Vector AUTOSAR Scripting Engine

User Manual

Version 2.0.9

Authors	Uwe Honekamp
Status	Released

Document Information

History

Author	Date	Version	Remarks
Uwe Honekamp	2014-12-11	1.0.0	Initial version
Uwe Honekamp	2014-12-12	1.0.1	Add explanation of how to access numerical values
Uwe Honekamp	2014-12-16	1.0.2	Fix minor issues
Uwe Honekamp	2015-07-16	1.1.0	Add support for command-line variable definitions, refine semantics of the “postfix” command-line argument
Uwe Honekamp	2015-09-29	1.1.1	Update required version of IronPython, update error codes
Uwe Honekamp	2016-09-22	1.2	Update document template, document validation behavior and schema support
Uwe Honekamp	2016-11-21	1.2.1	Document possible return values
Uwe Honekamp	2017-01-10	2.0.0	Update for VASE 2.0
Uwe Honekamp	2017-04-06	2.0.1	Clarify required version of IronPython
Uwe Honekamp	2017-04-20	2.0.2	Command line option for postfix
Uwe Honekamp	2017-07-14	2.0.3	Fix reference to supported AUTOSAR release
Uwe Honekamp	2017-10-04	2.0.4	Update reference to supported AUTOSAR XML Schema release
Uwe Honekamp	2018-01-04	2.0.5	Update reference to supported AUTOSAR XML Schema release, output files are only created if changes occurred, execution of multiple scripts in one go
Uwe Honekamp	2019-02-08	2.0.6	Update reference to supported AUTOSAR XML Schema release
Uwe Honekamp	2020-03-27	2.0.7	Remove note about future extensions, Update reference to supported AUTOSAR XML Schema release
Uwe Honekamp	2021-02-05	2.0.8	Add ability to detect whether the AUTOSAR Schema version fits the API version

			used in a script. Update reference to supported AUTOSAR XML Schema release.
Uwe Honekamp	2021-04-15	2.0.9	Fix reference to applicable AUTOSAR release

Reference Documents

No.	Source	Title	Version
[1]		AUTOSAR XML Schema	00049
[2]		IronPython (https://ironpython.codeplex.com/)	2.7.5
[3]		Visual Studio (http://www.visualstudio.com/)	
[4]		Python Tools for Visual Studio (https://pytools.codeplex.com/)	2.2

Contents

1 About this Document

This document describes the purpose and the usage of the *Vector AUTOSAR Scripting Engine*. It describes the principles behind the creation of the tool as well as how to use it. The document is completed by a description of how the applicable model API can be used.

2 Script-based Editing of AUTOSAR Models

2.1 Overview

The typical approach to apply changes to an AUTOSAR model is the use of a dedicated model editor (or AUTOSAR authoring tool) that provides a more or less sophisticated graphical user interface. The purpose of the authoring tool is mainly to (mostly by abstraction) manage the underlying complexity of the model and allow the user to stay focused on the task.

The more sophisticated model editors get, the more likely it is (as a cost of abstraction) that they will not be able to provide a high-quality access to model changes for every single aspect of the AUTOSAR XML Schema [1].

In other words, authoring tools tend to specialize on certain use cases in the application of the AUTOSAR methodology and the meta-model; therefore, they lack the versatility to be able to access the whole modeling range in an equal manner.

This results in authoring tools that provide significant support for e.g. the modeling of AUTOSAR software-components but lack the ability to assist the user in maintaining the modeling of the communication matrix.

Also, the authoring tools are typically aimed at the application of interactive model changes in order to develop a model from a given level of completeness towards a certain project goal.

In other words, the primary goal is to develop the model in terms of completeness but not in terms of reproducibility. The necessary model changes related to the underlying project are executed and once completed, will not be repeated again.

In summary, typical AUTOSAR authoring tools do not properly respond to the following requirements:

1. Be able to execute a model change in a strictly reproducible way.
2. Be able to formulate model changes algorithmically.
3. Be able to uniformly access every single detail of an AUTOSAR model.

On the other hand, these requirements can easily be met by an AUTOSAR tool with the following abilities:

1. Expose a uniform and detailed model API to the user.
2. Provide the ability to use the model API in a scripting environment.
3. Provide the ability to execute scripts that utilize the model API.

The *Vector AUTOSAR Scripting Engine* represents a working implementation of the above-mentioned tool characteristics.

2.2 Principles

The implementation of the *Vector AUTOSAR Scripting Engine* is based on a number of principles that are explained in the following sub-chapters.

2.2.1 Command-Line Tool

The *Vector AUTOSAR Scripting Engine* is implemented as a command-line tool. The configuration is done by means of command-line arguments. The details are explained in section 3.

2.2.2 Inbound Model and Outbound Model

The *Vector AUTOSAR Scripting Engine* does not change the model files. Instead, new files are created that take the name of the given inbound files **plus a given postfix string**.

For example, an inbound file named MyModelFile.arxml is (upon termination of the *Vector AUTOSAR Scripting Engine*) accompanied by an additional model file with the name MyModelFile_<postfix>.arxml that contains the modified model content.

Note
This way, it is not only possible to reproduce the model change but it is also feasible to check the changes on the model individually.

However, the creation of new files depends on the fulfillment of a condition.

Caution
Output files that are named after the input file plus a given postfix are **only created if there are actual changes** in the partial model contained in a specific file. Even if the overall model has been changed, output files will only be created for input files that carried partial modes updated by the execution of the Python script.

