the Data is stored sorted by signals, a significant size reduction can be expected.

10.8.4 FileExportEvaluator

The FileExportEvaluator is used to output data in file form. The MDF 4.1 (signal-based) and BLF and ASC (message-based) formats are supported. In addition, signal-based values can be written to a metadata database table. With the Evaluator, the entire measurement can be exported as well as partial measurements with pre- and post-trigger. The selection of the signal sections to be exported is done via a "Enable signal".

The Evaluator offers various settings, with which the output of the data can be influenced in the most different way. Especially the export to the MDF4 format can be influenced very strongly, which can have a big impact on both writing and subsequent reading performance. The parameters from the following table can be added as attributes to the `<content/>` tag. Some of them will also be revisited and described in more detail below.

Parameter	Possible Values	Description
fileName	file name	output file name
type	mdf mf4 blf asc db	format to export
compression	zip gzip none	Compresses the result files in the appropriate format
onlySamplingGroups	true false	Only the sampling groups will be considered for export, all other signals will be ignored (regardless of other filters set)
groupCriteria	one_group time message_channel	Criterion by which the signals are divided into groups. For some formats this setting is ignored.
maxGroupSize	Number	Maximum size of a signal group, signals with the same group criterion are separated if necessary. This can have advantages for the read performance of the generated file.
pattern	RegEx	A regular expression for filtering signals
prefix	text	A prefix that is truncated at the beginning of each signal name (RegEx possible)
timeBits	number	number of bits used for a timestamp
timeResolution	number	time resolution
enable	signal name	name of a signal that serves as a switch for export
pre	number	How many seconds are exported before a positive edge of the Enable signal
post	number	How many seconds are exported after a negative edge of the Enable signal
keepStartTimes	true false	The timestamps are kept, otherwise each file would start at 0s. The correct absolute timestamp is nevertheless stored in the file header if the file format provides for it. ATTENTION: If the timestamps are read in as Unix timestamps, for example, they can become very large and overflows can occur in the timestamp.
writeStartValues	true false	Writes the last valid signal value at each new enable cycle to the start
writeMessageNames	true false	Writes message names to the metadata fields of an MDF file
writeMessageIds	true false	Writes message IDs to the metadata fields of an MDF file
writeChannelNames	true false	Writes channel names to the metadata fields of an MDF file
writeChannelIds	true false	Writes channel numbers to the metadata fields of an MDF file
mdfZip	true false	use internal MDF compression (only for MDF4)
mdfMemoryManagement	true false	Enables a function that optimizes memory usage when writing MDF

Parameter	Possible Values	Description
fileName	file name	output file name
mdfBuffSizes	number	Specifies a maximum memory size to be filled with signal buffers
mdfRecsPerBlocks	number	Specifies the maximum number of records to be written to a block. One block per signal group must always be kept in memory.
writeFramesDirectly	true false	All signals coming from CAN messages are written more efficiently by copying the message content 1-to-1.
canSignal	signal name	signal containing all CAN messages to be considered
writeEmptyFrameFiles	true false	Specifies during frame export whether empty files should also be created
writeEmptyBinFiles	true false	Determines whether empty files should also be created during binary export

In this example, data is written away whenever the signal e is greater than 0.5. In addition, 1.5s before the rising edge and 1.5s after the falling edge of e data is also written away.

```
<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>FileExportEvaluator</evaluator>
    <instance />
    <content pre="1.5" post="1.5" enable="e" type="mf4" keepStartTime="true" />
  </template>
</templates>
```

Listing 103: File test.tem

In the case of signal-based export, it is still possible to specify a criterion according to which the signals are grouped. Options are “time” and “message_channel”. The default behavior (“time”) groups the signals by time data source. All signals that have exactly the same time stamps are then assigned to a group. The alternative is that the signals are sorted by additional information from the signal properties. For example, signals are assigned to a group if they have the same CAN ID or have been in a signal group in a source MDF file. Grouping can have an impact on file size and efficient readability of the written file. For some file formats this setting is ignored. E.g. the export to mdf formats (MDF3) always uses “time” and the export to a database always uses “one_group”.

```
<?xml version="1.0" encoding="UTF-8"?>
<content groupCriteria="message_channel" />
```

Listing 104: Example

Since the MDF4 format in particular assumes this, that the signals can be divided into groups with equal sampling, there is the possibility in AMeDA to sample and group the signals in this way, that they meet this criterion. The GroupResamplingEvaluator can be used to define and name such a signal group (sampling group). Under the name that this evaluator gives to the group, then also entire signal groups can be referenced for export, by creating for each group an entry of the form <samplingGroup>GroupName</samplingGroup> is added inside the <content/> tag. (Described in more detail in the GroupResamplingEvaluator section). The “maxGroupSize” attribute can also be used to specify the maximum number of signals that can be included in a Signal group come.

```
<?xml version="1.0" encoding="UTF-8"?>
<content maxGroupSize="10" />
```

Listing 105: Example

Furthermore, signals can be stored “on change”. This means that only then data will be written to the file, if the affected signal has changed. In the case of an MDF file, however, this means that data is already accruing, when a signal of the affected signal group changes.

```
<?xml version="1.0" encoding="UTF-8"?>
<content onChange="true" />
```

Listing 106: Example

In signal-based export, there is the possibility to specify the signals to be exported using a regular expression filter. The signal name is checked for the passed expression and added if necessary. It is also possible to use the “exclude” field. If this is set, all signals that are not assigned to the regular expression.

```
<?xml version="1.0" encoding="UTF-8"?>
<content pattern="[abc]*" />
```

Listing 107: Example

In signal-based export there is an additional option to remove a prefix from the signals to be exported. The prefix is checked for each signal and removed if necessary. If the prefix string starts with “^”, it is interpreted as a regular expression. In this way, multiple prefixes can also be removed. In the following example, the “ant.” and “mot.” prefixes are removed.

```
<?xml version="1.0" encoding="UTF-8"?>
<content prefix="^(ant\.|mot\.)" />
```

Listing 108: Example

For signal-based export as MDF4, it is possible to specify that the contents of the frames are taken over directly when writing. The individual signals are then no longer interpreted in AMeDA, but only the descriptions of the signals as metadata. The result is still a signal-based file, but the export is if very many signals are exported, more performant.

```
<content type="mf4" writeFramesDirectly="true">
```

Listing 109: Example

10.8.5 FilePassThroughEvaluator

The FilePassThroughEvaluator is used to output one or more files from the measurement as a result file.

In this example, all txt files from the measurement are added to the result.

```

<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>FilePassThroughEvaluator</evaluator>
    <description short="Test" />
    <content>
      <data fileName="*.txt" zip="false" />
    </content>
  </template>
</templates>

```

Listing 110: File passThrough.tem

With `<addToEvent/>`, files can be created based on a date contained in the filename, be assigned to the events in the measurement. The date is specified by means of a `SimpleDateFormat` string, which is common in Java programs, extracted from the file name. For this, the file name must have a form to be parsed via such a format string. In addition, the date must be included in the measurement and the time stamps in the measurement must be given as absolute time stamps or converted with a `RealTimeEvaluator`. Each event is assigned the files that lie within the start and end times of the event. Since files have only a timestamp (no time extension), it can happen especially with very short events, that the files cannot be assigned to an event in this way. Therefore, the files can be given a duration (`preTolerance` and `postTolerance`). In the example the file gets artificially a temporal extension of one second before and two seconds after your timestamp.

```

<content>\n"
<data fileName="*.txt" zip="false" />\n"
<addToEvent>\n"
<parseFileName dateFormat="'Testfile_'yyyy-MM-dd_HH-mm-ss'.txt'"
preTolerance="1.0" postTolerance="2.0"/>\n"
</addToEvent>\n"
</content>\n"

```

Listing 111: Example

If the measurement contains ZIP archives, they will be searched recursively by default. Files that are inside an archive that is in the measurement archive are thus output as well. To suppress this behavior, the `scanZip` parameter can be set to `false`.

```

<content>\n"
<data fileName="*.txt" zip="false" />\n"
<addToEvent>\n"
<parseFileName dateFormat="'Testfile_'yyyy-MM-dd_HH-mm-ss'.txt'"
scanZip="false"/>\n"
</addToEvent>\n"
</content>\n"

```

Listing 112: Example

10.8.6 LogbookEvaluator

The `LogbookEvaluator` summarizes the values of individual signals within a cycle. By default are calculated: `<p> startDate` - Absolute start time `<p> endDate` - Absolute end time `<p> startTime` - Relative start time `<p> endTime` - Relative end time `<p> duration` - cycle time `<p>` In addition, for any number of signals Minimum, Maximum, Average, Start value, End value, Difference from start and end value, cycle duration and the sum of the rising and falling edges are output. For each signal, the individual output values can be activated or deactivated. By default they are disabled.

In the following example, some values are output for each of the three signals `s1`, `s2`

and s3. The activation and deactivation of the individual values works thereby over the <options/> tags, which must be listed in the same order as the signals in the <instance/> tag. The cycle is determined by an enable signal (enable attribute in the <content/> tag). The signal can be interpreted as a binary signal, which is true if it is greater than threshold. A cycle always lasts from one positive to the next negative edge of this signal. To be exact Signal evaluations (such as min/max and averaging) related to signals that were present at the time of the rising Edge arrive, over the entire cycle, up to the falling edge. In the process, signals that are simultaneously arrive with the falling edge are not observed. Additional timeout values can be specified for the calculation of the min/max values. These are to be understood in such a way that the current value is only taken into account in the min/max calculation, if it is not older than the timeout value. By default the timeout is set to 0.0, thus only values are considered which occur within the considered interval.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>LogbookEvaluator</evaluator>
  <description>TTable</description>
  <instance>
    <signal name="s1"/>
    <signal name="s2"/>
    <signal name="s3"/>
  </instance>
  <content enable="enableName" threshold="0.5">
    <options min="true" max="true" timeOutMin="0.0" timeOutMax="0.0" mean="false" />
    <options min="false" max="true" />
    <options min="true" max="false" mean="true" diffStartEnd="true"
      startValue="true" endValue="true" />
  </content>
</template>
```

Listing 113: File test.tem

The mean value is a time average, which is calculated via a numerical integral without interpolation. To prevent that e.g. CAN timeouts falsify the result, a timeout time can optionally be specified via the timeOutMean attribute. Signal values that are not renewed for longer than the timeout period, are then ignored when calculating the mean value.

```
<options mean=true timeOutMean=0.5/>
```

Listing 114: Specifying a timeout period

In the <content/> element it can be determined whether an incomplete cycle at the beginning startClosedOnly and/or at the end endClosedOnly of the measurement should be written. If the user does not select anything, the beginning and the end of the measurement will be recorded.

```
<content enable="\enableName\" threshold="\0.5\" startClosedOnly="\false\">
```

Listing 115: If the beginning is incomplete the cycle is not recorded

If the startClosedOnly option is set to false and before the first value of the Enable signal there is a too signal value is read, the first cycle does not start even if the value of the Enable signal is greater than the threshold value. The change from a not yet present measured value of the enable signal to a measured value greater than the threshold therefore does not lead to the start of a cycle.

By setting the ignoreTime attribute, the minimum size of the cycles can additionally be determined. The cycles shorter than the minimum size are not considered.

```
<content enable="enableName" threshold="0.5" ignoreTime="1">
```

Listing 116: Specify minimum size of cycles

By setting the following attributes, the respective associated column name can be changed. min: minName. max: maxName. mean: meanName. diffStartEnd: diffStartEndName. start: startName. end: endName. cycleLength: cycleLengthName

```
<?xml version="1.0" encoding="UTF-8"?>
<options min="true" minName="ttl_min" />
```

Listing 117: Change the names of the columns

Setting the columnName attribute sets the prefix of the column names. If it is not set explicitly, the column name corresponds to the signal name, extended by a suffix indicating which calculation is involved (e.g. min). If a column name is explicitly entered for one of the options (e.g. by setting minName), the columnName prefix is ignored and the complete column name is substituted.

```
<?xml version="1.0" encoding="UTF-8"?>
<signal name="s1" columnName="newSignalname" />
```

Listing 118: Change prefix of column names

The number of edges of a signal is counted as soon as the signal exceeds the set threshold value thresholdForCounter exceeds. The increasingEdgeNumber attribute is used to determine the Sum of rising edges in the cycle activated. The decreasingEdgeNumber attribute activates the Determination of the sum of falling edges in the cycle.

```
<options thresholdForCounter="25.0" increasingEdgeNumber="true"
decreasingEdgeNumber="true"/>
```

Listing 119: Number of times a threshold is exceeded per cycle

By setting the retentionPeriod attribute, the sum of the Time of exceeding a threshold thresholdForCounter calculated.

```
<options thresholdForCounter="0.5" increasingEdgeNumber="true"
decreasingEdgeNumber="true" retentionPeriod="true"/>
```

Listing 120: Sum of the time of exceeding a threshold per cycle

10.8.7 MLClassificationExportEvaluator

The MLClassificationExportEvaluator processes histogram data from the G.i.N. Data loggers and writes them to the metadatabase. Prerequisite for the evaluation is the use of G.i.N. data as source, which, however, do not have to be evaluated. The raw data have file extension .RCL and can be displayed as tables.

The template has no mandatory parameters, but the following attributes must be defined in the <content/> tag. A file name pattern is defined as a regular expression with the filePattern attribute. An empty expression means all histogram data. It should be noted that the selection is limited only to the histogram data and thus each expression is extended by an addition '\.RCL'. The tableNamePrefix attribute can be used to customize all names of tables written with the evaluator. The attribute abortOnError defines with a boolean value the behavior when an error occurs. With 'false' all erroneous files are ignored and the evaluator processes all error-free files. With 'true' the evaluation is aborted at the first error. Optionally, the handleNames attribute can be used to define how to shorten names that are much

too long for the metadatabase. With 'trimStart' the characters are removed at the beginning and with 'trimEnd' at the end.

```
<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>MLClassificationExportEvaluator</evaluator>
    <description short="test" />
    <content filePattern="" tableNamePrefix="" abortOnError="false" />
  </template>
</templates>
```

Listing 121: File test.tem

10.8.8 ReducedSignalExportEvaluator

The ReducedSignalExportEvaluator is used to reduce an input signal to a specific Reduce sampling rate. Within the sampling rate, only the local minimum and Maximum values are stored, so that the number of measuring points is limited to the most important points limited. In addition, it is also possible to measure the input signal via a time series instead of via a Reduce control signal.

An input signal and an interval are required for signal reduction. The interval is the sampling rate in which the input signal is sampled for minima and maxima is reviewed.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>ReducedSignalExportEvaluator</evaluator>
  <description>test</description>
  <instance>
    <signal name="s1"/>
  </instance>
  <content period="1.0"/>
</template>
```

Listing 122: File test.tem

In order to reduce the input signal via a control signal, only the control signal can be specified as a parameter. The reduction takes place then at the specified interval via the control signal.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>ReducedSignalExportEvaluator</evaluator>
  <description>test</description>
  <instance>
    <signal name="s1"/>
  </instance>
  <content period="5.0" controlSignal="s3"/>
</template>
```

Listing 123: File test.tem

10.8.9 ResultFileListExportEvaluator

The ResultFileListExportEvaluator is used to output a table with all the files generated and corresponding meta data.

The only parameter of the evaluator is the table name, which is set from "result_test" in the following example.

```

<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>ResultFileListExportEvaluator</evaluator>
    <instance>
    </instance>
    <content tableName="result_test" />
  </template>
</templates>

```

Listing 124: File test.tem

10.8.10 SignalMetaDataExportEvaluator

The SignalMetaDataExportEvaluator writes metadata about each selected signal into a table. Via `<instance/>` the considered signals can be filtered. This can be done by means of a concrete signal list or with a regular expression. It is also possible to invert the signal filter, thus to process all signals except the selected ones. The min and max value is output for each selected signal. In addition, both values are available in filtered form, in which values with a text representation are ignored (these often correspond to invalid values). In addition, a counter is output for each, indicating the number of filtered and non-filtered values.

In this example, the pattern is used to specify that all signals are exported, which contain "Geschw". With `duplicationHandling` you can determine what should happen to signals that exist multiple times, therefore have identical names. In this case, the signal values of all signals are treated like this, as if they belonged to a signal. With "first" can be determined, that only the first signal is considered. With "first_value" the first signal is considered, that provides a value. In addition, "all" (default) can be used to determine, that all signals are to be output individually. Then the result table may contain several rows with the same signal name.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>SignalMetaDataExportEvaluator</evaluator>
  <description>Test</description>
  <instance exclude="false" />
  <content tableName="merge" duplicationHandling="merge" pattern=".*Geschw.*" />
</template>

```

Listing 125: File example.tem

10.8.11 SignalMonitorEvaluator

The SignalMonitorEvaluator is used to transfer complete signal waveforms into the MetaDatabase. This is used, for example, when the loading of a soot particle filter is displayed graphically over several weeks should be. In this case, measured value curves from several measurements can then be displayed together.

In the following example, the signal `s1` specified in the `name` attribute in the `<signal/>` tag is written to the MetaDatabase taken over. In the `<monitor/>` tag, the attribute `mode="mean"` a Averaging activated. The tag `<trigger/>` causes a periodic transmission of the mean values. Here, the attribute `mode` specifies that periodic transmission is to take place, with the frequency specified in the `period` attribute, here every 1000 seconds. In addition to the the time average, the respective maximum and minimum in each interval are also taken over. Thus, also an upper and lower envelope curve are plotted. In the associated table row, all values that occurred before the trigger time are taken into account. A value that occurs at the same time counts towards the next interval. When the time average is calculated, there is a peculiarity in the initialization of the min and max values. An additional timeout value can be specified here for the calculation of the min/max values. This is to be

understood in such a way that the current value is only taken into account in the min/max calculation, if it is not older than the timeout value. By default the timeout is set to 0.0, thus only values are considered which occur within the considered interval. When calculating the arithmetic mean, only values that fall within the current interval generally count. The behavior is applied to the min/max calculation as well.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>SignalMonitorEvaluator</evaluator>
  <description>Test</description>
  <instance><signal name="s1"/></instance>
  <content>
    <monitor mode="mean"/>
    <trigger mode="period" period="1000"/>
  </content>
</template>
```

Listing 126: File test.tem

In the `<trigger/>` tag, the `mode="signal"` attribute can also be used to trigger a Saving process can be triggered. In this case, an entry is always created in the metadata database, if the corresponding signal exceeds the threshold 0.5. In the `name` attribute the name of the trigger signal is specified.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>SignalMonitorEvaluator</evaluator>
  <description>Test</description>
  <instance> <signal name="s1"/></instance>
  <content>
    <monitor mode="mean"/>
    <trigger mode="signal" name="f1"/>
  </content>
</template>
```

Listing 127: File test.tem

If the tag `<trigger/>` is omitted, only one average value per measurement is stored.

With the tag `<signalValueToGet/>` values of further signals can be taken over into the database. However, only the current value at the time of saving is used. This is suitable especially for slowly changing signals, such as a temperature. It can also specify the mileage then a plot over the mileage is also possible in reporting, for example.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>SignalMonitorEvaluator</evaluator>
  <description>Test</description>
  <instance><signal name="s1"/></instance>
  <content>
    <monitor mode="mean"/>
    <trigger mode="period" period="1000"/>
  </content>
  <signalValueToGet><signal tag="s1Name" name="s1"/></signalValueToGet>
</template>
```

Listing 128: File test.tem

The `<enable/>` tag can be used to specify a signal in the `name` attribute, by which the storage mechanism can be switched on and off.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>SignalMonitorEvaluator</evaluator>
  <description>Test</description>
  <instance> <signal name="s1"/></instance>
  <content>
    <enable name="enable"/>
    <monitor mode="mean"/>
  </content>
</template>
```

Listing 129: File test.tem

If in the `<monitor/>` tag the attribute `mode="copy"` is set, each measured value will be copied to the metadatabase taken over. ATTENTION: This should be used only with great caution. It can easily be several 100 million Entries are created in the database, which poses problems for relational databases!

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>SignalMonitorEvaluator</evaluator>
  <description>Test</description>
  <instance> <signal name="s1"/></instance>
  <content>
    <enable name="enable" />
    <monitor mode="copy" />
  </content>
</template>
```

Listing 130: File test.tem

Even if triggered, a check is made at the end of the measurement to see if further data has been received. If data was still read after the last trigger event, these are taken into account by another table line. Such a line that is not triggered by any trigger event, but automatically, written at the end of the measurement can be seen from this, that the flag in the "final" column is set.

If the `<monitor/>` tag is set to `mode="arithmeticMean"` is set, the arithmetic mean is calculated. In contrast to the calculation of the time average, the time is not weighted. Each measured value receives the same weighting regardless of how long an individual measured value is valid. When applying the arithmetic Average, the timeout is omitted, otherwise the behavior corresponds to the formation of the time average. In the following example, in the `<monitor/>` tag, the attribute `mode="arithmeticMean"` is set, and thus the arithmetic averaging activated.

10.8.12 SignalToTableEvaluator

The `SignalToTableEvaluator` copies multiple signals into a common table. In the `<content/>` element, the `tableName` attribute can contain the name of the table be indicated. If this is missing, the name of the template is used as the table name.

In this example, the three signals `s1`, `s2` and `s3` are written to the `SignalToTable` table of the database. Instead of explicitly specifying all signals, all signals included in the measurement can also be written, by leaving the signal list empty. In addition, the `exclude` attribute can be used to specify, that all signals, except those included in the list, are to be output. By default `exclude` is set to false.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>SignalToTableEvaluator</evaluator>
  <description>test</description>
  <instance exclude="false">
    <signal name="s1"/>
    <signal name="s2"/>
    <signal name="s3"/>
  </instance>
  <content tableName="SignalToTable"/>
</template>

```

Listing 131: File test.tem

If all specified signals do not change between two points in time, no consecutive identical rows are written to the table. This behavior is in Figure 1 for an example with two signals clarified. At the times 1 and 2, the current signal values are written, since the signal values have changed. At time 3 no signal value changes, so no row is written to the table. At time 4, a row is written to the table again due to value changes. At the time 5 the light blue signal does not change its value, while no value is present for the dark blue signal. It will therefore no signal value change is detected and thus no value is written to the table. At the time points 6 and 7 lines again in each case, since at least one value has changed in each case.

