AutomationInterface - DaVinci Configurator 6

Development Documentation of the AutomationInterface v3.9.0
DaVinci Configurator v6.3.0

DaVinci Configurator Team

March 11, 2026

Vector Informatik GmbH
Ingersheimerstr. 24
70499 Stuttgart

Contents

1 Release Version Alignment

The table shows the release version alignments of the AutomationInterface and the DaVinci Configurator 6.

- AutomationInterface v3.0.0 - DaVinci Configurator v0.3.1
- AutomationInterface v3.1.0 - DaVinci Configurator v0.4.0
- AutomationInterface v3.2.0 - DaVinci Configurator v0.5.0
- AutomationInterface v3.3.0 - DaVinci Configurator v0.6.0
- AutomationInterface v3.4.0 - DaVinci Configurator v0.7.0
- AutomationInterface v3.5.0 - DaVinci Configurator v6.0.0
- AutomationInterface v3.6.0 - DaVinci Configurator v6.1.0
- AutomationInterface v3.7.0 - DaVinci Configurator v6.2.0
- AutomationInterface v3.8.0 - DaVinci Configurator v6.2.0
- **AutomationInterface v3.9.0 - DaVinci Configurator v6.3.0**

2 Introduction

2.1 General

With the usage of the AutomationInterface of the DaVinci Configurator 6, a user can create scripts, which will be executed in processing component of the DaVinci Configurator 6. AI scripts support many features like:

- Create projects
- Update projects
- Manipulate the data model with access to the whole AUTOSAR model
- Generate code
- Executed repetitive tasks with code, without user interaction
- More

2.2 Facts

Installation The CFG6 can execute user defined scripts out of the box. No additional scripting language installation is required by the customer.

Debugging Support Script projects can be debugged via IntelliJ IDEA. See Automation Build Gradle Documentation.

Code Completion The AutomationInterface supports code completion for Groovy, Kotlin and Java. CodeCompletion is only supported with IntelliJ IDEA.

[1]

[1] See Automation Build Gradle Documentation for details.

3 Getting started with Script Development

3.1 General

This chapter provides a brief introduction to getting started with automation scripting. Please be aware that there are specific coding conventions and limitations to follow when writing scripts.

These are outlined in chapter 5.5 on page 26.

3.2 Automation Script Development Types

The DaVinci Configurator 6 supports two types of automation scripts

- **Script File** (`*.dv.groovy`) it provides the **simplest way** to implement an automation script. When the script gets bigger, you should migrate to a script project.

  To create a script, proceed with chapter 3.4.

  To get more information about script files, proceed with chapter 3.5 on the following page.

- **Script Project** is the more efficient way to create and maintain a script.

  It is the **recommended way to develop** scripts, containing more tasks or multiple classes. It provides IDE support for *Code completion, Syntax highlighting, API Documentation, Debug support, Build support.*

  To create a script, proceed with chapter 3.4.

  To get more information about the script project, proceed with chapter 3.6 on the following page.

3.3 Script

An Automation Script in the DaVinci Configurator 6 automates tasks such as creating and updating projects, manipulating the data model, and generating code, with DaVinci Configurator 6 serving as the execution engine for these scripts.

For details to the script structure, see chapter 5.3 on page 24.

3.4 Script Creation

To create a script, please use the CLI command as shown at 3.1 on the following page.

```
Usage: dvcfg-b automation setup [-h] (-d | -p) <path>

Description:
Set up a script or dvgroovy project environment in the specified location.

Parameters:
*     <path>       The location where the script or dvgroovy project should be
    created.

Options:
* -d, --dvgroovy   Create a dvgroovy project environment.
* -p, --project    Create a script project environment.
  -h, --help       Display the help.
```

Listing 3.1: Creates a script project or script file via CLI

3.5 Script File

The script file is the simplest way to implement an automation script. It could be sufficient for small tasks and if the developer does not require full IDE support during implementing the script.

3.6 Script Project

The script project is the preferred way to develop an automation script, if the content is more than one simple task. A script project is an IDE project (IntelliJ recommended), with compile bindings to the DaVinci Configurator AutomationInterface. It is also called "Automation Script Project" throughout this document.

The DaVinci Configurator 6 will load a script project as a single `*.jar` file. So the script project must be built and packaged into a `*.jar` file before it can be executed by the DaVinci Configurator 6.

Jar Location The Jar location of the build script project is `<ProjectDir>/build/libs`. Gradle will automatically create the directories during the build and will generate the built `*.jar` file.

Prerequisites Before you start, **please make sure** that the following items are available on your system:

- **CFG6**: You need the DaVinci Configurator 6 available on your system.
- **Java JDK**: For the development with the IntelliJ a "Java SE Development Kit 21" (JDK 21) is required. Please install the JDK 21 as described in chapter 3.6.1 on the next page.
- **IDE**: For the script project development the *recommended* IDE is *IntelliJ*. Please install IntelliJ as described in chapter 3.6.2 on the following page.
- **Build system**: To build the script project the build system Gradle is required. See chapter 3.6.2 on page 15 for installation instructions.

3.6.1 Java JDK Setup

Install a JDK 21 on your system. The Java JDK website provides download versions for different systems. Download an appropriate version and make sure you get the x64 version.

The JDK is needed for the Java Compiler for IntelliJ and Gradle.

3.6.2 IntelliJ IDEA Setup

Install IntelliJ on your system. The IntelliJ IDEA website provides download versions for different applications.

Code completion and compilation additionally require that the Project SDK is set. Therefore, open the **File -> Project Structure** Dialog in IntelliJ and switch to the settings dialog for Project. If not already available, set an appropriate option for the Project SDK. Please set the value to a valid Java JDK (see 3.6.1).

Note: Do not select a JRE.

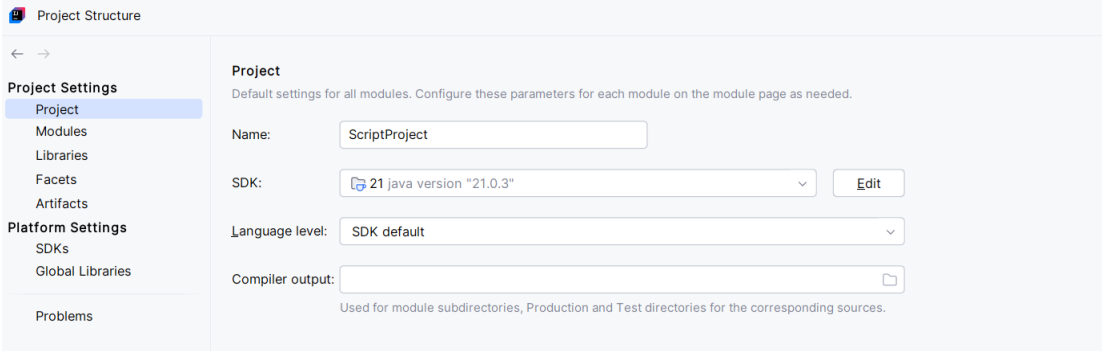


Figure 3.1: Project SDK Setting

To enable building of projects, ensure that the Gradle JVM is set.Therefore, open the **File -> Settings** Dialog in IntelliJ and find the settings dialog for Gradle.If not already available, set an appropriate option for the Gradle JVM. Please set the value to Project SDK to use the selected SDK above.

Note: Do not select a JRE.

If you do not have the Gradle settings, please make sure that the Gradle plugin inside of IntelliJ is installed. Open the **File -> Settings** Dialog then Plugins and select the Gradle plugin.

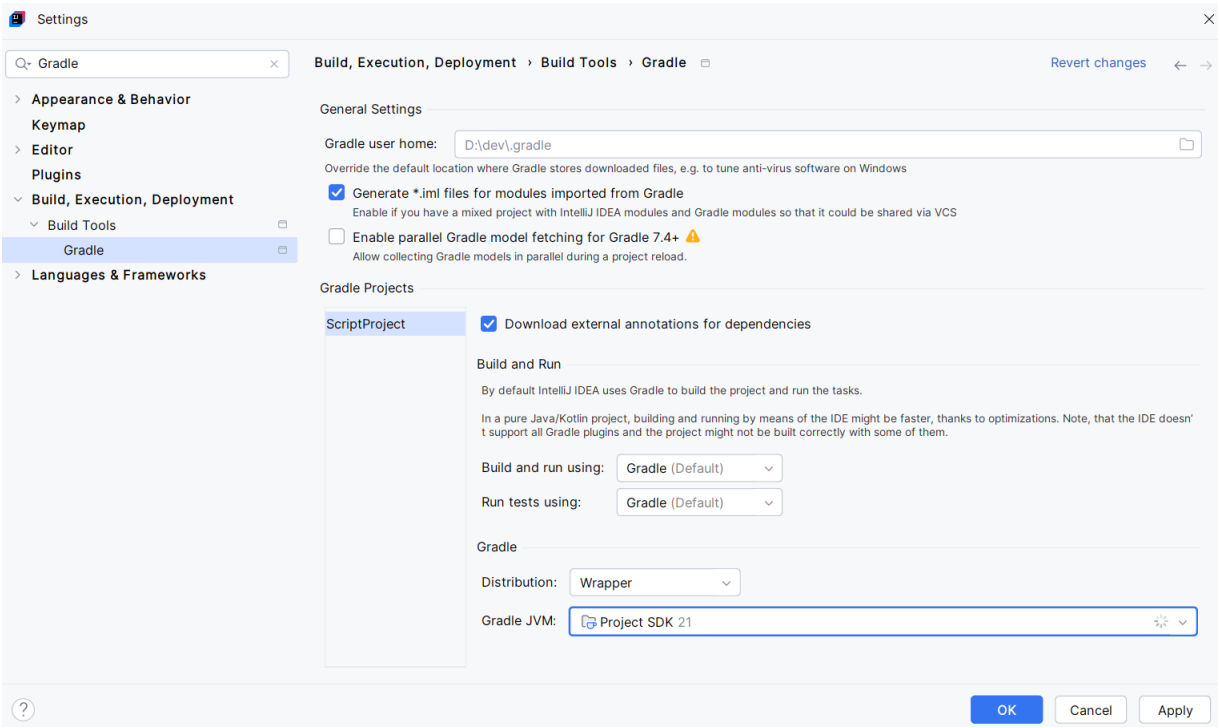


Figure 3.2: Gradle JVM Setting

Build System As build system Gradle is used.If you use IntelliJ as your IDE you can open the gradle view, **View -> Tool Windows -> Gradle**, to find the required gradle build tasks.

If your system has internet access, you can use the default Gradle Build System provided by the DaVinci Configurator 6. In this case, you **do not** have to install Gradle. If you are a Vector internal user, you could also **skip** the Gradle installation.

If you want to use your own Gradle Build System install it on your system. The Gradle website provides the required download version for the Gradle Build System.

Please **download the version 9.3.1**.

See the Automation Build Gradle Documentation for more details on the Build System.

3.7 Script Testing

To ensure the correctness and stability of your automation scripts, testing is highly recommended.

Details on how to write and execute tests can be found in the Automation Build Gradle Documentation.

3.8 Script Debugging

There are two ways to debug your script:

- Using the Testing Framework (recommended).
- Using Remote Debugging.

3.9 Script Location

A script location is a directory that contains built script projects `*.jar` or script files `*.dv.groovy`.

For more information, proceed with chapter 5.3.2 on page 25.

3.10 Kotlin Support

If you want to develop your scripts in Kotlin, just add the Kotlin JVM and Kotlin SAM-With-Receiver Plugins like in the snippet below:

```
plugins {
    // Other plugins...
    id "org.jetbrains.kotlin.jvm" version "$KT_VERSION"
    id "org.jetbrains.kotlin.plugin.sam.with.receiver" version "$KT_VERSION"
}
```

Listing 3.2: Application of relevant Kotlin plugins

You can then create a class that implements IScriptFactory in `src/main/kotlin`:

```
import com.vector.cfg.automation.scripting.api.IScriptCreationApi
import com.vector.cfg.automation.scripting.api.IScriptFactory
import com.vector.cfg.automation.scripting.api.IScriptTaskTypeApi.DV_APPLICATION

class MyKotlinScript : IScriptFactory {
    override fun createScript(creationApi: IScriptCreationApi) {
        creationApi.scriptTask("TaskName", DV_APPLICATION) {
            code {
                // Task execution code here
            }
        }
    }
}
```

Listing 3.3: MyKotlinScript.kt in /src/main/kotlin

Remarks:

- Nullability info on PAI API will be enhanced in the future. This may reveal missing null checks when updating the targeted CFG6 version
  - You can turn these errors into warnings by adding the following snippet in your build.gradle file:

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompile

tasks.withType(KotlinCompile).configureEach {
    it.compilerOptions.freeCompilerArgs.add("-Xnullability-annotations=@org.
        jspecify.annotations:warn")
}
```

Listing 3.4: Turn Nullability Errors into Warnings in build.gradle

- Mixing Groovy and Kotlin source files in a single script project is not natively supported by Gradle. To be able to use your Kotlin classes from your Groovy code, you can add the following snippet in your build.gradle file:

```
tasks.named("compileGroovy", GroovyCompile) {
    it.classpath += files(sourceSets.main.kotlin.classesDirectory)
}
```

Listing 3.5: Allow Groovy code to call Kotlin code in build.gradle

If you want to call Groovy code from your Kotlin code, you need to add the following snippet in your build.gradle file:

```
tasks.named("compileGroovy") {
    classpath = sourceSets.main.compileClasspath
}

tasks.named("compileKotlin") {
    dependsOn("compileGroovy")
    libraries.from(files(sourceSets.main.groovy.classesDirectory))
}
```

Listing 3.6: Allow Kotlin code to call Groovy code in build.gradle

Note: These snippets are mutually exclusive. You can only use one of them depending on the direction of the calls.

3.11 Logging Configuration

Logs intended for the user can be seen in the CLI output. But here, not all detailed development logs are visible. To view detailed logs of the CFG6 ConfigCore, you can find them in the following folder:

- Windows: `%LocalAppData%\Vector\DaVinci\dvcfg\logs`
- Linux: `$XDG_STATE_HOME/Vector/DaVinci/dvcfg/logs`

The Log4J configuration file specifies which logging level is assigned to each class or package. The logging level is applied to the specified element and automatically inherited by all child packages and classes.

The logging configuration of the DaVinci Configurator 6 can be extended with custom Log4J configuration file fragments.

```
<Configuration>
    <Loggers>
        <Logger name="your.generator" level="trace">
            <AppenderRef ref="LiveLogAppender"/>
        </Logger>
    </Loggers>
</Configuration>
```

Listing 3.7: Log4J configuration fragment example

- The logger for the custom `your.generator` package is set to `trace`.
  - The `LiveLogAppender` is attached to the `your.generator` logger, which additionally prints its output to the CLI console.

NOTE: The CLI console output will never show log messages below the `info` level, those are only available in the DaVinci Configurator 6 core logfile (see 3.11).

The `monitorInterval` attribute can be set on the `Configuration` element to enable automatic reloading of Log4J configuration changes while ConfigCore is running.

Log4J configuration fragments can be applied using one or both of the following methods:

- Place the configuration in the `<user home directory>/.DaVinciCfg/log4j.xml` file, and/or
- Specify the configuration file path in the `DVCFG_LOGGING_CONFIG` environment variable.

3.11.1 scriptLogger

The `scriptLogger` is configured with the `info` level by default and can be reconfigured with the following configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
    <Loggers>
        <Logger name="com.vector.cfg.user.com.vector.cfg.automation.script" level=
            "debug">
        </Logger>
    </Loggers>
</Configuration>
```

Listing 3.8: Sample of custom log4j.xml to adjust log level

In the following example, only messages printed via `scriptLogger.info(...)` and `scriptLogger.debug(...)` will be logged to the detailed log file (see 3.11 on the previous page). Messages printed via `scriptLogger.trace(...)` will be ignored.

```
import static com.vector.cfg.automation.api.ScriptApi.*
daVinci {
   scriptTask("Task1", DV_APPLICATION) {
      code {
           scriptLogger.info "!!!First info message!!!"
           scriptLogger.debug "!!!First debug message!!!"
           scriptLogger.trace "!!!First trace message!!!"
      }
   }
}
```

Listing 3.9: Script with log messages at different levels

The CLI console output will show the following entries:

```
[INFO]  Executing script task "script:Task1"
[INFO]  !!!First info message!!!
[INFO]  Execution of script task "script:Task1" successfully finished.
```

Listing 3.10: CLI Log

The DaVinci Configurator 6 log file will contain the following entries:

```
INFO  script                          - Executing script task "script:Task1"
INFO  script                          - !!!First info message!!!
DEBUG script                          - !!!First debug message!!!
INFO  script                          - Execution of script task "script:Task1"
    successfully finished.
```

Listing 3.11: DaVinci Configurator 6 Log

4 Command Line Usage

With the Command Line Interface (CLI) it is possible can manage and execute AutomationInterface scripts directly from your terminal. This chapter summarizes the most important CLI commands.

4.1 List Script Task

To list all script tasks, please use the CLI command as shown at 4.1.

```
Usage: dvcfg-b automation list [-h] -b=<folder> [-p=<file>] [-l=<location>[,<
    location>...]]... [--no-save]

Description:
List automation tasks from a specific project, bsw-package, location, or from all
    sources.

Options:
* -b, --bsw-package=<folder>                    Directory of the BSW package.
  -p, --project=<file>                          The .dvjson file of the project.
  -l, --location=<location>[,<location>...]     List of folders containing script
      files.
                                                E.g.: -l .\\locationA,.\\locationB
      --no-save                                 Prevent saving the project to disk.
  -h, --help                                    Display the help.
```

Listing 4.1: CLI command to list all script tasks of the current DaVinci Configurator session

4.2 Run Script Task

To execute a script task, please use the CLI command as shown at 4.2 on the next page.

Note: In case that the script tasks modifications shall not be saved to the project, you can use the `-no-save` option.

```
Usage: dvcfg-b automation run [-h] -b=<folder> [-p=<file>] -t=<task>[,<task>...]
    [-t=<task>[,<task>...]]... [-a=<arg>]...
                              [-l=<location>[,<location>...]]... [--debugger[=<port
                                  >]] [--no-save]

Description:
Run automation tasks with arguments.

Options:
* -b, --bsw-package=<folder>                    Directory of the BSW package.
  -p, --project=<file>                          The .dvjson file of the project.
* -t, --task=<task>[,<task>...]                 List of script tasks to execute.
                                                E.g.: -t task1,task2
  -a, --arg=<arg>                               Define a set of arguments specific
      to a single task.
                                                E.g.: -a 'task1' -a '--name=str1 --
                                                    value=1' -a 'task2' -a '-i 1,2,3
                                                    '
  -l, --location=<location>[,<location>...]     List of folders containing script
      files.
                                                E.g.: -l .\\locationA,.\\locationB
      --debugger[=<port>]                       Enable debugging for the ConfigCore
          instance.
                                                If no port is specified, uses
                                                    default port 5005.
                                                E.g.: --debugger or --debugger=5006
      --no-save                                 Prevent saving the project to disk.
  -h, --help                                    Display the help.
```

Listing 4.2: Run a script task via CLI

You can modify the implementation according to your needs. For the AutomationInterface API Reference see chapter 6 on page 29.

4.3 Add Script Location To Project

To add a script location to DaVinci Configurator 6 project, please use the CLI command as shown at 4.3.

```
Usage: dvcfg-b automation add [-h] -p=<file> -b=<folder> -l=<location>[,<location
    >...] [-l=<location>[,<location>...]]...
                              [--no-save]

Description:
Add locations containing automation tasks.

Options:
* -p, --project=<file>                          The .dvjson file of the project.
* -b, --bsw-package=<folder>                    Directory of the BSW package.
* -l, --location=<location>[,<location>...]     List of folders containing script
    files.
                                                E.g.: -l .\\locationA,.\\locationB
      --no-save                                 Prevent saving the project to disk.
  -h, --help                                    Display the help.
```

Listing 4.3: Adds a script location to a DaVinci Configurator project via CLI

4.4 Remove Script Location From Project

To remove a script location from a DaVinci Configurator 6 project, please use the CLI command as shown at 4.4.

```
Usage: dvcfg-b automation remove [-h] -p=<file> -b=<folder> -l=<location>[,<
    location>...] [-l=<location>[,
                                <location>...]]... [--no-save]

Description:
Remove locations containing automation tasks.

Options:
* -p, --project=<file>                          The .dvjson file of the project.
* -b, --bsw-package=<folder>                    Directory of the BSW package.
* -l, --location=<location>[,<location>...]     List of folders containing script
    files.
                                                E.g.: -l .\\locationA,.\\locationB
      --no-save                                 Prevent saving the project to disk.
  -h, --help                                    Display the help.
```

Listing 4.4: Removes a script location to a DaVinci Configurator project via CLI

5 AutomationInterface Architecture

5.1 Components

The DaVinci Configurator 6 consists of three components:

- Core Components
- AutomationInterface (AI) - also called Automation API
- Scripting Engine

The other part is the script provided by the user.

The scripting engine will load the script, and the script uses the AutomationInterface to perform tasks. The AutomationInterface will translate the requests from the script into Core component calls.

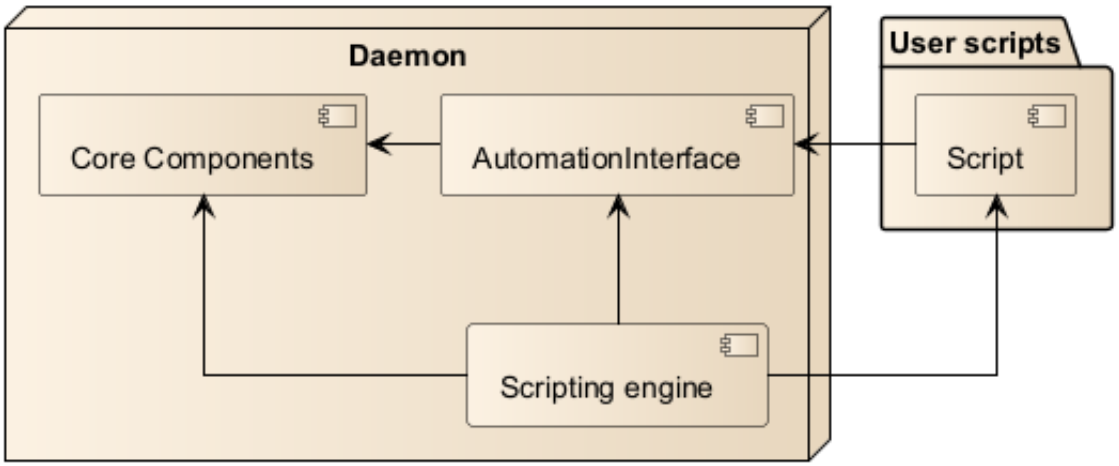


Figure 5.1: DaVinci Configurator components and interaction with scripts

The separation of the AutomationInterface and the Core components has multiple benefits:

- Stable API for script writers
  - Including checks, that the API will not break in following releases
- Well defined and documented API
- Abstraction from the internal heavy lifting
  - This eases the usage for the user, because the automation interfaces are tailored to the use cases.

PublishedApi All AutomationInterface classes are marked with a special annotation to **highlight** the fact that it is part of the published API. The annotation is called `@PublishedApi`.

So every class marked with `@PublishedApi` can be used by the client code. But if a class is **not** marked with `@PublishedApi` or is marked with `@Deprecated` it should not be used by any client code, nor shall a client call methods via reflection or other runtime techniques.

You should **not** access DaVinci Configurator private or package private classes, methods or fields.

5.2 Languages

The DaVinci Configurator provides out of the box language support for:

- Java
- Groovy
- Kotlin

The recommended scripting language is **Groovy** which shall be preferred by all users.

5.3 Script Structure

A script always contains one or more script tasks. A script is represented by an instance of `IScript`, the contained tasks are instances of `IScriptTask`.

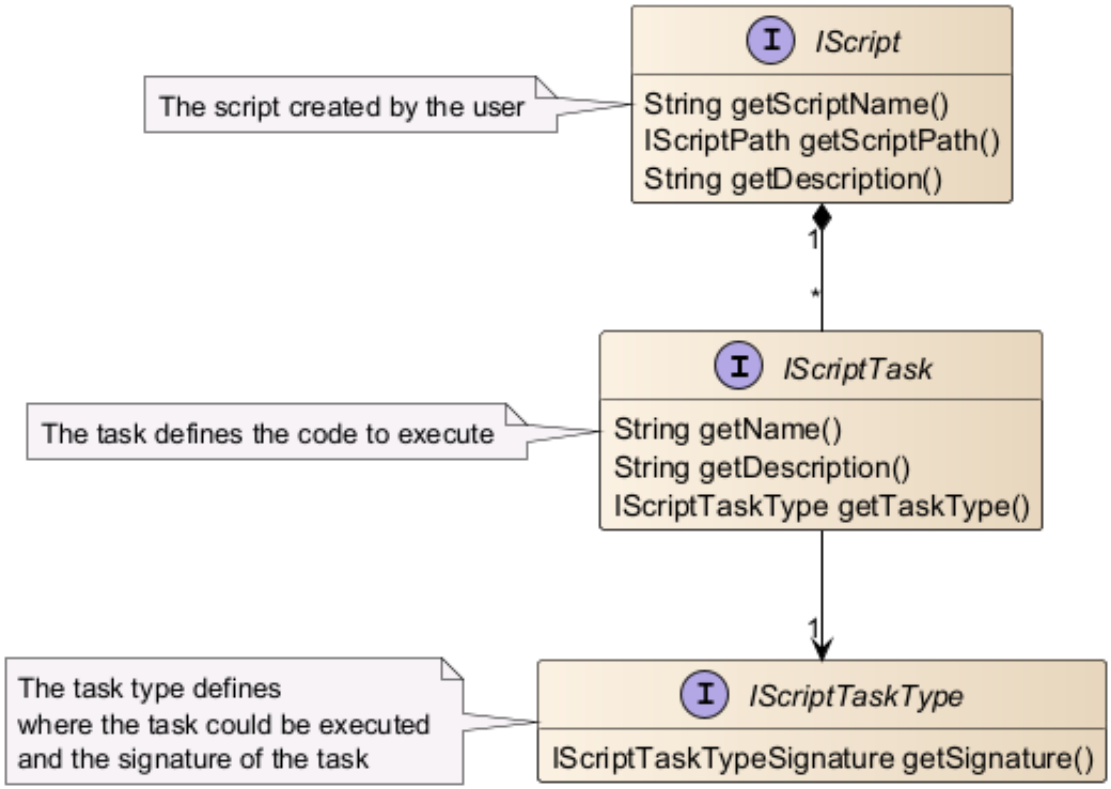


Figure 5.2: Structure of scripts and script tasks

You create the `IScript` and `IScriptTask` instance with the API described in chapter 6.2 on page 30.

The script task type (`IScriptTaskType`) defines where the task could be executed. It also defines the signature of the task's `code {}` block. See chapter 6.3 on page 34 for the available script task types.

5.3.1 Script Tasks

Script tasks are the executable units of scripts, which are executed at certain points in the DaVinci Configurator (specified by the `IScriptTaskType`). Every script task has a `code {}` block, which contains the logic to execute.

5.3.2 Script Locations

Script locations define where script files are loaded from. The script locations can be defined by the user, but there are also pre-set script locations.

Pre-set script locations

- BSW-Package based
  - <BSW-PKG>/Components/<MSN>/AutomationDvC6

User definable script locations

- Project based
  - via CLI from anywhere on the file system

5.4 Script loading

All scripts contained in the script locations are automatically loaded by the DaVinci Configurator. If new scripts are added to script locations these scripts are automatically loaded.

If a script changes during runtime of the DaVinci Configurator the whole script is reloaded and then executable, without a restart of the tool or a reload of the project.

This enables script development during the runtime of the DaVinci Configurator

- No project reload
- No tool restart
- Faster feedback loops

Note: The *.jar artifact file from a script project *should be updated by the Gradle build system*, not by hand. Because the Java VM is holding a lock to the file. If you try to replace the file in the explorer you will get an error message.

5.4.1 Internal Script Reload Behavior

Your script can be loaded and unloaded automatically multiple times during the execution of the DaVinci Configurator. More precise, when a script is currently not used and there are memory constraints your script will be automatically unloaded.

If the script will be executed again, it is automatically reloaded and then executed. So it is possible that the script initialization code is called multiple times in the DaVinci Configurator lifecycle. But this is no issue, because the script and the tasks **shall not** have any internal state during initialization.

Memory Leak Prevention The feature above is implemented to prevent leaking memory from an automation script into the DaVinci Configurator memory. So when the memory run low, all unused scripts are unloaded, which will also free leaked memory of scripts.

But this **does not** mean that is impossible to construct memory leaks from an automation script. E.g. Open file handles without closing them will still cause a memory leak.

5.5 Script Coding Conventions and Constraints

This section describes conventions, which you are advised to apply.

Requirement Levels - Wording

- Shall: This word, or the terms "Mandatory", "Required" or "Must", mean that the rule or convention is an absolute requirement.
- Shall not: This word, or the terms "Must not" mean that the rule or convention is an absolute prohibition.
- Should: This word, or the adjective "Recommended", mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.
- Should not: This phrase, or the phrase "Not recommended" mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.
- May: This word, or the adjective "Optional", mean that an item is truly optional.

See also "RFC 2119: Key words for use in RFCs to Indicate Requirement Levels"[1].

5.5.1 Usage of static fields

You **shall not** use any static fields in your script code or other written classes inside of your project. Except `static final` constants of simple immutable types like (normally compile time constants):

- int
- boolean
- double
- String
- ...

Static fields will cause memory leaks, because the fields are not garbage collected. Example:

[1] `https://www.ietf.org/rfc/rfc2119.txt`

```
scriptTask("Name") {
    code {
        MyClass.leakVariable.add("Leaked Memory")
    }

}

class MyClass {
    static List leakVariable = []
}
```

Listing 5.1: Static field memory leak

The use of static fields of the AutomationInterface is not allowed.

5.5.2 Usage of Outer Closure Scope Variables

The same static field rule applies to variables passed from outer `Closure` scopes into a script task `code{}` block. You **shall not** cache/save data into such variables.

Example:

```
scriptTask("Name"){
 def invalidVariable = [] //List

  code{
    invalidVariable.add("Leaked Memory")
  }

}
```

Listing 5.2: Memory leak with closure variable

5.5.3 States over script task execution

You **shall not** hold or save any states over multiple script task executions in your classes.

The script task should be state less. All states are provided by the Automation API or the data models.

If you need to cache data over multiple executions, see chapter 6.4.10 on page 56 for a solution.

5.5.4 Multithreading Support

A script task **shall not** create any `Thread, Executor, ThreadPool or ForkJoinPool` instances. Multithreading in automation scripts is not supported. Using multiple threads in automation scripts may lead to unexpected side effects, such as issues with live logging and debugging. If parallel execution is needed, different script tasks can be executed using multiple CLI instances.

5.5.5 Usage of DaVinci Configurator private Classes Methods or Fields

A script task **should not** call or rely on any non published API or private (also package private) classes, methods or fields. You also should not use any reflection techniques to reflect about Configurator internal APIs. Otherwise it is not guaranteed that your script will work with other DaVinci Configurator versions. See 5.1 on page 23 for details about PublishedApi.

But it is valid to use reflection for your own script code.

6 AutomationInterface API Reference

6.1 Introduction

This chapter contains the description of the DaVinci Configurators - AutomationInterface. The figure 6.1 shows the APIs and the containment structure of the different APIs.

The components have a hierarchical order, where and when the components are usable. When a component is contained in another the component is only usable, when the other is active.

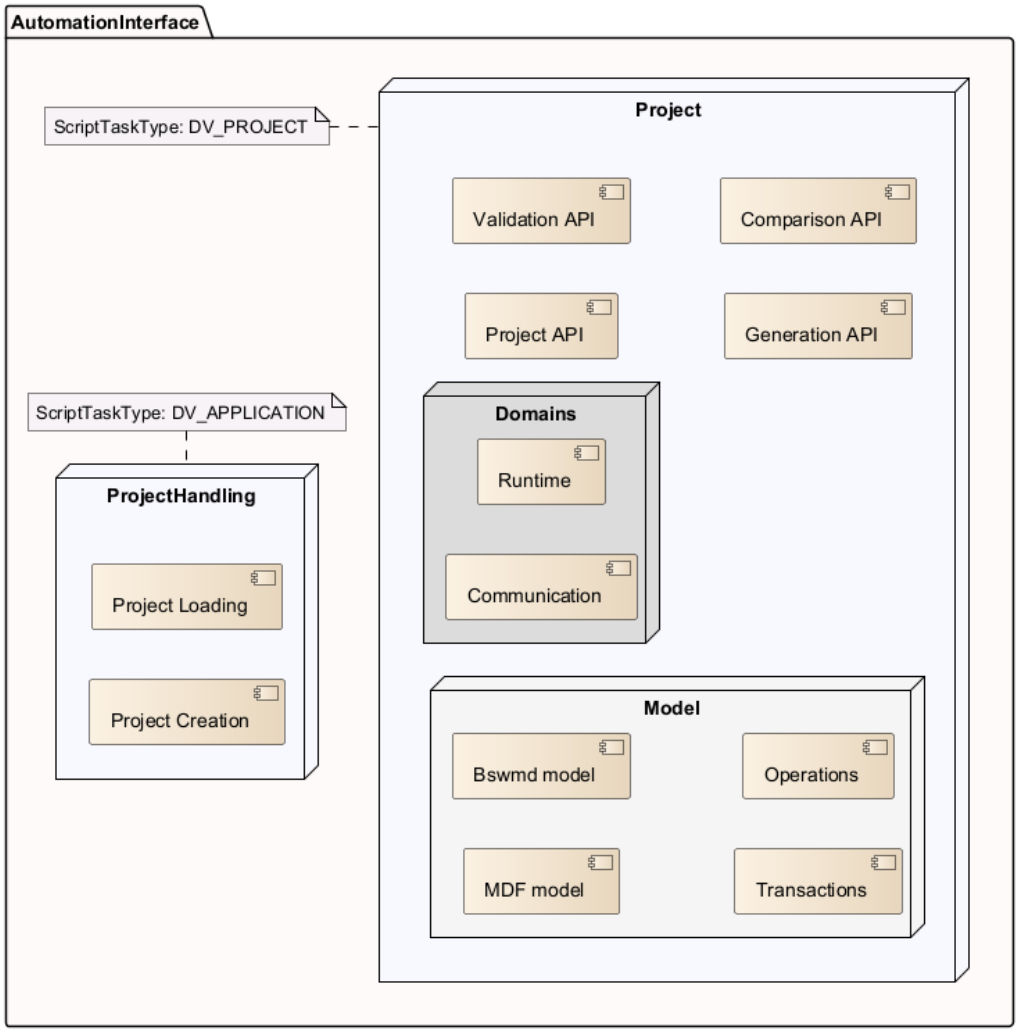


Figure 6.1: The API overview and containment structure

The components have a hierarchical order, where and when the components are usable. When a component is contained in another the inner component is only usable, if the outer component is active.

Usage examples:

- The Generation API is only usable inside a loaded project
- The Project creation API is only usable outside a loaded project

6.2 Script Creation

This section lists the APIs to create, execute and query information for script tasks. The sections document the following aspects:

- Script task creation
- Description and help texts
- Task executable query

6.2.1 Script Task Creation

To create a script task you have to call one of the `scriptTask()` methods. The last parameter of the `scriptTask` methods can be used to set additional options of the task. Every script task needs one `IScriptTaskType`. See chapter 6.3 on page 34 for all available task types.

The `code{ }` block is **required** for every `IScriptTask`. The block contains the code, which is executed when the task is executed.

Script Task with default Type The method `scriptTask()` will create a script task. If no script task type is given, the `IScriptTaskType DV_PROJECT` is used.

```
scriptTask("TaskName"){
    code{
        // Task execution code here
    }
}
```

Listing 6.1: Task creation with default type

Script Task with Task Type You could also define the used `IScriptTaskType` at the `scriptTask()` methods. The methods

- `scriptTask(String, IApplicationScriptTaskType, Action)`
- `scriptTask(String, IProjectScriptTaskType, Action)`

will create an script task for passed `IScriptTaskType`. The two methods differentiate, if a project is required or not. See chapter for all available task types 6.3 on page 34

```
scriptTask("TaskName", DV_APPLICATION){
    code{
        // Task execution code here
    }
}
```

Listing 6.2: Task creation with TaskType Application

```
scriptTask("TaskName", DV_PROJECT){
    code{
        // Task execution code here
    }
}
```

Listing 6.3: Task creation with TaskType Project

Multiple Tasks in one Script It is also possible to define multiple tasks in one script.

```
scriptTask("TaskName"){
    code{ }
}

scriptTask("SecondTask"){
    code{ }
}
```

Listing 6.4: Define two tasks is one script

6.2.1.1 Script Creation with IDE Code Completion Support

Due to the fact that the IDE can not know which API is available inside of a script file, a glue code is needed to tell the IDE, what API is callable inside of a script file.

The `ScriptApi.daVinci(Action)` method enables the IDE code completion support in a script file. You have to write the `daVinci{ }` block and inside of the block the code completion is available. The following sample shows the glue code for the IDE:

```
import static com.vector.cfg.automation.api.ScriptApi.*

//daVinci enables the IDE code completion support
daVinci{

    // Normal script code here
    scriptTask("TaskName"){
        code{
            // Script task execution code here
        }
    }
}
```

Listing 6.5: Script creation with IDE support

The `daVinci{}` block is only required for code completion support in the IDE. It has no effect during runtime, so the `daVinci{}` is optional in script files (`.dv.groovy`)

6.2.1.2 Script Task isExecutableIf

You can set an `isExecutableIf` handler, which is called before the `IScriptTask` is executed. The code can evaluate, if the `IScriptTask` shall be executable. If the handler returns `true`, the code of the `IScriptTask` is executable, otherwise `false`. See class `IExecutableTaskEvaluator` for details.

The `Closure` isExecutable has to return a `boolean`. The passed arguments to the closure are the same as the `code{ }` block arguments.

Inside of the `Closure` a property `notExecutableReasons` is available to set reasons why it is not executable. It is highly recommended to set reasons, when the `Closure` returns `false`.

```
scriptTask("TaskName"){

    isExecutableIf{ taskArgument ->
        // Decide, if the task shall be executable
        if(taskArgument == "CorrectArgument"){
            return true
        }
        notExecutableReasons.addReason "The argument is not 'CorrectArgument'"
        return false
    }

    code{ taskArgument ->
        // Task execution code here
    }
}
```

Listing 6.6: Task with isExecutableIf

6.2.2 Description and Help

Script Description The script can have an optional description text. The description shall list what this script contains. The method `scriptDescription(String)` sets the description of the script.

The description shall be a short overview. The `String` can be multiline.

```
// You can set a description for the whole script
scriptDescription "The Script has a description"

scriptTask("Task"){
    code{}
}
```

Listing 6.7: Script with description

Task Description A script task can have an optional description text. The description shall help the user of the script task to understand what the task does. The method `taskDescription(String)` sets the description of the script task.

The description shall be a short overview. The `String` can be multiline.

```
scriptTask("TaskName"){
    taskDescription "The description of the task"

    code{ }
}
```

Listing 6.8: Task with description

Task Help A script task can also have an optional help text. The help text shall describe in detail what the task does and when it could be executed. The method `taskHelp(String)` sets the help of the script task.

The help shall be elaborate text about what the task does and how to use it. The `String` can be multiline.

The help text is automatically expanded with the help for user defined script task arguments, see `IScriptTaskBuilder.newUserDefinedArgument(String, Class, String)`.

```
scriptTask("TaskName"){
  taskDescription "The short description of the task"
  taskHelp """
           The long help text
           of the script with multiple lines

           And paragraphs ...
           """.stripIndent()
  // stripIndent() will strip the indentation of multiline strings
  // The three """ are needed, if you want to write a multiline string

    code{ }
}
```

Listing 6.9: Task with description and help text

6.3 Script Task Types

The `IScriptTaskType` instances define where a script task is executed in the DaVinci Configurator. The types also define the arguments passed to the script task execution and what return type an execution has.

Every script task needs an `IScriptTaskType`. The type is set during creation of the script tasks.

Interfaces All task types implement the interface `IScriptTaskType`. The following figure show the type and the defined sub types:

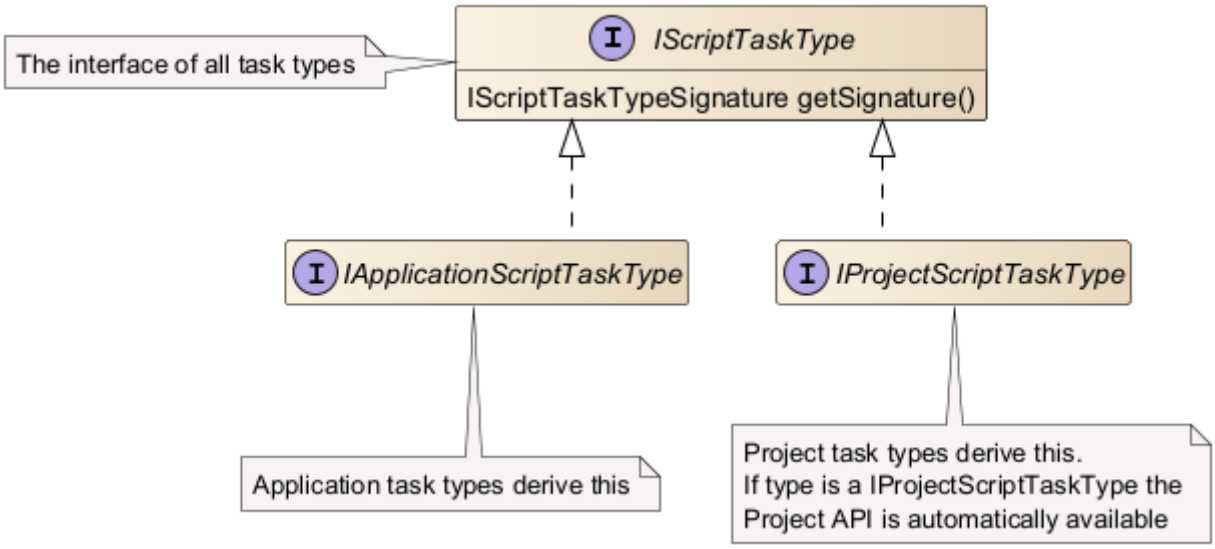


Figure 6.2: IScriptTaskType interfaces

6.3.1 Available Types

The class `IScriptTaskTypeApi` defines all available `IScriptTaskTypes` in the DaVinci Configurator. All task types start with the prefix `DV_`.

`None` at parameters and return types mean, that any arguments could be passed and return to or from the task. Normally it will be nothing. The arguments are used, when the task is called in unit tests for example.

ScriptTaskType input parameters can get accessed via script by adding them to the code block e.g. `code{ List<Object> inputParameter -> inputParameter }`

6.3.1.1 Application Types

Application The type `DV_APPLICATION` is for application wide script tasks. A task could create/open/close/update projects. Use this type, if you need full control over the project handling, or you want to handle multiple project at once.

Name	Application
Code identifier	DV_APPLICATION
Task type interface	`IApplicationScriptTaskType`
Parameters	None
Return type	None
Execution	Standalone

6.3.1.2 Project Types

Project The type `DV_PROJECT` is for project script tasks. A task could access the currently loaded project. Manipulate the data, generate and save the project. This is the default type, if no other type is specified.

Name	Project
Code identifier	DV_PROJECT
Task type interface	`IProjectScriptTaskType`
Parameters	None
Return type	None
Execution	Standalone

Module activation The type `DV_ON_MODULE_ACTIVATION` allows the script to hook any Module Activation in a loaded project. Every `DV_ON_MODULE_ACTIVATION` task is automatically executed, when an "Activate Module" operation is executed. The script task is called after the module was created.

Name	Module activation
Code identifier	DV_ON_MODULE_ACTIVATION
Task type interface	`IProjectScriptTaskType`
Parameters	`MIModuleConfiguration moduleConfiguration`
Return type	`Void`
Execution	Automatically during module activation

Module deactivation The type `DV_ON_MODULE_DEACTIVATION` allows the script to hook any Module Deactivation in a loaded project. Every `DV_ON_MODULE_DEACTIVATION` task is automatically executed, when an "Deactivate Module" operation is executed. The script task is called before the module is deleted.

Name	Module deactivation
Code identifier	DV_ON_MODULE_DEACTIVATION
Task type interface	`IProjectScriptTaskType`
Parameters	`MIModuleConfiguration moduleConfiguration`
Return type	`Void`
Execution	Automatically during module deactivation

6.3.1.3 UI Types

Editor selection The type `DV_EDITOR_SELECTION` allows the script task to access the currently selected element of an editor. The task is executed in context of the selection and is not callable by the user without an active selection.

Name	Editor selection
Code identifier	DV_EDITOR_SELECTION
Task type interface	`IProjectScriptTaskType`
Parameters	`MIObject selectedElement`
Return type	`Void`
Execution	In context menu of an editor selection

Editor multiple selections The type `DV_EDITOR_MULTI_SELECTION` allows the script task to access the currently selected elements of an editor. The task is executed in context of the selection and is not callable by the user without an active selection. The type is also usable when the `DV_EDITOR_SELECTION` apply.

Name	Editor multiple selections
Code identifier	DV_EDITOR_MULTI_SELECTION
Task type interface	`IProjectScriptTaskType`
Parameters	`List<MIObject> selectedElements`
Return type	`Void`
Execution	In context menu of an editor selection

Those ScriptTaskTypes can be executed via selecting Configuration Elements in the editor. See usage: 6.3

`Example:` Select at least two configuration elements in the editor.

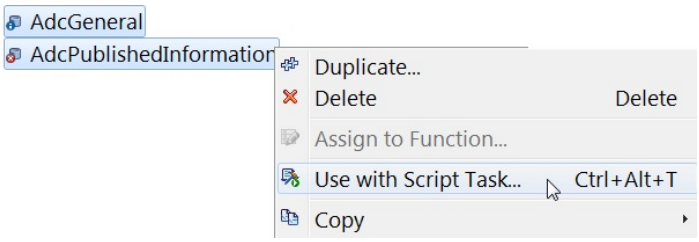


Figure 6.3: Access Editor Selection

```
scriptTask("EditorMultiSelectionTask", DV_EDITOR_MULTI_SELECTION) {

    // Add the selected elements as closure parameters
    code { List<MIObject> selectedElements ->
        selectedElements
    }
}
```

Listing 6.10: Usage of ScriptTaskType: DV_EDITOR_MULTI_SELECTION

6.3.1.4 Generation Types

Generation Step The type `DV_GENERATION_STEP` defines that the script task is executable as a GenerationStep during generation. The user has to explicitly create an GenerationStep in the Project Settings Editor, which references the script task.

Name	Generation Step
Code identifier	DV_GENERATION_STEP
Task type interface	`IProjectScriptTaskType`
Parameters	`EGenerationPhaseType phase`
	`EGenerationProcessType processType`
	`IValidationResultSink resultSink`
Return type	`Void`
Execution	Selected as GenerationStep in GenerationProcess

See chapter 6.7.2 on page 132 for usage samples.

Generation Process Start The type `DV_ON_GENERATION_START` defines that the script task is automatically executed when the generation is started.

Name	Generation Process Start
Task type interface	`IProjectScriptTaskType`
Code identifier	DV_ON_GENERATION_START
Parameters	`List<EGenerationPhaseType> generationPhases`
	`List<IGenerator> executedGenerators`
Return type	`Void`
Execution	Automatically before GenerationProcess

See chapter 6.7.2 on page 132 for usage samples.

Generation Process End The type `DV_ON_GENERATION_END` defines that the script task is automatically executed when the generation has finished.

Name	Generation Process End
Code identifier	DV_ON_GENERATION_END
Task type interface	`IProjectScriptTaskType`
Parameters	`EGenerationProcessResult processResult`
	`List<IGenerator> executedGenerators`
Return type	`Void`
Execution	Automatically after GenerationProcess

See chapter 6.7.2 on page 132 for usage samples.

6.4 Script Task Execution

This section lists the APIs to execute and query information for script tasks. The sections document the following aspects:

- Script task execution
- Logging API
- Path resolution
- Error handling
- User defined classes and methods
- User defined script task arguments

6.4.1 Execution Context

Every `IScriptTask` could be executed, and retrieve passed arguments and other context information. This execution information of a script task is tracked by the `IScriptExecutionContext`.

The `IScriptExecutionContext` holds the context of the execution:

- The script task arguments
- The current running script task
- The current active script logger
- The active project, if existing
- The script temp folder
- The script task user defined arguments

The `IScriptExecutionContext` is also the entry point into every automation API, and provide access to the different API classes. The classes are described in their own chapters like `IProjectHandlingApiEntryPoint`.

The context is immediately active, when the code block of an `IScriptTask` is called.

Groovy Code The client sample illustrates the seamless usage of the `IScriptExecutionContext` class in Groovy:

```
scriptTask("taskName", DV_APPLICATION){
  code{  // The IScriptExecutionContext is automatically active here
     // Call methods of the IScriptExecutionContext
     def logger = scriptLogger
     def temp = paths.tempFolder

     // Use an automation API
     generation{
         // Now the Generation API is active
     }
  }
}
```

Listing 6.11: Access automation API in Groovy clients by the IScriptExecutionContext

In Groovy the `IScriptExecutionContext` is automatically activated inside the `code{}` block.

Java Code For Java clients the method `IScriptExecutionContext.getInstance(Class)` provides access to the API classes, which are seamlessly available for the groovy clients:

```
// Java code
// Passed from the script task:
IScriptExecutionContext scriptContext = ...;

// Retrieve automation API in Java
IGenerationApi generation = scriptContext.getInstance(IGenerationApiEntryPoint.
    class).getGeneration();

// In groovy code it would be:
generation{

}
```

Listing 6.12: Access to automation API in Java clients by the IScriptExecutionContext

In Java code the context is always the first parameter passed to every task code (see `IScriptTaskCode`).

6.4.1.1 Code Block Arguments

The code block can have arguments passed into the script task execution. The arguments passed into the `code{ }` block are defined by the `IScriptTaskType` of the script task. See chapter 6.3 on page 34 for the list of arguments (including types) passed by each individual task type.

```
scriptTask("Task"){
  code{ arg1, arg2, ... -> // arguments here defined by the IScriptTaskType

  }
}

scriptTask("Task2"){
  // Or you could specify the type of the arguments for code completion
  code{ String arg1, List<Double> arg2 ->
  }
}
```

Listing 6.13: Script task code block arguments

The arguments can also can be retrieved with `IScriptExecutionContext.getScriptTaskArguments()`.

6.4.2 Task Execution Sequence

The figure 6.4 shows the overview sequence when a script task gets executed by the user and the interaction with the `IScriptExecutionContext`. Note that the context gets created each time the task is executed.

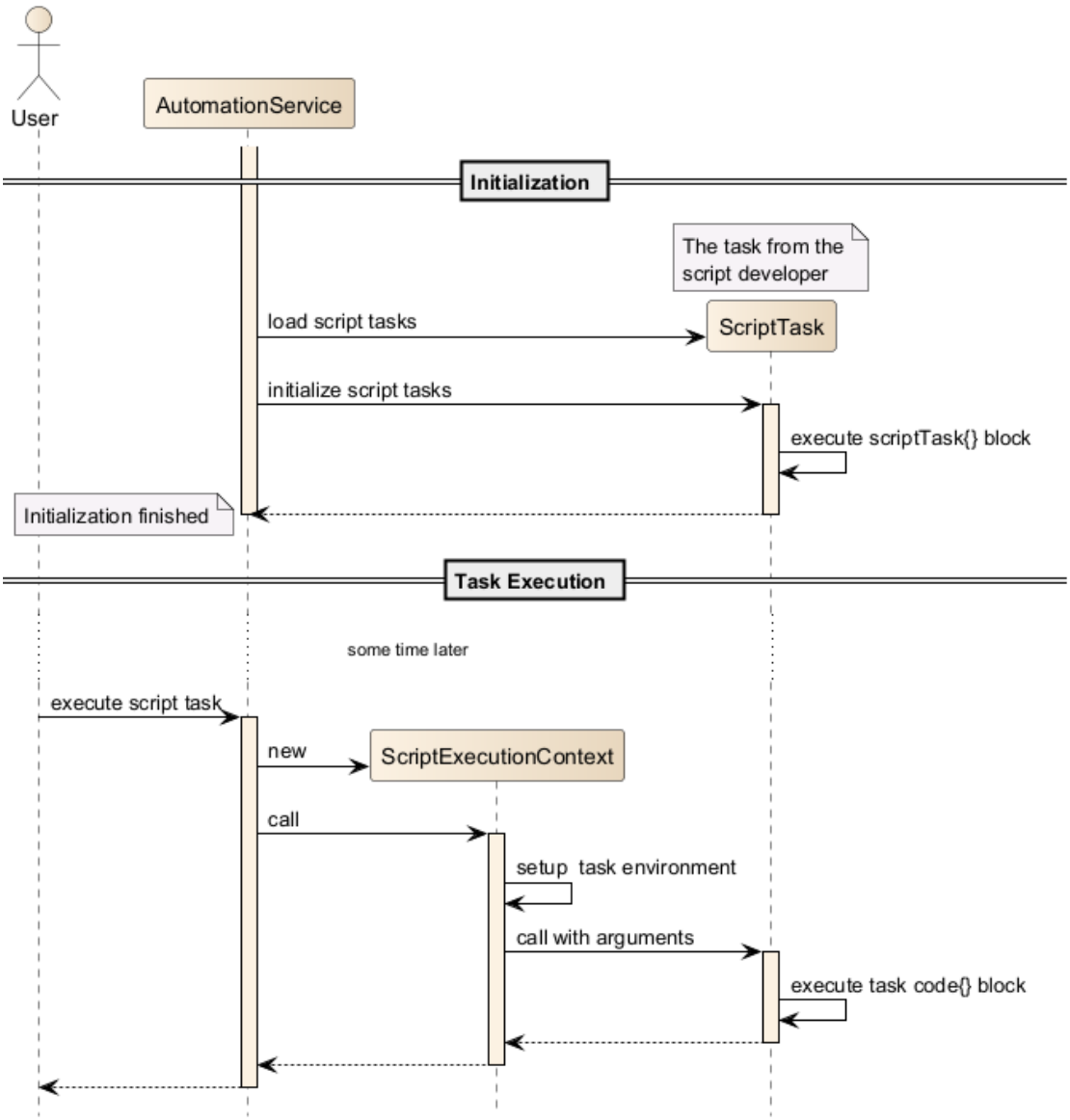


Figure 6.4: Script Task Execution Sequence

6.4.3 Automation Path Features

Available through the Execution Context during Script Task execution. Script tasks could resolve relative and absolute file system paths with the `IAutomationPathsApi`.

As entry point call `paths` in a `code{ }`.

There are multiple ways to resolve relative paths:

- by Script folder
- by Temp folder
- by BSW Package folder
- by Project folder
- by any parent folder

6.4.3.1 General Path Resolution

Resolves a file path and preserves the relative path `resolvePath(Object)`.

Note: It does **NOT** convert relative paths into absolute.

```
scriptTask("TaskName"){
    code{
       // Resolves a path and preserve relative paths
       Path p = paths.resolvePath("MyFile.txt")
    }
}
```

Listing 6.14: Resolves a path with the resolvePath() method

The `resolvePath(Path parent, Object path)` method resolves a file path relative to supplied parent folder.

This method converts the supplied path based on its type:

- A `CharSequence`, including `String` or `GString`. Interpreted relative to the parent directory. A string that starts with `file:` is treated as a file URL.
- A `File`: If the file is an absolute file, it is returned as is. Otherwise, the file's path is interpreted relative to the parent directory.
- A `Path`: If the path is an absolute path, it is returned as is. Otherwise, the path is interpreted relative to the parent directory.
- A `URI` or `URL`: The URL's path is interpreted as the file path. Currently, only `file:` URLs are supported.
- A `IHasURI`: The returned URI is interpreted as defined above.
- A `Closure`: The closure's return value is resolved recursively.
- A `Callable`: The callable's return value is resolved recursively.
- A `Supplier`: The supplier's return value is resolved recursively.
- A `Provider`: The provider's return value is resolved recursively.

The return type is `java.nio.file.Path`.

```
scriptTask("TaskName"){
    code{
        // Resolves a path relative to the supplied folder
        Path parentFolder = Paths.get('.')
        Path p = paths.resolvePath(parentFolder, "MyFile.txt")
    }
}
```

Listing 6.15: Resolves a path with the resolvePath() method

6.4.3.2 Path Resolution by Base Folder

- `getBswPackageRootFolder()`
- `getScriptFolder()`
- `getProjectFolder()`
- `getTempFolder()`

Resolution relative to Script Folder The `resolveScriptPath(Object)` method resolves a file path relative to the script directory of the executed `IScript`.

This method converts the supplied path same as the `resolvePath(Path, Object)` method. The return type is `java.nio.file.Path`. See 6.4.3.1 on the preceding page.

```
scriptTask("TaskName"){
    code{
        // Resolves a path relative to the script folder
        Path p = paths.resolveScriptPath("MyFile.txt")
    }
}
```

Listing 6.16: Resolves a path with the resolveScriptPath() method

Resolution relative to Project Folder The `resolveProjectPath(Object)` method resolves a file path relative to the project directory (see `getProjectFolder()`) of the current active project.

This method converts the supplied path same as the `resolvePath(Path, Object)` method. The return type is `java.nio.file.Path`. See 6.4.3.1 on the previous page.

There must be an active project to use this method. See chapter 6.5.2 on page 59 for details about active projects.

```
scriptTask("TaskName"){
    code{
        // Resolves a path relative active project folder
        Path p = paths.resolveProjectPath("MyFile.txt")
    }
}
```

Listing 6.17: Resolves a path with the resolveProjectPath() method

Resolution relative to BSW Package Folder The `resolveBswPackagePath(Object)` method resolves a file path relative to the BSW Package directory (see `getBswPackageRootFolder()`).

This method converts the supplied path same as the `resolvePath(Path, Object)` method. The return type is `java.nio.file.Path`. See 6.4.3.1 on page 41.

```
scriptTask("TaskName"){
    code{
        // Resolves a path relative BSW Package folder
        Path p = paths.resolveBswPackagePath("MyFile.txt")
    }
}
```

Listing 6.18: Resolves a path with the resolveBswPackagePath() method

Resolution relative to Temp Folder The `resolveTempPath(Object)` method resolves a file path relative to the script temp directory of the executed `IScript`. A new temporary folder is created for each `IScriptTask` execution.

This method converts the supplied path same as the `resolvePath(Path, Object)` method. The return type is `java.nio.file.Path`. See 6.4.3.1 on page 41.

```
scriptTask("TaskName"){
    code{
        // Resolves a path relative to the temp folder
        Path p = paths.resolveTempPath("MyFile.txt")
    }
}
```

Listing 6.19: Resolves a path with the resolveTempPath() method

6.4.3.3 Predefined Path Properties

```
scriptTask("TaskName"){
    code{
        scriptLogger.info "Ecuc file path: " + paths.ecucFile
        scriptLogger.info "Ecuc folder path: " + paths.ecucFolder
        scriptLogger.info "Generator output folder path: " + paths.outputFolder
        scriptLogger.info "Autosar folder path: " + paths.autosarFolder
        scriptLogger.info "Service SWC folder path: "+ paths.serviceSwcFolder
        // not complete list
    }
}
```

Listing 6.20: Resolves predefined path variables

```
scriptTask("TaskName", DV_PROJECT){
    code{
        // The property OutputFolder is the folder of the generated artifacts
        Path folder = paths.outputFolder
    }
}
```

Listing 6.21: Get the project output folder path

```
scriptTask("TaskName"){
    code{
       // The property bswPackageRootFolder is the folder of the used BSW
       Path folder = paths.bswPackageRootFolder
    }
}
```

Listing 6.22: Get the BSW Package folder path

6.4.4 Script logging

The script task execution API (`IScriptExecutionContext`) provides a script logger to log events during an execution. The method `getScriptLogger()` returns the logger. The logger can be used to log Errors, Warnings, Debug messages etc.

You shall **always prefer** the usage of the **logger** before using the `println()` of `stdout or stderr`.

In any code block without direct access to the script API, you can write the following code to access the logger: `ScriptApi.scriptLogger`

```
scriptTask("TaskName"){
    code{
        // Use the scriptLogger to log messages
        scriptLogger.info  "My script is running"
        scriptLogger.warn  "My Warning"
        scriptLogger.error "My Error"
        scriptLogger.debug "My debug message"
        scriptLogger.trace "My trace message"

        // Also log an Exception as second argument
        scriptLogger.error("My Error", new RuntimeException("MyException"))
    }
}
```

Listing 6.23: Usage of the script logger

The `ILogger` also provides a formatting syntax for the format String. The syntax is `{IndexNumber}` and the index of arguments after the format `String`.

It is also possible to use the Groovy `GString` syntax for formatting.

```
scriptTask("TaskName"){
    code{ argument ->
        // Use the format methods to insert data
        scriptLogger.info("My script {0} with:{1}", scriptTask, argument)
    }
}
```

Listing 6.24: Usage of the script logger with message formatting

```
scriptTask("TaskName"){
    code{ argument ->
        // Use the Groovy GString syntax to insert data
        scriptLogger.info "My script $scriptTask with: $argument"
    }
}
```

Listing 6.25: Usage of the script logger with Groovy GString message formatting

6.4.5 Versions API

The Versions API provides methods to identify the current used versions e.g. Configurator Version, Automation Interface Version, ...

```
scriptTask('taskName', DV_APPLICATION){
  code {
        versions{
            String cfgVersion = daVinciConfiguratorVersion
            println cfgVersion

            String aiVersion = daVinciAutomationInterfaceVersion
            println aiVersion

            String bswId = bswDeliveryId

            String bswPackageNumber = bswPackageNumber

        }
    }
}
```

Listing 6.26: Get Configurator and AI Version

daVinciAutomationInterfaceVersion Returns the AutomationInterface version in the format [major].[minor].[bugfix].

daVinciConfiguratorVersion Returns the DaVinciConfigurator version in the format [major].[minor].[bugfix].

bswDeliveryId Returns the BSW delivery ID, like CBD00000.

bswPackageNumber Returns the BSW package number, like 33.06.06.

6.4.6 Script Error Handling

6.4.6.1 Script Exceptions

All exceptions thrown by any script task execution are sub types of `ScriptingException`.

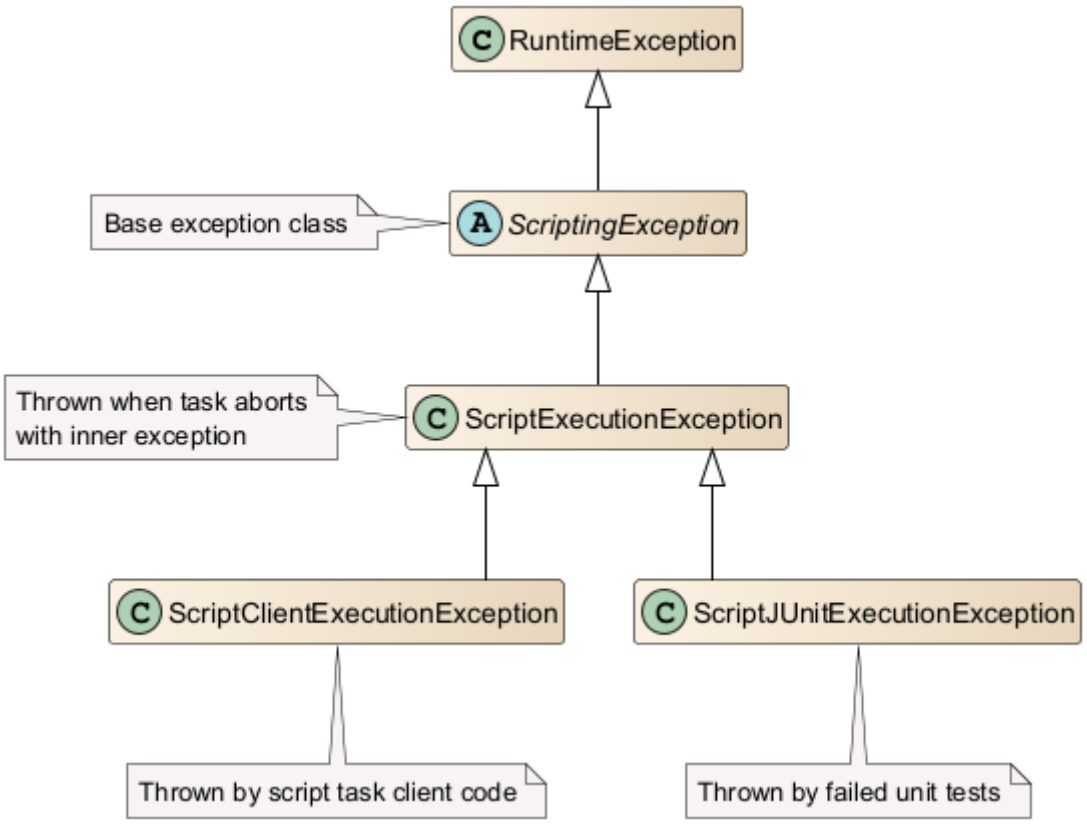


Figure 6.5: ScriptingException and sub types

6.4.6.2 Script Task Abortion by Exception

The script task can throw an `ScriptClientExecutionException` to abort the execution of an `IScriptTask`, and display a meaningful message to the user.

```
scriptTask("TaskName"){

    code{
        // Stop the execution and display a message to the user
        throw new ScriptClientExecutionException("Message to the User")
    }
}
```

Listing 6.27: Stop script task execution by throwing an ScriptClientExecutionException

Exception with Console Return Code An `ScriptClientExecutionException` with an return code of type `Integer` will also abort the execution of the `IScriptTask`.

But it *also changes the return code* of the console application, if the `IScriptTask` was executed in the console application. This could be used when the console application of the DaVinci Configurator is called for other scripts or batch files.

```
scriptTask("TaskName"){

    code{
        // The return code will be returned by the DvCmd.exe process
        def returnCode = 50
        throw new ScriptClientExecutionException(returnCode, "Message to the User"
            )
    }
}
```

Listing 6.28: Changing the return code of the console application by throwing an ScriptClientExecutionException

Reserved Return Codes The returns codes `0-20` are reversed for internal use of the DaVinci Configurator, and are not allowed to be used by a client script. Also negative returns codes are not permitted.

6.4.6.3 Unhandled Exceptions from Tasks

When a script task execution throws any type of `Exception` (more precise `Throwable`) the script task is marked as failed and the `Exception` is reported to the user.

6.4.7 User Defined Classes and Methods

You can define your own methods and classes in a script file. The methods a called like any other method.

```
scriptTask("Task"){
    code{
        userMethod()
    }
}

def userMethod(){
    return "UserString"
}
```

Listing 6.29: Using your own defined method

Classes can be used like any other class. It is also possible to define multiple classes in the script file.

```
scriptTask("Task"){
    code{
        new UserClass().userMethod()
    }
}

class UserClass{
    def userMethod(){
        return "ReturnValue"
    }
}
```

Listing 6.30: Using your own defined class

You can also create classes in different files, but then you have to write imports in your script like in normal Groovy or Java code.

The script should be structured as any other development project, so if the script file gets too big, please refactor the parts into multiple classes and so on.

daVinci Block The classes and methods must be outside the `daVinci{ }` block.

```
import static com.vector.cfg.automation.api.ScriptApi.*
daVinci{
    scriptTask("Task"){
        code{}
    }
}

def userMethod(){}

class UserClass{}
```

Listing 6.31: Using your own defined method with a daVinci block

Code Completion Note that the code completion for the Automation API will not work automatically in own defined classes and methods. You have to open for example a `scriptCode{}`

block. The chapter6.4.8 describes how to use the Automation API for your own defined classes and methods.

6.4.8 Usage of Automation API in own defined Classes and Methods

In your own methods and classes the automation API is not automatically available differently as inside of the script task `code{}` block. But it is often the case, that methods need access to the automation API.

The class **`ScriptApi` provides static methods as entry points** into the automation API. The static methods either return the API objects, or you could pass a `Closure`, which will activate the API inside of the `Closure`.

6.4.8.1 Access the Automation API like the Script code{} Block

The `ScriptApi.scriptCode(Transformer)` method provides access to all automation APIs the same way as inside of the normal script `code{}` block.