2.2.3 Scripting Approach

The *Vector AUTOSAR Scripting Engine* supports the execution of scripts written in *IronPython* [2]. The filename of the script that shall be executed by the *Vector AUTOSAR Scripting Engine* is passed as a command-line argument to the tool.

The restriction to supporting *IronPython* exclusively has mostly technical motivations. Different scripting languages impose different requirements on the integration with a tool like the *Vector AUTOSAR Scripting Engine.* Therefore, the integration of a specific scripting language needs to be implemented specifically into the tool.

2.2.4 User Variables

Scripts that modify a given class of models can further be parameterized by the definition of user variables that are defined outside the actual script source code and then injected into the scripting execution by the *Vector AUTOSAR Scripting Engine*.

This way it is possible to change the behavior of the script depending on the value of a given user variable that is passed to a specific script execution. The user variables can be defined in two ways (that can be mixed):

- by means of a simple XML notation
- by direct definition of a command-line argument

The definition using a simple XML file is exemplified in the following example:

Example:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<variables>
    <variable name="answer" value="42"/>
    <variable name="lorem" value="ipsum"/>
</variables>
```

Note that the variable definitions consist of just the name of the variable and the associated value. There is no definition of the corresponding data type required. This matches the dynamic nature of the Python scripting language which infers the data type from the value of the variable.

The definition of user variables by means of a command-line argument is explained in chapter 0.

2.2.5 Additional Search Path

The script executed by the *Vector AUTOSAR Scripting Engine* may use further modules. These modules are imported into the script and for this purpose the script compiler needs to be able to locate the module files.

Note
The compiler looks for the module files in the collection of available search paths. If the module cannot be located on the search path the compiler will issue an error message in response.

2.2.6 Creating Log Entries from within a Script

The *Vector AUTOSAR Scripting Engine* also exposes an API to the scripting environment that allows for creating log entries from within a script. Obviously, this ability can be utilized for monitoring whether the normal program flow is executed or for debugging purposes.

The API foresees the assignment of a verbosity level to each logging message. Upon execution of the *Vector AUTOSAR Scripting Engine*, it is possible to assign a global verbosity level for the filtering of logging entries.

Note
A particular log entry gets past the filter if the value assigned to a particular log entry is smaller or equal to the value of the global verbosity level. A value greater than the global verbosity level will lead to a cancellation of the specific log entry.

2.2.7 Execution of several Scripts in one go

The *Vector AUTOSAR Scripting Engine* supports the sequential execution of up to nine scripts on the same model in one go. This feature can be used to subsequently update the model according to different aspects using different scripts.

This may be useful if models with different levels of deficiencies need to be fixed. In this case it is possible to develop scripts with different focus and apply these sequentially on demand.

This approach delivers a much better performance than executing each script separately in one instance of the *Vector AUTOSAR Scripting Engine* because the model has to be overall loaded and saved just once, as opposed to once for every single script.

3 AUTOSAR Scripting Engine

3.1 Usage

The *Vector AUTOSAR Scripting Engine* comes as a command-line tool that is configured via command-line options. The specific supported command line options are documented in the following subchapters.

There are two alternative ways to specify command line option:

-x <arg>, i.e. the command-line option consists of a single "-", followed by a single character, at least one whitespace, and then a potentially optional[1] argument (e.g. the path to a specific file) to the command line option.

--x=<arg>, i.e. the command-line option consists of a double "-", followed by a pre-defined string of characters, an equal sign, and then a potentially optional argument (e.g. the path to a specific file) to the command line option.

Either alternative can be used to pass a specific command-line option to the program; it is also possible to mix the specification of the two forms of command-line options within a given command-line.

Example