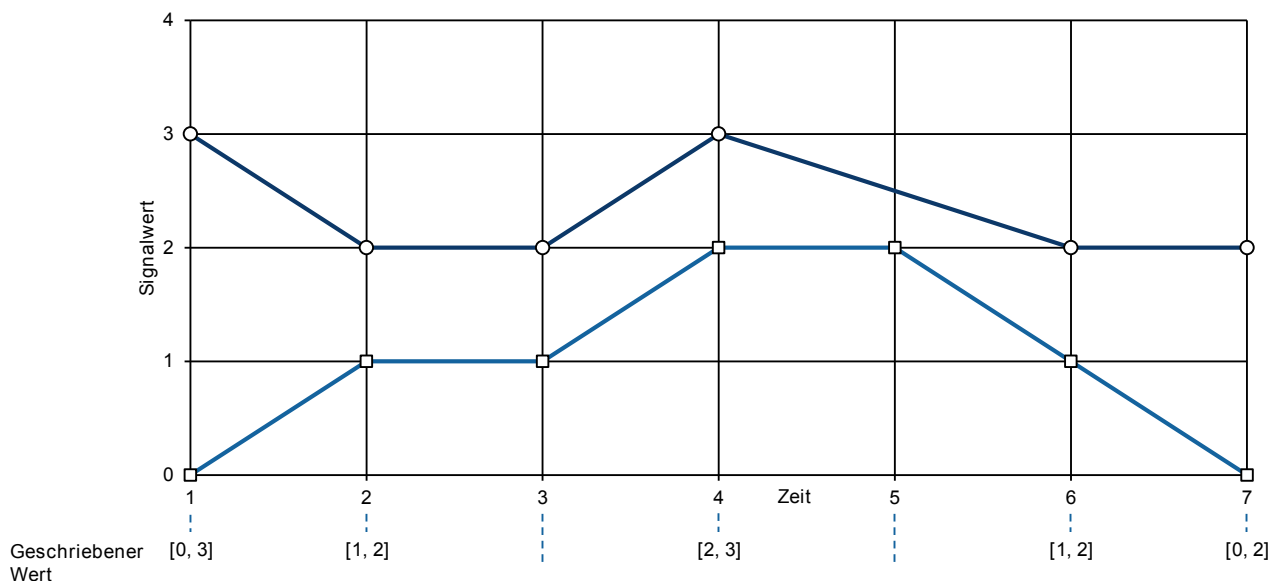


Figure 1: Example of writing for signal value changes

The `enable` attribute can be used to specify an enable signal that activates writing to the table and can be deactivated. The example in Figure 2 illustrates the use of the Enable signal. At the time points 1 and 2 signal value changes are present, while the Enable signal has a value >0 , so that the corresponding signal values are written. At time 3 is not written to the table due to lack of value change. At times 4 and 5 the enable signal is inactive ($=0$), so that no values are written. At time 6 the enable signal is >0 again. However, since no change in value with respect to the last value considered (time 2), no value is written to the table. Only at time 7 the current value is written again.

```

<content tableName="SignalToTable" enable="\EnableSignal\"/>

```

Listing 132: Specification of an enable signal

By setting the `lastValueOnChange` attribute to `true` the writing of the last signal values can be activated additionally, that were attached before the change. The example with three signals in Figure 3 illustrates this behavior. At the times 1 and 2 signal value changes are present in each case, so that values in be written to the

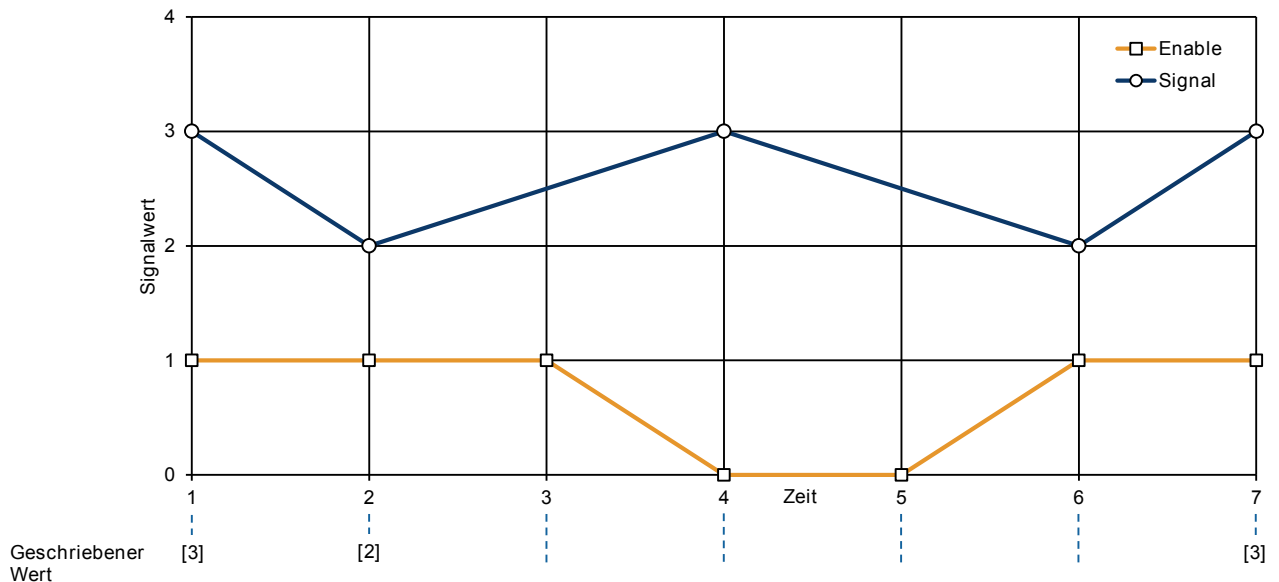


Figure 2: Example of using the Enable signal

table. At the times 3 and 4 the signal values do not change and no Values written to the table. At time 5 the signal values change. Therefore, first the last Values without change (from time 4) written and then the new changed values from time 5. Since to the Time 5 no value of the dark blue signal is present, the last known value is written. The rest of the behavior follows the previous pattern.

```
<?xml version="1.0" encoding="UTF-8"?>
<content tableName="SignalToTable" lastValueOnChange="true" />
```

Listing 133: Activate writing of the last signal values

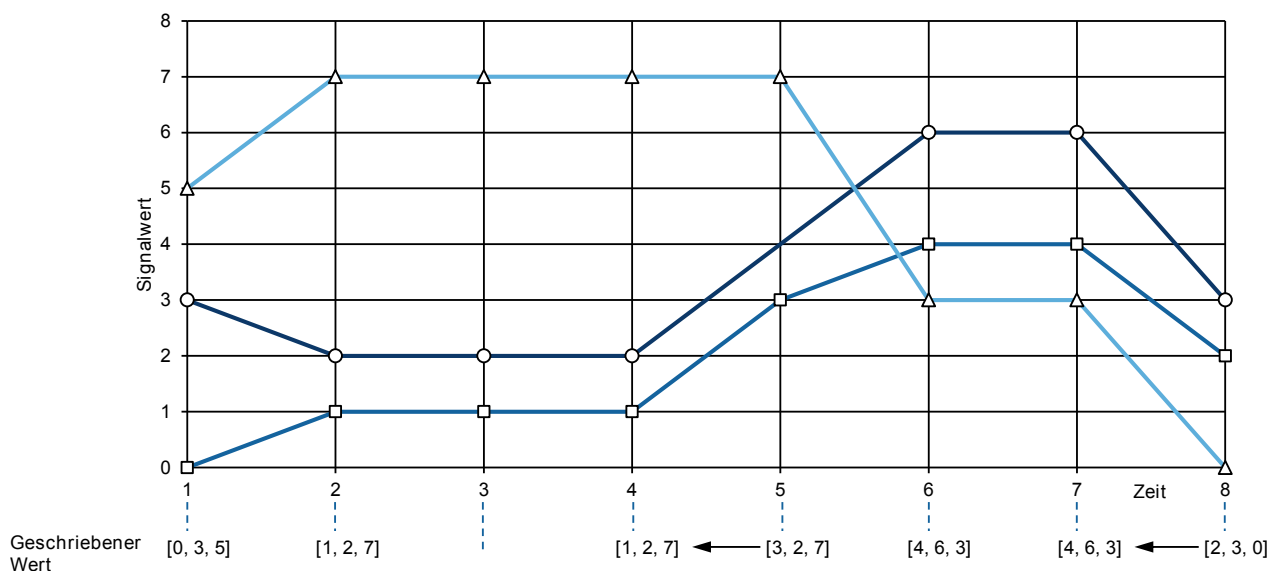


Figure 3: Example of using LastValueOnChange

By setting the attribute `ignore` of the element `<valueChange/>` can be specified whether a change of the signal value leads to an entry in the table or not. The `signal` attribute is used to select the affected signal. The example with two signals in Figure 4 illustrates this behavior. For the dark blue signal, let the `ignore` attribute be set. At the time points 1, 2 and 3 the value of the non-ignored light blue signal changes in each case, so that the corresponding values of all signals are written into the table. At time 4 only the ignored dark blue changes Signal and at time 5 there is no updated value of the light blue signal, so that in each case there

are no values should be written into the table. At time 6 an updated value of the light blue signal is present, so that the current value of the light blue signal and the last known value of the ignored signal are written. <p> The following example shows a configuration for three signals. The <valueChange/> tags for the "s1" and "s3" signals could also be omitted, since ignore="false" corresponds to the default behavior.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>SignalToTableEvaluator</evaluator>
  <description>test</description>
  <instance>
    <signal name="s1"/>
    <signal name="s2"/>
    <signal name="s3"/>
  </instance>
  <content tableName="SignalToTable" lastValueOnChange="true">
    <valueChanges>
      <valueChange signal="s1" ignore="false"/>
      <valueChange signal="s2" ignore="true"/>
      <valueChange signal="s3" ignore="false"/>
    </valueChanges>
  </content>
</template>
```

Listing 134: File test.tem

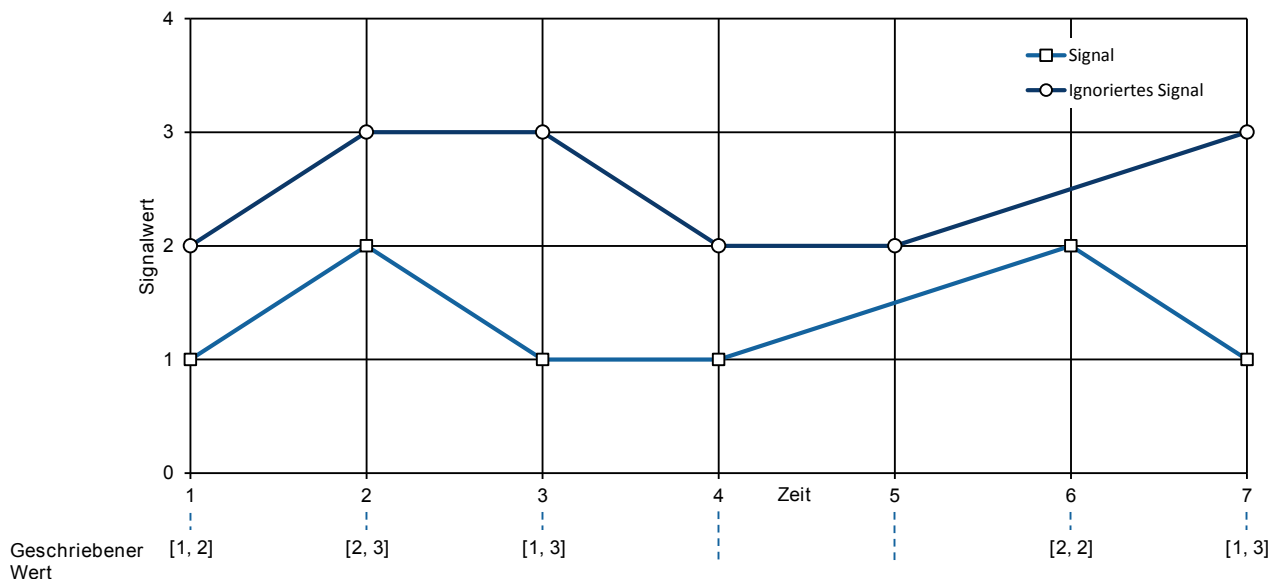


Figure 4: Example of using Ignore

With the attribute `writeLastRecord` it can be activated that at the end of the measurement still a Table row where the last values are written. When an enable signal is set, a corresponding table row is also written at the end of an enable phase, and at the end of the measurement only in the case that this is within an enable phase. <p> The example in Figure 5 illustrates this behavior. At times 1 and 2 there is a signal value change and the enable signal is so that the corresponding signal values are written into the table. At time 3 no value written, since there is no signal value change. At time 4 there is a falling edge of the enable signal, so that the last value (from time 3) is written. While the Enable signal is inactive, no values will be written. At time 6 the enable signal becomes active again. Since the signal concerning the last valid value has no change, no value is written. At the end of the measurement, because of the parameter `writeLastRecord` the last value is written.

```
<content tableName="SignalToTable" writeLastRecord="true">
```

Listing 135: Writing the last record

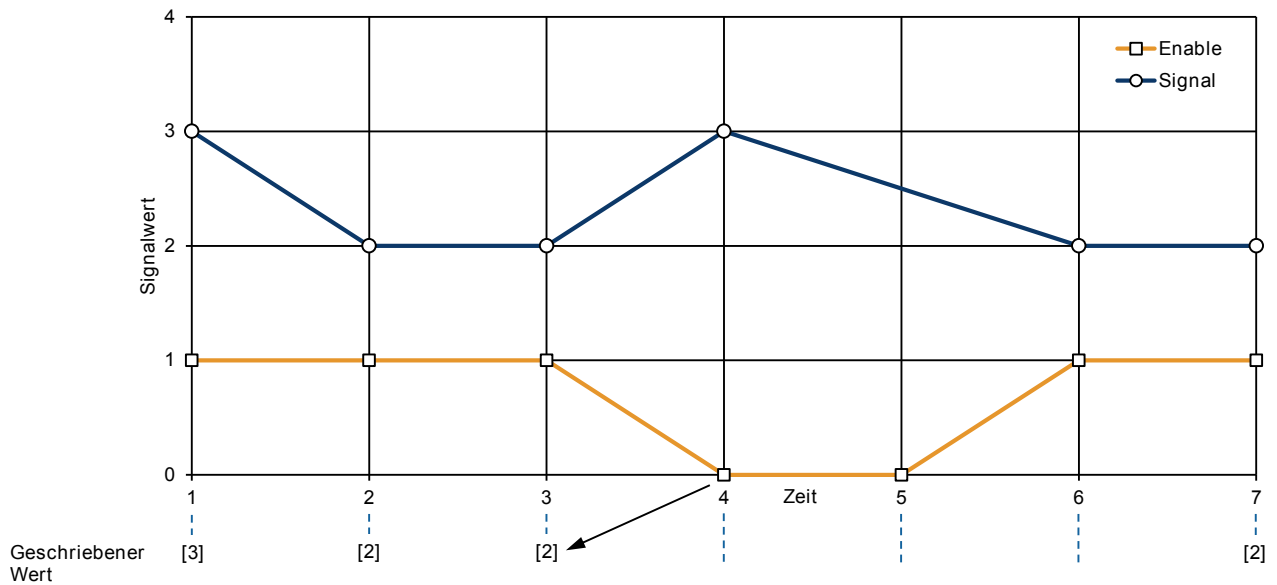


Figure 5: Example of using WriteLastRecord

With the `<trigger/>` tag inside the `<content/>` block you can instead of writing when signal values change, a triggered write can be activated. It can be triggered either by a signal, when this exceeds a certain threshold or periodically. In the following examples, once by the signal `t` with a threshold of 0.5 and triggered once periodically with the period 2s. `<p/>` The example in Figure 6 illustrates this behavior for a single signal in conjunction with a trigger and an enable signal. Value changes of the input signal are completely ignored. Data is only written to the table when the Enable signal is set and the trigger signal has reached the exceeds configured threshold (at times 2 and 9). `<p/>` The `<trigger/>` tag cannot be combined with the following attributes: `writeLastRecord`, `writeOnEachValue` and `ignore`.

```
<?xml version="1.0" encoding="UTF-8"?>
<trigger mode="signal" name="t" threshold="0.5" />
```

Listing 136: Signal controlled trigger

```
<?xml version="1.0" encoding="UTF-8"?>
<trigger mode="period" freq="2" />
```

Listing 137: Periodic trigger

The `writeOnEachValue` attribute can be used to force the writing of all signal values. It is can be combined with the attribute `enable` so that only all values are written while the enable signal is set is. A combination with the tag `<trigger/>` is not possible. `<p/>` The example in Figure 7 illustrates this behavior for a single signal in conjunction with an enable signal. Data is written to the table with each new signal value (i.e. even without a signal value change) as long as the Enable signal is set. `<p/>` The `writeOnEachValue` tag cannot be combined with the following attributes/tags: `writeLastRecord`, `IsatValueOnChange`, `ignore` and `<trigger/>`.

```
<content tableName="table" writeOnEachValue=\"true\" enable=\"EnableSignal\"/>
```

Listing 138: Write all data via writeOnEachValue

Since the data is to be written to a database table, no column names may appear twice. That is why the Evaluator provides settings to handle signals with the same name. These settings can be found under the tag `<signalCombines/>`. Each

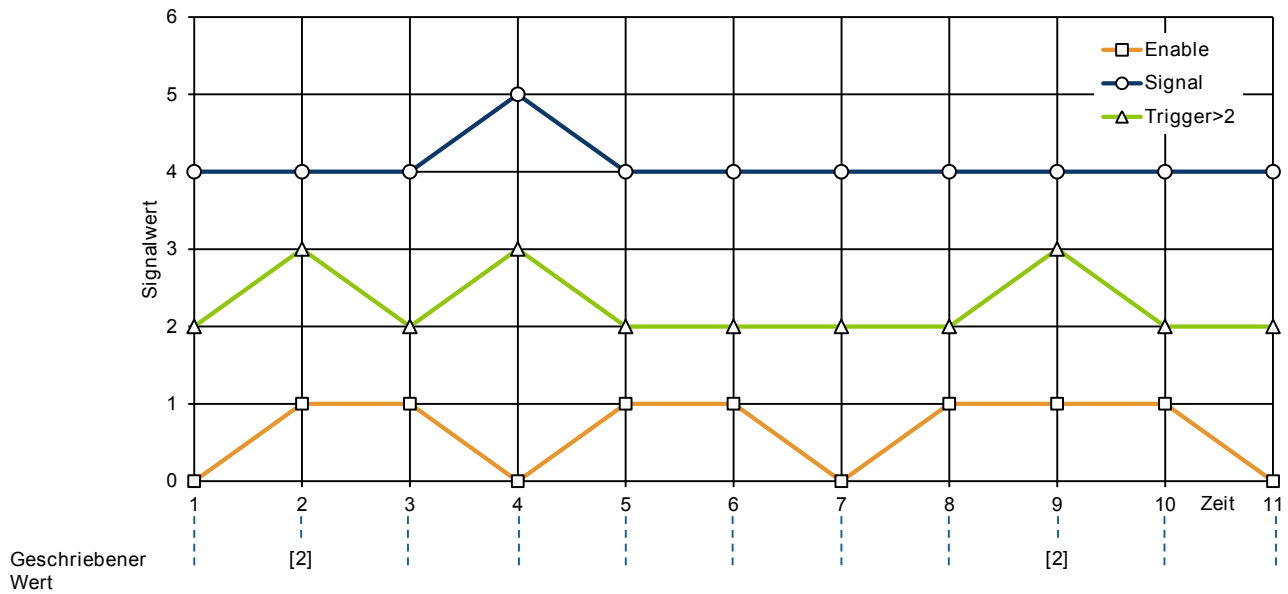


Figure 6: Example of using a trigger and enable signal

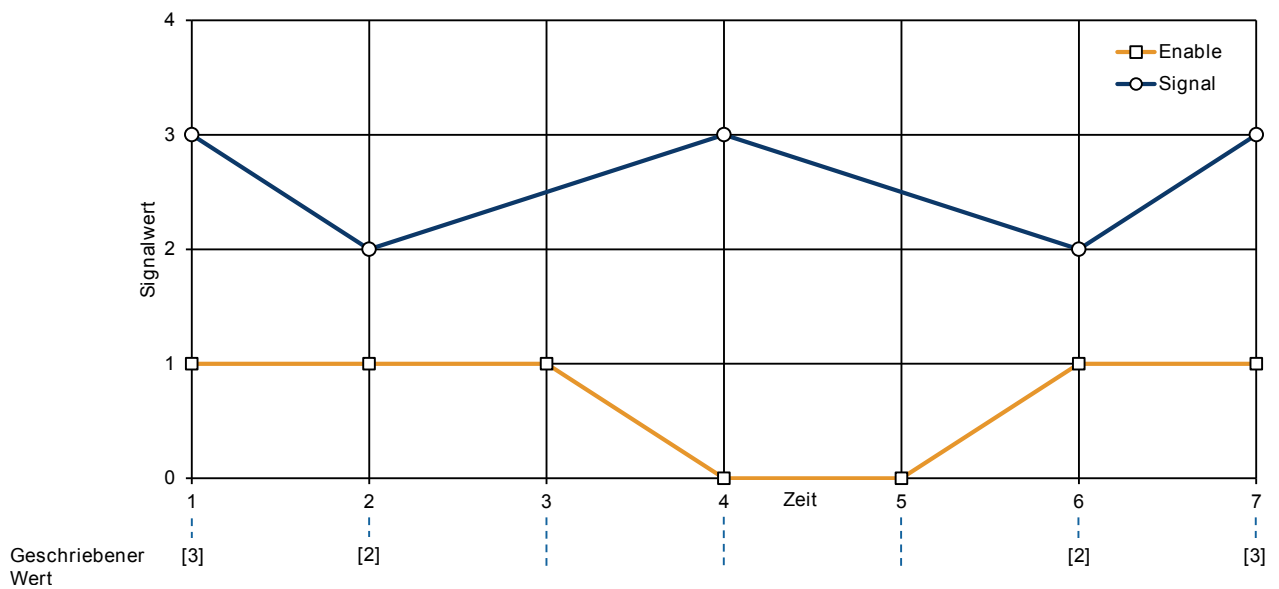


Figure 7: Example of using WriteOnEachValue and Enable signal

`<signalCombine/>` tag represents a signal, whose signal name is specified with the `signal` attribute. All signals with the specified name are then combined into one signal. If as `combineType` "merge" is specified, all signal values from all signals of the specified name are output. With "first" only the signal that outputs a value first is used. "none" means that the signals are not combined. In the case of a signal name assigned more than once, the evaluation terminates with an error. If no own `<signalCombine/>` tag exists for a signal, the `combineType` from the `<content/>` tag is used for this signal. The default value is "none".

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>SignalToTableEvaluator</evaluator>
  <description>test</description>
  <instance exclude="true">
    <signal name="enable"/>
  </instance>
  <content enable="enable">
    <valueChanges>
    </valueChanges>
    <signalCombines combineType="none">
      <signalCombine signal="s1" combineType="merge" />
    </signalCombines>
  </content>
</template>

```

Listing 139: File test.tem

10.9 Extras

10.9.1 FrameMultiplexEvaluator

The FrameMultiplexEvaluator allows to combine several frame signals. The prerequisite is that all signals involved have compatible types. For example CAN frames can be combined with CAN error frames, but not with Ethernet frames.

In this example the FrameMultiplexEvaluator is used. The type parameter can be used to specify the data type. Any frame data type can be used here. The signals specified in <instance/> must all be compatible to this type. All specified signals are combined in the signal "mux".

```

<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>FrameMultiplexEvaluator</evaluator>
    <description short="neuer Name" group="" level="0" />
    <content type="CANData" signalName="mux" />
    <instance>
      <signal name="canMessage"/>
      <signal name="canErrorFrame"/>
    </instance>
  </template>
</templates>

```

Listing 140: File test.tem

10.9.2 HandleSignalDuplicationEvaluator

The HandleSignalDuplicationEvaluator can be used to evaluate signals, that exist multiple times (same name). In such a case, evaluators normally cannot access the signals, as they cannot be clearly assigned. The HandleSignalDuplicationEvaluator generates a new signal from all signals of a name. This can now be clearly assigned.

This example shows how the evaluator works. One or more signals can be specified in the <content/> tag. All signals that have the name name are grouped under the signal with the name out. The optional combineType parameter specifies how the signals are combined. With merge the signal values from all signals are output. With first only the signal values of the first signal that provides a value are output.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>HandleSignalDuplicationEvaluator</evaluator>
  <description>test</description>
  <content>
    <signal name="s1" out="s1_comb" combineType="merge"/>
  </content>
</template>

```

Listing 141: File combineSignals.tem

10.9.3 MeasurementInfoEvaluator

The MeasurementInfoEvaluator can be used during the evaluation to extract the following information from the measurement extracted and written to the metadata database: Used database files, type and serial number of the file used for the recording of the measuring device used and the associated measuring device configuration.

The used database files are created together with the channels, for which they were used are output in a table. The name of the table can be set using the <databaseTable/> tag. If the name is not set, the table is called "DatabaseFiles". The other information is output in another table. The name of the table can be set via the tag <measDeviceTable/>. If the name is not set, the table is called "MeasurementDevice".

The entries in the tables receive the time stamp of the first read measured value. This includes generated measured values, such as constants.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>MeasurementInfoEvaluator</evaluator>
  <description>Test</description>
  <content measDeviceTable="mdTable" databaseTable="dbTable"/>
</template>

```

Listing 142: File MeasInfo.tem

10.9.4 SignalValidityCheckEvaluator

This is an evaluator that checks if signals are present and valid in the measurement. The signals of a measurement are checked against the min and max values stored in a DBC file. The name attribute in the <dbcFile/> tag is used to configure the DBC file to be used. The name attribute in the <Table/> tag can be used to specify the name for the metadatabase table. The name attribute in the <signalNameSuffix/> tag can be used to add a suffix to all signal names that are taken from the DBC file are attached. For example, if a meter writes a suffix (for example, the Channel names) to the signal names, there is no need to create a DBC file with customized signal names. In the Metadatabase table, the complete signal name, i.e. signal name with suffix, is output. There are four different statuses returned for the signals. - VALID: The signal has remained in the value range. - INVALID: The signal has not remained in the value range. - NO_DATA_FOUND: The signal is in the measurement, but does not provide any values. - NOT_EXISTING: The signal is specified in the DBC, but is not found in the measurement.

In the following example, the <dbcFile/> tag in the name attribute is used to specify that the DBC file is with the name "7411_05062018.dbc" should be used for validation. The <table/> tag is used in the name attribute to specify that "SVC" is the table name in the metadata database is to be used. Using the <signalNameSuffix/> tag, the name attribute specifies that "/ETCK:1" should be used as suffix for the signal names. In the table, in this case, the signal name will be replaced with the Suffix

output.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>SignalValidityCheckEvaluator</evaluator>
  <description short="name" />
  <content>
    <dbcFile name="7411_05062018.dbc"/>
    <table name="SVC"/>
    <signalNameSuffix name="/ETCK:1"/>
  </content>
</template>
```

Listing 143: File test.tem

In the following example, a platform is used to define a prefix. In this case, the signal name will be displayed with the Prefix output in the table.