This is useful, if you want to call script code API inside of your own methods and classes.

```
def yourMethod(){
    // Needs access to an automation API
    ScriptApi.scriptCode{
        // API is now available
        generation.generate()
    }
}
```

Listing 6.32: ScriptApi.scriptCode{} usage in own method

The `ScriptApi.scriptCode()` method can be used to call API in Java style.

```
def yourMethod(){
    // Needs access to an automation API
    ScriptApi.scriptCode().generation.generate()
}
```

Listing 6.33: ScriptApi.scriptCode() usage in own method

Java note: The `ScriptApi.scriptCode()` returns the `IScriptExecutionContext`.

6.4.8.2 Access the Project API of the current active Project

The `ScriptApi.activeProject()` method provides access to the project automation API of the currently active project. This is useful, if you want to call project API inside of your own methods and classes.

```
def yourMethod(){
    // Needs access to an automation API
    ScriptApi.activeProject{
        // Project API is now available
        transaction{
            // Now model modifications are allowed
        }
    }
}
```

Listing 6.34: ScriptApi.activeProject{} usage in own method

The `ScriptApi.activeProject()` method returns the current active `IProject`.

```
def yourMethod(){
     // Needs access to an automation API
     IProject theActiveProject = ScriptApi.activeProject()
}
```

Listing 6.35: ScriptApi.activeProject() usage in own method

6.4.9 User Defined Script Task Arguments

A script task can create `IScriptTaskUserDefinedArgument`, which can be set by the user (e.g. from the commandline) to pass user defined arguments to the script task execution. An argument can be optional or required. The arguments are type safe and checked before the task is executed. An argument can be specified with a value and also without one.
Example: "–count 25" or "-s"

Possible valueTypes are:

- `String`
- `Boolean`
- `Void`: For parameter where only the existence is relevant.
- `File`: The existence of the file is not checked by default. See argument validators.
- `Path`: Same as `File`
- `Integer`
- `Long`
- `Double`

The help text is automatically expanded with the help for user defined script task arguments.

```
scriptTask("TaskName"){
  def procArg = newUserDefinedArgument("p", Void, "Enables the processing of ...")
  code{
      if(procArg.hasValue){
        scriptLogger.info  "The argument -p was defined"
      }
  }
}
```

Listing 6.36: Script task UserDefined argument with no value

```
scriptTask("TaskName"){
  def countArg = newUserDefinedArgument("count", Integer,
                                        "The amount of elements to create")

  def nameArg = newUserDefinedArgument("name", String,
                                       "The element name to create")
  code{
      // NOTE: The value can only be retrieved within the code closure
      int count = countArg.value
      String name = nameArg.value

      scriptLogger.info  "The arguments --name and --count were $name, $count"
  }
}
```

Listing 6.37: Define and use script task user defined arguments from CLI

```
scriptTask("TaskName"){
  //  User Defined Argument with the default value 25.0
  def procArg = newUserDefinedArgument("p", Double, 25.0, "Help text ...")
  code{
      double value = procArg.value
      scriptLogger.info  "The argument -p was $value"
  }
}
```

Listing 6.38: Script task UserDefined argument with default value

```
scriptTask("TaskName"){
  def multiArg = newUserDefinedArgument("multiArg", String, "Help text ...")

  code{

      List<String> values = multiArg.values  // Call values instead of value
      scriptLogger.info  "The argument --multiArg  had values: $values"
  }
}
```

Listing 6.39: Script task UserDefined argument with multiple values

6.4.9.1 User defined Argument Validators

You could also specify a validator for the argument to check for special conditions, like the file must exist. This is helpful to provide a quick feedback to the user, if the task would be executable. Simply add the validator at the end of the `newUserDefinedArgument()` call. The validator code is called when the input is checked. There are also default validators available, like:

- Constraints.IS_EXISTING_FOLDER
- Constraints.IS_EXISTING_FILE
- Constraints.IS_VALID_AUTOSAR_SHORT_NAME

Please see chapter 6.4.9.2 on the following page for more available validators.

```
import  com.vector.cfg.util.contract.util.Constraints

scriptTask("TaskName"){
  def contArg = newUserDefinedArgument( "p", String,
                                        "Help text ...",
                                        Constraints.
                                            IS_VALID_AUTOSAR_SHORT_NAME_PATH )
  code{

      String value = contArg.value
      scriptLogger.info  "The argument -p was $value"
  }
}
```

Listing 6.40: Script task UserDefined argument with predefined validator

Or you implement your own validation logic, by passing a `Closure`, which throws an exception, if the value is invalid.

```
scriptTask("TaskName"){

  //  User Defined Argument with the validator code as parameter
  newUserDefinedArgument( "p", Integer, 20, "Help text ...",
                { value ->
                    if( value % 2){
                        throw new IllegalArgumentException("The value has to be
                            even.")
                    }
                } )

  code{
  }
}
```

Listing 6.41: Script task UserDefined argument with own validator

6.4.9.2 Constraints

`Constraints` provides general purpose constraints for checking given parameter values throughout the automation interface. These constraints are referenced from the AutomationInterface documentation wherever they apply. The AutomationInterface takes a fail fast approach verifying provided parameter values as early as possible and throwing appropriate exceptions if values violate the corresponding constraints.

The following constraints are provided:

IS_NOT_NULL Ensures that the given `Object` is not `null`.

IS_NON_EMPTY_STRING Ensures that the given `String` is not empty.

IS_VALID_FILE_NAME Ensures that the given `String` can be used as a file name.

IS_VALID_PROJECT_NAME Ensures that the given `String` can be used as a name for a project. A valid project name starts with a letter [a-zA-Z] contains otherwise only characters matching [a-zA-Z0-9_] and is at most 128 characters long.

IS_NON_EMPTY_ITERABLE Ensures that the given `Iterable` is not empty.

IS_VALID_AUTOSAR_SHORT_NAME Ensures that the given `String` conforms to the syntactical requirements for AUTOSAR short names.

IS_VALID_AUTOSAR_SHORT_NAME_PATH Ensures that the given `String` conforms to the syntactical requirements for AUTOSAR short name paths.

IS_ABSOLUTE Ensures that Ensures that given `Path` is absolute.

IS_WRITABLE Ensures that the file or folder represented by the given `Path` exists and can be written to.

IS_READABLE Ensures that the file or folder represented by the given `Path` exists and can be read.

IS_EXISTING_FOLDER Ensures that the given `Path` points to an existing folder.

IS_EXISTING_FILE Ensures that the given `Path` points to an existing file.

IS_CREATABLE_FOLDER Ensures that the given `Path` either points to an existing folder which can be written to or points to a location at which a corresponding folder could be created.

IS_DCF_FILE Ensures that the given `Path` points to a DaVinci Developer workspace file (.dcf file).

IS_DVJSON_FILE Ensures that the given `Path` points to a DaVinci project file (.dvjson file).

IS_ARXML_FILE Ensures that the given `Path` points to an .arxml file.

6.4.9.3 Run Script Task with User Defined Task Arguments from CLI

The help of the `run` command shows, how to execute a script task with user defined arguments.

```
Usage: dvcfg-b automation run [-h] -b=<folder> [-p=<file>]
                  -t=<task>[,<task>...] [-t=<task>[,<task>...]]...
                  [-a=<arg>]... [-l=<location>[,<location>...]]...
                  [--debugger[=<port>]] [--no-save]

Description:
Run automation tasks with arguments.
```

Listing 6.42: Help for run script task

```
* -t, --task=<task>[,<task>...]     List of script tasks to execute.
                                    E.g.: -t task1,task2
 -a, --arg=<arg>                    Define a set of arguments specific to a single
     task.
                                    E.g.: -a 'task1' -a '--name=str1 --value=1' -a
                                        'task2' -a '-i 1,2,3'
```

Listing 6.43: Help for user defined task arguments

Let's have a look at an example script task with user defined arguments as shown below.

```
import static com.vector.cfg.automation.api.ScriptApi.*

scriptTask("userArgTask", DV_APPLICATION) {

def arg_enable = newUserDefinedArgument("enable", Void, "Help text ...")
def arg_count = newUserDefinedArgument("count", Integer, "Help text ...")
def arg_name = newUserDefinedArgument("name ", String, "Help text ...")
def arg_double = newUserDefinedArgument("double", Double, 25.0, "Help text ...")
def arg_multiArg = newUserDefinedArgument("multiArg", String, "Help text ...")

code {
    if (arg_enable.hasValue) {
        scriptLogger.info "The argument --enable was defined."
    }
    if (arg_count.hasValue) {
        scriptLogger.info "The argument --count has the value ${arg_count.value}."
    }
    if (arg_name.hasValue) {
        scriptLogger.info "The argument --name has the value ${arg_name.value}."
    }
    if (arg_double.hasValue) {
        scriptLogger.info "The argument --double has the value ${arg_double.value
            }."
    }
    if (arg_multiArg.hasValue) {
        List<String> values = arg_multiArg.values
        scriptLogger.info "The argument --multiArg has values: ${values}."
    }
    }
}
```

Listing 6.44: Example script task with user defined arguments

To run this script task from CLI, you can use the following command.

```
automation run ...
           -t "userArgTask"
           -a "userArgTask"
           -a "--enable --count=25 --name=John --double=25.0 --multiArg=1,2,3,4,5"
```

Listing 6.45: Example CLI call with user defined task arguments

6.4.10 Stateful Script Tasks

Script tasks normally have no state or cached data, but it can be useful to cache data during an execution, or over multiple task executions. The `IScriptExecutionContext` provides two methods to save and restore data for that purpose:

- `getExecutionData()` - caches data during one task execution
- `getSessionData()` - caches data over multiple task executions

Execution Data Caches data during a single script task execution, which allows to save calculated values or services needed in multiple parts of the task, without recalculating or creating it. Note: When the task is executed again the `executionData` will be empty.

```
scriptTask("TaskName"){
  code{
      // Cache a value for the execution
      executionData.myCacheValue = 500

      def val = executionData.myCacheValue // Retrieve the value anywhere
      scriptLogger.info  "The cached value is $val"

      // Or access it from any place with ScriptApi.scriptCode like:
      def sameValue = ScriptApi.scriptCode.executionData.myCacheValue
  }
}
```

Listing 6.46: executionData - Cache and retrieve data during one script task execution

Session Data Caches data over multiple task executions, which allows to implement a stateful task, by saving and retrieving any data calculated by the task itself.

Caution: The data is saved globally so the usage of the `sessionData` can lead to memory leaks or `OutOfMemoryErrors`. You have to take care not to store too much memory in the `sessionData`. The DaVinci Configurator will also free the `sessionData`, when the system run low on free memory. So you have to deal with the fact, that the `sessionData` was freed, when the script task getting executed again. But the data is not deallocated during a running execution.

```
scriptTask("TaskName"){
  // Setup - set the value the first time, this is only executed once (during
      initialization)
  sessionData.myExecutionCount = 1

  code{
      // Retrieve the value
      def executionCount = sessionData.myExecutionCount

      scriptLogger.info  "The task was executed $executionCount times"

      // Update the value
      sessionData.myExecutionCount = executionCount + 1
  }
}
```

Listing 6.47: sessionData - Cache and retrieve data over multiple script task executions

API usage Both methods `executionData` and `sessionData` return the same API of type `IScriptTaskUserData`.

The `IScriptTaskUserData` provides methods to retrieve and store properties by a key (like a `Map`). The retrieval and store methods are `Object` based, so any `Object` can be a key. The exception are `Class` instances (like `String.class`, which required that the value is an instance of the `Class`).

On retrieval if a property does not exist an `UnknownPropertyException` is thrown. Properties can be set multiple times and will override the old value. The keys of the properties used to retrieve and store data are compared with `Object.equals(Object)` for equality.

The listing below describes the usage of the API:

```
scriptTask("TaskName"){
  code{
      def val
      // The sessionData and executionData have the same API

      // You have multiple ways to set a value
      executionData.myCacheId = "VALUE"
      executionData.set("myCacheId", "VALUE")
      executionData["myCacheId"] =  "VALUE"
      // Or with classes for a service locator pattern
      executionData.set(Integer.class, 50)  // Possible for any Class
      executionData[Integer] = 50

      // There are the same ways to retrieve the values
      val = executionData.myCacheId
      val = executionData.get("myCacheId")
      val = executionData["myCacheId"]
      // Or with classes for a service locator pattern
      val  = executionData.get(Integer.class)
      val  = executionData[Integer]

      // You can also ask if the property exists
      boolean exists = executionData.has("myCacheId")
  }
}
```

Listing 6.48: sessionData and executionData syntax samples

6.4.11 ScriptAccess - Calling ScriptTasks

Sometimes it can be helpful to call other script tasks from inside your task. The `scripts{}` block or `getScripts()` method provides API to retrieve existing `IScripts` and call other `IScriptTasks` from your running `IScriptTask`.

Note: If you **just want to reuse code** of your own scripts in an automation script project, create a normal method containing the code and call it, instead of calling the task. The method is typesafe, has code completion support and is **much faster** than calling a script task.

Calling script tasks To call a task you need the name of the task and the `IScriptTaskType`. The `IScriptTaskType` determines the argument types and the return type of the script task. Then you can use `scripts.callScriptTask(String, Object...)` to call the script.

You could also use `callScriptTaskWithUserArgs(String, String, Object...)`, if you want to pass user defined arguments.

```
scriptTask("TaskName"){
    code{
        scripts.callScriptTask("OtherTask")
        //The same
        scripts{
            callScriptTask("OtherTask")
        }
    }
}

scriptTask("OtherTask"){
    code{
        //Other task code
    }
}
```

Listing 6.49: Call another script task from a script task

Calling script tasks with task arguments If the `IScriptTaskType` requires task arguments, you have to pass the arguments to the `callScriptTask()` methods. The return value of the method is the returned value of the called script task.

```
scriptTask("TaskName", DV_PROJECT){
    code{
        def arg1 = "First argument"
        def arg2 = 5
        def result = scripts.callScriptTask("OtherTask", arg1, arg2)
        // Result contains the calculated value of OtherTask
    }
}

scriptTask("OtherTask"){
    code{arg1, arg2 ->
        return arg1 + arg2
    }
}
```

Listing 6.50: Call another script task with arguments

6.5 Project Handling

Project handling comprises creating new projects, opening existing projects or accessing the currently active project.

`IProjectHandlingApi` provides methods to access to the active project, for creating new projects and for opening existing projects.

`getProjects()` allows accessing the `IProjectHandlingApi` like a property.

```
scriptTask('taskName') {
  code {
    // IProjectHandlingApi is available as "projects" property
    def projectHandlingApi = projects
  }
}
```

Listing 6.51: Accessing IProjectHandlingApi as a property

`projects(Transformer)` allows accessing the `IProjectHandlingApi` in a scope-like way.

```
scriptTask('taskName') {
  code {
    projects {
      // IProjectHandlingApi is available inside this Closure
    }
  }
}
```

Listing 6.52: Accessing IProjectHandlingApi in a scope-like way

6.5.1 Projects

Projects in the AutomationInterface are represented by `IProject` instances. These instances can be created by:

- Creating a new project
- Loading an existing project

You can only access `IProject` instances by using a `code` block at `IProjectHandlingApi` or `IProjectRef` class. This shall prevent memory leaks, by not closing open projects.

6.5.2 Accessing the active Project

The `IProjectHandlingApi` provides access to the active project. The active project is either (in descending order):

- The last `IProject` instance activated with a `code` block
  - Stack-based - so multiple opened projects are possible and the last (inner) `code` block is used.
- The passed project to a project task
- Or the loaded project in the current DaVinci Configurator in an application task

The figure 6.6 on the next page describes the behavior to search for the active project of a script task.

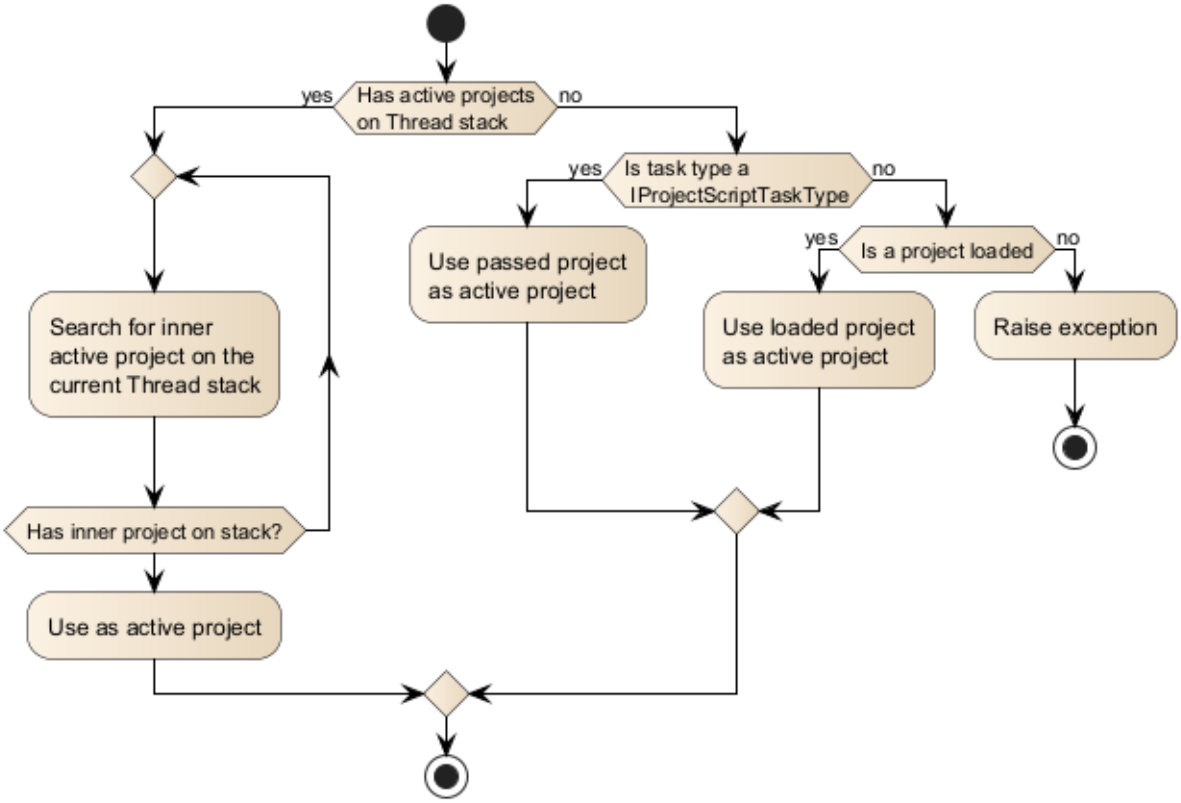


Figure 6.6: Search for active project in getActiveProject()

It is possible that there is no active project, e.g. no project was loaded.

You can switch the active project, by calling the `IProject.with(Transformer)` method on an `IProject` instance.

```
// Retrieve theProject from other API like load a project
IProject theProject = ...;
theProject.with {
    // Now theProject is the new active project inside of this lambda
}
```

Listing 6.53: Switch the active project

To access the active project you can use the `activeProject(Transformer)` and `getActiveProject()` methods.

```
scriptTask('taskName') {
  code {
    if (projects.projectActive) {
      // active IProject is available as "activeProject" property
      scriptLogger.info "Active project: ${projects.activeProject.projectName}"
      projects.activeProject {
        // active IProject is available inside this Closure
        scriptLogger.info "Active project: ${projectName}"
      }
    } else {
      scriptLogger.info 'No project active'
    }
  }
}
```

Listing 6.54: Accessing the active IProject

`isProjectActive()` returns `true` if and only if there is an active `IProject`. If `isProjectActive()` returns `true` it is safe to call `getActiveProject()`.

`getActiveProject()` allows accessing the active `IProject` like a property.

`activeProject(Transformer)` allows accessing the active `IProject` in a scope-like way. This will enable the project specific API inside of the `code` parameter.

6.5.3 Accessing the project search

The `SearchApiEntryPoint` provides the possibility to search parameters, containers or module configurations by several criteria. The API also provides the ability to execute several checks on the results.

```
scriptTask("TestSearchApi", DV_PROJECT) {
code {
        activeProject {
            // Use the search API with a certain datamining query from the GUI
               FindView
            def findings = search("Definition==EcucGeneral")

            // It is possible to use the search API multiply times
            def number = search("Definition==EcucGeneral").size()

            // Use the closure to operate on the search results
            search("Definition==EcucGeneral"){
                // To prove that the result
                def containsObjCondition = { sr -> sr.result().contains("/
                   ActiveEcuC/EcuC/EcucGeneral[0:DummyFunction]") }
                def result = isConditionMet(containsObjCondition)
                // Report or print result

                // Check if result size is equal to 22
                isSize(ESearchOperator.EQ, 22)
            }
        }
    }
}
```

Listing 6.55: Using the search API

`search(String)` is used to evaluate a `DataMiningService` query. If the query isn't correct, a `SearchApiException` will be thrown.

Example:

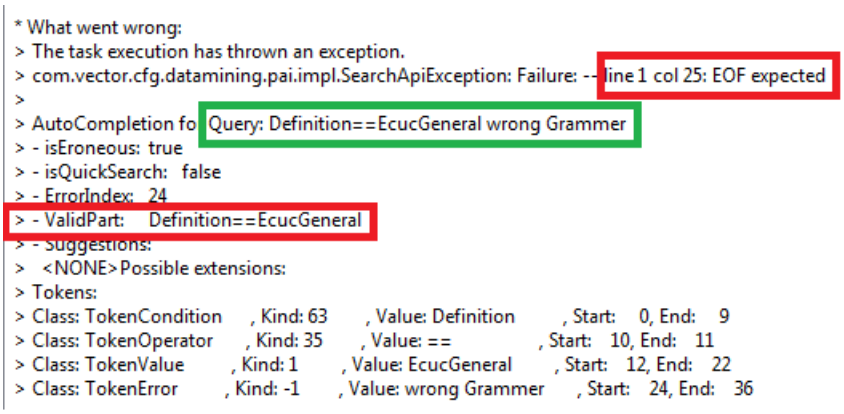


Figure 6.7: SearchApi Exception Message

The `ISearchResultApi` provides the possibility to make evaluations based on the search results.

Use `isConditionMet(Predicate)` to pass a condition to the search result. With this condition a certain expectation can be proven.

Use `isSize(ESearchOperator, int)` to pass a condition to compare it with the size of the search result. It can be used to check a certain expectation. The passed operation (`ESearchOperator`) defines the condition.

Use `size()` to get the result size of the search.

Use `result()` to get a list of the result elements. The list contains ObjectLinks as `Strings`. Returns an empty list if no elements are found.

6.5.4 Expression Evaluation API

The `IExpressionEvaluationApi` provides the possibility to evaluate the complex logical expressions composed of single queries.

The `evaluateExpression(String)` is used to evaluate `IDataMiningService` queries linked with logical operators.

6.5.5 Accessing Project Settings

The entrypoint enables querying or modifying the project settings. Every modification of the project settings must be done in a transaction. The following section describes how the project settings of a project can be accessed and modified.

Use `getProjectSettings()` or `projectSettings(Transformer)` to specify the project settings for a project.

6.5.5.1 Project Folder Api

`IProjectFolderApi` contains the methods to specify the current project's folder settings.

Use `getFolder()` or `folder(Transformer)` to specify the folder settings.

`IProjectFolderApi` contains the methods to specify the current project's folder settings.

Module Files Folder Get the module files folder for the current project with `getModuleFilesFolder()`.

Module Files Folder Gets the GenDataVtt folder for the current project with `getGenDataVtt()`.

Templates Folder Get the templates folder for the current project with `getTemplatesFolder()`.

Service Components Folder Get the service component files folder for the current project with `getServiceComponentFilesFolder()`.

Application Components Folder Get the application component files folder for the current project with `getApplicationComponentFilesFolder()`.

Log Files Folder Get the log files folder for the current project with `getLogFilesFolder()`.

Measurement And Calibration Files Folder Get the measurement and calibration files folder for the current project with `getMeasurementAndCalibrationFilesFolder()`.

AUTOSAR Files Folder Get the AUTOSAR files folder for the current project with `getAutosarFilesFolder()`.

Bsw package Folder Get the bsw package path of the current dvjson project file with `getBswPackageFolder()`.

Timing Extension Folders Get the TimingExtension folders for the current project with `getTimingExtensionFolders()`.

Structured Extract Folders Get the structured Extract folder for the current project with `getStructuredExtractFolder()`.

Definition Restriction folder Get the Definition Restriction files folder for the current project with `getDefinitionRestrictionFolder()`.

Bsw Internal Behavior folder Get the Bsw Internal Behavior files folder for the current project with `getBswInternalBehaviour()`.

6.5.5.2 Target Project Settings

`IProjectTargetApi` is the entry point for accessing and modifying the target settings.

Use `getTarget()` or `target(Transformer)` to specify the target project settings

Use the following methods to access the project settings. The example shows how to use the API.

```
scriptTask("TestsProjectSettings", DV_PROJECT) {
    code {
        activeProject {
            projectSettings {
                transaction {
                    target{

                    // Get the available derivatives as collection
                    def newDerivative = getAvailableDerivatives().getFirst()
                    // Set the derivative setting with the new value
                    derivative(newDerivative)
                    // Returns an Optional containing the new value
                    getDerivative()

                    def newCompiler = getAvailableCompilers().getFirst()
                    compiler(newCompiler)
                    getCompiler()

                    def newPinLayout = getAvailablePinLayouts().getFirst()
                    pinLayout(newPinLayout)
                    getPinLayout()

                    }
                }
            }
        }
    } // code
} // scriptTask
```

Listing 6.56: Access and modify Project Settings - Variant 1

Module The module which supported Derivatives shall be retrieved.

Available Derivatives `getAvailableDerivatives(String)` returns all possible input values for `setDerivative(String, DerivativeInfo)`. Note: This function will return value of `getAvailableDerivatives()` if module is not hardware-specific.

Module The module which the derivative shall be set for.

Derivative Set the derivative for given module with `setDerivative(String, DerivativeInfo)`. The value given here must be one of the values returned by `getAvailableDerivatives(String)`.

Module The module which derivative setting shall be retrieved.

Get Derivative `getDerivative(String)` returns the value of the derivative configured for given module. Note: This function will return value of `getDerivative()` (potentially null) if no module-specific derivative has been configured.

Available Compilers `getAvailableCompilers()` returns all possible input values for `setCompiler(ImplementationProperty)`. Note: the available compilers depend on the currently configured derivative. This method will return an empty collection if no derivative has been configured at the time it is called.

Compiler Set the compiler for the new project with `setCompiler(ImplementationProperty)`. The value given here must be one of the values returned by `getAvailableCompilers()`.

Get Compiler `getCompiler()` returns the value of the current compiler project setting. Note: If no compiler has been configured, it will return null.

Available PinLayouts `getAvailablePinLayouts()` returns all possible input values for `setPinLayout(ImplementationProperty)`. Note: The available pin layouts depend on the currently configured derivative. This method will return an empty collection if no derivative has been configured at the time it is called.

PinLayout Set the pinLayout of the selected derivative for the project with `setPinLayout(ImplementationProperty)`. The value given here must be one of the values returned by `getAvailablePinLayouts()`.

Get PinLayout `getPinLayout()` returns the value of the current pinLayout project setting. Note: If no pinLayout has been configured, it will return null.

6.5.5.3 UseCase Project Settings

The `IProjectUseCaseApi` enables querying or modifying the "use case" setting. Use case limits the application of pre-configuration or recommended configuration in particular application cases. The following section describes how an use case can be accessed and modified.

Use `getUseCases()` or `useCases(Transformer)` to specify the use case context of the project settings.

`IProjectUseCaseApi` is the entry point for accessing and modifying the use cases and their values.

```
scriptTask("TestsProjectSettings", DV_PROJECT) {
    code {
        projectSettings {

            // Entry point to access the the useCase context
            useCases {
                transaction {
                    // Get the useCase to modify
                    def useCase = getUseCaseByName("MyPreUseCase")
                    // Get the available values for the useCase to set
                    def availableUseCaseValues = useCase.getAvailableValues()
                    // Get the current useCase value and name
                    def name = useCase.name
                    def value = useCase.value

                    // Set the new useCase value
                    useCase.value availableUseCaseValues[6]
                }
            }
        }
    } // code
} // scriptTask
```

Listing 6.57: Access and modify Use Project Settings UseCases

Available UseCases The `getAvailableUseCases()` returns an immutable list of all available `IUseCase`.

UseCase The `getUseCaseByName(String)` returns a `IUseCase`.

Available values The `getAvailableValues()` returns an immutable list of available values for the `IUseCase`.

Name `getName()` returns the name of the current use case.

Value `getValue()` returns the value of the current use case.

Value The `setValue(String)` sets the use case value. The value to set must be one of the returned list by `getAvailableValues()`.

6.5.6 Accessing Advanced Project Settings

The method `advancedProjectSettings(Object, Action)` changes CFG6 project settings as specified by the given `Closure`. Inside the action the `IAdvancedProjectSettingsApi` is available. Note: This api can only be used with closed projects.

```
scriptTask('changeAdvancedSettings', DV_APPLICATION) {
    code { Path projectPath, Path newDcfFilePath ->
        projects.advancedProjectSettings(projectPath){
            Path dcfPath = getDaVinciDeveloperWorkspace()
        }
    }
}
```

Listing 6.58: Setting the Advanced Settings

Get DaVinci Developer Workspace `getDaVinciDeveloperWorkspace()` returns the current set DaVinci Developer workspace path, or `null` if the path was not set.

Set DaVinci Developer Workspace `setDaVinciDeveloperWorkspace(Object)` sets DaVinci Developer workspace to the project.

Add Application Components Folder Add an application component files folder for the current project with `addApplicationComponentFilesFolder(Object)`.

The value given here is converted to `Path` using `com.vector.cfg.automation.scripting.api.ScriptConverte` 6.15.1 on page 320. Normally a relative path (to be interpreted relative to the project folder) should be given here.

The following constraints apply:

- `Constraints.IS_CREATABLE_FOLDER` 6.4.9.2 on page 53

Remove Application Components Folder Remove an application component files folder from the existing project with `removeApplicationComponentFilesFolder(Object)`.

6.5.6.1 Firewall Files Settings

Use `getFirewallFiles()` to specify the current project's Firewall Files settings in a scope-like way.

`IFireFilesApi` contains the methods for specifying the current project's Firewall Files settings.

General File Set the `path` of the General FIRe file wit `setGeneral(Object)`.

General File Get the `path` of the General FIRe file with `getGeneral()` .

Limp Home File Set the `path` of the Limp Home FIRe file with `setLimpHome(Object)`.

Limp Home File Get the `path` of the Limp Home FIRe file with `getLimpHome()`.

Sync Files Calls the `IFireFilesSyncService.syncFiles(java.util.Set)` to sync all fire files.

6.5.7 Creating a new CFG6 Project

The method `createProject(Action)` creates a new project as specified by the given `Closure`. Inside the closure the `ICreateProjectApi` is available.

The new project is not opened and usable until `IProjectRef.openProject(Transformer)` is called on the returned `IProjectRef`.

```
scriptTask('taskName', DV_APPLICATION) {
  code {
    def newProject = projects.createProject {
      projectName 'NewProject'
      projectFolder paths.resolveTempPath('projectFolder')
    }

    scriptLogger.info("Project created and saved to: $newProject")

    // Now open the project
    newProject.openProject{
        // Inside here the project and project settings can be used
    }
  }
}
```

Listing 6.59: Creating a new CFG6 project (Minimal Setup)

The next is a more sophisticated example of creating a project with multiple settings:

```
scriptTask('taskName', DV_APPLICATION) {
  code {
    def newProject = projects.createProject {

      projectName 'NewProject'
      projectFolder paths.resolveTempPath('projectFolder')

      author 'projectAuthor'
      version '0.9'
    }
  }
}
```

Listing 6.60: Creating a new project (with some optional parameters)

The `ICreateProjectApi` contains the methods to parameterize the creation of a new project.

6.5.7.1 Mandatory Settings

`projectName(String)` Specifies the name of the newly created CFG6 project file. The following constraint was applied:

- Constraints.IS_VALID_FILE_NAME6.4.9.2 on page 52

`projectFolder(Object)` Specify the folder in which to create the new CFG6 project. The value given here is converted to `Path`. The following constraints were applied:

- Constraints.IS_ABSOLUTE 6.4.9.2 on page 53

- Constraints.IS_CREATABLE_FOLDER 6.4.9.2 on page 53

6.5.7.2 Optional Project Settings

Author The author for the new project can be specified with `setAuthor(String)`. This is an optional parameter defaulting to the name of the currently logged-in user if the parameter is not provided explicitly.

The following constraints apply:

- Constraints.IS_NON_EMPTY_STRING 6.4.9.2 on page 52

Version The version for the new project can be specified with `setVersion(Object)`. This is an optional parameter defaulting to "1.0" if the parameter is not provided explicitly. The value given here is converted to `IVersion` using `ScriptConverters.TO_VERSION` 6.15.1 on page 320.

The following constraints apply:

- Constraints.IS_NOT_NULL 6.4.9.2 on page 52

6.5.7.3 Target Settings

Use `getProjectTarget()` or `projectTarget(Action)` to specify the new project's target settings for compiler, derivatives and pin layouts.

`ICreateProjectTargetApi` contains the API to specify the new project's target settings.

Available Derivatives `getAvailableDerivatives()` returns all possible input values for `setDerivative(DerivativeInfo)`.

Derivative Set the derivative for the new project with `setDerivative(DerivativeInfo)`. The new default project derivative refers to the first element in the collection returned by `getAvailableDerivatives()`. Call `setDerivative(DerivativeInfo)` if you would like change the derivative.

Available Compilers `getAvailableCompilers()` returns all possible input values for `setCompiler(ImplementationProperty)`. Note: the available compilers depend on the currently configured derivative. This method will return the empty collection if no derivative has been configured at the time it is called.

Compiler Set the compiler for the new project with `setCompiler(ImplementationProperty)`. The new default project compiler refers to the first element in the collection returned by `getAvailableCompilers()`. Call `setCompiler(ImplementationProperty)` if you would like change the compiler.

Available Pin Layouts `getAvailablePinLayouts()` returns all possible input values for `setPinLayout(ImplementationProperty)`. Note: the available pin layouts depend on the currently configured derivative. This method will return the empty collection if no derivative has been configured at the time it is called.

Pin Layout Set the pin layout of the selected derivative for the new project with `setPinLayout(ImplementationProperty)`. The new default project pinLayout refers to the first element in the collection returned by `getAvailablePinLayouts()`. Call `setPinLayout(ImplementationProperty)` if you would like change the pinLayout.

6.5.7.4 Project Type Settings

Note: If no projectType settings will be set, the standard type will be used.

Use `type(EEnvironmentProjectType)` to specify the projects type setting.

Use `domain(EEnvironmentProjectTypeDomain)` to specify the projects domain type setting.

```
import com.vector.cfg.core.project.EEnvironmentProjectTypeDomain
import com.vector.cfg.core.project.EEnvironmentProjectType

scriptTask('taskName', DV_APPLICATION) {

  code {
        def tmpFolder = paths.resolveTempPath('projectFolder')
        def newProject = projects.createProject {
            projectName 'NewProject'
            projectFolder tmpFolder

            // Remove those settings to use the Standard ProjectType
            projectType{
                type EEnvironmentProjectType.SOFTWARE_CLUSTER_CONNECTION
                domain EEnvironmentProjectTypeDomain.GLOBAL_RESOURCE_DB
            }
        }
  }
}
```

Listing 6.61: Set project type and domain while project creation

6.5.7.5 Post Build Settings

Use `getPostBuild()` or `postBuild(Action)` to specify the new project's post build settings for Post-build selectable and or loadable projects.

`ICreateProjectPostBuildApi` contains the API to specify the new project's post build settings.

Post-build Loadable Support `setLoadable(boolean)` sets whether or not to support Post-build loadable for the new project. This is an optional parameter defaulting to `false` if the parameter is not provided explicitly.

Post-Build Selectable Support `setSelectable(boolean)` sets whether or not to support Post-build selectable for the new project. This is an optional parameter defaulting to `false` if the parameter is not provided explicitly.

6.5.7.6 Project Folder Settings

Use `getFolders()` or `folders(Action)` to specify the new project's folders settings.

`ICreateProjectFolderApi` contains the methods to specify the new project's folders settings.

```
def newProject = projects.createProject {
  projectName 'NewProject'
  projectFolder projectFolderToSet
  folders {
    applicationComponentFilesFolder tempPath.resolve("applComp")
    autosarFilesFolder tempPath.resolve("autosar")
    ecucFileStructure EEcucFileStructure.SINGLE_FILE
    logFilesFolder tempPath.resolve("logFiles")
    measurementAndCalibrationFilesFolder tempPath.resolve("mAndCal")
    moduleFilesFolder tempPath.resolve("modules")
    serviceComponentFilesFolder tempPath.resolve("serviceComps")
    templatesFolder tempPath.resolve("templates")
    timingExtensionFolder tempPath.resolve("timingExt")
    vttModuleFilesFolder tempPath.resolve("vtt")
  }
}
```

Listing 6.62: Creating a new CFG6 project with project folder

Module Files Folder Set the module files folder for the new project with `setModuleFilesFolder(Object)`. This is an optional parameter defaulting to "../Output/Source/GenData" if the parameter is not provided explicitly. The value given here is converted to `Path` using `ScriptConverters.TO_PATH` 6.15.1 on page 320. Normally an absolute path should be given here, which will be relativized to the correct settings file.

The following constraints apply:

- `Constraints.IS_ABSOLUTE` 6.4.9.2 on page 53

Templates Folder Set the templates folder for the new project with `setTemplatesFolder(Object)`. This is an optional parameter defaulting to "../Output/Source/Templates" if the parameter is not provided explicitly.

The value given here is converted to `Path` using `ScriptConverters.TO_PATH` 6.15.1 on page 320. Normally an absolute path should be given here, which will be relativized to the correct settings file.

The following constraints apply:

- `Constraints.IS_ABSOLUTE` 6.4.9.2 on page 53

Service Components Folder Set the service component files folder for the new project with `setServiceComponentFilesFolder(Object)`. This is an optional parameter defaulting to "../Output/Config/SoftwareComponents" if the parameter is not provided explicitly.

The value given here is converted to `Path` using `ScriptConverters.TO_PATH` 6.15.1 on page 320. Normally an absolute path should be given here, which will be relativized to the correct settings file.

The following constraints apply:

- `Constraints.IS_ABSOLUTE` 6.4.9.2 on page 53

Application Components Folder Set the application component files folder for the new project with `setApplicationComponentFilesFolder(Object)`. This is an optional parameter defaulting to "../Config/AppComponents" if the parameter is not provided explicitly.

The value given here is converted to `Path` using `ScriptConverters.TO_PATH` 6.15.1 on page 320. Normally an absolute path should be given here, which will be relativized to the correct settings file.

The following constraints apply:

- `Constraints.IS_ABSOLUTE` 6.4.9.2 on page 53

Log Files Folder Set the log files folder for the new project with `setLogFilesFolder(Object)`. This is an optional parameter defaulting to "../Output/Log" if the parameter is not provided explicitly.

The value given here is converted to `Path` using `ScriptConverters.TO_PATH` 6.15.1 on page 320. Normally an absolute path should be given here, which will be relativized to the correct settings file.

The following constraints apply:

- `Constraints.IS_ABSOLUTE` 6.4.9.2 on page 53

Measurement And Calibration Files Folder Set the measurement and calibration files folder for the new project with `setMeasurementAndCalibrationFilesFolder(Object)`. This is an optional parameter defaulting to "../Output/Source/McData" if the parameter is not provided explicitly.

The folder object passed to the method is converted to `Path` using `ScriptConverters.TO_PATH` 6.15.1 on page 320. Normally an absolute path should be given here, which will be relativized to the correct settings file.

The following constraints apply:

- `Constraints.IS_ABSOLUTE` 6.4.9.2 on page 53

AUTOSAR Files Folder Set the AUTOSAR files folder for the new project with `setAutosarFilesFolder(Object)`. This is an optional parameter defaulting to "../Config/AUTOSAR" if the parameter is not provided explicitly.

The value given here is converted to `Path` using `ScriptConverters.TO_PATH` 6.15.1 on page 320. Normally an absolute path should be given here, which will be relativized to the correct settings file.

The following constraints apply:

- `Constraints.IS_ABSOLUTE` 6.4.9.2 on page 53

ECUC File Structure The literals of `EEcucFileStructure` define the alternative ECUC file structures supported by the new project. The following alternatives are supported:

`SINGLE_FILE` results in a single ECUC file containing all module configurations.

`ONE_FILE_PER_MODULE` results in a separate ECUC file for each module configuration all located in a common folder.

`ONE_FILE_IN_SEPARATE_FOLDER_PER_MODULE` results in a separate ECUC file for each module configuration each located in its separate folder.

Set the ECUC file structure to use for the new project with the method `setEcucFileStructure(EEcucFileStructure)`. This is an optional parameter defaulting to `EEcucFileStructure.SINGLE_FILE` if the parameter is not provided explicitly.

6.5.7.7 External References

Use `getExternalReferences()` ()} or `externalReferences(Action)` to specify the new project's external references.

DEV Workspace Set the DaVinci Developer workspace for the new project with `setDaVinciDeveloperWorkspace(Object)`. This is an optional parameter defaulting to "../Config/AppConfig" if the parameter is not provided explicitly.

The value given here is converted to `Path` using `ScriptConverters.TO_PATH` 6.15.1 on page 320. Normally an absolute path should be given here, which will be relativized to the correct settings file.

The following constraints apply:

- `Constraints.IS_DCF_FILE` 6.4.9.2 on page 53
- `Constraints.IS_ABSOLUTE` 6.4.9.2 on page 53

6.5.7.8 Additional BSWMD modules

`IProjectAdditionalBswmdApi.getAdditionalBswmd()`

Returns the list of path to the additional BSWMD Module Definition folders.

`IProjectAdditionalBswmdApi.addAdditionalBswmd(Object)`

Adds additional BSWMD files folder. The Path is ignored if it is already present.

`IProjectAdditionalBswmdApi.replaceAdditionalBswmd(Object, Object)`

Replaces an additional BSWMD files folder. If the `#additionalBswmdFolderOld` is not present, the `#additionalBswmdFolderNew` is added at the end similar to `addAdditionalBswmd(Object)`.

`IProjectAdditionalBswmdApi.removeAdditionalBswmd(Object)`

Removes an additional BSWMD files folder. No change if the path to be removed is absent.

`IProjectAdditionalBswmdApi.clearAdditionalBswmds()`

Removes all additional BSWMD files folders.

6.5.8 Opening an existing Project

You can open an existing DaVinci Configurator project with the automation interface.

The method `openProject(Object, Transformer)` opens the project at the given project file location, delegates the given code to the opened `IProject`.

The project is automatically closed after leaving the `code` of the `openProject(Object, Transformer)` method.

The `Object` given as a project file is converted to `Path` using `ScriptConverters.TO_PATH` 6.15.1 on page 320

```
scriptTask('taskName', DV_APPLICATION) {
  code {
    // replace getProjectFileToLoad() with the path to the .dvjson file to be
        loaded
    projects.openProject(getProjectFileToLoad()) {

        // the opened IProject is available inside this Closure
        scriptLogger.info 'Project loaded and ready'
    }
  }
}
```

Listing 6.63: Opening a project from .dvjson file

Opens and migrates an existing DaVinci Configurator Classic dvjson project to the new BSW Package.

The project is automatically closed after leaving the `Closure` code of the `openAndMigrateProject(Object, Transformer)` method.

6.5.8.1 Parameterized Project Load

You can also configure how a Dpa project is loaded, e.g. by disabling the generators. The method `parameterizeProjectLoad(Action)` returns a handle on the project specified by the given `Action`. Using the `IOpenConfiguratorProjectApi`, the `Action` may further customize the project's opening procedure.

The project is not opened until `openProject()` is called on the returned `IProjectRef`.

```
scriptTask('taskName', DV_APPLICATION) {
  code {
    def project = projects.parameterizeProjectLoad {
      // replace getProjectFileToLoad() with the path to the .dvjson file to be
          loaded
      projectFile getProjectFileToLoad()
      // prevent activation of generators and validation
      loadGenerators false
      enableValidation false
    }

    project.openProject {
        // the opened IProject is available inside this Closure
        scriptLogger.info 'Project loaded and ready'
    }
  }
}
```

Listing 6.64: Parameterizing the project open procedure

`IOpenProjectApi` contains the methods for parameterizing the process of opening a project.

DVJSON File The method `setProjectFile(Object)` sets the .dvjson file of the project to be opened. The value given here is converted to `Path` using `ScriptConverters.TO_PATH` 6.15.1 on page 320. The given Path must be absolute.

Generators Using `setLoadGenerators(boolean)` specifies whether or not to activate generators (including their validations) for the opened project.

Validation `setEnableValidation(boolean)` specifies whether to activate validation for the opened project.

6.5.8.2 Open Project Details

`IProjectRef` is a handle on a project not yet loaded but ready to be opened. This could be used to open the project.

`IProjectRef` instances can be obtained from form the following methods:

- `IProjectHandlingApi.createProject(Closure)` 6.5.7 on page 68
- `IProjectHandlingApi.parameterizeProjectLoad(Action)` 6.5.8 on the preceding page

The `IProject` is not really opened until `IProjectRef.openProject(Transformer)` is called. Here, the project is opened and the given `code` block is executed on the opened project. When `IProjectRef.openProject(Transformer)` returns the project has already been closed.

Advanced Open Project Use Cases The method `IProjectRef.advanced()` provides methods for advanced usages of `IProject` instances. For example you can open a project which will not be closed when the open stack frame is left. This can be helpful for unit tests.

- `IProjectRefAdvancedUsage.openProject()`: Open the project and return the `IProject` as reference, but you have to manually close the project.

The `IProjectRefAdvancedUsage` API this only for special use cases, with have very narrow scope. If you are not sure that you need it don't use it.

6.5.9 Create Ecu Configuration Report

The `ICreateEcucReportApi` enables creating an ECU-Configuration report.
Note: If no report settings are set, the defaults will take place.

Use `createEcucReport{}` to specify the Ecu Configuration report settings and create the report in a scope-like way. The report gets created after the scope gets closed.

`ICreateEcucReportApi` is the entry point for setting the Ecu Configuration report settings.

```
scriptTask("TestEcucReportCreation", DV_PROJECT) {
  code { myPath ->
     activeProject {
        // Use closure to get access to the ecuc report settings
        createEcucReport {
          // Set only those options for which to change the default value
          outputFilePath                myPath // default: <ProjectFolder>/Log/
              EcucReport.html
          addAnnotations                true   // default: true
          overwriteExistingReportFile   true   // default: true
          openReportAfterGeneration     false  // default: true
        }
     }
  } // code
} // scriptTask
```

Listing 6.65: Create Ecu Configuration Report

Use `createEcucReport()` to create an Ecu Configuration report with default settings.

```
scriptTask("TestEcucReportCreation", DV_PROJECT) {
  code {
     // Use default ecuc report settings
     activeProject.createEcucReport()
  }
}
```

Listing 6.66: Create Ecu Configuration Report with default settings

6.5.10 Saving a Project

`IProject.saveProject()` saves the current state including all model changes of the project to disc.

```
scriptTask('taskName', DV_APPLICATION) {
  code {
    // replace getDpaFileToLoad() with the path to the .dpa file to be loaded
    def project = projects.openProject(getDpaFileToLoad()) {

      // modify the opened project
      transaction {
        operations.activateModuleConfiguration(bswDefRef.EcuC)
      }

      // save the modified project
      saveProject()
    }
  }
}
```

Listing 6.67: Opening, modifying and saving a project

6.5.11 Opening AUTOSAR Files as Project

Sometimes it could be helpful to load AUTOSAR `arxml` files instead of a full-fledged DaVinci Configurator project. For example to modify the content of a file for test cases with the AutomationInterface, instead of using an XML editor.

You could load multiple `arxml` files into a temporary project, which allowed to read and write the loaded file content with the normal model APIs.

The following elements are loaded by default, without specifying the AUTOSAR files:

- ModuleDefinitions from the BSW: To allow the usage of the BswmdModel
- AUTOSAR standard definition: Refinement resolution of definitions

Caution: Some APIs and services may not be available for this type of project, like:

- Validation: The validation is disabled by default
- Generation: The generators are not loaded by default

The method `parameterizeArxmlFileLoad(Action)` allows to load multiple arxml files into a temporary project. The given `Action` is used to customize the project's opening procedure by the `IOpenArxmlFilesProjectApi`.

The arxml file project is not opened until `openProject()` is called on the returned `IProjectRef`.

```
scriptTask('taskName', DV_APPLICATION) {
  code {
    def project = projects.parameterizeArxmlFileLoad {
      // Add here your arxml files to load
      arxmlFiles(arxmlFilesToLoad)
      rawAutosarDataMode = true
    }
    project.openProject {
        scriptLogger.info 'Project loaded and ready'
    }
  }
}
```

Listing 6.68: Opening Arxml files as project

Arxml Files Add `arxml` files to load with the method `arxmlFiles(Collection)`. Multiple files and method calls are allowed. The given values are converted to `Path` instances using `ScriptConverters.TO_PATH` 6.15.1 on page 320.

Raw AUTOSAR Data Mode the method `setRawAutosarDataMode(boolean)` specifies whether or not to use the raw AUTOSAR data model.

Currently only this mode is supported! You have to set `rawAutosarDataMode = true`.

Note: In raw mode most of the provided services and APIs will disabled, see below for details.

6.5.11.1 Raw AUTOSAR models as Project

Sometimes it could be helpful to create an empty AUTOSAR model or load single ARXML file. This is called raw mode (`IProjectHandlingRawApi`).

You could for example create an empty AUTOSAR model add elements and then export the snippet as an ARXML file.

In raw mode most of the provided services and APIs will disabled, like:

- Ecuc access
- BswmdModel support
- Generation
- Validation
- Workflow
- Domain API
- ChangeInspector
- and more

Empty AUTOSAR model The `emptyAutosarModel(String, AsrPath, BiTransformer)` method creates a new empty AUTOSAR model, only containing one `MIARPackage` created by this method with the path `AsrPath`. The passed AUTOSAR version defines the version of the AUTOSAR model, the version is specified in the format `"4.2.1" or "4.0.3", ...`

```
scriptTask("taskName", DV_APPLICATION) {
  code {
    def asrPkgToCreate = AsrPath.create("/MyPkg")
    def autosarVersion = "4.2.1"

    projects.raw.emptyAutosarModel(autosarVersion, asrPkgToCreate) {
      modelProject, myPkg ->
      // modelProject is the created IProject
      // myPkg is the MIARPackage specified above with asrPkgToCreate

      // Now you could use the model like any other project:
      transaction{
        // For example create a new sub package:
        def mySubPkg = myPkg.withSubPackage().byNameOrCreate("MySubPkg")
      }

      // Then export the package content
      def exportFolder = paths.getTempFolder()
      persistency.modelExport.exportModelTree(exportFolder, myPkg)
    }
  }
}
```

Listing 6.69: Create an empty AUTOSAR model

6.6 Model

6.6.1 Introduction

The model API provides means to retrieve AUTOSAR model content and to modify AUTOSAR data. This comprises Ecuc data (module configurations and their content) and System Description data.

In this chapter you'll first find a brief introduction into the model handling. Here you also find some simple cut-and-paste examples which allow starting easily with low effort. Subsequent sections describe more and more details which you can read if required.

Chapter 7 on page 324 may additionally be useful to understand detailed concepts and as a reference to handle special use cases.

6.6.2 Getting Started

The model API basically provides two different approaches:

- The **MDF model** is the low level AUTOSAR model. It stores all data read from AUTOSAR XML files. Its structure is based on the AUTOSAR MetaModel which can be found for example on the AUTOSAR website. In 7.1 on page 324 you find detailed information about this model.
- The **BswmdModel** is a model which wraps the MDF model to provide convenient and type-safe access to the Ecuc data. It contains, definition based classes for module configurations, containers, parameters and references. The class `CanGeneral` for example as type-safe implementation in contrast to the generic AUTOSAR class `MIContainer` in MDF.

It is strongly recommended to use the BswmdModel model to deal with Ecuc data because it simplifies scripting a lot.

6.6.2.1 Read the ActiveEcuc

This section provides some typical examples as a brief introduction for reading the Ecuc by means of the BswmdModel. See chapter 6.6.3.2 on page 90 for more details.

The following example specifies no types for the local variables. It therefore requires no import statements. A drawback on the other hand is that the type is only known at runtime and you have no type support in the IDE:

```
scriptTask("TaskName"){
    code {
        // Gets the module DefRef searching all definitions of this SIP
        def moduleDefRef = bswDefRef.EcuC

        // Creates all BswmdModel instances with this definition. A List<EcuC> in
            this case.
        def ecucModules = bswmdModel(moduleDefRef)

        // Gets the EcucGeneral container of the first found module instance
        def ecuc = ecucModules.single
        def ecucGeneral = ecuc.ecucGeneral

        // Gets an (enum) parameter of this container
        def cpuType = ecucGeneral.CPUType
    }
}
```

Listing 6.70: Read with BswmdModel objects starting with a module DefRef (no type declaration)

In contrast to the listing above the next one implements the same behavior but specifies all types:

```
// Required imports
import com.vector.cfg.automation.model.ecuc.microsar.ecuc.EcuC
import com.vector.cfg.automation.model.ecuc.microsar.ecuc.ecucgeneral.EcucGeneral
import com.vector.cfg.automation.model.ecuc.microsar.ecuc.ecucgeneral.cputype.
    CPUType
import com.vector.cfg.automation.model.ecuc.microsar.ecuc.ecucgeneral.cputype.
    ECPUType

scriptTask("TaskName"){
    code {
        // Gets the ecuc module configuration
        EcuC ecuc = bswmdModel(EcuC).single

        // Gets the EcucGeneral container
        EcucGeneral ecucGeneral = ecuc.ecucGeneral

        // Gets an enum parameter of this container
        CPUType cpuType = ecucGeneral.CPUType
        if (cpuType.value == ECPUType.CPU32Bit) {
            "Do something ..."
        }
    }
}
```

Listing 6.71: Read with BswmdModel objects starting with a module class (strong typing)

The `bswmdModel()` API takes an optional closure argument which is being called for each created BswmdModel object. This object is used as parameter of the closure:

```
// Required imports
import com.vector.cfg.automation.model.ecuc.microsar.ecuc.EcuC
import com.vector.cfg.automation.model.ecuc.microsar.ecuc.ecucgeneral.cputype.
    ECPUType

scriptTask("TaskName"){
    code {
        // Executes the closure with all instances of this definition
        bswmdModel(EcuC) {
            // The related BswmdModel instance is parameter of this closure
            ecuc ->

            if (ecuc.ecucGeneral.CPUType.value == ECPUType.CPU32Bit) {
                "Do something ..."
            }
        }
    }
}
```

Listing 6.72: Read with BswmdModel objects with closure argument

Additionally, to the DefRef, an already available MDF model object can be specified to create the related BswmdModel object for it:

```
// Required imports
import com.vector.cfg.automation.model.ecuc.microsar.ecuc.ecucgeneral.EcucGeneral
import com.vector.cfg.automation.model.ecuc.microsar.ecuc.ecucgeneral.cputype.
    ECPUType

scriptTask("TaskName"){
    code {
        // Gets the MDF model instance of the Ecuc General container
        def container = mdfModel(EcucGeneral.DefRef).single

        // Executes the closure with this MDF object instance
        bswmdModel(container, EcucGeneral.DefRef) {
            // The related BswmdModel instance is parameter of this closure
            ecucGeneral ->

            if (ecucGeneral.CPUType.value == ECPUType.CPU32Bit) {
                "Do something ..."
            }
        }
    }
}
```

Listing 6.73: Read with BswmdModel object for an MDF model object

For a generic access to Ecu configuration structure (e.g. to use the script with different BSW packages and different platforms/derivatives) the untyped model in combination with `BswDefRefs`

can be used. See chapter 6.6.3.5 on page 92 and 6.6.3.7 on page 94 for more details:

```
// Required imports
import com.vector.cfg.gen.core.bswmdmodel.GIContainer
import com.vector.cfg.gen.core.bswmdmodel.GIParameter

scriptTask("TaskName"){
    code {

        GIContainer ecucGen = bswmdModel(bswDefRef.EcucGeneral).single

        GIParameter<Boolean> ecuCSafeBswChecks = ecucGen.getParameter(bswDefRef.
            EcuCSafeBswChecks)

        if (ecuCSafeBswChecks.valueMdf.booleanValue()) {
            "Do something ..."
        }
    }
}
```

Listing 6.74: Read with BswmdModel objects with the untyped model (DefRefAPI)

6.6.2.2 Write the ActiveEcuc

This section provides some typical examples as a brief introduction for writing the Ecuc by means of the BswmdModel. See chapter 6.6.3.3 on page 91 for more details.

For the most cases the entry point for writing the ActiveEcuc is a (existing) module configuration object which can be retrieved with the `bswmdModel()` API. Because the model is in read-only state by default, every call to an API which creates or deletes elements has to be executed in a `transaction()` block.

```
// Required imports
import com.vector.cfg.automation.model.ecuc.microsar.ecuc.EcuC
import com.vector.cfg.automation.model.ecuc.microsar.ecuc.ecucgeneral.EcucGeneral

scriptTask("TaskName"){
    code {
        transaction {
            // Gets the first found ecuc module instance
            EcuC ecuc = bswmdModel(EcuC).single

            //Gets the EcucGeneral container or create one if it is missing
            EcucGeneral ecucGeneral = ecuc.ecucGeneralOrCreate

            // Gets an boolean parameter of this container or create one if it is
                missing
            def ecuCSafeBswChecks = ecucGeneral.ecuCSafeBswChecksOrCreate

            // Sets the parameter value to true
            ecuCSafeBswChecks.value = true
        }
       saveProject()
    }}
```

Listing 6.75: Write with BswmdModel required/optional objects

```
// Required imports
import com.vector.cfg.automation.model.ecuc.microsar.ecuc.EcuC
import com.vector.cfg.automation.model.ecuc.microsar.ecuc.ecuchardware.
    ecuccoredefinition.EcucCoreDefinition

scriptTask("TaskName"){
    code {
        transaction {
            // Gets the first found ecuc module instance
            EcuC ecuc = bswmdModel(EcuC).single

            //Gets the EcucCoreDefinition list (creates ecucHardware if it is
                missing)
            def ecucCoreDefinitions = ecuc.ecucHardwareOrCreate.ecucCoreDefinition

            //Adds two EcucCores
            EcucCoreDefinition core0 = ecucCoreDefinitions.createAndAdd("EcucCore0
                ")
            EcucCoreDefinition core1 = ecucCoreDefinitions.createAndAdd("EcucCore1
                ")

            if(ecucCoreDefinitions.exists("EcucCore0")) {
                //Sets EcucCoreId to 0
                ecucCoreDefinitions.byName("EcucCore0").ecucCoreId.setValue(0)
            }

            //Creates a new EcucCore by method 'byNameOrCreate'
            EcucCoreDefinition core2 = ecucCoreDefinitions.byNameOrCreate("
                EcucCore2")
        }
        saveProject()
}}
```

Listing 6.76: Write with BswmdModel multiple objects

```
// Required imports
import com.vector.cfg.automation.model.ecuc.microsar.ecuc.EcuC
import com.vector.cfg.automation.model.ecuc.microsar.ecuc.ecucgeneral.EcucGeneral

scriptTask("TaskName"){
    code {
        transaction {
            // Gets the first found ecuc module instance
            EcuC ecuc = bswmdModel(EcuC).single

            //Duplicates container 'EcucGeneral' and all its children
            EcucGeneral ecucGeneral_Dup = ecuc.ecucGeneral.duplicate()
        }
        saveProject()
    }
}
```

Listing 6.77: Write with BswmdModel - Duplicate a container

```
// Required imports
import com.vector.cfg.automation.model.ecuc.microsar.ecuc.ecucgeneral.EcucGeneral

scriptTask("TaskName"){
    code {
        transaction {
            // Gets the first found ecuc module instance
            EcucGeneral ecucGeneral = bswmdModel(EcucGeneral).single

            //Deletes 'ecucGeneral' from model
            ecucGeneral.moRemove()

            //Checks if the container 'ecucGeneral' was removed from repository
            if(ecucGeneral.moIsRemoved()) {
                "Do something ..."
            }
        }
       saveProject()
    }
}
```

Listing 6.78: Write with BswmdModel - Delete elements

6.6.2.3 Read the SystemDescription

This section contains only one example for reading the SystemDescription by means of the MDF model. See chapter 6.6.4.1 on page 95 for more details.

```
// Required imports
import com.vector.cfg.model.mdf.ar4x.swcomponenttemplate.datatype.dataprototypes.*
import com.vector.cfg.model.mdf.ar4x.commonstructure.datadefproperties.*

scriptTask("mdfModel", DV_PROJECT){
    code {
        // Create a type-safe AUTOSAR path
        def asrPath =
            AsrPath.create("/PortInterfaces/PiSignal_Dummy/DeSignal_Dummy",
                MIVariableDataPrototype)

        // Enter the MDF model tree starting at the object with this path
        mdfModel(asrPath) { MIVariableDataPrototype prototype ->

            // Traverse down to the swDataDefProps
            prototype.swDataDefProps.with { MISwDataDefProps swDataDefPropsParam
                ->

                // swDataDefPropsVariant is a List<MISwDataDefPropsConditional>
                // Execute the following for ALL elements of this List
                swDataDefPropsParam.swDataDefPropsVariant.each {
                   MISwDataDefPropsConditional swDataDefPropsCondParam ->

                    // Resolve the dataConstr reference (type MIDataConstr)
                    def target = swDataDefPropsCondParam.dataConstr.refTarget

                    // Get the swCalibrationAccess enum value
                    def access = swDataDefPropsCondParam.swCalibrationAccess
                    assert access == MISwCalibrationAccessEnum.NOT_ACCESSIBLE
                }
            }
        }
    }
}
```

Listing 6.79: Read system description starting with an AUTOSAR path in closure

The same sample as above, but in property access style instead of closures:

```
// Create a type-safe AUTOSAR path
def asrPath =
    AsrPath.create("/PortInterfaces/PiSignal_Dummy/DeSignal_Dummy",
        MIVariableDataPrototype)

def prototype = mdfModel(asrPath)
def swDataDefPropsParam = prototype.swDataDefProps

// Execute the following for ALL swDataDefPropsVariant
swDataDefPropsParam.swDataDefPropsVariant.each{ swDataDefPropsCondParam ->
    //  Resolve the dataConstr reference (type MIDataConstr)
    def target = swDataDefPropsCondParam.dataConstr.refTarget

    // Get the swCalibrationAccess enum value
    def access = swDataDefPropsCondParam.swCalibrationAccess
    assert access == MISwCalibrationAccessEnum.NOT_ACCESSIBLE
}
```

Listing 6.80: Read system description starting with an AUTOSAR path in property style

6.6.2.4 Write the SystemDescription

Writing the system description looks quite similar to the reading, but you have to use methods like (see chapter 6.6.4.3 on page 100 for more details):

- `get<Element>OrCreate()` or `<element>OrCreate`
- `createAndAdd()`
- `byNameOrCreate()`

You have to open a transaction before you can modify the MDF model, see chapter 6.6.6 on page 116 for details.

The following samples show the different types of write API:

```
transaction{
    // The asrPath points to an MIVariableDataPrototype
    mdfModel(asrPath) { dataPrototype ->
        dataPrototype.category = "NewCategory"
    }
}
```

Listing 6.81: Changing a simple property of an MIVariableDataPrototype

```
transaction{
    // The asrPath points to an MIVariableDataPrototype
    mdfModel(asrPath) {
        int count = 0
        assert adminData == null
        withAdminData().orCreate.with {
            count++
        }
        assert count == 1
        assert adminData != null
    }
}
```

Listing 6.82: Creating non-existing member by navigating into its content with OrCreate()

```
transaction{
    // The asrPath points to an MIVariableDataPrototype
    mdfModel(asrPath) {
        assert adminData.sdg.empty

        adminData.with {
            withSdg().create{
                it.gid = "NewGidValue"
            }
        }

        assert adminData.sdg.first.gid == "NewGidValue"
    }
}
```

Listing 6.83: Creating new members of child lists with createAndAdd() by type

```
transaction{
    // The path points to an MISenderReceiverInterface
    mdfModel(asrPath) { MISenderReceiverInterface sendRecIf ->
        def dataElementRelation = sendRecIf.withDataElement()

        def dataElement = dataElementRelation.byNameOrCreate("MyDataElement")
        dataElement.name = "NewName"

        def dataElement2 = dataElementRelation.byNameOrCreate("NewName")

        assert dataElement == dataElement2
    }
}
```

Listing 6.84: Updating existing members of child lists with byNameOrCreate() by type

6.6.3 BswmdModel in AutomationInterface

The AutomationInterface contains a generated BswmdModel. The BswmdModel provides classes for all Ecuc elements of the AUTOSAR model (ModuleConfigurations, Containers, Parameter, References). The BswmdModel is automatically generated from the SIP of the DaVinci Configurator.

You should use the BswmdModel whenever possible to access Ecuc elements of the AUTOSAR model. For accessing the Ecuc elements with the BswmdModel, see chapter 6.6.3.2.

For a detailed description of the BswmdModel, see chapter 7.3.1 on page 337.

6.6.3.1 BswmdModel Package and Class Names

The generated model is contained in the Java package `com.vector.cfg.automation.model.ecuc`. Every Module has its own sub packages with the name:

- `com.vector.cfg.automation.model.ecuc.<AUTOSAR-PKG>.<SHORTNAME>`
  - e.g. `com.vector.cfg.automation.model.ecuc.microsar.dio`
  - e.g. `com.vector.cfg.automation.model.ecuc.autosar.ecucdefs.can`

The packages then contain the class of the element like `Dio` for the module. The full path would be `com.vector.cfg.automation.model.ecuc.microsar.dio.Dio`.

For the container `DioGeneral` it would be:

- `com.vector.cfg.automation.model.ecuc.microsar.dio.diogeneral.DioGeneral`

To use the BswmdModel in script files, you have to write an import, when accessing the class:

```
//The required BswmdModel import of the class Dio
import com.vector.cfg.automation.model.ecuc.microsar.dio.Dio

scriptTask("TaskName"){
    code{
        Dio.DefRef //Usage of the class Dio
    }
}
```

Listing 6.85: BswmdModel usage with import

6.6.3.2 Reading with BswmdModel

The `bswmdModel()` methods provide entry points to start navigation through the ActiveEcuc. Client code can use the `Action/groovy.lang.Closure` overloads to navigate into the content of the found bswmd objects. Inside the called code the related bswmd object is available as closure parameter.

The following types of entry points are provided here:

- `bswmdModel(WrappedTypedDefRef)` searches all objects with the specified definition and returns the BswmdModel instances.
- `bswmdModel(Class)` searches all objects with the specified class and returns the BswmdModel instances. Finds the same elements as above.

- `bswmdModel(MIHasDefinition, WrappedTypedDefRef)` returns the BswmdModel instance for the provided MDF model instance.
- `bswmdModel(Class, String)` searches all objects with the specified class and the matching path, see `IMdfModelApi#mdfModel(String)` or chapter 6.6.4.2 on page 98 for details.

When a closure is being used, the object found by `bswmdModel()` is provided as parameter when the closure is called.

The `bswmdModel()` method itself returns the found objects too. Retrieving the objects member and children (Container, Parameter) as properties or methods are then possible directly using the returned object.

Examples:

```
code {
    // Gets the ecuc module configuration
    EcuC ecuc = bswmdModel(EcuC).single
}
```

Listing 6.86: Read with BswmdModel the EcuC module configuration

Or the same with a `DefRef` instead of a `Class`:

```
code {
    // Gets the ecuc module configuration
    EcuC ecuc = bswmdModel(EcuC.DefRef).single
}
```

Listing 6.87: Read with BswmdModel the EcuC module configuration with DefRef

For more usage samples please see chapter 6.6.2.1 on page 80.

6.6.3.3 Writing with BswmdModel

As well as for reading with BswmdModel the entry points for writing with BswmdModel are also the `bswmdModel()` methods. There has to be at least one existing element in the ActiveEcuc from which the navigation can be started. For the most cases the entry point for writing the ActiveEcuc is the module configuration.

Example:

```
code {
    transaction {
        // Gets the ecuc module configuration
        EcuC ecuc = bswmdModel(EcuC).single

        //Gets the EcucGeneral container or create one if it is missing
        EcucGeneral ecucGeneral = ecuc.ecucGeneralOrCreate
    }
   saveProject()
}
```

Listing 6.88: Write with BswmdModel the EcucGeneral container

For more usage samples please see chapter 6.6.2.2 on page 84.