```
Vector.DaVinci.Vase.exe –h
```

3.2 Indirect File

As there are a significant number of supported command-line options, the resulting command line may get quite long, potentially causing truncation issues in certain command-line processors that only support a limited number of characters on the command line.

To mitigate this issue, it is possible to specify the list of command line options in a text file that contains the specification of one command-line option per line.

The name of the file that contains the specification of command-line option shall be passed to the *Vector AUTOSAR Scripting Engine* along with a leading "@" that indicates the special nature of the option.

[1] Whether or not a specific command line option requires an argument is explained in section 3.3

Example
```
Vector.DaVinci.Vase.exe @args.txt
```

3.3 Command-Line Parameters

3.3.1 Help

The help page will be displayed if the command-line option **–h** or **--help** is found. The program will terminate after displaying the help page even if another command-line option are found.

3.3.2 AUTOSAR Model File

The path to a file that contains an AUTOSAR model shall be passed along with the command-line option **-a** or **--arxml**. This option may appear at multiple times on the command line, i.e. it is possible to split the model content over several files.

Example
```
--arxml=MySystemDescription.arxml
```

3.3.3 Definition of User Variables

The path to a file that contains the definition of the user-specific variable values shall be passed along with the command-line option **-u** or **--uservar**. This option may appear zero or once on the command line.

Example
```
--uservar=MyUserVar.xml
```

3.3.4 Additional Search Path

A path that shall be taken as an additional search path by the script shall be passed along with the command-line option **-s** or **--searchpath**. This option may appear multiple times on the command line, i.e. every additional search path shall be submitted separately to the *Vector AUTOSAR Scripting Engine*.

Example
```
--searchpath=c:\some\path\name
```

3.3.5 Postfix for the Naming of Outbound Model Files

The postfix appended to the filename of outbound model files shall be passed along with the command-line option **-p** or **--postfix**. This option shall appear at most once on the command line.

Note
If this option is missing the model is **not** saved to the file system at the end of script execution

Example
```
--postfix=patched
```

Caution
The value of this option shall not contain whitespaces, i.e.
```
--postfix=x y
```
Is not supported!

3.3.6 Python Script Filename

The path to the file that contains the actual script for model modification shall be passed along with the command-line option **-y** or **--python**. This option may appear only a single time on the command line.

Example
```
--python=MyScript.py
```

Note that the above-mentioned command-line option is just one way to provide script file names to the *Vector AUTOSAR Scripting Engine*.

In addition, it is possible to define multiple scripts at once for execution in one go. For this purpose, command-line option **-y<n>** resp. **--python<n>** can be used. <n> can be replaced by any value from 1 to 9.

In other words, it is possible to specify up to 9 script file names for sequential execution. The value of <n> determines the order of execution, i.e. a file provided as -y<n-1> is executed before another file provided as -y<n>.

Example
```
--python1=MyFirstScript.py --python2=MySecondScript --python5=MyThirdScript
```

It is not allowed that -y<n> with the same value of <n> appears several times in the same command-line. It is possible, however, to create “gaps” in the values of -y<n>. The files are simply executed in the order of values of <n>, with the lowest value first and in increasing order of <n>.

Caution
It is required that all script files provided by means of **-y<n>** resp. **--python<n>** use the same version of the model API. A mixture of e.g. API1 and API2 is not supported.

3.3.7 User Variable Definition

It is possible to inject user-defined variables into the script by providing the command-line option **-d** or **--define**. This option may appear zero or multiple times on the command line.

This option has to follow a sub-syntax such that the name and the value are expected to be separated by a hash-sign (#). A typical use case for this command line option could be to pass the name of a file that shall be written by the script.

Example
```
--define=answer#42
```

3.3.8 Log Verbosity

The threshold value for the log verbosity shall be passed along with the command-line option **-l** or **--logverbosity**. This option may appear at most once on the command line.

Example
```
--logverbosity=4
```

3.3.9 Help Page

Use the command line argument **-h** or **--help** to let the *Vector AUTOSAR Scripting Engine* display a help page that explains the usage of command-line arguments.

Example
```
> Vector.DaVinci.Vase.exe -h
Usage: Vector.DaVinci.Vase <options>

Options:
  -a <filename>,     --arxml=<filename>      [0..*]: AUTOSAR model files to Load
  -u <filename>,     --uservar=<filename>    [0..1]: Definition of user-defined
variables
  -s <filename>,     --searchpath=<pathame>  [0..*]: Additional user-defined search path
entries
  -p <postfix>,      --postfix=<postfix>     [0..1]: Postfix to add to the output file
  -y <filename>,     --python=<filename>     [0..1]: Script
  -l <int>,          --logverbosity=<int>    [0..1]: Threshold level for log output
  -d <name>#<value>, --define=<name>#<value> [0..*]: user-defined variable with name and
value
  -h,                --help                  [0..1]: Display this help text
  -v,                --verbose               [0..1]: Output verbose information
```

3.3.10 Verbose

The *Vector AUTOSAR Scripting Engine* will output information about the timestamp of starting, stopping, and the time consumption if the command-line option **-v** or **--verbose** is

found on the command line. This command line option may appear at most once on the command line.

Example

```
Vector.DaVinci.Vase.exe @args.txt
Vector.DaVinci.Vase (V1.0.0.0) started at 07:41:54 2013-10-08

Vector.DaVinci.Vase (V1.0.0.0) stopped at 07:41:54 2013-10-08

Time consumption: 00:00:00.2370237
```

3.4 The Environment Variable IRONPYTHONPATH

In addition to passing the search path by means of the -s command line option, it is also possible to set the environment variable IRONPYTHONPATH accordingly. The *Vector AUTOSAR Scripting Engine* will read this variable from the environment and add its content to the search path.

A possible candidate to add to the environment variable IRONPYTHONPATH is the location of the *lib* directory in the IronPython distribution itself. This way, standard modules of the Python lib are available to the script.

For example, a specific module that the script might want to access is the *uuid* module. This module can be taken to add a UUID to model elements where applicable. The following code fragment explains how a UUID can be added to a model element, in this case a SystemSignalGroup.

Example