```
<?xml version="1.0" encoding="UTF-8"?>
<plattform name="PQ3x_4X_ML_201105A.xml">
  <CAN-DBC name="Test.dbc" channel="1" prefix="ant" addId="false" addMessageName="false" />
</plattform>
```

Listing 144: File Test.xml

10.9.5 TwoPassEvaluator

The `TwoPassEvaluator` can be used to evaluate a measurement in several steps. Signals can be specified which will be generated in the first pass. These signals are then available in the second run available. This can be used if, for example, a signal is not detected until the end of the evaluation can be calculated completely, but is required for other templates right from the start. In this case, a second pass can be performed, in which the signal from the beginning of an is available.

A simple example is shown below. Two TKO files are required for this. The file `test.tko` is the TKO file which will be executed first. This uses the template `twopass.tem` which stores the signal `store`. This signal is usually a virtual Signal, which can only be generated at the end of the first pass. The file `two.tko` then describes the second run. In the second pass the signal `store` is available from the beginning.

```
use("test") use("twopass")
```

Listing 145: File test.tko

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>TwoPassEvaluator</evaluator>
  <description>Test</description>
  <virtualSignals>
    <signal out="store" unit="1" expression="s1+s2" />
  </virtualSignals>
  <instance> <signal name="store"/> </instance>
  <content>
    <tko name="two.tko"/>
  </content>
</template>
```

Listing 146: File twopass.tem

```
use("second") use("order") .visualize="no"
```

Listing 147: File two.tko

It is also possible to specify the content of the second TKO file directly in the evaluator.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>TwoPassEvaluator</evaluator>
  <description>Test</description>
  <virtualSignals>
    <signal out="store" unit="1" expression="s1+s2" />
  </virtualSignals>
  <instance> <signal name="store"/> </instance>
  <content>
    <tko>
      use("second")
    </tko>
  </content>
</template>
```

Listing 148: File twopass.tem

It is possible to omit the specification of signals. Then the only way to get information from the first to pass to the second pass the setting of values in the context.

10.10 FuzzyEvaluator

The fuzzy evaluator provides a probability of occurrence of an event as a result. As with the FunctionEvaluator, this is also an arbitrary function of signals. However, the result is calculated with a certain fuzziness (fuzzy), where the result value is between zero (does not apply at all) and one (applies 100%). The result or this The probability of occurrence is thus at the same time a quality measure for the occurrence of the event. The structure of the template is similar to the FunctionEvaluator. The fuzzy function is also used in the `<function/>` defined. The function is described in the FuzzyEvaluator in a verbalized language, where as before the used signals have to be defined in the `<parameter/>` before. A multitude of possible combinations of signals, calculations and (intermediate) results are conceivable. Similar to the FunctionEvaluator, a Java class is created from the fuzzy function, compiled at runtime and instantiated.

Comparison operations are performed according to the rules of fuzzy logic. In this way, the specification of hard limits can be avoided. The fuzziness of the comparison is specified in square brackets after the relational operator must be specified. If it is omitted, the blur is zero. Instead of `<` and `>`, `lower` and `greater` can also be used to keep the text more readable when `<` and `>` can only be used in XML escaping.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>FuzzyEvaluator</evaluator>
  <description>Test</description>
  <instance>
    <signal name="s1"/>
    <signal name="s2"/>
    <result name="res"/>
  </instance>
  <content>
    <parameter>
      <par name="a"/>
      <par name="b"/>
    </parameter>
    <function>
      result=(a lower[0.1] 5) and (b greater[1] 6)
    </function>
  </content>
</template>

```

Listing 149: File test.tem

The functions `abs`, `sqrt` and `ln` can be used. In addition, there is a special `ite([bed], [Then], [Else])` function.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>FuzzyEvaluator</evaluator>
  <description>Test</description>
  <instance> <signal name="s1"/> <result name="res"/></instance>
  <content>
    <parameter> <par name="a"/> </parameter>
    <function>
      result=ite(a>0,sqrt(a),-sqrt(-a))
    </function>
  </content>
</template>

```

Listing 150: File test.tem

In this example the result is the low pass filtered input signal. Accordingly, `highpass` can also be used.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>FuzzyEvaluator</evaluator>
  <description>Test</description>
  <instance> <signal name="s1"/> <result name="res"/></instance>
  <content>
    <parameter> <par name="a"/> </parameter>
    <function>
      result=lowpass(a,0.1)
    </function>
  </content>
</template>

```

Listing 151: File test.tem

In the following example a differentiation is performed!

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>FuzzyEvaluator</evaluator>
  <description>Test</description>
  <instance> <signal name="s1"/> <result name="res"/></instance>
  <content>
    <parameter> <par name="a"/> </parameter>
    <function>
      result=diff(a)
    </function>
  </content>
</template>

```

Listing 152: File test.tem

Values in the past can be accessed. Thus, `at -[t]` can be used to access a value that is `t` seconds in the past. In the example, the sum of the value of `a` from one and two seconds ago is formed. The future cannot be accessed, so the times must be negative.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>FuzzyEvaluator</evaluator>
  <description>Test</description>
  <instance> <signal name="s1"/> <result name="res"/></instance>
  <content>
    <parameter> <par name="a"/> </parameter>
    <function>
      result=a at -1 + a at -2
    </function>
  </content>
</template>

```

Listing 153: File test.tem

Intermediate values can be stored as values to simplify the expression.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>FuzzyEvaluator</evaluator>
  <description>Test</description>
  <instance> <signal name="s1"/> <result name="res"/></instance>
  <content>
    <parameter> <par name="a"/> </parameter>
    <function>
      l=lowpass(a,0.1)
      result=l>0.5
    </function>
  </content>
</template>

```

Listing 154: File test.tem

Access to the past can also be done on intervals. The interval is specified in round brackets, and can also only be specified in the Past. With `mean of a in (-3,0)`, for example, the mean value of the signal `a` of the last 3 seconds calculated. In addition to the mean value the maximum (`max of`), the minimum (`min of`), the width (difference between maximum and minimum) (`width of`) or the mean gradient (`mean grad of`) can be calculated.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>FuzzyEvaluator</evaluator>
  <description>Test</description>
  <instance> <signal name="s1"/> <result name="res"/></instance>
  <content>
    <parameter> <par name="a"/> </parameter>
    <function>
      a=mean of a in (-3,0)
      b=mean grad of a in (-3,0)
      c=max of a in (-3,0)
      d=min of a in (-3,0)
      e=width of a in (-3,0)
      result=a+b+c+d+e
    </function>
  </content>
</template>

```

Listing 155: File test.tem

If access to the past is required for the intermediate values generated, the definition can be preceded by a `signal`.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>FuzzyEvaluator</evaluator>
  <description>Test</description>
  <instance> <signal name="s1"/> <result name="res"/></instance>
  <content>
    <parameter> <par name="a"/> </parameter>
    <function>
      signal l=lowpass(a,0.1)
      result=l at -1 > 0.5
    </function>
  </content>
</template>

```

Listing 156: File test.tem

10.11 GPS

10.11.1 ReverseGeocodingEvaluator

The `ReverseGeocodingEvaluator` generates from a signal containing a geographical position in the form [longitude, latitude] represents, up to four output signals, which indicate the country, the region, the road name and the type of road included in this item. The properties of the 4 output signals are stored in the tags `<country/>`, `<region/>`, `<road/>` and `<typeOfRoad/>` are set. The attribute `enabled` can be used to specify whether one of the Signals should be output or not. The `signalName` attribute can be used to specify the names of the output signals change. The `radius` attribute of the `<road/>` and `<typeOfRoad/>` elements gives the algorithm for the Determination of the next street at the current position before, within what radius (in meters) after a street should be sought. If there is no street within this radius, a corresponding empty value is output.

In the example, the `ReverseGeocodingEvaluator` is used to evaluate the output signals `country`, `region`, `road` and `typeOfRoad` an input signal consisting of the values for latitude `lat` and length `lon` are assigned. It also searches for the nearest street within a radius of 100m.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>ReverseGeocodingEvaluator</evaluator>
  <description>Test</description>
  <instance>
    <signal name="lon"/>
    <signal name="lat"/>
  </instance>
  <content>
    <reverseGeocoder>
      <country enabled="true" signalName="country"/>
      <region enabled="true" signalName="region"/>
      <road enabled="true" signalName="road" radius="100"/>
      <typeOfRoad enabled="true" signalName="typeOfRoad" radius="100"/>
    </reverseGeocoder>
  </content>
</template>

```

Listing 157: File ReverseGeocoding.tem

10.11.2 RouteToDBEvaluator

The RouteToDBEvaluator is used to write GPS coordinates to the MetaDatabase. Two signals (latitude and longitude - in order) must be passed to the evaluator. These signals are then written to a table in the MetaDatabase. In addition, the table gets a column with a “trackCounter” that is always counted up, if there is either a minimum time or a minimum distance between two consecutive coordinates. Minimum time (in seconds) and minimum distance (in kilometers) can be set via the parameters <timeConstraint/> and <spatialConstraint/> can be set. If more signals are specified in the <instance/> tag, they appear in their own columns in the table.

In the example, the signals “lat” and “lon” are used as coordinates. The data is written to the “OSM” table and at 3s or 1km a new route segment is started.

```

<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>RouteToDBEvaluator</evaluator>
    <description short="OSM" />
    <instance>
      <signal name="lat"/>
      <signal name="lon"/>
    </instance>
    <content>
      <table name="OSM"/>
      <timeConstraint max="3"/>
      <spatialConstraint max="1"/>
    </content>
  </template>
</templates>

```

Listing 158: File RouteToDB.tem

10.12 GateEvaluator

The GateEvaluator is used to feed signal tuples controlled by an Enable signal to a further processing. A typical application is the storage of integrals: When a signal is integrated, e.g. the friction energy, that is entered into the coupling during a circuit, then one can generate a function that starts with the beginning of the Circuit sets the energy integral to zero, and from then on the energy is summed. As soon as the circuit is finished the current signal values of the energy are further processed, e.g. by histogram. In this case, the integral is called Use the input of the gate evaluator and

the signal indicating "circuit active" as enable signal. At the end of the circuit the value of the energy integral passes through the gate into the histogram and can be counted. Should the timing of the signals are not changed, the GateRealTimeEvaluator can be used. (Siehe auch 10.13)

In this example, the signal value `s1` is passed once to the signal `o1` when the signal is `en` shows a falling edge. (value drops to zero).

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>GateEvaluator</evaluator>
  <description>Test</description>
  <instance>
    <signal name="s1"/>
  </instance>
  <content>
    <gate enable="en">
      <out name="o1"/>
    </gate>
  </content>
</template>
```

Listing 159: File test.tem

With the attribute `singleShot="false"` in the `<gate/>` tag can be used to permanently enable pass-through. It takes place with the time raster of the enable signal, i.e. at each value of the enable signal once the input values passed through.

Several signals can also be "gated". There must always be so many output signals in the tag `<gate/>`, as signals were specified in the `<instance/>` tag.

If not all signals have been received yet, no signals are passed through, even if the Enable signal drops to zero.

10.13 GateRealTimeEvaluator

The GateRealTimeEvaluator is used to switch signals on and off with an Enable signal. For this purpose, the input signals are mapped to corresponding output signals. In contrast to the GateEvaluator, the timing of the output signals corresponds to that of the corresponding input signals. (Siehe auch 10.12)

In this example, the signal value `s1` is passed once to the signal `o1` when the signal is `en` is greater than 0.5.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>GateEvaluator.GateRealTimeEvaluator</evaluator>
  <description>Test</description>
  <instance>
    <signal name="s1"/>
  </instance>
  <content>
    <gate enable="en">
      <out name="o1"/>
    </gate>
  </content>
</template>
```

Listing 160: File test.tem

Several signals can also be "gated". But there must always be so many output signals in the tag `<gate/>`, as signals were specified in the `<instance/>` tag.

The default attribute of the `<gate/>` tag can be used to specify how the gate should behave when no signal value for the enable signal has been received yet. With `default="true"` the gate in this case is open. Without this specification it is closed

until the first enable value is received.

10.14 Geometry

10.14.1 GeometryCalculationEvaluator

The GeometryCalculationEvaluator allows calculations with 2-dimensional objects. The objects can be created using the GeometryObjectEvaluator and then exist in the form of special signals. The Evaluator enables translations (TRANSLATE), rotations (ROTATE) and scaling (SCALE) of the input object as well as distance calculations (DISTANCE). as well as distance calculations (DISTANCE) to another object. The distance is assumed to be the smallest distance between the object edges. If the objects touch or overlap, the distance is 0.

In this example, the object obj1 is moved by x and y and then the distance to the original object obj1 is calculated. A distance calculation can only ever be carried out as the last operation, because the result (output signal) of this calculation is scalar and therefore cannot be used as input for the other operations. The name of the output signal is dist.

```
<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>GeometryCalculationEvaluator</evaluator>
    <instance>
      <signal name="obj1" />
    </instance>
    <description short="o2" group="" level="4" />
    <noLog />
    <content outName="dist">
      <op opType="TRANSLATE">
        <signal name="x" />
        <signal name="y" />
      </op>
      <op opType="DISTANCE">
        <signal name="obj1" />
      </op>
    </content>
  </template>
</templates>
```

Listing 161: File calculation1.tem

10.14.2 GeometryObjectEvaluator

The GeometryObjectEvaluator creates 2-dimensional objects (polygons), which are output in the form of special signals. The signals can then be further processed with the GeometryCalculationEvaluator.

In this example, a square with an edge length of 2 is generated. The associated signal has the name "object". The object position and orientation angle are determined from the signals x, y and r. Alternatively, the three input signals (all or none) can also be omitted, then the object remains in the coordinate system origin.

```

<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>GeometryObjectEvaluator</evaluator>
    <description short="test" group="" level="1" />
    <noLog />
    <content outName="object">
      -1,-1
      -1,1
      1,1
      1,-1
    </content>
    <instance>
      <signal name="x" />
      <signal name="y" />
      <signal name="r" />
    </instance>
  </template>
</templates>

```

Listing 162: File object.tem

10.15 Graphic

10.15.1 PreviewGraphicCreator

The `PreviewGraphicCreator` can be used to create preview graphics for events that have occurred. It replaces the previous method. (Siehe auch 4.3)

This example shows how the evaluator works. It is only provided with general information over the graphic bedatet. All signal-specific information is stored in the `<signalToView/>` tag of the respective Templates that generate the events for which a preview graphic is to be created. The template `graphic.tem` must be present once in the evaluation. This template creates then the graphs for each event from each evaluator. In the tag `<times/>` the time settings for the graphic are made. In the `pre` attribute the pretrigger time is specified, in the `post` attribute the posttrigger time is specified and in the `maxEventTime` attribute the maximum event duration to be expected is specified. Lasts an event is longer, the post and pretrigger times can no longer be observed. The file `event.tem` represents an example of such a template. The signal used in the tag `<signalToView/>` specified signals are included in the graph. The `<graphic/>` tag is used to configure the actual graphic. The attributes `name`, `width` are defined, `height`, `lines`, `points`, `interpolate`, `showGrid`, `dynamic`, `mark`, `autoscale`, `backgroundColor`, `textColor`, `gridColor` and `fontSize`.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>GraphicEvaluator.PreviewGraphicCreator</evaluator>
  <description>Test</description>
  <content>
    <times pre="30" post="30" maxEventTime="10" />
    <graphic dynamic="true" />
  </content>
</template>

```

Listing 163: File graphic.tem

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>FunctionEvaluator</evaluator>
  <description>Test</description>
  <instance> <signal name="s2"/> </instance>
  <content>
    <parameter> <par name="a"/> </parameter>
    <function>
      if (a>0.999) result=1;
    </function>
  </content>
  <signalToView>
    <signal name="s2" ymin="0.8" ymax="1" yaxis="true"/>
    <signal name="s1" ymin="-1" ymax="3" yaxis="true"/>
  </signalToView>
</template>

```

Listing 164: File event.tem

10.15.2 StaticGraphicCreator

The `StaticGraphicCreator` can be used to create preview graphics that are not created by a `Event`, but are always generated. It replaces the previous method. (Siehe auch 4.3)

This example shows how the evaluator works. The attributes `yaxis`, `ymin` and `ymax` specified in the instance in the `<signal/>` tag are used to configure the signal. The allowed attributes of the `<graphic/>` tag are described at the `PreviewGraphicCreator`. (Siehe auch 10.15.1)

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>GraphicEvaluator.StaticGraphicCreator</evaluator>
  <description>TestGraphic</description>
  <instance>
    <signal name="s1" yaxis="true"/>
    <signal name="s2" yaxis="true"/>
  </instance>
  <content>
    <graphic ydynamic="true" autoscale="2"/>
  </content>
</template>

```

Listing 165: File graphic.tem

10.16 MapEvaluator

The map evaluator can be used to store maps and to evaluate values using the of these maps to be interpolated. The output signal of the evaluator reflects the quantity determined by the map. Between the bases of the map, the value is interpolated. One, two and three dimensional maps are possible.

The following example shows the usage for a simple one-dimensional map shown. The first line in the `Data` field indicates the input values, the second line indicates the Initial values. If the signal `s1` assumes the value 2, the output signal `out` take the value 4. It maps 3 to 9, 4 to 16, and 5 to 25. It is not extrapolated: All input values greater than 5 are mapped to 25, all Input values smaller than 1 to 1.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>KennfeldEvaluator</evaluator>
  <description>Test</description>
  <instance>
    <description name="Test"/>
    <signal name="s1"/>
  </instance>
  <result name="out"/>
  <content>
    <data>
      1,2,3, 4, 5
      1,4,9,16,25
    </data>
  </content>
</template>

```

Listing 166: Example

Here the use with a two-dimensional characteristic diagram shown. The first line in the Data field gives the input values of the first signal *s1*, the first value in each of the following lines the input values of the signal *s2*. The values inside the table indicate the output signal. If the signal *s1* takes the value 5 and *s2* takes the value 4, the Output signal out assume the value 6.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>KennfeldEvaluator</evaluator>
  <description>Test</description>
  <instance>
    <description name="Test"/>
    <signal name="s1"/>
    <signal name="s2"/>
  </instance>
  <result name="out"/>
  <content>
    <data>
      3, 4, 5, 6
      2, 0, 1, 2, 3
      3, 1, 2, 3, 4
      4, 4, 5, 6, 7
      5, 9,10,11,12
    </data>
  </content>
</template>

```

Listing 167: Example

Finally, the use of a three-dimensional map is shown. Several two-dimensional maps "are superimposed". The value attribute in the `<data/>` tag contains the values of the *s3* signal. Thus, the point (*s1*=4, *s2*=1, *s3*=3) is mapped to 3, the point (*s1*=4, *s2*=1.5, *s3*=3) to 4.5 and (*s1*=5, *s2*=1, *s3*=5) to 30.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>KennfeldEvaluator</evaluator>
  <description>Test</description>
  <instance>
    <description name="Test"/>
    <signal name="s1"/>
    <signal name="s2"/>
    <signal name="s3"/>
  </instance>
  <result name="out"/>
  <content>
    <data value="2">
      3, 4, 5, 6
      0, 0, 1, 2, 3
      1, 1, 2, 3, 4
      2, 4, 5, 6, 7
      3, 9, 10,11,12
    </data>
    <data value="3">
      3, 4, 5, 6
      0, 1, 2, 3, 4
      1, 2, 3, 4, 5
      2, 5, 6, 7, 8
      3,10,11,12,13
    </data>
    <data value="4">
      3, 4, 5, 6
      0, 8, 9, 10, 11
      1, 9, 10, 11, 12
      2, 12, 13, 14, 15
      3, 17, 18, 19, 20
    </data>
    <data value="5">
      3, 4, 5, 6
      0, 27, 28, 29, 30
      1, 28, 29, 30, 31
      2, 31, 32, 33, 34
      3, 36, 37, 38, 39
    </data>
  </content>
</template>

```

Listing 168: Example

10.17 LIN

10.17.1 LINMessageTimeoutEvaluator

This is a special evaluator that can monitor LIN messages for TimeOuts. Via the `cycles` attribute in the Tag `<ldf/>` the timeout time can be made dependent on the cycle time stored in the LDF. In the following example, the timeout time has been set to 10 cycles. It is also possible to specify a minimum value in the `min` attribute. Timeout time can be set, which must not be fallen short of. If, for example, messages with very short Cycle time e.g. 7ms should be monitored with only 500ms, this can be done with the attribute `min="0.5"` can be done. Likewise, a maximum timeout time of 2 seconds can be set with `max="2.0"` can be set. Furthermore, there is the possibility, in the `factor` attribute to multiply a factor onto the specified timeout time. `factor="1"`) In the `<message/>` tag, the name of the message to be monitored is specified in the `name` attribute. The LDF can be accessed to automatically determine the cycle time of a message. It requires the Specification of the schedule table in the `scheduleTable` attribute.

```
<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>LINMessageTimeoutEvaluator</evaluator>
    <description short="neuer LIN_Test mit LINEvaluator"/>
    <content>
      <message name="lin_BDM.GENs_03"/>
      <ldf scheduleTable="ZuendungEin" cycles="10.0" min="0.5" max="2.0" factor="1.0"/>
    </content>
  </template>
</templates>
```

Listing 169: Simple example

10.18 LogHandlingEvaluator

With the LogHandlingEvaluator the log messages, which are usually written into a log file or into the log window, can be redirected into signals. This way e.g. warnings from file readers can be handled in the analysis.

In the following example, the evaluator is configured to redirect only warnings. Furthermore, INFO messages could also be included.

```
<?xml version="1.0" encoding="UTF-8"?><templates>  <template>
<evaluator>LogHandlingEvaluator</evaluator>    <description short="log" group="" level="0" />
<content info="false" warning="true" out="LOG" keepTextLog="false" />
</template></templates>
```

Listing 170: File log.tem

The keepTextLog attribute can be used to specify that the messages are still written to the log file. If this flag is deactivated, the messages are only output in signal form.

If certain log messages should appear in the text log, it is possible to filter the messages programmatically with a JavaEvaluator. To do this, the desired log signal values must be passed to the forwardToLog function.

10.19 Logging

10.19.1 BinLoggingEvaluator

The BinLoggingEvaluator is used to generate binary data logs. Generally, one of the evaluators for binary data processing is used to generate the input signal for this evaluator. (Siehe auch 10.4.1, 10.4.3)

In this example, the data of the input signal in specified in the attribute in will be written in the file outFile.bin given in the attribute filename. In the attribute preBytes the number of bytes before the event and in the postBytes attribute the number of bytes after the event can be specified. This way it is possible to determine how large the generated section around the event should be. With the template event.tem an event is generated if the text Error occurs in the data stream.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>LoggingEvaluator.BinLoggingEvaluator</evaluator>
  <description>Test</description>
  <content signalName="in" preBytes="80" postBytes="80" filename="outFile.bin"/>
</template>
```

Listing 171: File binLog.tem

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>JavaEvaluator</evaluator>
  <description>Test</description>
  <instance> <signal name="in"/> </instance>
  <eventCreator type="state"/>
  <content>
    <parameter> <par name="a"/> </parameter>
    <function>
      String text=a.toString();
      if (text.indexOf("Error")>=0) result=1;
    </function>
  </content>
</template>

```

Listing 172: File event.tem

Alternatively, temporal limits for the output can be specified via the `preTrigger` and `postTrigger` attributes.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>LoggingEvaluator.BinLoggingEvaluator</evaluator>
  <description>Test</description>
  <content signalName="in" preTrigger="0.5" postTrigger="0.5" filename="outFile.bin"/>
</template>

```

Listing 173: File binLog.tem

The `zip` attribute in the `<content/>` tag can be used to enable compression of the binary data.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>LoggingEvaluator.BinLoggingEvaluator</evaluator>
  <description>Test</description>
  <content signalName="in" preBytes="80" postBytes="80" filename="outFile.bin" zip="true"/>
</template>

```

Listing 174: File binLog.tem

If the `oneFilePerTrigger` is set in the `<content/>` tag, only one trace file per trigger event is linked in the result file if several trace files overlap. The trace file is selected for which the event has the greatest distance to the start and end of the trace file. The attribute is set to `false` by default.

If the `onlyCausedFileToTrigger` is set in the `<content/>` tag, the trace files are only attached to the event that triggered the creation.

10.19.2 FrameLoggingEvaluator

The `FrameLoggingEvaluator` is used to write message based log files like ASC or BLF.

The example shows how the evaluator works. In the tag `<trace/>` the settings for the trace file are made. The `pre` attribute specifies the pretrigger time in seconds, in the `post` attribute the post trigger time. With `outFileName` the file name of the output file can be specified. If several files with the same name are written, a numbering is automatically appended to the file name. If an expression in curly brackets is used, it will be evaluated at the trigger time. Thus, as in the example, values from the context can be used. Here, of course, the variable `FILENAME` would have to be defined. For example, this could be done in the template from which the trigger event is fired. The file types ASC, BLF, IMG, LOG, OP2 and PCAPNG are supported. Only message types are written in the respective formats, supported by the respective file format.