The model is in read-only state by default, so no objects could be created. For this reason all calls which creates or deletes elements has to be executed within a `transaction()` block.

See 7.3.1.9 on page 345 for more details about the BswmdModel write API.

6.6.3.4 Declaration with BswmdModel

The BswmdModel supports declaration API to declare an AUTOSAR ECU configuration structure in code, which is then synchronized with the existing structure to create elements in a declarative way.

The model is in read-only state by default, so no objects could be created. For this reason all calls which creates or deletes elements has to be executed within a `transaction()` block.

Example:

```
code {
    transaction {
        EcuC ecuc = bswmdModel(EcuC).single
        ecuc.declare {
            EcucGeneral {
                EcuCSafeBswChecks(true)
            }
        }
    }
}
```

Listing 6.89: Usage of BswmdModel Declaration API with the EcucGeneral container

See 7.3.1.10 on page 349 for more details about the BswmdModel Declaration API.

6.6.3.5 Bsw DefRefs

The `sipDefRef` API provides access to retrieve generated `DefRef` instances from the SIP without knowing the correct Java/Groovy imports. This is mainly useful in script files, where no IDE helps with the imports.

If you are using an Automation Script Project you can ignore this API and use the `DefRef`s provided by the generated classes, which is superior to this API, because they are typesafe and compile time checked. See 6.6.3.6 on the following page for details.

The listing show the usage of the `bswDefRef` API with short names and definition paths.

```
code{
  def theDefRef
  // You can call bswDefRef.<ShortName>
  theDefRef = bswDefRef.EcucGeneral
  theDefRef = bswDefRef.Dio
  theDefRef = bswDefRef.DioPort

  // Or you can use the [] notation
  theDefRef = bswDefRef["Dio"]
  theDefRef = bswDefRef["DioChannelGroup"]

  // If the DefRef is not unique you have to specify the full definition
  theDefRef = bswDefRef["/MICROSAR/EcuC/EcucGeneral"]
  theDefRef = bswDefRef["/MICROSAR/Dio"]
  theDefRef = bswDefRef["/MICROSAR/Dio/DioConfig/DioPort"]

  //Wildcards are also allowed
  theDefRef = bswDefRef["/[ANY]/Adc"]
}
```

Listing 6.90: Usage of the bswDefRef API to retrieve DefRefs in script files

You can also check if a certain `DefRef` exists in the currently loaded SIP. The method `hasDefRef(String)` returns `true`, if the definition exists. This is helpful to check for existence of the definition before using it to prevent e.g. `LinkageErrors`.

```
if(bswDefRef.hasDefRef("Dio")){
  // Now we know the Dio module exists in the SIP
  def theDefRef = bswDefRef.Dio
}
```

Listing 6.91: Check if a definition exists in the SIP

6.6.3.6 BswmdModel DefRefs

The generated BswmdModel classes contain `DefRef` instances for each definition element (Modules, Containers, Parameters). You should always prefer this API over the Sip DefRefs, because this is type safe and checked during compile time.

You can use the DefRefs by calling `<ModelClassName>.DefRef`. The literal `DefRef` is a `static constant` in the generated classes.

For simple parameters like Strings, Integer there is no generated class, so you have to call the method on its parent container like `<ParentContainerClass>.<ParameterShortName>DefRef`.

There exist generated classes for Parameters of type `Enumeration` and `References` to Container and therefore you have both ways to access the DefRef:

- `<ModelClassName>.DefRef` or
- `<ParentContainerClass>.<ParameterShortName>DefRef`

To use the DefRefs of the classes you have to add imports in script files, see chapter 6.6.3.1 on page 90 for required import names.

```
// Required imports
import com.vector.cfg.automation.model.ecuc.microsar.ecuc.ecucgeneral.EcucGeneral
import com.vector.cfg.automation.model.ecuc.microsar.ecuc.ecucgeneral.cputype.
    CPUType

scriptTask("TaskName"){
  code {
    def theDefRef

    //DefRef from EcucGeneral container
    theDefRef = EcucGeneral.DefRef

    //DefRef from generated parameter
    theDefRef = CPUType.DefRef
    //Or the same
    theDefRef = EcucGeneral.CPUTypeDefRef

    //DefRef from simple parameter
    theDefRef = EcucGeneral.AtomicBitAccessInBitfieldDefRef
    theDefRef = EcucGeneral.DummyFunctionDefRef
  }
}
```

Listing 6.92: Usage of generated DefRefs form the bswmd model

6.6.3.7 Untyped Model with the DefRef API

The untyped Model provides a generic access to the Ecu configuration structure via `DefRefs`. There are NO generated classes for the Definition structure.

To use the untyped Model, the `BswDefRefs` can be used:

```
// Required imports
import com.vector.cfg.gen.core.bswmdmodel.GIModuleConfiguration
import com.vector.cfg.gen.core.bswmdmodel.GIContainer
import com.vector.cfg.gen.core.bswmdmodel.GIParameter

scriptTask("TaskName"){
    code {

        GIModuleConfiguration ecuc = bswmdModel(bswDefRef.EcuC).single

        // If the short name is not unique, you can use the full definition as
            string (DefRef wildcards are allowed, e.g. [/ANY])

        GIContainer ecucPduCollection = ecuc.getSubContainer(bswDefRef["/MICROSAR/
            EcuC/EcucPduCollection"])

        List<GIContainer> pdus = ecucPduCollection.getSubContainers(bswDefRef["/
            MICROSAR/EcuC/EcucPduCollection/Pdu"])

        GIContainer pdu = pdus.get(0)

        GIParameter<Integer> pduLength = pdu.getParameter(bswDefRef["/MICROSAR/
            EcuC/EcucPduCollection/Pdu/PduLength"])

        "PduLength: " + pduLength.getValueMdf().intValue()
    }
}
```

Listing 6.93: Usage of the untyped BswmdModel with BswDefRefs

See chapter 7.3.1.2 on page 339 for more details.

6.6.3.8 Switching from Domain Models to BswmdModel

You can switch from domain models to the BswmdModel, if the domain model is backed by ActiveEcuC elements. Please read the documentation of the different domain models, for whether this is possible for a certain domain model.

To switch from a domain model to the BswmdModel, you can call one of the methods for `IHasModelObjects` like, `bswmdModel(IHasModelObject, WrappedTypedDefRef)`. But you need a `DefRef` to get the type safe BswmdModel object. The domain model documents, which DefRef must be used for the certain domain model object.

```
// Domain model object of the communication domain
ICanController canDomainModel = ...

def canControllerBswmd = canDomainModel.bswmdModel(CanController.DefRef)

// Or use a closure
canDomainModel.bswmdModel(CanController.DefRef){ canControllerBswmd ->
  //Use the bswmd object
}
```

Listing 6.94: Switch from a domain model object to the corresponding BswmdModel object

6.6.4 MDF Model in AutomationInterface

Access to the MDF model is required in all areas which are not covered by the BswmdModel. This is the SystemDescription (non-Ecuc data) and details of the Ecuc model which are not covered by the BswmdModel.

The MDF model implements the raw AUTOSAR data model and is based on the AUTOSAR meta-model. For details about the MDF model, see chapter 7.1 on page 324.

For more details concerning the methods mentioned in this chapter, you should also read the JavaDoc sections in the described interfaces and classes.

6.6.4.1 Reading the MDF Model

The `mdfModel()` methods provide entry points to start navigation through the MDF model. Client code can use the `Closure` overloads to navigate into the content of the found MDF objects. Inside the called closure the related MDF object is available as closure parameter.

The following types of entry points are provided here:

- `mdfModel(TypedAsrPath)` searches an object with the specified AUTOSAR path
- `mdfModel(TypedDefRef)` searches all objects with the specified definition
- `mdfModel(Class)` searches all objects with the specified model type (meta class)
- `mdfModel(String)` searches for model elements with by different properties, see 6.6.4.2 on page 98 for details.
- `mdfModel(MIObject, String)` searches for model elements by giving a root element and a relative path, see 6.6.4.2 on page 99 for details.

When a closure is being used, the object found by `mdfModel()` is provided as parameter when this closure is called:

```
code {
    // Create a type-safe AUTOSAR path for a MIVariableDataPrototype
    def asrPath =
        AsrPath.create("/PortInterfaces/PiSignal_Dummy/DeSignal_Dummy",
            MIVariableDataPrototype)

    // Use the Java-Style syntax
    def dataDefPropsMdf = mdfModel(asrPath).swDataDefProps

    // Or use the Closure syntax to navigate

    // Enter the MDF model tree starting at the object with this path
    mdfModel(asrPath) {
        // Parameter type is MIVariableDataPrototype:
        dataPrototype ->

            // Traverse down to the swDataDefProps
            dataPrototype.swDataDefProps.each {MISwDataDefProps props ->
                scriptLogger.info "Do something ..."
            }
    }

   saveProject()
}
```

Listing 6.95: Navigate into an MDF object starting with an AUTOSAR path

The `mdfModel()` method itself returns the found object too. Retrieving the objects member (as property) is then possible directly using the returned object.

Naming of the interface classes to create the type safe AUTOSAR path is described in chapter 7.1 on page 324.

An alternative is using a closure to navigate into the MDF object and access its member there:

```
// Get an MDF object and get its members directly
def obj = mdfModel(asrPath)     // Type MIVariableDataPrototype
def props = obj.swDataDefProps  // Type MISwDataDefProps

// Get an MDF object and get its members using a closure
def props2
def obj2 = mdfModel(asrPath) {
    props2 = swDataDefProps
}

// The results are the same
assert obj == obj2
assert props == props2
```

Listing 6.96: Find an MDF object and retrieve some content data

Closures can be nested to navigate deeply into the MDF model tree:

```
mdfModel(asrPath) {
    int count = 0
    swDataDefProps.with {
        // swDataDefPropsVariant is a List<MISwDataDefPropsConditional>
        // Execute the following for ALL elements of this List
        List v = swDataDefPropsVariant.each {
            scriptLogger.info "Do something ..."
            count++
        }
    }
    assert count >= 1
}
```

Listing 6.97: Navigating deeply into an MDF object with nested closures

When a member doesn't exist during navigation into a deep MDF model tree, the specified closure is not called:

```
mdfModel(asrPath) {
    int count = 0
    assert adminData == null
    adminData?.with {
        count++
    }
    assert count == 0
}
```

Listing 6.98: Ignoring non-existing member closures

Retrieving a Child by Shortname or Definition There are multiple ways to retrieve children from an MDF model object, by the shortname or by its definition. The shortname can be used at the object with `childByName()` or at the child list with `byName()`.

childByName The `childByName(MIARObject, String, Action)` method calls the passed Action, if the request child exists. And returns the child [MIReferrable] below the specified object which has this relative AUTOSAR path (not starting with '/').

```
MIContainer canGeneral = ...
canGeneral.childByName("CanMainFunctionRWPeriods"){ child->
     //Do something
}
```

Listing 6.99: Get a MIReferrable child object by name

Lists containing Referrables

- The method `byName(String)` retrieves the child with the shortname, or `null`, if no child exists with this shortname.
- The method `byName(String, Closure)` retrieves the child with the shortname, or `null`, if no child exists with this shortname. Then the closure is executed with the child as closure parameter, if the child is not `null`. The child is finally returned.
- The method `byName(Class, String)` retrieves the child with the shortname and type, or `null`, if no child exists with this shortname.

- The method `byName(Class, String, Closure)` retrieves the child with the shortname and type, or `null`, if no child exists with this shortname. Then the closure is executed with the child as closure parameter, if the child is not `null`. The child is finally returned.
- The method `getAt(String)` all members with this relative AUTOSAR path. Groovy also allows to write `list["ShortnameToSearchFor"]`.

```
// The asrPath points to an MISenderReceiverInterface
MISenderReceiverInterface prototype = mdfModel(asrPath)

// byName() with shortname
def data1 = prototype.withDataElement().byName("DeSignal_Dummy")
assert data1.name == "DeSignal_Dummy"
```

Listing 6.100: Retrieve child from list with byName()

Lists containing Parameters and Containers

- The method `getAt(TypedDefRef)` returns all children with the passed definition. Groovy also allows to write `list[DefRef]`.

6.6.4.2 Reading the MDF Model by String

The method `mdfModel(String)` searches for model elements by multiple ways at once. The method evaluates the specified property in the following order, it will continue, if nothing was found:

- AUTOSAR path, see `mdfModel(AsrPath)`, if the path begins with an '/' and the model element is no definition object (`MIParamConfMultiplicity`)
  - Example: `/ActiveEcuc/MyCan/MyContainer`
- ObjectLink, see `AsrObjectLink`, if the path begins with an '/' and the model element is no definition object (type `MIParamConfMultiplicity`)
  - Example: `/ActiveEcuc/MyCan/MyContainer[0:ParameterDef]`
- Definition path, see `mdfModel(DefRef)`, if the path begins with an '/'
  - Example: `/MICROSAR/Can2`
- Relative path, see `mdfModel(MIObject, String)`, the relative path may not start with an '/'. See 6.6.4.2 on the next page for more details.
- MICROSAR QUERY, if the path begins with **"msrq:"**. The defined Microsar Query, filters the configuration elements
  by the given arbitrary filter code. The filter must be evaluable to a `String`, `Boolean` or `Pattern`. The Microsar Query can be used for modules, containers and parameters. See 6.6.4.2 on page 100 for more details.
- AUTOSAR path relative to the ActiveEcuc package, if it does not begin with an '/'
  - Example: `MyCan/MyContainer`
- Definition path as `DefRef` with wildcard `ANY` starting at the moduleConfiguration, if it does not begin with an '/'
  - Example: `Can/CanGeneral`

- Definition path as `DefRef` with wildcards, if it does begin with a valid wildcard like `/[ANY]`, see `EDefRefWildcard`.
  - Example: `/[ANY]/Can/CanGeneral`
- Shortname of an `MIARElement` if the path does not contain any '/'.
  - Example: `MyContainer`

This method does **not** limit the search to the ActiveEcuC, so it can be used to retrieve any object with the path `String`.

Remark: Even in post-build selectable variant models this method expects to find at most one object because script code will never run in an unfiltered context.

Caution: This is a potentially slow operation, you should use other `mdfModel()` methods, if possible. Because this method must traverse the whole model in some cases.

```
def moduleCfg1 = mdfModel("/ActiveEcuC/Can").single
def moduleCfg2 = mdfModel("Can").single
def moduleCfg3 = mdfModel("/[ANY]/Can").single
def parameter  = mdfModel("/ActiveEcuc/MyCan/MyContainer[0:ParameterDef]").
    singleOrNull
```

Listing 6.101: Get elements with mdfModel(String)

Relative search - `mdfModel(MIObject, String)` Retrieves model elements based on the root element. The system navigates relative to the model element based on the root element. The relative path may not start with an '/'. In case of a variant project the collection may have more than one entry.

```
// Required imports
import com.vector.cfg.model.access.AsrPath
import com.vector.cfg.model.mdf.model.autosar.ecucparamdef.MIContainerDef

scriptTask("mdfModel", DV_PROJECT){
    code {
        // Reading a definition element
        def asrPath = AsrPath.create("/MICROSAR/Can_CanoeemuCanoe/Can/CanConfigSet
            ", MIContainerDef)
        def root = mdfModel(asrPath)
        def reqElem = mdfModel(root, "CanController/CanFilterMask").getFirst()
    }
}
```

Listing 6.102: Read definitions elements with a relative path using the mdfModel

```
// Required imports
import com.vector.cfg.model.access.AsrPath
import com.vector.cfg.model.mdf.model.autosar.ecucdescription.MIContainer

scriptTask("mdfModel", DV_PROJECT){
    code {
        // Reading an activeEcuc element
        def asrPath = AsrPath.create("/ActiveEcuC/Can/CanConfigSet", MIContainer)
        def root = mdfModel(asrPath)
        def reqElem = mdfModel(root, "
            ECU_T_CTP_1_NWT_CTP_CANH_ak72ea5qpue3dstlfi5v4312z_090525f9_Rx_Ext").
            getFirst()
    }
}
```

Listing 6.103: Read activeEcuc elements with a relative path using the mdfModel

Msrq search - `msrQuery(String path)` The method `msrQuery(String)` searches for model elements by using an arbitrary filter code as closure. The method evaluates the specified pattern and returns the matching model elements. If nothing was found, it returns an empty list.

The input string defined as an MICROSAR QUERY, filters the configuration elements by the given arbitrary filter code. The arbitrary filter code must be defined inside of the **{ }** . The filter code must be evaluable to a `String`, `Boolean` or `Pattern`.

Examples:

- `/MICROSAR/Crc/CrcGeneral{ true }`
- `/MICROSAR/Crc/CrcGeneral{ ~"[\\w]*[123l]\$" }`
- `/MICROSAR/Crc/CrcGeneral{ "CrcGeneral" }`
- `/MICROSAR/Crc/CrcGeneral{ it.getName() == "CrcGeneral" }`
- `/MICROSAR/Crc/CrcGeneral{ elem -> elem.getName() == "CrcGeneral" }`
- `/MICROSAR/Crc/CrcGeneral{ getName() == "CrcGeneral" }`
- `/MICROSAR/Crc/CrcGeneral{ it.getName().contains("CrcGeneral") }`
- `/[ANY]/Crc/CrcGeneral/{ true }`

6.6.4.3 Writing the MDF Model

Writing to the MDF model can be done with the same `mdfModel(AsrPath)` API, but you have to call specific methods to modify the model objects. The methods are devided in the following use cases:

- Change a simple property like `Strings`
- Change or create a single child relateion (0:1)
- Create a new child for a child list (0:*)
- Update an existing child from a child list (0:*)

You have to open a transaction before you can modify the MDF model, see chapter 6.6.6 on page 116 for details about transactions.

6.6.4.4 Simple Property Changes

The properties of MDF model object simply be changed by with the setter method of the model object. Simple setter exist for example for the types:

- `String`
- `Enums`
- `Integer`
- `Double`

```
transaction{
    // The asrPath points to an MIVariableDataPrototype
    mdfModel(asrPath) { dataPrototype ->
        dataPrototype.category = "NewCategory"
    }
}
```

Listing 6.104: Changing a simple property of an MIVariableDataPrototype

6.6.4.5 Creating single Child Members (0:1)

For single child members (0:1), the automation API provides and additional method for the getter `get<Element>OrCreate()` for convenient child object creation. The methods will create the element, instead of returning `null`.

```
transaction{
    // The asrPath points to an MIVariableDataPrototype
    mdfModel(asrPath) {
        int count = 0
        assert adminData == null
        withAdminData().orCreate.with {
            count++
        }
        assert count == 1
        assert adminData != null
    }
}
```

Listing 6.105: Creating non-existing member by navigating into its content with OrCreate()

If the compile time child type is not instatiatable, you have to provide the concrete type by `get<Element>OrCreate(Class childType)`.

```
transaction{
    // The asrPath points to an MIVariableDataPrototype
    mdfModel(asrPath) {

        withIntroduction().getOrCreate(MIBlockLevelContent).with { docuBlock ->
            assert docuBlock instanceof MIBlockLevelContent
        }
    }
}
```

Listing 6.106: Creating child member by navigating into its content with OrCreate() with type

6.6.4.6 Creating and adding Child List Members (0:*)

For child list members, the automation API provides many `createAndAdd()` methods for convenient child object creation. These method will always create the element, regardless if the same element (e.g. same ShortName) already exists.

If you want to update element see the chapter 6.6.4.7 on page 104.

```
transaction{
    // The asrPath points to an MIVariableDataPrototype
    mdfModel(asrPath) {
        assert adminData.sdg.empty

        adminData.with {
            withSdg().create{
                it.gid = "NewGidValue"
            }
        }

        assert adminData.sdg.first.gid == "NewGidValue"
    }
}
```

Listing 6.107: Creating new members of child lists with createAndAdd() by type

These methods are available — but be aware that not all of these methods are available for all child lists. Adding parameters, for example, is only permitted in the parameter child list of an `MIContainer` instance.

All Lists:

- The method `createAndAdd()` creates a new MDF object of the lists content type and appends it to this list. If the type is not instantiatable the method will thrown a `ModelException`. The new object is finally returned.
- The method `createAndAdd(Closure)` creates a new MDF object of the lists content type and appends it to this list. If the type is not instantiatable the method will thrown a `ModelException`. Then the closure is executed with the new object as closure parameter. The new object is finally returned.
- The method `createAndAdd(Class)` creates a new MDF object of the specified type and appends it to this list. The new object is finally returned.
- The method `createAndAdd(Class, Closure)` creates a new MDF object of the specified type and appends it to this list. Then the closure is executed with the new object as closure parameter. The new object is finally returned.
- The method `createAndAdd(Class, Integer)` creates a new MDF object of the specified type and inserts it to this list at the specified index position. The new object is finally returned.
- The method `createAndAdd(Class, Integer, Closure)` creates a new MDF object of the specified type and inserts it to this list at the specified index position. Then the closure is executed with the new object as closure parameter. The new object is finally returned.

Lists containing Referrables

- The method `createAndAdd(String)` creates a new child with the specified shortname and appends it to this list. The new object is finally returned. The used type is the lists content type. If the type is not instantiatable the method will thrown a `ModelException`.
- The method `createAndAdd(String, Closure)` creates a new `MIReferrable` with the specified shortname and appends it to this list. Then the closure is executed with the new object as closure parameter. The new object is finally returned. The used type is the lists content type. If the type is not instantiatable the method will thrown a `ModelException`.
- The method `createAndAdd(Class, String)` creates a new `MIReferrable` with the specified type and shortname and appends it to this list. The new object is finally returned.
- The method `createAndAdd(Class, String, Closure)` creates a new `MIReferrable` with the specified type and shortname and appends it to this list. Then the closure is executed with the new object as closure parameter. The new object is finally returned.
- The method `createAndAdd(Class, String, Integer)` creates a new `MIReferrable` with the specified type and shortname and inserts it to this list at the specified index position. The new object is finally returned.
- The method `createAndAdd(Class, String, Integer, Closure)` creates a new `MIReferrable` with the specified type and shortname and inserts it to this list at the specified index position. Then the closure is executed with the new object as closure parameter. The new object is finally returned.

Lists containing Parameters and Containers

- The method `createAndAdd(TypedDefRef)` creates a new Ecuc object (container or parameter) with the specified definition and appends it to this list. The new object is finally returned.
- The method `createAndAdd(TypedDefRef, Closure)` creates a new Ecuc object (container or parameter) with the specified definition and appends it to this list. Then the closure is executed with the new object as closure parameter. The new object is finally returned.
- The method `createAndAdd(TypedDefRef, Integer)` creates a new Ecuc object (container or parameter) with the specified definition and inserts it to this list at the specified index position. The new object is finally returned.
- The method `createAndAdd(TypedDefRef, Integer, Closure)` creates a new Ecuc object (container or parameter) with the specified definition and inserts it to this list at the specified index position. Then the closure is executed with the new object as closure parameter. The new object is finally returned.
- The method `byDefOrCreate(TypedDefRef)` retrieves the child with the passed definition, if the child exists and has a definition multiplicity of 0:1 or 1:1. Otherwise a new child is created. The definition and shortname (using the definition name) are automatically set before returning the new child. So this method will always create a new child if the upper multiplicity is greater than 1.

Lists containing Containers

- The method `createAndAdd(TypedDefRef, String)` creates a new container with the specified definition and shortname and appends it to this list. The new container is finally returned.

- The method `createAndAdd(TypedDefRef, String, Closure)` creates a new container with the specified definition and shortname and appends it to this list. Then the closure is executed with the new container as closure parameter. The new container is finally returned.
- The method `createAndAdd(TypedDefRef, String, Integer)` creates a new container with the specified definition and shortname and inserts it to this list at the specified index position. The new container is finally returned.
- The method `createAndAdd(TypedDefRef, String, Integer, Closure)` creates a new container with the specified definition and shortname and inserts it to this list at the specified index position. Then the closure is executed with the new container as closure parameter. The new container is finally returned.

6.6.4.7 Updating existing Elements

For child list members, the automation API provides many `byNameOrCreate()` methods for convenient child object update and creation on demand. These method will create the element if id does not exists, or return the existing element.

```
transaction{
    // The path points to an MISenderReceiverInterface
    mdfModel(asrPath) { MISenderReceiverInterface sendRecIf ->
        def dataElementRelation = sendRecIf.withDataElement()

        def dataElement = dataElementRelation.byNameOrCreate("MyDataElement")
        dataElement.name = "NewName"

        def dataElement2 = dataElementRelation.byNameOrCreate("NewName")

        assert dataElement == dataElement2
    }
}
```

Listing 6.108: Updating existing members of child lists with byNameOrCreate() by type

These methods are available — but be aware that not all of these methods are available for all child lists. Updating container, for example, is only permitted in the parameter child list of an `MIContainer` instance.

Lists containing Referrables

- The method `byNameOrCreate(String)` retrieves the child with the passed shortname, or creates the child, if it does not exist. The shortname is automatically set before returning the new child.
- The method `byNameOrCreate(String,Closure)` retrieves the child with the passed shortname, or creates the child, if it does not exist. The shortname is automatically set before returning the new child. Then the closure is executed with the child as closure parameter. The child is finally returned.
- The method `byNameOrCreate(Class, String)` retrieves the child with the passed type and shortname, or creates the child, if it does not exist. The shortname is automatically set before returning the new child.
- The method `byNameOrCreate(Class, String,Closure)` retrieves the child with the passed type and shortname, or creates the child, if it does not exist. The shortname is automatically

set before returning the new child. Then the closure is executed with the child as closure parameter. The child is finally returned.

Lists containing Containers

- The method `byNameOrCreate(TypedDefRef, String)` retrieves the child with the passed definition and shortname, or creates the child, if it does not exist. The definition and shortname are automatically set before returning the new child.
- The method `byNameOrCreate(TypedDefRef, String, Closure)` retrieves the child with the passed definition and shortname, or creates the child, if it does not exist. The definition and shortname are automatically set before returning the new child. Then the closure is executed with the child as closure parameter. The child is finally returned.

6.6.4.8 Deleting Model Objects

The method `delete(MIObject)` deletes the specified object from the model. This method must be called inside a transaction because it changes the model content.

Special case: If this method is being called on an active module configuration, it actually calls `IOperations.deactivateModuleConfiguration(MIModuleConfiguration)` to deactivate the module correctly.

```
// MIParameterValue param = ...

transaction {
    assert !param.isDeleted()
    param.delete()
    assert param.isDeleted()
}
```

Listing 6.109: Delete a parameter instance

The method `moRemove()` does the same as `delete()`. For details about model object deletion and access to deleted objects, read section 7.1.7.4 on page 328 ff.

IsDeleted The `isDeleted(MIObject)` method returns `true` if the specified object has been deleted (removed) from the MDF model, or is invisible in the current active `IModelView`.

```
MIObject obj = ...
if (!obj.isDeleted()) {
    work with obj ...
}
```

Listing 6.110: Check is a model instance is deleted

Note: The return value is dependent on the current active thread and the current active `IModelView` in this thread!

The method `moIsRemoved()` does the same as `isDeleted()`.

6.6.4.9 Duplicating Model Objects

The `duplicate()` method copies (clones) a complete MDF model sub-tree and adds it as child below the same parent.

- The source object must have a parent. The clone will be added to the same MDF feature below the same parent then
- AUTOSAR UUIDs will not be cloned. The clone will contain new UUIDs to guarantee unambiguousness

This method can clone any model sub-tree, also see `IOperations.deepClone(MIObject, MIObject)` for details.

Note: This operation must be executed inside of a transaction.

```
// MIContainer container = ...
transaction {
   def newCont = container.duplicate()
   // The duplicated container newCont
}
```

Listing 6.111: Duplicates a container under the same parent

6.6.4.10 Special properties and extensions

asrPath The `getAsrPath(MIReferrable)` method returns the AUTOSAR path of the specified object.

```
MIContainer canGeneral = ...
AsrPath path = canGeneral.asrPath
```

Listing 6.112: Get the AsrPath of an MIReferrable instance

See chapter 7.4.2 on page 353 for more details about AsrPaths.

asrObjectLink The `getAsrObjectLink(MIARObject)` method returns the `AsrObjectLink` of the specified object.

```
MIParameterValue param = ...
AsrObjectLink link = param.asrObjectLink
```

Listing 6.113: Get the AsrObjectLink of an AUTOSAR model instance

See chapter 7.4.4 on page 354 for more details about AsrObjectLinks.

defRef The `getDefRef()` method returns the `DefRef` of the model object.

```
MIParameterValue param = ...
DefRef defRef = param.defRef
```

Listing 6.114: Get the DefRef of an Ecuc model instance

The `MIParameterValue.setDefRef(DefRef)` method sets the definition of this parameter to the defRef.

```

MIParameterValue param = ...
DefRef newDefinition = ...
param.defRef = newDefinition
```

Listing 6.115: Set the DefRef of an Ecuc model instance

If the specified `DefRef` has a wildcard, the parameter must have a parent to calculate the absolute definition path - otherwise a `ModelCeHasNoParentException` will be thrown.

If it has no wildcard and no parent, the absolute definition path of the defRef will be used.

If the parameter has a parent or and parents definition does not match the defRefs parent definition, this method fails with `InconsistentParentDefinitionException`.

The `MIContainer.setDefRef(DefRef)` method sets the definition of this container to the defRef.

See chapter 7.4.5 on page 355 for more details about `DefRefs`.

ceState The CeState is an object which aggregates states of a related MDF object. Client code can e.g. check with the CeState if an Ecuc object has a related pre-configuration value.

The `getCeState(MIObject)` method returns the CeState of the specified model object.

```
MIParameterValue param = ...
IParameterStatePublished state = param.ceState
```

Listing 6.116: Get the CeState of an Ecuc parameter instance

See chapter 7.4.6 on page 358 for more details about the CeState.

ceState - User-defined Flag The method `isUserDefined()` returns `true`, if the ecuc configuration element like parameters is flagged as user-defined.

```
MIParameterValue param = ...
def flag = param.ceState.userDefined
```

Listing 6.117: Retrieve the user-defined flag of an Ecuc parameter in Groovy

The method `setUserDefined(boolean)` sets or removes the user-defined flag of an ecuc parameter.

Note: This method must be executed inside a transaction because it modifies the model state.

```
MIParameterValue param = ...
transaction {
  param.ceState.userDefined = true
}
```

Listing 6.118: "Set an Ecuc Parameter instance to user defined"

EcuConfigurationAccess and EcucDefinitionAccess The Groovy automation interface also provides special access methods for Ecuc elements (module configurations, container and parameter) to the

- EcuConfigurationAccess (see 7.5.1 on page 359)
- EcucDefinitionAccess (see 7.5.2 on page 364)

The `getEcucDefinition()` method returns the `IEcucDefinition` of the model object.

```
MIParameterValue param = ...
IEcucDefinition definition = param.ecucDefinition
```

Listing 6.119: Get the IEcucDefinition of an Ecuc model instance

The `getEcuConfiguration()` method returns the `IEcucHasDefinition` of the model object.

```
MIParameterValue param = ...
IEcucHasDefinition cfg = param.ecuConfiguration
```

Listing 6.120: Get the IEcucHasDefinition of an Ecuc model instance

These methods are the same as for bswmd model objects.

6.6.4.11 Reverse Reference Resolution - ReferencesPointingToMe

You can resolve all references in the MDF model in the reverse direction, so you can start at a reference target and navigate to all references which point to the reference target.

referencesPointingToMe The `getReferencesPointingToMe()` method returns all reference parameters in the active ecuc pointing to specified target (`MIReferrable`) object. It returns an empty collection if the target object is invisible or removed.

The `getReferencesPointingToMe(DefRef)` method returns all reference parameters in the active ecuc with the specified definition ([DefRef]) pointing to the specified target ([MIReferrable]) object. It returns an empty collection if the target object is invisible, removed or the specified definition is `null`.

```
List<MIReferenceValue> refs = container.referencesPointingToMe
//Or
DefRef refDefRef = // DefRef to reference parameter
def refByDef = container.getReferencesPointingToMe(refDefRef)
```

Listing 6.121: referencesPointingToMe sample

systemDescriptionObjectsPointingToMe The method `getSystemDescriptionObjectsPointingToMe()` returns all objects located in the system description which are parent objects of references pointing to the specified target. It returns an empty collection if the object is invisible or removed.

```
List<MIObject> references =
        systemDescElement.systemDescriptionObjectsPointingToMe
```

Listing 6.122: systemDescriptionObjectsPointingToMe sample

6.6.4.12 Derived Containers

The `MIHasContainer.getDerived()` method provides access to derived container information. The method returns a `IDerivedElementInfo` object corresponding to the model element.

The `IDerivedElementInfo` can be used to retrieve information about element or delete it:

- `getRemovedDerivedSubContainers()`: Retrieves the removed children, which could be used to restore them the children

- `isDerived()`: returns `true` if the element is derived
- `delete()`: deletes the element regardless if it is derived or not

```
container.derived.isDerived()
// Or
container.derived {
     boolean isDerivedFlag = isDerived()
     def removedList = getRemovedDerivedSubContainers()
}
```

Listing 6.123: Derived Container API access

Deletion of Derived Containers The method `delete()` deletes the `MIContainer` regardless, if it is derived or not.

This method behaves as follows:

- If the container is a derived container it calls the derived container deletion operation to delete it.
- All other containers with be deleted by means of `MIObject.deleteFromModel()`.

```
transaction {
    container.derived.delete()
}
```

Listing 6.124: Delete a derived container unconditionally

6.6.4.13 AUTOSAR Root Object

The `getAUTOSAR()` method returns the AUTOSAR root object (the root object of the MDF model tree of AUTOSAR data).

```
MIAUTOSAR root = AUTOSAR
```

Listing 6.125: Get the AUTOSAR root object

6.6.4.14 ActiveEcuC

The activeEcuc access methods provide access to the module configurations of the Ecuc model.

```
// Get the modules as Collection<MIModuleConfiguration>
Collection modules = activeEcuc.allModules
```

Listing 6.126: Get the active Ecuc and all module configurations

```
// Iterate over all module configurations
activeEcuc {
    int count = 0
    allModules.each { moduleCfg ->
        count++
    }
    assert count > 1
}
```

Listing 6.127: Iterate over all module configurations

```
activeEcuc {
    // Parameter type is IActiveEcuc
    ecuc ->

    def defRef = DefRef.create(EDefRefWildcard.AUTOSAR, "EcuC")

    // Get the modules as Collection<MIModuleConfiguration>
    Collection foundModules = ecuc.modules(defRef)
    assert !foundModules.empty
}
```

Listing 6.128: Get module configurations by definition

6.6.4.15 DefRef based Access to Containers and Parameters

The Groovy automation interface for the MDF model provides some overloaded access methods for

- `MIModuleConfiguration.getSubContainer()`
- `MIContainer.getSubContainer()`
- `MIContainer.getParameter()`

to offer convenient filtering access to the subContainer and parameter child lists.

```
activeEcuc {
    // Parameter type is IActiveEcuc
    ecuc ->

    def module = ecuc.modules(EcuC.DefRef).first

    // Get containers as List<MIContainer>
    def containers = module.subContainer(EcucGeneral.DefRef)

    // Get parameters as List<MIParameterValue>
    def cpuType = containers.first.parameter(CPUType.DefRef)

    assert cpuType.size() == 1
}
```

Listing 6.129: Get subContainers and parameters by definition

6.6.4.16 Ecuc Parameter and Reference Value Access

The Groovy automation interface also provides special access methods for Ecuc parameter values. These methods are implemented as extensions of the Ecuc parameter and value types and can

therefore be called directly at the parameter or reference instance.

Value Checks

- `MIParameterValue.hasValue()` returns `true` if the parameter (or reference) has a value.
- `MINumericalValue.containsBoolean()` returns `true` if the parameter value contains a valid boolean with the same semantic as `IEcucModelAccess.containsBoolean(MINumericalValue)`.

  Call this method in advance to guarantee that `MINumericalValueVariationPoint.getAsBoolean()` doesn't lead to errors.
- `MINumericalValue.containsInteger()` returns `true` if the parameter value contains a valid integer with the same semantic as `IEcucModelAccess.containsInteger(MINumericalValue)`.

  Call this method in advance to guarantee that `MINumericalValueVariationPoint.getAsInteger()` doesn't lead to errors.
- `MINumericalValue.containsDouble()` returns `true` if the parameter value contains a valid double (AUTOSAR float) with the same semantic as `IEcucModelAccess.containsFloat(MINumericalVal`

  Call this method in advance to guarantee that `MINumericalValueVariationPoint.getAsDouble()` doesn't lead to errors.

```
// MINumericalValue param = ...

if (!param.hasValue()) {
    scriptLogger.warn "The parameter has no value!"
}

if (param.containsInteger()) {
    int value = param.value.asInteger
}
```

Listing 6.130: Check parameter values

Parameters

- `MINumericalValueVariationPoint.getAsLong()` returns the value as native `long`.

  Throws `NumberFormatException` if the value string doesn't represent an integer value.
  Throws `ArithmeticException` if the value will not exactly fit in a `long`.
- `MINumericalValueVariationPoint.getAsInteger()` returns the value as native `int`.

  Throws `NumberFormatException` if the value string doesn't represent an integer value.
  Throws `ArithmeticException` if the value will not exactly fit in an `int`.
- `MINumericalValueVariationPoint.getAsBigInteger()` returns the value as `BigInteger`.

  Throws `NumberFormatException` if the value string doesn't represent an integer value.
- `MINumericalValueVariationPoint.getAsDouble()` returns the value as `Double`

  Throws `NumberFormatException` if the value string doesn't represent a float value.
- `MINumericalValueVariationPoint.getAsBoolean()` returns the value as `Boolean`.

  Throws `NumberFormatException` if the value doesn't represent a boolean value.

- `MITextualValue.asCustomEnum(Class)` returns the value of the enum parameter as a custom enum literal. If the Class destClass implements the `IEcucEnum` interface, the literals are mapped via these information form the [IEcucEnum] interface. Read the JavaDoc of [IEcucEnum] for more details.

  @param destClass the destination enum class @return the literal mapped via the destClass or `null` if not found @see IEcucEnum

```
// MINumericalValue param = ...
// MINumericalValueVariationPoint is the type of param.value

long longValue = param.value.asLong
assert longValue == 10

int intValue = param.value.asInteger
assert intValue == 10

BigInteger bigIntValue = param.value.asBigInteger
assert bigIntValue == BigInteger.valueOf(10)

Double doubleValue = param.value.asDouble
assert Math.abs(doubleValue-10.0) <= 0.0001
```

Listing 6.131: Get integer parameter value

References

- `MIARRef#getAsAsrPath()` returns the reference value as AUTOSAR path.

- `MIReferenceValue#getAsAsrPath()` returns the reference value as AUTOSAR path.

- `MIReferenceValue.getRefTarget()` returns the reference parameters target object (the object referenced by this parameter). It returns `null` if the target cannot be resolved or the reference parameter doesn't contain a value reference.

```
// MIReferenceValue refParam = ...

def asrPath1 = refParam.asAsrPath
def asrPath2 = refParam.value.asAsrPath
assert asrPath1 == asrPath2

String pathString = refParam.value.value
assert asrPath1.autosarPathString == pathString

def target1 = refParam.refTarget
def target2 = refParam.value.refTarget
assert target1 == target2
```

Listing 6.132: Get reference parameter value

6.6.4.17 Getting and Setting Formula Expression Values

The Groovy automation interface provides special methods to evaluate a formula expression and replace its content as well. These methods are implemented as extensions of the `MIFormulaEx-`

pression and therefore they can be called directly at its instances.

Get Expression Value

- The formula expression can contain a simple numeric literal, a boolean literal, or a more advanced expression that handles references to autosar model elements, arithmetic functions, special functions, and special values.

  Therefore, the method `MIFormulaExpression.eval()` is used to do the evaluation and return the result as `IFormulaResult`.

- The `IFormulaResult` represents the evaluation result of a `MIFormulaExpression`. It contains the numeric value if the formula has been evaluated successfully or the `ModelFormulaException` if an error happened while evaluating the expression.

  This interface has the following convenient methods:

  - `isEvaluated()`: Returns `True` if the expression has been evaluated successfully, or `False` if an error has occurred.
  - `getValue()`: It is worthwhile to mention that formula expression always yields a numeric value. Thus, the numeric result of the evaluation will be mainly given using this method.
  - `getAsBoolean()`: If the provided formula expression is a condition, the result is expected to be boolean. Therefore, this method can be used to convert the numeric result to boolean. Zero is considered `False`, and any other value is considered `True`.
  - `getAsFloat()`: Returns the numeric value of the formula as double.
  - `getAsBigDecimal()`: Returns the numeric value of the formula as BigDecimal.
  - `getAsInteger()`: Returns the numeric value of the formula as integer:
    - First, if the origin value is of an integer type, it is returned as it is.
    - Second, if the origin value can be converted to integer without loss of any information, then it is converted and returned.
    - Otherwise, it throws `NumberFormatException` stating that 'The formula value is not an integer'.
  - `getError()`: Returns an `Optional` of `ModelFormulaException`. It will be empty if the formula has been evaluated successfully, or contains the error that occurred.

```
// arrayElement is MIImplementationDataTypeElement
// arraySizeOrCreate returns MIFormulaExpression
IFormulaResult evalResult = arrayElement.arraySizeOrCreate.eval()

if (evalResult.evaluated) {
    // Get the numeric value
    Number value = evalResult.value
    // Get the numeric value converted to other types
    boolean valueAsBoolean = evalResult.asBoolean
    BigInteger valueAsInteger = evalResult.asInteger
    double valueAsFloat = evalResult.asFloat
} else {
    ModelFormulaException exception = evalResult.error.get()
}

// This one line statement can also be used to provide quick access, but
// it will throw an exception if the expression was not evaluated successfully.
Number value = arrayElement.arraySizeOrCreate.eval().value
```

Listing 6.133: Evaluate formula expression

Set Expression Value

- The method `setValue(Number)` replaces the content of the passed `MIFormulaExpression` by the provided numeric value.

  Throws `NullPointerException` if any of the parameters is null.

- The method `setValue(Boolean)` replaces the content of the passed `MIFormulaExpression` by the string representation of the provided Boolean value.

  Throws `NullPointerException` if any of the parameters is null.

Important Note: These operations must run within a unit of work. The client code is responsible for opening the transaction before calling these two methods.

```
transaction {
    // arrayElement is MIImplementationDataTypeElement
    // arraySizeOrCreate returns MIFormulaExpression
    arrayElement.arraySizeOrCreate.setValue(5)
}
```

Listing 6.134: Set formula expression value

6.6.5 SystemDescription Access

The systemDescription API provides methods to retrieve system description data like the path to the flat extract or the flat map instance.

It is grouped by the AUTOSAR version. So the `getAutosar4()` methods provides access to AUTOSAR 4 model elements.

The `getPaths()` provides common paths to elements like:

- FlatMap path
- FlatExtract path
- FlatCompositionType path

```
AsrPath flatExtractPath = systemDescription.paths.flatExtractPath
AsrPath flatMapPath = systemDescription.paths.flatMapPath
```

Listing 6.135: Get the FlatExtract and FlatMap paths by the SystemDescription API

```
systemDescription{
    autosar4{
       flatExtract.ifPresent{ theFlatExtract ->
            // do something with the flatMap
       }
    }
}
// Or in property style
def theFlatExtractOpt = systemDescription.autosar4.flatExtract
if(theFlatExtractOpt){
    def theFlatExtract = theFlatExtractOpt.get()
}
```

Listing 6.136: Get FlatExtract instance by the SystemDescription API

6.6.6 Transactions

Model changes must always be executed within a transaction. The automation API provides some simple means to execute transactions.

For details about transactions read 7.1.7 on page 327.

```
scriptTask("TaskName", DV_PROJECT){
    code {
        transaction{
            // Your transaction code here
        }
    }
}
```

Listing 6.137: Execute a transaction

```
scriptTask("TaskName", DV_PROJECT){
    code {
        transaction("Transaction name") {
            // The transactionName property is available inside a transaction
            String name = transactionName
        }
    }
}
```

Listing 6.138: Execute a transaction with a name

```
import com.vector.cfg.model.uow.TransactionException
scriptTask("TaskName", DV_PROJECT){
    code {
      try {
        transaction("Transaction") {
            // Any exception occurs
            throw new RuntimeException()
        }
      } catch (TransactionException ex) {
        assert ex.getMessage() == "Failed executing transaction 'Transaction'"
      }
    }
}
```

Listing 6.139: Handle a TransactionException

The transaction name has no additional semantic. It is only be used for logging and to improve error messages.

Nested Transactions If you open a transaction inside a transaction the inner transaction is ignored and it is as no transaction call was done. So be aware that nested transactions are no real transaction, which leads to the fact the these nested transactions can not be undone.

If you want to know whether a transaction is already running, see the transactions API below.

6.6.6.1 Transactions API

The Transactions API with the keyword `transactions` provides access to running transactions or the transaction history.

You can use method `isTransactionRunning()` to check if a transaction is currently running. The method returns `true`, if a transaction is running in the current `Thread`.

```
scriptTask("TaskName", DV_PROJECT){
    code {
        // Switch to the transactions API
        transactions{

            //Check if a transaction is running
            assert isTransactionRunning() == false

            // Open a transaction
            transaction{
                // Now a transaction is running
                assert isTransactionRunning() == true
            }
        }
        // Or the short form
        transactions.isTransactionRunning()
    }
}
```

Listing 6.140: Check if a transaction is running

TransactionHistory The transaction history API provides some methods to handle transaction undo and redo. This way, complex model changes can be reverted quite easily.

- The `undo()` method executes an undo of the last transaction. If the last transaction frame cannot be undone or if the undo stack is empty this method returns without any changes.
- The `undoAll()` method executes undo until the transaction stack is empty or an undoable transaction frame appears on the stack.
- The `redo()` method executes an redo of the last undone transaction. If the last undone transaction frame cannot be redone or if the redo stack is empty this method returns without any changes.
- The `canUndo()` method returns `true` if the undo stack is not empty and the next undo frame can be undone. This method changes nothing but you can call it to find out if the next `undo()` call would actually undo something.
- The `canRedo()` method returns `true` if the redo stack is not empty and the next redo frame can be redone. This method changes nothing but you can call it to find out if the next `redo()` call would actually redo something.
- The `clearUndoRedoHistory()` method clears the undo/redo history. After this method was called, all previous undo and redo information are lost and can not be restored.
- The `setUndoHistoryLimit(int)` method allows to limit the number of the undo history stack size to the given value.

```
scriptTask("TaskName", DV_PROJECT){
    code {
        transaction("TransactionName") {
            // Your transaction code here
        }

        transactions{
            assert transactionHistory.canUndo()

            transactionHistory.undo()

            assert !transactionHistory.canUndo()
        }
    }
}
```

Listing 6.141: Undo a transaction with the transactionHistory

```
scriptTask("TaskName", DV_PROJECT){
    code {
        transaction("TransactionName") {
            // Your transaction code here
        }

        transactions{
            transactionHistory.undo()

            assert transactionHistory.canRedo()

            transactionHistory.redo()

            assert !transactionHistory.canRedo()
        }
    }
}
```

Listing 6.142: Redo a transaction with the transactionHistory

```
scriptTask("TaskName", DV_PROJECT){
    code {
        transaction("Transaction1") {
            // Your transaction code here
        }

        transaction("Transaction2") {
            // Your transaction code here
        }

        transactions{

            assert transactionHistory.canUndo()

            transactionHistory.undo()

            assert transactionHistory.canRedo()

            assert transactionHistory.canUndo()

            transactionHistory.clearUndoRedoHistory()

            assert !transactionHistory.canRedo()

            assert !transactionHistory.canUndo()
        }
    }
}
```

Listing 6.143: Clear the undo/redo history with the transactionHistory

```
scriptTask("TaskName", DV_PROJECT){
    code {
        transactions{
           transactionHistory.setUndoHistoryLimit(1)
        }

        transaction("Transaction1") {
            // Your transaction code here
        }

        transaction("Transaction2") {
            // Your transaction code here
        }

        transactions{

            assert transactionHistory.canUndo()

            transactionHistory.undo()

            assert !transactionHistory.canUndo()
        }
    }
}
```

Listing 6.144: Set the undo history limit with the transactionHistory

6.6.6.2 Operations

The model operations implement convenient means to execute complex model changes like AUTOSAR module activation or cloning complete model sub-trees. The operations API is available inside of a transaction with the keyword `operation`. The class `IOperations` defines the available methods.

- The method `activateModuleConfiguration(DefRef)` activates the specified module configuration. This covers:
  - Creation of the module including the reference in the ActiveEcuC (the `ECUC-VALUE-COLLECTION`)
  - Creation of mandatory containers and parameters (lower multiplicity > 0)
  - Applying the recommended configuration
  - Applying the pre-configuration values

  Note: If the `DefRef` has a wildcard, `activateModuleConfiguration(DefRef)` tries to activate the most specific module definition matching the wildcard, if unique. If it is not unique the method will throw an exception. For example the `DefRef /[ANY]/Dio` will activate the `/MICROSAR/Dio` instead of `/AUTOSAR/EcucDefs/Dio`.

```
transaction{
    // Activates the Dio module
    operations.activateModuleConfiguration(sipDefRef.Dio)
}
```

Listing 6.145: Activation of the ModuleConfiguration Dio

- The method `deactivateModuleConfiguration(MIModuleConfiguration)` deletes the specified module configuration from the model. In case of a split configuration, the related persistency location is being removed from the project settings. In XML file base configurations, the related file is being deleted during the next project save if it doesn't contain configuration objects anymore.

  If the module configuration is referenced from the active-ECUC this link is being removed too.

- The method `changeBswImplementation(MIModuleConfiguration, MIBswImplementation)` changes the BSW-implementation of a module configuration including the definition of all contained containers and parameters.

- `setConfigurationVariantOfAllModuleConfigurations(EEcucConfigurationVariant)` sets the implementation configuration variant of all active `MIModuleConfiguration`. If a module configuration does not support the requested variant it is ignored.

  Supported enum values are:

  - `EEcucConfigurationVariant`
    - `VARIANT_PRE_COMPILE`
    - `VARIANT_LINK_TIME`
    - `VARIANT_POST_BUILD_LOADABLE`

  This is for *post-build loadable* only! See the method `setConfigurationVariant()` in class `IEcucModuleConfiguration` for details.

- The `deepClone(MIObject, MIObject)` operation copies (clones) a complete MDF model sub-tree and adds it as child below the specified parent.
  - The source object must have a parent. The clone will be added to the same MDF feature below the destination parent then
  - AUTOSAR UUIDs will not be cloned. The clone will contain new UUIDs to guarantee unambiguousness
- The method `createModelObject(Class)` creates a new element of the passed modelClass (meta class). The modelObject must be added to the whole AUTOSAR model, before finishing the transaction.
- The method `createUniqueMappedAutosarPackage(AsrPath, Path, IVersion)` can be used to create new `MIARPackages` in new arxml files. It creates an new instance of the specified AUTOSAR package and adds it to the model tree. All non-existing parent packages will be created too.

  The new package (including new created parent packages) will be mapped uniquely to the specified location (Path and AUTOSAR version).

6.6.7 Model Synchronization

The Model synchronization provides operation to solve and synchronize common model related items. The model synchronization API is available inside of an active project with the keyword `modelSynchronization`. The class `IModelSynchronizationApi` defines the available methods.

The method `synchronize()` synchronizes the model for all registered model synchronization elements like validations and other operations. The method will open a transaction, if `isSynchronizationRequired()` returns `true`, otherwise this method does nothing.

```
// Execute the model synchronization
modelSynchronization.synchronize()

//Or more elaborated, but means the same
modelSynchronization{
  if(synchronizationRequired){
    synchronize()
  }
}
```

Listing 6.146: Model synchronization inside an open project

6.6.8 PreBuild and PostBuild Variance (Post-build selectable)

The variance access API is the entry point for convenient access to variant AUTOSAR model content. It provides means to filter variant model content and access variant specific data.

The DaVinci Configurator supports two types of variance:

- PostBuild variance (Post-build selectable)
- PreBuild variance

For details about PostBuild variance and model views read 7.2 on page 329.

6.6.8.1 Investigate Project Variance

The projects variance can be analyzed using the `variance` keyword. These methods can be called then:

- The method `getCurrentlyActiveView()` returns the currently active model view.
- The method `variantView(String)` returns the `IPostBuildPredefinedVariantView` with the given name. This may be a PreBuild of PostBuild view.

```
scriptTask("TaskName", DV_PROJECT){
    code{
        // Activates the DoorLeftFront variant
        variance.variantView("DoorLeftFront").activeWith{
            // Now all MDF model accesses are executed in the variant "
                DoorLeftFront"
        }
    }
}
```

Listing 6.147: Retrieve and use a variant view by name

```
scriptTask("TaskName", DV_PROJECT){
    code{
        def activeView1 = variance.currentlyActiveView
        assert activeView1 instanceof IPostBuildInvariantValuesView

        // ... or with a closure
        variance {
                def activeView2 = currentlyActiveView
                assert activeView1 == activeView2
                assert activeView1 == postBuildInvariantValuesView

                // Get number of variants
                int num = allPostBuildVariantViews.size()
                assert num == 4
        }
    }
}
```

Listing 6.148: The default view is the IPostBuildInvariantValuesView

Investigate Project Variance - PostBuild

- The method `hasPostBuildVariance()` returns `true` if the active project contains post-build variants.
- The method `getPostBuildInvariantValuesView()` returns the PostBuild invariant values view. This view contains objects which are not variant (Object or parent have no Variation-Point) or the values in all variants are equal.
- The method `getPostBuildInvariantEcucDefView()` returns the PostBuild invariant Ecuc definition view. This view contains the same objects as the invariant values view but excludes all objects which, by (EcuC / BSWMD) definition, support variance. Using this view you can avoid dealing with objects which are accidentally equal by value (in your test configurations) but potentially can be different because they support variance.
- The method `getAllPostBuildVariantViews()` returns the model views of all PostBuild predefined variants defined in the evaluated variant set. It never returns `null`. If the project

contains no PostBuild variants, the result will be an empty list.

The order of variant views returned is deterministic. It is the natural order of the names of the variants defined in the evaluated variant set.

- The method `getAllPostBuildVariantViewsOrInvariant()` returns the same as the method `getAllPostBuildVariantViews()` if the project contains PostBuild variants. If the project contains no PostBuld variants (see `hasPostBuildVariance()`) the method returns a list containing only the `IPostBuildInvariantValuesView`.

  This helps to create code working with both variant and non-variant projects.

6.6.8.2 Variant Model Objects

The following model object extensions provide convenient means to investigate model object variance in detail.

- The method `IModelView.activeWith(Supplier)` executes code under visibility of the specified model view.
- The method `MIObject.isModelInvariant()` returns `true` if the object and all its parents has no variation point conditions. If this is `true`, this model object instance is visible in all variant view.
- The method `MIObject.isVisible()` returns `true` if the object is visible in the current model view.
- The method `MIObject.isVisibleInModelView(IModelView)` returns `true` if the object is visible in the specified model view.
- The method `MIObject.asViewedModelObject()` returns a new `IViewedModelObject` instance using the currently active view.
- The method `MIObject.getPostBuildVariantSiblings()` returns MDF object instances representing the same object but in all variants.

  For details about the sibling semantic see 7.2.1.3 on page 331.
- The method `getPostBuildVariantSiblingsWithoutMyself(MIObject)` returns the same collection as `getPostBuildVariantSiblings(MIObject)` but without the specified object.

```
// IPostBuildPredefinedVariantView viewDoorLeftFront = ...
// MIParameterValue variantParameter = ...

viewDoorLeftFront.activeWith {
    assert variance.currentlyActiveView == viewDoorLeftFront

    // The parameter instance is not visible in all variants ...
    assert !variantParameter.isModelInvariant()

    // ... but all variants have a sibling with the same value
    assert variantParameter.isPostBuildValueInvariant()
}
```

Listing 6.149: Execute code in a model view

Variant Model Objects - PostBuild

- The method `MIObject.isPostBuildValueInvariant()` returns `true` if the object has the same value in all PostBuild variants.

  See `IPostBuildInvariantValuesView` for more details to the concept.

  Attention: This must also return true for elements in other variants as the first PostBuild Predefined Variant, when the element is invariant! This is not the same result as `IPostBuildInvariantValuesView isVisible()` method returns.

  For details about invariant views see 7.2.1.4 on page 332.

- The method `MIObject.isPostBuildEcucDefInvariant()` returns `true` if the object is invariant according to its EcuC definition.

  See `IPostBuildInvariantEcucDefView` for more details to the concept.

  Attention: This must also return true for elements in other variants as the first PostBuild Predefined Variant, when the element is invariant! This is not the same result as `IPostBuildInvariantValuesView isVisible()` method returns.

- The method `MIObject.isNeverPostBuildVisible()` returns `true`, if the object is *invisible* in all variant view.

- The method `MIObject.getVisiblePostBuildVariantViews(MIObject)` returns all variant views the specified object is visible in.

- The method `MIObject.getVisiblePostBuildVariantViews(MIObject)` returns all variant views the specified object is visible in.

6.6.9 Additional Model API

6.6.9.1 User Annotations

In DaVinci Configurator the user can add AUTOSAR annotations to configuration elements. You can create, modify, read and delete these annotations like in the UI editors.

All sub types of `MIHasAnnotation` elements support annotations like:

- `MIModuleConfigurations`
- `MIContainers`
- `MIParameterValues`
- `MIIdentifiables`

Although annotations are stored in the data model, their `changeable` state is independent of the configuration element `changeable` state. Annotations can be added/changed/deleted on every existing configuration element with valid definition, except the project was opened in read-only mode.

The `IUserAnnotation` interface provide methods like:

- `getLabel()` - Returns the label of the annotation, like `getName()` of a container
- `setLabel(String)` ()} - Changes the label
- `getText()` - Returns the text of the annotation.
- `setText(String)` ()} - Changes the text
- `isChangeable()` - Returns `true`, if the annotation is changeable
- `delete()` - Deletes the annotation

Access User Annotations The `getUserAnnotations(MIHasAnnotation)` method returns the `IUserAnnotations` for the model element. The returned list provides additional methods defined in `IUserAnnotationList`.

```
// We already have the container "cont" or any other model element
def myContainer = cont

def annos = myContainer.userAnnotations // Retrieve the list of annotations
def anno = annos.byLabel("MyLabel")     // Select the annotation with "MyLabel"
def text = anno.text                    // Get the Text

// Or short
text = myContainer.userAnnotations["MyLabel"].text
```

Listing 6.150: Get a UserAnnotation of a container

Creation and Modification of User Annotations You can create new User Annotations with the methods:

- `createAndAdd(label)`
- `byLabelOrCreate(label)`

```
transaction {
    // We already have the container "cont"
    def anno = cont.userAnnotations.createAndAdd("MyAnno")
    anno.text = "My Text"
}
```

Listing 6.151: Create a new UserAnnotation

```
transaction {
    // We already have the container "cont"
    def anno = cont.userAnnotations.byLabelOrCreate("MyAnno")
    anno.text = "My Text"
}
```

Listing 6.152: Create or get the existing UserAnnotation by label name

Notes The `IUserAnnotationList` is updated, when the underlying model changes.

The `IUserAnnotationList` is read only list and does not permit any modify operations defined in `java.util.List`, but certain operations like `createAndAdd(String)` will affect the list content. If you delete a contained `IUserAnnotation` the list will not be updated.

6.7 Generation

The Automation Interface provides generation API for different generation use cases:

- Normal code generation, see 6.7.1
  - Including external generation steps
- SWC Templates and Contract Phase Headers generation, see 6.7.3 on page 134

6.7.1 Code Generation

The block **generation** encapsulates all settings and commands which are related to code generation of BSW modules:

The basic structure is the following:

```
generation {
    settings {
        // Settings like the selection of generators for execution are done here
        externalGenerationSteps {
            // Settings related to externalGenerationSteps can be done here
        }
    }
    // The execution of the generation or validation can be started here
}
```

Listing 6.153: Basic structure

6.7.1.1 Generation Settings

The class `IGenerationSettingsApi` encapsulates all settings which belong to a generation process. E.g.

- Select the generators to execute
- Select the target type (Real, VTT)
- Select the external generation steps
- If the module supports multiple module configurations, select the configurations which shall be generated

The following chapters show samples for the standard use cases.

Generation with default Project Settings The following snippet executes a validation with the default project settings.

```
scriptTask("validate_with_default_settings"){
    code{
        generation{
            validate()
        }
    }
}
```

Listing 6.154: Validate with default project settings

To execute a generation with the standard project settings the following snippet can be used. The validation is executed implicitly before the generation because of AUTOSAR requirements.

```
scriptTask("generate_with_default_settings"){
    code{
        generation{
            generate()
        }
    }
}
```

Listing 6.155: Generate with standard project settings

Generation with Report `IGenerationReportApi` is the entry point for generation report settings. When the settings are set and generation has been finished, the report output path can be seen in the logs. The report has been generated in the project logs folder.

The following snippet sets the report settings and executes a generation.

```
scriptTask("generate_components_with_report"){
    code{
        generation{
            settings {

                selectGeneratorsByDefRef("/MICROSAR/Aaa")
                selectGeneratorsByDefRef("/MICROSAR/Hhh")

                // Open the report closure to get access to the report settings
                report {
                      // If no settings set, the configurator settings get used
                      createHtmlReport true
                }
            }

            // After generation the output paths can be found in the console view
            generate()
        }
    }
}
```

Listing 6.156: Generation of components with a result report

createHtmlReport If not set, project settings will be used.
`setCreateHtmlReport(Boolean)` defines if HTML report should be generated.

Generation of one Module This sample selects one specific module and starts the generation. There are two ways to open a settings block:

- **settings**
  - This keyword creates empty settings. E.g. no module is selected for execution.

```
scriptTask("generate
    code{
        generation{
            settings
                sele
            }
            generate
        }
    }
}
```

- **settingsFromProject**
  - Instead of using an empty generator selection, this keyword takes the generator selection from the project settings as template. This selection can optionally be refined by explicit selections. The generator project settings contain the latest generator selection of the generation dialog, that have been saved. Please note that the Target Type (VTT/REAL) is not saved and needs to be specified explicitly. The selection of a Target Type does not directly select or deselect generators within one settingsFromProject closure. So the API getSelectedGenerators() returns all selected generators, but this list may be further filtered according to the Target Type, before the generation is executed.

```
scriptTask("generate_one_module"){
    code{
        generation{
            settingsFromProject{ // loads the generator selection from project
                settings
                // further generators can be selected or deselected in here
            }
            generate()
        }
    }
}
```

Listing 6.158: Generate modules from project settings

Instead of selecting the generator directly by its `DefRef`, there is also the possibility to fetch the generator object and select this object for execution.

```
scriptTask("generate_one_module"){
    code{
        generation{
            settings{
            // To take the project settings as template use
            // settingsFromProject{
                def gens = generatorByDefRef ("/MICROSAR/Aaa")
                selectGenerators(gens)
            }
            generate()
        }
    }
}
```

Listing 6.159: Generate one module

Generation of multiple Modules To select more than one generator the following snippet can be used.

```
scriptTask("generate_two_modules"){
    code{
        generation{
            settings{
                selectGeneratorsByDefRef ("/MICROSAR/Aaa", "/MICROSAR/Bbb")
            }
            generate()
        }
    }
}
```

Listing 6.160: Generate two modules

Generation of Multi Instance Modules Some module definitions have a upper multiplicity greater than one. (E.g. [0:5] or [0:*]) This means it is allowed to create more than one module configuration from this module definition. If the related generator is started with the default API, all available module configurations are selected for generation. The following API can be used to generate only a subset of all related module configurations.

```
scriptTask("generate_one_module_with_two_configs"){
    code{
        generation{
            settings{
                def gen = generatorByDefRef ("/MICROSAR/MultiInstModule")
                // clear default selection
                gen.deselectAllModuleInstances()
                // Select the module configurations to generate
                gen.selectModuleInstance(AsrPath.create("/ActiveEcuC/
                    MultiInstModule1"))

                // Instead of the full qualified path, the module configuration
                    short name can also be used
                gen.selectModuleInstance("MultiInstModule2")
            }
            generate()
        }
    }
}
```

Listing 6.161: Generate one module with two configurations

6.7.1.2 Generation of Generation Steps

Besides the internal generators, which are covered by the topics above, there are also generation steps which can be executed with the following API. A new block `externalGenerationSteps` within the `settings` block encapsulates all settings related to external generation scripts.

```
scriptTask("generate_ext_gen_step"){
    code{
        generation{
            settings{
                externalGenerationSteps{
                // To take the project settings as template use
                // externalGenerationStepsFromProject{}
                    selectStep("ExtGen1")
                    selectStep("ExtGen2")
                }
            }
            generate()
        }
    }
}
```

Listing 6.162: Execute an external generation step

Retrieval of TargetType (REAL, VTT) of Generation Steps You can query the `EEnvironment-TargetType` of the generation step. This will give you the information if the step can be executed in `REAL`, `VTT` or both modes.

```
generation.settings.externalGenerationSteps{
    def step = stepByName("ExtGen1")
    def targetType = step.generationStep.targetType

    if(targetType.isRealAvailable()){
        // Real use case
    }else if(targetType.isVttAvailable()){
        // VTT use case
    }else{
        // None selected
    }
}
```

Listing 6.163: Retrieval of the TargetType of a Generation Step

Set a user defined logger It is possible to pass a specific logger to the generation settings. So all generation events (Phase startet, Module started...) are additionally logged to this logger.

```
    generation{
        settings{
            setUserLogger(userLogger)
        }
        generate()
    }
```

Listing 6.164: Set a user defined logger

6.7.1.3 Evaluate generation or validation results

Each validation and generation process has an overall result which states if the execution has been successfully or not. Additionally to the overall state, the state of one specific generator can also be of interest. To provide a possibility to access this information all methods for `validate` and `generate` return an `IGenerationResultModel`.

```
scriptTask("generate_with_default_settings"){
    code{
        generation{
           def result = generate()
           scriptLogger.info "Overall result : " + result.result
           scriptLogger.info "Duration       : " + result.formattedDuration

           // Access results of each generator or generation step
           result.generationResultRoot.allGeneratorAndStepElements.each {
              scriptLogger.info "Generator name : " + it.name
              scriptLogger.info "Result         : " + it.currentState
           }

        }
    }
}
```

Listing 6.165: Evaluate the generation result

6.7.2 Generation Task Types

There are three types of `IScriptTaskType` for the generation process:

- Generation Step: `DV_GENERATION_STEP`
- Generation Process Start: `DV_ON_GENERATION_START`
- Generation Process End: `DV_ON_GENERATION_END`

The general description of the type is in chapter 6.3.1.4 on page 36. The following code samples show the usage of these task types:

Generation Step A sample for the `DV_GENERATION_STEP` type:

```
scriptTask("GenStepTask", DV_GENERATION_STEP){
    taskDescription "Task is executed as Generation Step"

    def myArg = newUserDefinedArgument(
                    "myArgument",
                    String,
                    "Defines a user argument for the GenerationStep")

    code{ phase, generationType, resultSink ->

        def myArgVal = myArg.value
        // The value myArgVal was passed from the generation step in the project
            settings editor

        scriptLogger.info "MyArg is: $myArgVal"
        scriptLogger.info "GenerationType is: $generationType"

        if(phase.calculation){
            // Execute code before / after calculation

            transaction {
               // Modify the Model in the calculation phase
            }
        }

        if(phase.validation){
            // Execute code before / after validation

        }

        if(phase.generation){
           // Execute code before / after generation

        }
    }
}
```

Listing 6.166: Use a script task as generation step during generation

The *Generation Step* can also report validation results into the passed `resultSink`. See chapter 6.8.5.11 on page 147 for a sample how to create an validation-result and report it.

The generationType defines if the current generation is for the `REAL` or `VTT` platform.