```
import uuid

# create the UUID
signalGroup.UuidSpecified=True
signalGroup.Uuid=str(uuid.uuid1())
```

Please find more explanations regarding details of the code fragment in section 4.

3.5 Validation of AUTOSAR Model Content

The Vector AUTOSAR Scripting Engine executes a validation of the AUTOSAR model content in two different phases:

1. Model content is validated against the latest supported version of the AUTOSAR schema while the model is loaded. Issues found will be generated as Warnings. This allows for subsequent fixing of certain classes of validation issues inside the script executed by the Vector AUTOSAR Scripting Engine.
2. Model content is validated after the model was saved to the file system. Any Issue found during this phase will be reported as an Error. Output files will always be validated against the latest supported version of the AUTOSAR XML Schema.

3.6 Supported Versions of AUTOSAR

The version of the Vector AUTOSAR Scripting Engine documented in this user manual supports all AUTOSAR releases in the interval 4.0.1 to R20-11.

4 The Model API

The purpose of this chapter is to explain the usage of the model API exposed by the *Vector AUTOSAR Scripting Engine*. This means there are certain patterns of how the model API is supposed to be used for a given purpose. The patterns are explained in the following sub-chapters.

4.1 API Compatibility

The model API of the *Vector AUTOSAR Scripting Engine* has already undergone some evolution. In the course of changes of the underlying model engine, APIs have been changed or removed.

4.1.1 API Compatibility Token

As a consequence, existing scripts would not work any longer. In order to mitigate this effect, a so-called *API compatibility token* has been introduced to allow for a given script to announce its compatibility level.

The *API compatibility token* can be located on any line in a given script. The expected format can be summarized by the following regular expression:

Expert Knowledge
#\s*%\s*VASE-API\s*=\s*([0-9]+)\.([0-9]+)\s*%\s*

In other words, the *API compatibility token* has to start with a pound-sign in order to pass as a comment (as far as Python is concerned). The actual content is expected to appear between two percent signs.

An example of a valid *API compatibility token* can be found below:

Example
#%VASE-API=2.0%

The specification of the actual compatibility level of a given script is implemented by the assignment **`VASE-API=<major>.<minor>`** where **`<major>`** represents the major version number whereas **`<minor>`** represents the minor version number.

The information about the API compatibility will be used by VASE to apply the proper model API to the respective script.

This could also mean that a specific script cannot be executed by a given version of VASE because the script uses an API version that is not supported by the particular version of VASE.

4.1.2 Application of the API Compatibility Token

There is a correlation between supported releases of the AUTOSAR standard and the applicable *API compatibility token* such that the API 1.0 can only be used for a given

interval of AUTSOAR releases. API 2.0, on the other hand, is able to support a wider range of AUTOSAR releases.

The details can be found in Table 4-1.

	API 1.0	API2.0
Supported AUTOSAR Releases	4.0.1 - 4.2.2	4.0.1 – R20-11

Table 4-1: Supported AUTOSAR releases in relation to API version

4.1.3 Conditions for the Detection of a valid API Compatibility Token

The following conditions exist for the successful detection of an *API compatibility token*:

- The line in which the *API compatibility token* is placed shall contain **nothing but** the *API compatibility token* **with the exception of leading and trailing white spaces**. In particular:
  - Leading non-white-space characters **placed before** the *API compatibility token* sequence will prevent the successful detection of the *API compatibility token.*
  - **Trailing non-white-space characters** will lead to an error message that renders the detected *API compatibility token* invalid.
- The **first successfully recognized** *API compatibility token* will be used for **launching the script**. The search for *API compatibility tokens* will stop and valid or invalid *API compatibility tokens* possibly located further down the script will be ignored.
- If the **first detected** *API compatibility token* **is invalid** then the *Vector AUTOSAR Scripting Engine* will **stop searching** for a further valid *API compatibility token*, raise an error message, and **terminate execution**. Valid *API compatibility tokens* possibly located further down the script will be ignored.
- If no valid *API compatibility token* appears in the script file then it will be assumed that the following definition applies implicitly: **`VASE-API=1.0`**.

4.2 Patterns

The most important rule in the model API is that model elements are never created out of context. In other words, there is no case where the new-operator would have to be called in order to extend the model.

The model API contains all necessary methods and properties to add model elements within a given enclosing context.

4.2.1 Finding a specific Model Element by ShortName

In many cases it will be necessary to access a given model element that is known by its qualified **`ShortName`**. This is supported by way of calling the **`Find()`** method of the model variable:

Example

```
elementList = model.Find("/My/Qualified/Short/Name")
for element in elementList:
    pass # do something
```

Please note that the **`Find()`** method always returns list of model elements even if there is positively only a single model element available with the given **`ShortName`**. The API behaves this way in order to properly support the situation where multiple model elements with the same qualified **`ShortName`** exist, i.e. in models that are subject to variant handling.

The safe way to handle this situation is to actually iterate over the list returned from the **`Find()`** method in order to make sure every applicable model element is properly handled.

Of course, it might be applicable to raise an issue if the iteration actually handles multiple model elements when the pre-condition applies that there should be only one model element with this specific qualified **`ShortNam`**e.

Note

This API applies to all currently supported API versions

4.2.2 Iteration over a Collection of Model Elements

For the iteration over a collection of model elements it is important to understand that the model API is created such that the name of the collection is composed of the role name of the collection concatenated by the postfix "List".

Example

```
for p in model.Introduction.PList:
```

The example fragment accesses (in terms of the underlying AUTOSAR meta-model) the role **`Introduction.p`**. An example of the application of this iteration can be found in section 4.2.5.

Note

This API applies to all currently supported API versions

Please note that, although the API call is invariant of the supported API version, the result of the call is not.

With respect to the above mentioned example, access of **`model.Introduction.PList`** will return a collection of **`IARObject`**s using API Version 1.0 whereas the call will return a collection of **`IMultiLanguageParagraph`**s for API 2.0.

When using a dynamically typed language like Python the practical difference between these two variants is negligible, though.

4.2.3 Add Model Element with Multiplicity 0..1

A model element of multiplicity 0..1 is added by first setting the corresponding **<Role>Specified** property. In other words, the API foresees the existence of two properties for this case:

1. The **<Role>Specified** property needs to be set to true **first**.
2. The actual value of the model element can only be assigned if the **<Role>Specified** Property has been set to true before. For this purpose, another Property **<Role>** named after the role of the model element in the AUTOSAR meta-model is available.

Example