If the `zip` attribute is set to `true` in the `<trace/>`, the result file is written to a ZIP archive. The primary attribute (`true` or `false`) can be used to control whether the written file is stored as a primary result file.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>LoggingEvaluator.FrameLoggingEvaluator</evaluator>
  <description>Test</description>
  <content>
    <trace pre="5" post="3" outFileName="{FILENAME+'.asc'}" minMessageCount="1" />
  </content>
</template>
```

Listing 175: File ASCLog.tem

In addition, variables that were specified as “SignalValueToGet” during event creation can be used, should be used when generating the file name (e.g. `outFileName="{FILENAME+'_'+decimalFormat(test,0.00)+''.asc'}"`).

In addition, frame filters can be defined. A filter is defined by a range of message IDs and a channel. In the following example, only messages with IDs from 200 to 400 and from 700 to 800 (inclusive) are written. If in the `<filter/>` tag additionally the `exclude` attribute is set to `true`, the filter is inverted. Thus, all messages to which the filter parameters do not apply are written. Basically, if multiple filters are specified, all messages to which a filter condition applies are written. If further in the `<filters/>` tag the `exclude` attribute is set to `true`, the total filter resulting from the combination of all individual filters is inverted. In addition to CAN frames, FlexRay and Ethernet frames can also be filtered. Then the corresponding attribute `type` must be set for the filter. Ethernet frames can only be filtered by channel, the `from` and `to` attributes have no effect here.

```
<?xml version="1.0" encoding="UTF-8"?>
<content>
  "
  <trace pre="30" post="30" outFileName="{FILENAME+'.asc'}" />
  "
  <filters exclude="false">
    "
    <filter from="200" to="400" channel="1" />
    "
    <filter from="700" to="800" channel="1" />
    "
    <filter from="700" to="800" channel="1" type="FR" />
    "
    <filter channel="1" type="ETH" />
    "
  </filters>
  "
</content>
```

Listing 176: Use of frame filters

If the `oneFilePerTrigger` flag is set in the `<content/>` tag, only one trace file per trigger event is linked in the result file if several trace files overlap. The trace file is selected for which the event has the greatest distance to the start and end of the trace file. The attribute is set to `false` by default.

If the `onlyCausedFileToTrigger` is set in the `<content/>` tag, the trace files are only attached to the event that triggered the creation.

Normally the `FrameLoggingEvaluator` starts a new file only then, if no file is currently open. That is, when a second event occurs after one within the post-trigger time, the data belonging to both trigger events are written to a file. The `overlapped` attribute can be used to change this behavior. If the attribute is set to `true`, a file is written for each event.

```
"<trace overlapping="true" pre="30" post="30" outFileName="{FILENAME+'.asc'}"/>
```

Listing 177: Overlapping output

10.19.3 SignalLoggingEvaluator

The `SignalLoggingEvaluator` is used to write signal based log files like MDF or CSV.

The example shows how the evaluator works. In the tag `<times/>` the time settings for the trace file are made. In the `pre` attribute the pretrigger time is specified, in the `post` attribute the posttrigger time is specified and in the `maxEventTime` attribute the maximum event duration to be expected is specified. Lasts an event is longer, the post and pretrigger times can no longer be observed.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>LoggingEvaluator.SignalLoggingEvaluator</evaluator>
  <description>Test</description>
  <content>
    <author>Author</author>
    <comment>Comment</comment>
    <organization>Organization</organization>
    <identification>Identification</identification>
    <project>Project</project>
    <times pre="30" post="30" maxEventTime="10" />
    <mdf name="Test.mdf" />
  </content>
</template>
```

Listing 178: File `mdfLog.tem`

The boolean attribute `startMdfAtZero` in the `<content/>` tag can be used to control, whether the output file should start with a timestamp of 0s, so a corresponding offset is subtracted from all time stamps. The default value is false. This means that the file retains the original timestamps.

DAT files can also be created. These are normal MDF files with the extension DAT.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>LoggingEvaluator.SignalLoggingEvaluator</evaluator>
  <description>Test</description>
  <content>
    <times pre="30" post="30" maxEventTime="10" />
    <mdf name="Test.dat" />
  </content>
</template>
```

Listing 179: File `datLog.tem`

If the `zip` attribute is set to `true` in the `<mdf/>` tag, the output file compressed.

By using curly brackets, the name can refer to the variables of the Context can be accessed.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>LoggingEvaluator.SignalLoggingEvaluator</evaluator>
  <description>Test</description>
  <content>
    <times pre="30" post="30" maxEventTime="10" />
    <mdf name="{sourcedata+'_Test.mdf'}" />
  </content>
</template>

```

Listing 180: File mdLog.tem

Likewise, the values that are queried via SignalValueToGet can be accessed.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>LoggingEvaluator.SignalLoggingEvaluator</evaluator>
  <description>Test</description>
  <content>
    <times pre="30" post="30" maxEventTime="10" />
    <mdf name="{format(s4Tag,3)+'_Test.mdf'}" />
  </content>
</template>

```

Listing 181: File mdLog.tem

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>FunctionEvaluator</evaluator>
  <description>Test</description>
  <instance> <signal name="s2"/> </instance>
  <eventLevel level="1"/>
  <content>
    <parameter> <par name="a"/> </parameter>
    <function>
      if (a>0.99) result=1;
    </function>
  </content>
  <signalValueToGet>
    <signal tag="s4Tag" name="s4"/>
  </signalValueToGet>
</template>

```

Listing 182: File event.tem

If a list of signals is specified, only these will be included in the MDF file.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>LoggingEvaluator.SignalLoggingEvaluator</evaluator>
  <description>Test</description>
  <instance>
    <signal name="s1"/>
    <signal name="s2"/>
    <signal name="s3"/>
  </instance>
  <content>
    <times pre="30" post="30" maxEventTime="10" />
    <mdf name="Test.mdf" />
  </content>
</template>

```

Listing 183: File mdLog.tem

If a list of signals is specified and the attribute exclude is used in the tag <instance/>

is set to true, the specified signals are not included in the MDF file.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>LoggingEvaluator.SignalLoggingEvaluator</evaluator>
  <description>Test</description>
  <instance exclude="true">
    <signal name="s1"/>
    <signal name="s2"/>
    <signal name="s3"/>
  </instance>
  <content>
    <times pre="30" post="30" maxEventTime="10" />
    <mdf name="Test.mdf" />
  </content>
</template>
```

Listing 184: File mdfLog.tem

If the file ends before the post-trigger time has expired, a correspondingly shortened trace is generated.

If the file ends before the event is finished, a correspondingly shortened trace is also generated.

If many events occur almost simultaneously, not for each event a File is written, but the events are summarized. In the group attribute in the tag `<times/>` specifies the time that must elapse between two events in order for two Files are written. This time should not be greater than the post-trigger time.

Similar to MDF files, a CSV file can be written with the tag `<csv/>`. It both formats can also be generated at the same time. In this case, the other attributes `decimalSeparator`, `columnSeparator`, `completeTable` and `defaultValue` must be specified in the `<csv/>` tag. With `decimalSeparator` the decimal separator for the values in the table is set, with `columnSeparator` the separator for the table columns. With `completeTable="true"` it is ensured that the table contains a value in each cell. Signal values that do not occur in a line (at a point in time) are filled with their predecessor value. If there is no predecessor, the value specified by `defaultValue` is used.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>LoggingEvaluator.SignalLoggingEvaluator</evaluator>
  <description>Test</description>
  <content>
    <times pre="30" post="30" maxEventTime="10" />
    <mdf name="Test.mdf" zip="true"/>
    <csv name="Test.csv" zip="true"/>
  </content>
</template>
```

Listing 185: File mdfCsvLog.tem

The RPC3 format supports only signals of the same type and sampling rate within a file. `FLOATING_POINT` is always selected as the type. Not all software that is able to read RPC3 supports this format. To set the sampling rate, the time interval between each sample (`deltaT`) is specified. If not all signals start at the same time, the signals that start later will be padded with values from `defaultValue`. If the number of sampled readings is not a multiple of 256, so many measuring points with the value `defaultValue` are added at the end of the file that a multiple of 256 is reached.

```

<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>LoggingEvaluator.SignalLoggingEvaluator</evaluator>
    <description short="Logging" />
    <instance />
    <content>
      <times pre="30.0" post="30.0" maxEventTime="10.0" />
      <rpc3 name="out.rsp" zip="false" deltaT="0.1" defaultValue="0.0" />
    </content>
  </template>
</templates>

```

Listing 186: File rpcLog.tem

If the `oneFilePerTrigger` is set in the `<content/>` tag, only one trace file per trigger event is linked in the result file if several trace files overlap. The trace file is selected for which the event has the greatest distance to the start and end of the trace file. The attribute is set to `false` by default.

If the `onlyCausedFileToTrigger` is set in the `<content/>` tag, the trace files are only attached to the event that triggered the creation.

10.20 MeasurementAnalysisEvaluator

The `MeasurementAnalysisEvaluator` is used to check the signals contained in the measurement for plausibility. Two phenomena are detected: The "backward jumping" of the time and an implausibly large duration of the measurement or individual signals. In parallel, information about the signals is collected and entered into the `MetaDatabase`. This way you can get an overview of which signals are included in which measurements and so on.

In the following example, the evaluator is used without a specific condition. The table `signalStatistic` is created, and the following columns are created in this table: `signalName`, `valueCount`, `changesCount`, `minSignalValue`, `maxSignalValue`, `deltaMean`, `deltaDev`, `minSignalTime`, `maxSignalTime` and `timeErrors`. In the column `changesCount` you can find the number of the actual changes of the signal value, in the column `deltaMean` the mean value of the Differences of two successive measured values, and in the `deltaDev` column you can find the Standard deviation of the difference between two successive measured values. The `timeErrors` column contains the Frequency of time-returns (time not monotonously increasing).

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>MeasurementAnalysisEvaluator</evaluator>
  <description>Test</description>
  <content>
  </content>
</template>

```

Listing 187: File test.tem

If on average over all signals mentioned in the header of the file there are only 10 measured values per signal, an event is generated for each signal that does not contain any measured values, and the signal name is specified as Event attribute "MissingValues" attached to the event. This function should be used with caution when using DBCs, as it can the rule here is that messages are defined but not sent in all vehicles.

It can be checked whether a measurement actually contains required signals. For this purpose, the names of the signals are specified in a lab file. If one of the signals is not included generates an event.

```
[Labels]
s1
s2
s3
```

Listing 188: File test.lab

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>MeasurementAnalysisEvaluator</evaluator>
  <description>Test</description>
  <content>
    <signalList name="test.lab" type="lab" />
  </content>
</template>
```

Listing 189: File test.tem

If desired, a measurement can be marked as inconsistent in MeDaPF if there are time setbacks or if the measuring duration assumes an implausibly large value. In the enable of the <markNotValid/> tag, the function is enabled, and in the maxTime attribute the maximum possible measuring duration is specified. If individual signals have time lags, or if one of the signals is exceeds the maximum measurement duration, the file is set To inconsistent. In the inconsistency message is displayed with tNotMonoton=[n] specifies how many signals have time lags and toLong=[m] specifies how many signals have time lags are too long.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>MeasurementAnalysisEvaluator</evaluator>
  <description>Test</description>
  <content>
    <markNotValid enable="true" maxTime="20"/>
  </content>
</template>
```

Listing 190: File test.tem

10.21 MinMaxEvaluator

The MinMaxEvaluator is suitable for determining the minimum and maximum values of a series of input signals. Maximum to spend.

In this example, the minimum and maximum of the three signals s1, s2 and s3 is calculated and output in the test_min and test_max signals. The prefix which is put in front of the max or min to define the signal names is specified in the name attribute in the <minmax/> tag.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>MinMaxEvaluator</evaluator>
  <description>Test</description>
  <instance>
    <signal name="s1"/>
    <signal name="s2"/>
    <signal name="s3"/>
  </instance>
  <content>
    <minmax name="test_" />
  </content>
</template>
```

Listing 191: File test.tem

10.22 PreprocessingEvaluator

The PreprocessingEvaluator can be used to preprocess signals. In particular, it is possible to move signals into the past in time or to calculate statistical values (e.g. an average value) or modified signals in advance, so that the values can be used together with other signals during evaluation.

Basically, there are three types of calculations. A simple calculation can be made with a "processing". For example, an average or a pure time shift. An "virtualSignal" can be used to calculate any virtual signal via an expression. The expressions have the same syntax as described in the chapter on virtual signals. However, the signals can also be shifted backwards in time to compensate for the latencies of the calculation. A filter can be calculated with a "filter". Butterworth and Bessel filters are available as low pass, high pass, band pass and band lock. In addition, a "phaseless" calculation is possible, in which the filter is repeated after execution applied backwards to the result to compensate for phase effects.

The order in which the calculations are defined does not play a role in the execution. They can also be defined in multiple evaluator instances for better visibility. If signals reference each other, there must only be a working calculation sequence. Circular dependencies cause an error.

In the example, several calculations are applied to the "input" signal.

First, two filters are calculated, one with a Butterworth low pass and one with a Bessel band pass. For the low pass, only the limit frequency "f0" is set to 10Hz. For the bandpass filter, the passing range is set by "f0" and "f1". Both filters have the filter order 5 and use a sampling rate of 100Hz for the calculation. The low pass filter also has "padding" and "phaseless" active. Padding prevents the filter from swinging in for longer by initializing it as if the input signal had been the same value as the start value for an infinite time before the beginning of the measurement. The phaseless calculation compensates for phase shifts that would occur by the filter, By performing the filtering backwards a second time after execution.

As an example of a virtual signal, the difference between input and mean is calculated here. Here the mean signal, which is generated in another calculation, is used as an additional input signal.

An average, minimum and maximum are calculated as examples of "processings". In addition, the signal delay is calculated as the input signal shifted one second into the past. The "delay" parameter, which causes the time shift, is available for all calculations. To compensate latencies, for example.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<template>
  <evaluator>PreprocessingEvaluator</evaluator>
  <description>TestTemplate</description>
  <content>
    <filter name="filteredTP" signal="input" samplingRate="100"
      filterType="butterworth,tp" f0="10" f1="" order="5"
      delay="" phaseless="true" padding="true" />
    <filter name="filteredBP" signal="input" samplingRate="100"
      filterType="bessel,bp" f0="10" f1="20" order="5"
      delay="" phaseless="false" padding="false" />
    <virtualSignal name="vs" expression="input - mean" delay="" />
    <processing name="mean" signal="input" algorithm="mean" delay="" />
    <processing name="min" signal="input" algorithm="min" delay="" />
    <processing name="max" signal="input" algorithm="max" delay="" />
    <processing name="delay" signal="input" algorithm="delay" delay="-1.0" />
  </content>
</template>
```

Listing 192: File example.tem

10.23 Programmable

10.23.1 FunctionEvaluator

The FunctionEvaluator allows to implement a function directly in Java. Thereby the Java code can be inserted directly into the template.

In this example, the FunctionEvaluator is used to evaluate the temporal distance between two signal values of a signal and generate a event, if the distance exceeds 1.5s. In the `<parameter/>` tag symbolic names for the signals mentioned in the instance are assigned. The number of `<par/>` tags must be set with the Match number of `<signal/>` tags in the instance. In the example the signal `s1` from the measurement is used and addressed with the name `a` in the template. In the `<global/>` tag, global variables or functions can also be declared. Here the variable `lastTime` is declared. The code in the `<first/>` tag is executed once, as soon as for all used signals Signal values are known. In the `<function/>` tag the actual function is described. This code is called every time a signal value changes. In tag `<first/>` and in the `<function/>` tag the implicit variable `time` can be used. You contains the current measurement time in seconds. The variable `result` takes over the Result of the evaluation: A one means that the event has occurred for sure, a zero, that the event has not occurred. Default value `result` is zero.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>FunctionEvaluator</evaluator>
  <description>Test</description>
  <instance> <signal name="s1"/> </instance>
  <content>
    <parameter> <par name="a"/> </parameter>
    <global> double lastTime; </global>
    <first> lastTime=time; </first>
    <function>
      if (time-lastTime>1.5) result=1;
      lastTime=time;
    </function>
  </content>
</template>
```

Listing 193: File test.tem

The Java functions of the evaluator are basically called only then, if all input variables have a defined value. Normally, this means that each input signal must have supplied a value. To avoid that individual signals prevent the call, default values can be set. Default values can be set in the instance or in the parameters for each signal individually. If necessary, the default value of the parameter overwrites the value from the instance. In the following example, `s1` (or `a`) is thus initialized to the value 2.0, while the value 1.0 is used for the other two signals.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>FunctionEvaluator</evaluator>
  <description>Test</description>
  <instance>
    <signal name="s1" default="1.0"/>
    <signal name="s2" default="1.0"/>
    <signal name="s3" default="1.0"/>
  </instance>
  <content>
    <parameter>
      <par name="a" default="2.0"/>
      <par name="b"/>
      <par name="c"/>
    </parameter>
    <first>
      getKontext().setValueParent("a", a);
      getKontext().setValueParent("b", b);
      getKontext().setValueParent("c", c);
    </first>
  </content>
</template>

```

Listing 194: File test.tem

The following example uses the `<finally/>` tag. Here code can be specified which is executed only once at the end of the measurement. The example declares the Variable `valueHigh`. This is initialized with `false` and set to `true` is set as soon as a signal value greater than 10 occurs. At the end, in the `<finally/>` tag, a event is generated if `valueHigh` has the value `false`. So an event is created, if no signal value has exceeded the value 10. A statement that can be made first, when all signal values have been considered.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>FunctionEvaluator</evaluator>
  <description>Test</description>
  <instance> <signal name="s1"/> </instance>
  <content>
    <parameter> <par name="a"/> </parameter>
    <global> boolean valueHigh=false; </global>
    <function>
      if (a>10) valueHigh=true;
    </function>
    <finally>
      if (!valueHigh) result=1;
    </finally>
  </content>
</template>

```

Listing 195: File test.tem

Besides the shown tags for Java code there are `<before/>` and `<after/>`. The corresponding code sections are executed before or after the complete analysis, i.e. before or after all evaluators have been initialized or deinitialized. In `<before/>` code can be executed that creates preconditions for evaluator initialization, such as generating global signals or even generating additional evaluators. In `<after/>`, for example, result files can be manipulated, as this ensures that all evaluators have already completed their files. In the following example some functions are shown, which are used within the Java code are available. (`setDoubleValue`, `setStringValue`, `wasReceived`, `createEvent`, `setDescription`, `format`, `getSigName`, `setEventLevel`, `getRTDate`, `isDataCollector`, `getDataCollector`, `getLogger` and `abortEvaluation`). To use the Java classes `DataCollector` and `Logger` to use, these classes must be imported with the `<import/>` tag.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>FunctionEvaluator</evaluator>
  <description>Test</description>
  <instance>
    <signal name="s1"/>
    <signal name="s2"/>
  </instance>
  <content>
    <import>IAV.Ausw.util.DataCollector.DataCollector</import>
    <import>java.util.logging.Logger</import>
    <parameter> <par name="a"/> <par name="b"/> </parameter>
    <function>
      setDoubleValue("a_Wert",a); // Attaches the value of a as an event attribute
                                   // with the name "a_Wert" to the upcoming event.
      setStringValue("a_Text","Hallo");// Attaches the string "Hallo" as an event attribute
                                   // with the name "a_Text" to the upcoming event.
      if (inCurrentCall(PAR_B)) // Tests if "b" has been updated in current call.
        createEvent(time,time);// Creates an event
      else
        createEvent(time,time,"name");// Creates an event incl. description.
      setDescription("test"); // Sets the description \"test\" for the upcoming event.
      out(format(b,2)); // Outputs "b" with two decimal digits.
      out(getSigName(PAR_B)); // Outputs the signal name of "b", here "s2".
      setEventLevel(1); // Sets the EventLevel for all following events to 1.
      out(getRTDate(time)); // Outputs the real time as a Date object.
      if (isDataCollector()) { // Checks if there is a DataCollector.
        DataCollector dc = getDataCollector(); // Writes a value a
        dc.setValue(getRTDate(time),"Test",a); // into the database.
      }
      Logger log = getLogger(); // Returns a Java logger
      log.warning("This is a warning!"); // with which e.g. warnings can be issued.
      // These also appear in the result in the measurement data platform.
      abortEvaluation("Ende"); // The evaluation is canceled here
      log.warning("This warning will not be generated anymore.");
    </function>
  </content>
</template>

```

Listing 196: File test.tem

In the following example, the abortEvaluation function is used in combination with the <finally/> tag. The evaluation is aborted if the signal is greater than 2. Nevertheless, after aborting still the instructions is executed in the Finally block.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>FunctionEvaluator</evaluator>
  <description>Test</description>
  <instance> <signal name="s1"/> </instance>
  <content>
    <parameter> <par name="a"/> </parameter>
    <function>
      if (a>2) abortEvaluation("Signal > 2");
    </function>
    <finally>
      createEvent(time,time);
    </finally>
  </content>
</template>

```

Listing 197: File test.tem

There are also various auxiliary methods and classes available for use in the Func-

tionEvaluator with the help of which it is possible, access signals and files and insert virtual files into the AMeDA context. In this way, templates can even be generated at runtime or existing templates can be adapted. The following table shows a list of these auxiliary methods.

	function	description
String	getSignalName(final int n)	Returns the name of the nth signal from the evaluator instance. In the process, any alias used will be resolved.
String	getSignalNameFromTemplate(final int n)	Returns the name of the nth signal from the evaluator instance directly.
List<SignalInfo>	getSignalList(final String pattern)	Returns a list of all signals included in the measurement. The pattern parameter is optional and specifies a regular expression used to filter them by name.
List<String>	getMeasurementFiles(final String pattern)	Returns all files from the measurement archive as URL string, on which the pattern fits. If pattern is omitted, all files are simply returned.
TemplateBuilder	template(String templateFile)	Returns a TemplateBuilder which can be used to manipulate the corresponding existing template.
void	addFileToAnalysis(final String fileName, final String content)	Adds a virtual file with given file name and content to the analysis.
List<CsvLine>	getCsvLines(final String url, final boolean withHeader, final String format, final String charset)	Reads a CSV file and returns the content in structured form. With code withHeader=false the first line is not interpreted as a header but as a normal data line. The format can be excel, mysql or rfc4180. With charset a character set can be selected. Default is UTF-8. If the last three parameters are omitted, the header is interpreted as such and the ordinary CSV format with UTF-8 character set is used.
List<CsvLine>	getCsvLines(final InputStream inputStream, final boolean withHeader, final String format, final String charset)	Behaves like the previous method, but uses an InputStream to read in instead of a URL.
org.jdom2.Element	getRootXmlElement(final String url)	Returns the root element of an XML file as a JDOM element.
Attributes	measurementAttributes()	Returns an Attributes object for manipulating measurement attributes in the measurement data platform.
Attributes	vehicleAttributes()	Returns an Attributes object for manipulating vehicle attributes in the measurement data platform.
Attributes	loggerAttributes()	Returns an Attributes object for manipulating data logger attributes in the measurement data platform.
void	publish(final String alias, final Object value)	Publish a signal value for a signal declared as output. The alias corresponds to the one set in the output tag. Value must have the appropriate type.
boolean	inCurrentCall(final int signalIndex)	Checks whether the signal with the given signal index (0 corresponds to the time signal) was received at the current time.
void	abortEvaluation(final String reason)	Aborts the evaluation and throws an exception with corresponding error message.
void	abortEvaluationWithoutError()	Aborts the evaluation without error message.
boolean	isErrorTermination()	Can be called in the Finally block to check if the analysis was terminated by an error.
boolean	evaluationWillContinue()	Can be called in the Finally block to check if the analysis is still continued (e.g. by continueMulti-log)

The `SignalInfo` class is used to describe signals. It provides the following methods. Depending on the file format, individual methods may not return usable values.

function		description
long	getChannelId()	Returns the ID of the channel from which the signal was recorded.
String	getChannelName()	Returns the name of the channel from which the signal was recorded.
long	getMessageId()	Returns the ID of the message from which the signal was recorded.
String	getMessageName()	Returns the name of the message from which the signal was recorded.
String	getFile()	Returns the name of the file from which the signal originates.
String	getPduName()	Returns the name of the PDU from which the signal originates.
String	getName()	Returns the signal name.
String	getDescription()	Returns the description of the signal.
String	getSourceInfo()	Returns the content of the SI block of an MDF file

The CsvLine class is used to abstract a line from a CSV file. The class provides the following methods.

method		description
String	get(int col)	Returns the entry from the column col.
String	get(String col)	Returns the entry from the column col, where col corresponds to the name of the column from the header. For this, the first column must have been interpreted as a header

The TemplateBuilder class can be used to build or manipulate templates. It provides the following methods according to the builder pattern principle.

method		description
	TemplateBuilder(final String evaluatorType, final String description)	Creates a TemplateBuilder for the specified evaluator type.
	TemplateBuilder(final element rootElement)	Creates a TemplateBuilder from a given root element.
TemplateBuilder	removeTag(final String tagPath)	Removes a tag (referenced by an XPath expression). If several matching tags exist, all will be removed.
TemplateBuilder	setAttribute(final String elementPath, final String attribute, final String value)	Sets the given attribute in the given element (tag) to the corresponding value.
TemplateBuilder	addElement(final String parentPath, final String elementName, final String text)	Adds an element (tag) with given parent element, name and text.
TemplateBuilder	withInstance(final String ... signalNames)	Adds an instance with given signal list to the evaluator.
String	build()	Creates the template as an XML string

The Attributes class is used to read and write attributes in the measurement data plate format. An instance of the vehicle, data logger, or measurement attribute class can be created using one of the methods vehicleAttributes(), loggerAttributes() and measurementAttributes().

method		description
TreeMap<Date, Object>	getAttributeValues(final String label)	Returns the attributes for a given label.
Object	getAttributeValue(final String label)	Returns the static attribute to a given label.
void	setAttribute(final String label, final String int double boolean value, final Date date)	Sets the attribute for a label. If the 'date' parameter is not passed, the attributes are set statically

The following example shows how an existing template "function.tem" can be manipulated in this way, that it is called for each signal with a signal name matching the regular expression "s.*". The template can then be used in a second analysis run. For this, a TwoPassEvaluator is assumed here, which uses "second.tko" as TKO file.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>FunctionEvaluator</evaluator>
  <description>Test</description>
  <content>
    <finally>
      TemplateBuilder templ = template("function.tem").removeTag("//template/instance");
      for (final SignalInfo info : getSignalList("s.*")) {
        templ = templ.withInstance(info.getName());
      }
      addFileToAnalysis("second.tko", "use(\"test\")");
      addFileToAnalysis("test.tem", templ.build());
    </finally>
    <import>com.iav.functions.TemplateBuilder</import>      <import>com.iav.functions.SignalInfo
  </content>
</template>

```

Listing 198: File functionGenerator.tem

Besides returning a signal value via the result variable there is the possibility, Output signals via the publish() method. The signals must be registered under the <instance/> tag for this. Here also an alias is assigned, under which the signal is accessible with the publish() method. With name the output signal name is specified. In addition, a type can be specified with. All data types supported by AMeDA can be used as type, so frame types as well as simple types like string, long or double. Default is Double.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>FunctionEvaluator</evaluator>
  <description>Test</description>
  <instance>
    <signal name="s1"/>
    <output name="v1" alias="a1"/>
    <output name="v2" alias="a2" type="CANData"/>
  </instance>
  <content>
    <parameter> <par name="a"/> </parameter>
    <function>
      publish("a1", a * 2);
      publish("a2", new IAV.Ausw.FileFilter.CAN.CANData(time, 1, 0x100, 1, 2, 3));
    </function>
  </content>
</template>

```

Listing 199: File test.tem

If more complex Java code is to be included in the analysis, it is also possible to include complete Java classes. For this purpose, the corresponding Java file must be added as an additional file. Then it can be referenced with the <javaFile/> tag. Content (esp. classes) from the Java files included in this way can then be used in the code block. If the Java file used contains a package directive, the package must be included in name by using the fully-qualified name.