Generation Process Start A sample for the `DV_ON_GENERATION_START` type:

```
scriptTask("GenStartTask", DV_ON_GENERATION_START){
    taskDescription "The task is automatically executed at generation start"

    code{ phasesToExecute, generators ->

        scriptLogger.info "Phases are: $phasesToExecute"
        scriptLogger.info "Generators to execute are: $generators"

        // Execute code before the generation will start
    }
}
```

Listing 6.167: Hook into the GenerationProcess at the start with script task

Generation Process End A sample for the `DV_ON_GENERATION_END` type:

```
scriptTask("GenEndTask", DV_ON_GENERATION_END){
    taskDescription "The task is automatically executed at generation end"

    code{ generationResult, generators ->

        scriptLogger.info "Process result was: $generationResult"
        scriptLogger.info "Executed Generators: $generators"

        // Execute code after the generation process was finished
    }
}
```

Listing 6.168: Hook into the GenerationProcess at the end with script task

6.7.3 Software Component Templates and Contract Phase Headers Generation

The Software Component Templates and Contract Phase Headers (Swct) generation automation API provides access to configure and start the Swct generation.

The block **generation.swct** encapsulates all settings and commands which are related to this use case.

The basic structure is the following:

```
generation.swct {
    settings {
        // Settings like the selection of components to generate
    }
    // The execution of the generation can be started here
    generate()
}
```

Listing 6.169: Basic Swct structure

6.7.3.1 Swct Generation Settings

The class `IGenerationSwctSettingsApi` encapsulates all settings which belong to a Swct generation process.
Examples:

- Select the software components to execute
- Retrieve the available software components

The following chapters show samples for the standard use cases.

6.7.3.2 Generation with default Project Settings

To execute the Swct generation with the standard project settings the following snippet can be used:

```
scriptTask("generate_with_default_settings"){
    code{
        generation.swct{
            generate()
        }
    }
}
```

Listing 6.170: SWC Templates and Contract Headers generation with standard project settings

6.7.3.3 Generation of all Software Components

To execute the Swct generation for all available software components the following snippet can be used:

```
scriptTask("generate_with_default_settings"){
    code{
        generation.swct{
            settings.selectAll()
            generate()
        }
    }
}
```

Listing 6.171: SWC Templates and Contract Headers generation of all components

6.7.3.4 Generation of one Software Component

This sample selects one specific software component and starts the generation. There are two ways to open an settings block:

- **settings**
  - This keyword creates empty settings. E.g. no component is selected for execution.
- **settingsFromProject**
  - This keyword takes the project settings as template. E.g. component from the project settings are initially activated and can optionally be refined by explicit selections.

```
scriptTask("generate_one_component"){
    code{
        generation.swct{
            settings{
                selectSoftwareComponent("MyApplType")
            }

            generate()
        }
    }
}
```

Listing 6.172: SWC Templates and Contract Headers generation of one selected component

Instead of selecting the software component directly by its `Name`, there is also the possibility to fetch the software component object and `select()` this object for execution.

```
scriptTask("generate_one_component"){ code{
    generation.swct{
        settings{
            def sw = softwareComponentByName("MyApplType")
            // Select the software component
            sw.select()

            // You could also retrieve information about the component
            def asrPath = sw.asrPath
            if(sw.selected){  /* Do something */ }
        }
        generate()
   }
}}
```

Listing 6.173: Swct generation get component and select component

6.7.3.5 Generation of multiple Software Components

To select more than one Software Component the following snippet can be used.

```
scriptTask("generate_one_component"){
    code{
        generation.swct{
            settings{

                // Select the tow software components
                selectSoftwareComponent("MyApplType","MySecondApplType")
            }

            generate()
        }
    }
}
```

Listing 6.174: Swct generation of multiple components

6.7.3.6 Set a user defined logger

It is possible to pass a specific logger to the Swct generation settings. So all generation events (Phase startet, Module started...) are additionally logged to this logger.

```
    generation.swct{
        settings{
            setUserLogger(userLogger)
        }
        generate()
    }
```

Listing 6.175: Set a user defined logger

6.7.3.7 Evaluate generation results

The same API is used as for the normal generation, see chapter 6.7.1.3 on page 131 for details.

6.8 Validation

6.8.1 Introduction

All examples in this chapter are based on the scenario shown below. The module and the validators are not from the real MICROSAR stack, but just for the examples.

As shown in the 6.8, there is a module `Tp` that has 3 `Buffer` containers and each `Buffer` has a `Size` parameter with value=3. There is also a validator that requires the `Size` parameter to be a multiple of 4. For each `Size` parameter that violates this constraint, a validation-result with ID `TP00012` is created, as shown in the 6.8.

Such a validation-result has 2 solving-actions. One that sets the `Size` to the next smaller valid value, and one that sets the `Size` to the next bigger valid value. The later solving-action is marked as preferred-solving-action.
There is also a `TP00011` result that stands for any other result. The examples will not touch it.

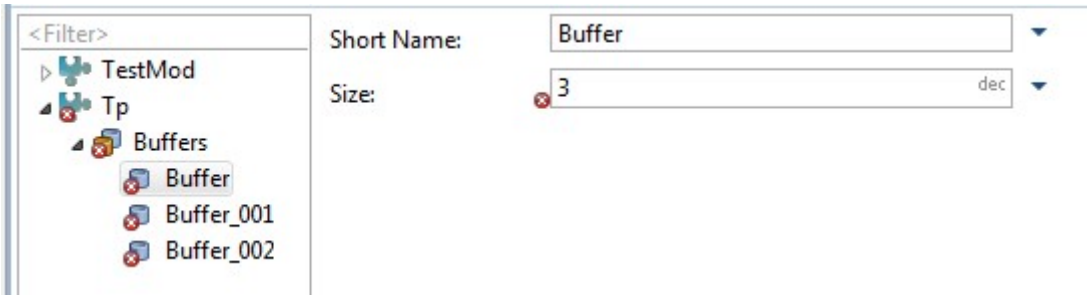


Figure 6.8: example module

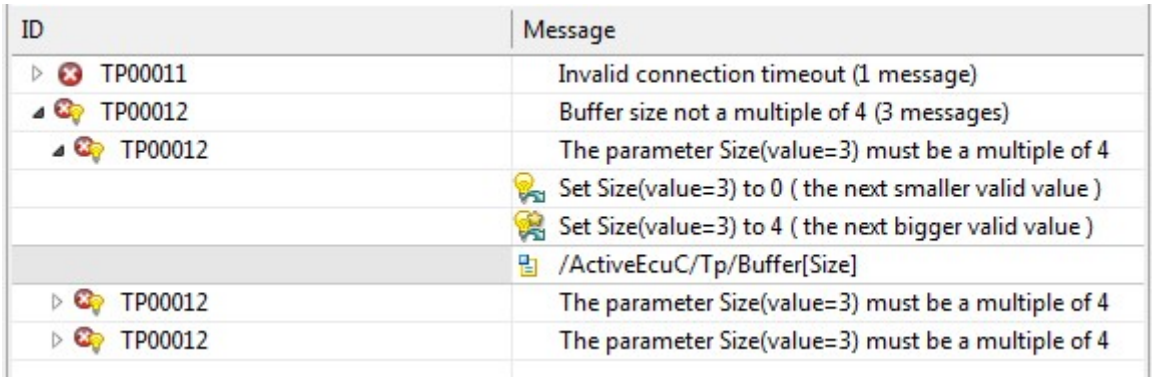


Figure 6.9: example validation results

Remark:

The validation-results to solve are identified via their ID. This ID is case-sensitive. Validation-Result-IDs of MICROSAR BSW modules are usually in capital letters (e.g. `COM02325`). Other validation-results may use validation-IDs in camel-case style (e.g. `Cfg00022`).

6.8.2 Access Validation-Results

A `validation{}` block gives access to the validation API of the consistency component. That means accessing the validation-results that are existing in the consistency, and solving them by executing solving-actions which belong to each individual result.

`validationResults` in AutomationIf waits for background-validation-idle and returns all validation-results of any kind. The returned collection has no deterministic order

```
scriptTask("CheckValidationResults_filterByOriginId", DV_PROJECT){
    code{
        validation{
            // access all validation-results
            def allResults = validationResults
            assert allResults.size() > 3

            // filter based on methods of IValidationResultUI e.g. isId()
            def tp12Results = validationResults.filter{it.isId("TP", 12)}
            assert tp12Results.size() == 3
        }

        // alternative access to validation-results without a validation block
        assert validation.validationResults.size() > 3
    }
}
```

Listing 6.176: Access all validation-results and filter them by ID

6.8.3 Model Transaction and Validation-Result Invalidation

Before we continue in this chapter with solving validation-results, the following information is import to know:

Relation to model `transactions`:

Solving validation-results with solving-actions always creates a transaction implicitly. An `IllegalStateException` will be thrown if this is done within an explicitly opened `transaction`.

Invalidation of validation-results:

Any model modification may invalidate any validation-result. In that case, the responsible validator creates a new validation-result if the inconsistency still exists. Whether this happens for a particular modification/validation-result depends on the validator implementation and is not visible to the user/client.

Trying to solve an invalidated validation-result will throw an `IllegalStateException`. Therefore it is not safe to solve a particular `ISolvingActionUI` that was fetched before the last transaction. Instead, please fetch a solving-action after the last transaction, or use the method `ISolver.solve(Closure)` which is the most preferred way of solving validation-results with solving-actions.

See chapter 6.8.4.1 on the next page for details.

6.8.4 Solve Validation-Results with Solving-Actions

A single validation-result can be solved by calling `solve()` on one of its solving-actions.

```
scriptTask("SolveSingleResultWithSolvingAction", DV_PROJECT){
    code{
        validation{
            def tp12Results = validationResults.filter{it.isId("TP", 12)}
            assert tp12Results.size() == 3

            // Take first (any) validation-result and filter its solving-actions
                based on methods of ISolvingActionUI
            tp12Results.first.solvingActions.filter{
                it.description.contains("next bigger valid value")

            }.single.solve() // reduce the collection to a single ISolvingActionUI
                 and call solve()

            assert validationResults.filter{it.isId("TP", 12)}.size() == 2
            // One TP12 validation-result solved
        }
    }
}
```

Listing 6.177: Solve a single validation-result with a particular solving-action

6.8.4.1 Solver API

`getSolver()` gives access to the `ISolver` API, which has advanced methods for bulk solutions.

`ISolver.solve(Action)` allows to solve multiple validation-results within one transaction.
You should always use this method to solve multiple validation-results at once instead of calling `ISolvingActionUI.solve()` in a loop. This is very important, because solving one validation-result, may cause invalidation of another one. And calling `ISolvingActionUI.solve()` of an invalidated validation-result throws an `IllegalStateException`. Also, invalidated validation-results may get recalculated and you would miss the recalculated validation-results with the loop approach. But with `ISolver.solve(Action)` you can solve invalidated->recalculated results as well as results which didn't exist at the time of the call (but have been caused by solving some other validation-result).

`ISolver.solve(Action)` first waits for background-validation-idle in order to have reproducible results.

The closure may contain multiple statements like:

```
result{specify result predicate}.withAction{select solving action}
```

All statements together will be used as a mapper from any solvable validation-result to a particular solving-action. The order of these statements does not affect the solving action execution order. The statement order might only be relevant if multiple statements match on a particular result, but would select a different solving-action. In that case, the first statement that successfully selects a solving-action wins.

```
scriptTask("SolveMultipleResults", DV_PROJECT){
  code{
    validation{
      assert validationResults.size() == 4
      solver.solve{
        // Call result() and pass a closure that works as filter
        // based on methods of IValidationResultUI.
        result{
          isId("TP", 12)
        }
        // On the return value, call withAction() and pass a closure that
        // selects a solving-action based on methods
        // of IValidationResultForSolvingActionSelect
        .withAction{
          containsString("next bigger valid value")
        }

        // multiple result() calls can be placed in one solve() call.
        result{isId("COM", 34)}.withAction{containsString("recalculate")}
      }

      // Three TP12 and zero COM34 (didn't exist) results solved. One other left
      assert validationResults.size() == 1
}}}
```

Listing 6.178: Fast solve multiple results within one transaction

Solve all PreferredSolvingActions `ISolver.solveAllWithPreferredSolvingAction()` solves all validation-results with their preferred solving-action (solving-action return by `IValidationResultUI.getPreferredSolvingAction()`). Validation-results without a preferred solving-action are skipped.

This method first waits for background-validation-idle in order to have reproducible results.

```
scriptTask("SolveAllWithPreferred", DV_PROJECT){
  code{
    validation{
      assert validationResults.size() == 4

      solver.solveAllWithPreferredSolvingAction()

      assert validationResults.size() == 1

      // this would do the same
      transactions.transactionHistory.undo()
      assert validationResults.size() == 4

      solver.solve{
        result{true}.withAction{preferred}
      }

      assert validationResults.size() == 1
}}}
```

Listing 6.179: Solve all validation-results with its preferred solving-action (if available)

6.8.5 Advanced Topics

6.8.5.1 Erroneous CEs of a Validation-Result

To check if a certain model element is affected by the result please use the following methods:

- `IValidationResultUI.matchErroneousCE(MIObject)`
- `IValidationResultUI.matchErroneousCE(IHasModelObject)`
- `IValidationResultUI.matchErroneousCE(MIHasDefinition, DefRef)`

```
scriptTask("IValidationResultUIErroneousCEs", DV_PROJECT){
    code{
        validation{
            // sampleDefRefs contains DefRef constants just for this example.
                Please use the real DefRefs from your SIP

            def result = validationResults.filter{it.isId("TP", 12)}.first

            // Retrieve the model element to check
            def modelElement // = retrieveElement ...

            // Check if the model object is affected by the validation-result
            assert result.matchErroneousCE(modelElement)

        }
    }
}
```

Listing 6.180: CE is affected by (matches) an IValidationResultUI

6.8.5.2 Access Validation-Results of a Model Object

You can retrieve validation-results also from any model object (MDF, Domain or BswmdModel).

`MIObject.validationResults` returns the validation-results of an `MIObject`.

```
scriptTask("CheckValidationResultsOfObject", DV_PROJECT){
    code{
        // sampleDefRefs contains DefRef constants just for this example. Please
            use the real DefRefs from your SIP

        // a Buffer container
        def buffer002 = mdfModel(AsrPath.create("/ActiveEcuC/Tp/Buffer_002"))
        // the Size parameter
        def sizeParam = buffer002.parameter(sampleDefRefs.tpBufferSizeDefRef).
            single

        // the result exists for the Size parameter, not for the Buffer container
        assert sizeParam.validationResults.size() == 1
        assert buffer002.validationResults.size() == 0
    }
}
```

Listing 6.181: Access all validation-results of a particular object

`MIObject.validationResultsRecursive` returns the validation-results of an `MIObject` and all its children.

`IViewedModelObject.validationResults` returns the validation-results for the element matching the model object and model view.

The following condition must be true to match:

```
IValidationResultUI.matchErroneousCE(theObject) &&
(
  IValidationResultUI.isGeneralVariantContext() ||
  IValidationResultUI.getPredefinedVariantContexts().contains(theView)
)
```

`IViewedModelObject.validationResultsRecursive` returns the validation-results of an `MIObject` and all its children. This will also filter for the correct `com.vector.cfg.model.asr.view.IModelView`. So this will return all results of the whole subtree, like an editor displays results at parent objects.

6.8.5.3 Access Validation-Results of a DefRef

`DefRef.validationResults` returns all validation-results which match the given definition. This means for each validation-result that is returned, at least one of its configuration elements has the given definition.

```
scriptTask("CheckValidationResultsOfDefRef", DV_PROJECT){
    code{
        // sampleDefRefs contains DefRef constants just for this example. Please
            use the real DefRefs from your SIP

        assert sampleDefRefs.tpBufferSizeDefRef.validationResults.size() == 3
    }
}
```

Listing 6.182: Access all validation-results of a particular DefRef

6.8.5.4 Filter Validation-Results using an ID Constant

Groovy allows you to spread list elements as method arguments using the spread operator. This allows you to define constants for the `isId(String,int)` method.

```
scriptTask("FilterResultsUsingAnIdConstant2", DV_PROJECT){
    code{
        validation{
            def tp12Const = ["TP",12]

            assert validationResults.size() > 3
            assert validationResults.filter{it.isId(*tp12Const)}.size() == 3
        }
    }
}
```

Listing 6.183: Filter validation-results using an ID constant

6.8.5.5 Identification of a Particular Solving-Action

A so called solving-action-group-ID identifies a solving-action group globally unique.

If solving-action groups are used, it is much safer to use the solving-action-group-IDs for solving-action identification than description-text matching, because a description-text may change.

```
final String SA_GROUP_ID_TP12_NEXT_BIGGER_VALID_VALUE = "ESolvingActionGroup#2"

scriptTask("SolveMultipleResultsByGroupId", DV_PROJECT){
  code{
    validation{
      assert validationResults.size() == 4

      solver.solve{
        result{isId("TP", 12)}
          .withAction{
            byGroupId(SA_GROUP_ID_TP12_NEXT_BIGGER_VALID_VALUE)
          }
          // instead of .withAction{containsString("next bigger valid value")}
      }

      assert validationResults.size() == 1
      // Three TP12 validation-results solved.
    }
  }
}
```

Listing 6.184: Fast solve multiple validation-results within one transaction using a solving-action-group-ID

6.8.5.6 Validation-Result Description as MixedText

`IValidationResultUI.getDescription()` returns an `IMixedText` that describes the inconsistency.

`IMixedText` is a construct that represents a text, whereby parts of that text can also hold the object which they represent. This allows a consumer e.g. a GUI to make the object-parts of the text clickable and to reformat these object-parts as wanted.
Consumers which don't need these advanced features can just call `IMixedText.toString()` which returns a default format of the text.

6.8.5.7 Further IValidationResultUI Methods

The following listing gives an overview of other "properties" of an IValidatonResultUI.

```
scriptTask("IValidationResultUIApiOverview", DV_PROJECT){
  code{
    validation{
      def r = validationResults.filter{it.isId("TP", 12)}.first
      assert r.id.origin == "TP"
      assert r.id.id == 12
      assert r.description.toString().contains("must be a multiple of")
      assert r.severity == EValidationSeverityType.ERROR
      assert r.solvingActions.size() == 2
      assert r.getSolvingActionByGroupId("ESolvingActionGroup#2").description.
          contains("next bigger valid value")

      // this result has a preferred-solving-action
      assert r.preferredSolvingAction == r.getSolvingActionByGroupId("
          ESolvingActionGroup#2")

      // results with lower severity than ERROR can be acknowledged
      assert r.acknowledgement.isPresent() == false

      // if the cause was an exception, r.cause.get() returns it
      assert r.cause.isPresent() == false

      // an ERROR result gets reduced to WARNING if one of its erroneous CEs is
          user-defined (user-overridden)
      assert r.isReducedSeverity() == false

      // on-demand results are reported with the on-demand generator validation
      assert r.isOnDemandResult() == false
    }
  }
}
```

Listing 6.185: IValidationResultUI overview

6.8.5.8 IValidationResultUI Acknowledgement

An IValidatonResultUI can have an acknowledgement and this acknowledgement will be stored within the project. The acknowledgement can be read and edited with the following APIs:

- `boolean isAcknowledged()`
- `Optional<String> getAcknowledgement()`
- `void setAcknowledgement(String)`

However, the acknowledgement can only be edited if the relevant files are writable. This information is stored in project settings.

The following API can be used to check if the acknowledgement is read-only:

- `boolean isAcknowledgementReadOnly()`

Note: When the `setAcknowledgement(String)` method is called, it will internally open a transaction to persist the acknowledgement. It is not allowed to call this method inside another transaction, otherwise, an IllegalStateException exception will be thrown from the `setAcknowledgement(String)` method.

6.8.5.9 IValidationResultUI in a variant (Post-Build selectable) Project

```
scriptTask("IValidationResultUIInAVariantProject", DV_PROJECT){
    code{
        validation{
            def r = validationResults.filter{it.isId("TP", 12)}.first
            assert r.isGeneralVariantContext() // either it is a general result...
            assert r.predefinedVariantContexts.size() == 0 // or it is assigned to
                 one or more (but never all) variants
            // If a validator assigns a result to all variants, it will be a
                general result at UI-side.
        }
    }
}
```

Listing 6.186: IValidationResultUI in a variant (post build selectable) project

Advanced Descriptor Details An `IDescriptor` is a construct that can be used to "point to" some location in the model. A descriptor can have several kinds of aspects to describe where it points to. Aspect kinds are e.g. `IMdfObjectAspect`, `IDefRefAspect`, `IMdfMetaClassAspect`, `IMdfFeatureAspect`.

`getAspect(Class)` gets a particular aspect if available, otherwise null.

A descriptor has a parent descriptor. This allows to describe a hierarchy.
E.g. if you want to express that something with definition X is missing as a child of the existing MDF object Y. In this example you have a descriptor with an `IDefRefAspect` containing the definition X. This descriptor that has a parent descriptor with an `IMdfObjectAspect` containing the object Y.

The term descriptor refers to a descriptor together with its parent-descriptor hierarchy.

```
import com.vector.cfg.model.cedescriptor.aspect.*

scriptTask("IValidationResultUIErroneousCEs", DV_PROJECT){
    code{
        validation{
            // sampleDefRefs contains DefRef constants just for this example.
                Please use the real DefRefs from your SIP

            def result = validationResults.filter{it.isId("TP", 12)}.first
            def descriptor = result.erroneousCEs.single // this result in this
                example has only a single erroneous-CE descriptor
            def defRefAspect = descriptor.getAspect(IDefRefAspect.class)
            assert defRefAspect != null // this descriptor in this example has an
                IDefRefAspect
            assert defRefAspect.defRef == sampleDefRefs.tpBufferSizeDefRef
            def objectAspect = descriptor.getAspect(IMdfObjectAspect.class)
            assert objectAspect != null // // this descriptor in this example has
                an IMdfObjectAspect
            // An IMdfObjectAspect would be unavailable for a descriptor
                describing that something is missing
            def parentObjectAspect = descriptor.parent.getAspect(IMdfObjectAspect.
                class)
            assert parentObjectAspect != null

            // Dealing with descriptors is universal, but needs more code. Using
                these methods might fit your needs.
            assert result.matchErroneousCE(objectAspect.getObject())
            assert result.matchErroneousCE(parentObjectAspect.getObject(),
                sampleDefRefs.tpBufferSizeDefRef)
        }
    }
}
```

Listing 6.187: Advanced use case - Retrieve Erroneous CEs with descriptors of an IValidationResultUI

6.8.5.10 Examine Solving-Action Execution

The easiest and most reliable option for verifying solving-action execution is to check the presence of validation-results afterwards.

Apart from that, there are other options of examination:

`ISolvingActionUI.solve()` returns an `ISolvingActionExecutionResult`. An `ISolvingActionExecutionResult` represents the result of one solving action execution. Use `isOk()` to find out if it was successful. Call `getUserMessage()` to get the failure reason.

`ISolver.solve(Action)` returns an `ISolvingActionSummaryResult`. An `ISolvingActionSummaryResult` represents the execution of multiple results. `ISolvingActionSummaryResult.isOk()` returns true if `getExecutionResult()` is `EExecutionResult.SUCCESSFUL` or `EExecutionResult.WARNING`, this is if at least one sub-result was ok.

Call `getSubResults()` to get a list of `ISolvingActionExecutionResults`.

```
import com.vector.cfg.util.activity.execresult.EExecutionResult

scriptTask("SolvingReturnValue", DV_PROJECT){
    code{
        validation{
            assert validationResults.size() == 4
            // In this example, three validation-results have a preferred solving
                action.
            // One of the three cannot be solved because a parameter is user-
                defined.
            def summaryResult = solver.solveAllWithPreferredSolvingAction()
            assert validationResults.size() == 2 // Two have been solved, one with
                a preferred solving-action is left.
            assert summaryResult.executionResult == EExecutionResult.WARNING

            // DemoAsserts is just for this example to show what kind of sub-
                results the summaryResult contains.
            DemoAsserts.summaryResultContainsASubResultWith("OK",summaryResult)
            //two such sub-results for the validation-results with preferred-
                solving-action that could be solved

            DemoAsserts.summaryResultContainsASubResultWith(["invalid modification"
                ,"not changeable","Reason","is user-defined"],summaryResult)
            // such a sub-result for the failed preferred solving action due to the
                 user-defined parameter

            DemoAsserts.summaryResultContainsASubResultWith("Maximum solving
                attempts reached for the validation-result of the following solving
                -action",summaryResult)
            // Multiple attempts are taken to solve a result because other changes
                may eliminate a blocking reason, but stops after an execution limit
                 is reached.
        }
    }
}
```

Listing 6.188: Examine an ISolvingActionSummaryResult

6.8.5.11 Create a Validation-Result in a Script Task

The `resultCreation` API provides methods to create new `IValidationResults`, which could then be reported to a `IValidationResultSink`. This is can be used to report validation-results similar to a validator/generator, but from within a script task.

ValidationResultSink You can retrieve an `IValidationResultSink` from the method `getResultSink()`. Or you get it by the context, e.g. some script tasks pass an `IValidationResultSink` as argument (like `DV_GENERATION_STEP`).

Reporting ValidationResult in Task providing a ResultSink This sample applies to task types providing a ResultSink in the Task API, like `DV_GENERATION_STEP`.

```
scriptTask("ScriptTaskCreationResult" /* Insert with task type providing
    resultSink */ ){
  code{
    validation{
      resultCreation{
        // The ValidationResultId group multiple results
        def valId = createValidationResultIdForScriptTask(
                /* ID */ 1234,
                /* Description */ "Summary of the ValidationResultId",
                /* Severity */ EValidationSeverityType.ERROR)
        // Create a new resultBuilder
        def builder = newResultBuilder(valId, "Description of the Result")

        // You can add multiple elements as error objects to mark them
        builder.addErrorObject(bswDefRef.EcucGeneral.bswmdModel().single)
        // Add more calls when needed

        // Create the result from the builder
        def valResult = builder.buildResult()

        // You need to report the result to a resultSink
        // You have to get the sink from the context, e.g. script task args
        // a sample line would be
        resultSinkForTask.reportValidationResult(valResult)
      }
    }
}}
```

Listing 6.189: Create a ValidationResult

```
scriptTask("ScriptTaskCreationResult", DV_PROJECT){
  code{
    validation{
      resultCreation{
        // The ValidationResultId group multiple results
        def valId = createValidationResultIdForScriptTask(
                /* ID */ 1234,
                /* Description */ "Summary of the ValidationResultId",
                /* Severity */ EValidationSeverityType.ERROR)
        // Create a new resultBuilder
        def builder = newResultBuilder(valId, "Description of the Result")

        // Create the result from the builder
        def valResult = builder.buildResult()

        // Access a resultSink to report an on-demand result
        resultSink.reportValidationResult(valResult)

        // With this method you can clear all the on-demand results
        clearOnDemandValidationResults()
 }}}}
```

Listing 6.190: Report a ValidationResult

6.8.5.12 Clear the on-demand ValidationResult

As shown in the above example, you can clear on-demand results when needed.

With the method `clearOnDemandValidationResults()`, you can clear all OnDemand results that are currently existing in the Consistency.

6.8.5.13 Turn off auto-solving-action execution

Auto-solving-action execution is a feature to simplify configuration by automatically adjusting dependent data after a change was made by the user. This feature runs synchronous to the user change and may have impact on UI responsiveness. If UI response time is not acceptable, this should be reported to Vector.

Using `setEnabled(boolean)`, auto-solving-action execution can be disabled to find out if this is the cause and as an interim workaround.

If auto-solving-action execution is disabled, data might get out of sync after a user change, E.g. Vtt dual target sync, BSW Internal Behavior, ... In that case, these have to be solved manually with the corresponding validaton-result's solving action.

This setting is stored as user-independent project setting.

This setting can only be changed if `isChangeable()` returns true (false e.g. due to read-only project), otherwise an `IllegalStateException` is thrown.

```
scriptTask("SolvingReturnValue", DV_PROJECT){
    code{
        validation{
            settings{
                if (autoSolvingActionExecution.changeable) {
                    autoSolvingActionExecution.enabled = false
                }
            }
        }
    }
}
```

Listing 6.191: Turn off auto solving action execution

6.9 SystemDescription and StructuredExtract

Beside the raw MDF Model to access the AUTOSAR model, the Configurator provides a lightweight facade model abstracting parts of the system description modelling of AUTOSAR. This so-called `SIModel` provides more convenience in general, and is especially needed to work on the `StructuredExtract` in a comfortable way.

`StructuredExtract` in AUTOSAR denotes the `System with category SYSTEM_DESCRIPTION` which contains the hierarchic application software composition. A key feature of the `SIModel` is the ability to create component prototypes and to connect them to arbitrary other component prototypes in the composition hierarchy.

Another key feature of the `SIModel` is the ability for permanent bridging to the underlying MDF Model: The `SIModel`-facades will never provide a complete wrapper for the AUTOSAR model. Instead, at any point in the `SIModel` it is possible to navigate to the underlying MDF model via `SIModelObject::getMdfObject()` to access non-abstracted features. Since the `SIModel`-facades are stateless, this will never interfere with the `SIModel`.

AUTOSAR also defines the `FlatExtract` as `System with category ECU_EXTRACT`, which contains the ECU flat view of all software components. The Configurator promotes working directly on the `StructuredExtract`, even for the ECU-centric use case. To this end, it utilizes the `SIModel` to also provide a `FlatView` of the `StructuredExtract`, letting a user browse the structured AUTOSAR model as if it was a `FlatExtract`. Figure 6.10 shows an example illustrating the relationship between the contents of a `StructuredExtract` and what becomes visible in the `FlatView`.

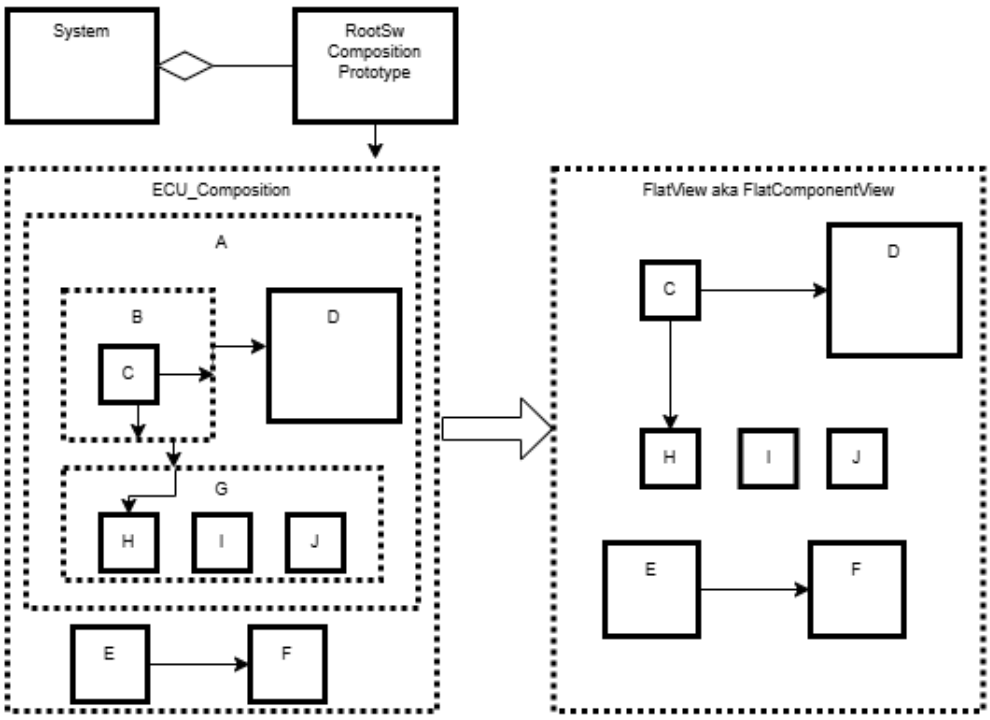


Figure 6.10: StructuredExtract and FlatView

The `FlatExtract` as defined by AUTOSAR is only available on demand as an export use-case. The `FlatExtract` export includes specific extensions of the AUTOSAR modelling which are hidden behind the `SIModel` for convenient handling.

6.9.1 ISysDescService and sysDescModel-keyword

The `sysDescModel`-block is fluent API entry point for accessing the `SIModel` for the system description. It provides the same methods as the project service `ISysDescService`.

The `ISysDescService` project service is the central entry point for accessing the `SIModel` for the system description. It provides:

- the **flat component view** of the `StructuredExtract` via `getFlatComponentView()`.
- the **structured component view** of the `StructuredExtract` via `getStructuredComponentView()`.
- the `StructuredExtract` itself via `getStructuredExtract()`.
- access to all `StructuredExtracts` for split use cases via `getStructuredExtracts()`.
- access to the **top-level composition** of the structured extract via `getStructuredExtractTopLevelComposition()`.
- generic access to a specific `SIModel`-object via `getSIModelObject(AsrPath)` or `getSIModelObject(MIObject)`.
- generic access to all instances of a certain SIModel-type via `getAllInstancesOfType(Class, EInstanceFiltering)`.
- the **AUTOSAR root element** via `getAutosarRoot()`.
- the preparation for `StructuredExtract`-usage in case no or an incomplete structured extract is provided externally via `prepareStructuredExtractUsage()`.
- access to the `SIFlatMap` and its `SIFlatInstanceDescriptors` via `getFlatMap()`.
- additional convenience-methods for structured extract usage.

6.9.2 StructuredExtract and FlatView

AUTOSAR describes the relation between the `System with category SYSTEM_DESCRIPTION` and the `System with category ECU_EXTRACT` as follows:

```
'AUTOSAR VFB Descriptions naturally form hierarchies of CompositionSwComponentTypes.
Consequently, in the System Configuration the SWC-related information for different
EcuInstances is not separated but in general is intermingled.  In contrast, for the
task of ECU configuration (RTE configuration, Service Configuration, Measurement and
Calibration) a hierarchically "flat view" on the SwComponentPrototypes running on the
EcuInstances is preferable over a hierarchical view, which is more favored by application-
software development.  Thus, deriving an System with category ECU_EXTRACT actually
is a model transformation, following a set of rules.'
```

This transformation (example shown in Figure 6.10 on the previous page) is provided via two APIs. `ISysDescService.getFlatComponentView()` provides the view of the software components, and `SIComponentPort.getFlatViewConnectedPorts()` provides the view of their connections. The `flattening` is done virtually, so each `SIComponent` has its full hierarchy information provided via `SIComponent.getContext()`. The details of the context concept is described in chapter 6.9.2.3 on the following page.

6.9.2.1 StructuredComponentView vs. FlatComponentView

Beside the flat component view - which only contains atomic software components as defined by AUTOSAR - the `ISysDescService` also provides a full list of software components which also contains all components of composition types and the top-level composition itself via `ISysDescService.getStructuredComponentView()`

The components in both lists provide their full hierarchy context. Since the shortname of a component may not be unique anymore in the hierarchy - either by choosing the same name for a component in different compositions or by multi-instantiation of the same composition - the according `SIComponent` provides a unique name in the hierarchy via `SIComponent.getUniqueSwcNameInHierarchy()`.

6.9.2.2 Component-Instantiation

Component-Instantiation in general is possible in all composition types. The `SIModel` works with feature-lists which provide according `createAndAdd`-methods for the actual list content. The most common use-case is to instantiate an atomic software component type in the top-level composition of the structured extract. Listing 6.192 shows an example where an application software component type is retrieved and instantiated as software component prototype in the top-level composition.

```
transaction {
   sysDescModel {
      // retrieve the application component type
      SIApplicationComponentType applicationComponentType = siModelObject(AsrPath.
          create("/a/b/c/MyApplicationSwcType")).orElse(null)

      // retrieve the top-level composition
      SICompositionComponentSubstitute topLevelComposition =
          structuredExtractTopLevelComposition.orElse(null)

      if (applicationComponentType != null && topLevelComposition != null) {

          // instantiate the component type by creating and adding a component to
              the top-level composition component feature-list
          SIApplicationComponent myComponent = topLevelComposition.component
                  .createAndAdd("MyComponent", SIApplicationComponent.class,
                      applicationComponentType)

      }

  }
}
```

Listing 6.192: Example of component instantiation

6.9.2.3 Context and CompositionComponentSubstitute

Working on a structured extract implies challenges regarding the software composition hierarchy, connections between and instance references to dedicated software components nested inside the hierarchy. Therefore, the SIModel facades introduce the `SICompositionComponentSubstitute` to represent the top-level composition of a composition hierarchy.

For all `SIComponents` obtained as children of an `SICompositionComponentSubstitute`, their `SIComponent.getContext()` method provides the full nesting hierarchy of compositions up to the `SICompositionComponentSubstitute`. For example, the context lists for the structured extract shown in figure 6.10 on page 150 can be illustrated as in figure 6.11 on the next page.

This has the following advantages and implications:

- The `SICompositionComponentSubstitute` allows to handle a top-level composition the same way as a component.
- Traversing the software composition hierarchy via an `SICompositionComponentSubstitute` ensures correct contexts for each contained `SIComponent`.
- The context of a component is a list always starting with the top-level `SICompositionComponentSubstitute`, followed by all `SICompositionComponents` in the hierarchy.
- All `SIComponentPorts` retrieved via `SIComponent.getComponentPort()` with full context can be connected via `IConnectionBuilder` without a caller having to consider the composition hierarchy (see 6.9.2.4).
- `SICommunicationElements` retrieved from `SIComponentPort` with full context can be used as target for `SIDataMapping.setCommunicationElement(SICommunicationElement)` to components nested in the hierarchy.
- `ISysDescService.getFlatComponentView()` and `ISysDescService.getStructuredComponentView()` each provide flat lists of all components in the hierarchy, with contexts set correctly for the `ISysDescService.getStructuredExtract()`.

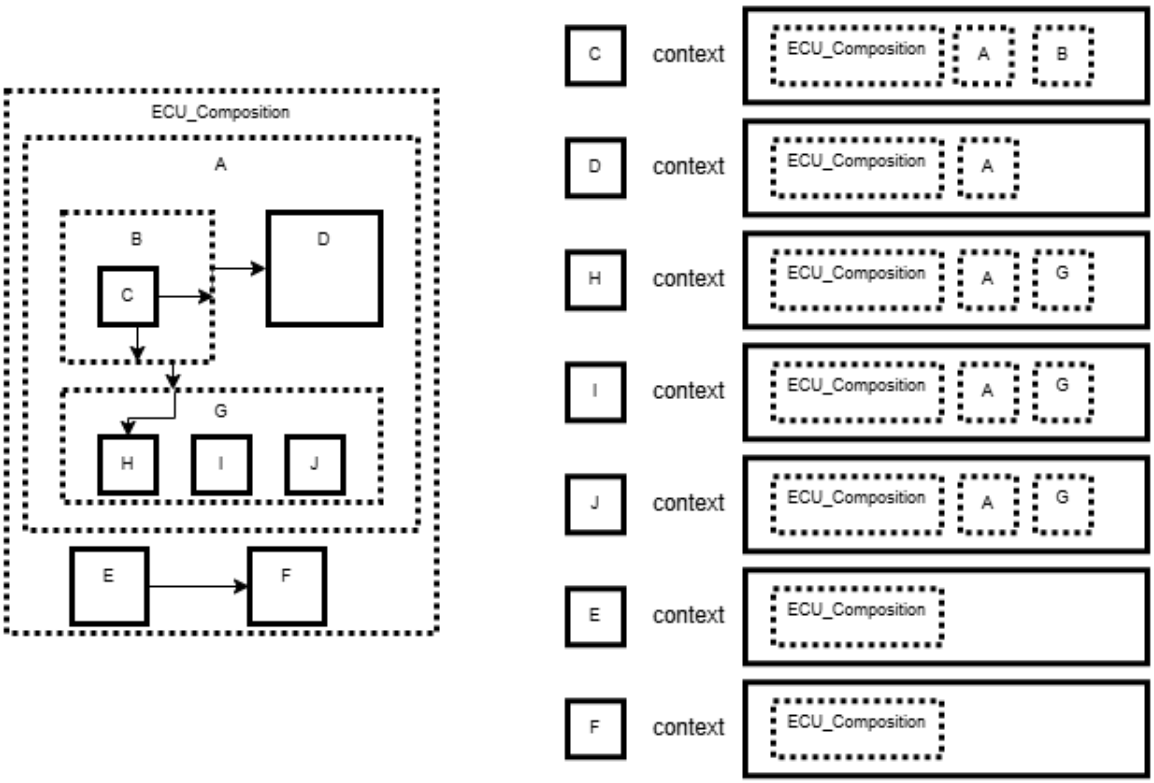


Figure 6.11: FlatViewComponents with context

6.9.2.4 ComponentPorts and ConnectionBuilder

The `SIComponentPort`-facade represents a port prototype bound to an actual component prototype: in AUTOSAR, this relation is only represented by an instance-reference e.g. to identify the requester and provider port of an assembly connector. In combination with the full context of the

bound `SIComponent`, this allows to build connection between arbitrary `SIComponentPorts`: therefore the SIModel provides the `IConnectionBuilder` to define connections with an appropriate provisioning of additional properties.

The listing 6.193 shows an example of how to create the connections as seen in figure 6.10 on page 150.

The `IConnectionBuilder` should be the preferred solution when building more than one connection due to performance reasons. When only one connection needs to be build occasionally and not all features are needed, the `SIComponentPort.connectTo(SIComponentPort)` creates one connection with a simpler entry.

```
transaction {
   sysDescModel {
      // prepare connection builder, flat component view and retrieval functions
      IConnectionBuilder connectionBuilder = getProjectContext().getService(
          ISysDescBuilderFactory.class)
               .createConnectionBuilder()
      List<SIComponent> flatComponentView = getFlatComponentView()
      Function<String, SIComponent> getComponent = (name) -> flatComponentView.
          stream()
               .filter(comp -> comp.getName().equals(name))
               .findFirst().orElse(null)
      BiFunction<String, SIComponent, SIComponentPort> getComponentPort = (name,
          comp) -> comp
               .getComponentPort().stream()
               .filter(compPort -> compPort.getPortName().equals(name))
               .findFirst().orElse(null)

      // retrieve components to connect
      final SIComponent c = getComponent.apply("C")
      final SIComponent d = getComponent.apply("D")
      final SIComponent h = getComponent.apply("H")
      final SIComponent e = getComponent.apply("E")
      final SIComponent f = getComponent.apply("F")

      // prepare the connection builders
      IPreparedConnectionBuilder builder1 = connectionBuilder
               .setProviderPort(getComponentPort.apply("PPort1", c))
               .setRequesterPort(getComponentPort.apply("RPort", d))
               .prepare()
      IPreparedConnectionBuilder builder2 = connectionBuilder
               .setProviderPort(getComponentPort.apply("PPort2", c))
               .setRequesterPort(getComponentPort.apply("RPort", h))
               .prepare()
      IPreparedConnectionBuilder builder3 = connectionBuilder
               .setProviderPort(getComponentPort.apply("PPort", e))
               .setRequesterPort(getComponentPort.apply("RPort", f))
               .prepare()

      // now build the connections
      IConnectionData connections1 = builder1.build()
      IConnectionData connections2 = builder2.build()
      IConnectionData connections3 = builder3.build()
    }
}
```

Listing 6.193: Example of connection builder

6.9.3 Examples

```
sysDescModel {
    flatComponentView.each { // flat view of the structured extract

        // components in the flat component view provide a unique name for the
            flat view
        def uniqueSwcNameInHierarchy = it.getUniqueSwcNameInHierarchy()

        // components in the flat component view contain the full context
            hierarchy
        scriptLogger.info("Component '{0}' with context: {1}",
            uniqueSwcNameInHierarchy, it.context.stream().map(SIComponent::getName
            ).collect(Collectors.joining(".")))

        switch (it) {
            case it instanceof SIApplicationComponent:
                def appType = it.componentType

                // example bridge to mdf
                appType.runnableEntity.each {
                   handleAppTypeEntitiesOnMdfLevel(appType.mdfObject, it.mdfObject
                       )
                }
            break
        case it instanceof SIServiceComponent:
            handleServiceComponent(it)
            break
        default:
           handleDefault()
        }
    }
}
```

Listing 6.194: Example of accessing the flat component view

```
transaction {
  sysDescModel {
    List<SIComponent> flatComponentViewComponents = flatComponentView

    SIComponentPort sourcePort = flatComponentViewComponents.stream()
        .filter(comp -> comp.getName().equals("SourceComponent"))
        .flatMap(comp -> comp.getComponentPort().stream())
        .filter(componentPort -> componentPort.getPortName().equals("SourcePort"))
        .findFirst().orElse(null)

    SIComponentPort targetPort = flatComponentViewComponents.stream()
        .filter(comp -> comp.getName().equals("TargetComponent"))
        .flatMap(comp -> comp.getComponentPort().stream())
        .filter(componentPort -> componentPort.getPortName().equals("TargetPort"))
        .findFirst().orElse(null)

    if (sourcePort != null && targetPort != null) {

        // Connects either direct (same Composition-Level, or ServiceConnector).
        // Or does create delegation ports (different Composition-Level and App-
            Components).
        sourcePort.connectTo(targetPort)


        boolean isDirectConnected = sourcePort.getConnectedPorts().contains(
            targetPort)
        boolean isConnectedViaChain = sourcePort.getFlatViewConnectedPorts().
            stream()
          .filter(condata -> condata.getConnectors().size() > 1)
          .anyMatch(condata -> targetPort.equals(condata.getConnectedComponentPort
              ()))

    }
  }
}
```

Listing 6.195: Example of connecting component ports in the flat component view

```
transaction {
   sysDescModel {
      // retrieve toplevel-composition of StructuredExtract
      def topLevelComposition = structuredExtract.get().topLevelComposition

      // retrieve the service component
      topLevelComposition.component.stream().filter(comp -> "ServiceSWCTopLevel".
          equals(comp.name)).findFirst()
          .ifPresent(serviceComponent -> {

              // retrieve an application component type, create the toplevel
                  prototype and connect it to a service component prototype
              def myAppComponentTypePath = AsrPath.create("/ComponentTypes/
                  ApplicationSWCTopLevel")
              siModelObject(myAppComponentTypePath).ifPresent(appType -> {
                  // create the component prototype
                  def myAppComponentPrototype = topLevelComposition.component.
                      createAndAdd("MyApp", SIApplicationComponent.class, appType)
                  def appComponentPPort = myAppComponentPrototype.componentPort.
                      getFirst()

                   // connector one of its ports to a service component
                  def serviceComponentRPort = serviceComponent.getComponentPort().
                      getFirst()
                  // create a connection on toplevel
                  topLevelComposition.getConnector().createAndAdd(
                      SIAssemblyConnector.class, appComponentPPort,
                      serviceComponentRPort)
              })

              // make a service connection into a composition hierarchy
              topLevelComposition.component.stream().filter(comp -> "
                  CompositionLayer1".equals(comp.name)).findFirst().ifPresent(
                  compositionComponent -> {

                  // get the inner component directly from the component prototype
                       to know the correct context
                  def hierarchicComponent = compositionComponent.component.
                      getFirst()
                  def hierarchicComponentPort = hierarchicComponent.componentPort.
                      getFirst()

                  // connect it to a service component
                  def serviceComponentRPort = serviceComponent.componentPort.get
                      (1)
                  // create a service connection into the hierarchy
                  topLevelComposition.connector.createAndAdd(SIServiceConnector.
                      class, hierarchicComponentPort,
                          serviceComponentRPort)
              })
          })
  }
}
```

Listing 6.196: Example of accessing the StructuredExtract

```
transaction {
   sysDescModel {
      // retrieve a AR package to create a new application component type
      def arPackage = siModelObject(AsrPath.create("/ComponentTypes")).get()

      // create a new application component type
      def appType = arPackage.element.createAndAdd("ExampleAppType",
          SIApplicationComponentType.class)

      // create a runnable entity and event
      def runnable = appType.runnableEntity.createAndAdd("ExampleRunnable")
      def timingEvent = appType.event.createAndAdd("ExampleTimingEvent",
          SITimingEvent.class)
      timingEvent.period = 0.1

      // connect the event to the runnable
      timingEvent.startOnEvent = runnable

  }
}
```

Listing 6.197: Example of accessing internal behavior elements

6.10 Domains

The domain APIs are specifically designed to provide high convenience support for typical domain use cases.

The domain API is the entry point for accessing the different domain interfaces. It is available in opened projects in the form of the `IDomainApi` interface.

`IDomainApi` provides methods for accessing the different domain-specific APIs. Each domain's API is available via the domain's name. For an example see the communication domain API 6.10.1.

`getDomain()` allows accessing the `IDomainApi`

```
scriptTask('taskName') {
  code {
    // IDiagnosticsApi is available as "diagnostics" property
    def diagnostics = domain.diagnostics
  }
}
```

Listing 6.198: Accessing IDiagnosticsApi as a property

`domain(Transformer)` allows accessing the `IDomainApi` in a scope-like way.

```
scriptTask('taskName') {
  code {
    domain.diagnostics {
      // IDiagnosticsApi is available here
    }
  }
}
```

Listing 6.199: Accessing IDiagnosticsApi in a scope-like manner

6.10.1 Communication Domain

The communication domain API is specifically designed to support communication related use cases. It is available from the `com.vector.cfg.automation.scripting.base.IAutomationContext.IDomainAp` 6.10 in the form of the `ICommunicationApi` interface.

`getCommunication()` allows accessing the `ICommunicationApi` like a property.

```
scriptTask('taskName') {
  code {
    // ICommunicationApi is available as "communication" property
    def communication = domain.communication
  }
}
```

Listing 6.200: Accessing ICommunicationApi as a property

`communication(Transformer)` allows accessing the `ICommunicationApi` in a scope-like way.

```
scriptTask('taskName') {
  code {
    domain.communication {
      // ICommunicationApi is available inside this Closure
    }
  }
}
```

Listing 6.201: Accessing ICommunicationApi in a scope-like way

The following use cases are supported:

Accessing Can Controllers `getCanControllers()` returns a list of all `ICanControllers` in the configuration 6.10.1.1 on the next page.

Accessing Can Pdus `getCanPdus()` returns a list of all `ICanPdus` in the configuration. Can Pdus of a certain Can Controller can also be accessed via `ICanController.getCanPdus()`. 6.10.1.3 on page 162

6.10.1.1 CanControllers

An `ICanController` instance represents a CanController `MIContainer` providing support for use cases exceeding those supported by the model API.

```
scriptTask('OptimizeAcceptanceFilters') {
  code {
      transaction {
        domain.communication {
          // open acceptance filters of all CanControllers
          canControllers*.openAcceptanceFilters()

          // open acceptance filters of first CanController
          canControllers.first.openAcceptanceFilters()
          canControllers[0].openAcceptanceFilters() // same as above

          // open acceptance filters of second CanController
          // (if there is a second CanController)
          canControllers[1]?.openAcceptanceFilters()

          // open acceptance filters of a dedicated CanController
          canControllers.filter { it.getMdfObject().getName().contains 'CH0' }.
              single.openAcceptanceFilters()

          // accessing a dedicated CanController
          def ch0 = canControllers.filter { it.getMdfObject().getName().contains '
              CH0' }.single

          // assert: ch0's first CanFilterMask value is XXXXXXXXXXX
          assert 'XXXXXXXXXXX' == ch0.canFilterMasks[0].filter

          // set CanFilterMask value to 01111111111
          ch0.canFilterMasks[0].filter = '01111111111'
          assert '01111111111' == ch0.canFilterMasks[0].filter

          // automatic acceptance filter optimization
          ch0.optimizeFilters { fullCan = true }
        }
      }
    scriptLogger.info('Successfully optimized Can acceptance filters.')
  }
}
```

Listing 6.202: Optimizing Can Acceptance Filters

Opening Acceptance Filters `openAcceptanceFilters()` opens all of this `ICanController`'s acceptance filters.

Optimizing Acceptance Filters `optimizeFilters(Action)` optimizes this `ICanController`'s acceptance filter mask configurations. The given `Closure` is delegated to the `IOptimizeAcceptanceFiltersApi` interface for parameterizing the optimization.

Using `setFullCan(boolean)` it can be specified whether the optimization shall take full can objects into account or not.

Creating new CanFilterMasks `createCanFilterMask()` creates a new `ICanFilterMask` for this `ICanController`.

Accessing a CanController's CanFilterMasks `getCanFilterMasks()` returns all of this `ICanController`'s `ICanFilterMasks`.

Accessing a CanController's MIContainer `getMdfObject()` returns the `MIContainer` represented by this `ICanController`.

6.10.1.2 CanFilterMasks

An `ICanFilterMask` instance represents a CanFilterMask `MIContainer` providing support for use cases exceeding those supported by the model API.

For example code see 6.10.1.1 on the previous page. The following use cases are supported:

Filter Types `ECanAcceptanceFilterType` lists the possible values for an `ICanFilterMask`'s filter type.

`STANDARD` results in a standard Can acceptance filter value with length 11.

`EXTENDED` results in an extended Can acceptance filter value with length 29.

`MIXED` results in a mixed Can acceptance filter value with length 29.

Accessing a CanFilterMask's Filter Type `getFilterType()` returns this `ICanFilterMask`'s filter type.

Specifying a CanFilterMask's Filter Type Using `setFilterType(ECanAcceptanceFilterType)` this `ICanFilterMask`'s filter type can be specified.

Accessing a CanFilterMask's Filter Value `getFilter()` returns this `ICanFilterMask`'s filter value. A CanFilterMask's filter value is a `String` containing the characters '0', '1' and 'X' (don't care). For determining if a given Can ID passes the filter it is matched bit for bit against the `String`'s characters. The character at index 0 is matched against the most significant bit. The character at index `length() - 1` is matched against the least significant bit. The length of the `String` corresponds to the CanFilterMask's filter type.

Specifying a CanFilterMask's Filter Value Using `setFilter(String)` this `ICanFilterMask`'s filter value can be specified.

Accessing a CanFilterMask's MIContainer `getMdfObject()` returns the `MIContainer` represented by this `ICanFilterMask`.

6.10.1.3 CanPdus

An `ICanPdu` represents a Pdu of the CanIf module.

- `disableFullCan()` disables the FullCAN feature for this `ICanPdu`. In other words the corresponding FullCAN hardware object will be deleted and the references redirected to a suitable BasicCAN hardware object.

To enable the FullCAN feature please use `enableFullCan()`.

- `enableFullCan()` enables the FullCAN feature for this `ICanPdu`. In other words a corresponding FullCAN hardware object will be created and the references of the relevant objects redirected to it from the previously referenced BasicCAN hardware object.

  To disable the FullCAN feature please use `disableFullCan()`. If the FullCAN feature is already enabled can be checked using `isFullCanEnabled()`. This might be useful to avoid multiple hardware objects for the same PDU, when using `enableFullCan()` more than once on the same PDU.

- `setFullCanLocked(boolean)` switches whether the filter optimization algorithm is allowed to change the configured CAN handle type (BasicCAN/FullCAN) or not.

  If locked the selected CAN handle type (FullCAN/BasicCAN) of the corresponding Rx-PDU is NOT touched and NOT changed by the filter optimization algorithm.

  Tx `ICanPdu`s are ignored by this method, since the lock is only available for Rx `ICanPdu`s.

- `isFullCanEnabled()` determines whether the configured CAN handle type is FullCAN or BasicCAN .

  The CAN handle type can be changed between FullCAN and BasicCAN using `enableFullCan()` and `disableFullCan()` methods.

- `isFullCanLocked()` determines whether the CAN handle type is locked for the filter optimization algorithm.
  Note: Only Rx `ICanPdu`s can be locked, so returns always false for Tx `ICanPdu`s.

Accessing a CanController's CanPdus `getCanPdus()` returns all of this `ICanController`'s `ICanPdu`s.

Accessing a CanPdu's MIContainer `getMdfObject()` returns the `MIContainer` represented by this `ICanPdu`.

```
import com.vector.cfg.model.asr.view.IModelViewExecutionContext

scriptTask("enableFullCAN", DV_PROJECT){
  code {
    transaction {
      domain.communication {
        variance {
          getAllPostBuildVariantViewsOrInvariant().each {
            if (it.getName().equals("Left")) {
              final IModelViewExecutionContext context = it.executeWithThisView()
              context.withCloseable {
                canPdus.filter {
                  it.getMdfObject().getName().equals("
                      RxFullCanDisabled1_0a94227e_Rx")
                }*.enableFullCan()
              }
            }
          }
        }
      }
    }
  }
}
```

Listing 6.203: Enable FullCAN feature for PDU

```
import com.vector.cfg.model.asr.view.IModelViewExecutionContext

scriptTask("disableFullCAN", DV_PROJECT){
  code {
    transaction {
      domain.communication {
        variance {
          getAllPostBuildVariantViewsOrInvariant().each {
            if (it.getName().equals("Left")) {
              final IModelViewExecutionContext context = it.executeWithThisView()
              context.withCloseable {
                canControllers.filter {
                  it.getMdfObject().getName().equals("Controller_MyEcu_1_2514e902"
                      )
                }*.canPdus
                .flatten()
                *.disableFullCan()
              }
            }
          }
        }
      }
    }
  }
}
```

Listing 6.204: Disable FullCAN feature for all PDUs of CanController

6.10.1.4 J1939 Requestable Configuration

`IJ1939RCApi` provides high convenience support for J1939RC related use cases.

- `executeValidationRules()` checks the current EcuC configuration and sets the J1939 Requestable Flag on the PDU if necessary.

6.10.2 Diagnostics Domain

The diagnostics domain API is specifically designed to support diagnostics related use cases. It is available from the `com.vector.cfg.automation.scripting.base.IAutomationContext.IDomainApi` 6.10 on page 159 in the form of the `IDiagnosticsApi` interface.

`getDiagnostics` allows accessing the `IDiagnosticsApi` like a property.

```
scriptTask('taskName') {
  code {
    // IDiagnosticsApi is available as "diagnostics" property
    def diagnostics = domain.diagnostics
  }
}
```

Listing 6.205: Accessing IDiagnosticsApi as a property

`diagnostics(Transformer)` allows accessing the `IDiagnosticsApi` in a scope-like way.

```
scriptTask('taskName') {
  code {
    domain.diagnostics {
      // IDiagnosticsApi is available here
    }
  }
}
```

Listing 6.206: Accessing IDiagnosticsApi in a scope-like manner

The following use cases are supported:

Dem Events The API provides access and creation of `IDemEvents` in the configuration. See chapter 6.10.2.1 on the next page for more details.

Check for OBD II `isObd2Enabled()` checks, if OBD II is available in the configuration.

Enable OBD II `setObd2Enabled(boolean)` enables or disables OBD II in the configuration. Note, that OBD II can only be enabled, if a valid SIP license was found.

Check for WWH-OBD `isWwhObdEnabled()` checks, if WWH-OBD is available in the configuration.

Enable WWH-OBD `setWwhObdEnabled(boolean)` enables or disables WWH-OBD in the configuration. Note, that WWH-OBD can only be enabled, if a valid SIP license was found.

6.10.2.1 DemEvents

An `IDemEvent` instance represents a diagnostic event and and provides usecase centric functionalities to modify and query diagnostic events.

Accessing Dem Events `getDemEvents()` returns a list of all `IDemEvents` in the configuration.

Creating Dem Events `createDemEvent(Action)` is used to create diagnostic events of different kinds.

The method can be configured to create different types of DTCs/Events:

1. **UDS Event**: This is the default type of event, when only an 'eventName' and a 'dtc' number is specified. A new DemEventParameter container with the given shortname and a new DemDTCClass with the given DemUdsDTC is created.

```
scriptTask('taskName') {
    code {
        transaction {
            domain.diagnostics {

                def udsEvent = createDemEvent {
                    eventName = "NewUdsEvent"
                    dtc = 0x30
                }
}}}}
```

Listing 6.207: Create a new UDS DTC with event

2. **OBD II Event**: If OBD II is enabled for the loaded configuration, and a 'obd2Dtc' is specified instead of a 'dtc', the method will create an OBD II relevant event. The difference is, that it will set the parameter DemObdDTC instead of DemUdsDTC. It is also possible to specify 'dtc' as well as 'obd2dtc', which will result in both DTC parameters are set.

```
scriptTask('taskName') {
    code {
        transaction {
            domain.diagnostics {
                // OBD must be enabled and legislation must be OBD2
                // Enable OBD2
                obd2Enabled = true

                def obd2Event = createDemEvent {
                    eventName = 'NewOBD2Event'
                    obd2Dtc = 0x40
                }

                def obd2CombinedEvent = createDemEvent {
                    eventName = 'UDS_OBD2_Combined_Event'
                    dtc = 0x31
                    obd2Dtc = 0x41
                }
}}}}
```

Listing 6.208: Enable OBD II and create a new OBD related DTC with event

3. **WWH-OBD Event**: If WWH-OBD is enabled for the loaded configuration, and a 'wwhObdDtcClass' with a value other than 'NO_CLASS' is specified, the method will create a WWH-OBD relevant event. Note that WWH-OBD relevant events usually du reference the so called MIL indicator, thus this reference will be set by default in the newly created DemEventParameter.

```
scriptTask('taskName') {
    code {
        transaction {
            domain.diagnostics {
                // OBD must be enabled, and legislation must be WWH-OBD
                // The parameter '/Dem/DemGeneral/DemMILIndicatorRef' must be
                    set
                wwhObdEnabled = true

                def wwhObdEvent = createDemEvent {
                    eventName = 'WWHOBD_Event'
                    dtc = 0x50
                    // wwhObdClass != NO_CLASS indicates WWH-OBD event
                    wwhObdDtcClass = CLASS_A
                }
}}}}
```

Listing 6.209: Enable WWH-OBD and create a new OBD related DTC with event

4. **J1939 Event**: The last type of event is a J1939 related event, which can be created when J1939 is licensed and available for the loaded configuration. This is done in a similar way as for UDS events, but additionally specifying 'spn', 'fmi' values as well as the name of the referenced 'nodeAddress'.

```
scriptTask('taskName') {
    code {
        def nodeAddressContainer = mdfModel(AsrPath.create("/ActiveEcuC/Dem/
            DemConfigSet/DemJ1939NodeAddress", MIContainer))

        transaction {
            domain.diagnostics {
                // J1939 Event creation
                // J1939 must be enabled and License must be available.
                j1939Enabled = true

                def j1939Event = createDemEvent {
                    eventName 'J1939_Event'
                    dtc 0x30
                    spn 90
                    fmi 13
                    nodeAddress nodeAddressContainer
                }
}}}}
```

Listing 6.210: Open a project, enable J1939 and create a new J1939 DTC with event

Important Note:
For every DTC numbers apply the rule, that if there are already DemDTCClasses with the given number, they will be used. In such a case, no new DemDTCClass container is created.

6.10.3 Mode Management Domain

The mode management domain API is specifically designed to support mode management related use cases. It is available from the `com.vector.cfg.automation.scripting.base.IAutomationContext.IDomai` 6.10 on page 159 in the form of the `IModeManagementApi` interface.

`getModeManagement()` allows accessing the `IModeManagementApi` like a property.

```
scriptTask('taskName') {
  code {
    // IModeManagementApi is available as "modeManagement" property
    def modeManagement = domain.modeManagement
  }
}
```

Listing 6.211: Accessing IModeManagementApi as a property

`modeManagement(Transformer)` allows accessing the `IModeManagementApi` in a scope-like way.

```
scriptTask('taskName') {
  code {
    domain.modeManagement {
      // IModeManagementApi is available inside this Closure
    }
  }
}
```

Listing 6.212: Accessing IModeManagementApi in a scope-like way

6.10.3.1 BswM Auto Configuration

The `IBswMAutoConfigurationApi` allows for semi-automatic creation of dedicated parts of the BswM configuration. The BswM auto configuration takes an input consisting of "features" and "parameters" to be provided via the `IBswMAutoConfigurationApi`. Each feature may have zero, one or more sub-features and zero, one or more parameters.

The corresponding BswM configuration content is derived based on the (de)activation of features and the values assigned to the parameters.

The available features and parameters depend strongly on the project's input data and general project setup. They can be addressed by `String` identifiers. These identifiers are best obtained from the corresponding auto configuration assistant of the BSW management editor in the Cfg5 GUI.

```
scriptTask('EcuStateHandlingAutoConfiguration', DV_PROJECT) {
  code {
    // In projects with post-build selectable variance switching to an
    // IPredefinedVariantView for performing auto configuration is mandatory
    variance.variantView('Left').activeWith {

      domain.modeManagement.bswMAutoConfig('Ecu State Handling') {
        activate '/ECU State Machine/Support ComM'
        set '/ECU State Machine/Self Run Request Timeout' to 0.2
        set '/ECU State Machine/Number of Run Request User' to 4
        overrides {
          if (addition || removal) {
            keepOverride
          } else if (BswMArgumentRef.DefRef.isDefinitionOf(element)
                  && feature('/ECU State Machine/Support ComM/CAN00_f26020e5').
                      enabled
                  && parameter('/ECU State Machine/Number of PostRun Request User'
                      ).value == 4) {
            discardOverride
          } else {
            keepOverride
          }
        }
      }
    }
  }
}
```

Listing 6.213: ECU State Handling Auto Configuration

Executing the BswM Auto Configuration `IModeManagementApi.bswMAutoConfig(String, Transformer)` delegates the given code to the `IBswMAutoConfigurationApi` of the given BswM auto configuration domain. Also see overload `bswMAutoConfig(MIContainer, String, Transformer)` for using this API in multi partition use case.

Activating BswM Auto Configuration Features `activate(String)` activates the BswM auto configuration feature with the given identifier. All enabled sub-features of the specified feature are also activated. Imagine the features displayed in a tree structure (like in Cfg5 GUI) where checking a tree node automatically checks all children.

Deactivating BswM Auto Configuration Features `deactivate(String)` deactivates the BswM auto configuration feature with the given identifier. All enabled sub-features of the specified feature are also deactivated. Imagine the features displayed in a tree structure (like in Cfg5 GUI) where unchecking a tree node automatically unchecks all children.

Assigning Values to BswM Auto Configuration Parameters `set(String)` sets the parameter with the given identifier to the specified value. Supported value types are `boolean`, `BigInteger`, `Double`, `String` and `MIReferrable` (reference parameters).

Manually Adapting the BswM Auto Configuration Content The BswM auto configuration mechanism is useful for creating large parts of the BswM configuration based on certain built-in heuristics. Where these heuristics fail to fulfill detailed project specific requirements manual adaptations to the auto-generated configuration content become necessary.

Per default manual adjustments are kept in the configuration. But subsequent BswM auto configuration runs may render previously applied adjustments obsolete or dysfunctional. Using `overrides(Action)` a callback can be registered to be called for each detected adaptation. The callback can decide for each adjustment if it is to remain in the configuration or if it is to be overwritten by the BswM auto configuration. For details on which information is provided to this callback please refer to the javadoc provided with `IBswMAutoConfigurationOverride`.

Inspecting BswM Auto Configuration Domains The `getBswMAutoConfigDomains()` method of the `IModeManagementApi` interface provides read-access to all available BswM auto configuration domains. Available features and parameters can be inspected for various properties. See javadoc of `IBswMAutoConfigurationDomain`, `IBswMAutoConfigurationFeature` and `IBswMAutoConfigurationParameter` for details. Also see overload `getBswMAutoConfigDomains(MIContainer)` for using this API in multi partition use case.

```
domain.modeManagement {
  // In projects with post-build selectable variance switching to an
  // IPredefinedVariantView for inspecting auto configuration is mandatory
  variance.variantView('Left').activeWith {

    // get all BswM auto configuration domains
    bswMAutoConfigDomains.forEach {
      scriptLogger.info it.identifier
    }

    def isEnabled = bswMAutoConfigDomain 'Ecu State Handling' feature '/ECU State
        Machine/Support ComM' enabled
    def isActivated = bswMAutoConfigDomain 'Ecu State Handling' feature '/ECU
        State Machine/Support ComM' activated
    if (isEnabled && isActivated) {
      // activation state can be toggled at enabled features only
      bswMAutoConfig('Ecu State Handling') {
        deactivate '/ECU State Machine/Support ComM'
      }
    }

    bswMAutoConfigDomain('Ecu State Handling') {
      // this code is delegated to the 'Ecu State Handling'
      // auto configuration domain
      def p1 = parameter '/ECU State Machine/Self Run Request Timeout' value
      scriptLogger.info 'Self Run Request Timeout = ' + p1
      def p2 = parameter '/ECU State Machine/Number of Run Request User' value
      scriptLogger.info 'Number of Run Request User = ' + p2

      // get all root features
      rootFeatures.forEach { scriptLogger.info it.identifier }

      // get all sub-features of a feature
      feature '/ECU State Machine/Support ComM' subFeatures.forEach {
        scriptLogger.info it.identifier
      }

      // get all parameters of a feature
      feature '/ECU State Machine' parameters.forEach {
        scriptLogger.info it.identifier
      }
    }
  }
}
```

Listing 6.214: Inspecting Auto Configuration Elements

6.10.4 Runtime System Domain

The runtime system domain API is specifically designed to support runtime system related use cases. It is available from the `IAutomationContext.IDomainApi` (see 6.10 on page 159) in the form of the `IRuntimeSystemApi` interface.