```
# create the UUID
signalGroup.UuidSpecified=True
signalGroup.Uuid=str(uuid.uuid1())
```

A typical example for this behavior is the creation and assignment of a UUID, as already mentioned in section 3.4. In this specific case, the **<Role>Specified** property is named **UuidSpecified** and the **<Role>** property is named **Uuid** accordingly.

Note that the **Uuid** property in the model API is of type *string* and thus the script needs to create a string via the **str()** method out of the created UUID.

Note
This API applies to all currently supported API versions

4.2.4 Delete Model Element with Multiplicity 0..1

The deletion of an existing model element with multiplicity 0..1 (or even with multiplicity 1) can be done by setting the **<Role>Specified** property to false.

Note
This API applies to all currently supported API versions

4.2.5 Add Model Element with Multiplicity 0..*

The adding of one model element to a collection is done by calling a constructor method in the context of the parent model element. The name pattern for the method is created by the prefix **AddNew<Role>** resp. **AddNew<Role>As<Class>** concatenated by the applicable role name.

Example
```
signalGroup = arPackage.AddNewElementAsSystemSignalGroup()
signalGroup.ShortName = signalGroupName
```

This specific approach is exemplified by the creation of a new **`SystemSignalGroup`** in the contents of an **`ARPackage`**.

The result of the call to **`AddNewElementAsSystemSignalGroup()`** is assigned to a new variable named **`signalGroup`**. There is no need to provide a data type specification for the variable **`signalGroup`**. The variable gets its characteristics by way of the assignment.

Please note that the assignment of the **`ShortName`** is done by property rather than as an argument to the call of the constructor method.

Note
This API applies to all currently supported API versions

4.2.6 Delete Model Element with Multiplicity 0..*

The deletion of a model element that is part of a collection is done by calling the **`Remove<Role>`** method at the parent model element.

Example
```
arPackage.RemoveElement(signalGroup)
```

The example for this behavior is quite simple and refers to the creation of the **`signalGroup`** that is explained in section 4.2.5.

Note
This API applies to all currently supported API versions

4.2.7 Move Model Element from one Parent to another (API1.0)

It is possible to move a given model element from one parent to another but caveats apply in any case. In particular, references to the moved model element are **not** automatically migrated as well.

Example
```
clonedCompuMethod = targetPackage.AddNewElementAsCompuMethod();
targetPackage.Root.Domain.CopyContents(compuMethod, clonedCompuMethod)
sourcePackage.RemoveElement(compuMethod)
```

The source code explains the basic approach for the move from one parent to another by example of a **`CompuMethod`** in three steps:

1. The target **CompuMethod** is added to the target **ARPackage** as a new **CompuMethod**.
2. The **CopyContents(source,target)** method is called on any given model element. That's right; it does not really matter where the call is placed because the actual method is provided by an interface that can be accessed by calling the property **Root.Domain** on any model element.
3. The source **CompuMethod** is removed from the source **ARPackage**.

Again, this approach does not consider the fact that other model elements may be referring to the moved **CompuMethod**. After the move is completed, these references consequently become invalid (unless they are fixed in a further processing step).

4.2.8 Move Model Element from one Parent to another (API2.0)

The approach for moving an element from one parent to another is slightly more complex using the API 2.0.

```
Example
clonedCompuMethod = targetPackage.AddNewElementAsCompuMethod();
builder = IDomain.NewTransfer(targetPackage.Root.Domain, compuMethod)
copyBuilder = IBuilder[compuMethod.GetType()].Copy(builder)
transferer = ICopyBuilder[compuMethod.GetType()].Build(copyBuilder)
ITransferer[compuMethod.GetType()].TransferChildrenTo(transferer, clonedCompuMethod)
```

The source code explains the basic approach for the move from one parent to another by example of a **CompuMethod** in three steps:

1. The target **CompuMethod** is added to the target **ARPackage** as a new **CompuMethod**.
2. An **IBuilder<ICompuMethod>** is created by calling **IDomain.NewTransfer()**.
3. An **ICopyBuilder<ICompuMethod>** is created out of the recently created IBuilder**<ICompuMethod>**.
4. An **ITransferer<ICompuMethod>** is created out of the recently created **ICopyBuilder<ICompuMethod>**.
5. The **ITransferer<ICompuMethod>** takes care of taking over the content of the **CompuMethod** to the new place.

As indicated in the example fragment there is a specific way to access generic types in IronPython, i.e. by using square brackets instead of angle brackets (like in e.g. C#).

Expert Knowledge
It should be noted that the usage of generics is such that the actual type argument can be computed at runtime, by means of calling **`<object>.GetType()`**. Admittedly, this approach would **not work in a statically typed language at all** but is no problem in a dynamically typed language such as IronPython.

This approach does not consider the fact that other model elements may be referring to the moved **`CompuMethod`**. After the move is completed, these references consequently become invalid (unless they are fixed in a further processing step).

4.2.9 Add Reference to Model Element

Another important use case for the model API is the creation of a reference. For this purpose model elements typed by meta-classes that maintain references to other meta-classes provide constructor methods with the name pattern *AddNew<Role>().*

Example
```
systemSignalRef = systemSignalGroup.AddNewSystemSignal()
systemSignalRef.DestType = systemSignal.IdType
systemSignalRef.Value = anotherSystemSignal.AsrPath
```

The code fragment explains the way how the change can be done. It is important to also set the destination type of the reference by way of accessing the property **DestType** accordingly.

Then, the new reference target of the reference can be assigned to the property **Value** on the actual reference.

An existing reference can be redirected to a new target by just assigning a new value to the **Value** property.

4.2.10 Remove Reference to another Model Element

The removal of a reference can be done by calling a method with the name pattern **`Remove<Role>(ref)`** on the origin model element of the reference.

Example
```
systemSignalGroup.RemoveSystemSignal(systemSignalRef)
```

Of course, this operation simply removes the reference to an existing model element which is itself not affected.

Note
This API applies to all currently supported API versions

4.2.11 Use Interface provided by Model API

Python is a dynamic language and therefore it is usually not required to use an interface name provided by the model API to e.g. define a variable of this type. The variable is defined by the return value of a model API call instead.

However, there are cases where it will still be necessary to explicitly access classes provided by the model API

For more explanation, please have a look at the example mentioned in this chapter: a new element is added to the role **`MultiLanguageParagraph.l1`**. In other words, an object that represents a **`MultiLanguageParagraph`** is taken to call the method **`AddNewL1()`** onto it. The result of the method call is returned to a new object that represents the created **`LParagraph`**.

Next, the l attribute is set to **`LEnum.eLa`** to indicate the language. Finally, the value of the property is set.

Example