```

<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>FunctionEvaluator</evaluator>
    <description short="function" />
    <content>
      <javaFile name="TestClass" file="TestClass.java" />
      <parameter>
        <par name="s1" />
      </parameter>
      <function>result = TestClass.timesTwo(s1);</function>
    </content>
    <instance>
      <signal name="s1" />
    </instance>
    <result name="out"/>
  </template>
</templates>

```

Listing 200: File test.tem

There is a possibility to include JAR files in the analysis, which can be configured via the analysis configuration are delivered as additional files. However, only JAR files that have been previously verified and signed by the AMeDA team can be used. A signature file can be created for this upon request, which then (for each JAR file) must also be stored as an additional file. To account for the additional JAR files when using Java evaluators, a context variable with the name "classpath" must be created. The variable must be of type string and the names of all additional JAR files to be used, separated by a semicolon. So that the additional dependencies in Idea are taken into account during code completion, you can add them as system-scoped dependencies to the POM file of the Idea project. It should be noted that everything that is not within the comment lines provided for this purpose, will be overwritten the next time Idea is opened.

```

...
<dependencies>
...
</dependency>
<!-- Place additional dependencies here -->
<dependency>
<groupId>de.me</groupId>
<artifactId>anyName</artifactId>
<version>1.0</version>
<scope>system</scope>
<systemPath>D:\data\myJars\myOwnLibrary.jar</systemPath>
</dependency>
<!-- end additional dependencies -->
</dependencies>
...

```

Listing 201: Example

10.23.2 JavaEvaluator

The JavaEvaluator is very similar to the FunctionEvaluator. The main difference from this is that the JavaEvaluator, unlike the FunctionEvaluator, can handle types other than double.

As with the FunctionEvaluator, the Java code is executed inside a container that is Calls to safety-critical methods (such as `System.exit(0)`), prohibits. When using of such methods an `AccessControlException` is thrown. (Siehe auch 10.23.1)

In the JavaEvaluator the signal value is not passed directly, but the data container of the signal value. This means that not as in the FunctionEvaluator directly e.g. a

double is passed but a DataDouble object. The signal value can then be accessed here with the getter `getValue`. No AutoBoxing is required in this way. So it is possible to calculate with double values throughout. There is no detour via a Double object, which must be allocated for each new signal value would have to. (The DataDouble instance is used over and over again, so only one instance is created per measurement)

A special feature of using the Data objects is that they contain the raw values of the signals. So with `getValue()` you do not necessarily get the physical value of the signal depending on the type of input data, but the raw value, as found for example in a CAN message. The physical value is obtained from a conversion formula, which can be found, for example, in the DBC description. To still be able to work with physical values, either the FunctionEvaluator can be used, or a virtual signal can be applied, which is used instead of the desired signal. An alias is not sufficient in this case. Since AMeDA virtual signals that pass only one signal, to aliases, the virtual signal function must be explicitly assigned a factor of 1 in this case.

As default value the JavaEvaluator allows the value "null". Other values are not possible.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>JavaEvaluator</evaluator>
  <description>Test</description>
  <instance> <signal name="s1"/> </instance>
  <content>
    <parameter> <par name="a"/> </parameter>
    <global> boolean checkAlarm=true; </global>
    <function>
      if (a.getValue()>10) checkAlarm=false;
    </function>
    <finally>
      if (checkAlarm) result=1;
    </finally>
  </content>
</template>
```

Listing 202: File test.tem

The following example shows how to access a CAN message. The signal `canMessage` is an internal signal, via which all CAN messages be made available. This signal can be accessed in the same way as a numerical value.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>JavaEvaluator</evaluator>
  <description>Test</description>
  <instance>
    <signal name="ant.TestSignal1"/>
    <signal name="canMessage"/>
  </instance>
  <content>
    <parameter>
      <par name="a"/>
      <par name="can"/>
    </parameter>
    <function>
      if (a.getValue()>10 & & can.getBytes(1)==12 & & can.dlc==2)
        result=1;
    </function>
  </content>
</template>
```

Listing 203: File test.tem

The signal values of the signal `canMessage` are of type `CANData`. The `CANData` class provides the following methods, with which information about the message can be retrieved.

	function	return	parameter
boolean	<code>isExtended()</code>	True if it is the Extended format (29-bit identifier)	none
int	<code>getId()</code>	ID of the CAN message (bit 30 is set for the Extended format)	none
boolean	<code>isLin()</code>	True, if it is a LIN message (LIN messages are also called as CAN-Data objects processed)	none
int	<code>getChannel()</code>	channel on which the message was received (at LIN offset of 100)	none
int	<code>getBusBits()</code>	size of the message on the bus (stuffing is not considered)	none
int	<code>getDlc()</code>	length of message	none
int	<code>getBytes(int i)</code>	i-th data byte of the message	number of the desired data byte

The following example shows how to output logger warnings.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>JavaEvaluator</evaluator>
  <description>Test</description>
  <instance> <signal name="s1"/> </instance>
  <content>
    <parameter> <par name="a"/> </parameter>
    <function>
      getLogger().warning("a logger warning message");    </function>
    </content>
  </template>
```

Listing 204: File test.tem

For signals that have either a numeric type or a string representation, a class (string or double) can be specified with the `dataClass` attribute of the `<par/>` tag. The variable can then be processed directly as a string or double variable.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>JavaEvaluator</evaluator>
  <description>Test</description>
  <instance> <signal name="Fehlertext"/> </instance>
  <content>
    <parameter> <par name="a" dataClass="String"/> </parameter>
    <function>
      if (a.equals("kein_Fehler")) {
        result = 1;
      } else {
        result = 0;
      };
    </function>
    <finally>
    </finally>
  </content>
</template>
```

Listing 205: File test.tem

The following example uses the `abortEvaluation` function. The evaluation is aborted, but the `Finally` block is executed with the time of the abort.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>JavaEvaluator</evaluator>
  <description>Test</description>
  <instance> <signal name="s1"/> </instance>
  <content>
    <parameter> <par name="a"/> </parameter>
    <function>
      createEvent(time, time);
      if (a.getValue()>1) abortEvaluation("Abbruch gewollt!");
    </function>
    <finally>
    </finally>
  </content>
</template>
```

Listing 206: File test.tem

10.24 RealTimeEvaluator

The RealTimeEvaluator is used to generate a real-time. In the vast majority of data formats the time is given only relative to a start time. Thus, for most data loggers, the time starts at Zero when the logger is switched on. In order to provide events with an absolute timestamp, this evaluator is used. It determines the time from measurement signals stored in the measurement itself. This can be e.g. a Can message from the FZG gateway. The time of this evaluator is then used for the timing of all events of the measurement is used. If several templates are included with the RealTimeEvaluator, the first one serves in a `use` statement specified as time reference. A possible drift between the time of the logger and the time of the vehicle or other source is taken into account and corrected as far as possible.

The following example shows individual signals for the values of hour, minute, second, day, month and year specified. In this case, the order of the signals in the tags `<signal/>` under the tag `<instance/>` match the type description in the `<time/>` tags under the `<content/>` tag. With the attribute `interval` in the `<mode/>` tag is used to specify whether the absolute start and end times of the measurement are included in the MetaDatabase is to be transferred. A `true` means that the values are determined. If a timestamp in the format "dd.MM.yyyy HH:mm" is specified in the context in the variable `realtimeevent`, an event is generated at this time. In this way, time slices can be created.

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<template>
  <evaluator>RealTimeEvaluator</evaluator>
  <description>Echtzeiterzeugung</description>
  <instance>
    <signal name="fah.UH_Stunde" default="0"/>
    <signal name="fah.UH_Minute" default="0"/>
    <signal name="fah.UH_Sekunde" default="0"/>
    <signal name="fah.UH_Tag" default="0"/>
    <signal name="fah.UH_Monat" default="0"/>
    <signal name="fah.UH_Jahr" default="0"/>
  </instance>
  <content>
    <mode interval = "true" />
    <time is="HOUR"/>
    <time is="MINUTE"/>
    <time is="SECOND"/>
    <time is="DAY"/>
    <time is="MONTH"/>
    <time is="YEAR"/>
  </content>
</template>

```

Listing 207: File RealTime.tem

If a timestamp is available in seconds UTC, which is the number of milliseconds passed since the 01.01.1970 UTC, only this signal can be specified.

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<template>
  <evaluator>RealTimeEvaluator</evaluator>
  <description>Echtzeiterzeugung</description>
  <instance>
    <signal name="UTCtime" default="0"/>
  </instance>
  <content>
    <mode interval = "true" />
  </content>
</template>

```

Listing 208: File RealTime.tem

When using a UTC time, the time which is read out from the file can be compared with the readout time of the Measurement data platform can be checked for plausibility. If the difference is greater than specified in the `<timeCorrect/>` tag (in days), the time of the measurement is ignored and instead the readout date of the measurement data platform is used as the Reference time used. This can be used to process measurements that are not available due to problems with the Real time clock of a measuring device were dated far into the future or far into the past.

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<template>
  <evaluator>RealTimeEvaluator</evaluator>
  <description>Echtzeiterzeugung</description>
  <instance>
    <signal name="UTCtime" default="0"/>
  </instance>
  <content timeCorrect="2192">
    <mode interval = "true" />
  </content>
</template>

```

Listing 209: File RealTime.tem

10.25 Controller

10.25.1 Schwingen

The `ReglerSchwingen` evaluator is used to adjust the to determine maximum and minimum deviations from the set point. During the control process, indicated by an enable signal, the relative maxima and minima of the actual signal are determined, and each in these extremes determines the deviations from the nominal value. The maximum upward and downward deviations will then be indicated by a falling edge of the enable signal in two signals after the end of the control process. The maximum of the two values is output as the result of the evaluator, and can be used for event generation.

A corresponding template is described below. In the tag `hyst` the hysteresis for the extreme value detection is specified, In the `minOutName` and `maxOutName` attributes they are specified signal names, in which the respective maximum and minimum Regler deviations are output.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Regler.ReglerSchwingen</evaluator>
  <description>Test</description>
  <instance>
    <signal name="ist"/>
    <signal name="soll"/>
    <signal name="enable"/>
  </instance>
  <content hyst="0.01" minOutName="min" maxOutName="max"/>
</template>
```

Listing 210: File `regler.tem`

10.26 Resampling

10.26.1 GroupResamplingEvaluator

The `GroupResamplingEvaluator` is used to convert signals into groups with equal sampling. This is especially useful as pre-processing for an export to the MDF4 format, since MDF4 basically assumes groups of equally sampled signals in order to optimize data storage. To meet the requirements of the MDF4 format, in addition to the sampled signals, a Validation signal formed, which contains a flag with the information for each signal from the group, whether this signal has already been assigned a valid value. In the MDF4 file this flag is used, to mask out the values that are still invalid. Each evaluator creates a signal group that can also be referenced in the `FileExportEvaluator`, to export all signals of the group as well.

The following example shows the generation of a signal group with a sampling period of 1s. The group gets the name "g1" that can be referenced in `FileExportEvaluator` and each generated signal is prefixed with "g1_" in the name. The `timeout` is set to 0.0, which means that there is no timeout. A larger value would skip correspondingly large gaps in the measurement, so that no unnecessary data is generated. The timeout applies on a group basis, not to each individual signal. The `timeOffset` can be set to shift the sampling grid. For example, with a period of 1 and an offset of 0.1, at locations 1.1, 2.1, 3.1, ... to be scanned.

The parameter can also be used to clean up inaccuracies in timestamps during export. Due to floating point inaccuracies, the time stamps may not be exactly on an integer Multiples of the sampling period. With a period of 1, values would then only be displayed with almost one second Delay can be sampled. This effect can be circumvented by a very small offset. If the offset is small enough, it then disappears during export due to the resolution of the timestamp in the export format again, if necessary.

If the sampled signals are not to be used further, they can also be made invisible.

This is what the `visible` flag is for. If it is set, the signals are not fed to the AMeDA mechanisms for finding signals. Thus, they are invisible to other evaluators. The only way to reference them then is via the group name in `FileExportEvaluator`. This has the advantage that the signals cannot collide with their original signals. This way you can also keep your original name without restriction (no prefix).

```
<?xml version="1.0" encoding="UTF-8"?><templates>
  <template>
    <evaluator>GroupResamplingEvaluator</evaluator>
    <description short="g1" />
    <instance />
    <content visible="true" period="1.0" timeOffset="0.0" prefix="g1_"
      group="g1" timeout="0.0">
      <includes>
        <signal>s.*</signal>
      </includes>
      <excludes />
    </content>
  </template>
</templates>
```

Listing 211: File g1.tem

10.26.2 ResampleEvaluator

The `ResampleEvaluator` is used for resampling signals. The signals specified in the instance are issued at a fixed rate. The frequency of the output signals is set to specified in the `<resample/>` tag in the `rate` attribute. The last value is always displayed, no interpolation takes place. The names of the new signals are generated by prepending a string to the old signal names, which is specified in the `name` attribute.

Furthermore, a timeout monitoring of the signals can be set up, by specifying a value greater than 0.0 in the `<resample/>` tag in the `timeout` attribute.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>ResampleEvaluator</evaluator>
  <description>Test</description>
  <instance><signal name="s1"/></instance>
  <content>
    <resample rate="0.5" name="rs_" timeout="3.0"/>
  </content>
</template>
```

Listing 212: File resample.tem

10.26.3 ResampleMeanEvaluator

The `ResampleMeanEvaluator` is used to resample a signal. Thereby the frequency of the output signals specified in the `<resample/>` tag in the `rate` attribute. All signals that are specified in the instance, are resampled. The average value of the signal values received in a resampling interval is output. The names of the created new signals are formed by prefixing the old signal name with a string, which is specified in the `name` attribute.

In the given example, the input signals `s1` and `s2` are set to a frequency of of 1/3 Hz is output. The new signals are called `rs_s1` and `rs_s2`.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>ResampleEvaluator.ResampleMeanEvaluator</evaluator>
  <description>Test</description>
  <instance><signal name="s1"/><signal name="s2"/></instance>
  <content>
    <resample rate="0.33333333" name="rs_"/>
  </content>
</template>
```

Listing 213: File resample.tem

If the input signal fails, the last output signal is repeated!

This repetition in case of signal failure can be switched off with the gap="true" attribute.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>ResampleEvaluator.ResampleMeanEvaluator</evaluator>
  <description>Test</description>
  <instance><signal name="s1"/><signal name="s2"/></instance>
  <content>
    <resample rate="0.33333333" name="rs_" gap="true"/>
  </content>
</template>
```

Listing 214: File resample.tem

Furthermore, a timeout monitoring of the signals can be set up, by specifying a value greater than 0.0 in the <resample/> tag in the timeout attribute.

Simultaneous use of gap="true" and timeout>0 attributes is not allowed.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>ResampleEvaluator.ResampleMeanEvaluator</evaluator>
  <description>Test</description>
  <instance><signal name="s1"/></instance>
  <content>
    <resample rate="2" name="rs_" timeout="2"/>
  </content>
</template>
```

Listing 215: File resample.tem

10.27 SignalCleaningEvaluator

The SignalCleaningEvaluator is used to easily remove invalid values from signals. In contrast to the filter function with the virtual signals, the possibility is to be given here, Filter out signal values that are invalid by convention in many or all signals. For example, in many signal descriptions (e.g. DBC files) there are dedicated values in the Text representation of the signals that do not represent a real physical value, but an error, a default value or similar. With a SignalCleaningEvaluator, corresponding text representations can be (e.g. "Error") can be specified to remove the associated values from all signals. A suffix is included when configuring the evaluator. Each signal can then be saved under the signal name with appended suffix can be retrieved in its corresponding filtered form.

In the following example, the text values "Error" and "Undervoltage" are removed from the signals. The suffix is set to "_filtered". For example, in the case of an engine control unit, the speed can be filtered with "speed_filtered" in the analysis can be used to evaluate only real speed values. If the control unit does not provide a real measured value, no value is included in the analysis. With filterInitValues the

start values can also be filtered out, which are identified by the "GenSigStartValue" attribute for DBCs. However, caution is advised here, as this is only a starting value, which may still be a valid measured value under certain circumstances.

```
<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>SignalCleaningEvaluator</evaluator>
    <description short="test" />
    <content suffix="_filtered" filterInitValues="true" >
      <ignoreString>Fehler</ignoreString>
      <ignoreString>Unterspannung</ignoreString>
    </content>
  </template>
</templates>
```

Listing 216: File filter.tem

10.28 Signal processing

10.28.1 FourierEvaluator

The Fourier evaluator performs a blockwise Fourier transform of a signal. For the calculation a window function can be selected, with which each block to be transformed can be multiplied point by point before the transformation. This can reduce "blur effects" in the frequency domain. Supported are the Von-Hann function (`vonHann`) and the square wave function (no change of the signal - default), which are determined with the `window` parameter. A unitary normalization is used in the Fourier transform. The parameters `length` (block length in samples) and `period` (block spacing in samples) can be used to set the Position and length of the transformation blocks can be influenced. The `frequency` parameter can be used to specify the sampling frequency of the input signal. This is needed for a suitable axis labeling in the frequency domain. In principle, equidistant sampling of the input signal is assumed, which can be ensured by using the `ResampleEvaluator`. The block length must be equal to a power of two. Does `length` not correspond to a power of two, the block length is automatically set to the next higher power of two. The issue is in amount and phase. The corresponding values are written to the metadata database in both two tables (magnitude and phase), as well as output as signals. The signals are of the `DataArray` type. For further processing of these signals the `JavaEvaluator` can be used.

The following example shows the use of the evaluator. In this example, every 32. Sample position, a 64-point Fourier transform is performed. Thus, the transformation blocks overlap by half. The rectangle function is used as the window function. Each sample must correspond to one second.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>FourierEvaluator</evaluator>
  <description>fft</description>
  <instance>
    <signal name="s"/>
  </instance>
  <content window="rectangle" length="64" period="32" frequency="1">
  </content>
</template>
```

Listing 217: File test.tem

10.28.2 SlidingWindowFunctionEvaluator

The SlidingWindowFunctionEvaluator can calculate certain values from a signal over a time window. The window size and the output frequency can be defined independently. Furthermore still a timeout is defined. If the signal fails for a longer time than defined in the timeout, no values are output. Possible output values are: Mean (mean), median (median), minimum (min), maximum (max), standard deviation (sdev) and variance (var). A default value and result name must be defined for each input signal.

In the following example, the average value of signals s1 and s2 is output in signals s1_out and s2_out.

```
<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>SlidingWindowFunctionEvaluator</evaluator>
    <description short="slidingWindow" />
    <content stepSize="1.0" samplingPeriod="0.5" windowSize="2.0" timeout="1.0" type="mean" />
    <instance>
      <signal name="s1" default="0.0" />
      <result name="s1_out" />
    </instance>
    <instance>
      <signal name="s2" default="0.0" />
      <result name="s2_out" />
    </instance>
  </template>
</templates>
```

Listing 218: File test.tem

10.29 Statistics

10.29.1 ClassEvaluator

The ClassEvaluator is used to calculate histograms. The attribute type in the tag <mode/> of the template is used to specify which type of summation should be used. Optionally, the memEfficientMode attribute can be set in the <content/> tag. If this attribute is set, a counting mode is used that uses memory more efficiently, especially when using many dimensions. However, in normal cases the usual counting mode is faster.

A dwell time histogram is selected e.g. via the duration type. The value of the attribute delay in the <mode/> tag determines the signal downtime from which the counting should be interrupted. In the <signal/> tag the description of the axis can be modified. Here the axis is divided equidistantly from min=1 to max=10 with a bin width width=1.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>ClassEvaluator</evaluator>
  <description>Test</description>
  <instance><signal name="s1"/></instance>
  <content memEfficientMode="false">
    <mode type="duration" delay="200"/>
    <signal name="Temperatur" type="equidistant" min="1" max="10" width="1"/>
  </content>
</template>
```

Listing 219: Example

An entry histogram is selected via the countEnter type. An entry histogram counts

how many values fall into each bin. The start of the measurement is counted as entry only if the `countOnFirstValue` attribute is true. The rising edge of an enable signal is counted as an entry if the `countOnEnable` attribute is true.