Remark: The runtime system domain API is designed to work on the flat extract of the System Description. Since the flat extract is now only created on demand, the API works on the flat-view of the structured extract

`getRuntimeSystem()` allows accessing the `IRuntimeSystemApi` like a property.

```
scriptTask('taskName') {
  code {
    // IRuntimeSystemApi is available as "runtimeSystem" property
    def runtimeSystem = domain.runtimeSystem
  }
}
```

Listing 6.215: Accessing IRuntimeSystemApi as a property

`runtimeSystem(Transfomer)` allows accessing the `IRuntimeSystemApi` in a scope-like way.

```
scriptTask('taskName') {
  code {
    domain.runtimeSystem {
      // IRuntimeSystemApi is available inside this Closure
    }
  }
}
```

Listing 6.216: Accessing IRuntimeSystemApi in a scope-like way

The access point for elements of the runtime system domain are the selections. They are most of the time your starting point and offer predicates to filter for elements which are relevant. These selections can be used to get the elements for further work, but they also offer direct methods for actions that can be performed for them.

As default the selection APIs select always from all elements, but you can also put elements into most selections and use predicates to filter them. The put methods helps you to transfer the elements from selection to selection and put newly created elements into the next selection to perform the next step of your workflow.

We will start by introducing each selection API and will then have a look on common use cases. Before starting to implement a use case it might be helpful to have a quick look into the chapters of the selections that are required for it. The objects such as communication element or component port used in the runtime system domain API are explained in the chapter of the corresponding selection API.

6.10.4.1 Component Port Selection

A component port (`SIComponentPort`) represents a port prototype and its corresponding component prototype, and in case of a delegation port the corresponding top level composition type (ECU Composition).

`selectComponentPorts(Action)` allows the selection of `SIComponentPorts` using predicates.

The component port selection can be used to select and filter component ports and either do further operations on them, such as connecting to other ports, terminating them or to just return a list of component ports with which you can continue working.

`getComponentPorts()` allows access to the single component ports in the `IComponentPortSelection`.

Component Port Predicates To select component ports predicates can be provided to narrow down the result.

Per default the predicates are combined via logical AND. To realize other combinations, use the 'or','not' and 'and' predicates.

- `unconnected()` matches unconnected component ports.
- `connected()` matches connected component ports.
- `completed()` matches component ports which are completed.

  A delegation port is completed if and only if the port is connected and each of the port's communication elements is either data mapped to a system signal or a system signal group or referenced by flat instance descriptors referring to cross sw cluster RTE implementation plug-ins.

  An inner port is completed if the port is connected or each of the port's communication elements is data mapped to a system signal or a system signal group. This means an inner port which is connected and data mapped is also completed.
- `notCompleted()` matches component ports which are not completed.

  See `completed()` for the conditions a port has to meet to be a completed port.
- `terminated()` matches terminated component ports.
- `notTerminated()` matches non-terminated component ports.
- `senderReceiver()` matches component ports whose port has a sender/receiver port interface.
- `clientServer()` matches component ports whose port has a client/server port interface.
- `modeSwitch()` matches component ports whose port has a mode-switch port interface.
- `nvData()` matches component ports whose port has a NvData port interface.
- `parameter()` matches component ports whose port has a parameter (calibration) port interface.
- `trigger()` matches component ports whose port has a trigger port interface.
- `provided()` matches provided component ports (p-port).
- `required()` matches required component ports (r-port).
- `providedRequired()` matches provided-required component ports (pr-port).
- `delegation()` matches delegation ports (ports of the Ecu composition).
- `application()` matches component ports whose port interface is an application port interface.
- `service()` matches component ports whose port interface is an service port interface.
- `applicationComponent()` matches component ports whose component type is an application component type. Application component types are all component types which are not service component types, as displayed in the ECU Software Components Editor, not ApplicationSwComponentTypes as defined by AUTOSAR.
- `serviceComponent()` matches component ports whose component type is a service component type.

- `parameterComponent()` matches component ports whose component type is a parameter component type.
- `nvBlockComponent()` matches component ports whose component type is a nv block component type.
- `sensorActuatorComponent()` matches component ports whose component type is a sensor actuator component type.
- `ioHwAbstractionComponent()` matches component ports whose component type is a I/O hardware abstraction component type, also called EcuAbstractionSwComponentType.
- `complexDeviceDriverComponent()` matches component ports whose component type is a complex device driver component type.
- `serviceProxyComponent()` matches component ports whose component type is a service proxy component type.
- `name(String)` matches component ports with the given port name.
- `names(Collection)` matches component ports with the given port names. The order of the names is not relevant in any kind.
- `name(Pattern)` matches component ports with the given port name pattern.
- `componentPortName(String)` matches component ports with the given component port name.
  The component name and port name are separated by a dot, e.g. `'MySwc.MyApplicationPort'`, `'ECU Composition.MyDelegationPort'`. See also `SIComponentPort.getName()`.
- `componentPortNames(Collection)` matches component ports with the given component port names.
  The component name and port name are separated by a dot, e.g. `'MySwc.MyApplicationPort'`, `'ECU Composition.MyDelegationPort'`. See also `SIComponentPort.getName()`.
  The order of the names is not relevant in any kind.
- `asrPath(String)` matches component ports with the given port autosar path.
- `asrPath(Pattern)` matches component ports with the given port autosar path pattern.
- `component(String)` matches component ports with the given component name.
- `components(Collection)` matches component ports with the given component names. The order of the names is not relevant in any kind.
- `component(Pattern)` matches component ports with the given component name pattern.
- `componentAsrPath(String)` matches the component ports with the given component autosar path.
- `componentAsrPath(Pattern)` matches component ports with the given component autosar path pattern.
- `componentType(String)` matches component ports whose component type's name equals the given component type name.
- `componentType(Pattern)` matches component ports whose component type's name matches the given component type name pattern.
- `componentTypeAsrPath(String)` matches the component ports whose component type's autosar path equals the given component type autosar path.

- `componentTypeAsrPath(Pattern)` matches component ports whose component type's autosar path matches the given component type autosar path pattern.
- `portInterfaceMapping(String)` matches component ports for whose port interfaces a port interface mapping with the given port interface mapping name exists.
- `portInterfaceMapping(Pattern)` matches component ports for whose port interfaces a port interface mapping with the given port interface mapping name pattern exists.
- `portInterfaceMappingAsrPath(String)` matches component ports for whose port interfaces a port interface mapping with the given port interface mapping autosar path exists.
- `portInterfaceMappingAsrPath(Pattern)` matches component ports for whose port interfaces a port interface mapping with the given port interface mapping autosar path pattern exists.
- `originComponentPortName(String)` matches component ports having an origin component port with the given originPortName. That means the port has an incomplete delegation connection in the structured extract to a composition port with the given originPortName.
- `originComponentPortNames(Collection)` matches component ports having an origin component port with one of the given originPortNames. That means the port has an incomplete delegation connection in the structured extract to a composition port with one of the given originPortNames. The order of the names is not relevant in any kind.
- `originComponentPortName(Pattern)` matches component ports having an origin component port with the given origin port name pattern. That means the port has an incomplete delegation connection in the structured extract to a composition port with the given origin port name pattern.
- `originComponentPortComponent(String)` matches component ports having an origin component port with the given originComponentName. That means the port has an incomplete delegation connection in the structured extract to a composition port whose composition owner instance has the given originComponentName.
- `originComponentPortComponent(Pattern)` matches component ports having an origin component port with the given origin component name pattern. That means the port has an incomplete delegation connection in the structured extract to a composition port whose composition owner instance has the given origin component name pattern.
- `hasInnerTopLevelDelegationOriginComponentPort()` matches component ports which have an inner top level origin component port. That means the port in the structured extract has an incomplete delegation connection which ends at a composition that is instantiated directly inside the top level composition (ECU Composition).
- `diagnosticConnection()` narrows down selection to component ports which are derived from diagnostic mappings. See `IComponentPortSelector.hasDiagnosticConnection()` for more details.

  `diagnosticConnection()` cannot be combined with `and(Runnable)`, `or(Runnable)` and `not(Runnable)`.
  If possible always prefer using `diagnosticConnection()` over `hasDiagnosticConnection()` which can be also combined with `not(Runnable)`, `and(Runnable)` and `or(Runnable)` due to performance reasons.
- `hasDiagnosticConnection()` matches component ports for which a corresponding diagnostic event port mapping, diagnostic FiM function mapping, diagnostic service data mapping

or a diagnostic service sw mapping exists so that a connection to another port can be derived from this mapping.

- `diagnosticPortRole(EDiagnosticPortRole)` matches component ports with the given `EDiagnosticPortRole`.
- `filterAdvanced(Predicate)` matches component ports for which the given predicate results to true.
- `and(Runnable)` combines the predicates inside the lambda with a logical AND.
- `or(Runnable)` combines the predicates inside the lambda with a logical OR.
- `not(Runnable)` negates the combination of predicates inside the lambda.
- `put(List)` can be used to set `SIComponentPort`s into the selection. This is the most efficient way to create a selection from existing objects. If further predicates are specified the predicates will be applied only on the component ports that were given into this `put(List)` method. The iteration order is relevant for getting deterministic results on further usage of the selection API. This method should only be called once.

Examples

```
scriptTask("selectAllPorts", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def selectedPorts =
             selectComponentPorts {
                // no predicates: select ALL component ports
             } getComponentPorts()
        scriptLogger.info("Selected {0} component ports.", selectedPorts.size())
      }
    }
  }
}
```

Listing 6.217: Selects all component ports

```
scriptTask("selectAllUnconnectedPorts", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def selectedPorts =
             selectComponentPorts {
                unconnected() // select all unconnected component ports
             } getComponentPorts()
        scriptLogger.info("Selected {0} component ports.", selectedPorts.size())
      }
    }
  }
}
```

Listing 6.218: Selects all unconnected component ports

```
scriptTask("selectAllUnconnectedSRAndConnectedModePorts", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def selectedPorts =
             selectComponentPorts {
                // start with logical OR
                or {
                 and { // unconnected sender/receiver ports
                    unconnected()
                    senderReceiver()
                 }
                 and { // connected modeSwitch ports
                    connected()
                    modeSwitch()
                 }
                }
             } getComponentPorts()
        scriptLogger.info("Selected {0} component ports.", selectedPorts.size())
      }
    }
  }
}
```

Listing 6.219: Select all unconnected sender/receiver or connected mode-switch component ports

```
scriptTask("selectNotCompletedPorts", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def selectedPorts =
             selectComponentPorts {
                // this predicate filters for ports
                // which needs to be connected and/or a data mapping
                notCompleted()
             } getComponentPorts()
        scriptLogger.info("Selected {0} component ports.", selectedPorts.size())
      }
    }
  }
}
```

Listing 6.220: Selects not completed component ports

```
scriptTask ("selectComponentPortsUsingOriginContextPredicates", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {

                // we want to use information from the structured extract to
                    select flat extract ports

                def selectedComponentPorts = selectComponentPorts {
                    // use component port related predicates
                    senderReceiver()
                    required()

                    // combine them with origin context related predicates
                    // but remember that we are still selecting flat extract ports
                        here
                    // we only use the origin context ports as additional criteria

                    // we want only ports for which the connection ends at the
                        highest composition level inside the top level composition
                    // of the structured extract
                    hasInnerTopLevelDelegationOriginComponentPort()

                    // and only ports whose origin context has a special name
                        pattern for their composition owner instance
                    originComponentPortComponent(~"Origin.*")
                }.getComponentPorts()

                scriptLogger.info("Selected {0} component ports using their origin
                    context as additional selection criteria.",
                    selectedComponentPorts.size())
            }
        }
    }
}
```

Listing 6.221: Use origin context predicates for selecting component ports

6.10.4.2 Signal Instance Selection

The system signals and system signal groups to be data-mapped are represented by a signal instance (`SIAbstractSignalInstance`). `SISignalInstance` represents a system signal, `SISignalGroupInstance` represents a system signal group. 'Signal instance' means that the system signal or system signal group is at least referenced by one ISignal or ISignalGroup. System signals or system signal groups which are not referenced by an ISignal or ISignalGroup are not represented as signal instance and so are not available for data mapping.

`selectSignalInstances(Action)` allows the selection of `SIAbstractSignalInstance`s using predicates.

The signal instance selection can be used to select and filter signal instances and either do further operations on them, such as map them to communication elements or to just return a list of signal instances with which you can continue working.

`getSignalInstances()` allows access to the single signal instances in the `ISignalInstanceSelection`.

Signal Instance Predicates To select signal instances predicates can be provided to narrow down the result.

Per default the predicates are combined via logical AND. To realize other combinations, use the 'or','not' and 'and' predicates.

- `unmapped()` matches signal instances which are not data-mapped.
- `mapped()` matches signal instances which are data-mapped.
- `signalGroup()` matches signal instances which are a signal group instance.
- `groupSignal()` matches signal instances which are a group signal.
- `transformed()` matches signal instances which are transformation signals.
- `tx()` matches signal instances whose direction is compatible to `EDirection.Tx`.
- `rx()` matches signal instances whose direction is compatible to `EDirection.Rx`.
- `name(String)` matches signal instances with the given name.
- `names(Collection)` matches signal instances with the given names. The order of the names is not relevant in any kind.
- `name(Pattern)` matches signal instances with the given name pattern.
- `asrPath(String)` matches signal instances with the given autosar path.
- `asrPaths(Collection)` matches signal instances with the given autosar paths. The order of the names is not relevant in any kind.
- `asrPath(Pattern)` matches signal instances with the given autosar path pattern.
- `iSignal(String)` matches signal instances which are referenced at least by one ISignal/ISignalGroup with the given name.
- `iSignal(Pattern)` matches signal instances which are referenced at least by one ISignal/ISignalGroup with the given name pattern.
- `iSignalAsrPath(String)` matches signal instances which are referenced at least by one ISignal/ISignalGroup with the given autosar path.
- `iSignalAsrPath(Pattern)` matches signal instances which are referenced at least by one ISignal/ISignalGroup with the given autosar path pattern.
- `physicalChannel(String)` matches signal instances which are referenced by at least an ISignal/ISignalGroup for which an ISignalTriggering exists for a PhysicalChannel with the given name.
- `physicalChannel(Pattern)` matches signal instances which are referenced by at least an ISignal/ISignalGroup for which an ISignalTriggering exists for a PhysicalChannel with the given name pattern.
- `physicalChannelAsrPath(String)` matches signal instances which are referenced by at least an ISignal/ISignalGroup for which an ISignalTriggering exists for a PhysicalChannel with the given autosar path.
- `physicalChannelAsrPath(Pattern)` matches signal instances which are referenced by at least an ISignal/ISignalGroup for which an ISignalTriggering exists for a PhysicalChannel with the given autosar path pattern.

- `communicationCluster(String)` matches signal instances which are referenced by at least an ISignal/ISignalGroup which is sent via a PhysicalChannel of a CommunicationCluster with the given name.
- `communicationCluster(Pattern)` matches signal instances which are referenced by at least an ISignal/ISignalGroup which is sent via a PhysicalChannel of a CommunicationCluster with the given name pattern.
- `communicationClusterAsrPath(String)` matches signal instances which are referenced by at least an ISignal/ISignalGroup which is sent via a PhysicalChannel of a CommunicationCluster with the given autosar path.
- `communicationClusterAsrPath(Pattern)` matches signal instances which are referenced by at least an ISignal/ISignalGroup which is sent via a PhysicalChannel of a CommunicationCluster with the given autosar path pattern.
- `pdu(String)` matches signal instances which are referenced by at least an ISignal/ISignalGroup for which an ISignalToIPduMapping exists for a Pdu with the given name.
- `pdu(Pattern)` matches signal instances which are referenced by at least an ISignal/ISignalGroup for which an ISignalToIPduMapping exists for a Pdu with the given name pattern.
- `pduAsrPath(String)` matches signal instances which are referenced by at least an ISignal/ISignalGroup for which an ISignalToIPduMapping exists for a Pdu with the given autosar path.
- `pduAsrPath(Pattern)` matches signal instances which are referenced by at least an ISignal/ISignalGroup for which an ISignalToIPduMapping exists for a Pdu with the given autosar path pattern.
- `frame(String)` matches signal instances which are referenced by at least an ISignal/ISignalGroup which is sent via a Pdu for that a PduToFrameMapping exists for a Frame with the given name.
- `frame(Pattern)` matches signal instances which are referenced by at least an ISignal/ISignalGroup which is sent via a Pdu for that a PduToFrameMapping exists for a Frame with the given name pattern.
- `frameAsrPath(String)` matches signal instances which are referenced by at least an ISignal/ISignalGroup which is sent via a Pdu for that a PduToFrameMapping exists for a Frame with the given autosar path.
- `frameAsrPath(Pattern)` matches signal instances which are referenced by at least an ISignal/ISignalGroup which is sent via a Pdu for that a PduToFrameMapping exists for a Frame with the given autosar path pattern.
- `filterAdvanced(Predicate)` matches signal instances for which the given lambda results to true.
- `and(Runnable)` combines the predicates inside the lambda with a logical AND.
- `or(Runnable)` combines the predicates inside the lambda with a logical OR.
- `not(Runnable)` negates the combination of predicates inside the lambda.
- `put(List)` can be used to set `SIAbstractSignalInstance`s into the selection. This is the most efficient way to create a selection from existing objects. If further predicates are specified the predicates will be applied only on the signal instances that were given into this `put(List)`

method. The iteration order is relevant for getting deterministic results on further usage of the selection API. This method should only be called once.

Examples

```
scriptTask("SelectAllUnmappedSignalInstances", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def signalInstances =
                  selectSignalInstances {
                    unmapped() // select all signal instances which are not yet
                        data mapped
                  } getSignalInstances()
        scriptLogger.info("Selected {0} signal instances.",signalInstances.size())
      }
    }
  }
}
```

Listing 6.222: Select all unmapped signal instances

```
scriptTask("SelectAllUnmappedRxOrTransformedSignalInstances", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def signalInstances =
                  selectSignalInstances {
                    // the signal instances should not be data-mapped yet
                    unmapped()
                    or { // and should either be a rx signal or a transformation
                        signal
                      rx()
                      transformed()
                    }
                  } getSignalInstances()
        scriptLogger.info("Selected {0} signal instances.",signalInstances.size())
      }
    }
  }
}
```

Listing 6.223: Select all unmapped rx or transformed signal instances

```
import com.vector.cfg.sysdesc.model.communication.instance.
    SIAbstractSignalInstance
scriptTask("SelectSignalInstancesUsingAdvancedFilter", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def signalInstances =
                  selectSignalInstances {
                    filterAdvanced { SIAbstractSignalInstance signalInstance ->
                           // implement own custom filter
                           def mdfObject = signalInstance.getMdfObject()
                           // work on directly on autosar model level ...
                           // select signal instance only which has admin data
                           def select = mdfObject.adminData != null
                           select
                    }
                  } getSignalInstances()
        scriptLogger.info("Selected {0} signal instances.",signalInstances.size())
      }
    }
  }
}
```

Listing 6.224: Select signal instances using an advanced filter

6.10.4.3 Communication Element Selection

A data element, an operation or a trigger to be data-mapped is represented by an `SICommunicationElement`. A data element is represented by the subtype `SIDataCommunicationElement`, an operation is represented by the subtype `SIOperationCommunicationElement` and a trigger is represented by the subtype `SITriggerCommunicationElement`. A communication element contains the full context information (component prototype, port prototype, data type hierarchy) necessary for data mapping.

`selectCommunicationElements(Action)` allows the selection of `SICommunicationElements` using predicates.

The communication element selection can be used to select and filter communication elements and either do further operations on them, such as map them to signal instances or to just return a list of communication elements with which you can continue working.

`getCommunicationElements()` allows access to the single communication elements in the `ICommunicationElementSelection`. Please make sure you are using the correct expansion mode, see `ICommunicationElementSelector.selectFullyExpanded()` and `ICommunicationElementSelector.selectFullyExpandedButPrimitiveArraysAsLeafs()`.

Communication Element Predicates To select communication elements predicates can be provided to narrow down the result.

Per default the predicates are combined via logical AND. To realize other combinations, use the 'or','not' and 'and' predicates.

- `unconnected()` matches communication elements whose component port is unconnected.
- `connected()` matches communication elements whose component port is connected.
- `senderReceiver()` matches communication elements whose port has a sender/receiver port interface.

- `clientServer()` matches communication elements whose port has a client/server port interface.
- `trigger()` matches communication elements whose port has a trigger port interface.
- `provided()` matches communication elements whose port is a provided port (p-port).
- `required()` matches communication elements whose port is a required port (r-port).
- `delegation()` matches communication elements whose port is delegation port.
- `unmapped()` matches communication elements whose are not data-mapped.
- `mapped()` matches communication elements whose are data-mapped.
- `ownerPortTerminated()` matches communication elements whose component port is terminated.
- `ownerPortNotTerminated()` matches communication elements whose component port is not terminated.
- `name(String)` matches communication elements with the given data element or operation name.
- `names(Collection)` matches communication elements with the given data element or operation names. The order of the names is not relevant in any kind.
- `name(Pattern)` matches communication elements with the given data element or operation name pattern.
- `fullyQualifiedName(String)` matches communication elements with the given full qualified communication element name. E.g. 'App1.Port1.DataElement1', 'ECU Composition.DelegationPort2.Data
  See also `SICommunicationElement.getFullyQualifiedName()`.
- `fullyQualifiedNames(Collection)` matches communication elements with the given full qualified communication element names. E.g. 'App1.Port1.DataElement1', 'ECU Composition.DelegationPort2.DataElement2'. See also `SICommunicationElement.getFullyQualifiedName()`. The order of the names is not relevant in any kind.
- `asrPath(String)` matches communication elements with the given data element or operation autosar path.
- `asrPath(Pattern)` matches communication elements with the given data element or operation autosar path pattern.
- `component(String)` matches communication elements with the given component name.
- `components(Collection)` matches communication elements with the given component names. The order of the names is not relevant in any kind.
- `component(Pattern)` matches communication elements with the given component name pattern.
- `componentAsrPath(String)` matches communication elements with the given component name autosar path.
- `componentAsrPath(Pattern)` matches communication elements with the given component name autosar path pattern.
- `port(String)` matches communication elements with the given component port name.

- `ports(Collection)` matches communication elements with the given component port names. The order of the names is not relevant in any kind.
- `port(Pattern)` matches communication elements with the given component port name pattern.
- `portAsrPath(String)` matches communication elements with the given component port autosar path.
- `portAsrPath(Pattern)` matches communication elements with the given component port autosar path pattern.
- `filterAdvanced(Predicate)` Add a custom predicated which matches communication elements for which the given lambda results to true.
- `and(Runnable)` combines the predicates inside the lambda with a logical AND.
- `or(Runnable)` combines the predicates inside the lambda with a logical OR.
- `not(Runnable)` negates the combination of predicates inside the lambda.
- `put(List)` can be used to set `SICommunicationElements` into the selection. This is the most efficient way to create a selection from existing objects. If further predicates are specified the predicates will be applied only on the communication elements that were given into this `put(List)` method. The iteration order is relevant for getting deterministic results on further usage of the selection API. This method should only be called once.
- `selectFullyExpanded()` modifies the behavior of the selection. Using this option you are not only selecting the root communication elements (as you know from the data mapping assistant in GUI), but also the leafs. See `SIDataCommunicationElement.getLeafsFullExpanded()` for more details.
- `selectFullyExpandedButPrimitiveArraysAsLeafs()` modifies the behavior of the selection. Using this option you are not only selecting the root communication elements (as you know from the data mapping assistant in GUI), but also the leafs. Arrays of primitives (e.g. uint8 arrays) will not be expanded. See `SIDataCommunicationElement.getLeafsFullExpandedExceptPri` for more details.

Examples

```
scriptTask("SelectAllUnmappedDelPortComElements", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def comElements =
                  selectCommunicationElements {
                    // select all unmapped delegation communication elements
                    delegation()
                    unmapped()
                  } getCommunicationElements()
        scriptLogger.info("Selected {0} communication elements.",comElements.size
            ())
      }
    }
  }
}
```

Listing 6.225: Select all unmapped delegation port communication elements

```
scriptTask("SelectComplexFullyExpanded", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def comElements =
                  selectCommunicationElements {
                    // expand all complex communication elements except primitive
                        arrays
                    // and select all communication elements with their leafs
                    // e.g. for a record we select both here, the record and its
                        record elements
                    selectFullyExpandedButPrimitiveArraysAsLeafs()
                  } getCommunicationElements()
        scriptLogger.info("Selected {0} communication elements.",comElements.size
            ())
      }
    }
  }
}
```

Listing 6.226: Select all communication elements with their leafs

```
import com.vector.cfg.sysdesc.model.communication.SICommunicationElement
import com.vector.cfg.sysdesc.model.communication.SIDataCommunicationElement
scriptTask("SelectComElementsUsingAdvancedFilter", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def comElements =
             selectCommunicationElements {
               // advanced filter:
               // only select communication elements
               // which represent data elements of a specific data type
               filterAdvanced { SICommunicationElement comElement ->
                   if (comElement instanceof SIDataCommunicationElement) {
                      def mdfDataElement = comElement.getTargetElement().
                          getMdfObject()
                      // check directly on autosar model level
                      return mdfDataElement.type.refTarget.name.equals("
                          myCustomDataType")
                   }
                   false
               }
             } getCommunicationElements()
        scriptLogger.info("Selected {0} communication elements.",comElements.size
            ())
      }
    }
  }
}
```

Listing 6.227: Select communication elements using an advanced filter

6.10.4.4 Component Type Selection

An `SIComponentType` represents a software component type according to AUTOSAR. It might define functionality and behavior of a software component in case of an atomic component type, contain other software components and connections in case of a composition component type or define parameters and characteristic values in case of a parameter component type.

`SIComponent` is an instance of a software component type.

`selectComponentTypes(Action)` allows the selection of `SIComponentTypes` using predicates.

The component type selection can be used to select and filter component types and either do further operations on them, such as instantiating them by creating new component prototypes or to just return a list of component types with which you can continue working.

`getComponentTypes()` allows access to the single component types in the `IComponentTypeSelection`.

Component Type Predicates To select the component types predicates can be provided to narrow down the result.

Per default the predicates are combined via logical AND. To realize other combinations, use the 'or','not' and 'and' predicates.

- `name(String)` matches component types with the given component type name.
- `names(Collection)` matches component types with the given component type names. The order of the names is not relevant in any kind.
- `name(Pattern)` matches component types with the given component type name pattern.
- `asrPath(String)` matches component types with the given componet type autosar path.
- `asrPath(Pattern)` matches component types with the given component type autosar path pattern.
- `component(String)` matches component types for which a component prototype with the given component name exists.
- `component(Pattern)` matches component types for which a component prototype with the given component name pattern exists.
- `application()` matches component types which are application component types. Application component types are all component types which are not service component types, as displayed in the ECU Software Components Editor, not ApplicationSwComponentTypes as defined by AUTOSAR.
- `service()` matches component types which are service component types.
- `parameter()` matches component types which are parameter (calibration) component types.
- `nvBlock()` matches component types which are nv block component types.
- `sensorActuator()` matches component types which are sensor actuator component types.
- `ioHwAbstraction()` matches component types which are I/O hardware abstraction component types, also called EcuAbstractionSwComponentType.
- `complexDeviceDriver()` matches component types which are complex device driver component types.
- `serviceProxy()` matches component types which are service proxy component types.
- `instantiated()` matches component types that are already instantiated. In other words matches if a component prototype of that component type already exists.
- `supportsMultipleInstantiation()` matches component types which support multiple instantiation.

- `filterAdvanced(Predicate)` matches component types for which the given lambda results to true.
- `and(Runnable)` combines the predicates inside the lambda with a logical AND.
- `or(Runnable)` combines the predicates inside the lambda with a logical OR.
- `not(Runnable)` negates the combination of predicates inside the lambda.
- `put(List)` can be used to set `SIComponentTypes` into the selection. This is the most efficient way to create a selection from existing objects. If further predicates are specified the predicates will be applied only on the component types that were given into this `put(List)` method. The iteration order is relevant for getting deterministic results on further usage of the selection API. This method should only be called once.

Examples

```
scriptTask ("selectComponentTypeByName", DV_PROJECT ){
    code {
        domain.runtimeSystem {
            def selectedComponentTypes = selectComponentTypes {
                name "App1"
            }.getComponentTypes()

            scriptLogger.info("Selected '{0}' component types.",
                selectedComponentTypes.size())
        }
    }
}
```

Listing 6.228: Select component type by name

```
scriptTask ("selectNotInstantiatedComponentTypes", DV_PROJECT ){
    code {
        domain.runtimeSystem {
            def selectedComponentTypes = selectComponentTypes {
                not {
                    instantiated()
                }
            }.getComponentTypes()

            scriptLogger.info("Selected '{0}' component types.",
                selectedComponentTypes.size())
        }
    }
}
```

Listing 6.229: Select not instantiated component types

6.10.4.5 Event Selection

An event `SIEvent` (called AbstractEvent in AUTOSAR) represents a RTEEvent or a BswEvent. Events are raised on different conditions and are used to implement application or basic software in AUTOSAR. (Sometimes they are also called triggers.)

A task mapping (`SITaskMapping`) represents an `SIEvent` (also called trigger) that is mapped to a task in the context of a component prototype or a module configuration. It corresponds to the task mapping container in the RTE configuration.

`selectEvents(Action)` allows the selection of `SIEvents` using predicates.

The event selection can be used to select and filter events and either do further operations on them, such as mapping the executable entities they trigger to tasks or to just return a list of events or the task mappings for them with which you can continue working.

`getEvents()` allows access to the single events in the `IEventSelection`.

`getTaskMappings()` retrieves all `SITaskMapping`s for the selected events (see `getEvents()`).

Note:
1. In case of multi instantiation of component prototypes, the different instances share the same events, since the event is part of the internal behavior of the component type. Therefore if the event is selected, `getTaskMappings()` will always return the task mappings for all component prototypes.
2. Since this method can be run outside of a transaction, there might be selected events for which no task mapping container does exist yet. The container cannot be created by calling `getTaskMappings()`, so no task mapping can be returned. This happens if the system description is not synchronized, after changes in the structured extract were done (see Automation Interface Documentation, chapter about Model Synchronization for examples how to synchronize).

Event Predicates To select the events predicates can be provided to narrow down the result.

Per default the predicates are combined via logical AND. To realize other combinations, use the 'or','not' and 'and' predicates.

- `name(String)` matches events (triggers) with the given event name.
- `names(Collection)` matches events (triggers) with the given event names. The order of the names is not relevant in any kind.
- `name(Pattern)` matches events (triggers) with the given event name pattern.
- `asrPath(String)` matches events (triggers) with the given event autosar path.
- `asrPath(Pattern)` matches events (triggers) with the given event autosar path pattern.
- `applicationComponent()` matches events (triggers) which belong to an application component.
- `serviceComponent()` matches events (triggers) which belong to a service component.
- `component(String)` matches events (triggers) which belong to components with the given component name.
- `components(Collection)` matches events (triggers) which belong to components with the given component names. The order of the names is not relevant in any kind.
- `component(Pattern)` matches events (triggers) which belong to components which matches the given component name pattern.
- `componentType(String)` matches events (triggers) which are part of the internal behavior of component types with the given component type name.
- `componentType(Pattern)` matches events (triggers) which are part of the internal behavior of component types which matches the given component type name pattern.
- `componentTypeAsrPath(String)` matches events (triggers) which are part of the internal behavior of component types with the given component type autosar path.

- `componentTypeAsrPath(Pattern)` matches events (triggers) which are part of the internal behavior of component types whose autosar path matches the given component type autosar path pattern.
- `moduleConfiguration(String)` matches events (triggers) which belong to module configurations with the given module configuration name.
- `moduleConfigurations(Collection)` matches events (triggers) which belong to module configurations with the given module configuration names. The order of the names is not relevant in any kind.
- `moduleConfiguration(Pattern)` matches events (triggers) which belong to module configurations which matches the given module configuration name pattern.
- `moduleConfigurationAsrPath(String)` matches events (triggers) which belong to module configurations with the given module configuration autosar path.
- `moduleConfigurationAsrPath(Pattern)` matches events (triggers) which belong to module configurations whose autosar path matches the given module configuration autosar path pattern.
- `task(String)` matches events (triggers) which are mapped to a task with the given task name.
- `task(Pattern)` matches events (triggers) which are mapped to a task whose name matches the given task name pattern.
- `bswEvent()` matches events (triggers) which are bsw events.
- `rteEvent()` matches events (triggers) which are rte events.
- `unmapped()` matches unmapped events (triggers). In case of multi instantiated components/-modules matches if unmapped at least in one context. Use `fullyUnmapped()` to determine whether an event is unmapped in all contexts.
- `fullyUnmapped()` matches events (triggers) which are not mapped in any context. If no multi instantiation is used, the result is the same as for `unmapped()`.
- `mapped()` matches mapped events (triggers). In case of multi instantiated components/modules matches if mapped at least in one context. Use `fullyMapped()` to determine whether an event is mapped in all contexts.
- `fullyMapped()` matches events (triggers) which are mapped in every context. If no multi instantiation is used, the result is the same as for `mapped()`.
- `timing()` matches events which are timing events (triggers).
- `timing(Double)` matches events (triggers) which are timing events with the given period (seconds).
- `init()` matches events (triggers) which are init events.
- `dataReceived()` matches events (triggers) which are data received events.
- `dataReceiveError()` matches events (triggers) which are data receive error events.
- `dataSendCompleted()` matches events (triggers) which are data send completed events.
- `dataWriteCompleted()` matches events (triggers) which are data write completed events.
- `operationInvoked()` matches events (triggers) which are operation invoked events.

- `operationInvoked(String)` matches operation invoked events (triggers) which are invoked by an operation with the given operationName.
- `serverCallReturns()` matches events (triggers) which are asynchronous server call returns events.
- `modeSwitch()` matches events (triggers) which are mode switch events.
- `modeEntry()` matches events (triggers) which are mode switch events with activation kind ON-ENTRY.
- `modeExit()` matches events (triggers) which are mode switch events with activation kind ON-EXIT.
- `modeTransition()` matches events (triggers) which are mode switch events with activation kind ON-TRANSITION.
- `modeSwitchedAck()` matches events (triggers) which are mode switched acknowledgement events.
- `externalTrigger()` matches events (triggers) which are external trigger occurred events.
- `internalTrigger()` matches events (triggers) which are internal trigger occurred events.
- `background()` matches events (triggers) which are background events.
- `transformerHardError()` matches events (triggers) which are transformer hard error events.

- `mandatory()` matches events (triggers) which must be mapped. (The mapping of operation invoked events and bsw events whose schedulable entity has no via symbol matching runnable is optional.)
- `filterAdvanced(Predicate)` matches events (triggers) for which the given lambda results to true.
- `and(Runnable)` combines the predicates inside the lambda with a logical AND.
- `or(Runnable)` combines the predicates inside the lambda with a logical OR.
- `not(Runnable)` negates the combination of predicates inside the lambda.
- `put(List)` can be used to set `SIEvents` into the selection. This is the most efficient way to create a selection from existing objects. If further predicates are specified the predicates will be applied only on the events that were given into this `put(List)` method. The iteration order is relevant for getting deterministic results on further usage of the selection API. This method should only be called once.

Examples

```
scriptTask ("selectEvents", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def selectedEvents = selectEvents {
                    // select all unmapped events of component 'App1'
                    unmapped()
                    component("App1")
                }.getEvents()

                scriptLogger.info("Selected '{0}' events.", selectedEvents.size())
            }
        }
    }
}
```

Listing 6.230: Select events example

6.10.4.6 Executable Entity Selection

An executable entity (`SIExecutableEntity`) represents a RunnableEntity or a BswSchedulableEntity. Both are abstractions of executable code in AUTOSAR. (Sometimes they are also called functions.)

A task mapping (`SITaskMapping`) represents an `SIEvent` (also called trigger) that is mapped to a task in the context of a component prototype or a module configuration. It corresponds to the task mapping container in the RTE configuration.

`selectExecutableEntities(Action)` allows the selection of `SIExecutableEntity`s using predicates.

The executable entity selection can be used to select and filter executable entities and either do further operations on them, such as mapping them to tasks or to just return a list of executable entities or the task mappings for them with which you can continue working.

`getExecutableEntities()` allows access to the single executable entities in the `IExecutableEntitySelection`.

`getTaskMappings()` retrieves all `SITaskMapping`s for the selected executable entities (see `getExecutableEntities()`).

Executable Entity Predicates To select the executable entities predicates can be provided to narrow down the result.

Per default the predicates are combined via logical AND. To realize other combinations, use the 'or','not' and 'and' predicates.

- `symbol(String)` matches runnable entities with the given symbol and bsw schedulable entities whose corresponding bsw module entry short name matches the given symbol.
- `symbol(Pattern)` matches runnable entities whose symbol matches the given symbol pattern and bsw schedulable entities whose corresponding bsw module entry short name matches the given symbol pattern.
- `name(String)` matches executable entities (functions) with the given name.

- `names(Collection)` matches executable entities (functions) with the given names. The order of the names is not relevant in any kind.
- `name(Pattern)` matches executable entities (functions) with the given name pattern.
- `asrPath(String)` matches executable entities (functions) with the given autosar path.
- `asrPath(Pattern)` matches executable entities (functions) with the given autosar path pattern.
- `applicationComponent()` matches executable entities (functions) whose owner is an application component.
- `serviceComponent()` matches executable entities (functions) whose owner is a service component.
- `component(String)` matches executable entities (functions) which belong to components with the given component name.
- `components(Collection)` matches executable entities (functions) which belong to components with the given component names. The order of the names is not relevant in any kind.
- `component(Pattern)` matches executable entities (functions) which belong to components which matches the given component name pattern.
- `componentType(String)` matches executable entities (functions) which are part of the internal behavior of component types with the given component type name.
- `componentType(Pattern)` matches exectuable entities (functions) which are part of the internal behavior of component types which matches the given component type name pattern.
- `componentTypeAsrPath(String)` matches executable entities (functions) which are part of the internal behavior of component types with the given component type autosar path.
- `componentTypeAsrPath(Pattern)` matches executable entities (functions) which are part of the internal behavior of component types whose autosar path matches the given component type autosar path pattern.
- `moduleConfiguration(String)` matches executable entities (functions) which belong to module configurations with the given module configuration name.
- `moduleConfigurations(Collection)` matches executable entities (functions) which belong to module configurations with the given module configuration names. The order of the names is not relevant in any kind.
- `moduleConfiguration(Pattern)` matches executable entities (functions) which belong to module configurations which matches the given module configuration name pattern.
- `moduleConfigurationAsrPath(String)` matches executable entities (functions) which belong to module configurations with the given module configuration autosar path.
- `moduleConfigurationAsrPath(Pattern)` matches executable entities (functions) which belong to module configurations whose autosar path matches the given module configuration autosar path pattern.
- `task(String)` matches executable entities (functions) which have at least one event (trigger) that is mapped to a task with the given task name.
- `task(Pattern)` matches executable entities (functions) which have at least one event (trigger) that is mapped to a task whose name matches the given task name pattern.

- `bswSchedulableEntity()` matches executable entities (functions) which are bsw schedulable entities.
- `runnableEntity()` matches executable entities (functions) which are runnable entities.
- `unmapped()` matches executable entities (functions) with at least one unmapped event (trigger).
- `fullyUnmapped()` matches executable entities (functions) with all of its events (triggers) being not mapped in any context to a task.
- `mapped()` matches executable entities (functions) with at least one mapped event (trigger).
- `fullyMapped()` matches executable entities (functions) with all of its events (triggers) being mapped in each context to a task.
- `filterAdvanced(Predicate)` matches executable entities (functions) for which the given predicate results to true.
- `and(Runnable)` combines the predicates inside the lambda with a logical AND.
- `or(Runnable)` combines the predicates inside the lambda with a logical OR.
- `not(Runnable)` negates the combination of predicates inside the lambda.
- `put(List)` can be used to set `SIExecutableEntitys` into the selection. This is the most efficient way to create a selection from existing objects. If further predicates are specified the predicates will be applied only on the executable entities that were given into this `put(List)` method. The iteration order is relevant for getting deterministic results on further usage of the selection API. This method should only be called once.

Examples

```
scriptTask ("selectExecutableEntities", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def selectedExecutables = selectExecutableEntities {
                    // select all runnables with symbol 'MySymbol'
                    symbol("MySymbol")
                    runnableEntity()
                }.getExecutableEntities()

                scriptLogger.info("Selected '{0}' executable entities.",
                    selectedExecutables.size())
            }
        }
    }
}
```

Listing 6.231: Select executable entities example

6.10.4.7 Port Interface Selection

A `SIPortInterface` represents a port interface according to AUTOSAR. A port interface is an interface that is either provided or required by a port of a software component.

`selectPortInterfaces(Action)` allows the selection of `SIPortInterfaces` using predicates.

The port interface selection can be used to select and filter port interfaces and either do further operations on them, such as instantiating them by creating new delegation port prototypes or to just return a list of port interfaces with which you can continue working.

`getPortInterfaces()` allows access to the single port interfaces in the `IPortInterfaceSelection`.

Port Interfaces Predicates To select the port interfaces predicates can be provided to narrow down the result.

Per default the predicates are combined via logical AND. To realize other combinations, use the 'or','not' and 'and' predicates.

- `name(String)` matches port interfaces with the given port interface name.
- `names(Collection)` matches port interfaces with the given port interface names. The order of the names is not relevant in any kind.
- `name(Pattern)` matches port interfaces with the given port interface name pattern.
- `asrPath(String)` matches port interfaces with the given port interface autosar path.
- `asrPath(Pattern)` matches port interfaces with the given port interface autosar path pattern.
- `service()` matches port interfaces which are service interfaces.
- `application()` matches port interfaces which are application interfaces.
- `senderReceiver()` matches port interfaces which are sender receiver interfaces.
- `clientServer()` matches port interfaces which are client server interfaces.
- `modeSwitch()` matches port interfaces which are mode switch interfaces.
- `nvData()` matches port interfaces which are NvData interfaces.
- `trigger()` matches port interfaces which are trigger interfaces.
- `parameter()` matches port interfaces which are parameter interfaces.
- `componentType(String)` first matches all component types with the given component type name, then retrieves all port interfaces of the component type's port prototypes.
- `componentType(Pattern)` first matches all component types with the given component type name pattern, then retrieves all port interfaces of the component type's port prototypes.
- `componentTypeAsrPath(String)` first matches all component types with the given component type asr path, then retrieves all port interfaces of the component type's port prototypes.
- `componentTypeAsrPath(Pattern)` first matches all component types with the given component type asr path pattern, then retrieves all port interfaces of the component type's port prototypes.
- `component(String)` first matches all components with the given component name, then retrieves all port interfaces of the component's ports.
- `components(Collection)` first matches all components with the given component names, then retrieves all port interfaces of the component's ports. The order of the names is not relevant in any kind.

- `component(Pattern)` first matches all components with the given component name pattern, then retrieves all port interfaces of the component's ports.
- `componentPort(String)` first matches all `SIComponentPorts` with the given name, then retrieves all port interfaces of the component ports.
- `componentPorts(Collection)` first matches all `SIComponentPorts` with the given names, then retrieves all port interfaces of the component ports. The order of the names is not relevant in any kind.
- `componentPort(Pattern)` first matches all `SIComponentPorts` with the given name pattern, then retrieves all port interfaces of the component ports.
- `filterAdvanced(Predicate)` matches port interfaces for which the given lambda results to true.
- `and(Runnable)` combines the predicates inside the runnable with a logical AND.
- `or(Runnable)` combines the predicates inside the lambda with a logical OR.
- `not(Runnable)` negates the combination of predicates inside the lambda.
- `put(List)` can be used to set `SIPortInterfaces` into the selection. This is the most efficient way to create a selection from existing objects. If further predicates are specified the predicates will be applied only on the port interfaces that were given into this `put(List)` method. The iteration order is relevant for getting deterministic results on further usage of the selection API. This method should only be called once.

Examples

```
scriptTask("selectPortInterfacesByName", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def selectedPortInterfaces =
             selectPortInterfaces {
                // selects all sender receiver application port interfaces with
                    the name 'MyPortInterface'
                senderReceiver()
                application()
                name "MyPortInterface"

             // getPortInterfaces() will filter all port interfaces for the given
                 predicates
             // so in our example we will receive
             // all sender receiver application port interfaces with the short
                 name 'MyPortInterface'
             } getPortInterfaces()
        scriptLogger.info("Selected {0} port interfaces.", selectedPortInterfaces.
            size())
      }
    }
  }
}
```

Listing 6.232: Select PortInterface by name and type

```
scriptTask("selectPortInterfacesByComponentPorts", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def selectedPortInterfaces =
             selectPortInterfaces {
                // selects the port interface of the port 'pCSPort1' of component
                    'App1'
                componentPort "App1.pCSPort1"
             } getPortInterfaces()
        scriptLogger.info("Selected {0} port interfaces.", selectedPortInterfaces.
            size())
      }
    }
  }
}
```

Listing 6.233: Select PortInterfaces by component ports

6.10.4.8 Origin Component Port Selection

Origin Context According to AUTOSAR the flat extract is created out of the structured extract. During the flattening process, the inner compositions and their ports are lost. However sometimes the information about this objects is helpful to perform actions on the flat extract, such as for example connecting ports or doing the data mapping. We call the objects of the structured extract, which are related to the flat extract objects, origin contexts.

The component port connection provides an option to use the origin context's names as additional mapping criteria. This will be introduced below (see 6.10.4.9 on page 206).

Remark: Since the RuntimeSystem-API now works on a flat-view of the StructuredExtract, the origin-context-APIs are doing the same, but behave as original designed for flat extract usage.

Origin Component Port There is an own model element for the component ports of the structured extract. A component port (see also `SIComponentPort`) represents a port prototype and its corresponding component prototype. The `SIOriginComponentPort` represents a port in context of a component prototype for the upstream model (structured extract).

Remark:

The original design of the origin component ports was done for the flattened components of the structured extract. Since system description SIModel allows working directly on the structured extract and provides the `ISysDescService.getFlatComponentView()` as view of the flattened components (which is used in the RuntimeSystem-APIs), the SIOriginComponentPorts now represent the SIComponentPorts of the structured extract directly and are interface compatible to the original design, but do not work on the flat extract anymore.

Further we want to clarify some of the terminology which is used in context of the origin component port. The term delegation port is pretty clear for the flat extract, since there are no other compositions beside the top level composition. However it is possible to instantiate also compositions inside the top level composition and inside other compositions in the structured extract. We call all ports whose owner is a composition, delegation ports.

Another term used here is the inner top level. Since a project can have only one top level composition, the first hierarchy level defining the rough structure of the software components is directly

inside the top level composition. So everything directly inside the top level composition is called the inner top level.

For example if a composition 'MyComposition' is instantiated directly in the top level composition and has a port named 'SendData', so we call the origin component port 'MyComposition.SendData' an inner top level delegation port. Let's assume the composition named 'OtherComposition' has a port named 'ReceiveData' and is instantiated in ' MyComposition'. The origin component port 'OtherComposition.ReceiveData' is not an inner top level delegation port, since we use this term only for the hierarchy level inside the top level composition.

`selectOriginComponentPorts(Action)` allows the selection of `SIOriginComponentPorts` using predicates. The origin component ports which are selected here, are ends of incomplete delegation connections in the structured extract.

The origin component port selection can be used to select and filter origin component ports and either do further operations on them, such as creating new delegation ports in the flat extract for them or to just return a list of origin component ports with which you can continue working.

`getOriginComponentPorts()` allows access to the single origin component ports in the `IOrigin-ComponentPortSelection`.

Origin Component Port Predicates To select the origin component ports predicates can be provided to narrow down the result.

Per default the predicates are combined via logical AND. To realize other combinations, use the 'or','not' and 'and' predicates.

- `name(String)` matches origin component ports with the given port name.
- `names(Collection)` matches origin component ports with the given port names. The order of the names is not relevant in any kind.
- `name(Pattern)` matches origin component ports with the given port name pattern.
- `asrPath(String)` matches origin component ports with the given port autosar path.
- `asrPath(Pattern)` matches origin component ports with the given port autosar path pattern.
- `component(String)` matches origin component ports with the given component name.
- `components(Collection)` matches origin component ports with the given component names. The order of the names is not relevant in any kind.
- `component(Pattern)` matches origin component ports with the given component name pattern.
- `componentAsrPath(String)` matches the origin component ports with the given component autosar path.
- `componentAsrPath(Pattern)` matches origin component ports with the given component autosar path pattern.
- `componentType(String)` matches origin component ports whose component type's name equals the given component type name.
- `componentType(Pattern)` matches origin component ports whose component type's name matches the given component type name pattern.

- `componentTypeAsrPath(String)` matches origin component ports whose component type's autosar path equals the given component type autosar path.
- `componentTypeAsrPath(Pattern)` matches origin component ports whose component type's autosar path matches the given component type autosar path pattern.
- `provided()` matches provided origin component ports (p-port).
- `required()` matches required origin component ports (r-port).
- `providedRequired()` matches provided-required origin component ports (pr-port).
- `innerTopLevelDelegation()` matches origin component ports on the highest hierarchy level. In other words the component of the matched ports is instantiated directly inside the top level composition (ECU Composition).
- `ofFlatExtractPort(SIComponentPort)` retrieves the origin component ports of the given flatComponentPort.
- `senderReceiver()` matches origin component ports whose port has a sender/receiver port interface.
- `clientServer()` matches origin component ports whose port has a client/server port interface.
- `trigger()` matches origin component ports whose port has a trigger port interface.
- `filterAdvanced(Predicate)` matches origin component ports for which the given predicate results to true.
- `and(Runnable)` combines the predicates inside the lambda with a logical AND.
- `or(Runnable)` combines the predicates inside the lambda with a logical OR.
- `not(Runnable)` negates the combination of predicates inside the lambda.
- `completed()` matches origin component ports which are completed.

  An `SIOriginComponentPort` is completed if an only if all of its flat extract ports and additionally all of the delegation ports which are connected to these flat extract ports are completed. (See `SIComponentPort` for definition of completed state for flat extract ports.)
- `notCompleted()` matches origin component ports which are not completed.

  See `completed()` for the conditions an origin port has to meet to be a completed port.
- `put(List)` can be used to set `SIOriginComponentPorts` into the selection. This is the most efficient way to create a selection from existing objects. If further predicates are specified the predicates will be applied only on the origin component ports that were given into this `put(List)` method. The iteration order is relevant for getting deterministic results on further usage of the selection API. This method should only be called once.

Examples

```
scriptTask("selectOriginComponentPorts", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def selectedOriginPorts = selectOriginComponentPorts {
            innerTopLevelDelegation()
            provided()
        } getOriginComponentPorts()

        scriptLogger.info("Selected '{0}' origin component ports.",
            selectedOriginPorts.size())
     }
    }
  }
}
```

Listing 6.234: Select ends of incomplete connections

```
scriptTask("originPortsForFlatExtractPort", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        // first we need to find our flat view port
        // we will use the component port selection for this example
        def componentPort = selectComponentPorts {
            name "pDataSend"
            component "App1"
        } getComponentPorts().iterator().next()

        def selectedOriginPorts = selectOriginComponentPorts {
            ofFlatExtractPort(componentPort)
        } getOriginComponentPorts()

        scriptLogger.info("Selected '{0}' origin component ports for {1}.",
            selectedOriginPorts.size(),
            componentPort.getName())
     }
    }
  }
}
```

Listing 6.235: Select the ends of incomplete connections for a specific flat view component port

6.10.4.9 Component Port Connection

This chapter is about connecting (a.k.a. mapping) component ports to other component ports. There are two ways to do that.
The first way is using a the component port selection API (see 6.10.4.1 on page 172) and calling a method to connect the selected ports to other ports. It is possible to filter the targets and to evaluate and change by the auto-mapper suggested connections. So this way is similar to the component connection assistant from the GUI.
The second way is to use a simple API that requires already prepared data structures e.g. two lists of component ports that are sorted applying certain custom rules and to map them via index matching. To initially find the appropriate component ports you can use the common selection APIs.

Auto-Mapping The use case of auto-mapping component ports is based on the selection of component ports. The auto-mapper matches component ports using their names.

`autoMap()` tries to auto-map the selection of component ports according the component connection assistant default rules.

Examples for autoMap()

```
scriptTask("automapAll", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def mappedConnectors =
                 selectComponentPorts {
                    // no predicates: select ALL component ports
                 } autoMap()
        scriptLogger.info("Created {0} mappings.", mappedConnectors.size())
      }
    }
  }
}
```

Listing 6.236: Tries to auto-map all ports

```
scriptTask("automapAllUnconnected", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def mappedConnectors =
             selectComponentPorts {
                 unconnected() // select all unconnected component ports
             } autoMap()
        scriptLogger.info("Created {0} mappings.", mappedConnectors.size())
      }
    }
 }
}
```

Listing 6.237: Tries to auto-map all unconnected component ports

```
scriptTask("autoMapUnconnectedSRCS", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def mappedConnectors =
                selectComponentPorts {
                   // select all unconnected client/server and unconnected sender/
                       receiver ports
                   unconnected()
                   or {
                     clientServer()
                     senderReceiver()
                   }
                } autoMap()
        scriptLogger.info("Created {0} mappings.",mappedConnectors.size())
      }
    }
  }
}
```

Listing 6.238: Tries to auto-map all unconnected sender/receiver and client/server ports

```
import com.vector.cfg.sysdesc.model.component.SIComponentPort

scriptTask("autoMapAdvancedfilter", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def mappedConnectors =
                selectComponentPorts {
                  // select component port by own custom filter predicate
                  filterAdvanced { SIComponentPort port ->
                       "MyUUID".equals(port.getPort().getMdfObject().getUuid2())
                  }
                } autoMap()
        scriptLogger.info("Created {0} mappings.",mappedConnectors.size())
      }
    }
  }
}
```

Listing 6.239: Tries to auto-map port determined by advanced filter

`autoMapTo(Action)` tries to auto-map the selection of component ports according the component connection assistant rules but offers more control for the auto-mapping: Inside the lambda additional predicates for narrowing down the target component ports can be defined and code to evaluate and change the auto-mapper results can be provided.

Narrowing down the target component ports may be useful to gain better matches for the auto-mapper: In case several target component ports match equally, no auto-mapping is performed. So reducing the target component ports may improve the results of the auto-mapping.

The component port selection will produce trace, info and warning logs. To see them, activate the '`IComponentPortSelection`' logger with the appropriate log level.

The provided list of connections will contain all created connections for each connected component port pair: since some connections need a connection chain through the composition hierarchy this includes `SIDelegationConnector`s as well (which are usually not visible in the flat component view of the structured extract). But it ensures completeness of the result.

Control the auto-mapping in autoMapTo(Closure)

`selectTargetPorts(Action)` allows to define predicates to narrow down the target ports for the auto-mapping. The predicates are used to filter the possible target component ports which were computed from the source component port selection.

```
scriptTask("autoMapUnconnectedToComponentPrototype", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
       def mappedConnectors =
                 selectComponentPorts {
                    unconnected() // select all unconnected ports
                 } autoMapTo {
                    selectTargetPorts {
                         component "App1" // and auto-map them to all ports of
                             component "App1"
                    }
                 }
        scriptLogger.info("Created {0} mappings.",mappedConnectors.size())
      }
    }
  }
}
```

Listing 6.240: Tries to auto map all unconnected ports to the ports of one component prototype

`evaluateMatches(IMultiAutoMappingEvaluator)` allows to evaluate and change the results of the auto-mapping. It corresponds to the confirm page of the component connection assistant.

For each source component port the provided lambda is called: Parameters are the source component port, the optional matched target component port (or null), and a list of all potential target component ports (respecting the `selectTargetPorts(Action)` predicates). The return value must be a list of target component ports.

```
import com.vector.cfg.sysdesc.model.connector.SISourceComponentPort
import com.vector.cfg.sysdesc.model.connector.SITargetComponentPort

scriptTask("automapAllUnconnectedAndEvaluateMatches", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def mappedConnectors =
             selectComponentPorts {
               unconnected()
             } autoMapTo {
                evaluateMatches {
                    SISourceComponentPort sourcePort,
                    SITargetComponentPort optionalMatchedTargetPort,
                    List<SITargetComponentPort> potentialTargetPorts ->
                        if (sourcePort.getPortName().equals("MyExceptionalPort"))
                            {
                            // example for excluding a port from auto-mapping by
                                having a close look
                            // sourcePort.getMdfPort()....
                            return null
                        }
                        // default: do not change the auto-matched port
                        [optionalMatchedTargetPort]
                }
             }
        scriptLogger.info("Created {0} mappings.",mappedConnectors.size())
      }
    }
  }
}
```

Listing 6.241: Tries to auto-map all unconnected ports and evaluate matches

```
import com.vector.cfg.sysdesc.model.connector.SISourceComponentPort
import com.vector.cfg.sysdesc.model.connector.SITargetComponentPort

scriptTask("anotherExampleForUsingEvaluateMatches", DV_PROJECT){
   code {
      transaction {
         domain.runtimeSystem {
            def mappedConnectors =
               selectComponentPorts {
               unconnected()
                    } autoMapTo {
                        evaluateMatches {
                            SISourceComponentPort sourcePort,
                            SITargetComponentPort optionalMatchedTargetPort,
                            List<SITargetComponentPort> potentialTargetPorts ->

                            // iterate over potential target ports to find the
                                correct target

                            // like in java you can use a for loop
                            for (SITargetComponentPort targetCP :
                                potentialTargetPorts) {
                                if (targetCP.getPortName().startsWith("MyPort_"))
                                    {
                                    return [targetCP]
                                }
                            }

                            // or you can use a stream
                            def myTargets = potentialTargetPorts.findAll {
                                it.getPortName().startsWith("OtherPort_")
                            }
                            return myTargets
                        }
                    }
             scriptLogger.info("Created {0} mappings.",mappedConnectors.size())
         }
       }
    }
 }
```

Listing 6.242: Another example for using evaluate matches

```
import com.vector.cfg.sysdesc.model.connector.SISourceComponentPort
import com.vector.cfg.sysdesc.model.connector.SITargetComponentPort

scriptTask("automap1ToN", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def mappedConnectors =
                  selectComponentPorts {
                    // select single delegation port
                    delegation()
                    name "rDelegationSRPort1"
                  } autoMapTo {
                     selectTargetPorts {
                        // select a collection of target ports (names start with "
                            rSRPort")
                        name ~"rSRPort.*"
                     }
                     evaluateMatches {
                          SISourceComponentPort sourcePort,
                          SITargetComponentPort optionalMatchedTargetPort,
                          List<SITargetComponentPort> potentialTargetPorts ->
                             // return all potentialTargetPorts for 1:n connections
                                 , not only the one matched best
                             potentialTargetPorts
                     }
                  }
        scriptLogger.info("Created {0} mappings.",mappedConnectors.size())
      }
    }
  }
}
```

Listing 6.243: Auto-map a component port and realize 1:n connection by using evaluate matches

`forceConnectionWhen1To1()` allows to force a mapping even the usual auto-mapping rules will not match. Precondition is that the collections of source component ports and target component ports only contain one component port each. Otherwise no mapping is done.

```
scriptTask("autoMapTwoNonMatchingPorts", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def mappedConnectors =
                  selectComponentPorts {
                    // select a single source component port
                    name "prNVPort1"
                    component "NvApp1"
                  } autoMapTo {
                    selectTargetPorts {
                        // select a single target component port
                        name "rSRPort2"
                        component "App2"
                    }
                    // force the connection even names do not match at all
                    forceConnectionWhen1To1()
                  }
        scriptLogger.info("Created {0} mappings.",mappedConnectors.size())
      }
    }
  }
}
```

Listing 6.244: Create mapping between two ports which names do not match.

`useOriginContextForMatch()` uses another algorithm to match the component ports.
The standard algorithm matches only the names of the ports (delegation port of flat extract or port of a SWC of the flat extract).
This option uses also the origin context's name for the name matching.

Incomplete Connections:
Below we will talk about complete and incomplete connections. A connection is complete if the port of a SWC is connected to another SWC port or to a delegation port of the top level ECU composition. A connection is incomplete if a SWC port is connected to a delegation port of a composition which is not the top level ECU composition and the connection stops at this port (the delegation port is unconnected on the other side).

Origin Context:
Origin contexts of an inner port are delegation ports of the structured extract which are connected to this port and are the outermost ports of an incomplete connection. Delegation ports of the flat extract cannot have any origin contexts.

Example:
We want to map the port 'pData' of 'App1' of the flat extract. The corresponding component in the structured extract is instantiated inside the composition 'Comp1'. 'pData' is connected to the port 'pOriginContext' of 'Comp1'. 'pOriginContext' has no further ports connected to it.
When not using `useOriginContextForMatch()` option, only 'pData' would be used for the port name matching.
When using the `useOriginContextForMatch()` option, not only (but also) the name 'pData' is used for the port name matching, but also the origin context 'pOriginContext'.

```
scriptTask("mapUsingOriginContext", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def createdConnections = selectComponentPorts {
            component("App1OriginContextMatch")
            name("A")
        } autoMapTo {
            // this option will not only match the name of the port 'A'
            // but also the delegation ports' names of incomplete connections of
                the structured extract
            useOriginContextForMatch()
        }

        scriptLogger.info("Created '{0}' connections.", createdConnections.size())
     }
    }
  }
}
```

Listing 6.245: Use the origin context for the port name matching

Diagnostic Connections For diagnostic connections (previously created by RTE59002 solving actions) a special mode can be used while selecting the source component port.

`diagnosticConnection()` narrows down selection to component ports which are derived from diagnostic mappings. See `IComponentPortSelector.hasDiagnosticConnection()` for more details.

`diagnosticConnection()` cannot be combined with `and(Runnable)`, `or(Runnable)` and `not(Runnable)`.
If possible always prefer using `diagnosticConnection()` over `hasDiagnosticConnection()` which can be also combined with `not(Runnable)`, `and(Runnable)` and `or(Runnable)` due to performance reasons.

This mode has the similar behavior as the diagnostic connections mode in the component connection assistant. It also creates potential missing port interface mappings.

The enum EDiagnosticPortRole allows to filter ports for the different uses cases:

The `EDiagnosticPortRole` can be used to determine the role of a port when connecting diagnostic ports which depends on the used service needs.

```
scriptTask("connectDiagPorts", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
       def mappedConnectors =
                selectComponentPorts {
                   // activate diagnostic connection mode
                   // this mode shall not be part of a or{...}, and{...} or not
                       {...} closure
                   // hasDiagnosticConnection can be used instead if required, but
                        is very expensive in terms of performance
                   diagnosticConnection()
                   // further predicates can also be specified here if required
                } autoMapTo {
                   selectTargetPorts {
                       component "App1" // narrow down selection of target ports
                          if necessary
                   }
                }
        scriptLogger.info("Created {0} mappings.", mappedConnectors.size())
      }
    }
  }
}
```

Listing 6.246: Connects diagnostic ports of App1 based on diagnostic mappings

```
scriptTask("connectDiagPortsWithoutNvPorts", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
       def mappedConnectors =
                selectComponentPorts {
                   diagnosticConnection()
                   not {
                       nvData() // exclude nv ports
                   }
                } autoMapTo {
                }
        scriptLogger.info("Created {0} mappings. Excluded Nv Ports.",
            mappedConnectors.size())
      }
    }
  }
}
```

Listing 6.247: Connects diagnostic ports based on diagnostic mappings, excludes Nv Ports

```
import com.vector.cfg.sysdesc.model.port.EDiagnosticPortRole

scriptTask("connectDiagPortsIOControl", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
       def mappedConnectors =
                selectComponentPorts {
                   diagnosticConnection()
                   diagnosticPortRole(EDiagnosticPortRole.IO_CONTROL)
                } autoMap()
        scriptLogger.info("Created {0} mappings for IOControl ports.",
            mappedConnectors.size())
      }
    }
  }
}
```

Listing 6.248: Connects IOControl ports

Simple API for connection between ports This API covers the use case when the names of the components and ports which should be connected are known, but the naming rules are too specific or if you just want to increase the level of control by giving exactly the pairs that should be mapped into the API. There is one method for connecting exactly one component port to another and one API for doing multiple connections at once requiring a list of source and a list of target ports.

`componentPort(String, String)` allows to select an `SIComponentPort` and to do further operations on it, e.g. connecting the port to another port.

`ISelectedComponentPort` represents a selection of exactly one `SIComponentPort` and provides further actions on it.

`connectTo(String, String, Optional)` creates an `SIConnector` between the `SIComponentPort` which is represented by this `ISelectedComponentPort` and the `SIComponentPort` which is retrieved by the given component and port name. It is possible to connect the component port to a delegation port by using 'COMPOSITIONTYPE' as componentName.

The `connectTo(String, String, Optional)` returns only one connector, even when the created connection involves a connection chain instead. Use `connectToWithFullConnectorChain(String, String, Optional)` instead for getting the complete connection chain.

The API provides only the very basic checks.

- Direction of the ports is checked. E.g. it is not allowed to connect two PPorts within an AssemblySwConnector.
- Connecting incompatible types of port interfaces is not allowed. E.g. it is not allowed to connect a mode switch with a sender receiver port.
- Connecting two delegation ports is not allowed.
- The port interface mapping has to reference the port interfaces of the selected component ports.
- Connecting a terminated port is not allowed. Please remove the port terminator first. Use `ISelectedComponentPort.removePortTerminator()`.

- Creating redundant connections is not allowed. The ports cannot be connected by a second connector.

If you want to use some internal rules other than name matching to connect ports you can apply this rules to sort a list of source and another list of target ports and then use the simple API below that connects two lists of ports via index.

`connectTo(List)` creates `SIConnectors` between the `SIComponentPorts` which are represented by this `ISelectedComponentPorts` with the given targetPorts. The ports are connected using the index. The first port of this `ISelectedComponentPorts` will be connected to the first target port, the second to the second target port and so on.

In case you want to ignore already connected port pairs please use `assureConnectedTo(List)`.

If you want to connect ports using name matching please use `IRuntimeSystemApi.selectComponentPorts(Acti`

`assureConnectedTo(List)` checks whether the `SIComponentPorts` represented by this `ISelectedComponentPorts` are already connected to the given targetPorts matching them via index. In contrast to `connectTo(List)` this method does nothing if the ports are already connected. If the ports are not connected a new connector will be created.