```
import clr
clr.AddReference("GenTool_CsDataServerDomAsr4.dll")
from GenTool_CsDataServerDomAsr4.Iface import LEnum

for p in model.Introduction.PList:
    newL1 = p.AddNewL1()
    newL1.L = LEnum.eLa
    newL1.Value="Lorem ipsum dolor sit amet"
```

In order to be able to access the **`LEnum`** class in this example it must be specifically imported into the script scope. The approach for this import is explained in the first three lines of the example.

Should more than one class be needed for dedicated access it will be possible to use a comma-separated list of class names in the **`from`** … **`import`** … clause.

Note
This approach applies to all currently supported API versions

4.2.12 Access a numerical Value

The access to numerical values is less straightforward that what could probably be expected. The reason for this is that the numerical values are implemented in an own class within the model API.

Access to a property that represents a numerical value actually returns an instance of the class, which in turn has properties that need to be accessed to get to the "real" numerical value.

Also, the value of the property that represents the "real" value is actually encoded by a string. This means that set-access to a numerical property requires the construction of a string out of a given numerical value in order to have the ability to assign the value to the property.

Example

```
baseTypeDirectDefinition.BaseTypeSizeSpecified = True
baseTypeDirectDefinition.BaseTypeSize.Value = str(48)
```

In the example above the property **`BaseTypeDirectDefinition.baseTypeSize`** is first created and then assigned to the value 48.

Note

This approach applies to all currently supported API versions

5 The Logging API

This chapter contains an explanation of the logging API available from within an executed script.

5.1 Error

Scripts have the ability to insert error messages into the error framework implemented in the *Vector AUTOSAR Scripting Engine.*

Example (call in script)

```
logger.Error("Something went wrong")
```

Example (output in console)

```
[Error] S1001 - Something went wrong
```

A call to this API makes the *Vector AUTOSAR Scripting Engine* return to the calling shell with a return value unequal to 0. This way, the calling shell will be able to determine that the execution of the tool terminated unsuccessfully.

Note
A call to the Error() API from within a script does not immediately terminate the script execution. Several errors can be reported subsequently.

Please note that calls to this API are handled asynchronously, i.e. the error framework gathers all error messages before they are listed right before the *Vector AUTOSAR Scripting Engine* terminates.

5.2 Warning

Scripts have the ability to insert warning messages into the error framework implemented in the *Vector AUTOSAR Scripting Engine.*

Example (call in script)

```
logger.Warning("Something suspicious just happened")
```

Example (output in console)

```
[Error] S1002 - Something suspicious just happened
```

A call to this API does not have any impact on the return value to the calling shell.

Please note that calls to this API are handled asynchronously, i.e. the error framework gathers all error messages before they are listed right before the *Vector AUTOSAR Scripting Engine* terminates.

5.3 Log Entry

Scripts have the ability to insert log entries into the textual output created by the *Vector AUTOSAR Scripting Engine.*

Example (call in script)

```
logger.Log(2,"Something worth mentioning happened")
```

Example (output in console)