The `<trigger/>` tag is used to write a periodic entry to the metadata database in the following example. Here, the attribute `mode` specifies that periodic transmission is to take place, with the frequency specified in the `period` attribute, here every 3 seconds. In the associated table row, all values that occurred before the trigger event are taken into account. A value that occurs at the same time like the trigger event counts towards the next interval.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
<evaluator>ClassEvaluator</evaluator>
<description>Test</description>
<instance><signal name="s1"/></instance>
<content>
<mode type="duration" delay="200"/>
<signal name="Temperatur" type="equidistant" min="1" max="10" width="1"/>
<trigger mode="period" freq="3"/>
</content>
</template>
```

Listing 220: Example

With the tag `<trigger/>` and the attribute `mode="signal"` a trigger signal can be used to initiate a storage event. In this case, an entry is always created in the metadata database, if the corresponding signal exceeds the threshold 0.5. The name of the trigger signal can be specified in the `name` attribute.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
<evaluator>ClassEvaluator</evaluator>
<description>Test</description>
<instance><signal name="s1"/></instance>
<content>
<mode type="duration" delay="200"/>
<signal name="Temperatur" type="equidistant" min="1" max="10" width="1"/>
<trigger mode="signal" name="s2"/>
</content>
</template>
```

Listing 221: Example

In a two-dimensional histogram, two input signals and corresponding two axis descriptions are used. The tag `<storeEveryFile/>` causes that in the case of several measurements in an archive, a database entry is created for each measurement. If this tag is missing, only one database entry is generated per measurement.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
<evaluator>ClassEvaluator</evaluator>
<description>Test</description>
<instance>
<signal name="s1"/>
<signal name="s2"/>
</instance>
<content>
<storeEveryFile/>
<mode type="duration" delay="200"/>
<signal name="Temperatur1" type="equidistant" min="1" max="5" width="1"/>
<signal name="Temperatur2" type="equidistant" min="1" max="5" width="1"/>
</content>
</template>

```

Listing 222: Example

If you do not want to use the automatically generated table headers, the column names can be set using the tag `<columnNames/>`.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
<evaluator>ClassEvaluator</evaluator>
<description>Test</description>
<instance><signal name="s1"/></instance>
<content>
<mode type="countEnter" delay="200"/>
<signal name="Temperatur" type="equidistant" min="0" max="1"/>
<columnNames>
<column name="aus"/>
<column name="an"/>
</columnNames>
</content>
</template>

```

Listing 223: Example

A special case is the histogram of the dwell times themselves. With this type of histogram, the times for which the signal value is in a bin are not simply summed up, instead they are themselves bined again. This means that a new axis is added, which is used for the dwell time. If, for example, the signal to be classified remains in class 1 for 3 seconds, then the entry in the results table at which the value 1 and the duration 3 are located is incremented by one.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
<evaluator>ClassEvaluator</evaluator>
<description>Test</description>
<instance><signal name="s1"/></instance>
<content>
<mode type="durationClass" delay="200">
<signal name="Zeit" type="equidistant" min="0" max="5" width="1"/>
</mode>
<signal name="Temperatur" type="equidistant" min="1" max="5" width="1"/>
</content>
</template>

```

Listing 224: Example

If a timeout occurs in a signal that is analyzed with a dwell time histogram, then a double counting will be caused by this Timeout. If you e.g. set up the delay time to 5 sec and a signal takes the value 1 for 90 seconds, but there is a gap in the middle of the signal for 10 sec long, then you will end up with increasing the bin for 40 sec by two. The Timeout is thus treated as if the signal had changed the bin.

If instead of the type "durationClass", the type "durationClassTime" is used for the histogram of the dwell times, the length of stay is counted, instead of the frequency of stays in the corresponding bin. For example, if the signal to be analyzed stays for 3 seconds in class 1, the result of the 2-dimensional class, in which the value 1 and the duration 3 is located, is incremented by 3.

Peak Counting

With the `peakCounting` mode, a histogram according to DIN 45667 (peak method II) can be performed. The binning can be done in the usual way (as with other classifiers). Counted are all maxima that lie above a certain limit and all minima that lie below the limit. If nothing else is specified, this limit is 0. The following example performs a peak counting histogram with an equidistant bin division with bin width 1. With the input signal from figure 8 the result is for example

class (smaller)	-5	-4	-3	-2	-1	0	1	2	3	4	5	...
Counter	0	1	1	0	0	0	0	0	2	2	0	

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>ClassEvaluator</evaluator>
  <description>Test</description>
  <instance><signal name="s1"/></instance>
  <content>
    <mode type="count"/>
    <signal name="Temperatur" mode="peakCounting" type="equidistant" min="-5" max="5" width=
  </content>
</template>;
```

Listing 225: Example

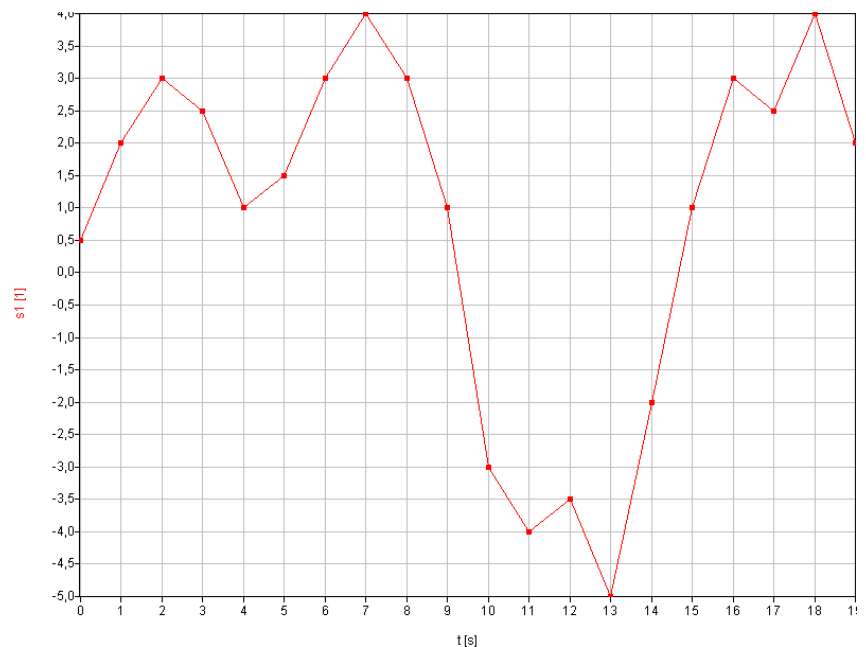


Figure 8: Example signal for peakCounting

In addition, a hysteresis can be specified via the `hyst` parameter. Oscillations whose amplitudes are smaller than this value are thus ignored. In our example, with a hysteresis of 1, the result is as follows.

class (smaller)	-5	-4	-3	-2	-1	0	1	2	3	4	5	...
Counter	0	1	0	0	0	0	0	0	1	2	0	

```
<?xml version=\ "1.0\ " encoding=\ "UTF-8\ "?>
<template>
  <evaluator>ClassEvaluator</evaluator>
  <description>Test</description>
  <instance><signal name=\ "s1\ "/></instance>
  <content>
    <mode type="count"/>
    <signal name="Temperatur" mode="peakCounting" type="equidistant" hyst="1.0" min="-5" max=
  </content>
</template>);
```

Listing 226: Example

The parameter zero can be used to adjust the above mentioned limit. For example, the above example with a “zero level” of 2 results in

class (smaller)	-5	-4	-3	-2	-1	0	1	2	3	4	5	...
Counter	0	1	1	0	0	1	0	0	2	2	0	

```
<?xml version=\ "1.0\ " encoding=\ "UTF-8\ "?>
<template>
  <evaluator>ClassEvaluator</evaluator>
  <description>Test</description>
  <instance><signal name=\ "s1\ "/></instance>
  <content>
    <mode type="count"/>
    <signal name="Temperatur" mode="peakCounting" type="equidistant" zero="2.0" min="-5" max=
  </content>
</template>);
```

Listing 227: Example

Peak Through

The `peakThrough` mode can be used to perform histogram using the peak through counting method. The histogram of the bins can be done in the usual way (as with other histograms). The method is a two-parametric method, in which in a matrix the half oscillations are counted from one extreme value to the next extreme value. The first dimension corresponds to the start bin, the second to the target bin. The best way to explain this is with an example. The following example shows a peak through histogram with equidistant bins of width 1 in a range from 1 to 10. Thereby a hysteresis (`hyst`) of 0.99 is used to neglect oscillations whose amplitude is smaller than a bin width. With the input signal from figure 9 the result is for example

of class	1.5	1.5	2.5	2.5	2.5	3.5	3.5	3.5	4.5	4.5		
by class	5.5	6.5	3.5	5.5	6.5	7.5	2.5	7.5	8.5	9.5	2.5	7.5
Number	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	1.0	2.0	
of class	4.5	5.5	5.5	6.5	6.5	7.5	7.5	8.5	9.5	9.5		
by class	9.5	1.5	3.5	7.5	1.5	3.5	2.5	3.5	4.5	3.5	4.5	5.5
Number	1.0	1.0	1.0	1.0	1.0	2.0	1.0	1.0	2.0	1.0		

In matrix representation the whole thing looks as follows.

class	...	0.5	1.5	2.5	3.5	4.5	5.5	6.5	7.5	8.5	0.9
...	0	0	0	0	0	0	0	0	0	0	0
0.5	0	0	0	0	0	1	1	0	0	0	0
1.5	0	0	0	1	0	1	1	1	0	0	0
2.5	0	0	1	0	0	0	0	1	1	2	0
3.5	0	0	1	0	0	0	0	2	0	1	0
4.5	0	1	0	1	0	0	0	1	0	0	0
5.5	0	1	0	1	0	0	0	0	0	0	0
6.5	0	0	2	1	1	0	0	0	0	0	0
7.5	0	0	0	1	0	0	0	0	0	0	0
8.5	0	0	0	0	2	1	0	0	0	0	0
9.5	0	0	0	0	0	0	0	0	0	0	0

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>ClassEvaluator</evaluator>
  <description>Test</description>
  <instance><signal name="s1"/></instance>
  <content>
    <mode type="count"/>
    <signal name="Temperatur" mode = "peakThrough" hyst="0.99"
      type="equidistant" min="1" max="10" width="1"/>
  </content>
</template>

```

Listing 228: File test.tem

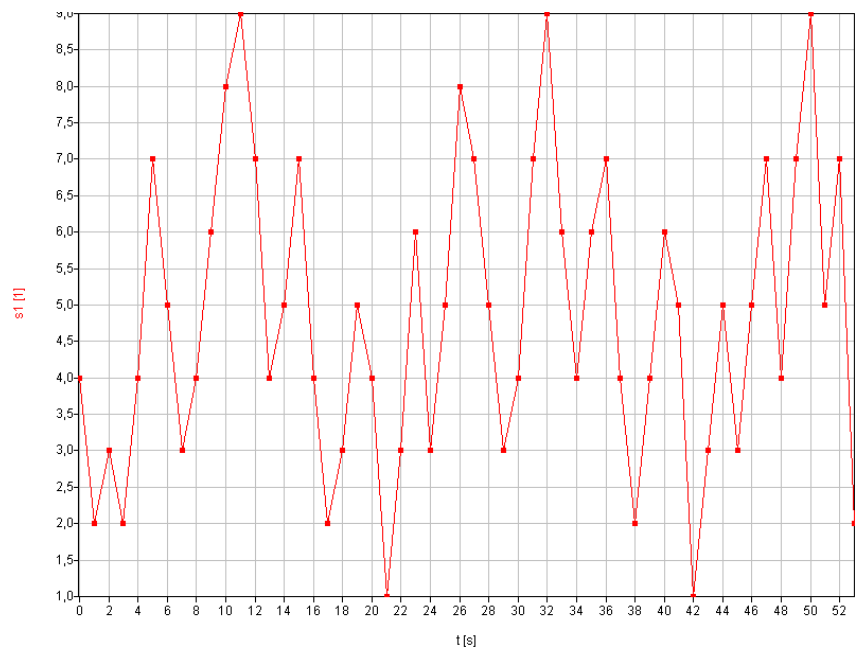


Figure 9: Example signal for peakThrough

Rainflow

The rainflow mode can be used to perform histogram according to the Rainflow method. The histogram of the bins can be done in the usual way (as with other histograms). In the following example, a Rainflow histogram with equidistant bins of width 1 in a range from 1 to 10 is performed. Thereby a hysteresis (*hyst*) of 0.99 is used to neglect oscillations whose amplitude is smaller than a bin width. With the input signal from figure 10 the result is for example

of class	1.5	2.5	2.5	4.5	5.5	6.5	7.5	7.5	8.5
by class	9.5	3.5	5.5	6.5	7.5	3.5	3.5	4.5	3.5
Number	1.0	1.0	1.0	1.0	2.0	1.0	1.0	1.0	1.0

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>ClassEvaluator</evaluator>
  <description>Test</description>
  <instance><signal name="s1"/></instance>
  <content>
    <mode type="count"/>
    <signal name="Temperatur" mode = "rainflow" hyst="0.99"
      type="equidistant" min="1" max="10" width="1"/>
  </content>
</template>

```

Listing 229: File test.tem

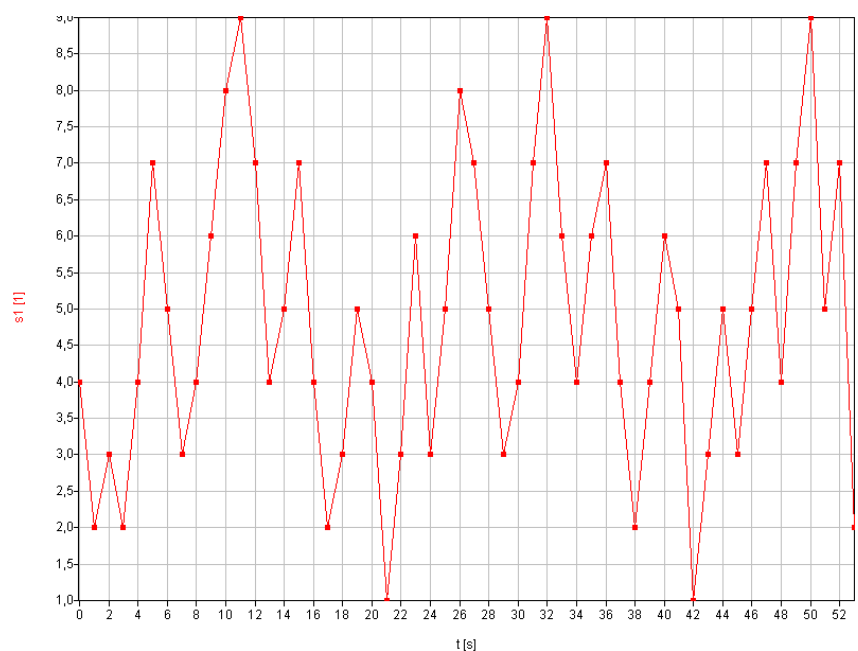


Figure 10: Example signal for Rainflow

In some cases, however, unfinished half oscillations are also of interest. In this case, one speaks of the residual. The residual can be output with the parameter `useResidual="true"`.

```

<?xml version="1.0" encoding="UTF-8"?>
<signal name="Temperatur" mode="rainflow" hyst="0.99" useResidual="true" type="equidistant" m

```

Listing 230: Example

Usually AMeDA processes the residuals in such a way that as far as possible completed oscillations are counted and added to the result. For this purpose, the residual is run backwards according to the Rainflow method. With `residualMode="lth"` can be set to handle the residual in this way, as Matlab does in the WaFo toolbox. The residual is then traversed forward, with each pair consisting of minimum and maximum is counted as oscillation.

Range Counting

The `rangeCounting` mode can be used to perform histogram according to the

DIN45667 clamping method. The histogram of the bins can be done in the usual way (as with other histograms). The ranges (differences) between two adjacent extreme values are counted. The following example shows a range counting histogram with equidistant bins of width 1 in a range from -8 to 8. With the input signal from figure 11 the result is for example

class (smaller)	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	...
Counter	0	1	0	0	0	0	2	1	1	1	1	0	0	0	1			

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>ClassEvaluator</evaluator>
  <description>Test</description>
  <instance><signal name="s1"/></instance>
  <content>
    <mode type="count"/>
    <signal name="Temperatur" mode = "rangeCounting" type="equidistant"
min="-8.0" max="8.0" width="1"/>
  </content>
</template>
```

Listing 231: File test.tem

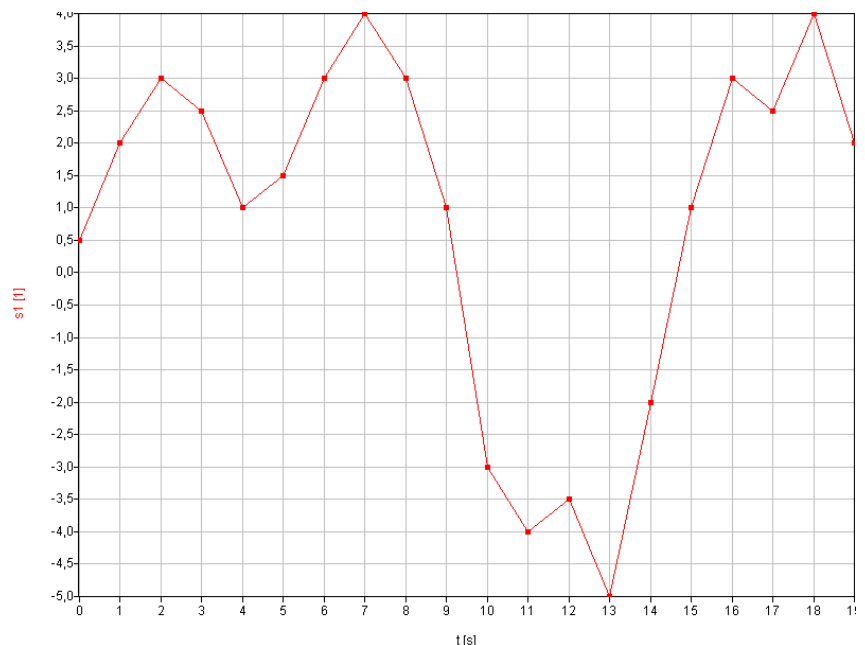


Figure 11: Example signal for rangeCounting

Zero Crossing Peak Counting

The zeroCrossingPeakCounting mode can be used to perform a histogram according to the peak method I according to DIN45667. The histogram of the bins can be done in the usual way (as with other histograms). The largest amplitudes between two passes through a reference line are counted. The height of the reference line can be determined by the zero parameter and is 0 in the absence of this parameter. The following example shows a Zero Crossing Peak Counting histogram with equidistant bins of width 1 in a range from -5 to 5. With the input signal from figure 12 the result is

class	-5	-4	-3	-2	-1	0	1	2	3	4	5	...
for example (smaller)	0	1	0	0	0	0	0	0	0	1	0	
Counter	0	1	0	0	0	0	0	0	0	1	0	

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>ClassEvaluator</evaluator>
  <description>Test</description>
  <instance><signal name="s1"/></instance>
  <content>
    <mode type="count"/>
    <signal name="Temperatur" mode = "zeroCrossingPeakCounting"
      type="equidistant" min="-5" max="5" width="1"/>
  </content>
</template>

```

Listing 232: File test.tem

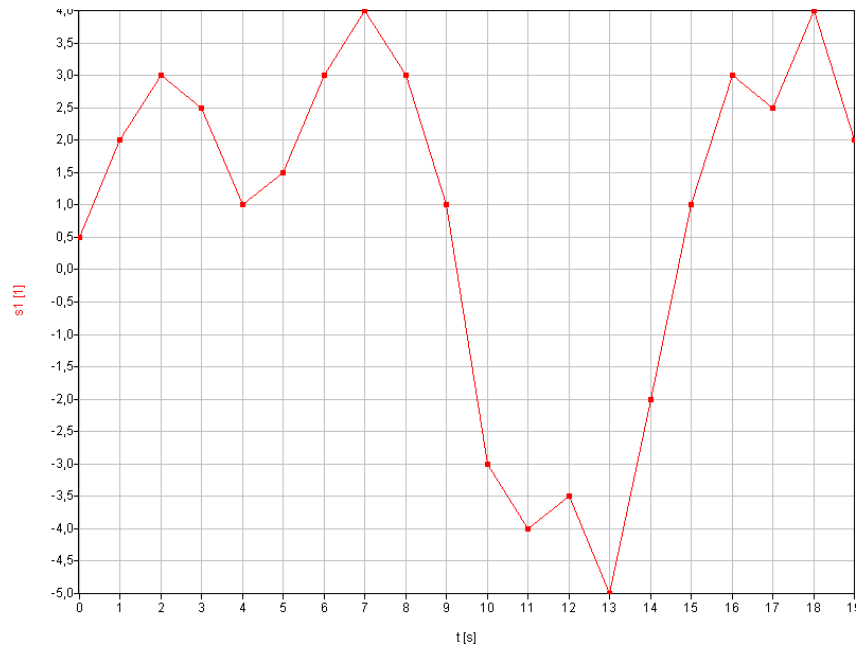


Figure 12: Example signal for zeroCrossingPeakCounting

10.29.2 CounterBZEvaluator

The CounterBZEvaluator is used when the operating time of a component is to be determined on the basis of the transmission of can messages. The operating time of a complete Can channel can also be determined.

In this example, the operating time of the device is measured, which is connected to Can 1, specified in the tag `<can/>` in the Attribute `num` on id 1, specified in tag `<message/>` in attribute `id`. In tag `<delay/>` a TimeOut time in seconds is specified in the `time` attribute, here one second. Missing the message for a longer time than the TimeOut time, it is assumed to be disabled, and the operating time is not more increased until the message is sent again. Optionally, in tag `<canSignal/>` in attribute `Name` the input signal can be defined. In the absence of a definition, `canMessage` is used.

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<template>
  <evaluator>CounterBZEvaluator</evaluator>
  <description>TestTemplate</description>
  <content>
    <delay time="1"/>
    <can num="1">
      <message id="1"/>
    </can>
    <canSignal name="canMessage" />
  </content>
</template>

```

Listing 233: File test.tem

Several tags `<message/>` can be given. In this case, the counter will run when at least one message is available.

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<template>
  <evaluator>CounterBZEvaluator</evaluator>
  <description>TestTemplate</description>
  <content>
    <delay time="1"/>
    <can num="1">
      <message id="1"/>
      <message id="2"/>
    </can>
  </content>
</template>

```

Listing 234: File test.tem

If the tag `<message/>` is omitted, the operating time of the complete CAN is determined, i.e. the time during which messages were sent.

Multiple tags `<can/>` can be specified if the sum operating time of multiple CAN channels is to be determined. Tags `<message/>` can be specified again for each tag `<can/>`.

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<template>
  <evaluator>CounterBZEvaluator</evaluator>
  <description>TestTemplate</description>
  <content>
    <delay time="1"/>
    <can num="1"/>
    <can num="2"/>
  </content>
</template>

```

Listing 235: File test.tem

If a higher time resolution than one counter value per measurement is required, the tag `<trigger/>` can be used to induce that the values are stored more often. In the following example, the totals are saved periodically every 5 seconds.

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<template>
  <evaluator>CounterBZEvaluator</evaluator>
  <description>TestTemplate</description>
  <content>
    <delay time="1"/>
    <can num="1">
      <message id="1"/>
      <message id="2"/>
    </can>
    <trigger mode="period" freq="5"/>  </content>
  </template>

```

Listing 236: File test.tem

A signal can also be used to trigger a save operation. In the following example a value is stored when the signal s2 exceeds the value 0.5.

```

...
<trigger mode="signal" name="s2"/>
...

```

Listing 237: - Signal based trigger

10.29.3 CounterEvaluator

The CounterEvaluator is used when events are to be counted. An interval can be specified in which the signal value should move. It is determined how many times the signal has entered the interval, how many times it has entered the interval left, and how much time it spent in the interval. It is therefore a kind of combined histogram with only one bin.

For simplicity, the CounterEvaluator is conditioned in the instance. In the instance a description can be specified in the <description/> tag. This is used, among other things, to specify the table name in the MetaDatabase. If min and max are not specified, min=0 and max=1 are assumed.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>CounterEvaluator</evaluator>
  <description>Test</description>
  <instance>
    <description name="CounterTest"/>
    <signal name="s1"/>
  </instance>
</template>

```

Listing 238: Example

Two signals can also be specified in one instance. In this case not sums the time the first signal is in the specified interval, but the change of the second signal is used. If in the second signal e.g. the Mileage is specified, the number of kilometers traveled is summed up while the value of the first signal was in the specified interval.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>CounterEvaluator</evaluator>
  <description>Test</description>
  <instance>
    <description name="CounterTest"/>
    <signal name="s1"/>
    <signal name="s2"/>
  </instance>
</template>

```

Listing 239: Example

If a higher temporal resolution than one counter value per measurement is required, you can use with the `<trigger/>` tag to cause the values to be saved more often. In the following example, the totals are saved periodically every 3 seconds.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>CounterEvaluator</evaluator>
  <description>Test</description>
  <instance>
    <description name="CounterTest"/>
    <signal name="s1"/>
    <trigger mode="period" freq="3"/>
  </instance>
</template>

```

Listing 240: Example

A signal can also be used to trigger a save operation. In the following Example a value is stored when the signal `s2` exceeds the value 0.5.