Examples

```
import java.util.Optional

scriptTask ("assemblyConnectionExample", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {

                // enter component and port name of a component port and the
                    component port to connect it to
                // optionally you can add a port interface mapping
                def createdConnector = componentPort("App1", "pDataSend").
                    connectTo("App2", "rSRPort2", Optional.empty())

                scriptLogger.info("Created a connection between '{0}' and '{1}'.",
                createdConnector.getProviderPort().getName(),
                createdConnector.getRequesterPort().getName())
            }
        }
    }
}
```

Listing 6.249: Example how to create a simple assembly connection

```
import java.util.Optional

scriptTask ("delegationConnectionExample", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                // to select a delegation port the name of the ECU Composition is
                    used as component name
                def createdConnector = componentPort("COMPOSITIONTYPE", "
                    rDelegationSRPort1").connectTo("App2", "rSRPort2", Optional.
                    empty())

                // the provided and required port getter work also for delegation
                    connections
                // you can find more info in the java doc of IConnector
                scriptLogger.info("Created a connection between '{0}' and '{1}'.",
                createdConnector.getProviderPort().getName(),
                createdConnector.getRequesterPort().getName())
            }
        }
    }
}
```

Listing 6.250: Example how to create a simple delegation connection

```
import java.util.Optional

scriptTask ("delegationConnectionExample", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def portInterfaceMappingPath = "/ComponentTypes/
                    DataTypeMappingSets/PortInterfaceMappingSet1/Mapping1"

                // to add a port interface mapping to the connection, use an
                    optional with the AUTOSAR path to the port interface mapping
                def createdConnector = componentPort("COMPOSITIONTYPE", "
                    pDelegationSRPort2").connectTo("App1", "pSRPort1", Optional.of
                    (portInterfaceMappingPath))

                scriptLogger.info("Created a connection between '{0}' and '{1}'.",
                createdConnector.getProviderPort().getName(),
                createdConnector.getRequesterPort().getName())
            }
        }
    }
}
```

Listing 6.251: Create connector with port interface mapping

```
import com.vector.cfg.sysdesc.model.connector.SIConnector
import com.vector.cfg.sysdesc.model.component.SIComponentPort

scriptTask("UseSimpleAPIToCreateMultipleConnections", DV_PROJECT) {
    code {
        transaction {
            domain.runtimeSystem {
                // in this example we know for each port the target port (for
                    example stored in some external file)
                // and we know that the names of source and target ports do not
                    always match
                // so we use the simple API instead of the auto mapping
                List<String> sourcePortNames = ["App1.pDataSend",
                        "App1.pNvPort1",
                        "App1.pCSPort1"]
                List<String> targetPortNames = ["ECU Composition.
                    pDelegationSRPort2",
                        "App2.rNVPort1",
                        "App3.rOtherPort1CS"]

                // retrieve the component ports for the names
                List<SIComponentPort> sourcePorts = new ArrayList<SIComponentPort
                    >(selectComponentPorts {
                     componentPortNames(sourcePortNames)
                }.getComponentPorts())

                // since our used predicate does not guarantee any order, we have
                    to sort our ports to assure they are in correct order
                // because the connections below will be created matching index of
                     source and target ports
                sourcePorts.sort{a,b -> sourcePortNames.indexOf(a.getName()) <=>
                    sourcePortNames.indexOf(b.getName())}

                // do the same for the target ports
                List<SIComponentPort> targetPorts = new ArrayList<SIComponentPort
                    >(selectComponentPorts {
                     componentPortNames(targetPortNames)
                }.getComponentPorts())

                targetPorts.sort{a,b -> targetPortNames.indexOf(a.getName()) <=>
                    targetPortNames.indexOf(b.getName())}

                // we just want to make sure that the ports are connected
                // so we use assureConnectedTo(...) instead of connectTo(...)
                // in other words we do not care, which ports are already
                    connected
                List<SIConnector> newConnectors = componentPorts(sourcePorts).
                    assureConnectedTo(targetPorts)

                // finally do some reporting for the port pairs that were
                    unconnected
                for (SIConnector connector in newConnectors) {
                     scriptLogger.info("Connected {0} to {1}.",
                             connector.getProviderPort().getName(),
                             connector.getRequesterPort().getName())
                }
            }
        }
    }
}
```

Listing 6.252: Connect ports using simple API

6.10.4.10 Disconnect (unmap) Component Ports

The previous chapter was about connecting component ports. Now we want to have a look how to remove such connections again. We call it unmapping component ports.
This can be done using the component port selection API (see 6.10.4.1 on page 172) and calling a method to unmap the selected ports from other ports. It is possible to filter and evaluate the targets, so that you can have the control also for 1:n or n:m connected ports.

Unmapping Component Ports The use case of unmapping component ports is based on the selection of component ports. The targets are the ports which are connected to the selected port.

`unmap()` unmaps the selected component ports of this selection from ALL connected ports. In case that not all connections shall be removed the targets can be narrowed down using `unmapFrom(Action)`.

Examples for unmap()

```
import com.vector.cfg.dom.runtimesys.pai.api.IUnmappedPortsResult

scriptTask("unmapComponentPorts", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def unmappedPorts =
                selectComponentPorts {
                  // select the component ports to be unmapped
                  connected()
                  componentPortName("App1.pDataSend")
                } unmap()

        // the simple 'unmap()' removes all connectors connecting the selected
            component ports to any other ports

        // now print info of for the component ports that were unmapped from each
            other
        // the data structure is a pair representing the source and the target
            port which were unmapped
        for (final IUnmappedPortsResult unmappedPortPair : unmappedPorts) {
            scriptLogger.info("Removed connector between {0} -> {1}.",
                    unmappedPortPair.getSourcePort().getName(),
                    unmappedPortPair.getTargetPort().getName())
        }
      }
    }
  }
}
```

Listing 6.253: Remove Connectors between Component Ports

Control unmapping in unmapFrom(Closure)

`selectTargetPorts(Action)` allows to define predicates to narrow down the target ports which shall be disconnected from the in previous step selected ports.

`evaluateMatches(IUnmappingEvaluator)` allows to evaluate and change the results of the unmapped component ports.

For each selected component port the provided lambda is called: Parameters are the current handled component port and a list of all connected target component ports (respecting the `selectTargetPorts(Action)` predicates). The return value must be a list of component ports which are connected to the current handled source port.

```
import com.vector.cfg.dom.runtimesys.pai.api.IUnmappedPortsResult
import com.vector.cfg.sysdesc.model.component.SIComponentPort

scriptTask("unmapComponentPorts", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def unmappedPorts =
                selectComponentPorts {
                  // select the component ports to be unmapped
                  connected()
                  componentPortName("App1OriginContextMatch.
                      CompletedOriginPortTest")
                } unmapFrom {
                  // the connected (or target) ports can be filtered in this
                      closure
                  selectTargetPorts {
                    componentPortNames(["App1.ConnectedToCompletedOriginTest",
                        "App1_1.ConnectedToCompletedOriginTest"])
                  }

                  // additionally the target matches to be unmapped can be
                      evaluated
                  // the 'targetPorts' below are all ports connected to the '
                      sourcePort' considering the 'selectTargetPorts' call above
                  evaluateMatches { SIComponentPort sourcePort, List<
                      SIComponentPort> targetPorts ->
                    // in our example we want to unmap the source ports only from
                        ports of the component 'App1_1'
                    targetPorts.each {
                        if (it.getComponent().getName().equals("App1_1")) {
                            return [it]
                        }
                    }
                  }
                }

        // print info for newly unmapped ports
        for (final IUnmappedPortsResult unmappedPortPair : unmappedPorts) {
            scriptLogger.info("Removed connector between {0} -> {1}.",
                    unmappedPortPair.getSourcePort().getName(),
                    unmappedPortPair.getTargetPort().getName())
        }
      }
    }
  }
}
```

Listing 6.254: Remove Connectors between Component Ports Filtering Targets

6.10.4.11 Terminating Component Ports

Port terminators can be used to acknowledge the fact, that the port is not connected yet. This will prevent validation rules to produce validation results reporting these ports as unconnected or missing data mappings for these ports. It also allows you to filter such ports very easily using the

according predicates of the selection APIs.

Starting point is the component port selection (see 6.10.4.1 on page 172).

`terminate()` terminates all selected `SIComponentPorts`. If one of the selected ports is already terminated or connected to another component port, the port will be ignored.

The termination of component ports disables the validation of these ports. In other words, these ports are acknowledged as not connected yet. This should give a better overview of the open ports, which still have to be connected or need a data mapping.

`removePortTerminators()` removes the port terminators of all selected component ports. If one of the selected component ports is not terminated the method will ignore that port.

See `terminate()` for more information about the purpose of terminating ports.

It is also possible to create and remove port terminators via the simple API starting with the selection of one component port (componentPort(String, String)).

`terminate()` simple API to terminate the selected `SIComponentPort` if it is not already terminated or connected to another component port. You can use `SIComponentPort.isTerminated()` to check if a port is terminated and `SIComponentPort.isConnected()` if a port is connected to other ports.

The termination of component ports disables the validation of these ports. In other words, these ports are acknowledged as not connected yet. This should give a better overview of the open ports, which still have to be connected or need a data mapping.

`removePortTerminator()` simple API to remove the port terminator of the selected component port.

See `terminate()` for more information about the purpose of terminating ports.

Examples

```
import com.vector.cfg.sysdesc.model.component.SIComponentPort

scriptTask("terminatePort", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        // the statements below will return us a collection with all component
            ports which will be terminated
        def terminatedPorts =
             selectComponentPorts {
                delegation()
                name("pDelegationCSPort1")

             // use terminate() to terminate all selected component ports
             // if a selected port is already terminated or connected, it will not
                  be terminated (again)
             } terminate()

        // this result may contain less ports than the actual selected ports,
        // if some of the selected ports were connected or terminated
        for (final SIComponentPort terminatedPort : terminatedPorts) {
            scriptLogger.info("Terminated component port '{0}'.", terminatedPort.
                getName())
        }
      }
    }
  }
}
```

Listing 6.255: Terminate port using the component port selection API

```
import com.vector.cfg.sysdesc.model.component.SIComponentPort

scriptTask("removePortTerminator", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        // the statements below will return us a collection with all component
            ports for which a port terminator was removed
        def removedPortTerminators =
             selectComponentPorts {
                // filter for all terminated delegation ports
                delegation()
                // there are predicates to filter for terminated() and
                    notTerminated() ports
                terminated()

             // if a port is not terminated it will be skipped
             } removePortTerminators()

        // the result may contain less component ports than the selection,
        // if some of the ports were not terminated
        for (final SIComponentPort componentPort : removedPortTerminators) {
            scriptLogger.info("Removed port terminator of component port '{0}'.",
                componentPort.getName())
        }
      }
    }
  }
}
```

Listing 6.256: Remove port terminator using the component port selection API

```
scriptTask("createPortTerminator", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        // select the component port via the simple API and call terminate()
        // this API is more strict and throws an exception if the selected port
            cannot be terminated
        def terminatedPort = componentPort("COMPOSITIONTYPE", "rDelegationSRPort1"
            ).terminate()

        scriptLogger.info("Terminated component port '{0}'.", terminatedPort.
            getName())
      }
    }
  }
}
```

Listing 6.257: Create a port terminator using the simple API

```
scriptTask("removePortTerminator", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        // select the component port via the simple API and call terminate()
        // this API is more strict and throws an exception if the selected port
            has no port terminator to remove
        def removedTerminatorPort = componentPort("App1_1", "pSRPort1").
            removePortTerminator()

        scriptLogger.info("Terminated component port '{0}'.",
            removedTerminatorPort.getName())
      }
    }
  }
}
```

Listing 6.258: Remove a port terminator using the simple API

Terminating Component Ports of Communication Elements As already mentioned above port terminators can be used to acknowledge the fact, that the port is not connected yet. It is possible to terminate owner ports of communication elements using the communication element selection.

`terminateOwnerPorts()` terminates the `SIComponentPorts` of all selected `SICommunicationElements`. If one of the selected ports is already terminated or connected to another component port, the port will be ignored.

The termination of component ports disables the validation of these ports. In other words, these ports are acknowledged as not connected yet. This should give a better overview of the open ports, which still have to be connected or need a data mapping.

`removePortTerminatorsForOwnerPorts()` removes the port terminators of the `SIComponentPorts` of all selected `SICommunicationElements`.

See `terminateOwnerPorts()` for more information about the purpose of terminating ports.

Examples

```
import com.vector.cfg.sysdesc.model.component.SIComponentPort

scriptTask("terminatePort", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        // the the statements below will return us a collection with all component
             ports which will be terminated
        def terminatedPorts =
             selectCommunicationElements {
                // also for the communication elements there are predicates to
                    filter for terminated owner ports
                ownerPortNotTerminated()
                port("pSRPort1")

             // this call will terminate the component port owner of each selected
                  communication element
             } terminateOwnerPorts()

        // if multiple communication elements has the same component port owner
        // the component port will be terminated and returned only once
        for (final SIComponentPort terminatedPort : terminatedPorts) {
            scriptLogger.info("Terminated component port '{0}'.", terminatedPort.
                getName())
        }
      }
    }
  }
}
```

Listing 6.259: Terminate port using the communication element selection API

```
import com.vector.cfg.sysdesc.model.component.SIComponentPort

scriptTask("removePortTerminator", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        // the statements below will return us a collection with all component
            ports for which a port terminator was removed
        def removedPortTerminators =
             selectCommunicationElements {
                component("App1_1")
                port("pSRPort1")

             // if a port is not terminated it will be skipped
             } removePortTerminatorsForOwnerPorts()

        for (final SIComponentPort componentPort : removedPortTerminators) {
            scriptLogger.info("Removed port terminator of component port '{0}'.",
                componentPort.getName())
        }
      }
    }
  }
}
```

Listing 6.260: Remove port terminator using the communication element selection API

6.10.4.12 Data Mapping

The data mapping use case allows to connect signal instances and data elements / operations / triggers. We will introduce two ways for that.
The first one will be using a selection API allowing to define predicates to filter communication elements/signals for the data mapping and calling a method to filter the target signals/communication elements. This is the way to map communication elements to system signals in a way like the data mapping assistant from the GUI. See 6.10.4.3 on page 182 and 6.10.4.2 on page 178 for the selection starting points.
The second way is to use a simple API that requires already prepared data structures e.g. a list of communication elements and a list of signal instances. To initially find the appropriate communication elements and signals you can use the common selection APIs.

Mapping signal instances The use case of auto-mapping signal instances is based on the selection of signal instances.

`autoMap()` tries to auto-map the selection of `SIAbstractSignalInstances` (`SISignalInstance` or `SISignalGroupInstance`) according the data mapping assistant default rules. Therefore the selection of possible target communication elements is computed and tried to match to the selected signal instances.

Examples for autoMap()

```
scriptTask("autoDatamapAllUnmappedSignalInstances", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def dataMappings =
                            selectSignalInstances {
                              unmapped()
                            } autoMap()
                scriptLogger.info("Created {0} data mappings.",dataMappings.size()
                    )
            }
        }
    }
}
```

Listing 6.261: Auto data map all unmapped signal instances

`autoMapTo(Action)` tries to auto-map the selection of signal instances according the data mapping assistant rules but offers more control for the auto-mapping: Inside the lambda additional predicates for narrowing down the target communication elements can be defined and code to evaluate and change the auto-mapper results can be provided.

`autoMapTo(Action)` will produce trace, info and warning logs. To see them, activate the 'com.vector.cfg.dom.r logger with the appropriate log level.

Control the auto-mapping in autoMapTo(Closure)

`selectTargetCommunicationElements(Action)` allows to define predicates to narrow down the target communciation elements for the auto-mapping. The predicates are used to filter the possible target communication elements which were computed from the signal instance selection.

`evaluateMatches(IAutoMappingEvaluator)` allows to evaluate and change the results of the auto-mapping. It corresponds to the confirm page of the data mapping assistant.

For each signal instance the provided lambda is called: Parameters are the signal instance, the optional matched target communication element (or null), and a list of all potential target communication elements (respecting the `selectTargetCommunicationElements(Action)` predicates). The return value must be a communication element or null.

```
import com.vector.cfg.sysdesc.model.communication.instance.
    SIAbstractSignalInstance
import com.vector.cfg.sysdesc.model.communication.SICommunicationElement

scriptTask("autoDatamapAllUnmappedSignalInstancesAndEvaluate", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def dataMappings =
                     selectSignalInstances {
                       unmapped()
                     } autoMapTo {
                          selectTargetCommunicationElements {
                               unmapped()
                          }
                          evaluateMatches {
                             SIAbstractSignalInstance signal,
                             SICommunicationElement optionalMatchedComElement,
                             List<SICommunicationElement>  potentialComElements ->
                                  // evaluate
                                  optionalMatchedComElement
                          }
                     }
                scriptLogger.info("Created {0} data mappings.",dataMappings.size()
                    )
            }
        }
    }
}
```

Listing 6.262: Auto data map all unmapped signal instances to unmapped communication elements and evaluate

Nested Array of Primitives `expandNestedArraysOfPrimitive(boolean)` allows to control the expansion of nested arrays of primitive globally. Per default, arrays are fully expanded (allowing to data map each array element). By setting the value to 'false', all nested arrays of primitive are not expanded and can be directly data-mapped to a signal.

```
import com.vector.cfg.sysdesc.model.communication.instance.
    SIAbstractSignalInstance
import com.vector.cfg.sysdesc.model.communication.SICommunicationElement

scriptTask("autoDatamapAllSignalInstancesAndDoNotExpandNestedArrayElements",
    DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def dataMappings =
           selectSignalInstances {
           } autoMapTo {
              // do not expand nested array elements
              expandNestedArraysOfPrimitive false
              evaluateMatches {
               SIAbstractSignalInstance signal,
               SICommunicationElement optionalMatchedComElement,
               List<SICommunicationElement>  potentialComElements ->
                 // perform manual mapping to a signal group
                 if (signal.getName().equals("elemB_c255f5e38fd8b21d")) {
                  for (final SICommunicationElement comElement :
                      potentialComElements) {
                    if (comElement.getFullyQualifiedName().equals("App2.rSRPort1.
                        Element_2")) {
                      return comElement
                    }
                  }
                }
                // now check: for the group signal the the record element
                    representing an array is not expanded
                if (signal.getName().equals("fieldA_f1d3783e235e88d3")) {
                  // group signal
                  for (final SICommunicationElement comElement :
                      potentialComElements) {
                    if (comElement.getFullyQualifiedName().equals("App2.rSRPort1.
                        Element_2.RecordElement")) {
                      // do some direct mapping here
                    }
                  }
                }
                optionalMatchedComElement
              }
           }
        scriptLogger.info("Created {0} data mappings.",dataMappings.size())
      }
    }
  }
}
```

Listing 6.263: Auto data map all signal instances and do not expand nested array elements

`expandNestedArraysOfPrimitive(String,boolean)` allows to control the expansion of nested arrays of primitive for single nested arrays. Per default, the `expandNestedArraysOfPrimitive(boolean)` applies. For the given fully qualified communication element name, the global setting can be overridden.

```
import com.vector.cfg.sysdesc.model.communication.instance.
    SIAbstractSignalInstance
import com.vector.cfg.sysdesc.model.communication.SICommunicationElement

scriptTask("autoDatamapAllSignalInstancesAndDoExpandSpecificNestedArrayElement",
    DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def dataMappings =
           selectSignalInstances {
           } autoMapTo {
              // do not expand nested array elements
              expandNestedArraysOfPrimitive false
              expandNestedArraysOfPrimitive( "App2.rSRPort1.Element_2.
                  RecordElement",true)
              evaluateMatches {
               SIAbstractSignalInstance signal,
               SICommunicationElement optionalMatchedComElement,
               List<SICommunicationElement>  potentialComElements ->
                 // perform manual mapping to a signal group
                 if (signal.getName().equals("elemB_c255f5e38fd8b21d")) {
                  for (final SICommunicationElement comElement :
                      potentialComElements) {
                    if (comElement.getFullyQualifiedName().equals("App2.rSRPort1.
                        Element_2")) {
                     return comElement
                    }
                  }
                 }
                 // now check: for the group signal the the record element
                     representing an array is expanded:
                 // the single array elements can be mapped
                 if (signal.getName().equals("fieldA_f1d3783e235e88d3")) {
                   // group signal
                   for (final SICommunicationElement comElement :
                       potentialComElements) {
                     if (comElement.getFullyQualifiedName().equals("App2.rSRPort1.
                         Element_2.RecordElement[0]")) {
                       // do some direct mapping to array element here
                     }
                   }
                 }
                 optionalMatchedComElement
              }
           }
        scriptLogger.info("Created {0} data mappings.",dataMappings.size())
      }
    }
  }
}
```

Listing 6.264: Auto data map all signal instances and expand specific nested array element

`evaluateMatchesWithCompatibility(IAutoMappingEvaluatorWithCompatibility)` allows to evaluate and change the results of the auto-mapping. It corresponds to the confirm page of the data mapping assistant. In contrast to `evaluateMatches(IAutoMappingEvaluator)` this method also provides the compatibility of the optional match.

For each signal instance the provided lambda is called: Parameters are the signal instance, the optional matched target communication element (or null), their compatibility and a list of all potential

target communication elements (respecting the `selectTargetCommunicationElements(Action)` predicates). The return value must be a communication element or null.

```
import com.vector.cfg.sysdesc.model.communication.ECompatibility
import com.vector.cfg.sysdesc.model.communication.instance.
    SIAbstractSignalInstance
import com.vector.cfg.sysdesc.model.communication.SICommunicationElement
import com.vector.cfg.sysdesc.model.datamapping.SIDataMapping

scriptTask("evaluateCommunicationElementsByCompatibility", DV_PROJECT) {
    code {
        transaction {
            domain.runtimeSystem {

                // in this example we want to accept full matches for data mapping
                // and apply some custom rules for non-full matches

                List<SIDataMapping> createdMappings = selectSignalInstances {
                    unmapped()
                } autoMapTo {
                    evaluateMatchesWithCompatibility {
                        SIAbstractSignalInstance signal,
                        SICommunicationElement optionalMatchedCommunicationElement
                            ,
                        ECompatibility compatibility,
                        List<SICommunicationElement> potentialComElements ->

                        if (compatibility == ECompatibility.FULL) {
                            return optionalMatchedCommunicationElement
                        }
                        // for non-full matches we return the first potential
                            match if present
                        if (potentialComElements.size() > 0) {
                            return potentialComElements.get(0)
                        }
                        return null
                    }
                }

                scriptLogger.info("Created {0} data mappings.", createdMappings.
                    size())
            }
        }
    }
}
```

Listing 6.265: Evaluate matched communication elements using compatibility

`confirmByCompatibility(IAutoMappingConfirmation)` allows to evaluate the mappings which should be created.

For each signal instance the provided lambda is called: Parameters are the signal instance, the optional matched target communication element (or null) and the compatibility of them (`ECompatibility.NULL` if no optional match present) respecting all previous evaluations. So this is the final verifying step to confirm or reject the mapping. The return value must be true if the mapping should be created or false if you want to reject the mapping.

```
import com.vector.cfg.sysdesc.model.communication.ECompatibility
import com.vector.cfg.sysdesc.model.communication.instance.
    SIAbstractSignalInstance
import com.vector.cfg.sysdesc.model.communication.SICommunicationElement
import com.vector.cfg.sysdesc.model.datamapping.SIDataMapping

scriptTask("confirmMappingToComElementByCompatibility", DV_PROJECT) {
    code {
        transaction {
            domain.runtimeSystem {

                // in this example we want only to accept data mappings with a
                    full match
                // and report all other

                List<SIDataMapping> createdMappings = selectSignalInstances {
                    unmapped()
                } autoMapTo {
                    confirmByCompatibility {
                        SIAbstractSignalInstance signal,
                        SICommunicationElement optionalMatchedCommunicationElement
                            ,
                        ECompatibility compatibility ->

                        if (compatibility == ECompatibility.FULL) {
                            // accept full match
                            return true
                        }

                        if (optionalMatchedCommunicationElement != null) {
                            // report non-full matches
                            scriptLogger.info("Compatibility {0} between signal
                                {1} and communication element {2}.",
                                compatibility,
                                signal.getName(),
                                optionalMatchedCommunicationElement.
                                    getFullyQualifiedName())
                        } else {
                            // report signals for which the auto mapper could not
                                find a match
                            scriptLogger.info("No match for signal {0}.",
                                signal.getName())
                        }
                        return false
                    }
                }

                scriptLogger.info("Created {0} data mappings.", createdMappings.
                    size())
            }
        }
    }
}
```

Listing 6.266: Decide which mappings should be created by compatibility

Mapping communication elements `autoMap()` tries to auto-map the selection of `SICommunicationElements` (`SIDataCommunicationElement` or `SIOperationCommunicationElement`) according the data mapping assistant default rules. Therefore the selection of possible target signal instances is computed and tried to match to the selected communication elements. You do not

have to expand the communication elements in the previous selection step. They will be expanded after the root is mapped automatically as you know from the data mapping assistant.

Examples for autoMap()

```
scriptTask("autoDatamapAllUnmappedSRDelPortComElements", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def dataMappings =
                  selectCommunicationElements {
                    // select all unmapped sender/receiver delegation ports
                    delegation()
                    unmapped()
                    senderReceiver()
                  } autoMap()
        scriptLogger.info("Created {0} data mappings.",dataMappings.size())
      }
    }
  }
}
```

Listing 6.267: Auto data map all unmapped sender/receiver delegation port communication elements

`autoMapTo(Action)` tries to auto-map the selection of communication elements according the data mapping assistant rules but offers more control for the auto-mapping: Inside the lambda additional predicates for narrowing down the target signal instances can be defined and code to evaluate and change the auto-mapper results can be provided. You do not have to expand the communication elements in the previous selection step. They will be expanded after the root is mapped automatically as you know from the data mapping assistant.

`autoMapTo(Action)` will produce trace, info and warning logs. To see them, activate the

'`com.vector.cfg.dom.runtimesys.pai.api.ICommunicationElementSelection`'

logger with the appropriate log level.

Control the auto-mapping in autoMapTo(Closure)

`selectTargetSignalInstances(Action)` allows to define predicates to narrow down the target signal instances for the auto-mapping. The predicates are used to filter the possible target signal instances which were computed from the communication element selection.

`evaluateMatches(IAutoMappingEvaluator)` allows to evaluate and change the results of the auto-mapping. It corresponds to the confirm page of the data mapping assistant.

For each communication element the provided lambda is called: Parameters are the communication element, the optional matched target signal instance (or null), and a list of all potential target signal instances (respecting the `selectTargetSignalInstances(Action)` predicates). The return value must be a signal instance or null.

```
import com.vector.cfg.sysdesc.model.communication.instance.
    SIAbstractSignalInstance
import com.vector.cfg.sysdesc.model.communication.SICommunicationElement

scriptTask("autoDatamapAllUnmappedComElementsAndEvaluate", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def dataMappings =
             selectCommunicationElements {
               unmapped() // only unmapped communication elements
             } autoMapTo {
                  selectTargetSignalInstances {
                       // only map to unmapped rx signal instances
                       unmapped()
                       rx()
                  }

                  // we selected only the root communication elements, but for
                      performing the data mapping the complex data elements are
                      expanded
                  // this is a behavior which you can also notice in the data
                      mapping assistant in the GUI
                  // that means the evaluateMatches method will also offer child
                      communication elements and the corresponding matched group
                      signals
                  evaluateMatches {
                     SICommunicationElement  communicationElement,
                     SIAbstractSignalInstance optionalMatchedSignalInstance,
                     List<SIAbstractSignalInstance>  potentialSignalinstances ->
                          // evaluate the match here
                          if (optionalMatchedSignalInstance != null) {
                                def mdfSystemSignal =
                                    optionalMatchedSignalInstance.getMdfObject()
                                // check more specific ...
                          }
                          optionalMatchedSignalInstance
                  }
             }
        scriptLogger.info("Created {0} data mappings.", dataMappings.size())
      }
    }
  }
}
```

Listing 6.268: Auto data map all unmapped communication elements to unmapped rx signal instances and evaluate

Nested Array of Primitives `expandNestedArraysOfPrimitive(boolean)` allows to control the expansion of nested arrays of primitive globally. Per default, arrays are fully expanded (allowing to data map each array element). By setting the value to 'false', all nested arrays of primitive are not expanded and can be directly data-mapped to a signal.

```
import com.vector.cfg.sysdesc.model.communication.instance.
    SIAbstractSignalInstance
import com.vector.cfg.sysdesc.model.communication.SICommunicationElement
import com.vector.cfg.sysdesc.model.communication.instance.SISignalGroupInstance

scriptTask("autoDatamapDoNotExpandNestedArrayElements", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def dataMappings =
         selectCommunicationElements {
         } autoMapTo {
             expandNestedArraysOfPrimitive false // do not expand nested arrays of
                 primitive
             evaluateMatches {
                 SICommunicationElement  communicationElement,
                 SIAbstractSignalInstance optionalMatchedSignalInstance,
                 List<SIAbstractSignalInstance>  potentialSignalInstances ->
                    if ("App2.rSRPort1.Element_2".equals(communicationElement.
                        getFullyQualifiedName())) {
                        // manual matching: map to first signal group
                        for (SIAbstractSignalInstance potentialSignal:
                            potentialSignalInstances) {
                            if (potentialSignal instanceof SISignalGroupInstance)
                                {
                                return potentialSignal
                            }
                        }
                    }
                    if ("App2.rSRPort1.Element_2.RecordElement".equals(
                        communicationElement.getFullyQualifiedName())) {
                        // now the RecordElement which represents an array is
                            directly offered to map
                        // ....
                    }
                    optionalMatchedSignalInstance
             }
         }
        scriptLogger.info("Created {0} data mappings.",dataMappings.size())
      }
    }
  }
}
```

Listing 6.269: Autodatamap and do not expand nested array elements

`expandNestedArraysOfPrimitive(String,boolean)` allows to control the expansion of nested arrays of primitive for single nested arrays. Per default, the `expandNestedArraysOfPrimitive(boolean)` applies. For the given fully qualified communication element name, the global setting can be overridden.

The fully qualified communication element name is e.g. determinable when using the data mapping assistant, performing an arbitrary signal group mapping of the root data element, and using the right-mouse menu its 'Copy fully qualified name' action on the nested array element.

```
import com.vector.cfg.sysdesc.model.communication.instance.
    SIAbstractSignalInstance
import com.vector.cfg.sysdesc.model.communication.SICommunicationElement
import com.vector.cfg.sysdesc.model.communication.instance.SISignalGroupInstance

scriptTask("autoDatamapDoExpandSpecificNestedArrayElement", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def dataMappings =
             selectCommunicationElements {
             } autoMapTo {
                 // do not generally expand nested arrays of primitive
                 expandNestedArraysOfPrimitive false
                 // but expand the following specific record element
                 expandNestedArraysOfPrimitive("App2.rSRPort1.Element_2.
                     RecordElement",true)
                 evaluateMatches {
                     SICommunicationElement  communicationElement,
                     SIAbstractSignalInstance optionalMatchedSignalInstance,
                     List<SIAbstractSignalInstance>  potentialSignalInstances ->
                        if ("App2.rSRPort1.Element_2".equals(communicationElement.
                            getFullyQualifiedName())) {
                            // manual matching: map to first signal group
                            for (SIAbstractSignalInstance potentialSignal:
                                potentialSignalInstances) {
                                if (potentialSignal instanceof
                                    SISignalGroupInstance) {
                                    return potentialSignal
                                }
                            }
                        }
                        if ("App2.rSRPort1.Element_2.RecordElement[0]".equals(
                            communicationElement.getFullyQualifiedName())) {
                            // the RecordElement (representing an array of
                                primitive)  is expanded to map the single array
                                elements
                            // ....
                        }
                        optionalMatchedSignalInstance
                 }
             }
        scriptLogger.info("Created {0} data mappings.",dataMappings.size())
      }
    }
  }
}
```

Listing 6.270: Autodatamap and do expand a specific nested array element

`evaluateMatchesWithCompatibility(IAutoMappingEvaluatorWithCompatibility)` allows to evaluate and change the results of the auto-mapping. In contrast to `evaluateMatches(IAutoMappingEvaluator)` this method also provides the compatibility of the optional match.

For each communication element the provided lambda is called: Parameters are the communication element, the optional matched target signal instance (or null), the compatibility of them and a list of all potential target signal instances (respecting the `selectTargetSignalInstances(Action)` predicates). The return value must be a signal instance or null.

```
import com.vector.cfg.sysdesc.model.communication.ECompatibility
import com.vector.cfg.sysdesc.model.communication.instance.
    SIAbstractSignalInstance
import com.vector.cfg.sysdesc.model.communication.SICommunicationElement
import com.vector.cfg.sysdesc.model.datamapping.SIDataMapping

scriptTask("evaluateSignalsByCompatibility", DV_PROJECT) {
    code {
        transaction {
            domain.runtimeSystem {

                // in this example we want to accept full matches for data mapping
                // and apply some custom rules for non-full matches

                List<SIDataMapping> createdMappings = selectCommunicationElements
                    {
                     unmapped()
                     senderReceiver()
                } autoMapTo {
                     evaluateMatchesWithCompatibility {
                         SICommunicationElement communicationElement,
                         SIAbstractSignalInstance optionalMatchedSignal,
                         ECompatibility compatibility,
                         List<SIAbstractSignalInstance> potentialSignals ->

                         if (compatibility == ECompatibility.FULL) {
                             return optionalMatchedSignal
                         }
                         // for non-full matches we return the first potential
                             match if present
                         if (potentialSignals.size() > 0) {
                             return potentialSignals.get(0)
                         }
                         return null
                     }
                }

                scriptLogger.info("Created {0} data mappings.", createdMappings.
                    size())
            }
        }
    }
}
```

Listing 6.271: Evaluate matched signal instances using compatibility

`confirmByCompatibility(IAutoMappingConfirmation)` allows to evaluate the mappings which should be created.

For each communication element the provided lambda is called: Parameters are the communication element, the optional matched target signal instance (or null) and the compatibility of them (`ECompatibility.NULL` if no optional match present) respecting all previous evaluations. So this is the final verifying step to confirm or reject the mapping. The return value must be true if the mapping should be created or false if you want to reject the mapping.

```
import com.vector.cfg.sysdesc.model.communication.ECompatibility
import com.vector.cfg.sysdesc.model.communication.instance.
    SIAbstractSignalInstance
import com.vector.cfg.sysdesc.model.communication.SICommunicationElement
import com.vector.cfg.sysdesc.model.datamapping.SIDataMapping

scriptTask("confirmMappingToSignalByCompatibility", DV_PROJECT) {
    code {
        transaction {
            domain.runtimeSystem {

                // in this example we want only to accept data mappings with a
                    full match
                // and report all other


                List<SIDataMapping> createdMappings = selectCommunicationElements
                    {
                    unmapped()
                    senderReceiver()
                } autoMapTo {
                    confirmByCompatibility {
                        SICommunicationElement communicationElement,
                        SIAbstractSignalInstance optionalMatchedSignal,
                        ECompatibility compatibility ->

                        if (compatibility == ECompatibility.FULL) {
                            return true
                        }
                        if (optionalMatchedSignal != null) {
                            // report non-full matches
                            scriptLogger.info("Compatibility {0} between signal
                                {1} and communication element {2}.",
                                compatibility,
                                optionalMatchedSignal.getName(),
                                communicationElement.getFullyQualifiedName())
                        } else {
                            // report signals for which the auto mapper could not
                                find a match
                            scriptLogger.info("No match for communication element
                                {0}.",
                                communicationElement.getName())
                        }
                        return false
                    }
                }

                scriptLogger.info("Created {0} data mappings.", createdMappings.
                    size())
            }
        }
    }
}
```

Listing 6.272: Decide which mappings should be created by compatibility

Compatibility Between Communication Elements and Signal Instances `ECompatibility` represents the compatibility between an `SIAbstractSignalInstance` and an `SICommunicationElement` for a potential or existing `SIDataMapping`. This enum helps to determine how good the elements are matching in terms of names and types and is available for example at the data mapping

assistant or the data mapping automation API.

FULL For a `FULL` match the `SIAbstractSignalInstance` and the `SICommunicationElement` have to be compatible regarding their types (see `TYPE_INCOMPATIBLE`) and the name of the signal has to match either the owner port name of the communication element, the communication element name itself or a combination of the two. All names are normalized (e.g. removing some common signal group and group signal suffixes and convert capital to small letters) before performing the match.

`FULL` matches will be mapped automatically by the auto-mapper when not using further evaluation.

Examples:
Fully qualified `SICommunicationElement` names on the left mapped to fully qualified `SIAbstractSignalInstance` names on the right.

- 'MyPort.MyData' -> 'MyData': full match, signal and data element names are equal.
- 'MyData.DataElement' -> 'MyData': full match, signal and port names are equal.
- 'MyData_Record' -> 'MyData_SignalGroup': full match, the suffix is recognized as marker of signal group and marker of record without further meaning.
- 'ParkingBrake_Status' -> 'ParkingBrake_Position': no match, the suffixes are no obvious markers which can be ignored, compatibility `NONE`.
- 'MyPort.MyStructure.MyData1' -> 'MyStructure.MyData1': full match, group signal and record element have equal names.
- 'ParkingBrake.ParkingBrakeStatus' -> 'MyStatus1': no full match, compatibility `NONE`.
- 'ParkingBrake.ParkingBrakeStatus' -> 'BrakeStatus': no full match, but a `PARTIAL_NAME` match because signal name is contained in data element name.

PARTIAL_NAME For a `PARTIAL_NAME` match the `SIAbstractSignalInstance` and the `SICommunicationElement` have to be compatible regarding their types (see `TYPE_INCOMPATIBLE`). Additionally the name of the signal should be contained in the owner port name of the communication element or in the communication element name itself, but also vice versa, that means if the owner port name or the communication element name is contained in the signal name. All names are normalized (e.g. removing some common signal group and group signal suffixes and convert capital to small letters) before performing the match.

`PARTIAL_NAME` matches will be mapped automatically by the auto-mapper when not using further evaluation.

Examples:
Fully qualified `SICommunicationElement` names on the left mapped to fully qualified `SIAbstractSignalInstance` names on the right.
- 'ParkingBrake.BrakeStatus' -> 'ParkingBrakeStatus': partial name match, data element name is part of signal name.
- 'ParkingBrake.ParkingBrakeStatus' -> 'BrakeStatus': partial name match, signal name is part of data element name.
- 'SendBrakeStatus.ParkingBrake' -> 'Status': partial name match, signal name is part of port name.
- 'SendStatus.ParkingBrake' -> 'ParkingBrake_SendStatus': partial name match, port name is part of signal name.

TYPE_INCOMPATIBLE An SIAbstractSignalInstance is TYPE_INCOMPATIBLE to an SICommunicationElement if they match FULL or by PARTIAL_NAME and the values of dynamic length attribute of the signal and variable size of the communication element's data type do not match, but also if the signal is an SISignalGroupInstance and the communication element's data type uses variable size. Signals using data transformation are never TYPE_INCOMPATIBLE.

TYPE_INCOMPATIBLE matches will be mapped automatically by the auto-mapper when not using further evaluation.
Hint: At first it sounds strange that the auto-mapper accepts TYPE_INCOMPATIBLE matches. The reason is that the auto-mapper is strongly based on name matching and since the names match (as it is the case here), the auto-mapper will already do the mapping, but you have probably to correct some attributes at the signal or the data type which do not match some expectations of our validations.

STRUCTURE_INCOMPATIBLE Compatibility STRUCTURE_INCOMPATIBLE is only used for SISignalGroupInstances. It is used if the compatibility between the signal group and the root communication element is either FULL or PARTIAL_NAME and at the same time there is at least one communication element leaf for which neither a group signal that matches FULL nor by PARTIAL_NAME exist below the signal group.

Examples:
A signal group named 'MyComplexData' has the group signals 'SubData1' and 'OtherGroupSignal'. The communication element of port 'MyPort' to be matched is named 'MyComplexData' and is representing a record with the record elements 'SubData1' and 'OtherRecordElement'. So that the roots will match by names and types, but no match for the record element 'OtherRecordElement' does exist at the signal group, since group signal 'OtherGroupSignal' is not matching by name.
The auto-mapper will produce the following result for that:
MyPort.MyComplexData -> MyComplexData
MyPort.MyComplexData.SubData1 -> MyComplexData.SubData1
MyPort.MyComplexData.OtherRecordElement -> unmapped

STRUCTURE_INCOMPATIBLE matches will NOT be mapped automatically by the auto-mapper.

NONE Compatibility NONE is used in the following cases:

1. SIAbstractSignalInstance and SICommunicationElement names are neither matching FULL, nor by PARTIAL_NAME.
2. EDirections of SIAbstractSignalInstance and SICommunicationElement are incompatible.
3. If the SIAbstractSignalInstance does not use data transformation but the communication element is an SIOperationCommunicationElement.
4. The length of the signal is not 0 but the communication element is an SITriggerCommunicationElement.
5. The SIAbstractSignalInstance is not a signal group and does not use data transformation but the communication element represents a record, a union or an array of non-primitives.
6. If the SIAbstractSignalInstance is a group signal but the communication element is not a leaf of a complex SIDataCommunicationElement.

7. `SIAbstractSignalInstance` is a signal group but the communication element is neither representing a record nor an array.

`NONE` matches will NOT be mapped automatically by the auto-mapper.

NULL The compatibility `NULL` is only used in case the auto-mapper did not find any matching communication elements.

Simple API for data mapping This API can be used when the names of the communication elements, which should be data mapped, are known, as well as the AUTOSAR paths to the system signal groups and system signals or if you want to use custom rules which you apply to sort the elements and want to map a list of communication elements and a list of signals via index which helps you to increase the level of control. Possible entry points are the selection of a communication element and potentially also its child communication elements by using the fully qualified names or using the communication and signal selections introduced above to select and later sort a list of communication elements and a list of signals. For complex data mappings there is a comfort function. If you define the root mapping only (e.g. mapping a record to a system signal group) the auto-mapper will try to match the children (e.g. record elements and group signals) by naming. This comfort function does not expand primitive arrays.

`communicationElement(String...)` allows to select an `SICommunicationElement` and to do further operations on it, e.g. performing a data mapping. For complex communication elements also the children can be selected. In such case the first communicationElementName has to be the name of the root element.

`ISelectedCommunicationElement` represents a selection of exactly one `SICommunicationElement` and provides further actions on it. For complex communication elements, also a subset of the child elements is represented.

`mapTo(String...)` creates an `SIDataMapping` for the `SICommunicationElement` which is represented by this `ISelectedCommunicationElement`. The mapped signal is the given abstractSignalInstance.

In case of a complex mapping first the system signal group has to be referenced. The child mappings are created considering the given order or in other words, the first communication element is mapped to the first signal, the second element to the second signal and so on.
For a client server to signal mapping, select the operation communication element and enter the paths to two serialized signals. Which signal is the call and which one the return signal is determined by the direction of the signals.
Hint: Use 'ECU Composition' or 'COMPOSITIONTYPE' as component name to select communication elements of delegation ports.

There are a few checks also for the simple API.

- Before creating the data mapping check if an equal mapping already exists. Do not create redundant mappings.
- Check that the amount of the selected communication elements is equal to the amount of the selected signals. For the client server use case there will be one communication element for the call direction and one for the return direction.

- Checks that the direction of the signal and communication element are compatible. Checks also the type compatibility of signal (group) and communication element.
  An operation communication element can only be mapped to a serialized signal.
  Records can only be mapped to serialized signals or signal groups.
  Unions can only be mapped to serialized signals.
  Arrays with complex array element can only be mapped to serialized signals or signal groups.
  Trigger communication elements cannot be mapped to signal groups.

- For the client server data mappings, check that exactly two singals are selected and that both signals are serialized signals.
- Checks for complex mappings, that the first abstract signal instance is a signal group and the other instances are group signals of this signal group.
- Check that the selected communication elements are suitable for data mapping. That means they should belong to a sender receiver, client server or a trigger port, which is not a service port and not a PRPort.
- Check the hierarchy of the selected communication elements for complex mappings. First selected element should be the root element and all further selected elements children of the root element.
- Check for the client server use case, that both communication elements, one for the call and one for the return direction can be found. The communication elements have to be operation communication elements.
- Check for a complex mapping that the first selected communication element is really a complex root element.
- Check if the owner port of the communication element is terminated. Terminated ports should not be data mapped. Please remove the port terminator first.

If you want to use some internal rules other than name matching to map communication elements to system signals you can apply this rules to sort a list of communication elements and a list of abstract signal instances and then use the simple API below that maps them via index.

`communicationElements(List)` wraps `SICommunicationElement`s to do further operations on them, e.g. map them to system signals.

`mapTo(List)` maps the `SICommunicationElement`s which are represented by this `ISelectedCommunicationElements` to the given `SIAbstractSignalInstance`s via index matching. So the first communication element will be mapped to the first abstract signal instance, the second to the second abstract signal instance and so on.

In case you want to ignore communication element and system signal pairs for which a mapping does already exist please use `assureMappedTo(List)`.

If you want to map your communication elements to system signals using name matching please use `IRuntimeSystemApi.selectCommunicationElements(com.vector.cfg.util.function.Action)`.

If you want to create complex mappings (currently only SenderReceiverToSignalGroupMappings) the given abstractSignalInstances should contain the group signals right after the parent signal group instances.

For a client server to signal mapping, select call and return signal instance directly after each other.

`assureMappedTo(List)` does the same as `mapTo(List)` but ignores communication element and abstract signal instance pairs for which a data mapping does already exist.

If you want to create complex mappings (currently only SenderReceiverToSignalGroupMappings) the given abstractSignalInstances should contain the group signals right after the parent signal group instances.

For a client server to signal mapping, select call and return signal instance directly after each other.

Examples

```
scriptTask ("createSimpleMapping", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                // specify path to the system signal
                def signalPath = "/VectorAutosarExplorerGeneratedObjects/
                    SYSTEM_SIGNALS/Element_1_b16df82332bcf915"

                // enter the fully qualified communication element name
                // that means ComponentName.PortName.DataElementName
                def communicationElementName = "App1.pDataSend.Element"

                def createdMapping = communicationElement(communicationElementName
                    ).mapTo(signalPath)

                scriptLogger.info("Mapped '{0}' to '{1}'.",
                createdMapping.getCommunicationElement().getFullyQualifiedName(),
                createdMapping.getSystemSignal().getAutosarPath())
            }
        }
    }
}
```

Listing 6.273: Create sender receiver to signal mapping

```
scriptTask ("mapDelegationPort", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def signalPath = "/VectorAutosarExplorerGeneratedObjects/
                    SYSTEM_SIGNALS/Element_1_b16df82332bcf915"

                // for delegation ports use 'ECU Composition' instead of the
                    component name
                def communicationElementName = "ECU Composition.pDelegationSRPort2
                    .Element"

                def createdMapping = communicationElement(communicationElementName
                    ).mapTo(signalPath)

                scriptLogger.info("Mapped '{0}' to '{1}'.",
                createdMapping.getCommunicationElement().getFullyQualifiedName(),
                createdMapping.getSystemSignal().getAutosarPath())
            }
        }
    }
}
```

Listing 6.274: Create data mapping for delegation port

```
scriptTask ("createClientServerMapping", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def callSignalPath = "/VectorAutosarExplorerGeneratedObjects/
                    SYSTEM_SIGNALS/rSRPort2_d4aecc362f1feef3"
                def returnSignalPath = "/VectorAutosarExplorerGeneratedObjects/
                    SYSTEM_SIGNALS/pSRPort3_2264a06bc04fc81d"

                // if an operation communication element is selected, a client
                    server to signal mapping will be created
                // the assignment of call and return signal role is depending on
                    the direction of the signal
                def communicationElementName = "ECU Composition.pDelegationCSPort1
                    .Operation"

                def createdMapping = communicationElement(communicationElementName
                    ).mapTo(callSignalPath, returnSignalPath)

                scriptLogger.info("Mapped '{0}' to '{1}' as call signal and '{2}'
                    as return signal.",
                createdMapping.getCommunicationElement().getFullyQualifiedName(),
                createdMapping.getSystemSignal().getAutosarPath(),
                createdMapping.getReturnSystemSignal().getAutosarPath())
            }
        }
    }
}
```

Listing 6.275: Create client server to signal mapping

```
import com.vector.cfg.sysdesc.model.datamapping.SISenderReceiverDataMapping

scriptTask ("mapRecordToSignalGroup", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                // specify the path to the signal group
                def signalGroup = "/VectorAutosarExplorerGeneratedObjects/
                    SYSTEM_SIGNAL_GROUPS/Element_2_de8db6949370c6b4"

                // specify the paths to the group signals of the signal group
                def groupSignal1 = "/VectorAutosarExplorerGeneratedObjects/
                    SYSTEM_SIGNALS/ay1_48523fe229ba8c99"
                def groupSignal2 = "/VectorAutosarExplorerGeneratedObjects/
                    SYSTEM_SIGNALS/ay2_071a3305d39fcca4"
                def groupSignal3 = "/VectorAutosarExplorerGeneratedObjects/
                    SYSTEM_SIGNALS/ay3_84eba37e401eacd1"

                // the name of the root element
                def record = "ECU Composition.pDelegationSRPort1.Element_2"

                // the names of the child elements
                def recordElement1 = "ECU Composition.pDelegationSRPort1.Element_2
                    .RecordElement"
                def recordElement2 = "ECU Composition.pDelegationSRPort1.Element_2
                    .RecordElement_1"
                def recordElement3 = "ECU Composition.pDelegationSRPort1.Element_2
                    .RecordElement_2"

                // create the mapping, first argument should be the root element,
                    followed by the leaf elements for the child mappings
                // for the signals, first argument should be the signal group,
                    followed by the group signals

                // the mapping will be done using the given order
                // e.g. the first element (record) will be mapped to the signal
                    group (signalGroup),
                // the last record element (recordElement3) will be mapped to the
                    last group signal (groupSignal3)
                def createdMapping = communicationElement(record, recordElement1,
                    recordElement2, recordElement3)
                    .mapTo(signalGroup, groupSignal1, groupSignal2, groupSignal3)

                scriptLogger.info("Mapped '{0}' to '{1}'.",
                createdMapping.getCommunicationElement().getFullyQualifiedName(),
                createdMapping.getSystemSignal().getAutosarPath())

                // print info for the child mappings
                for (final SISenderReceiverDataMapping childMapping :
                    createdMapping.getChildDataMapping()) {
                    scriptLogger.info("Mapped '{0}' to '{1}'.",
                    childMapping.getCommunicationElement().getFullyQualifiedName()
                        ,
                    childMapping.getSystemSignal().getAutosarPath())
                }
            }
        }
    }
}
```

Listing 6.276: Map record to signal group

```
import com.vector.cfg.sysdesc.model.datamapping.SISenderReceiverDataMapping

scriptTask ("mapArrayToSignalGroup", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                // specify the path to the signal group
                def signalGroup = "/VectorAutosarExplorerGeneratedObjects/
                    SYSTEM_SIGNAL_GROUPS/Element_2_de8db6949370c6b4"

                // specify the paths to the group signals of the signal group
                def groupSignal1 = "/VectorAutosarExplorerGeneratedObjects/
                    SYSTEM_SIGNALS/ay1_48523fe229ba8c99"
                def groupSignal2 = "/VectorAutosarExplorerGeneratedObjects/
                    SYSTEM_SIGNALS/ay2_071a3305d39fcca4"
                def groupSignal3 = "/VectorAutosarExplorerGeneratedObjects/
                    SYSTEM_SIGNALS/ay3_84eba37e401eacd1"

                // the name of the root element
                def array = "ECU Composition.pDelegationSRPort2.Element_1"

                // select the element of the array using the position index of the
                     element
                def arrayElement1 = "ECU Composition.pDelegationSRPort2.Element_1
                    [0]"
                def arrayElement2 = "ECU Composition.pDelegationSRPort2.Element_1
                    [1]"
                def arrayElement3 = "ECU Composition.pDelegationSRPort2.Element_1
                    [2]"

                def createdMapping = communicationElement(array, arrayElement1,
                    arrayElement2, arrayElement3)
                    .mapTo(signalGroup, groupSignal1, groupSignal2, groupSignal3)

                scriptLogger.info("Mapped '{0}' to '{1}'.",
                createdMapping.getCommunicationElement().getFullyQualifiedName(),
                createdMapping.getSystemSignal().getAutosarPath())

                // print info for the child mappings
                for (final SISenderReceiverDataMapping childMapping :
                    createdMapping.getChildDataMapping()) {
                    scriptLogger.info("Mapped '{0}' to '{1}'.",
                    childMapping.getCommunicationElement().getFullyQualifiedName()
                        ,
                    childMapping.getSystemSignal().getAutosarPath())
                }
            }
        }
    }
}
```

Listing 6.277: Map array to signal group

```
import com.vector.cfg.sysdesc.model.datamapping.SISenderReceiverDataMapping

scriptTask("completeComplexDataMapping", DV_PROJECT) {
    code {
        transaction {
            domain.runtimeSystem {

                // in this example we want to map a record to a system signal
                    group
                // we will map only the roots
                // a comfort function will try to find matches in record elements
                    and group signals via naming

                String recordName = "ECU Composition.pDelegationSRPort1.Element_2"
                String signalGroupPath = "/VectorAutosarExplorerGeneratedObjects/
                    SYSTEM_SIGNAL_GROUPS/Element_2_de8db6949370c6b4"

                SISenderReceiverDataMapping createdDataMapping =
                    communicationElement(recordName).mapTo(signalGroupPath)

                scriptLogger.info("Mapped {0} -> {1}.",
                createdDataMapping.getCommunicationElement().getFullyQualifiedName
                    (),
                createdDataMapping.getSystemSignal().getName())

                // we want also to print info for the child mappings
                for (SISenderReceiverDataMapping childMapping in
                    createdDataMapping.getLeafDataMappings()) {
                    scriptLogger.info("Mapped {0} -> {1}.",
                    childMapping.getCommunicationElement().getFullyQualifiedName()
                        ,
                    childMapping.getSystemSignal().getName())
                }
            }
        }
    }
}
```

Listing 6.278: Map complex data element to system signal group and let the auto-mapper complete the mapping if possible

```
import com.vector.cfg.sysdesc.model.communication.SICommunicationElement
import com.vector.cfg.sysdesc.model.datamapping.SIDataMapping
import com.vector.cfg.sysdesc.model.datamapping.SIClientServerToSignalDataMapping
import com.vector.cfg.sysdesc.model.communication.instance.
    SIAbstractSignalInstance

scriptTask("UseSimpleAPIToCreateMultipleDataMappings", DV_PROJECT) {
    code {
        transaction {
            domain.runtimeSystem {

                // in this example we know for each communication element the name
                    of the system signal to map it to
                // (for example stored in some external file)
                // so we use the simple API instead of the auto mapping

                List<String> comElementNames =
                    ["ECU Composition.TriggerDataMappingZeroLengthSignal.
                        SomeTrigger",
                    // com element below for client server to signal mapping
                    "App1_1.pCSPort1.Operation",
                    "App2.rSRPort1.Element",
                    // com elements below are a record and its record elements
                    "App2.rSRPort1.Element_2",
                    "App2.rSRPort1.Element_2.RecordElement",
                    "App2.rSRPort1.Element_2.RecordElement_2"]
                List<String> signalNames =
                    ["TriggerDataMappingZeroLengthSignal_896bde67e5a0f5b4",
                    // the two signals below are used for a client server to
                        signal mapping
                    // one call and one return signal
                    "rSRPort2_d4aecc362f1feef3", "pSRPort3_2264a06bc04fc81d",
                    "RElement_1_c07c9ba68bc545ba",
                    // com elements below is a signal group and its group signals
                    "elemB_c255f5e38fd8b21d",
                    "fieldA_f1d3783e235e88d3",
                    "fieldB_344fdc16e87cfdaa"]

                // we use the communication element selection to retrieve the
                    communication elements
                // for the client server to signal mapping the selection will find
                    two com elements for the one name
                // one for the call direction and one for the return direction
                List<SICommunicationElement> comElements =
                    selectCommunicationElements {
                    fullyQualifiedNames(comElementNames)
                    // we need to select also unmapped record elements, so use
                        fully expanded selection
                    // another way would be to select only the root
                    // and then access the leafs e.g. via
                        IDataCommunicationElement.
                        getLeafsFullExpandedExceptPrimitiveArrays()
                    selectFullyExpanded()
                }.getCommunicationElements()

                // since our used predicate does not guarantee any order, we have
                    to sort our communication elements to assure they are in
                    correct order
                comElements.sort{a,b -> comElementNames.indexOf(a.
                    getFullyQualifiedName()) <=> comElementNames.indexOf(b.
                    getFullyQualifiedName())}
```

Listing 6.279: Map communication elements to system signals using simple API Part1

```
            // now select and sort the signal instances
            List<SIAbstractSignalInstance> signals = new ArrayList<
                SIAbstractSignalInstance>(selectSignalInstances {
                 names(signalNames)
            }.getSignalInstances())

            signals.sort{a,b -> signalNames.indexOf(a.getName()) <=> signalNames
                .indexOf(b.getName())}

            // map communication elements via index to the system signals
            // for complex mappings the children need to be inserted directly
                after the parent

            // for client server mappings call and return elements right after
                each other
            // call first or return first does not matter

            // mapTo(...) would fail if one of the mappings does already exist
            // but in this example we just want to make sure that the mapping
                exist, if not assureMappedTo(...) will create it, if it does
                exist it will do nothing
            List<SIDataMapping> dataMappings = communicationElements(comElements
                ).assureMappedTo(signals)

            // finally do some reporting for the pairs that were previously not
                mapped to each other
            for (SIDataMapping dataMapping in dataMappings) {
                scriptLogger.info("Mapped {0} to {1}.",
                        dataMapping.getCommunicationElement().
                            getFullyQualifiedName(),
                        dataMapping.getSystemSignal().getName())

                if (dataMapping instanceof SIClientServerToSignalDataMapping) {
                    scriptLogger.info("Return signal is {0}.",
                            ((SIClientServerToSignalDataMapping) dataMapping).
                                getReturnSystemSignal().getName())
                }
            }
        }
    }
}
          }
```

Listing 6.280: Map communication elements to system signals using simple API Part2

6.10.4.13 Remove Data Mappings

The previous chapter was about mapping communication elements and signal instances. Now we want to have a look how to remove such mappings again.
This can be done using the communication element selection API (see 6.10.4.3 on page 182) or the signal instance selection API (see 6.10.4.2 on page 178) and calling a method to unmap the selected communication elements / signal instances.

Unmapping by Selecting Communication Elements The use case of unmapping communication elements is based on the selection of communication elements. The targets to be unmapped are signal instances, which can be also narrowed down by further closures.

`unmap()` unmaps the selected communication elements from all mapped system signals. In case that not all data mappings of the selected communication elements shall be removed the mapped signal instances can be narrowed down using `unmapFrom(Action)`.

For SenderReceiverToSignalGroupMappings (complex mappings) it is already enough to select one of the communication elements (for example the root element), all mappings belonging to the complex mapping will be removed. In other words removing the root mapping also removes the child mappings, removing a child mapping also removes the root mapping and the other child mappings of the same root.

Examples for unmap()

```
import com.vector.cfg.dom.runtimesys.pai.api.IUnmappedComElementAndSignalResult

scriptTask("UnmapCommunicationElements", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {

        // we want to remove all data mappings of delegation ports in this example
        def result = selectCommunicationElements {
                  mapped()
                  delegation()
                } unmap()


        // now print a detailed info for all removed data mappings
        // we want also detailed info for client server to signal mappings and
            sender receiver to signal group mappings
        scriptLogger.info("Removed {0} (root) mappings.", result.size())

        for (IUnmappedComElementAndSignalResult unmappedResult : result) {
            scriptLogger.info("{0} -> {1}",
                    unmappedResult.getCommunicationElement().getFullyQualifiedName
                        (),
                    unmappedResult.getSignalInstance().getName())

            // additional info for return signal mapping
            if (unmappedResult.getReturnSignalCommunicationElement() != null) {
                scriptLogger.info("{0} -> {1}",
                        unmappedResult.getReturnSignalCommunicationElement().
                            getFullyQualifiedName(),
                        unmappedResult.getReturnSignalInstance().getName())
            }

            //  additional info for mapped record and array elements
            if (!unmappedResult.getChildCommunicationElements().isEmpty()) {

                for (int i = 0; i < unmappedResult.getChildCommunicationElements()
                    .size(); i++) {
                    scriptLogger.info("{0} -> {1}",
                            unmappedResult.getChildCommunicationElements().get(i).
                                getFullyQualifiedName(),
                            unmappedResult.getGroupSignalInstances().get(i).
                                getName())
                }
            }
        }
      }
    }
  }
}
```

Listing 6.281: Remove Data Mapping of Communication Element

Control unmapping in unmapFrom(Closure)

`selectTargetSignalInstances(Action)` allows to define predicates to narrow down the target signal instances to be unmapped from the previously selected communication elements.

`evaluateMatches(IUnmappingEvaluator)` allows to evaluate and change the results of the communication elements which are about to be unmapped from signal instances.

For each selected communication element the provided lambda is called: Parameters are the cur-

rent handled communication element and a list of all mapped signal instances (respecting the `selectTargetSignalInstances(Action)` predicates). The return value must be a list of signal instances which are mapped to the current handled communication element.

Examples for unmapFrom(Closure)

```
scriptTask("UnmapCommunicationElementsAdvanced", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {

        // we want to remove only data mappings for delegation ports to signals of
            a special frame
        def result = selectCommunicationElements {
                  mapped()
                  delegation()
                } unmapFrom {
                    // we have already filtered the communication elements
                    // in this closure we can now additionally filter also for the
                        signals of the data mapping to be removed
                    selectTargetSignalInstances {
                        frame("MyFrame")
                    }
                }


        // see the simple example above how to print more detailed info
        scriptLogger.info("Removed {0} (root) mappings.", result.size())
      }
    }
  }
}
```

Listing 6.282: Remove Data Mapping of Communication Element Considering Signals

Unmapping by Selecting Signal Instances The use case of unmapping system signals is based on the selection of signal instances. The targets to be unmapped are communication elements, which can be also narrowed down by further closures.

`unmap()` unmaps the selected signal instances from all mapped communication elements. In case that not all data mappings of the selected signal instances shall be removed the mapped communication elements can be narrowed down using `unmapFrom(Action)`.

For SenderReceiverToSignalGroupMappings (complex mappings) it is already enough to select the signal group or one of the group signals, all mappings belonging to the complex mapping will be removed. In other words removing the root mapping also removes the child mappings, removing a child mapping also removes the root mapping and the other child mappings of the same root.

Examples for unmap()

```
import com.vector.cfg.dom.runtimesys.pai.api.IUnmappedComElementAndSignalResult

scriptTask("UnmapSignalInstances", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {

        // we want to remove all data mappings of all mapped tx signals in this
            example
        def result = selectSignalInstances {
                  mapped()
                  tx()
                } unmap()


        // now print a detailed info for all removed data mappings
        // we want also detailed info for client server to signal mappings and
            sender receiver to signal group mappings
        scriptLogger.info("Removed {0} (root) mappings.", result.size())

        for (IUnmappedComElementAndSignalResult unmappedResult : result) {
            scriptLogger.info("{0} -> {1}",
                    unmappedResult.getCommunicationElement().getFullyQualifiedName
                        (),
                    unmappedResult.getSignalInstance().getName())

            // additional info for return signal mapping
            if (unmappedResult.getReturnSignalCommunicationElement() != null) {
                scriptLogger.info("{0} -> {1}",
                        unmappedResult.getReturnSignalCommunicationElement().
                            getFullyQualifiedName(),
                        unmappedResult.getReturnSignalInstance().getName())
            }

            //  additional info for mapped record and array elements
            if (!unmappedResult.getChildCommunicationElements().isEmpty()) {

                for (int i = 0; i < unmappedResult.getChildCommunicationElements()
                    .size(); i++) {
                    scriptLogger.info("{0} -> {1}",
                            unmappedResult.getChildCommunicationElements().get(i).
                                getFullyQualifiedName(),
                            unmappedResult.getGroupSignalInstances().get(i).
                                getName())
                }
            }
        }
      }
    }
  }
}
```

Listing 6.283: Remove Data Mapping of Signal

Control unmapping in unmapFrom(Closure)

`selectTargetCommunicationElements(Action)` allows to define predicates to narrow down the target communciation elements to be unmapped from the previously selected signal instances.

`evaluateMatches(IUnmappingEvaluator)` allows to evaluate and change the results of the signal instances which are about to be unmapped from communication elements.

For each selected signal instance the provided lambda is called: Parameters are the current handled signal instance and a list of all mapped communication elements (respecting the `selectTargetCommunicationElements(Action)` predicates). The return value must be a list of communication elements which are mapped to the current handled signal instance.

Examples for unmapFrom(Closure)

```
scriptTask("UnmapSignalInstancesAdvanced", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {

        // we want to remove only data mappings for transformed signals which are
            mapped to communication elements of component 'App1'
        def result = selectSignalInstances {
                  mapped()
                  transformed()
                } unmapFrom {
                    // we have already filtered the signal instances
                    // in this closure we can now additionally filter also for
                        communication elements of the data mapping to be removed
                    selectTargetCommunicationElements {
                        component("App1")
                    }
                }


        // see the simple example above how to print more detailed info
        scriptLogger.info("Removed {0} (root) mappings.", result.size())
      }
    }
  }
}
```

Listing 6.284: Remove Data Mapping of Signals Considering Communication Elements

6.10.4.14 Configure RTE Implementation Plug-ins

RTE implementation plug-ins (RIPs) can be configured directly at the communication element (recommended) or using the according methods at the communication element selection. They are stored in the flat map at the entry of the flat instances descriptor which belongs to the communication element. It is a reference to the according ECUC container representing the RIP. When setting a RIP the flat instance descriptor will be created automatically if missing. The getters and setters at the ICommunicationElement are not explicitly listed here. At which communication elements the references need to be configured is defined in [SWS_Rte_CONSTR_80002].

`mapToRIPs(Action)` offers the possibility to map the selected communication elements to RTE implementation plug-ins or to modify the name of their flat instance descriptors.

`setLocalRIPsReference(String)` sets the RTE implementation plug-in reference for local communication (associatedRtePlugin reference).

`setCrossClusterRIPsReference(String)` sets the RTE implementation plug-in reference for cross-cluster communication (associatedCrossSwClusterComRtePlugin reference).

`name(String)` sets the name of the flat instance descriptor referencing the communication element.

`name(Function)` allows to define a function which maps the communication element to the flat instance descriptor name.

`useCommunicationGraph()` extends the selection of each communication element by its whole communication graph. Communication graph means all required and provided ports taking part in the communication. For example in a 1:n connection it would be the provider port and all n required ports.

`removeLocalRIPsReference()` removes the associatedRtePlugin references of the selected communication elements, so that they do not use an RTE implementation plug-in for local communication anymore.

`removeCrossClusterRIPsReference()` removes the associatedCrossSwClusterComRtePlugin references of the selected communication elements, so that they do not use an RTE implementation plug-in for cross-cluster communication anymore.

`deleteFlatInstanceDescriptors()` deletes the flat instance descriptors of the selected communication elements.

Since the RIP references are stored at the flat instance descriptor they might not always be changeable. This can be checked via the according methods at the SICommunicationElement (SICommunicationElement.isAssociatedLocalClusterRtePluginReferenceChangeable(), SICommunicationElement.isAssociatedCrossClusterRtePluginReferenceChangeable(), SICommunicationElement.isFlatInstanceDescriptorNameChangeable() and SICommunicationElement.isFlatInstanceDescriptorDeletable()).

There are also methods available at the runtime system API to get the according ECUC containers which can be referenced as RIP.

`getAvailableLocalRIPsContainers()` collects and returns all containers which are associated with RTE implementation plug-ins (RIPs) for local communication.

`getAvailableCrossClusterRIPsContainers()` collects and returns all containers which are associated with RTE implementation plug-ins (RIPs) for cross-cluster communication.

Examples

```
scriptTask("ConfigureRIPs", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
          def ripContainers = getAvailableLocalRIPsContainers()
          String ripContainerPath = AsrPath.create(ripContainers.first).toString()

          def comElementsWithRIP = selectCommunicationElements {
              component("App1_1")
              port("pDataWrite")
              name("Element")
          }.mapToRIPs {
              // set the RTE implementation plug-in
              setLocalRIPsReference(ripContainerPath)
              // there are also according methods to set the cross-cluster plug-in
              // and to modify the name of the created flat instance descriptor
          }
          scriptLogger.info("Mapped {0} communication elements to RTE
              implementation plug-in {1}.",
            comElementsWithRIP.size(),
            ripContainerPath)
       }
    }
  }
}
```

Listing 6.285: Configure RTE implementation plug-ins

```
scriptTask("RemoveRIPs", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
          def comElementsRemovedRIPs = selectCommunicationElements {
              component("App1")
          }.removeLocalRIPsReference()
          // it is also possible to remove the flat instance descriptor or the
              cross-cluster reference

          scriptLogger.info("Removed RTE implementation plug-ins of {0}
              communication elements.", comElementsRemovedRIPs.size())
      }
    }
  }
}
```

Listing 6.286: Remove RTE implementation plug-ins

```
import com.vector.cfg.model.mdf.model.autosar.ecucdescription.MIContainer

scriptTask("RemoveRIPOfComElement", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
          def selectedComElements = selectCommunicationElements {
              component("App1")
          }.getCommunicationElements()

          // the communication element has getters and setter which can be also
              used
          // check if comElement has a RIP reference and remove it if it is
              changeable
          def comElement = selectedComElements.first
          MIContainer ripContainer = comElement.getAssociatedLocalClusterRtePlugin
              ()
          if (ripContainer != null && comElement.
              isAssociatedLocalClusterRtePluginReferenceChangeable()) {
             comElement.setAssociatedLocalClusterRtePluginReference(null)
          }
       }
    }
  }
}
```

Listing 6.287: Access RTE implementation plug-ins directly at the communication element

6.10.4.15 Create Component Prototypes

In the create component prototypes use case, components can be instantiated after a component type was selected. So the entry point is the component type selection (see 6.10.4.4 on page 186).

Instantiate Components `createPrototype()` creates a SwComponentPrototype in the StructuredEXtract for each selected component type. The names of the created SwComponentPrototypes are derived from the selected component types.