```
[Script] - Something worth mentioning happened
```

A call to this API does not create an entry into the error handling framework and thus does not use a unique event ID. Furthermore, calls to this API are directly inserted into the textual output synchronously.

This way it is possible to watch the occurrence of log entries as the script executes (as opposed to error messages and warnings that are created immediately before the *Vector AUTOSAR Scripting Engine* terminates).

6 Prerequisites

The *Vector AUTOSAR Scripting Engine* needs additional resources on the target computer in order to become executable:

1. The .NET framework (version 4.7 or newer) needs to be installed
2. The distribution of IronPython (version 2.7.5) needs to be installed

On top of that, it would be useful to have the following components installed on computers used by script authors:

1. Microsoft Visual Studio [3] (version 2010 or newer), for creating scripts and provide a basic level of debugging.
2. The *Python Tools for Visual Studio* [4] (PTVS, version 2.2 or newer) support the script authors with syntax coloring for Python files and a basic syntax checker right in the editor.

7 Checker Tool for the Installation of IronPython

The execution of the *Vector AUTOSAR Scripting Engine* depends on the existence of IronPython on the respective machine.

Note
The *Vector AUTOSAR Scripting Engine* is linked against a specific version of IronPython. **It will only work if this version of IronPython is installed**.

In order to mitigate potential issues with the installation of IronPython (i.e. installation could be missing or it could be a different version than the required one) a dedicated checker tool has been created and added to the installation of the *Vector AUTOSAR Scripting Engine*.

The checker tool can be executed by a higher-level program that also launches the *Vector AUTOSAR Scripting Engine* as part of its workflow. By executing the checker tool, it can be made sure that the execution of the *Vector AUTOSAR Scripting Engine* is possible.

Upon execution, the checker tool will return **one of** the following values to the calling shell:

- The value **0** is returned if the proper version of IronPython has been found on the respective machine. This means that the *Vector AUTOSAR Scripting Engine* will **execute just fine** with respect to the required version of IronPython.
- The value **-1** is returned if a version of IronPython has been found that is different from the expected version. This means that the *Vector AUTOSAR Scripting Engine* will **not be able to execute**.
- The value **-2** is returned if no version of IronPython has been found. This means that the *Vector AUTOSAR Scripting Engine* will **not be able to execute**.

Caution
Please note that the *Vector AUTOSAR Scripting Engine* depends on an **installed** (i.e. created by running an installer program) version of IronPython. A **portable** version that **does not create any entries into the global assembly cache** (GAC) is not detected and therefore cannot be used for running the *Vector AUTOSAR Scripting Engine*.

As a consequence of the need of an **installed version of IronPython** that creates entries into the GAC, the checker tool may return with the value -2 even if a **portable** installation of IronPython is available on the respective machine.

Note
The checker tool can be executed by launching **Vector.DaVinci.Vase.Checker.exe** from the installation directory of the *Vector AUTOSAR Scripting Engine*.

8 Error Codes

This chapter contains a list of all possible error codes along with an explanation for each code. In particular, every error message issued one of the tools in the tool chain will contain a unique error code that allows to unambiguously identify a given error situation.

Each error code consists of a letter and a four-digit number. The letter indicates the component of one of the tools in the tool chain that issued the error. The documentation of error codes is therefore structured in subchapters where each discusses the error codes of a specific component.

8.1.1 Main Program

Error Code	Explanation
P1001	The parsing of command-line arguments failed because arguments with the wrong multiplicity were found.
P1002	A general problem occurred while parsing command-line arguments.
P1003	*No longer used*
P1004	An issue with the definition of the log verbosity was found. As the result, the logging from within the script execution will effectively be deactivated.
P1005	The script failed to compile. Note that it is unfortunately not feasible to provide further information like, e.g. the line number in the script file.
P1006	A runtime error occurred during script execution.
P1007	An exception occurred during program execution.
P1008	A model filename passed with the argument -a does not match to an existing file.
P1009	The base-name of the model file could not be determined successfully.
P1010	An invalid start token was found on the command-line. The valid tokens are described in section 3.3.
P1011	The filename passed as the command-line argument to –u was malformed and could not be matched with an existing file.
P1012	The filename passed as the command-line argument to –u could not be matched with an existing file.
P1013	The filename passed as the command-line argument to –u could not be matched with an existing file.
P1014	The XML Schema file for the validation of the user variable XML file was not found. This could be a sign of a damaged installation of the *Vector AUTOSAR Scripting Engine*.
P1015	The command line argument -d could not be parsed.
P1016	The command line argument -d could not be parsed.
P1017	The command line argument -d could not be parsed.
P1018	The command line argument -d could not be parsed.
P1019	The symbol defined via the command line argument -d is already defined in the user variable XML file.
P1020	Unable to set the search path because it was not possible to obtain the path to the binary files of VASE.

Error Code	Explanation
P1021	Unable to set the search path because the path where VASE is installed is too long.
P1022	*No longer used.*
P1023	Invalid path name of script file provided (possible reasons: invalid characters, only white spaces, path name has length 0).
P1024	Required permission to access the script file is not available.
P1025	Path to the script file contains a colon character that is not part of a volume identifier.
P1026	The path of the script file passed to the program is too long.
P1027	The directory of the path to the script file was not found.
P1028	Either the path to the script file specified a directory instead of a file or the permission to access the file was not available.
P1029	The file name within the path to the script file could not be found.
P1030	Out of memory.
P1031	Incorrect path name.
P1032	Internal error: access to secondary executable was not possible.
P1033	Internal error: no permission to access secondary executable.
P1034	Internal error: path name of secondary executable cannot be resolved.
P1035	Internal error: execution of secondary executable failed.
P1036	API token is empty.