```

...
<trigger mode="signal" name="s2"/>
...

```

Listing 241: Example

10.29.4 GenericClassEvaluator

This evaluator can be used to perform special evaluations similar to a histogram. The first signals that are specified are used for bin formation and the further Signals can be used for any calculations. The result, which was finally presented to the bin is assigned is always a double value, as in normal histogram. A possible use case would be, for example, the histogram of the engine speed and the determination of the maximum exhaust gas temperatures for each bin.

The way it works is as follows: In the example, in the `<instance/>` tag, the attribute `name` the two signals `s1` and `s2` are specified. However, in the `<classes/>` tag, only one signal with a Classifying axis provided. So also only the first signal is used to span the classes. In the `<classes/>` tag, you must always specify at least one axis less than signals in the instance must be specified. The function of the remaining tags under the `<content/>` tag corresponds for the most part to the function of the corresponding tags in the `FunctionEvaluator`. The program code specified in the `<first/>` tag is executed exactly once, when the class belonging to the signal value is entered for the first time. In figure 13 is exemplary the sequence of calls of the individual code blocks with two classes is shown. The tags used in the `<function/>` and `<finally/>` specified code blocks have a return value that can be set by setting of the double variable named `result` can be used on any double value. Thereby to note that only return values that are not equal to `Double.NaN` will result in entries in the MetaDB.

In the GenericClassEvaluator, an instance is created for each defined class of the global variable specified in the <global/> tag. In the example, eleven classes are defined, and therefore there are also eleven max variables. During the evaluation, the functions are compared with the Class are called.

The code specified in the <finally/> tag finally returns the result of the respective class back. So the example generates a kind of histogram which is used for each by s1 defined class contains the respective maximum of the signal s2. (Siehe auch 10.23.1)

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>ClassEvaluator.generic.GenericClassEvaluator</evaluator>
  <description>Test</description>
  <instance>
    <signal name="s1"/>
    <signal name="s2"/>
  </instance>
  <content>
    <classes>
      <signal name="s1" type="equidistant" min="1" max="10" width="1"/>
    </classes>
    <parameter>
      <par name="s1"/>
      <par name="s2"/>
    </parameter>
    <global>
      double max;
    </global>
    <first>
      max=s2;
    </first>
    <function>
      if (s2>max) max=s2;
    </function>
    <finally>
      result=max;
    </finally>
  </content>
</template>
```

Listing 242: File test.tem

In the following example, the mean value of a signal in the respective class is to be determined. This is only possible if the time of entering and leaving a bin is determined. To determine these times, the <enter/> and <leave/> tags can be used. The one in the day <enter/> code is called when the bin is entered, and the code specified in the <leave/> tag is called Code accordingly when the bin is exited. At the time of the last measured value of the measurement file, the value stored in the Tag <finally/> specified code is called for each bin that was previously entered.

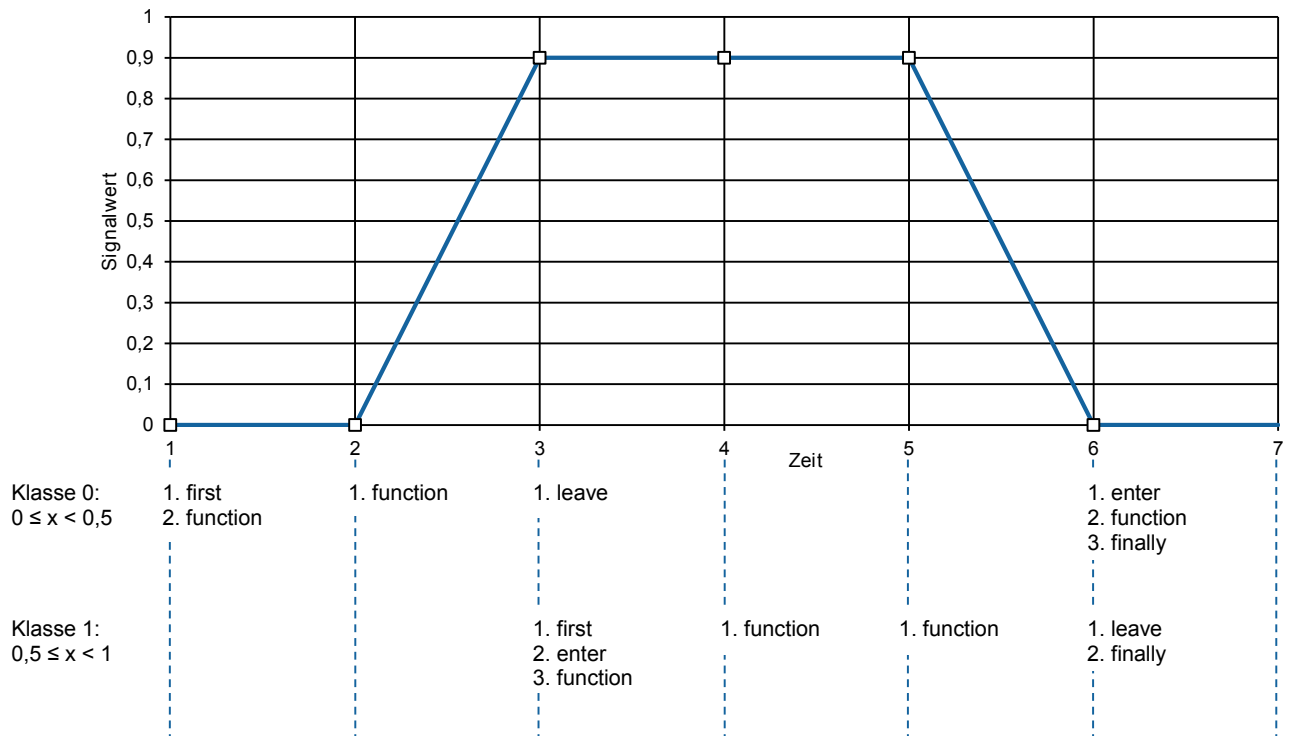


Figure 13: Example of calls to the code blocks with two classes

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>ClassEvaluator.generic.GenericClassEvaluator</evaluator>
  <description>Test</description>
  <instance>
    <signal name="s1"/>
    <signal name="s2"/>
  </instance>
  <content>
    <classes>
      <signal name="s1" type="equidistant" min="1" max="10" width="1"/>
    </classes>
    <parameter>
      <par name="s1"/>
      <par name="value"/>
    </parameter>
    <global>
      double sumValue;
      double sumTime;
      double lastValue;
      double lastTime;
    </global>
    <enter>
      lastValue=value;
      lastTime=time;
    </enter>
    <function>
      double dt=time-lastTime;
      sumValue=sumValue+dt*lastValue;
      sumTime=sumTime+dt;
      lastTime=time;
      lastValue=value;
    </function>
    <leave>
      double dt=time-lastTime;
      sumValue=sumValue+dt*lastValue;
      sumTime=sumTime+dt;
    </leave>
    <finally>
      if (sumTime!=0) result=sumValue/sumTime;
    </finally>
  </content>
</template>
```

When calculating the mean value, it makes sense to not only file the mean value, but to also the value sum and the time sum. This is important when classifying results of several measurements are to be aggregated into one result. In the following example, by calling the method `addMetaDBValue` additional values are entered into the table. Care must be taken to ensure that all bins are generate the same values. It must not be the case that some classes generate values and others do not. Although the method `addMetaDBValue` can be called from any block (`<enter/>`, `<function/>`, `<leave/>`), this should only be done in the `<finally/>` block happen.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>ClassEvaluator.generic.GenericClassEvaluator</evaluator>
  <description>Test</description>
  <instance>
    <signal name="s1"/>
    <signal name="s2"/>
  </instance>
  <content>
    <classes>
      <signal name="s1" type="equidistant" min="1" max="10" width="1"/>
    </classes>
    <parameter>
      <par name="s1"/>
      <par name="value"/>
    </parameter>
    <global>
      double sumValue;
      double sumTime;
      double lastValue;
      double lastTime;
    </global>
    <enter>
      lastValue=value;
      lastTime=time;
    </enter>
    <function>
      double dt=time-lastTime;
      sumValue=sumValue+dt*lastValue;
      sumTime=sumTime+dt;
      lastTime=time;
      lastValue=value;
    </function>
    <leave>
      double dt=time-lastTime;
      sumValue=sumValue+dt*lastValue;
      sumTime=sumTime+dt;
    </leave>
    <finally>
      if (sumTime!=0) result=sumValue/sumTime;
      addMetaDBValue("sumValue",sumValue);
      addMetaDBValue("sumTime",sumTime);
    </finally>
  </content>
</template>
```

Listing 244: File test.tem

Two-dimensional histograms are also possible. The following example shows again the determination of a Maximums. Here the signals `s1` and `s2` span the bins, and the maximum of the Signal `s3` is determined.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>ClassEvaluator.generic.GenericClassEvaluator</evaluator>
  <description>Test</description>
  <instance>
    <signal name="s1"/>
    <signal name="s2"/>
    <signal name="s3"/>
  </instance>
  <content>
    <classes>
      <signal name="s1" type="equidistant" min="-1" max="1" width="0.2"/>
      <signal name="s2" type="equidistant" min="-1" max="1" width="0.2"/>
    </classes>
    <parameter>
      <par name="s1"/>
      <par name="s2"/>
      <par name="s3"/>
    </parameter>
    <global>
      double max;
    </global>
    <first>
      max=s3;
    </first>
    <function>
      if (s3>max) max=s3;
    </function>
    <finally>
      result=max;
    </finally>
  </content>
</template>

```

Listing 245: File test.tem

Analogous to the ClassEvaluator, the writing of intermediate results to the meta-database can be done periodically or by a signal can be triggered. When a trigger is fired, the code specified in the `<finally/>` tag for each class, previously entered is called at the time of the trigger.

The behavior after a trigger corresponds to that of the start of the measurement. So it will be again the one in the tag `<first/>` specified code is called once when entering each class. In figure 14 is the behavior in the Trigger case shown as an example. (Siehe auch 10.29.1)

For calls in code blocks where time is used (variable name `time`), it should be noted that that signals can fail. In the previously presented example for calculating the mean values in a timeout time should therefore be specified for the respective classes. This time specifies after which temporal Difference between two successive values of a signal to be left class. The Failure to take into account a signal failure would, under certain circumstances, strongly influence the result of the calculation of the mean value.

In the following configuration snippet, the timeout time is specified in the `timeout` attribute of the `<content/>` tag set to one second. If at least one second elapses between two values of a signal, the value entered in tag `<leave/>` specified code of the current classes is called. The same applies in the case that between a measured value and a directly following trigger or the end of the measurement (without intermediate occurrence of a further Measured value of this signal) at least as much time has passed as specified in the `timeout` attribute.

In contrast to triggering, after a timeout occurs, the code specified in the `<first/>` tag will be executed only if is called if this class was not entered before the timeout. In figure 15 is the behavior in the event of a timeout is shown as an example.

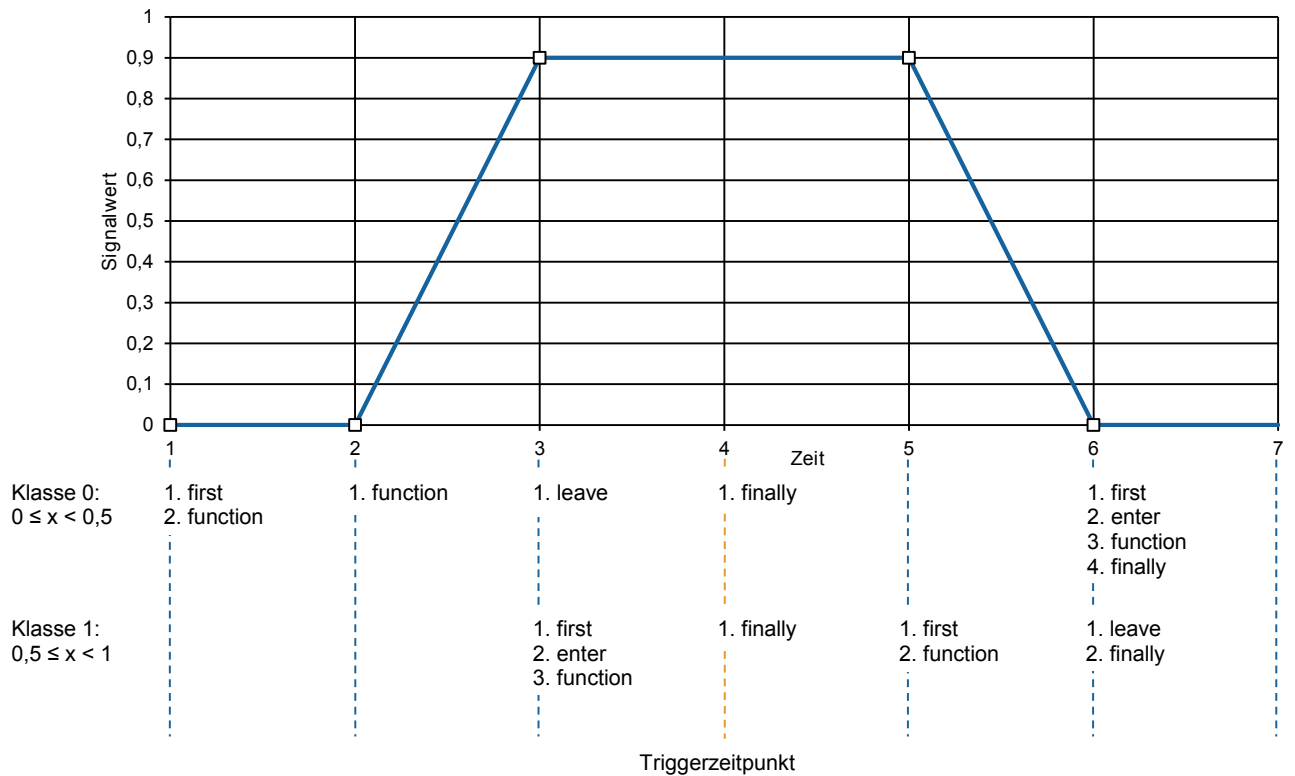


Figure 14: Example of calls to code blocks in trigger case

```
<content name=\"result\" unit=\"u\" timeout=\"1.0\" >
...
</content>
```

Listing 246: Specifying a timeout period

The `wasReceived` function can be used to check, whether a signal was received at the time currently being processed. The function is passed an index by which the signal can be determined. Here, index 1 belongs to the first signal from the `<instance/>` Tag, the index 2 to the second, and so on.

```
<?xml version=\"1.0\" encoding=\"UTF-8\"?>
<function>if (wasReceived(2)) { ... }</function>
```

Listing 247: Calling the `wasReceived` function

Summary

The following table summarizes the conditions of the calls to the code blocks.

code block	condition of the call	name of the result variable
first	first entry into class after start of measurement and first entry into class after trigger time	
enter	change a class (not measurement start)	
function	value within a class	result
leave	change of a class (not end of measurement) and time- out time	
finally	last measured value of the measurement and trigger time	result

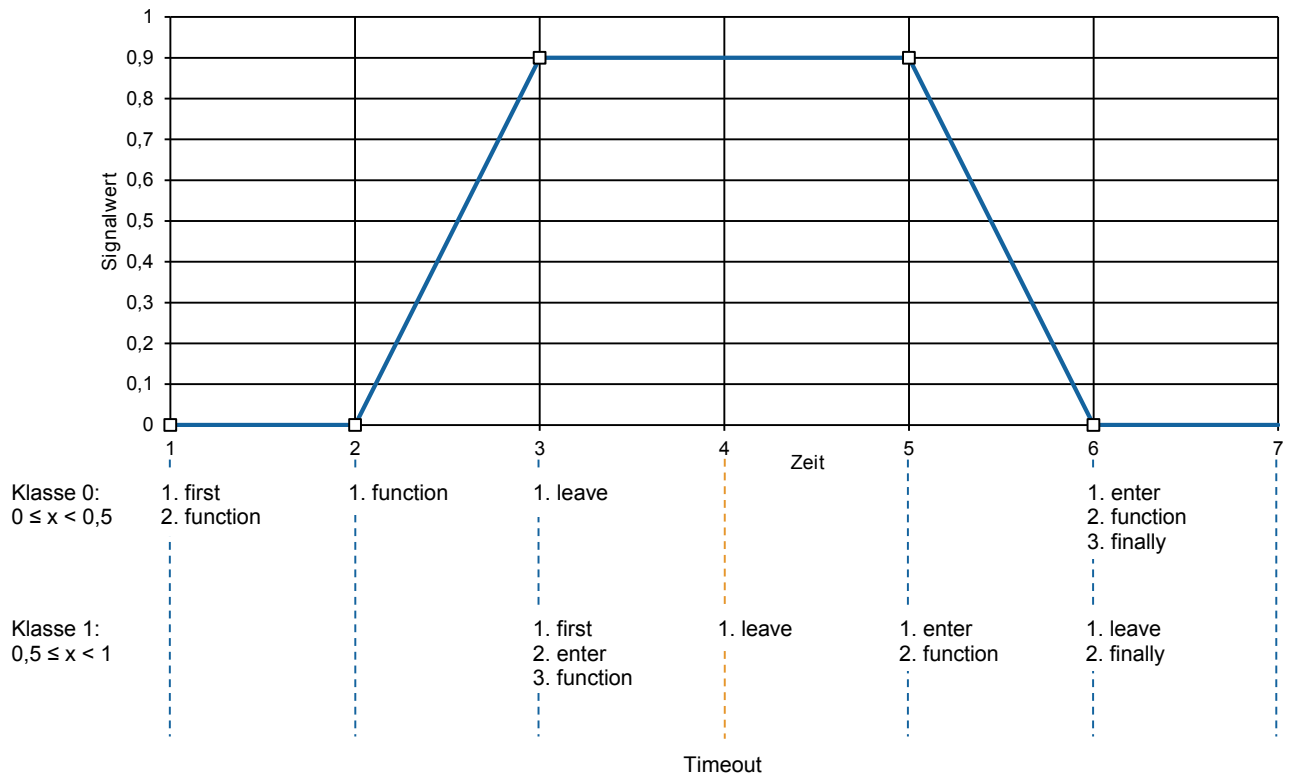


Figure 15: Example of calls to the code blocks in timeout case

10.29.5 QuantileEvaluator

The QuantileEvaluator calculates quantiles of any order. If an output signal is determined via the signalName attribute, a signal is output which shows the corresponding quantile of the input signal at any time up to this point. If a table is specified via the tableName attribute, a table in the metadata database is populated, whose only column q contains the quantile. The quantile is determined by taking the value from a sorted list that is at the position (order * number of signal values). In case of doubt, the next larger value is taken. Attention: For a meaningful calculation of a quantile, the input signal should be sampled equidistantly. If necessary, this can be achieved by the ResampleEvaluator.

In this example, the quantile of order 0.5 (the median) is calculated for signal s1 and both as signal, as well as a table entry in the metadata database.

```
<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>QuantileEvaluator</evaluator>
    <description short="q" />
    <instance>
      <signal name="s1" />
    </instance>
    <content>
      <parameters order="0.5" signalName="q" tableName="Quantil" />
    </content>
  </template>
</templates>
```

Listing 248: File Quantile.tem

The attribute enableSignal can be used to select a signal that can temporarily switch the calculation on and off. Only those signal sections are then taken into account where the enable signal is greater than or equal to 0.5.

```

<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>QuantileEvaluator</evaluator>
    <description short="q" />
    <instance>
      <signal name="s1" />
    </instance>
    <content>
      <parameters order="0.5" signalName="q" tableName="Quantil" enableSignal="e" />
    </content>
  </template>
</templates>

```

Listing 249: File QuantileEnable.tem

The attribute `triggerSignal` can also be used to select a signal, which triggers writing to the metadata database. After the values are written to the database, the calculation is reset. In this way, a block-by-block calculation can be achieved.

```

<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>QuantileEvaluator</evaluator>
    <description short="q" />
    <instance>
      <signal name="s1" />
    </instance>
    <content>
      <parameters order="0.5" triggerSignal="tr" signalName="q" tableName="Quantil" />
    </content>
  </template>
</templates>

```

Listing 250: File QuantileTrigger.tem

The `StatisticsEvaluator` calculates different statistical quantities (mean, standard deviation and variance). The parameters `mean`, `sdev` and `var` are used to name the corresponding output signals. The output signals are to be interpreted in such a way that they always represent the mean value, the standard deviation or contain the variance up to this point. If a table is specified via the `tableName` attribute, a table in the metadata database is populated, which contains the corresponding quantities (calculated over the entire measurement). In addition, a signal can be specified via the `triggerSignal` attribute, which triggers a write to the metadata database whenever it has a value greater than 0.5. Also, each time triggering occurs, the calculation of the statistical quantities is reset. In this way, a blockwise calculation can be achieved. In addition to the trigger signal, an enable signal (`enableSignal`) can be used for this, that certain signal values are ignored in the calculation.

In the following example, all three quantities are calculated for the signal "s1". A write to the metadata database and reset of the calculation is triggered by the signal "tr". All signal values where the signal "e" has a value less than 0.5 are ignored. The results are output as signals "mean", "sdev" and "var" and written to the metadata database table "stats" on each trigger.

```

<?xml version="1.0" encoding="UTF-8"?>
<templates>
  <template>
    <evaluator>StatisticsEvaluator</evaluator>
    <description short="q" />
    <instance>
      <signal name="s1" />
    </instance>
    <content>
      <parameters triggerSignal="tr" enableSignal="e" mean="mean" sdev="sdev"
        var="var" tableName="stats" />
    </content>
  </template>
</templates>

```

Listing 251: File Statistics.tem

11 Virtual signals

11.1 Expressions

Virtual signals can be defined by expressions. This is the easiest way to perform calculations on signals to perform.

In each template, virtual signals can be created in the `<virtualSignals/>` tag. These signals have global validity, i.e. they can also be used in any other template. In the attribute `out` is used the name of the generated signal is specified, and in the attribute `expression` the expression to be calculated. Any signal names can be used in this expression. The following operators are available for the definition of Expressions are available, where the execution order of the operators can be defined via brackets (and).

Operator	Type	Meaning
+	Arithmetic operator	Addition
-	Arithmetic operator	Subtraction
*	Arithmetic operator	Multiplication
/	Arithmetic operator	Division
not, !	binary operator	logical NOT
or,	binary operator	logical OR
and, &	binary operator	logical AND
xor, ^	binary operator	logical XOR
bitOr	binary operator	bitwise OR
bitAnd	binary operator	bitwise AND
=	comparison operator	check for equality
!=	comparison operator	check for inequality
greater, >	comparison operator	check for greater
greaterEqual, >=	comparison operator	check for greater-or-equal
lower, <	comparison operator	check for lower
lowerEqual, <=	comparison operator	check for less-than-or-equal
pi	constant	number π
NaN	Reserved Name	No Signal Assignment
contains	comparison operator	Checks whether a pattern (defined by RegEx) is contained in a string signal value.

In the example, the signal `s1` is sent with `Two` and provided as a signal `out`.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <signal out="out" expression="s1*s2"/>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>
```

Listing 252: File signal.tem

If more than one signal is used, the output signal is updated each time one of the input signals is changed. In the example, for each input of one of the signals *s1* and *s2* a new value of the signal *out* generated.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <signal out="out" expression="s1*s2"/>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>
```

Listing 253: File signal.tem

In some cases, it may be desirable to omit this update for individual signals. This can be achieved with the prefix "&". It should be noted that when directly processing the XML files a simple "&" cannot be used but must be replaced by "&". So a simple *s1*s2* becomes a *s1*&s2* in the XML file.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <signal out="out" expression="s1*&s2"/>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>
```

Listing 254: File signal.tem

In some databases there are signal names that are misinterpreted because they contain special characters that are would be interpreted as mathematical symbols. So a signal analog voltage cannot be used here, since it would be interpreted as operation Analog minus Voltage. To prevent this must such signal names are enclosed in curly brackets.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <signal out="out" expression="{Analog - Spannung}*2"/>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>
```

Listing 255: File signal.tem

A whole range of functions can be used. These are *sin*, *cos*, *tan*, *sign*, *abs*, *ln*,

log, exp, sqrt and round. For the trigonometric functions, the angle is given in radians.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <signal out="o1" expression="sin(s1)"/>
    <signal out="o2" expression="cos(s1)"/>
    <signal out="o3" expression="tan(s1)"/>
    <signal out="o4" expression="sign(s1)"/>
    <signal out="o5" expression="abs(s1)"/>
    <signal out="o6" expression="exp(s1)"/>
    <signal out="o7" expression="sqrt(abs(s1))"/>
    <signal out="o8" expression="ln(abs(s1))"/>
    <signal out="o9" expression="log(abs(s1))"/>
    <signal out="o10" expression="round(s1)"/>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>
```

Listing 256: File signal.tem

Highpass and lowpass filters can be used (lowpass, highpass). These are First order filters, whose Implementation takes into account the usually non-equidistant sampling of the signals. The second parameter is the frequency in Hz at which the signal amplitude has dropped to half.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <signal out="o1" expression="lowpass(s1,1)"/>
    <signal out="o2" expression="highpass(s1,1)"/>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>
```

Listing 257: File signal.tem

The ite function is special. It corresponds to an If-Then-Else and is similar in implementation to the “?:” operator from C or Java. This function has three parameters: If the first parameter is true (>0), then the second parameter is evaluated and returned, if not, the third parameter is evaluated and returned.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <signal out="out" expression="ite(s1>0,s2,s3)"/>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>
```