Examples for createPrototype()

```
scriptTask ("createComponentPrototypesForNotInstantiatedTypes", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def createdComponents = selectComponentTypes {
                    not {
                        instantiated()
                    }
                }.createPrototype()

                scriptLogger.info("Created '{0}' component prototypes.",
                    createdComponents.size())
            }
        }
    }
}
```

Listing 6.288: Create component prototypes for not instantiated types

Specify the component prototype instantiation in createPrototypeWith(Closure)

`IComponentPrototypeCreator` provides an Api to control some aspects, e.g. the naming, of newly created components.

- `name(Function)` computes a name for the component prototypes that should be created for, by the `IComponentTypeSelection` provided, component types.
- `count(int)` defines how many component prototypes should be created for each selected component type. The default is 1.

Examples for customizing the instantiation

```
import com.vector.cfg.sysdesc.model.component.SIComponentType

scriptTask ("specifyNameOfCreatedComponent", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def createdComponents = selectComponentTypes {
                    application()
                }.createPrototypeWith {
                    name {
                        // define the naming of new created prototypes
                        SIComponentType<?> type -> type.getName() + "_postfix"
                    }
                }

                scriptLogger.info("Created '{0}' component prototypes.",
                    createdComponents.size())
            }
        }
    }
}
```

Listing 6.289: Specify name of created component

```
import com.vector.cfg.sysdesc.model.component.SIComponentType

scriptTask ("specifyNameOfCreatedComponent", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def createdComponents = selectComponentTypes {
                    name ~"App.*"
                }.createPrototypeWith {
                    name {
                        // you can still define a naming pattern
                        SIComponentType<?> type -> type.getName() + "_CP"
                    }

                    // and at the same time define how many prototypes should be
                        created for each component type
                    count(3)
                }

                scriptLogger.info("Created '{0}' component prototypes.",
                    createdComponents.size())
            }
        }
    }
}
```

Listing 6.290: Create more than 1 component prototype

6.10.4.16 Create Delegation Ports

The automation interface offers a possibility to create delegation ports at the TopLevel-Composition of the StructuredExtract. Therefore a port interface needs to be selected and a direction specified. Alternatively the component port and the origin port selections offer also APIs for that use case. For the origin context based creation of delegation ports see 6.10.4.16 on page 254 below, the component port based creation of delegation ports can be found in the examples at the end of following chapter (6.10.4.16 on page 253).

The entry points are the port interface selection (see 6.10.4.7 on page 193), the component port selection (see 6.10.4.1 on page 172) or the origin component port selection (see 6.10.4.8 on page 197).

Instantiate Delegation Ports `createPrototype(Action)` creates a PortPrototype on the ecu composition of the StructuredExtract for each selected port interface. If the naming is not specified, the names of the created delegation ports are derived from the selected port interfaces. The direction of the ports has to be specified.

Specify the delegation port prototype instantiation

`IDelegationPortCreator` provides an Api to control some aspects, e.g. the naming or the direction, of newly created delegation ports.

- `name(Function)` computes a name for the delegation port prototypes that should be created for port interfaces, which are provided by the used selection API.
- `direction(EDirection)` defines the direction of the port prototype that should be created. `EDirection.Tx` will create provided ports, `EDirection.Rx` will create required ports. Delegation provided-required ports are not supported.

- `count(int)` defines how many delegation port prototypes should be created for each selected port interface. The default is 1.

For the origin and component port selection APIs the naming can be also done based on the selected ports.

- `nameFromOriginPort(Function)` computes a name for the delegation port prototypes that should be created for origin ports, which are provided by the used `IOriginComponentPortSelection`.
- `nameFromComponentPort(Function)` computes a name for the delegation port prototypes that should be created for component ports, which are provided by the used `IComponentPortSelection`.

Examples

```
import com.vector.cfg.sysdesc.model.communication.EDirection

scriptTask("createDelegationPortSimple", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def createdComponentPorts =
             selectPortInterfaces {
                // select all client server application port interfaces of
                    component 'App3'
                componentType "App3"
                clientServer()
                application()
             } createPrototype {
                // EDirection.Tx to create a PPort and EDirection.Rx to create a
                    RPort
                direction(EDirection.Tx)
                // if no name is specified, the name of the port interface will be
                     taken for the port
              }
        scriptLogger.info("Created {0} delegation ports.", createdComponentPorts.
            size())
      }
    }
  }
}
```

Listing 6.291: Create delegation port simple

```
import com.vector.cfg.sysdesc.model.communication.EDirection
import com.vector.cfg.sysdesc.model.port.SIPortInterface

scriptTask("createDelegationPortCustomized", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        def createdComponentPorts =
             selectPortInterfaces {
                // select the port interface of 'App1.ppFirst' and the port
                    interface named 'Second'
                or {
                    componentPort "App1.ppFirst"
                    name "Second"
                }
             } createPrototype {
                name {
                    // specify the naming of the new ports
                    SIPortInterface<?> portInterface -> "pp" + portInterface.
                        getName() + "_new"
                }
                // EDirection.Rx is leading to the creation of required ports
                direction(EDirection.Rx)
                // for each selected port interface two delegation ports will be
                    created
                count(2)
              }
        scriptLogger.info("Created {0} delegation ports.", createdComponentPorts.
            size())
      }
    }
  }
}
```

Listing 6.292: Create delegation port customized

```
import com.vector.cfg.sysdesc.model.component.SIComponentPort

scriptTask("createDelegationPortFromComponentPort", DV_PROJECT){

  // in this example we want to create a delegation port for an existing SWC port

  code {
    transaction {
      domain.runtimeSystem {
        def createdComponentPorts =
             selectComponentPorts {
                 component "App3"
                 name "rOtherSR1Port"
             } createDelegationPorts {

                // we can use the selected component port to specify the name of
                    the new port
                // also we could have used the name(Closure) method from examples
                    above and use the port interface for naming
                nameFromComponentPort {
                    // specify the naming of the new ports
                    SIComponentPort componentPort -> componentPort.getPortName() +
                        "_new"
                }
                // since we have selected a component port previously, we can now
                    use the direction of it
                // of course it is also possible to use direction(EDirection) here
                     as in the examples above
                useDefaultDirection()
              }
        scriptLogger.info("Created {0} delegation ports.", createdComponentPorts.
            size())
      }
    }
  }
}
```

Listing 6.293: Create delegation port based on existing component port

Create Delegation Ports Using Origin Context We have already learned how to create new delegation ports in the flat extract using the port interface selection (see 6.10.4.16 on page 251). Sometimes a delegation connection was not completed in the structured extract because the delegation port was missing and is not flattened out. To complete this connection in the flat extract we need a new delegation port and want to use exactly the same port interfaces as in the structured extract. For this use case there is a shortcut directly at the origin component port selection, that simplifies the transition from origin port to its port interface. To specify the name of the new port, the API offers to do it using the port interface or the origin component port itself (see examples below).

For more details about origin context see 6.10.4.8 on page 196.

`createFlatExtractDelegationPorts(Action)` creates a PortPrototype on the ecu composition of the FlatExtract for each selected origin component port using their port interfaces. If the naming is not specified, the names of the created delegation ports are derived from the port interfaces. The direction of the ports has to be specified.

Examples

```
import com.vector.cfg.sysdesc.model.communication.EDirection

scriptTask("createPortFromOriginContext", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {

        // in this example we create a delegation port which is missing
        // the port interface for our new port is provided by an origin context
            port

        def selectedOriginPorts
        def createdDelegationPorts = selectOriginComponentPorts {
            provided()
            name "OriginContext"

            // remember the selected ports to print better info later
            selectedOriginPorts = getSelectedOriginComponentPorts()

        }.createFlatExtractDelegationPorts {
            // this call retrieves the port interfaces of the selected origin
                component ports
            // and creates for each origin component port a delegation port

            // specify the name and direction of the new port
            name {
                // optionally you could use the port interface as help for
                    specifying the name here
                // IPortInterface portInterfaceOfOriginPort -> ...
                "PortWithInterfaceOfOriginContext"
            }
            direction(EDirection.Tx)
        }

        for (int i = 0; i < selectedOriginPorts.size(); i++) {
            scriptLogger.info("Created new delegation port {0} for origin port
                {1}.",
                createdDelegationPorts.get(i).getName(),
                selectedOriginPorts.get(i).getName())
        }
      }
    }
  }
}
```

Listing 6.294: Create a delegation port using the port interface of an origin context port

```
import com.vector.cfg.sysdesc.model.component.origin.SIOriginComponentPort

scriptTask("createPortFromOriginContextUsingPort", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {

        // in this example we create a delegation port which is missing
        // the port interface for our new port is provided by an origin context
            port
        // and we use the original port for naming and direction this time

        def selectedOriginPorts
        def createdDelegationPorts = selectOriginComponentPorts {
            provided()
            name "OriginContext"

            // remember the selected ports to print better info later
            selectedOriginPorts = getSelectedOriginComponentPorts()

        }.createFlatExtractDelegationPorts {

            // specify the name and direction of the new port
            nameFromOriginPort {  SIOriginComponentPort originPort ->
                originPort.getPortName() + "_new"
            }
            // for the origin component port selection we can use the direction of
                the previously selected origin ports
            useDefaultDirection()
        }

        for (int i = 0; i < selectedOriginPorts.size(); i++) {
            scriptLogger.info("Created new delegation port {0} for origin port
                {1}.",
                createdDelegationPorts.get(i).getName(),
                selectedOriginPorts.get(i).getName())
        }
      }
    }
  }
}
```

Listing 6.295: Create a delegation port using the origin context port to specify name and direction

6.10.4.17 Task Mapping

The task mapping use case allows to map executable entities (also called functions) directly or using their events (also called triggers) to tasks.

The entry point for the task mapping is either to select events (see 6.10.4.5 on page 187) or executable entities (see 6.10.4.6 on page 191). After that a task can be selected and the task mappings customized.

Mapping to a Task

Event selection `mapToTask(Action)` tries to perform a task mapping for the selection of events (triggers). Inside the lambda the task mapping can be controlled, e.g. selecting the task to which the events should be mapped to and order the event's positions. Does not consider events (triggers) which do not reference an executable entity (function).

`unmapTaskMappings(Action)` performs the unmapping of task mappings from OSTasks for the selection of events (triggers).

ExecutableEntity selection `mapToTask(Action)` tries to perform a task mapping for the selection of executable entities (functions). Inside the lambda the task mapping can be controlled, e.g. selecting the task to which the events (triggers) of the selected executable entities should be mapped to and order the event's positions.

`unmapTaskMappings(Action)` performs the unmapping of task mappings from OSTasks for the selection of executable entities (functions).

Select a task Exactly one task has to be selected to perform a task mapping. Since the task selection is only available for the task mapping use case there is no own chapter for it.

`selectTask(Action)` allows to define predicates to select a task for the task mapping.

Per default the predicates are combined via logical AND. To realize other combinations, use the 'or','not' and 'and' predicates.

Task Predicates

- `name(String)` matches tasks with the given task name.
- `name(Pattern)` matches tasks with the given task name pattern.
- `core(String)` matches tasks running on a core with the given name / number (whether a core name or a core number is used, depends on the OS, if core number is used the String to be matched is 'Core<number>', e.g. 'Core1').
- `core(Pattern)` matches tasks running on a core with the given name pattern / number pattern (whether a core name or a core number is used, depends on the OS, if core number is used the String to be matched is 'Core<number>', e.g. 'Core1').
- `application(String)` matches tasks which belong to an application with the given name.
- `application(Pattern)` matches tasks which belong to an application whose name matches the given name pattern.
- `numberOfTaskMappings(int)` matches tasks which already have the given number of task mappings. The predicate can also be used to search for empty tasks with '0' as argument.
- `priority(BigInteger)` matches tasks with the given priority value.
- `filterAdvanced(Predicate)` matches tasks for which the given predicate results to true.
- `and(Runnable)` combines the predicates inside the lambda with a logical AND.
- `or(Runnable)` combines the predicates inside the lambda with a logical OR.
- `not(Runnable)` negates the combination of predicates inside the lambda.

Examples for mapToTask(Closure)

```
import com.vector.cfg.sysdesc.model.taskmapping.SITaskMapping

scriptTask ("doTaskMappingOfApp1", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def taskMappings = selectEvents {
                    // select all events of component App1
                    component("App1")
                } mapToTask {
                    selectTask {
                        // select a task
                        name("OsTask")
                    }
                }

                scriptLogger.info("Created '{0}' task mappings.", taskMappings.
                    size())

                // let's print more information to check the created task mappings
                for (SITaskMapping taskMapping : taskMappings) {
                    scriptLogger.info("Mapped '{0}' triggered by '{1}' to position
                         '{2} on task '{3}'.",
                        taskMapping.getExecutableEntity().getName(),
                        taskMapping.getEvent().getName(),
                        taskMapping.getPositionInTask(),
                        taskMapping.getMappedTask().getName())
                }
            }
        }
    }
}
```

Listing 6.296: Perform task mapping example

```
import com.vector.cfg.sysdesc.model.internalbehavior.SIEvent
import com.vector.cfg.model.mdf.ar4x.swcomponenttemplate.swcinternalbehavior.
    rteevents.MIDataReceivedEvent

scriptTask ("advancedFilterForEvents", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                    def taskMappings = selectEvents {
                        // use advanced filter if you cannot find a suitable
                            predicate
                        filterAdvanced { SIEvent event ->
                           // use the mdf model if the SIEvent does not offer
                               required methods
                           def mdfEvent = event.getMdfObject()

                           // for example, filter for data received events with
                               special criteria
                           if (mdfEvent instanceof MIDataReceivedEvent) {
                               // filter here for the special criteria
                               return true
                           }

                           // do not select other events
                           return false
                        }
                } mapToTask {
                    selectTask {
                        name("OtherName")
                    }
                }

                scriptLogger.info("Created '{0}' task mappings.", taskMappings.
                    size())
            }
        }
    }
}
```

Listing 6.297: Advanced Filter for Events

Examples for unmapTaskMappings(Closure)

```
scriptTask ("unmapTaskMappings", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def unmappedTaskMappings = selectEvents {
                    // define predicate for event selection
                    task("OsTask")
                    componentType("App1")
                } unmapTaskMappings {
                    filterTaskMappings {
                        // define further task mapping predicates
                        // in our example we have multi-instantiation of SWCs
                        // and want to unmap only one of them.
                        component "App1_1"
                    }
                }
                scriptLogger.info("Unmapped '{0}' task mappings.",
                    unmappedTaskMappings.size())
            }
        }
    }
}
```

Listing 6.298: Unmaps task mappings

Additional Comfort Functions The API provides some comfort functions listed below.

Combine via Symbol `combineViaSymbol(boolean)` determines whether the BswModuleEntities and the RunnableEntities should be combined using their symbol. That means they will be mapped to the same position on the same task. It is enough to select only the RunnableEntity or only the BswModuleEntity, when using this option both will be mapped. The default is true.

The condition is that the symbol of a RunnableEntity and the BswModuleEntry short name of a BswModuleEntity are equal.

Example

```
scriptTask ("combineViaSymbol", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def taskMappings = selectEvents {
                    component("Service1")
                    timing()
                } mapToTask {
                    selectTask {
                        name("OtherName")
                    }
                    // the default is true
                    // call this if you do not want to combine runnables and bsw
                        module entities via their symbol
                    combineViaSymbol(false)
                }

                scriptLogger.info("Created '{0}' task mappings.", taskMappings.
                    size())
            }
        }
    }
}
```

Listing 6.299: Do not combine runnable and bsw module entity via symbol

Map Events of a Runnable Entity Together `mapAllEventsOfRunnableEntity(boolean, boolean)` is a possibility to map all events of a RunnableEntity to the same position on a task. In case of the selection of events, the task mapping will be extended, by all events (triggers) of runnable entities (functions) for which at least one event (trigger) is selected.

With help of the two boolean arguments, the behavior of ignoring already mapped events and ignoring events whose mapping is optional can be controlled.

Example

```
scriptTask ("mapAllEventsOfRunnable", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def taskMappings = selectEvents {
                    name("background_event")
                    component("App1_1")
                } mapToTask {
                    selectTask {
                        name("OsTask")
                    }
                    // decide whether to consider only unmapped events
                    // and whether to consider only events whose mapping is
                        mandatory
                    mapAllEventsOfRunnableEntity(true, false)
                }

                scriptLogger.info("Created '{0}' task mappings.", taskMappings.
                    size())
            }
        }
    }
}
```

Listing 6.300: Map all events of a runnable together

Order Task Mappings by Defining Successor Mappings If you do not care about the absolute position value of the task mappings, but want to define an order, there is an option to specify successor relationships between the selected task mappings. This is the preferred way to define an order to using order(Closure) which will be introduced below, since it is easier to use in most cases. One exception is for example if you already read in a sorted list of executable entity names from external files.

`defineSuccessors(Action)` allows to specify the order of the selected task mappings by defining successor relationships between the task mappings. The order defined by this method may still be overridden by `order(Consumer)` and `queue()` if used.

First you have to specify one task mapping as the starting point.

`forTaskMapping(Action)` allows to select a starting task mapping for which successors can be defined.

After that you can define a direct successor or a (logical) successor for the previously selected task mapping.

`successor(Action)` allows to select a successor for the previously selected task mapping of the sequence. The sequence can be continued by another `successor(Action)` or `directSuccessor(Action)` call. To start a new sequence call `ITaskMappingSuccessorDefinition.forTaskMapping(Action)`

`directSuccessor(Action)` allows to select a direct successor for the previously selected task mapping of the sequence. The sequence can be continued by another `successor(Action)` or `directSuccessor(Action)` call. To start a new sequence call `ITaskMappingSuccessorDefinition.forTaskMapping(Action)`.

Within these calls you can select task mappings using task mapping predicates (see 6.10.4.17 on page 272). If the task mappings cannot be ordered to match the defined rules (for example if cycles were defined), an exception is thrown.

Example

```
import com.vector.cfg.sysdesc.model.taskmapping.SITaskMapping

scriptTask ("defineSuccessorsSimple", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {

                // you can use both the event selection or the executable entity
                    selection API here
                def taskMappings = selectEvents {
                    component("App1")
                } mapToTask {
                    selectTask {
                        name("OsTask")
                    }
                    defineSuccessors {
                        forTaskMapping {
                            // define task mapping predicates to select the start
                                of your sequence
                            // here we really use the predicates for task mappings
                                , not for events nor executable entities
                            // but the task mapping selection offers us a lot of
                                predicates
                            // it allow us to filter e.g. for the events which the
                                 task mappings reference or the executable entity
                                which is triggered by the referenced event
                            executableEntity "Runnable1"
                        } successor {
                            // now define predicates to select successors for
                                Runnable1
                            executableEntity "Runnable2"
                        } successor {
                            // we can continue by defining the successors for
                                Runnable2 now
                            externalTrigger()
                        }
                    }
                }

                // so the result on OsTask will be Runnable1 -> Runnable2 -> all
                    runnables triggered by external trigger events

                scriptLogger.info("Created '{0}' task mappings.", taskMappings.
                    size())

                for (final SITaskMapping taskMapping : taskMappings) {
                    scriptLogger.info("Mapped runnable {0} to position {1}.",
                    taskMapping.getExecutableEntity().getName(),
                    taskMapping.getPositionInTask())
                }
            }
        }
    }
}
```

Listing 6.301: Order the task mappings by defining successor relationships

```
import com.vector.cfg.sysdesc.model.taskmapping.SITaskMapping

scriptTask ("successorForGroup", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {

                def taskMappings = selectExecutableEntities {
                    runnableEntity()
                } mapToTask {
                    selectTask {
                        name("OsTask")
                    }
                    defineSuccessors {
                        // you do NOT have to narrow the selection down to one
                            task mapping
                        // we want to order the task mappings by their SWC owner
                            in this example

                        // we want the runnables of App1 be executed before the
                            runnables of App2 and App3
                        // but we do not care about any order for the runnables of
                             App2 and App3 or want define them later

                        // execute runnables of App1 before runnables of App2
                        forTaskMapping {
                            component "App1"
                        } successor {
                            component "App2"
                        }

                        // you can define multiple sequences for the same task
                            mappings
                        // execute runnables of App1 before runnables of App3
                        forTaskMapping {
                            component "App1"
                        } successor {
                            component "App3"
                        }
                    }

                }

                // the script will map the runnables to OsTask and guarantees that
                     the runnables of App1 are mapped before the runnables of App2
                     and App3

                scriptLogger.info("Created '{0}' task mappings.", taskMappings.
                    size())

                for (final SITaskMapping taskMapping : taskMappings) {
                    scriptLogger.info("Mapped runnable {0} to position {1}.",
                    taskMapping.getExecutableEntity().getName(),
                    taskMapping.getPositionInTask())
                }
            }
        }
    }
}
```

Listing 6.302: Order groups by using successor calls

```
// in this example we have a task on which periodically 50ms trigger are mapped
    after periodically 10ms trigger
// additionally we want the new task mappings will be insert always at the bottom
    of each group
// so new 50ms triggered task mappings shall be inserted below the other 50ms
    triggered task mappings
// and the new 10ms below the other 10ms triggered task mappings

scriptTask("DefineSuccessorsAndInsertBelowAlreadyMapped", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {

        def result = selectEvents {
           or {
               timing(0.01)
               timing(0.05)
           }
        }.mapToTask {
           selectTask {
               name("OsTask")
           }
           mapAllEventsOfRunnableEntity(false, true)

           // example continues on next page
```

Listing 6.303: Insert new task mappings always below existing - Part 1

```
defineSuccessors {

    // you can define it in one forTaskMapping sequence if you want to
    // we use in this example multiple sequences to demonstrate that it is
        possible to split one complex rule into a few simple rules
    // Note: we use task("OsTask") and the negation instead of mapped() / unmapped
        (), to eventually correct 10ms and 50ms trigger mapped to other tasks by
        accident

    // at first define that 10ms trigger shall be mapped ahead of 50ms trigger
    forTaskMapping {
        timing(0.01)
    }.successor {
        timing(0.05)
    }

    // already to 'OsTask' mapped 10ms trigger shall be mapped ahead of the other
        10ms trigger
    forTaskMapping {
        timing(0.01)
        task("OsTask")
    }.successor {
        timing(0.01)
        not {
           task("OsTask")
        }
    }

    // and already to 'OsTask' mapped 50ms trigger ahead of the other 50ms trigger
    forTaskMapping {
        timing(0.05)
        task("OsTask")
    }.successor {
        timing(0.05)
        not {
           task("OsTask")
        }
    }
}
                    }

                 scriptLogger.info("Created {0} task mappings.", result.size())
               }
             }
           }
         }
```

Listing 6.304: Insert new task mappings always below existing - Part 2

Additionally Sort Successors When you automate the task mapping using successor definition, there might be use cases where you still need the mappings in a 'well human readable' form. For example if you look them up in the task mapping editor from time to time. In that case you can sort the task mappings using the methods below so you can find them faster in the GUI. The sorting algorithm will respect the logical structure that is defined by the successor calls, so that these constraints are still guaranteed.

Quick example, if you define that all periodical 50ms trigger shall be mapped after the periodical 10ms trigger, all 50ms trigger will be sorted among the other 50ms trigger and all 10ms among the other 10ms trigger, without messing up the successor constraint.

`sortSuccessorsInternally(Comparator)` allows to define an additional comparator to increase the overview. Therefore the defined successors will be analyzed and divided in logical groups. The given comparator is applied on each logical group separately, so that the defined successor relationships will not be violated.
Hint: You can use only one comparator at the same time. So defining a custom comparator and using a default comparator at the same time will cause an exception.

The following default comparators can be used instead:
`sortSuccessorsInternallyByExecutableEntity()` - sorts the task mappings alphabetically by their executable entity names.

`sortSuccessorsInternallyByExecutableEntity()` sorts the task mappings of the defined successors alphabetically by their executable entity names. See `sortSuccessorsInternally(Comparator)` for more details.

To use a custom comparator use `sortSuccessorsInternally(Comparator)` instead.

Example

```
// in this example we have a task were all mode exit events shall be mapped ahead
    of  all mode entry events
// but additionally we want to sort them alphabetically without breaking this
    constraint

// the result will be a task on which all mode exit events are mapped first,
    sorted alphabetically
// followed by as well alphabetically sorted mode entry events

scriptTask("DefineSuccessorsAndSort", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {

        def result = selectEvents {
           or {
               modeExit()
               modeEntry()
           }
        }.mapToTask {
           selectTask {
               name("OsTask")
           }
           mapAllEventsOfRunnableEntity(false, true)

           defineSuccessors {
               forTaskMapping {
                   modeExit()
               }.successor {
                   modeEntry()
               }

               // you can use an own comparator calling sortSuccessorsInternally(
                   Comparator<ITaskMapping>)
               // in our example we sort alphabetically by executable entity names
                    with a default comparator
               sortSuccessorsInternallyByExecutableEntity()
           }
       }

       scriptLogger.info("Created {0} task mappings.", result.size())
      }
    }
  }
}
```

Listing 6.305: Define successors and sort elements for better overview

Specify an Order The order of the task mappings can be specified also with the help of an internal structural element, the so called position in task entry.

An `SIPositionInTaskEntry` represents a position in task for the task mapping. The entry is able to combine several events that are mapped to one position (e.g. needed when mapping a main function of a service component and its corresponding schedulable entity).

`order(Consumer)` allows to evaluate and change the order of the task mappings. The received `SIPositionInTaskEntrys` are already sorted respecting the defined successors by `defineSuccessors(Action)` code. If used, the `queue()` option may override the order defined by this `order(Consumer)` call.

It provides a possibility to order the already existing task mappings of the selected task and the new task mappings that should be created.

Example

```
import com.vector.cfg.sysdesc.model.taskmapping.SIPositionInTaskEntry
import com.vector.cfg.sysdesc.model.taskmapping.SITaskMapping

scriptTask ("orderTaskMapppingsSimple", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                     def taskMappings = selectEvents {
                    component("App1")
                } mapToTask {
                    selectTask {
                        name("OsTask")
                    }
                    order {
                        List<SIPositionInTaskEntry> entries ->

                        for (SIPositionInTaskEntry entry : entries) {
                            // identify by executable entity name and set the
                               position
                            if (entry.getTriggeredExecutableEntity().equals("
                               DataSendComp")) {
                                // Runnable 'DataSendComp' will be mapped
                                // to position 0 on task 'OsTask'
                                entry.setPosition(0)
                                continue
                            } else if (entry.getTriggeredExecutableEntity().equals
                               ("Runnable1")) {
                                entry.setPosition(1)
                                continue
                            } else if (entry.getTriggeredExecutableEntity().equals
                               ("Runnable2")) {
                                entry.setPosition(2)
                                continue
                            }
                        }
                    }
                }

                // print info to the console which runnable was mapped to which
                   position
                for (final SITaskMapping taskMapping : taskMappings) {
                    scriptLogger.info("Mapped runnable {0} to position {1}.",
                    taskMapping.getExecutableEntity().getName(),
                    taskMapping.getPositionInTask())
                }
            }
        }
    }
}
```

Listing 6.306: Simple example of ordering the task mappings

```
import com.vector.cfg.sysdesc.model.taskmapping.SIPositionInTaskEntry

scriptTask ("orderTaskMapppings", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                     def taskMappings = selectEvents {
                    component("App1")
                } mapToTask {
                    selectTask {
                        name("OtherName")
                    }
                    order {
                        List<SIPositionInTaskEntry> entries ->

                        int mappedIndex = 0
                        int index = 10

                        for (SIPositionInTaskEntry entry : entries) {
                            // identify by executable entity name
                            if (entry.getTriggeredExecutableEntity().equals("
                                DataSendComp")) {
                                entry.setPosition(9)
                                continue
                            }

                            // already mapped on task
                            def alreadyMapped = entry.getAssociatedTaskMappings().
                                find {
                                taskMapping -> taskMapping.getMappedTask() != null
                            }
                            if (alreadyMapped != null) {
                                entry.setPosition(mappedIndex)
                                mappedIndex++
                                continue
                            }

                            // newly mapped
                            entry.setPosition(index)
                            index++
                        }
                    }
                }

                scriptLogger.info("Created '{0}' task mappings.", taskMappings.
                    size())
            }
        }
    }
}
```

Listing 6.307: Manually order the task mappings

```
import com.vector.cfg.sysdesc.model.taskmapping.SIPositionInTaskEntry

scriptTask ("orderTaskMapppingsOfOsTask", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                     def taskMappings = selectEvents {
                    task("OsTask")
                } mapToTask {
                    filterTaskMappings {
                        task("OsTask")
                    }
                    selectTask {
                        name("OsTask")
                    }
                    order {
                        List<SIPositionInTaskEntry> entries ->

                        // in this example runnables of App1, App2 and App3 (with
                            only 1 task mapping) are mapped on OsTask
                        // sort the runnables by owner
                        int runnablesOfApp1 = 0
                        int runnablesOfApp2 = 0
                        for (SIPositionInTaskEntry entry : entries) {
                            if (entry.getOwner().equals("Component App1")) {
                                runnablesOfApp1++
                            }
                            if (entry.getOwner().equals("Component App2")) {
                                runnablesOfApp2++
                            }
                        }

                        // we sort in this example first runnables of 'App1'
                        // followed by the runnabels of 'App2'
                        // and last but not least the runnable of 'App3'
                        int maxIndex = entries.size() - 1
                        int indexForApp1 = 0
                        int indexForApp2 = runnablesOfApp1

                        for (SIPositionInTaskEntry entry : entries) {
                            // the runnable of App3 should be mapped to the last
                                position on OsTask
                            if (entry.getOwner().equals("Component App3")) {
                                entry.setPosition(maxIndex)
                            }
                            if (entry.getOwner().equals("Component App1")) {
                                entry.setPosition(indexForApp1)
                                indexForApp1++
                            }
                            if (entry.getOwner().equals("Component App2")) {
                                entry.setPosition(indexForApp2)
                                indexForApp2++
                            }
                        }
                    }
                }
                scriptLogger.info("Created '{0}' task mappings.", taskMappings.
                    size())
            }
        }
    }
}
```

Listing 6.308: Order task mappings on OsTask

Filter Task Mappings There is a way to narrow down the selected task mappings after selecting events or executable entities. This might be helpful especially in case you use multi-instantiation of software components. Since the selection of task mappings is only available for the task mappings use case, there is no own chapter for it.

`filterTaskMappings(Action)` allows to filter the task mappings that should be created. This might be especially helpful to narrow down the task mappings after selecting events or executable entities when using multi instantiation (e.g. to filter the task mappings for only one instance of a multi instantiated component prototype).

Per default the predicates are combined via logical AND. To realize other combinations, use the 'or','not' and 'and' predicates.

Task Mapping Predicates

- `component(String)` matches task mappings whose event is part of the internal behavior of a component with the given component name.
- `component(Pattern)` matches task mappings whose event is part of the internal behavior of a component with the given component name pattern.
- `moduleConfiguration(String)` matches task mappings whose event is part of the internal behavior of a module configuration with the given module configuration name.
- `moduleConfiguration(Pattern)` matches task mappings whose event is part of the internal behavior of a module configuration with the given module configuration name pattern.
- `moduleConfigurationAsrPath(String)` matches task mappings whose event is part of the internal behavior of a module configuration with the given module configuration autosar path.
- `moduleConfigurationAsrPath(Pattern)` matches task mappings whose event is part of the internal behavior of a module configuration with the given module configuration autosar path pattern.
- `unmapped()` matches task mappings which are not mapped to a task.
- `mapped()` matches task mappings which are mapped to a task.
- `task(String)` matches task mappings which are mapped to a task with the given task name.
- `task(Pattern)` matches task mappings which are mapped to a task whose name matches the given task name pattern.
- `componentType(String)` matches task mappings which belongs to component types with the given component type name.
- `componentType(Pattern)` matches task mappings which belongs to component types matching the given component type name pattern.
- `componentTypeAsrPath(String)` matches task mappings which belongs to component types with the given component type autosar path.
- `componentTypeAsrPath(Pattern)` matches task mappings which belongs to component types matching the given component type autosar path pattern.
- `event(String)` matches task mappings for events with the given event name.

- `event(Pattern)` matches task mappings for events matching the given event name pattern.
- `eventAsrPath(String)` matches task mappings for events with the given event autosar path.
- `eventAsrPath(Pattern)` matches task mappings for events matching the given event autosar path pattern.
- `bswEvent()` matches task mappings for bsw events.
- `rteEvent()` matches task mappings for rte events.
- `timing()` matches task mappings for timing events.
- `timing(Double)` matches task mappings for timing events with the given period (seconds).
- `init()` matches task mappings for init events.
- `dataReceived()` matches task mappings for data received events.
- `dataReceiveError()` matches task mappings for data receive error events.
- `dataSendCompleted()` matches task mappings for data send completed events.
- `dataWriteCompleted()` matches task mappings for data write completed events.
- `operationInvoked()` matches task mappings for operation invoked events.
- `operationInvoked(String)` matches task mappings for operation invoked events which are invoked by an operation with the given operationName.
- `serverCallReturns()` matches task mappings for events which are asynchronous server call returns events.
- `modeSwitch()` matches task mappings for mode switch events.
- `modeEntry()` matches task mappings for mode switch events with activation kind ON-ENTRY.
- `modeExit()` matches task mappings for mode switch events with activation kind ON-EXIT.
- `modeTransition()` matches task mappings for mode switch events with activation kind ON-TRANSITION.
- `modeSwitchedAck()` matches task mappings for mode switched acknowledgement events.
- `externalTrigger()` matches task mappings for external trigger occurred events.
- `internalTrigger()` matches task mappings for internal trigger occurred events.
- `background()` matches task mappings for background events.
- `transformerHardError()` matches task mappings for transformer hard error events.
- `mandatory()` matches task mappings for events which must be mapped. (The mapping of operation invoked events and bsw events whose schedulable entity has no via symbol matching runnable is optional.)
- `symbol(String)` matches task mappings for runnable entities with the given symbol and bsw schedulable entities whose corresponding bsw module entry short name matches the given symbol.
- `symbol(Pattern)` matches task mappings runnable entities whose symbol matches the given symbol pattern and bsw schedulable entities whose corresponding bsw module entry short name matches the given symbol pattern.

- `executableEntity(String)` matches task mappings for executable entities (functions) with the given name.
- `executableEntity(Pattern)` matches task mappings for executable entities (functions) with the given name pattern.
- `executableEntityAsrPath(String)` matches task mappings for executable entities (functions) with the given autosar path.
- `executableEntityAsrPath(Pattern)` matches task mappings for executable entities (functions) with the given autosar path pattern.
- `filterAdvanced(Predicate)` matches task mappings for which the given lambda results to true.

Example

```
scriptTask ("firstTaskMappings", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def taskMappings = selectExecutableEntities {
                    componentType("App1")
                } mapToTask {
                    selectTask {
                        name("OsTask")
                    }

                    // in this example two components ('App1' and 'App1_1') are of
                        component type 'App1'
                    // do the task mapping only for 'App1_1'
                    filterTaskMappings {
                        component("App1_1")
                    }
                }

                scriptLogger.info("Created '{0}' task mappings.", taskMappings.
                    size())
            }
        }
    }
}
```

Listing 6.309: Filter task mappings

Apply Execution Order Constraints An ExecutionOrderConstraint restricts the execution order of a set of ExecutableEntities. Therefore successor and direct successor relationships can be defined for executable entities (functions), but also for events (triggers).

Since the selection of execution order constraints is available only for the task mapping use case, there is no own chapter for it.

`selectExecutionOrderConstraints(Action)` allows to define predicates to select execution order constraints that should be applied.

Per default the predicates are combined via logical AND. To realize other combinations, use the 'or','not' and 'and' predicates.

Execution Order Constraint Predicates

- `name(String)` matches execution order constraints with the given execution order constraint name.
- `name(Pattern)` matches execution order constraints with the given execution order constraint name pattern.
- `filterAdvanced(Predicate)` matches execution order constraints for which the given lambda results to true.
- `and(Runnable)` combines the predicates inside the lambda with a logical AND.
- `or(Runnable)` combines the predicates inside the lambda with a logical OR.
- `not(Runnable)` negates the combination of predicates inside the lambda.

Example

```
import com.vector.cfg.sysdesc.model.eoc.SIExecutionOrderConstraint

scriptTask ("applyEOC", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def selectedConstraints
                def taskMappings = selectExecutableEntities {
                    component("App1_1")
                } mapToTask {
                    selectTask {
                        name("OsTask")
                    }

                    // select execution order constraints that should be applied
                    selectExecutionOrderConstraints {
                        name("App1ExecutionOrderConstraint")
                        selectedConstraints = getSelectedExecutionOrderConstraints
                            ()
                    }
                }

                scriptLogger.info("Created '{0}' task mappings.", taskMappings.
                    size())

                for (SIExecutionOrderConstraint eoc : selectedConstraints) {
                    scriptLogger.info("Applied execution order constraint '{0}'.",
                        eoc.getName())
                }
            }
        }
    }
}
```

Listing 6.310: Use execution order constraints for the task mapping

Check Current Task Mapping The event and the executable entity selections offer getters to retrieve task mappings. Therefore first the given predicate is evaluated to identify which events are selected, then all task mappings which references the selected events are collected.

`getTaskMappings()` retrieves all `SITaskMappings` for the selected events (see `getEvents()`).

Note:
1. In case of multi instantiation of component prototypes, the different instances share the same events, since the event is part of the internal behavior of the component type. Therefore if the event is selected, `getTaskMappings()` will always return the task mappings for all component prototypes.
2. Since this method can be run outside of a transaction, there might be selected events for which no task mapping container does exist yet. The container cannot be created by calling `getTaskMappings()`, so no task mapping can be returned. This happens if the system description is not synchronized, after changes in the structured extract were done (see Automation Interface Documentation, chapter about Model Synchronization for examples how to synchronize).

`getTaskMappings()` retrieves all `SITaskMappings` for the selected executable entities (see `getExecutableEntities()`).

Example

```
import com.vector.cfg.sysdesc.model.taskmapping.SITaskMapping

scriptTask("checkTaskMapping", DV_PROJECT){
  code {

      // because we use only a getter method and no transaction below,
      // make sure that your system description is synchronized,
      // otherwise task mapping container may be missing or obsolete.
      // since Cfg 5.18 this call is enough to make sure your task mapping
          containers are up to date
      modelSynchronization.synchronize()

      domain.runtimeSystem {
          def taskName = "OsTask"

          def taskMappings = selectEvents {
              task(taskName)
          // getTaskMappings will apply the predicate and return all task mappings
               for the selected events
          } getTaskMappings()

          // be careful in case of multi-instantiated component prototypes,
          // since events and executable entities are part of the internal
              behavior of the component type,
          // you will receive always the task mappings for all instances here
          // (even if the task mappings of the other component prototype instances
               are not mapped to "OsTask")

          // print info to the console with owner of the task mapping and the
              mapped task
          for (final SITaskMapping taskMapping : taskMappings) {
              scriptLogger.info("TaskMapping of '{1}' for event '{0}' is mapped to
                   '{2}'.",

              taskMapping.getEvent().getName(),
              taskMapping.getOwnerDescription(),
              taskMapping.getMappedTask() == null ? "no task" : taskMapping.
                  getMappedTask().getName())
          }
      }
  }
}
```

Listing 6.311: Check which events are currently mapped to OsTask

Keep Existing Task Mappings on Current Position `queue()` is an option that allows to keep the task mappings which are already mapped to the selected task on the current position. The new task mappings will be placed in the defined order into existing gaps. That means they are mapped to the lowest free position. This method may override the order defined in `defineSuccessors(Action)` and `order(Consumer)`.

Example: The selected task 'Task1' has already a TaskMappingA at position 1 and a TaskMappingB at position 3. The task mappings TaskMappingC, TaskMappingD and TaskMappingE (in that order) should be mapped to the same task using the `queue()` option. The new task mappings will be assigned in the defined order to the next free position on 'Task1'.
So the result will be:
0 -> TaskMappingC (new1)
1 -> TaskMappingA (old)
2 -> TaskMappingD (new2)
3 -> TaskMappingB (old)
4 -> TaskMappingE (new3)

If the options `mapAllEventsOfRunnableEntity(boolean, boolean)` or `combineViaSymbol(boolean)` needs to combine an existing mapping with other mappings, the combined task mapping is seen as a new mapping. Its position will be assigned according to the rules of a new mapping explained above. The old mapping which was not combined will be removed. In other words combined task mappings are preferred over single task mappings.

Example

```
scriptTask ("queueExample", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def taskMappings = selectExecutableEntities {
                    componentType("App1")
                } mapToTask {
                    selectTask {
                        name("OsTask")
                    }
                    // this option will consider the existing mappings on the task
                    // if an existing task mapping is not combined for another new
                         incoming mapping, it will remain on its current position
                    queue()
                }

                // if the existing task mapping was combined for another new
                    incoming mapping, it is considered as a new mapping and will
                    appear in the taskMappings below
                scriptLogger.info("Created '{0}' task mappings.", taskMappings.
                    size())
            }
        }
    }
}
```

Listing 6.312: Use the queue option example

Set Activation Offset You can set the value of the activation offset for the events/executable entities which will be newly mapped.

`setActivationOffset(Double)` allows to set the activation offset at the created task mappings. The offset will be set for all created task mappings to the given offsetInSeconds value.

It is also possible to set the activation offset parameter value of a task mapping using the event or the executable entity selection without mapping the events/executable entities to a task. You can use this option for example if you just want to set the activation offset and do not care whether the event/executable is already mapped to a task or not.

`setActivationOffset(double)` sets the activation offset at the task mapping containers for the selected events. If the symbol of a schedulable entity matches a runnable name, the activation offset will be set for both, even if only one of them is selected. (Matching works as for `ITaskMapper.combineViaSymbol(boolean)`.)

`setActivationOffset(double)}` sets the activation offset at the task mapping containers for the selected executable entities. If the symbol of a schedulable entity matches a runnable name, the activation offset will be set for both, even if only one of them is selected. (Matching works as for `ITaskMapper.combineViaSymbol(boolean)`.)

Examples

```
import com.vector.cfg.sysdesc.model.taskmapping.SITaskMapping

scriptTask("setActivationOffset", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
          def taskMappings = selectEvents {
              // define predicates to select your event
              component("App1")
              timing()

          // set the activation offset parameter value
          // for the task mapping to 10 ms
          }.setActivationOffset(0.01)

            // print info to the console with the runnable name which is triggered
                by the event
            // and the activation offset of the task mapping
            for (final SITaskMapping taskMapping : taskMappings) {
                scriptLogger.info("TaskMapping of runnable {0} has the activation
                    offset {1}.",
                taskMapping.getExecutableEntity().getName(),
                taskMapping.getActivationOffset())
            }
      }
    }
  }
}
```

Listing 6.313: Set the activation offset using the event selection

```
import com.vector.cfg.sysdesc.model.taskmapping.SITaskMapping

scriptTask("setActivationOffset", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
          def taskMappings = selectExecutableEntities {
              // define predicates to select your runnable
              component("App1")
              name("Runnable1")

          // set the activation offset parameter value
          // for the task mapping to 100 ms
          }.setActivationOffset(0.1)

            // print info to the console with runnable name and the activation
                offset
            for (final SITaskMapping taskMapping : taskMappings) {
                scriptLogger.info("TaskMapping of runnable {0} has the activation
                    offset {1}.",
                taskMapping.getExecutableEntity().getName(),
                taskMapping.getActivationOffset())
            }
      }
    }
  }
}
```

Listing 6.314: Set the activation offset using the executable entity selection

```
import com.vector.cfg.sysdesc.model.taskmapping.SITaskMapping

scriptTask("mapAndSetActivationOffset", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
          def taskMappings = selectEvents {
              // define predicates to select your event
              component("App1")
              init()
          } mapToTask {
                // map the event to task 'OsTask'
                selectTask {
                    name("OsTask")
                }
                // set the activation offset parameter value
                // for the task mapping to 10 ms
                setActivationOffset(0.01)
            }

            // print info to the console which runnable was mapped and the
                activation offset
            for (final SITaskMapping taskMapping : taskMappings) {
                scriptLogger.info("Mapped runnable {0} to position {1} with
                    activation offset {2}.",
                taskMapping.getExecutableEntity().getName(),
                taskMapping.getPositionInTask(),
                taskMapping.getActivationOffset())
            }
      }
    }
  }
}
```

Listing 6.315: Set the activation offset while mapping to a task

Set Os Schedule Point You can also set the value of the OsSchedulePoint parameter for the events/executable entities which will be newly mapped.

`setOsSchedulePoint(String)` allows to set the OsSchedulePoint at the created task mappings. The schedule point will be set for all created task mappings to the given osSchedulePoint value.

Also, it is possible to set the OsSchedulePoint parameter value of a task mapping using the event or the executable entity selection without mapping the events/executable entities to a task. You can use this option for example if you just want to set the OsSchedulePoint and do not care whether the event/executable is already mapped to a task or not. Typical values are "CONDITIONAL" and "UNCONDITIONAL".

`setOsSchedulePoint(String)` sets the OsSchedulePoint at the task mapping containers for the selected events. If the symbol of a schedulable entity matches a runnable name, the schedule point value will be set for both, even if only one of them is selected. (Matching works as for `ITaskMapper.combineViaSymbol(boolean)`.)

`setOsSchedulePoint(String)` sets the OsSchedulePoint at the task mapping containers for the selected executable entities. If the symbol of a schedulable entity matches a runnable name, the OsSchedulePoint will be set for both, even if only one of them is selected. (Matching works as for `ITaskMapper.combineViaSymbol(boolean)`.)

Examples

```
import com.vector.cfg.sysdesc.model.taskmapping.SITaskMapping

scriptTask("setOsSchedulePoint", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
          def taskMappings = selectEvents {
              // define predicates to select your event
              component("App1")
              timing()

          // set the OsSchedulePoint value
          }.setOsSchedulePoint("CONDITIONAL")

            // print info to the console with the runnable name which is triggered
                by the event
            // and the schedule point of the task mapping
            for (final SITaskMapping taskMapping : taskMappings) {
                scriptLogger.info("TaskMapping of runnable {0} has the
                    OsSchedulePoint value {1}.",
                taskMapping.getExecutableEntity().getName(),
                taskMapping.getOsSchedulePoint())
            }
      }
    }
  }
}
```

Listing 6.316: Set the OsSchedulePoint using the event selection

```
import com.vector.cfg.sysdesc.model.taskmapping.SITaskMapping

scriptTask("setOsSchedulePoint", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
          def taskMappings = selectExecutableEntities {
              // define predicates to select your runnable
              component("App1")
              name("Runnable1")

          // set the OsSchedulePoint parameter value
          }.setOsSchedulePoint("UNCONDITIONAL")

            // print info to the console with runnable name and the
                OsSchedulePoint
            for (final SITaskMapping taskMapping : taskMappings) {
                scriptLogger.info("TaskMapping of runnable {0} has the
                    OsSchedulePoint value {1}.",
                taskMapping.getExecutableEntity().getName(),
                taskMapping.getOsSchedulePoint())
            }
      }
    }
  }
}
```

Listing 6.317: Set the OsSchedulePoint using the executable entity selection

```
import com.vector.cfg.sysdesc.model.taskmapping.SITaskMapping

scriptTask("mapAndSetOsSchedulePoint", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
          def taskMappings = selectEvents {
              // define predicates to select your event
              component("App1")
              init()
          } mapToTask {
                // map the event to task 'OsTask'
                selectTask {
                    name("OsTask")
                }
                // set the OsSchedulePoint parameter value
                setOsSchedulePoint("CONDITIONAL")
            }

            // print info to the console which runnable was mapped and the
                OsSchedulePoint
            for (final SITaskMapping taskMapping : taskMappings) {
                scriptLogger.info("Mapped runnable {0} to position {1} with
                    OsSchedulePoint value {2}.",
                taskMapping.getExecutableEntity().getName(),
                taskMapping.getPositionInTask(),
                taskMapping.getOsSchedulePoint())
            }
      }
    }
  }
}
```

Listing 6.318: Set the OsSchedulePoint value while mapping to a task

Set Cyclic Trigger Implementation You can set the value of the cyclic trigger implementation for the events/executable entities which will be newly mapped.

`setCyclicTriggerImplementation(String)` allows to set the cyclic trigger implementation at the created task mappings. The cyclic trigger implementation will be set for all created task mappings to the given cyclicTriggerImplementation value.

It is also possible to set the cyclic trigger implementation parameter value of a task mapping using the event or the executable entity selection without mapping the events/executable entities to a task. You can use this option for example if you just want to set the cyclic trigger implementation and do not care whether the event/executable is already mapped to a task or not.

`setCyclicTriggerImplementation(String)` sets the CyclicTriggerImplementation at the task mapping containers for the selected events. If the symbol of a schedulable entity matches a runnable name, the CyclicTriggerImplementation value will be set for both, even if only one of them is selected. (Matching works as for `ITaskMapper.combineViaSymbol(boolean)`.)

`setCyclicTriggerImplementation(String)` sets the CyclicTriggerImplementation at the task mapping containers for the selected executable entities. If the symbol of a schedulable entity matches a runnable name, the CyclicTriggerImplementation will be set for both, even if only one of them is selected. (Matching works as for `ITaskMapper.combineViaSymbol(boolean)`.)

Examples

```
import com.vector.cfg.sysdesc.model.taskmapping.SITaskMapping

scriptTask("setCyclicTriggerImplementation", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
          def taskMappings = selectEvents {
              // define predicates to select your event
              component("App1")
              timing()

          // set the CyclicTriggerImplementation value
          }.setCyclicTriggerImplementation("Auto")

            // print info to the console with the runnable name which is triggered
                 by the event and the cyclic trigger implementation of the task
                mapping
            for (final SITaskMapping taskMapping : taskMappings) {
                scriptLogger.info("TaskMapping of runnable {0} has the
                    CyclicTriggerImplementation value {1}.",
                taskMapping.getExecutableEntity().getName(),
                taskMapping.getCyclicTriggerImplementation())
            }
      }
    }
  }
}
```

Listing 6.319: Set the CyclicTriggerImplementation using the event selection

```
import com.vector.cfg.sysdesc.model.taskmapping.SITaskMapping

scriptTask("setCyclicTriggerImplementation", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
          def taskMappings = selectExecutableEntities {
              // define predicates to select your runnable
              component("App1")
              name("Runnable1")

          // set the CyclicTriggerImplementation parameter value
          }.setCyclicTriggerImplementation("Auto")

            // print info to the console with runnable name and the
                CyclicTriggerImplementation
            for (final SITaskMapping taskMapping : taskMappings) {
                scriptLogger.info("TaskMapping of runnable {0} has the
                    CyclicTriggerImplementation value {1}.",
                taskMapping.getExecutableEntity().getName(),
                taskMapping.getCyclicTriggerImplementation())
            }
      }
    }
  }
}
```

Listing 6.320: Set the CyclicTriggerImplementation using the executable entity selection

```
import com.vector.cfg.sysdesc.model.taskmapping.SITaskMapping

scriptTask("mapAndSetCyclicTriggerImplementation", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
          def taskMappings = selectEvents {
              // define predicates to select your event
              component("App1")
              init()
          } mapToTask {
                // map the event to task 'OsTask'
                selectTask {
                    name("OsTask")
                }
                // set the CyclicTriggerImplementation parameter value
                setCyclicTriggerImplementation("Auto")
            }

            // print info to the console which runnable was mapped and the
                CyclicTriggerImplementation
            for (final SITaskMapping taskMapping : taskMappings) {
                scriptLogger.info("Mapped runnable {0} to position {1} with
                    CyclicTriggerImplementation value {2}.",
                taskMapping.getExecutableEntity().getName(),
                taskMapping.getPositionInTask(),
                taskMapping.getCyclicTriggerImplementation())
            }
      }
    }
  }
}
```

Listing 6.321: Set the CyclicTriggerImplementation value while mapping to a task

6.10.4.18 Bridge Between MDF and SI Model elements

The Runtime System Domain uses SI Model elements as model abstractions to simplify the structure of the AUTOSAR model. All objects which you select using the selection APIs are SI model elements.

`SIModelObject` is the common super interface for all SI model elements (as e.g. `Object` for all java classes). It defines common functionality which all SI model elements provide for generic handling of model abstractions.

On MDF level the base interface for AUTOSAR model objects is the MIObject.

It is possible to switch between model abstractions and MDF objects. This might be helpful for advanced script tasks that extend the current scope of the model abstractions.

`getModelAbstractionsForMdfObjects(Collection)` is a method for an arbitrary access to all SI-model abstractions which correspond to the given collection of MDF objects.

`getMdfObject()` is a bridge from the `SIModelObject` to the underlying MDF object. For compound model abstractions, the main object will be returned, e.g. returns the port for a component port.

Example for navigating between MDF model and model abstractions

```
import com.vector.cfg.model.asr.access.IAsrReferrableAccess
import java.util.Collections
import com.vector.cfg.model.si.base.SIModelObject
import com.vector.cfg.sysdesc.model.communication.instance.
    SIAbstractSignalInstance

scriptTask ("switchBetweenMdfAndModelAbstraction", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {

                // ------------------------------------------------
                // get a model abstraction object for your MDF object
                // ------------------------------------------------
                def referrableAccess = ScriptApi.activeProject.getInstance(
                    IAsrReferrableAccess)

                // get some MDF objects by e.g. using the referrable access
                def mdfSystemSignal = referrableAccess.getReferrableByPath("/
                    VectorAutosarExplorerGeneratedObjects/SYSTEM_SIGNALS/
                    Element_1_b16df82332bcf915")

                def mdfObjects = Collections.singletonList(mdfSystemSignal)

                // get the model abstractions for the MDF objects
                def modelAbstractions = getModelAbstractionsForMdfObjects(
                    mdfObjects)

                // for the system signal an IAbstractSignalInstance is returned,
                    if it is referenced by at least one ISignal
                // so there will be exactly one model abstraction in the
                    collection in this example
                def signalInstanceModelAbstraction
                for (SIModelObject modelAbstraction : modelAbstractions) {
                    if (modelAbstraction instanceof SIAbstractSignalInstance) {
                        signalInstanceModelAbstraction = modelAbstraction
                    }
                }

                if (signalInstanceModelAbstraction == null) {
                    scriptLogger.info("System Signal '{0}' is not referenced by
                        any ISignals",
                    mdfSystemSignal.getName())
                }

                // ------------------------------------------------
                // get a MDF object for your model abstraction object
                // ------------------------------------------------
                def mdfObject = signalInstanceModelAbstraction.getMdfObject()
                // now the system signal can be used on MDF level
            }
        }
    }
}
```

Listing 6.322: Switch between MDF and model abstraction example

6.10.4.19 Deleting Elements

Removing elements is not covered by the runtime system API yet. So we have to use the MDF model for that use case for now (see chapter for MDF model). You can of course use the selection

APIs to find the correct elements first (e.g. for the data mappings by selecting the signals and call getDataMappings() method) and then get their MDF objects by calling getMdfObject() which is supported for all objects of the runtime system domain model.

You can find examples for some common use cases below.

Example

```
import com.vector.cfg.sysdesc.model.connector.SIConnector
import com.vector.cfg.sysdesc.model.component.SIComponentPort

scriptTask("diconnectDelegationPorts", DV_PROJECT) {
    code {
        transaction {
            domain.runtimeSystem {

                // in this example we want to delete the connectors to all
                    delegation ports

                // select the component ports which should be disconnected first
                List<SIComponentPort> connectedDelegationPorts =
                    selectComponentPorts {
                     connected()
                     delegation()
                }.getComponentPorts()

                // get the connectors to those component ports
                List<SIConnector> connectorsToDelegationPorts = []
                connectedDelegationPorts.each {
                    connectorsToDelegationPorts.addAll(it.getConnectedConnectors()
                        )
                }

                // we want to do a report for the disconnected ports
                // therefore we will remember them
                List<SIComponentPort> disconnectedPPorts = []
                List<SIComponentPort> disconnectedRPorts = []

                // now delete the connectors
                connectorsToDelegationPorts.each { SIConnector connector ->
                    // we need to check whether we did not already have deleted
                        the connector
                    // and we need to check if the connector is deletable
                    // connectors which were not created in CFG5 cannot be deleted
                    if (!connector.getMdfObject().isDeleted()
                            && connector.getMdfObject().getCeState().isDeletable()
                                ) {
                        // add the component ports for a final report
                        disconnectedPPorts.add(connector.getProviderPort())
                        disconnectedRPorts.add(connector.getRequesterPort())
                        // finally delete the connector
                        connector.getMdfObject().delete()
                    }
                }

                for (int i = 0; i < disconnectedPPorts.size(); i++) {
                    scriptLogger.info("Deleted connector between {0} and {1}.",
                            disconnectedPPorts.get(i).getName(),
                            disconnectedRPorts.get(i).getName())
                }
            }
        }
    }
}
```

Listing 6.323: Delete connectors to delegation ports

```
import com.vector.cfg.sysdesc.model.communication.SICommunicationElement
import com.vector.cfg.sysdesc.model.datamapping.SIDataMapping
import com.vector.cfg.sysdesc.model.datamapping.SISenderReceiverDataMapping
import com.vector.cfg.sysdesc.model.communication.SIAbstractSystemSignal

scriptTask("deleteDataMappingsForDelPorts", DV_PROJECT) {
    code {
        transaction {
            domain.runtimeSystem {

                // in this example we want to delete the data mappings of sender
                    receiver delegation ports

                // select the communication elements first
                List<SICommunicationElement> communicationElements =
                    selectCommunicationElements {
                     delegation()
                     senderReceiver()
                     mapped()
                }.getCommunicationElements()

                // get the data mappings of these communication elements
                List<SIDataMapping> dataMappingsToDelete = []
                communicationElements.each {
                    dataMappingsToDelete.addAll(it.getDataMappings())
                }
                // example continues on next page
```

Listing 6.324: Delete data mappings of sender receiver delegation ports - Part 1

```
                // we want to do a report for the removed data mappings
                // therefore we will remember communication element and signal
                List<SICommunicationElement> comElementsOfDeletedMapping = []
                List<SIAbstractSystemSignal> signalsOfDeletedMappings = []

                // now delete the data mappings
                dataMappingsToDelete.each { SIDataMapping dataMapping ->
                    // we need to check if the data mapping is deletable
                    // data mappings which were not created in CFG5 cannot be
                        deleted
                    if (!dataMapping.getMdfObject().isDeleted()
                            && dataMapping.getMdfObject().getCeState().isDeletable
                                ()) {
                        // add the component ports for a final report
                        comElementsOfDeletedMapping.add(dataMapping.
                            getCommunicationElement())
                        signalsOfDeletedMappings.add(dataMapping.getSystemSignal()
                            )

                        // we want to extend the reporting for sender receiver to
                            signal group mappings
                        // report not only the signal group but also the group
                            signal mappings
                        if (dataMapping instanceof SISenderReceiverDataMapping) {
                            ((SISenderReceiverDataMapping) dataMapping).
                                getLeafDataMappings().each {
                                SISenderReceiverDataMapping childMapping ->
                                    comElementsOfDeletedMapping.add(childMapping.
                                        getCommunicationElement())
                                    signalsOfDeletedMappings.add(childMapping.
                                        getSystemSignal())
                            }
                        }

                        // finally delete the data mappings
                        // for sender receiver to signal group mappings it is
                            enough to delete the root mapping
                        dataMapping.getMdfObject().delete()
                    }
                }

                for (int i = 0; i < comElementsOfDeletedMapping.size(); i++) {
                    scriptLogger.info("Deleted data mapping for {0} to signal {1}"
                        ,
                            comElementsOfDeletedMapping.get(i).
                                getFullyQualifiedName(),
                            signalsOfDeletedMappings.get(i).getName())
                }
            }
        }
    }
}
```

Listing 6.325: Delete data mappings of sender receiver delegation ports - Part 2

6.10.4.20 Variant Handling

The only use case that supports PostBuild selectable variance in the runtime system domain is the mapping between communication elements and signals (data mapping). If you have a variant project the data mapping can only be done in an active model view (see chapter about model views).

There is no edit variance function for the data mapping in the automation API yet. But if a signal is visible in exactly one variant, the created data mapping will automatically be created only for the variant in which the signal is visible. For invariant signals the created data mappings will also be invariant.

So a good approach for PostBuild variant configurations is to loop over the model views and run your script logic for each view.

```
import com.vector.cfg.model.asr.view.IModelViewExecutionContext

// this example runs also successfully for project with no PostBuild variance
// because there just be only one model view - the invariant model view

scriptTask("dataMapVariant", DV_PROJECT) {
    code {
        transaction {
            domain.runtimeSystem {

                for (def modelView in variance.allPostBuildVariantViews) {
                    final IModelViewExecutionContext context = modelView.
                        executeWithThisView()

                    // make sure to close the view when finish (even if exceptions
                         occur) to not run further actions still in this view by
                        accident
                    context.withCloseable {

                        // do the data mapping inside this closure, we will keep
                            the example simple here
                        // remember: IAbstractSignalInstaces may also be variant
                        // so they should be selected also with an active model
                            view
                        selectCommunicationElements {
                            // the auto mapper will not create mappings which are
                                real duplicates
                            // but it is better in terms of performance to filter
                                for unmapped here
                            unmapped()
                        }.autoMap()
                    }
                }
            }
        }
    }
}
```

Listing 6.326: Create variant data mappings

6.10.4.21 Retrieving Short Name Paths and Fully Qualified Names

The runtime system automation API requires/allows to use short name paths and fully qualified names to select elements. This chapter describes how these can be retrieved.
In general you can find a 'Copy' -> 'Copy Short Name Path' command in the context menu for most elements in the GUI.
For the automation interface you can use appropriate getters depending on your use case.

Path of Port Interface Mapping If you have already connected two ports and use a port interface mapping for the connection, you can search for your connector in the ECU Software Components

Editor. You can find it in the Application Ports grid or the Service Mappings grid under the ECU Composition node or under the Application or Service Ports node of your application or service component.

'Copy' -> 'Copy Short Name Path' in the context menu available in the Port Interface Mapping column for a given connection copies the short name path of the connection's port interface mapping to the clip board.

If you have no connection yet which uses the port interface mapping, search for the port interface mapping in another way. Expand the Port Interface Mapping Sets node of the ECU Software Components Editor and the node of the set which contains your mapping. Now you can do the 'Copy' -> 'Copy Short Name Path' command in the context menu of the tree node which belongs to the port interface mapping your were looking for.

Paths of System Signals and System Signal Groups GUI: If you have already used your signal / signal group for a data mapping, you can find the data mapping in the Data Mapping or the Application Ports grid under the ECU Composition node. Once you found your mapping you can retrieve the short name path of the signal / signal group via the 'Copy' -> 'Copy Short Name Path' context menu on the cell of the Signal column in the Data Mapping grid or the Mapped Signal column of the Application Ports grid.

AI: The following getter methods might be useful (see Javadoc of the method for more details):

- `SIAbstractSignalInstance.getAutosarPath()`
- `SIAbstractSignalInstance.getFullyQualifiedName()` - This getter might help you identifying group signals.

Fully Qualified Name of Communication Elements GUI: Communication elements are used for data mappings. Their fully qualified name is build out of three parts (in general, children of complex communication elements might have more). The name of the component, the name of the port and the name of the data element/operation/trigger itself. In case of a delegation port you can use 'ECU Composition' or 'COMPOSITIONTYPE' as component name.

Examples

Communication element of non-delegation port:
'ComponentName.PortName.DataElementName'

Communication element of delegation port:
'ECU Composition.DelegationPortName.DataElementName'

Complex communication element (record element of a record):
'ComponentName.PortName.RecordName.RecordElement1Name'

Complex communication element (array element at certain index):
'ComponentName.PortName.ArrayName[0]'

Open the data mapping assistant. You can find it in the navigation view under Runtime System -> Add Data Mapping or as hyperlink above grids which shows data mappings. Select the direction

'Find matching signals for the communication elements' on the first page and after that your communication elements on the second page. Now on the third page (Confirm page) you can open a context menu on the cell in the Communication Element column of your communication element and select copy fully qualified name. The name will be copied into the clip board. If you need the name of a child communication element which is not shown yet, you might have map the parent to a signal group first.

You can also just use the examples above and replace the names with the names of your component, port and communication element.

AI: The following getter methods might be useful (see Javadoc of the method for more details):

- `SICommunicationElement.getFullyQualifiedName()`

6.10.4.22 Best Practice And Further Examples

Create Selection Based on Existing Elements Most selection APIs offer a put method at the selector. This method allows us to put elements into a selection and continue working with elements we already have created or selected previously. The purpose of that is to optimize performance, avoid defining the same predicates over and over again and increase the level of control.

```
import com.vector.cfg.sysdesc.model.component.origin.SIOriginComponentPort
import com.vector.cfg.sysdesc.model.connector.SIConnector
import com.vector.cfg.sysdesc.model.component.SIComponentPort

scriptTask("createAndConnectDelegationPort", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {

        List<SIComponentPort> createdDelegationPorts = selectOriginComponentPorts
            {
            // select some origin context ports
            provided()
            innerTopLevelDelegation()
        }.createFlatExtractDelegationPorts {

            // create a delegation port for each selected origin port
            nameFromOriginPort {  SIOriginComponentPort originPort ->
                originPort.getPortName()
            }
            useDefaultDirection()
        }

        // now use the new ports to put them into a selection
        // we want to use the auto mapper (using the origin context names) to
            connect them to inner SWC ports
        List<SIConnector> createdConnections = selectComponentPorts {
            put(createdDelegationPorts)
            // we can still use predicates here, they would be applied only to our
                new ports which we put into the selection
            // for example we could connect only the provided new delegation ports
                here using additionally the predicate provided()
        } autoMapTo {
            useOriginContextForMatch()
        }

        // and finally do some reporting
        for (final SIConnector connector : createdConnections) {
            final SIComponentPort pPort = connector.getProviderPort()
            final SIComponentPort rPort = connector.getRequesterPort()

            scriptLogger.info("Created new delegation port {0} and connected it to
                {1}.",
                pPort.isDelegationPort() ? pPort.getName() : rPort.getName(), //
                    our new created delegation port
                pPort.isDelegationPort() ? rPort.getName() : pPort.getName()) //
                    the connected inner application port
        }

      }
    }
  }
}
```

Listing 6.327: Create delegation ports for selected origin ports and connect them

Optimizing Performance This chapter will give some advice on what may help if the script execution times get too high. In general we recommend to optimize the readability of your code in first place, but in some cases when processing a large number of objects restructure the code according to the points below may reduce the execution time of the script code.

Check complexity of the script code: Scripts may get slow if the code iterate over big collections of elements and while doing that performs performance critical operations or iterates over another big collection (nÂ² complexity). This may be also hidden somewhere between the lines. For example if the script needs to do many data mappings or connectors make sure you use the auto map call directly at the selection APIs and as few times as possible or use the simple API inside which you can put lists of elements at once instead of doing the calls for one single object after another.

In general the script performs better, when suitable data structures are prepared first so that the total amount of modification calls (e.g. autoMapTo(Closure) at the component port selection) at the selection APIs can be reduced.

Do not open unnecessary transactions and not over extend usage of transactions: Each transaction has an overhead, for example the validation results needs to be updated. So avoid opening multiple transactions if there is no benefit for you by doing that. A good example is if you want to create 500 data mappings. You can do every single mapping in an own transaction or all in one transaction. If you do not need the undo operation for every single mapping separately, you should always prefer to do all 500 mappings in one transaction.

Narrow down selections where possible without significant effort: For example when you connect ports and you have many already connected ports which you do not care about, use the unmapped predicate to not selecting them at all. The auto mapper will have to match less elements.

Prefer reusing elements over selecting elements: Let's assume you have created new delegation ports and want to connect them to other component ports. After creating the new ports use the returned list and put it into the component port selection to connect them. This will be more efficient then defining predicates that match to that newly created ports.

Use @CompileStatic for advanced filters: If your scripts use advanced filters whose closures need to be evaluated a large number of times, the code should be refactored so that the closure is built by an own method which uses @CompileStatic. That makes the advanced filter faster, since the closure needs to be compiled only one single time.

Avoid switching very often between selecting and creating elements We use internal caches for many attributes and dependencies between elements which needs to be invalidated every time when model changes are performed. So if your selection does not depend on the model changes which you want to do in next steps, prefer to select all elements you need first and then start modifying the model.

So for example instead of:

- select communication elements for sender receiver data mappings
- create sender receiver data mappings
- select operation communication elements for client server data mappings
- create client server data mappings

The following workflow will use the caches more efficient:

- select communication elements for sender receiver data mappings
- select operation communication elements for client server data mappings
- create sender receiver data mappings
- create client server data mappings

6.10.4.23 Access to CEState of SI Model elements

For the domain model objects of the runtime system domain, the CEState can be retrieved via the underlying MDF object. So, first call getMdfObject at the according model abstraction and then call the getCEState getter. The CEState helps to identify if the object can be changed or deleted.