P1037	The major version found in the *API compatibility token* is not supported. This could mean that the major version is either higher or lower than interval of supported major versions.
P1038	The API token found in the script file is malformed.
P1039	The API token found in the script file is malformed.
P1040	The API token found in the script file is malformed.
P1041	The major version number of the API token could not be retrieved from analyzing the script file.
P1042	The minor version number of the API token could not be retrieved from analyzing the script file.
P1043	It is not supported to mix the command-line option for passing a single script with the command line options for passing a series of scripts.
P1044	Neither the command-line option for passing a single script nor the command line options for passing a series of scripts was found on the command line. Without a script, an execution is not possible.
P1045	A mixture of API versions was found in script files provided on the command-line. This is not supported.
P1046	It is not supported to mix the command-line option for passing a single script with the command line options for passing a series of scripts.
P1047	Neither the command-line option for passing a single script nor the command line options for passing a series of scripts was found on the command line. Without a script, an execution is not possible.

Error Code	Explanation
P1048	It is not supported to mix the command-line option for passing a single script with the command line options for passing a series of scripts.
P1049	Neither the command-line option for passing a single script nor the command line options for passing a series of scripts was found on the command line. Without a script, an execution is not possible.
P1050	The parsing of command-line arguments failed because arguments with the wrong multiplicity were found.
P1051	The parsing of command-line arguments failed because arguments with the wrong multiplicity were found.
P1052	An issue with the definition of the log verbosity was found. As the result, the logging from within the script execution will effectively be deactivated.
P1053	Unable to set the search path because it was not possible to obtain the path to the binary files of VASE.
P1054	VASE needs to introspectively determine the directory from which it is executed. The returned path was too long.
P1055	An exception happened during the compilation of the Python script.
P1056	An exception occurred during the execution of the Python script.
P1057	A general exception occurred during the processing of the Python script.
P1058	The content of the command line argument -d could not be extracted.
P1059	The parsing of the command line argument -d failed.
P1060	The parsing of the *symbol* part of the command line argument -d failed.
P1061	The parsing of the *value* part of the command line argument -d failed.
P1062	The symbol defined by means of command line argument -d is already defined in a user variable XML file.
P1063	Internal error in the processing of ARXML files passed to VASE.
P1064	Internal error in the processing of ARXML files passed to VASE.
P1065	Error occurred while parsing the command-line arguments.
P1066	Error occurred while parsing the command-line arguments.
P1067	The content of the command line argument -d could not be extracted.
P1068	The parsing of the command line argument -d failed.
P1069	The parsing of the *symbol* part of the command line argument -d failed.
P1070	The parsing of the *value* part of the command line argument -d failed.
P1071	The symbol defined by means of command line argument -d is already defined in a user variable XML file.
P1072	An ARXML file passed to VASE refers to a version of the AUTOSAR XML schema that is not compatible with the usage of API 1.
P1073	VASE failed to extract the version of the AUTOSAR schema from an ARXML file for the purpose of checking compatibility between AUTOSAR XML schema and API usage.

Table 8-1 Error codes used by the main program

8.1.2 User Variable XML File

Error Code	Explanation
X1001	An XML element was expected, but a different type of XML content was found.
X1002	An XML element was expected, but a different type of XML content was found.
X1003	The variable name of a user variable was found empty.
X1004	A warning was raised during the validation of the user variable XML file.
X1005	An error was raised during the validation of the user variable XML file.
X1006	The variable value of a user variable was found empty.
X1007	A duplicate definition of a user variable was found in the user variable XML file.
X1008	An exception was raised during the processing of the user variable XML file.

Table 8-2 Error codes used by the User Variable XML file reader

8.1.3 AUTOSAR Model Abstraction

Error Code	Explanation
M1001	An exception was raised during the loading of a model.
M1002	The filename of a model file passed as a command-line argument does not match with an existing file.
M1003	The applicable XML Schema is invalid. This could be a sign of a damaged installation of the *Vector AUTOSAR Scripting Engine*.
M1004	The URL passed as a schema location is invalid. This could be a sign of a damaged installation of the *Vector AUTOSAR Scripting Engine*.
M1005	A warning was raised during the validation of the saved model ARXML file.
M1006	An error was raised during the validation of the saved model ARXML file.
M1007	A violation of the ARXML file against the AUTOSAR XML Schema was detected while loading an ARXML file into the *Vector AUTOSAR Scripting Engine*. Please note that this will be reported as a warning and the tool workflow will continue. The reason for this behavior is that there is a certain class of invalidities that can be fixed automatically inside the *Vector AUTOSAR Scripting Engine* and therefore there is at least a chance that some invalid configurations can still be handled.
M1008	An exception was raised while trying to delete superfluous output files.
M1009	An exception was raised while trying to delete superfluous output files.

Table 8-3 Error codes used by the model abstraction

8.1.4 Script Execution

Error Code	Explanation
S1001	The existence of an error was indicated by calling the logger.Error() API in the executed script.
S1002	The existence of a warning was indicated by calling the logger.Warning() API in the executed script.

Table 8-4 Error codes used by script execution

9 Return Values

Please note that, in case of an error has been detected, the *Vector AUTOSAR Scripting Engine* will **return the value 1 to the calling shell**. This way it is possible to detect that the execution of the tool ended unsuccessfully and the calling shell can therefore react accordingly.

If the execution of the *Vector AUTOSAR Scripting Engine* can be completed without issues the **value 0 is returned to the calling shell**.

10 Glossary and Abbreviations

10.1 Glossary

Term	Description

10.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
AUTOSAR	Automotive Open Software Architecture
ARXML	AUTOSAR XML
PTVS	Python Tools for Visual Studio
UUID	Universally Unique Identifier
VASE	Vector AUTOSAR Scripting Engine
XML	Extensible Markup Language

11 Contact

Visit our website for more information on

- News
- Products
- Demo software
- Support
- Training data
- Addresses

www.vector.com