Listing 258: File signal.tem

The constant NaN can be used. This stands for “NotANumber”. Will this value is returned with an expression, this corresponds to the return of no value.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <signal out="out" expression="ite(s1!=0,s2/s1,NaN)"/>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>

```

Listing 259: File signal.tem

Boolean expressions can also be defined as returns. In the following example, a 1 (true) is returned, if s1 is not 0 and s1 is greater than 0. If s1 is not equal to 0 and s1 is not greater than 0, then a 0 (false) returned. If, on the other hand, s1 is 0, s2 is evaluated accordingly.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <signal out="out" expression="ite(s1!=0,s1>0,s2>0)"/>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>

```

Listing 260: File signal.tem

Note that the keywords true and false are not directly supported. Instead are 1 (true) and 0 (false) to be used. In the following example, a 1 (true) is output if s1 is greater than 0, otherwise a 0 (false) is returned.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <signal out="out" expression="ite(s1>0,1,0)"/>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>

```

Listing 261: File signal.tem

Negation is possible, as you would expect, with !

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <signal out="out" expression="ite(s1>0,!s1,!s2)"/>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>

```

Listing 262: File signal.tem

The period function is used to determine the time interval between two successive signal values issued. This can be useful, for example, to react to signal timeouts, or to perform a histogram via the To perform transmission distances of signals.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <signal out="out" expression="period(s1)"/>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>

```

Listing 263: File signal.tem

With the `debounce` function, the output is not updated until it has not been has changed. The time for which the value must be stable is specified in the second parameter. So in the given example the output changes only when it was stable for at least 1.1 sec. Fast Changes of the signal `s1` are thus ignored.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <signal out="out" expression="debounce(s1,1.1)"/>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>

```

Listing 264: File signal.tem

The `nonZeroTime` function outputs the time the signal is already different from zero. This can be used, for example, to measure how long a certain state is set. It also makes sense to use this function together with the `GateEvaluator`. Here the gate signal can be set with of this function can be generated. If this signal is additionally "gated" the output tuple of the `GateEvaluators` also notes the duration of an event. (Siehe auch 10.12)

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <signal out="out" expression="nonZeroTime(s1)"/>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>

```

Listing 265: File signal.tem

With the functions `min` and `max` the minimum or maximum of two values can be output. If one of the two input parameters is NaN, then NaN is also output.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <signal out="o1" expression="min(s1,s2)"/>
    <signal out="o2" expression="max(s1,s2)"/>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>

```

Listing 266: File signal.tem

The `diff` and `integral` functions can be used to differentiate and integrate signals. In differentiation, the difference between two signal values is formed and divided by the time interval. The integral is calculated according to the trapezoidal rule. In addition to the signal to be integrated, a timeout can be determined as a second parameter. Signal values that are further apart in time than the timeout are then ignored during integration.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <signal out="o1" expression="diff(s1)"/>
    <signal out="o2" expression="integral(s1)"/>
    <signal out="o3" expression="integral(s1,1.0)"/>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>
```

Listing 267: File signal.tem

There are two versions of a special filter function: AMeDA internally uses a data flow algorithm for all calculations. This means that every change of a signal value by a directed graph is propagated, with one operation being performed in each node. A special feature is that this propagation can be terminated in any node. This means that a function which has this Mechanism, can have no output by not outputting a signal value, thus terminating propagation. In the following example, the filter function only generates a new value for the `out` signal, if the signal `s2` is greater than zero. If this is the case, `s1*2` is output.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <signal out="out" expression="filter(s1*2,s2>0)"/>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>
```

Listing 268: File signal.tem

In the second manifestation of the filter function, an interval is specified. If the value of the first Parameters between the values of the second and third parameters, it is output. Otherwise, no result will be generated. In the following example the signal `out` has the value `s1*2` if `1<=s1*2<=10` applies. If not, there is no new signal value for `out`.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <signal out="out" expression="filter(s1*2,1,10)"/>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>
```

Listing 269: File signal.tem

With the function `onChange` the signal is only output if it has changed. The first incoming Value is always output. Otherwise no output is generated, which is internally realized via NaN (not a number). In the following example the signal `out` is described

with the signal `s1` if the value of `s1` has changed compared to its direct predecessor value.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <signal out="out" expression="onChange(s1)"/>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>
```

Listing 270: File signal.tem

11.2 Declaration

Virtual signals can also be generated by self-defined functions. These functions must be implemented in Java can be implemented, and can have a “memory”, so that filters or similar can also be defined.

In the `<defineFunction/>` tag the function is declared. The tag `<parameter/>` contains the parameters of the Function specified. In the `<global/>` tag, global definitions or variables can be declared, and in the `<function/>` tag finally the actual function is described. As with expressions, a signal is generated via the `<signal/>` tag. Thereby in the attribute `function` the name of the declared function is specified, and in the `out` attribute the name of the output signal. The required input signals are specified below the `<signal/>` tag in `<in/>` tags. The signal name is in the `name` attribute. In the `unit` attribute the unit of the generated signal can be specified. In the following example, a function named `incr` is created which outputs a one whenever the current value of the input signal is greater than the last value, i.e. whenever the input signal increases. Then the signal `out` is generated, which contains the result of the function `incr`, where signal `s1` is used as input signal.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <defineFunction name="incr">
      <parameter>
        <par name="a" />
      </parameter>
      <global>
        double last=Double.MAX_VALUE;
      </global>
      <function>
        double res;
        if (a>last) res=1;
            else res=0;
        last=a;
        return res;
      </function>
    </defineFunction>
    <signal function="incr" out="out" unit="1">
      <in name="s1" />
    </signal>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>
```

Listing 271: File signal.tem

Again, no value can be returned. This is achieved when the Double constant Double.NaN is returned. This value ends the propagation. In the following example, whenever the input signal is increases, the difference is output, otherwise no value is output.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <defineFunction name="incr">
      <parameter>
        <par name="a" />
      </parameter>
      <global>
        double last=Double.MAX_VALUE;
      </global>
      <function>
        double res;
        if (a>last) res=a-last;
        else res=Double.NaN;

        last=a;
        return res;
      </function>
    </defineFunction>
    <signal function="incr" out="out" unit="1">
      <in name="s1" />
    </signal>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>
```

Listing 272: File signal.tem

11.3 Special functions

There are some very special functions available, which can be used for special tasks.

The MovingAverage function can be used to generate moving averages. In the in attribute the Input signal, in the attribute out the output signal to be generated is specified. In the points attribute is specified, over how many points the averaging should be performed. In the attribute unit a unit can still be specified.

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <signal function="MovingAverage" in="s1" out="out" points="4" unit="1"/>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>
```

Listing 273: File signal.tem

With the function DelayCount measured values can be delayed by a fixed number of measuring points. At Attribute in the input signal is specified, in the attribute out the output signal to be generated. In the points attribute is specified, for how many points the output is delayed. In the attribute unit a unit can still be specified. Note that the last n values are not output when delayed by n values.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <signal function="DelayCount" in="s1" out="out" points="3" unit="1"/>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>

```

Listing 274: File signal.tem

The DelayTime function can be used to delay measured values by a fixed time. At Attribute in the input signal is specified, in the attribute out the output signal to be generated. In the time attribute is specified, by how many seconds the output is delayed. In the attribute unit a unit can still be specified.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <signal function="DelayTime" in="s1" out="out" time="2.5" unit="1"/>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>

```

Listing 275: File signal.tem

The high pass and low pass functions can be used to filter measured values. At Attribute in the input signal is specified, in the attribute out the output signal to be generated. The freq attribute specifies at which frequency the signal amplitude is reduced by half. In the attribute unit a unit can still be specified. In the meantime, the filters can be replaced by the corresponding functions. (Siehe auch VSE.FreqFilt)

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <signal function="Hochpass" in="s1" out="o1" freq="1" unit="1"/>
    <signal function="Tiefpass" in="s1" out="o2" freq="1" unit="1"/>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>

```

Listing 276: File signal.tem

With the function RealTime a virtual signal can be generated which outputs the current relative time. This signal can be used to generate functions, for example. For this purpose, the attribute start contains the start time and the step size is specified in the step attribute. An end time must be specified in the end attribute. However, no more values are generated if there is no more measurement data. So if a measurement has values up to t=1000 the evaluation is terminated, even if the attribute end contains a significantly larger value is specified. In the example the signal myTime is generated, which creates a measuring point every 250ms, and then the signal sin is calculated from it.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>Evaluator</evaluator>
  <virtualSignals>
    <signal function="RealTime" out="myTime" start="0" step="0.25" end="100" />
    <signal out="sin" expression="sin(myTime/2*pi)"/>
  </virtualSignals>
  <description>Test</description>
  <content/>
</template>

```

Listing 277: File signal.tem

11.4 Anonymous functions

In this example, a simple template is shown, which outputs an unchanged input signal as a result. A signal name is expected in the attribute name of the tag `<signal/>`. In this example, this simple name is replaced by `$(s1*2)`. The expression implemented in `$ (...)` is added and calculated as a virtual signal. As such, possible `<virtualSignal/>` tags can be saved.

```

<?xml version="1.0" encoding="UTF-8"?>
<template>
  <evaluator>FunctionEvaluator</evaluator>
  <description>Test</description>
  <instance>
    <signal name="$(s1*2)"/>
    <result name="out"/>
  </instance>
  <content>
    <parameter> <par name="a"/> </parameter>
    <function>
      result=a;
    </function>
  </content>
</template>

```

Listing 278: File test.tem

Index

- *.ASC, 6
- *.LOG, 6
- *.MDF, 6
- *.TEM, 7
- *.TKO, 7
- *.WAV, 6

- abortOnError="...", 58
- abortEvaluation, 92, 93, 99
- abs, 72, 126
- abs(x), 13
- <addToEvent/>, 56
- ADTFExportEvaluator, 26
- <after/>, 92
- Alias, 9
- alias, 9
- all="...", 32
- and, 13
- Anonymous functions, 134
- at -[t], 73
- ATFXClassificationExportEvaluator, 26
- AttributeGetter, 18
- Attributes in the measurement data platform, 18
- AttributeSetter, 19
- autoscale="...", 78

- backgroundColor="...", 78
- <bapAddrParam/>, 32
- <bapError/>, 32
- BAPEvaluator, 32
- <bapXML/>, 32
- baud="...", 33
- <before/>, 92
- bidirectional="...", 44
- BinLoggingEvaluator, 82
- BinLoggingEvaluator, 82
- BluePiraT (OTMT/XTMT/TMT), 19
- ByteStream, 27
- ByteStreamMultiplexer, 27
- ByteStreamToFileEvaluator, 30
- ByteStreamToStringEvaluator, 30

- call="false", 11
- CAN, 32, 98
- <can/>, 33, 114, 115
- can="...", 38, 40
- CANBusLoadEvaluator, 33
- CANCheckEvaluator, 34
- CANMessageTimeoutEvaluator, 35
- <canSignal/>, 114
- CanToByteStreamEvaluator, 31
- case, 12
- CaTe (CLG), 20
- ceil(x), 13
- channelNum="...", 42
- <classes/>, 117
- ClassEvaluator, 106
- classWidth="...", 26
- columnName="...", 58
- <columnNames/>, 108
- columnSeparator="...", 50, 87
- combineType="...", 67, 68
- completeTable="...", 50, 87
- Constants, 11
- constSignal, 11
- <content/>, 7, 26, 39, 41, 42, 49, 52, 54, 57, 58, 62, 66–68, 83–85, 88, 100, 106, 117, 121
- Control commands, 12
- Controller, 102
- cos, 126
- CounterBZEvaluator, 114
- CounterEvaluator, 116
- countOnEnable="...", 107
- countOnFirstValue="...", 107
- <country/>, 74
- createEvent, 92
- createList, 12
- CSV, 20
- <csv/>, 87
- cycleLengthName="...", 58
- cycles="...", 35, 81

- <data/>, 30, 80
- Data and message types, 24
- <databaseTable/>, 69
- dataClass="...", 99
- DataCollector, 35
- DataDouble, 98
- DataExtractEvaluator, 49
- DataExtractRPC3Evaluator, 51
- DataExtractSimpleEvaluator, 51
- DBC, 10, 35
- <dbc/>, 35
- DBCChannelSwap, 10
- dbcChannelSwapList, 10
- <dbcFile/>, 69
- DBCs, 21
- debounce, 129
- decimalFormat, 13
- decimalSeparator="...", 50, 87
- Declaration, 131
- decreasingEdgeNumber="...", 58
- default="...", 76
- defaultValue="...", 50, 87
- <defineFunction/>, 131
- <delay/>, 114
- delay="...", 106
- DelayCount, 132
- DelayTime, 133
- <description/>, 7, 116
- diagInfo="...", 38, 40, 41
- DiagMessageEvaluator, 36
- DiagMessageEvaluatorOdx, 37

Diagnosis, 36
 diff, 130
 diffStartEndName="...", 58
 DoIPFilterEvaluator, 42
 Drehzahl.tem, 7
 Drive Recorder TCS (PST), 21
 DTCTableS="...", 39
 DTCTableT="...", 39, 41
 duplicationHandling="...", 60

 e7="...", 32
 <ecu/>, 43
 element="...", 47
 <enable/>, 61
 enable="...", 35, 57, 63, 66, 89
 enabled="...", 74
 enableSignal="...", 124
 end="...", 133
 endClosedOnly="...", 57
 endName="...", 58
 endswith, 13
 <enter/>, 118, 120
 <errors/>, 35
 <etherFilter/>, 44
 Ethernet, 42
 <ethernet/>, 44
 EthernetFilterEvaluator, 43
 EthernetTrafficEvaluator, 44
 <evaluator/>, 7
 Evaluators, 25
 event, 10, 91, 92
 Event-based, 10
 <exclude/>, 43
 exclude="...", 36, 50, 62, 84, 86
 exist, 8
 exp, 127
 Export, 49
 expression="...", 125
 Expressions, 125
 ext="...", 49, 50
 extractFilename, 13
 Extras, 68

 factor="...", 35, 81
 FIBEX AUTOSAR, 37
 File formats, 19
 FileExportEvaluator, 52
 FILENAME="...", 83
 fileName="...", 26
 filename="...", 82
 FilePassThroughEvaluator, 55
 filePattern="...", 58
 <filter/>, 36, 45, 84
 filter, 130
 <filters/>, 36, 45, 84
 filters="...", 47
 <finally/>, 92, 93, 117, 118, 120, 121
 <first/>, 91, 117, 121
 Flea-CSV (LOG), 21
 FlexRay messages, 24
 floor(x), 13

 fontSize="...", 78
 format="...", 30
 format, 13, 92
 FourierEvaluator, 105
 FrameFilterEvaluator, 36
 FrameFilterEvaluator, 33, 36
 FrameFilterEvaluator, 36
 FrameLoggingEvaluator, 83
 FrameLoggingEvaluator, 36, 83
 FrameMultiplexEvaluator, 68
 freq="...", 133
 frequency, 103
 from="...", 36, 84
 <function/>, 7, 71, 91, 117, 120, 131
 function="...", 131
 <functionCatalog/>, 32
 FunctionEvaluator, 7, 91, 98
 FuzzyEvaluator, 71

 G.i.N. logger (GJF), 21
 G.i.N. logger (ML), 22
 gap="...", 104
 <gate/>, 76
 GateEvaluator, 75
 GateRealTimeEvaluator, 76
 General information, 27
 GenericClassEvaluator, 117
 Geometry, 77
 GeometryCalculationEvaluator, 77
 GeometryObjectEvaluator, 77
 getDataCollector, 92
 getDate, 13
 getLogger, 92
 getRTDate, 92
 getSigName, 92
 getValue, 98
 getValue(), 98
 <global/>, 91, 118, 131
 GNLogGeneratorEvaluator, 31
 GPS, 74
 Graphic, 78
 <graphic/>, 78, 79
 graphicHeight, 11
 Graphics generation, 10
 graphicWidth, 11
 gridColor="...", 78
 group="...", 87
 GroupResamplingEvaluator, 102

 handleNames="...", 58
 HandleSignalDuplicationEvaluator, 68
 HandleSignalDuplicationEvaluator, 68
 height="...", 78
 hex="...", 30
 high pass, 133
 highpass, 127
 hyst="...", 102

 id="...", 114
 IdexExport, 39

if, 12	mean="...", 124
ignore="...", 64, 66	mean grad of, 73
ignoreTime="...", 57	meanName="...", 58
<import/>, 92	<measDeviceTable/>, 69
<in/>, 131	Measurement Data Format (MDF), 23
in="...", 82, 132, 133	MeasurementAnalysisEvaluator, 88
increasingEdgeNumber="...", 58	MeasurementInfoEvaluator, 69
initDataCollector(), 14	memEfficientMode="...", 106
inSignal="...", 31–33, 38, 40	<message/>, 35, 81, 114, 115
instance, 91	MetaDatabase, 116
<instance/>, 7, 25, 42, 57, 60, 68, 75, 76, 86, 96, 100, 117, 122	MetaTemplateEditor, 6
integral, 130	min="...", 35, 81
interpolate="...", 78	min, 116, 129
interval="...", 33, 100	min of, 73
isDataCollector, 92	min(x, y), 13
ISOCan, 38	<minmax/>, 89
ite, 127	MinMaxEvaluator, 89
ite([bed], [Then], [Else]), 72	minName="...", 58
JavaEvaluator, 97	minOutName="...", 102
<javaFile/>, 96	MLClassificationExportEvaluator, 58
keepTextLog="...", 82	<mode/>, 100, 106
KWP2000, 39	mode="...", 60–62, 107
	<monitor/>, 60, 62
	MovingAverage, 132
	mux="...", 27
lastValueOnChange="...", 63	
LDF, 81	Name="...", 114
<ldf/>, 81	name="...", 27, 33, 35, 36, 38–41, 43, 49, 60, 61, 68, 69, 78, 81, 89, 96, 103, 107, 117, 131, 134
<leave/>, 118, 120, 121	NDPFilterEvaluator, 46
limit="...", 35	nodeAddr="...", 43
LIN, 81	nonZeroTime, 129
lines="...", 78	not, 13
lineSep="...", 30	num="...", 33, 114
LINMessageTimeoutEvaluator, 81	number="...", 47
Listen, 12	numClasses="...", 26
ln, 72, 126	
log, 127	onChange, 130
LogbookEvaluator, 56	oneFilePerTrigger="...", 83, 84, 88
Logging, 82	onlyCausedFileToTrigger="...", 83, 84, 88
LogHandlingEvaluator, 82	operation="...", 45
low pass, 133	operation-="...", 47
lowpass, 127	<options/>, 57
lsatValueOnChange="...", 66	or, 13
	out="...", 26, 30, 38–42, 68, 125, 131–133
makeSignals="...", 32	outFileName="...", 83
mandatory="...", 35	<output/>, 36
MapEvaluator, 79	overlapped="...", 84
mark="...", 78	
<markNotValid/>, 89	<par/>, 91, 99
Mathematical operations, 13	<parameter/>, 7, 71, 91, 131
Matlab (MAT), 23	parentFile, 13
max="...", 35, 81	pattern="...", 60
max, 116, 129	PCAPNG, 23
max of, 73	pdx="...", 39, 41, 42
max(x, y), 13	Peak Counting, 109
maxEventTime="...", 78, 85	Peak Through, 110
maxName="...", 58	period="...", 44, 60, 107
maxOutName="...", 102	
maxTime="...", 89	
<mdf/>, 85	
MDF 4.0 (MF4), 23	

period, 128
 plain="...", 50
 points="...", 78, 132
 <port/>, 47
 <ports/>, 47
 post="...", 78, 83, 85
 postBytes="...", 82
 postTolerance="...", 56
 postTrigger="...", 83
 pre="...", 78, 83, 85
 preBytes="...", 82
 PreprocessingEvaluator, 90
 preTolerance="...", 56
 preTrigger="...", 83
 PreviewGraphicCreator, 78
 PreviewGraphicCreator, 10, 78, 79
 primary="...", 84
 Programmable, 91
 <protocol/>, 37
 PTPFilterEvaluator, 46

 QuantileEvaluator, 123

 radius="...", 74
 Rainflow, 111
 Range Counting, 112
 rate="...", 103
 RealTime, 133
 RealTimeEvaluator, 100
 realtimeevent, 100
 ReducedSignalExportEvaluator, 59
 <region/>, 74
 ReglerSchwingen, 102
 requireWsProtocolSwitch="...", 49
 <resample/>, 103, 104
 ResampleEvaluator, 103
 ResampleMeanEvaluator, 103
 Resampling, 102
 <result/>, 33
 result, 91
 ResultFileListExportEvaluator, 59
 retentionPeriod="...", 58
 ReverseGeocodingEvaluator, 74
 <road/>, 74
 round, 127
 round(x), 13
 RouteToDBEvaluator, 75
 RPC 3, 23

 <save/>, 49–51
 scanZip="...", 56
 scheduleTable="...", 81
 Schwingen, 102
 SCR file, 13
 script, 8
 sdev="...", 124
 set, 8
 setDescription, 92
 setDoubleValue, 92
 setEventLevel, 92
 setStringValue, 92

 showGrid="...", 78
 <sids/>, 42
 sign, 126
 sign(x), 13
 <signal/>, 25, 27, 36, 39, 41, 42, 60, 79, 91, 100, 106, 131, 134
 signal="...", 64, 67
 Signal name conventions, 25
 Signal processing, 105
 signalAlias="...", 26
 SignalCleaningEvaluator, 104
 <signalCombine/>, 67
 <signalCombines/>, 66
 SignalLoggingEvaluator, 85
 SignalLoggingEvaluator, 85
 SignalMetaDataExportEvaluator, 60
 SignalMonitorEvaluator, 60
 signalName="...", 74
 <signalNameSuffix/>, 69
 SignalToTableEvaluator, 62
 <signalToView/>, 78
 SignalValidityCheckEvaluator, 69
 <signalValueToGet/>, 27, 61
 simpleTable="...", 26
 sin, 126, 133
 singleShot="...", 76
 SlidingWindowFunctionEvaluator, 106
 SomeIPFilterEvaluator, 46
 <spatialConstraint/>, 75
 Special functions, 132
 speed="...", 40
 sqrt, 72, 127
 sqrt(x), 13
 start="...", 26, 133
 startClosedOnly="...", 57
 startID="...", 41
 startMdfAtZero="...", 49, 85
 startName="...", 58
 startswith, 13
 Static, 11
 StaticGraphicCreator, 79
 StaticGraphicCreator, 10, 79
 Statistical Data Message (LDC), 24
 Statistics, 106
 step="...", 133
 <storeEveryFile/>, 107
 streamNameCan="...", 26
 streamNameEthernet="...", 26
 streamNameFlexray="...", 26
 streamNameUart="...", 26
 Strings, 13
 strLen, 13
 strPos, 13
 stylesheet="...", 39
 subStr, 13
 Summary, 122
 switch, 12

 <Table/>, 69
 <table/>, 69
 tableName="...", 26, 62, 124

tableNamePrefix="...", 58	xor, 13
tag="...", 27	XPath, 9
tan, 126	
TCPCheckEvaluator, 48	yaxis="...", 79
<template/>, 7	ydynamic="...", 78
textColor="...", 78	ymin="...", 79
The context, 14	
threshold="...", 57	
thresholdForCounter="...", 58	Zero Crossing Peak Counting, 113
<time/>, 100	zip="...", 26, 30, 51, 83–85
time="...", 114, 133	
time, 91	
<timeConstraint/>, 75	
<timeCorrect/>, 101	
timeout="...", 103, 104, 121	
timeout>0="...", 104	
timeOutMean="...", 57	
<times/>, 78, 85, 87	
TKO, 70	
TKO file, 8	
to="...", 36, 84	
TP16, 40	
tp16adr="...", 40	
TP20, 40	
tp20adr="...", 41	
<tpmessage/>, 43	
<trace/>, 83, 84	
<trigger/>, 60, 61, 66, 107, 115, 117	
triggerSignal="...", 124	
TwoPassEvaluator, 70	
TwoPassEvaluator, 70	
type="...", 36, 68, 84, 106	
<typeOfRoad/>, 74	
UDS, 41	
UDS (ODX), 41	
UDSRequest="...", 38	
UDSResponse="...", 38, 39	
UniCAN2/3 (CSM), 24	
unit="...", 131–133	
upper, 13	
usedata="...", 45	
usingBV="...", 42	
value="...", 47, 80	
valueAtEnd="...", 27	
<valueChange/>, 64, 65	
var="...", 124	
variant="...", 42	
Virtual signals, 125	
<virtualSignal/>, 134	
<virtualSignals/>, 125	
ViWiFilterEvaluator, 49	
wasReceived, 92	
while, 12	
width="...", 78	
width of, 73	
windowSize="...", 44	
writeLastRecord="...", 65, 66	
writeOnEachValue="...", 66	