For example when deleting data mappings, some of them might have been created in the DaVinci Developer workspace and therefore cannot be deleted in Cfg5.

```
import com.vector.cfg.sysdesc.model.communication.SICommunicationElement
import com.vector.cfg.sysdesc.model.datamapping.SIDataMapping
import com.vector.cfg.model.mdf.ar4x.systemtemplate.datamapping.MIDataMapping

scriptTask("DeleteDataMappingDependingOnCEState", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def selectedComElements = selectCommunicationElements {
                }.getCommunicationElements()

                for (SICommunicationElement comElement : selectedComElements) {
                    for (SIDataMapping dataMapping : comElement.getDataMapping())
                        {
                        // to get the CEState we need the underlying MDF object
                        MIDataMapping mdfDataMapping = (MIDataMapping) dataMapping
                        // now get CEState of MDF object
                        // if the Data Mapping is from DaVinci DEV workspace
                        // the CEState would return 'false' here
                        if (mdfDataMapping.getCeState().isDeletable()) {
                            mdfDataMapping.delete()
                        }
                    }
                }
            }
        }
    }
}
```

Listing 6.328: Evaluate CEState of Data Mapping

6.10.5 Crypto Domain

The crypto domain API is specifically designed to support crypto related use cases. It is available from the `com.vector.cfg.automation.scripting.base.IAutomationContext.IDomainApi` 6.10 on page 159 in the form of the `ICryptoApi` interface.

`getCrypto` allows accessing the `ICryptoApi` like a property.

`crypto(Transformer)` allows accessing the `ICryptoApi` in a scope-like way.

The `ICryptoApi` is an API for accessing crypto handling interfaces and setting up jobs and keys in an easier way.

Create Empty Job `createEmpyJob()` Creates an empty job in the crypto domain.

Create Empty Key `createEmptyKey()` Creates an empty key in the crypto domain.

Create Primitive `createPrimitive(String, String, String, String,String)` Creates a new primitive object with the given parameters.

Create Job `createJob(String, String, String, String)` Creates a new job object with the given parameters.

Set Job Priority `setJobPriority(String, int)` Sets the priority of the job object identified by its jobByName.

Set Use Port `setUsePort(String, boolean)`Sets the use port flag of the job object identified by its jobByName.

6.11 Unresolved Reference API

The Unresolved Reference APIs are specifically designed to manage unresolved references in specific scopes.

The Unresolved Reference API is the entry point for accessing the unresolved references. It is available on `IProject` instances.

The `IUnresolvedReferenceApi` provides the methods to access the different APIs which are designed to manage unresolved references of specific scopes. For an example see the `IEcucUnresolvedReferenceApi` 6.11.1.

`IProjectApi.getUnresolvedReferences()` allows accessing the `IUnresolvedReferenceApi` like a property.

```
scriptTask('AccessAsPropertyTask') {
  code {
    // IUnresolvedReferenceApi is available as unresolvedReferences property
    def unresolvedReferencesApi = unresolvedReferences
  }
}
```

Listing 6.329: Accessing IUnresolvedReferenceApi as a property

`IProjectApi.unresolvedReferences(Transformer)` allows accessing the `IUnresolvedReferenceApi` in a scope-like way.

```
scriptTask('AccessLikeScopeTask') {
  code {
    unresolvedReferences {
      // IUnresolvedReferenceApi is available inside this Closure
    }
  }
}
```

Listing 6.330: Accessing IUnresolvedReferenceApi in a scope-like way

6.11.1 Active ECUC Unresolved Reference API

The ECUC Unresolved Reference Api is specifically designed to read and edit unresolved references of the active ECUC.

`getActiveEcuc()` allows accessing the `IEcucUnresolvedReferenceApi` like a property.

```
scriptTask('AccessApiAsPropertyTask') {
    code {
        def ecucUnresolvedReferenceApi = unresolvedReferences.activeEcuc
    }
}
```

Listing 6.331: Accessing IEcucUnresolvedReferenceApi as a property.

`activeEcuc(Transformer)` allows accessing the `IEcucUnresolvedReferenceApi` in a scope-like way.

```
scriptTask('AccessApiLikeScopeTask') {
    code {
        unresolvedReferences.activeEcuc {
            // IEcucUnresolvedReferenceApi is available inside this Closure
        }
    }
}
```

Listing 6.332: Accessing IEcucUnresolvedReferenceApi in a scope-like way.

6.11.1.1 Selecting unresolved references

`select(Action)` allows the selection of all unresolved references of the active ECUC using predicates. The returned selection is read only. Use `selectChangeable(Action)` for changes. The selection is returned by an `IEcucUnresolvedReferenceSelection`.

`selectChangeable(Action)` allows the selection of the changeable unresolved references of the active ECUC using predicates. The selection is returned by an `IChangeableEcucUnresolvedReferenceSelection`.

Examples

```
scriptTask('SelectAllReferencesTask') {
    code {
        def selection = unresolvedReferences.activeEcuc.select {
            // filters can be set here
        }
    }
}
```

Listing 6.333: Get a filtered set of all unresolved references at the ECUC configuration.

```
scriptTask('SelectAllReferencesExampleTask') {
    code {
        def selection = unresolvedReferences.activeEcuc.select {
            owner(~ ".*/Rte/Com/.*")
        }
    }
}
```

Listing 6.334: Get a filtered set of all unresolved references at the ECUC configuration with a specific pattern at the owner path.

```
scriptTask('SelectChangeableTask') {
    code {
        def selection = unresolvedReferences.activeEcuc.selectChangeable {
            // filters can be set here
        }
    }
}
```

Listing 6.335: Get a filtered set of the changeable unresolved references at the ECUC configuration.

Predicates Predicates can be set by several methods. These can be used to filter the unresolved references.

- `reference(String)` filters the unresolved references by the reference path as String.
- `reference(Pattern)` filters the unresolved references by the reference path as a pattern.
- `owner(MIObject)` filters the unresolved references by the owner object.
- `owner(String)` filters the unresolved references by the exact owner path string.
- `owner(Pattern)` filters the unresolved references by the owner as a pattern.
- `container(MIContainer)` filters the unresolved references by a container. This container contains the referenced elements of the filtered references.

6.11.1.2 Set changeable unresolved references

`setCommonReferencePath(AsrPath)` sets a common reference for all filtered unresolved references.

`setReferencesByFunction(Function)` sets a common reference for all filtered unresolved references by using a String -> String function.

`replaceInReferences(String, String)` sets a new reference for all unresolved references of the selection by replacing an included string.

`replacePrefixInReferences(String, String)` sets a new reference for all unresolved references of the selection by replacing a prefix of the unresolved references.

Examples

```
scriptTask('SetChangeableTask'){
    code{
        def selection = unresolvedReferences.activeEcuc.selectChangeable {
            // filters can be set here
        }
        transaction{
            // this path is a created AsrPath as an example. Use a valid path
                instead.
            def path = AsrPath.create("/Changed/Reference")
            selection.setCommonReferencePath(path)
        }
    }
}
```

Listing 6.336: Set changeable unresolved references.

```
scriptTask('ReplaceInChangeableTask'){
    code{
        def selection = unresolvedReferences.activeEcuc.selectChangeable {
            // filters can be set here
        }
        transaction{
            selection.replaceInReferences("MICROSAR", "OTHER")
        }
    }
}
```

Listing 6.337: Replace 'MICROSAR' with 'OTHER' in all changeable unresolved references.

```
scriptTask('ReplacePrefixInChangeableTask'){
    code{
        def selection = unresolvedReferences.activeEcuc.selectChangeable {
            // filters can be set here
        }
        transaction{
            selection.replacePrefixInReferences("/MICROSAR", "/OTHER")
        }
    }
}
```

Listing 6.338: Replace the prefix '/MICROSAR' with '/OTHER' in all changeable unresolved references.

6.12 Persistency

The `persistency` API provides methods which allow to import and export model data from and to files. The files are normally in the AUTOSAR `.arxml` format.

6.12.1 Model Export

The `modelExport` allows to export MDF model data into `.arxml` files. To access the export functionality use one of the `getModelExport()` or `modelExport(Closure)` methods.

```
// You can access the API in every active project
def exportApi = persistency.modelExport

//Or you use a closure
persistency.modelExport {
}
```

Listing 6.339: Accessing the model export persistency API

6.12.1.1 Export ActiveEcuc

The method `exportActiveEcucToFile(Object)` exports the whole ActiveEcuC configuration into a single file of type `Path` specified by the user.

```
scriptTask('taskName') {
    code {
        def destinationFile // Define the file to export into...
        Path resultFile = persistency.modelExport.exportActiveEcucToFile(
            destinationFile)
    }
}
```

Listing 6.340: Export the ActiveEcuc to a file

The method `exportActiveEcuc(Object)` exports the whole ActiveEcuC configuration into a single file of type `Path` in the folder specified by the user.

```
scriptTask('taskName') {
    code {
        def tempExportFolder // Define the folder to export into...
        Path resultFile = persistency.modelExport.exportActiveEcuc(
            tempExportFolder)
    }
}
```

Listing 6.341: Export the ActiveEcuc into a folder

6.12.1.2 Export PostBuild Variants (Post-build selectable)

The method `exportPostBuildVariants(Object)` exports the PostBuild variants info into the given folder specified by the user. This will export the ActiveEcuc and miscellaneous data. The ActiveEcuC is exported into one file per variant (even for split projects), named as `<project-name>.<variant-name>.ecuc.arxml`. Miscellaneous data is exported into one file per variant, named as `<project-name>.<variant-name>.misc.arxml`.

The files contain all data of the project except:

- ModuleConfigurations, ModuleDefinitions
- BswImplementations, EcuConfigurations
- Variant information like EvaluatedVariantSet

The method returns a `List<Path>` of exported files.

```
scriptTask('taskName') {
    code {
        persistency.modelExport {
            def tempExportFolder = paths.resolveTempPath(".")
            List<Path> fileList = exportPostBuildVariants(tempExportFolder)
        }
    }
}
```

Listing 6.342: Export a PostBuild project into files per predefined variant

6.12.1.3 Export PreBuild Variants

The method `exportPreBuildVariants(Object)` exports the PreBuild variants info into the given folder specified by the user. This will export the ActiveEcuc and miscellaneous data. The ActiveEcuC is exported into one file per variant (even for split projects), named as `<project-name>.<variant-name>.ecuc.arxml`. Miscellaneous data is exported into one file per variant, named as `<project-name>.<variant-name>.misc.arxml`.

The files contain all data of the project except:

- ModuleConfigurations, ModuleDefinitions
- BswImplementations, EcuConfigurations

The method returns a `List<Path>` of exported files.

```
scriptTask('taskName') {
    code {
        persistency.modelExport {
            def tempExportFolder = paths.resolveTempPath(".")
            List<Path> fileList = exportPreBuildVariants(tempExportFolder)
        }
    }
}
```

Listing 6.343: Export a PreBuild project into files per predefined variant

6.12.1.4 Export Module Configuration

The method `exportModelTree(Object, MIObject, MIObject...)` exports the specified model objects and their subtrees into a single file of type `Path` in the folder specified by the user.

```
scriptTask('taskName') {
  code {

        Path location = paths.resolveScriptPath(".")
        def moduleList = mdfModel("EcuC")
        MIModuleConfiguration ecuC = moduleList.getFirst()
        Path resultFile = persistency.modelExport.exportModelTree(location, ecuC)
  }
}
```

Listing 6.344: Exports a module configuration

6.12.1.5 Advanced Exports

The advanced export use case provides access to multiple `IModelExporter` for special export use cases like exporting the system description for the RTE.

Normally you would retrieve an `IModelExporter` by its ID via `getExporter(String)`. Each exporter also provides multiple export methods, for example

- `IModelExporter.export(Object, Object...)` to export the model or
- `IModelExporter.exportAsPostBuildVariants(Object, Object...)` to export the model divided into files per PostBuild predefined variant.

You can retrieve a list of supported exporters from method `getAvailableExporter()`. The list can differ based on the loaded data in your project.

```
scriptTask('taskName') {
    code {
        def tempExportFolder = paths.resolveTempPath(".")

         // Export with an exporter in one line
        persistency.modelExport["activeEcuc"].export(tempExportFolder)
    }
}
```

Listing 6.345: Export the project with an exporter into a folder

```
scriptTask('taskName') {
    code {
        def tempExportFolder = paths.resolveTempPath(".")
        persistency.modelExport['everything'].exportAsPreBuildVariants(
            tempExportFolder)
    }
}
```

Listing 6.346: Export the prebuild variants with an exporter into a folder

```
scriptTask('taskName') {
    code {
        def tempExportFolder = paths.resolveTempPath(".")

        def fileList
        //Switch to the persistency export API
        persistency.modelExport{
            // The getAvailableExporter() returns all exporters in the system
            def exporterList = getAvailableExporter()

            // Select an exporter by its ID
            def exporterOpt = getExporter("activeEcuc")

            exporterOpt.ifPresent { exporter ->
                // Export into folder, if exporter exists
                fileList = exporter.export(tempExportFolder)
            }
        }
    }
}
```

Listing 6.347: Export the project with an exporter and checks

Export a Model Tree The method `exportModelTreeToFile(Object, MIObject, MIObject...)` exports the specified model objects and their subtrees into a single file of type `Path` specified by the user.

```
scriptTask('taskName') {
    code {
        def destinationFile // Define the file to export into...
        MIARPackage autosarPkg = mdfModel(AsrPath.create("/MICROSAR"))

        persistency.modelExport{
            def resultPath = exportModelTreeToFile(destinationFile, autosarPkg)
        }
    }
}
```

Listing 6.348: Export an AUTOSAR package into a file

The method `exportModelTree(Object, MIObject, MIObject...)` exports the specified model objects and their subtrees into a single file of type `Path` in the folder specified by the user.

```
scriptTask('taskName') {
    code {
        def exportFolder = paths.resolveTempPath(".")
        MIARPackage autosarPkg = mdfModel(AsrPath.create("/MICROSAR"))

        def resultFile = persistency.modelExport.exportModelTree(exportFolder,
            autosarPkg)
    }
}
```

Listing 6.349: Export an AUTOSAR package into a folder

Export a Model Tree including all referenced Elements You could also export model trees including all referenced elements with the exporter `modelTreeClosure`:

```
scriptTask('taskName') {
    code {
        def exportFolder = paths.resolveTempPath(".")
        MIARPackage microsarPkg = mdfModel(AsrPath.create("/MICROSAR"))
        MIARPackage autosarPkg = mdfModel(AsrPath.create("/AUTOSAR"))

        persistency.modelExport["modelTreeClosure"].export(exportFolder,
            autosarPkg, microsarPkg)
    }
}
```

Listing 6.350: Exports two elements and all referenced elements

Usage of Exporter Arguments You can use `withExporterArgs(Map, Transformer)` to specify exporter arguments like in the command line with `-exporterArgs` argument. The key is the exporter ID, the value are the arguments to the exporter. See command line help for details.

```
persistency.modelExport {
    // Specify the arguments with exporterId: "arguments"
    withExporterArgs(modelTree: "--element /MICROSAR") {
        // Call any export code with the active arguments.
        getExporter("modelTree").get().exportToFile(destinationFile)
    }
}
```

Listing 6.351: Use exporter arguments like in the commandline

6.12.2 Model Import

To access the import functionality use one of the `getModelImport()` or `modelImport(Transformer)` methods.

```
// You can access the API in every active project
def importApi = persistency.modelImport

//Or you use a closure
persistency.modelImport {

}
```

Listing 6.352: Accessing the model import persistency API

6.12.2.1 Module Configuration Import

To access the module import functionality use one of the `importModuleConfigurations` methods.

```
def importFile // Define input file ...
// You can access the API inside the closure
persistency.modelImport {
    importModuleConfigurations(importFile)
}
```

Listing 6.353: Accessing the import module configuration persistency API

The method `importModuleConfigurations(Path)` imports `MIModuleConfiguration` from the specified `.arxml` file into the current ActiveEcuC.

The method `importModuleConfigurations(Path, Action)` imports `MIModuleConfiguration` from the specified `.arxml` file into the current ActiveEcuC. The Closure can be used to specify the import mode and filter if necessary.

The method `importModuleConfigurations(List)` imports `MIModuleConfiguration` from the specified `.arxml` files into the current ActiveEcuC.

The method `importModuleConfigurations(List, Action)` imports `MIModuleConfiguration` from the specified `.arxml` files into the current ActiveEcuC. The Closure can be used to specify the import mode and filter if necessary.

6.12.2.2 Specify Import Mode and Module Filter

Use the methods `addToModel(Action)`, `replaceInModel(Action)` and `mergeIntoModel(Action)` to specify an import mode.

- The method `replaceInModel(Action)` *(this is the default mode)* replaces already existing module configurations with the imported one.
- The method `addToModel(Action)` adds new module configurations to the model. The selected module configurations must not yet exist.
- The method `mergeIntoModel(Action)` merges the selected modules into the model. If the selected module does not yet exist, it behaves the same as `addToModel(Action)`. Otherwise, the imported configuration will be merged into the existing module. On conflicts, the imported data will replace the existing data.

To specify a filter for the module configurations to import use one of the methods:

- `module(DefRef)` Use a single `DefRef`
- `module(List)` Use a list of `AsrPath`, `DefRef` or Definition Refs as `String`
- `module(AsrPath)` Use a single `AsrPath`
- `module(String)` Use a single Definition Ref as `String`

```
// You can access the API inside the closure
def importFile = paths.resolvePath("./ImportFile.arxml")
persistency.modelImport {
    importModuleConfigurations(importFile) {
        // add module configurations to the current ActiveEcuC
        addToModel() {
            module("/MICROSAR/LinIf")   // -> filter on DefRef as String
            def linNmAsrPath =  AsrPath.create("/ActiveEcuC/LinNm")
            module(linNmAsrPath) // -> filter on Autosar path as AsrPath instance
        }
        // replace already existing module configurations in the current
            ActiveEcuC
        replaceInModel() {
            List<String> modulesToImport = Arrays.asList("/MICROSAR/LinSM")
            module(modulesToImport) // filter on list of DefRef as String
        }
        mergeIntoModel() {
            // add modules to merge
        }
    }
}
```

Listing 6.354: Specify the module configuration import mode and filter

6.12.3 Check BSW Package Compatibility

To check if the BSW package and project are compatible, use the method `checkBswPkgCompatibility`.

The method `checkBswPkgCompatibility(Object)` verifies if a BSW migration is needed for the given project.

`isCompatible()` returns true if the BSW Package and project are compatible otherwise false.

`getMessage()` returns the message about the BSW Package compatibility state.

```
scriptTask('taskName', DV_APPLICATION) {
  code {

        def res
        res = bswCheck.checkBswPkgCompatibility("<PATH-TO-DVJSON-FILE>")
        boolean isCompatible=res.compatible
        String compatibilityStatus=res.message
  }
}
```

Listing 6.355: Check BSW Package Compatibility for project

6.13 Compare and Merge

The “Compare and Merge” feature is an essential tool for managing and integrating changes in projects. It allows developers to identify differences between various versions of files or projects and merge them efficiently. This feature plays a crucial role in collaborative development by highlighting conflicts and facilitating the merging of changes.

6.13.1 Read Only Project Comparison

The ’Read Only Project Comparison’ feature allows developers to compare different versions of files or projects without the ability to merge changes. This feature is particularly useful for reviewing changes, understanding differences, ensuring quality before any integration occurs and creating own reports.

6.13.1.1 Structure

The entry point for the read only project comparison is the project service **IProjectCompare**.

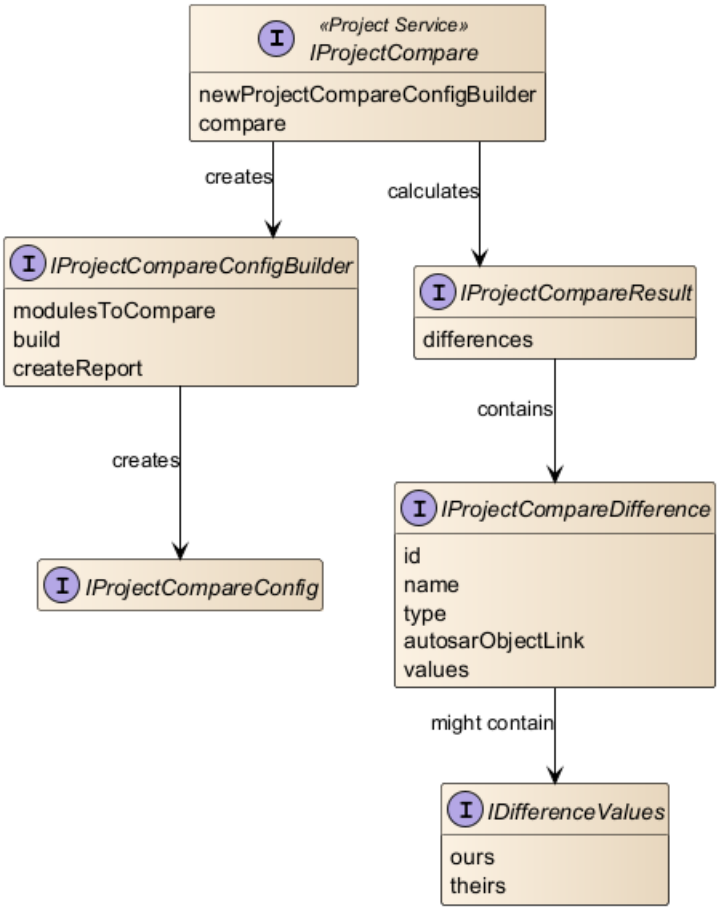


Figure 6.12: Structure of the read only project comparison interfaces

6.13.1.2 Accessing the API

In order to access the read only project compare API the project service **IProjectCompare** is used.

```
IProjectCompare projectCompare = projects.activeProject.projectContext[
    IProjectCompare]
IProjectCompareConfigBuilder configBuilder = projectCompare.
    newProjectCompareConfigBuilder(projectToCompareWith)

IProjectCompareResult result = projectCompare.compare(configBuilder.build())
def differences = result.getDifferences()

def autosarLinkAndTypeOfDifferences = differences.collect { difference ->
    [difference.autosarObjectLink, difference.type]
}
```

Listing 6.356: The general usage of the read only project comparison API

6.13.1.3 IProjectCompare

Represents the entry point for the read only project compare API. To configure the API use `IProjectCompareConfigBuilder` which can be created via

`IProjectCompare.newProjectCompareConfigBuilder(Object)`.

The compare operation can be executed by calling

`IProjectCompare.compare(IProjectCompareConfig)`

with the created `IProjectCompareConfig`.

Creating new comparison config builder `newProjectCompareConfigBuilder(Object)` creates a new builder instance for the `IProjectCompareConfig`. Supported are:

- Absolute paths
- Relative paths are resolved to the location of the script

Executing the comparison `compare(IProjectCompareConfig)` executes the compare operation with the given `IProjectCompareConfig`.

6.13.1.4 IProjectCompareConfigBuilder

Represents the configuration builder for the read only project compare operation. To create an instance use `IProjectCompare.newProjectCompareConfigBuilder(Object)`.

Compare only distinct module configurations `addModulesToCompare(List)` adds the short names of the module configurations to compare.

Build comparison config `build()` builds the `IProjectCompareConfig` containing all the settings made so far.

Create report `createReport()` sets a flag indicating that a report should be generated. The report will be located in the logging directory.

6.13.1.5 IProjectCompareResult

Represents the result of a read-only project comparison.

Retrieving the found differences `getDifferences()` gets the collection of found differences.

6.13.1.6 IProjectCompareDifference

Represents a read-only difference resulting from the comparison of projects.

The ID of the difference `getId()` gets the ID of the difference. This ID is based on the path of the associated element. E.g.

- `/Root/AUTOSAR/ActiveEcuC/EcuC/EcucPduCollection/PduB`
- `/Root/EcuC/EcucPduCollection/PduB`

The name of the difference `getName()` gets the name of the corresponding element. E.g.

`PduB`

The type of the difference `getType()` gets the type of the difference indicating that an element is only available in one of the projects or an element is changed.

The `AsrObjectLink` of the difference `getAutosarObjectLink()` gets the `AsrObjectLink` for the corresponding element of the difference. This method takes into account that the associated model element is not available (only in mine / only in other). In case mine is available the link of mine is returned otherwise other is used.

The values of the difference `getValues()` gets the values of the difference.

6.13.1.7 IDifferenceValues

Represents the values in the projects of a difference.

The value in project Ours `getOurs()` gets the value of project Ours.

The value in project Theirs `getTheirs()` gets the value of project Theirs.

6.13.1.8 Examples

To take only one specific module into account in the comparison, this must already be specified in the comparison configuration: *IProjectCompareConfigBuilder.addModulesToCompare*

```
IProjectCompare projectCompare = projects.activeProject.projectContext[
    IProjectCompare]
IProjectCompareConfigBuilder configBuilder = projectCompare.
    newProjectCompareConfigBuilder(projectToCompareWith)

// We only want to compare 'EcuC' module configuration
configBuilder.addModulesToCompare(List.of("EcuC"))

IProjectCompareResult result = projectCompare.compare(configBuilder.build())
def differences = result.getDifferences()

def nameAndTypeOfDifferences = differences.collect { difference ->
    [difference.name, difference.type]
}
```

Listing 6.357: Specifying a filter for module configurations

To create a report from the comparison result, the following method is used: *IProjectCompareConfigBuilder.createReport* To get the path to the created report, the following method is used: *IProjectCompareResult.getReport*

```
IProjectCompare projectCompare = projects.activeProject.projectContext[
    IProjectCompare]
IProjectCompareConfigBuilder configBuilder = projectCompare.
    newProjectCompareConfigBuilder(projectToCompareWith)

// We want to create a report that contains the differences in the 'EcuC' module
    configuration
configBuilder.addModulesToCompare(List.of("EcuC"))
configBuilder.createReport()

// Retrieve the comparison result - including the report
IProjectCompareResult result = projectCompare.compare(configBuilder.build())
result.getReport()
```

Listing 6.358: Create and retrieve report of the read only project comparison API

6.13.2 Auto merge

The auto merge functionality allows changes from various sources to be automatically merged without the need for manual intervention. This feature is particularly useful in collaborative environments where multiple developers work on the same project simultaneously.

6.13.2.1 Structure

The entry point for the auto merge is the project service **IAutomerge**.

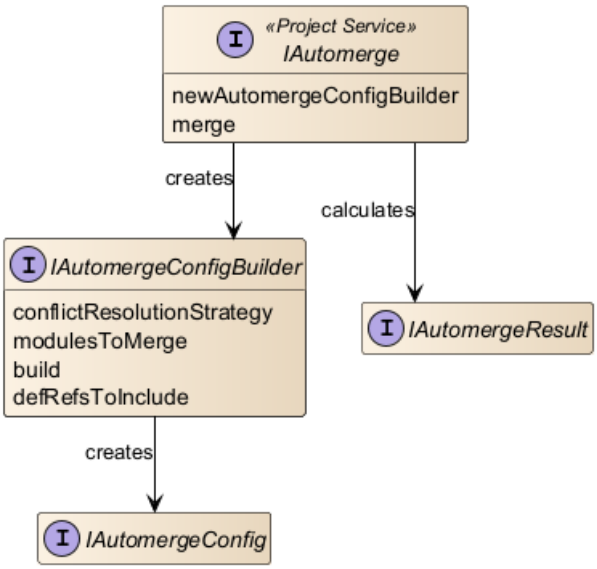


Figure 6.13: Structure of the auto merge interfaces

6.13.2.2 Accessing the API

In order to access the auto merge API the project service **IAutomerge** is used.

```
IAutomerge automerge = projects.activeProject.projectContext[IAutomerge]
IAutomergeConfigBuilder configBuilder = automerge.newAutomergeConfigBuilder(
    projectToCompareWith, projectBase)

automerge.merge(configBuilder.build())
```

Listing 6.359: The general usage of the auto merge API

6.13.2.3 IAutomerge

Represents the entry point for the auto merge API. To configure the API use `IAutomergeConfigBuilder` which can be created via `IAutomerge.newAutomergeConfigBuilder(Object, Object)`. The merge operation can be executed by calling `IAutomerge.merge(IAutomergeConfig)` with the created `IAutomergeConfig`.

Creating new auto merge config builder `newAutomergeConfigBuilder(Object, Object)` creates a new builder instance for the auto merge configuration. Supported are:

- Absolute paths
- Relative paths are resolved to the location of the script

Executing the auto merge `merge(IAutomergeConfig)` executes the auto merge operation with the given `IAutomergeConfig`.

6.13.2.4 IAutomergeConfigBuilder

Represents the configuration builder for the auto merge operation. To create an instance use `IAutomerge.newAutomergeConfigBuilder(Object, Object)` .

Merge only distinct module configurations `addModulesToMerge(List)` adds the short names of the module configurations to merge.

Conflict resolution `setConflictResolutionStrategy(EConflictResolutionStrategy)` sets the conflict resolution strategy which is used in case a conflict is detected (e.g. use value of 'Theirs' or 'Ours').

Include DefRefs `addDefRefsToInclude(List)` adds the `DefRefs` of the elements to merge.

Exclude DefRefs `addDefRefsToExclude(List)` adds the `DefRefs` of the elements to exclude from merge.

Create XML report `setCreateXmlReport()` sets a flag which indicates that a XML report should be created. Please note that the structure of this report may change from version to version.

Create HTML report `setCreateHtmlReport()` sets a flag which indicates that a HTML report should be created. Note that in this case an XML report is also created as the HTML is created based on this. Please note that the structure of this report may change from version to version.

Build comparison config `build()` builds the `IAutomergeConfig` containing all the settings made so far.

For details about the possible filter use cases see also 6.13.2.7 on the next page.

6.13.2.5 IAutomergeResult

Represents the result of an auto merge operation.

Path to XML report `getPathToXmlReport()` gets the path to the generated XML report for not merged differences. See also `IAutomergeConfigBuilder.setCreateXmlReport()`.

Path to HTML report `getPathToHtmlReport()` gets the path to the generated HTML report for not merged differences. See also `IAutomergeConfigBuilder.setCreateHtmlReport()`.

Differences which couldn't be merged `getNotAutomergeableDifferences()` gets the differences which couldn't be merged during the auto merge operation.

6.13.2.6 INotAutomergeableDifference

Represents a difference which couldn't be auto merged as an result of the auto merge operation.

The AUTOSAR object link `getAsrObjectLink()` gets the link to the model element. Note that the link might point to a model element which is not available in the project.

The not merged reasons `getReasons()` gets the reason(s) why the auto merge was not able to merge the difference.

6.13.2.7 Filter Use Cases

With the existing filter options, there are numerous possibilities which will be explained here in detail using examples. **Please note that the code shown in the user note of the images is just pseudocode.**

No filter specified In case no filter is specified all available module configuration are considered during auto merge.

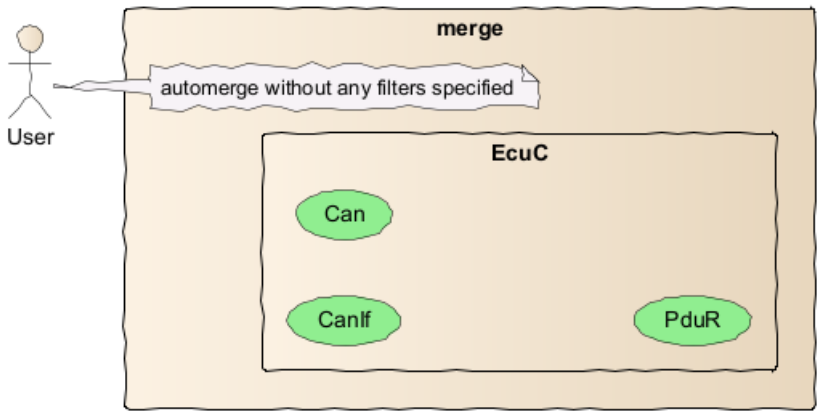


Figure 6.14: Auto merge without any filter specified

Short name of module configuration In case the user specified a distinct module configuration short name only the corresponding module configuration is considered.

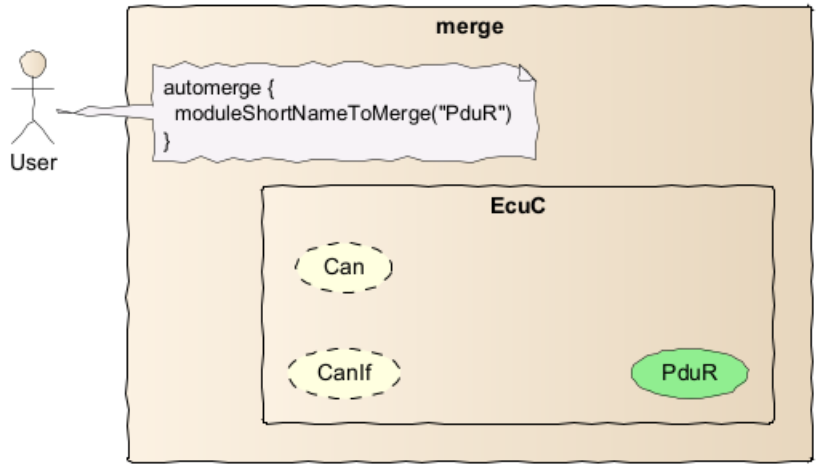


Figure 6.15: Auto merge with short name of module configuration

Include DefRef In case the user specified a distinct (including) DefRef only elements matching this DefRef are considered.

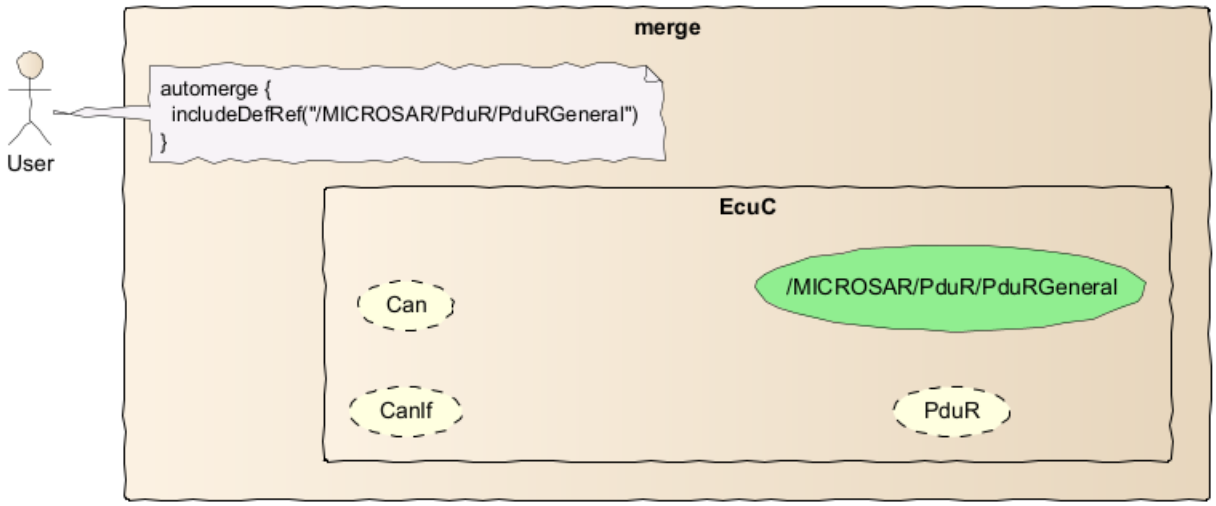


Figure 6.16: Auto merge with including DefRef

Short name of module configuration and include DefRef It's also possible to combine filters like module configuration short name and an include DefRef. Here is an important edge case if the user specified a module configuration short name and a DefRef pointing to the same module configuration. In this case the merge only considers elements matching the specified DefRef.

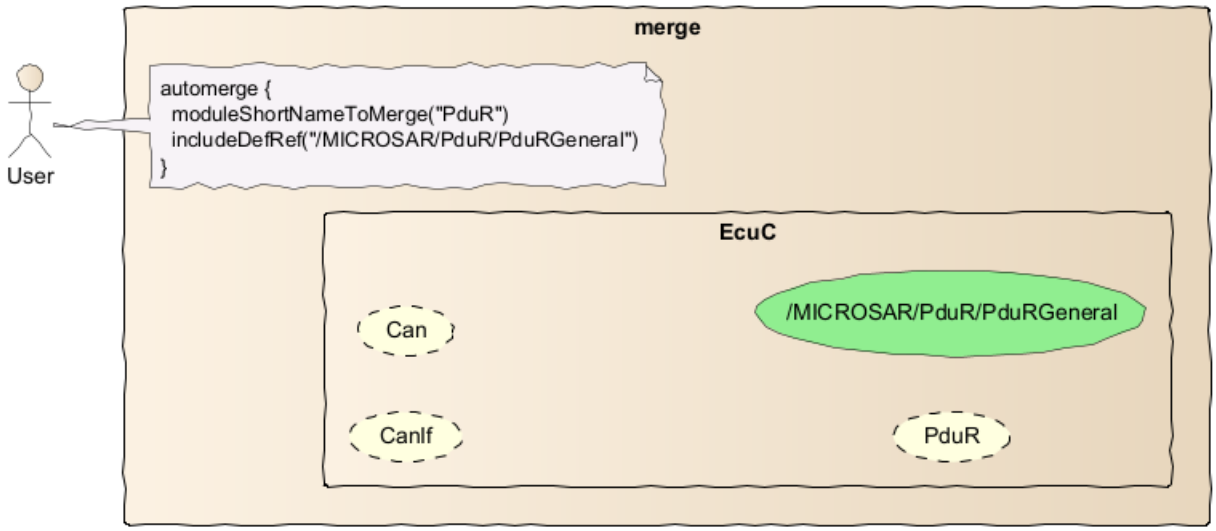


Figure 6.17: Auto merge with module configuration short name and include DefRef

Exclude DefRef In case the user specified a distinct (excluding) DefRef these elements and the sub-elements are ignored. It's not possible to include an element below an excluded element.

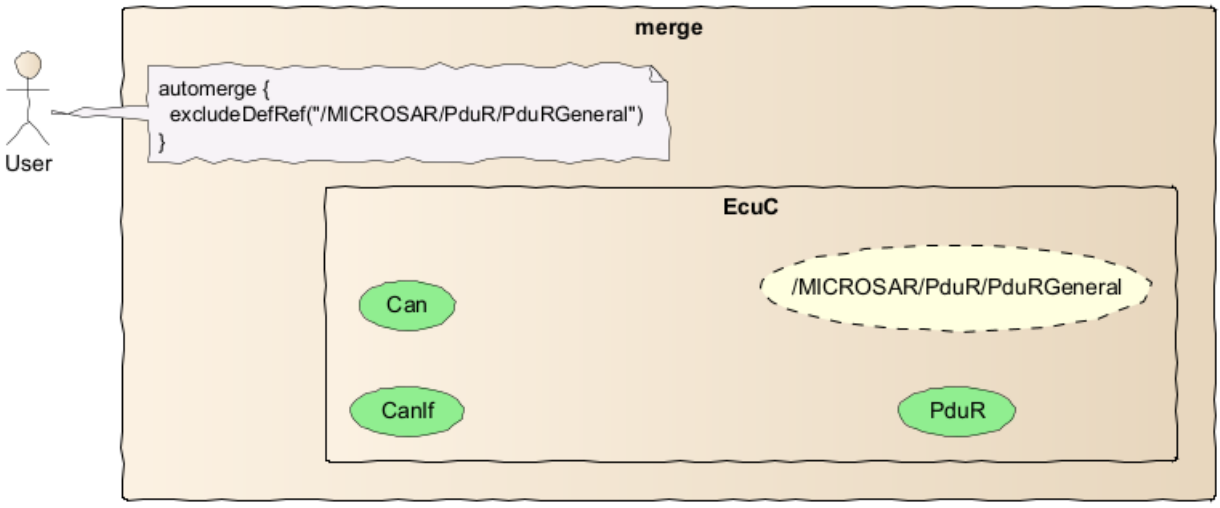


Figure 6.18: Auto merge with excluding DefRef

6.13.3 Unified Diff

The Unified Diff format is a widely used format for displaying differences between two versions of a file. It was developed to present differences in a compact and clear manner by omitting redundant context lines and highlighting only the relevant changes. This format is commonly used in version control systems like Git, Subversion, and others to track changes in source code and other text files.

The Unified Diff format offers several advantages:

- Compactness: By omitting redundant lines, the size of diff files is reduced.
- Readability: Changes are presented in a clear and understandable format, making it easier to review and track modifications.
- Compatibility: It is compatible with many tools and systems that can process and apply diffs.

6.13.3.1 Structure

The entry point for the unified diff creation is the project service **IUnifiedDiff**.

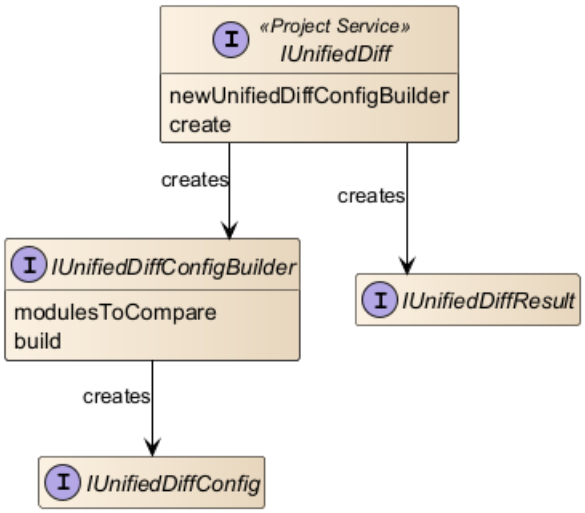


Figure 6.19: Structure of the unified diff creation interfaces

6.13.3.2 Accessing the API

In order to access the unified diff creation API the project service **IUnifiedDiff** is used.

```
IUnifiedDiff unifiedDiff = projects.activeProject.projectContext[IUnifiedDiff]
def resultFile = tempFolder.resolve("MyUnified.diff")
IUnifiedDiffConfigBuilder configBuilder = unifiedDiff.newUnifiedDiffConfigBuilder(
    projectToCompareWith, resultFile)
unifiedDiff.create(configBuilder.build())
```

Listing 6.360: The general usage of the unified diff API

6.13.3.3 IUnifiedDiff

Represents the entry point for the unified diff API. To configure the API use `IUnifiedDiffConfigBuilder` which can be created via `IUnifiedDiff.newUnifiedDiffConfigBuilder(Object, Object)`. The create operation can be executed by calling `IUnifiedDiff.create(IUnifiedDiffConfig)` with the created `IUnifiedDiffConfig`.

Creating new comparison config builder `newUnifiedDiffConfigBuilder(Object, Object)` creates a new builder instance for the unified diff configuration. It's expected that the unified diff result file ends with the '.diff' file extension. Supported are:

- Absolute paths
- Relative paths are resolved to the location of the script

Executing the comparison `create(IUnifiedDiffConfig)` executes the unified diff creation with the given `IUnifiedDiffConfig`.

6.13.3.4 IUnifiedDiffConfigBuilder

Represents the configuration builder for the unified diff creation operation. To create an instance use `IUnifiedDiff.newUnifiedDiffConfigBuilder(Object, Object)`.

Compare only distinct module configurations `addModulesToCompare(List)` adds the short names of the module configurations to compare.

Build comparison config `build()` builds the `IUnifiedDiffConfig` containing all the settings made so far.

6.13.3.5 IUnifiedDiffResult

Represents the result of a unified diff.

This interface currently contains no fields or members but has already been made available for compatibility reasons.

6.14 Project Update API

The `IProjectUpdateApi` provides methods for updating the project configuration..

`applyInputFileChanges` Applies input file changes to the project. It runs the background validation and solves designated solving actions for the project.

`updateRteConfiguration` Updates the Rte configuration for the current state of the system description:

- Synchronizes existence of RteSwComponentInstance and RteSwComponentType container for each used atomic software component.
- Synchronizes existence of RteEventToTaskMapping container for each RTEEvent of each used atomic software component.
- Synchronizes existence of RteExclusiveAreaImplementation container for each exclusive area of each used atomic software component.
- Synchronizes existence of RteNvRamAllocation container for each NvBlockNeeds/NvBlockDescriptor of each used atomic software component.

`automaticReferenceCorrection` Automatically correct references in the project.

`applyEvaluatedVariantSetChanges` Cleans up the model after the EvaluatedVariantSet has been changed.

See examples below to call the Project Update API.

```
scriptTask("ProjectUpdate", DV_PROJECT) {
code {
        activeProject {
            projectUpdate {
                applyInputFileChanges()
                updateRteConfiguration()
                automaticReferenceCorrection()
                applyEvaluatedVariantSetChanges()
            }
        }
    }
}
```

Listing 6.361: Project Update API

6.15 Utilities

6.15.1 Converters

General purpose converters (`java.util.Functions`) for performing value conversions throughout the automation interface are provided. These converters are referenced from the AutomationInterface documentation wherever they apply. The AutomationInterface is typed strongly. In some cases, however, e.g. when specifying file locations, it is desirable to allow for a range of possibly parameter types. This is achieved by accepting parameters of type `Object` and converting the given parameters to the desired type.

The following converters are provided:

ScriptConverters.TO_PATH Attempts to convert arbitrary `Objects` to `Paths` using `IAutomationPathsApi.resolvePath(Object)` 6.4.3.1 on page 41.

ScriptConverters.TO_SCRIPT_PATH Attempts to convert arbitrary `Objects` to `Paths` using `IAutomationPathsApi.resolveScriptPath(Object)` 6.4.3.2 on page 42.

ScriptConverters.TO_VERSION Attempts to convert arbitrary `Objects` to `IVersions`. The following conversions are implemented:

- For `null` or `IVersion` arguments the given argument is returned. No conversion is applied.
- `Strings` are converted using `Version.valueOf(String)`.
- `Numbers` are converted by converting the `int` obtained from `Number.intValue()` using `Version.valueOf(int)`.
- All other `Objects` are converted by converting the `String` obtained from `Object.toString()`.

ScriptConverters.TO_BIG_INTEGER Attempts to convert arbitrary `Objects` to `BigIntegers`. The following conversions are implemented:

- For `null` or `BigInteger` arguments the given argument is returned. No conversion is applied.
- `Integers`, `Longs`, `Shorts` and `Bytes` are converted using `BigInteger.valueOf(long)`.
- All other types of objects are interpreted as `Strings` (`Object.toString()`) and passed to `BigInteger.BigInteger(String)`.

ScriptConverters.TO_BIG_DECIMAL Attempts to convert arbitrary `Objects` to `Doubles`. The following conversions are implemented:

- For `null` or `Double` arguments the given argument is returned. No conversion is applied.
- `Floats` and `Doubles`, are converted using `Double.valueOf(double)`.
- `Integers`, `Longs`, `Shorts` and `Bytes` are converted using `Double.valueOf(double)`.
- All other types of objects are interpreted as `Strings` (`Object.toString()`) and passed to `Double.Double(String)`.

ModelConverters.TO_MDF Attempts to convert arbitrary `Objects` to `MDFObjects`. The following conversions are implemented:

- For `null` or `MDFObject` arguments the given argument is returned. No conversion is applied.
- `IHasModelObjects` are converted using their `IHasModelObject.getMdfObject()` method.
- `IViewedModelObjects` are converted using their `IViewedModelObject.getMdfObject()` method.
- For all other `Objects` `ClassCastExceptions` are thrown.

For thrown `Exceptions` see the used functions described above.

6.16 Advanced Topics

This chapter contains advanced use cases and classes for special tasks.

For a normal script these items are not relevant.

6.16.1 Java Development

It is also possible to write automation scripts in plain Java code, but this is not recommended. There are some items in the API, which need a different usage in Java code.

This chapter describes the differences in the Automation API when used from Java code.

6.16.1.1 Script Task Creation in Java Code

Java code could not use the Groovy syntax to provide script tasks. So another way is needed for this. The `IScriptFactory` interface provides the entry point that Java code could provide script tasks. The `createScript(IScriptCreationApi)` method is called when the script is loaded.

This interface is **not** necessary for Groovy clients.

```
public class MyScriptFactoryAsJavaCode implements IScriptFactory {
    @Override
    public void createScript(final IScriptCreationApi creation) {
        creation.scriptTask("TaskFromFactory", IScriptTaskTypeApi.DV_APPLICATION,
                (taskBuilder) -> {
                    taskBuilder.code(
                            (scriptExecutionContext, taskArgs) -> {
                                // Your script task code here
                                return null;
                            });
                });

        creation.scriptTask("Task2", IScriptTaskTypeApi.DV_PROJECT,
                (taskBuilder) -> {
                    taskBuilder.code(
                            (scriptExecutionContext, taskArgs) -> {
                                // Your script task code for Task2  here
                                return null;
                            });
                });
    }
}
```

Listing 6.362: Java code usage of the IScriptFactory to contribute script tasks

You should try to use Groovy when possible, because it is more concise than the Java code, without any difference at script task creation and execution.

6.16.1.2 Java Code accessing Groovy API

Most of the Automation API is usable from both languages Java and Groovy, but some methods are written for Groovy clients. To use it from Java you have to write some glue code.

Differences are:

- Accessing Properties
- Using API entry points.

- Creating Closures

Accessing Properties Properties are not supported by Java so you have to use the getter/setter methods instead.

API Entry Points Most of the Automation API is added to the object by so called DynamicObjects. This is not available in Java, so you have to call `IScriptExecutionContext.getInstance(Class)` instead. So if you want to access The IWorkflowApi you have to write:

```
//Java code:
IScriptExecutionContext scriptCtx = ...;
IWorkflowApi workflow = scriptCtx.getInstance(IWorkflowApiEntryPoint.class).
    getWorkflow()

//Instead of Groovy code:
workflow {

}
```

Listing 6.363: Accessing WorkflowAPI in Java code

Creating Closure instances from Java lambdas The class `Closures` provides API to create `Closure` instances from Java `FunctionalInterfaces`.

The `from()` methods could be used to call Groovy API from Java classes, which only accepts `Closure` instances.

Sample:

```
Closure<?> c = Closures.from((param) -> {
  // Java lambda
});
```

Listing 6.364: Java Closure creation sample

Creating Closure Instances from Java Methods You could also create arbitrary `Closures` from any Java method with the class `MethodClosure`. This is describe in:
http://melix.github.io/blog/2010/04/19/coding_a_groovy_closure_in.html[1]

6.16.1.3 Java Code in dvgroovy Scripts

It is not possible to write Java classes when using the `.dv.groovy` script file. You have to create an automation script project.

[1]Last accessed 2016-05-24

7 Data models in detail

This chapter describes several details and concepts of the involved data models. Be aware that the information here is focused on the Java API. In most cases it is more convenient using the Groovy APIs described in 6.6 on page 80. So, whenever possible use the Groovy API and read this chapter only to get background information when required.

7.1 MDF model - the raw AUTOSAR data

The MDF model is being used to store the AUTOSAR model loaded from several ARXML files. It consists of Java interfaces and classes which are generated from the AUTOSAR meta-model.

7.1.1 Naming

The MDF interfaces have the prefix `MI` followed by the AUTOSAR meta-model name of the class they represent. For example, the MDF interface related to the meta-model class `ARPackage` (AUTOSAR package in the top-level structure of the meta-model) is `MIARPackage`. The AUTOSAR meta model can be found for example on the AUTOSAR website.

7.1.2 The models inheritance hierarchy

The MDF model therefore implements (nearly) the same inheritance hierarchy and associations as defined by the AUTOSAR model. These interfaces provide access to the data stored in the model.

See figure 7.1 on the next page shows the (simplified) inheritance hierarchy of the ECUC container type `MIContainer`. What we can see in this example:

- A container is an `MIIdentifiable` which again is a `MIReferrable`. The `MIReferrable` is the type which holds the shortname (getName()). All types which inherit from the `MIReferrable` have a shortname (MIARPackage, `MIModuleConfiguration`, ...)
- A container is also a `MIHasContainer`. This is an artificial base class (not part of the AUTOSAR meta-model) which provides all features of types which have sub-containers. The `MIModuleConfiguration` therefore has the same base type
- A container also inherits from `MIHasDefinition`. This is an artificial base class (not part of the AUTOSAR meta-model) which provides all features of types which have an AUTOSAR definition. The `MIModuleConfiguration` and `MIParameterValue` therefore has the same base type
- All `MIIdentifiable`s can hold ADMIN-DATA and ANNOTATIONs
- All MDF objects in the AUTOSAR model tree inherit from `MIObject` which is again an `MIObject`

7.1.2.1 MIObject and MDFObject

The `MIObject` is the base interface for all AUTOSAR model objects in the DaVinci Configurator data model. It extends `MDFObject` which is the base interface of all model objects. Your client code shall always use `MIObject`, when AUTOSAR model objects are used, instead of `MDFObject`.

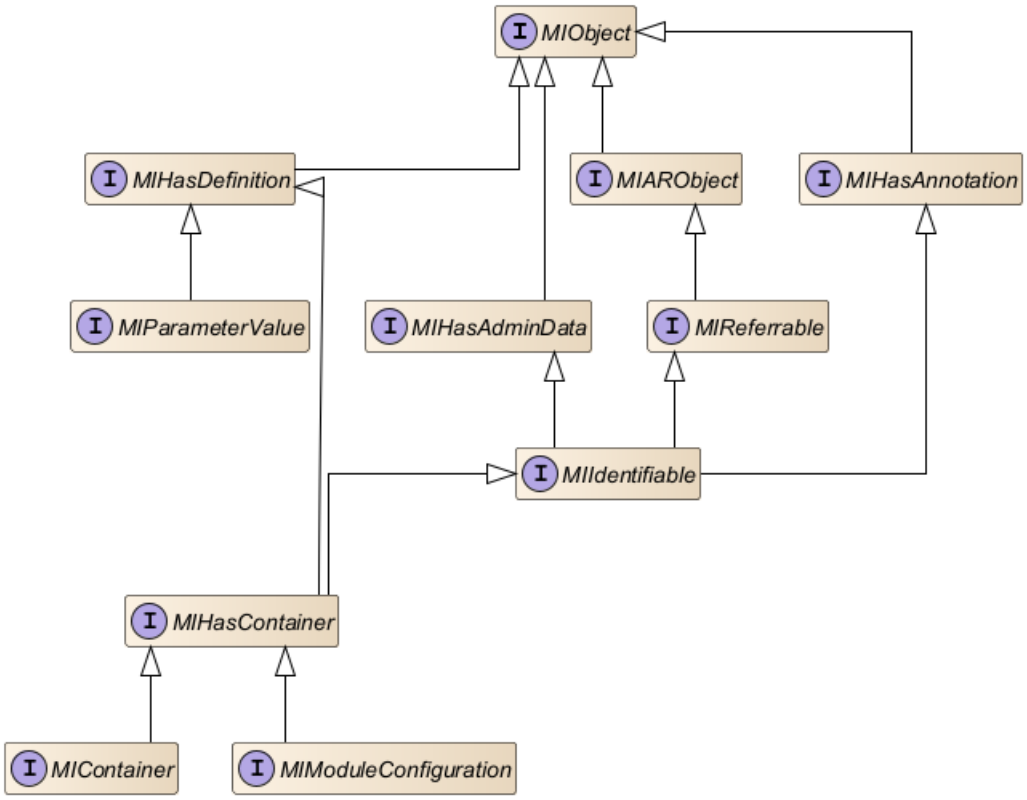


Figure 7.1: ECUC container type inheritance

The figure 7.2 describes the class hierarchy of the `MIObject`.

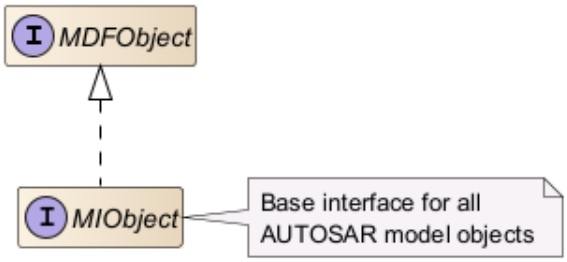


Figure 7.2: MIObject class hierarchy and base interfaces

7.1.3 The models containment tree

The root node of the AUTOSAR model is `MIAUTOSAR`. Starting at this object the complete model tree can be traversed. `MIAUTOSAR.getSubPackage()` for example returns a list of `MIARPackage` objects which again have child objects and so on.

Figure 7.3 on the next page shows a simple example of an MDF object containment hierarchy. This example contains two AUTOSAR packages with module configurations below.

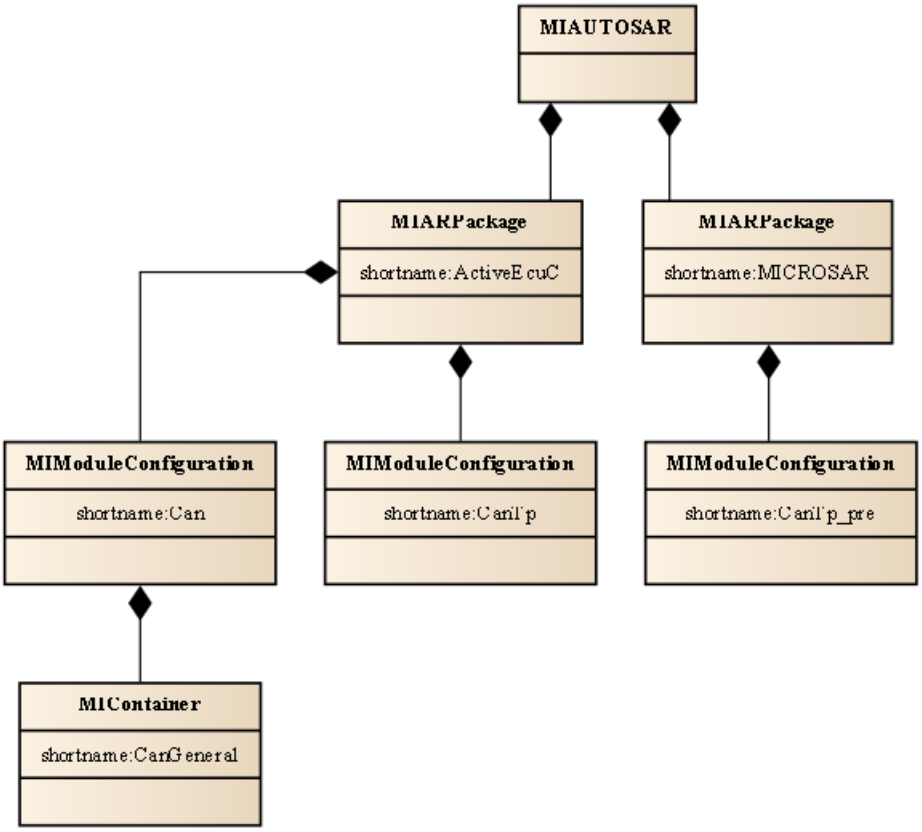


Figure 7.3: Autosar package containment

In general, objects which have child objects provide methods to retrieve them.

- `MIAUTOSAR.getSubPackage()` for example returns a list of child packages
- `MIContainer.getSubContainer()` returns the list of sub-containers and `MIContainer.getParameter()` all parameter-values and reference-values of a container

7.1.4 The ECUC model

The interfaces and classes which represent the ECUC model don't exactly follow the AUTOSAR meta-model naming. because they are designed to store AUTOSAR 3 and AUTOSAR 4 models as well.

Affected interfaces are:

- `MIModuleConfiguration` and its child objects (containers, parameters, ...)
- `MIModuleDef` and its child objects (containers definitions, parameter definitions, ...)

The ECUC model also unifies the handling of parameter- and reference-values. Both, parameter-values and reference-values of a container, are represented as `MIParameterValue` in the MDF model.

7.1.5 Order of child objects

Child object lists in the MDF model have the same order as the data specified in the ARXML files. So, loading model objects from AXRML doesn't change the order.

7.1.6 AUTOSAR references

All AUTOSAR reference objects in the MDF model have the base interface `MIARRef` and the base generic interface `MIGARRef`.

Figure 7.4 shows the type hierarchy for the AUTOSAR reference.

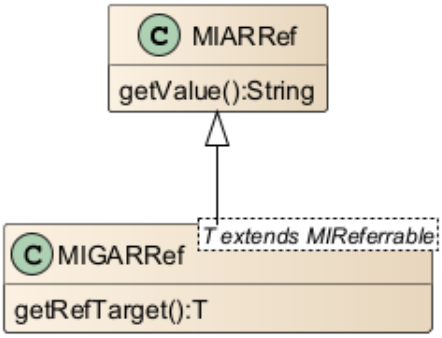


Figure 7.4: The ECUC container definition reference

In ARXML, such a reference can be specified as:

```
<TYPE-TREF DEST="IMPLEMENTATION-DATA-TYPE">
    /DataTypes/MyImplDataType
</TYPE-TREF>
```

- `MIARRef.getValue()` returns the AUTOSAR path of the object, the reference points to (as specified in the ARXML file). In the example above `"/DataTypes/MyImplDataType"` would be this value
- `MIGARRef.getRefTarget()` on the other hand returns the referenced MDF object if it exists. This method is located in a specific, typesafe (according to the type it points to) reference interface which extends `MIARRef`. So, if an object with the AUTOSAR path `"/DataTypes/MyImplDataType"` exists in the model, this method will return it

7.1.7 Model changes

7.1.7.1 Transactions

The MDF model provides model change transactions for grouping several model changes into one atomic change.

A solving action, for example, is being executed within a transaction for being able to change model content. Validation and generator developers don't need to care for transactions. The tools framework mechanisms guarantee that their code is being executed in a transaction were required.

The tool guarantees that model changes cannot be executed outside of transactions. So, for example, during validation of model content the model cannot be changed. A model change here would lead to a runtime exception.

7.1.7.2 Undo/redo

On basis of model change transactions, MDF provides means to undo and redo all changes made within one transaction. A GUI can allow users to execute undo/redo on this granularity.

7.1.7.3 Event handling

MDF also supports model change events. All changes made in the model are reported by this asynchronous event mechanism. Validations, for example, detect this way which areas of the model need to be re-validated. The GUI can use this mechanism to update its editors and views when model content changes.

7.1.7.4 Deleting model objects

Model objects must be deleted by a dedicated API. In Java code that's: `MIObject.deleteFromModel()`.

Deleting an object means:

- All associations of the object are deleted. The connection to its parent object, for example, is being deleted which means that the object is not a member of the model tree anymore
- The object itself is being deleted. In fact, it is not really deleted (and garbage collected) as a Java object but only marked as removed. Undo of the transaction, which deleted this object, removes this marker and restores the deleted associations

7.1.7.5 Access to deleted objects

All subsequent access to content of deleted objects throws a runtime exception. Reading the shortname of an `MIContainer`, for example.

7.1.7.6 Set-methods

Model interfaces provide get-methods to read model content. MDF also offers set-methods for fields and child objects with multiplicity `0..1` or `1..1`.

These set-methods can be used to change model content.

- `MIARRef.getValue()` for example returns a references AUTOSAR path
- `MIARRef.setValue(String newValue)` sets a new path

7.1.7.7 Changing child list content

MDF doesn't offer set-methods for fields and child objects with multiplicity `0..*` or `1..*`. `MIContainer.getSubContainer()`, for example, returns the list of sub-containers but there is no `MIContainer.setSubContainer()` method to change the sub-containers.

Changing child lists means changing the list itself.

- To add a new object to a child list, client code must use the lists add() method. `MIContainer.getSubContainer().add(container)`, for example, adds a container as additional sub-container. This added object is being appended at the end of the list
- Removing child list objects is a side-effect of deleting this object. The delete operation removes it from the list automatically

7.1.7.8 Change restrictions

The tools transaction handling implements some model consistency checks to avoid model changes which shall be avoided. Such changes are, for example:

- Creating duplicate shortnames below one parent object (e.g. two sub-containers with the same shortname)
- Changing or deleting pre-configured parameters

When client code tries to change the model this way, the related model change transaction is being canceled and the model changes are reverted (unconditional undo of the transaction). A special case here are solving actions. When a solving action inconsistently changes the model, only the changes made by this solving action are reverted (partial transaction undo of one solving action execution).

7.2 Post-build selectable

7.2.1 Model views

7.2.1.1 What model views are

After project load, the MDF model contains all objects found in the ARXML files. Variation points are just data structures in the model without any special meaning in MDF.

If you want to deal with variants, you must use model views. A model view filters access to the MDF model based on the variant definition and the variation points.

There is one model view per variant. If you use this variants model view, the MDF model filters exactly what this variant contains. All other objects become invisible. When your retrieve parameters of a container for example, you'll see only parameters contained in your selected variant.

```
final boolean isVisible = t.paramVariantA.moIsVisible();
```

Listing 7.1: Check object visibility

7.2.1.2 The IModelViewManager project service

The `IModelViewManager` handles model visibility in general. It provides the following means:

- Get all available variants
- Execute code with visibility of a specific predefined variant only. This means your code sees all objects contained in the specified variant. All objects which are not contained in this variant will be invisible
- Execute code with visibility of invariant data only (see `IInvariantView`).
- Execute code with unfiltered model visibility. This means that your code sees all objects unconditionally. If the project contains variant data, you see all variants together

It additionally provides detailed visibility information for single model objects:

- Get all variants, a specific object is visible in
- Find out if an object is visible in a specific variant

```
final List<IPostBuildPredefinedVariantView> variants = viewMgr.
    getAllPostBuildVariantViews();
```

Listing 7.2: Get all available variants

```
final List<IPostBuildView> allVaraintsOrInvariantView = viewMgr.
    getAllPostBuildVariantViewsOrInvariant();
```

Listing 7.3: Get all available variants or if project does not contain variants the invariant view

```
try (final IModelViewExecutionContext context = viewMgr.executeWithModelView(t.
    variantViewA)) {
    assertIsVisible(t.paramInvariant);
    assertIsVisible(t.paramVariantA);
    assertNotVisible(t.paramVariantB);
}

try (final IModelViewExecutionContext context = viewMgr.executeWithModelView(t.
    variantViewB)) {
    assertIsVisible(t.paramInvariant);
    assertNotVisible(t.paramVariantA);
    assertIsVisible(t.paramVariantB);
}

try (final IModelViewExecutionContext context = viewMgr.executeUnfiltered()) {
    assertIsVisible(t.paramInvariant);
    assertIsVisible(t.paramVariantA);
    assertIsVisible(t.paramVariantB);
}
```

Listing 7.4: Execute code with variant visibility

Important remark: It is essential that the `execute...()` methods are used exactly as implemented in the listing above. The `try (...) {...}` construct is a new Java 7 feature which guarantees that resources are closed whenever (and how ever) the try block is being left. For details read:
`http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html`

```
Collection<IPostBuildPredefinedVariantView> visibleVariants = viewMgr.
    getVisiblePostBuildVariantViews(t.paramInvariant);
assertThat(visibleVariants.size(), equalTo(2));
assertThat(visibleVariants, containsInAnyOrder(t.variantViewA, t.variantViewB));

visibleVariants = viewMgr.getVisiblePostBuildVariantViews(t.paramVariantA);
assertThat(visibleVariants.size(), equalTo(1));
assertThat(visibleVariants, containsInAnyOrder(t.variantViewA));
```

Listing 7.5: Get all variants, a specific object is visible in

7.2.1.3 Variant siblings

Variant siblings of an MDF object are MDF object instances which represent the same object but in other variants.

The method `IModelVarianceAccessPublished.getPostBuildVariantSiblings()` provides access to these sibling objects:

This method returns MDF object instances representing the same object but in all variants. The collection returned contains the object itself including all siblings from other PostBuild variants.

The calculation of siblings depends on the object-type as follows:

- **Ecuc Module Configuration:**
  Since module configurations are never variant, this method always returns a collection which contains the specified object only
- **Ecuc Container:**
  For siblings of a container all of the following conditions apply:
  - They have the same AUTOSAR path
  - They have the same definition path (containers with the same AUTOSAR path but different definitions may occur in variant models - but they are not variant siblings because they differ in type)
- **Ecuc Parameter:**
  For siblings of a parameter all of the following conditions apply:
  - The parent containers have the same AUTOSAR path
  - The parameter siblings have the same definition path

  The parameter values are **not** relevant so parameter siblings may have different values. Multi-instance parameters are special. In this case the method returns all multi-instance siblings of all variants.
- **System description object:**
  For siblings of `MIReferrable`s all of the following conditions apply:
  - They have the same meta-class
  - They have the same AUTOSAR path

  For siblings of non-`MIReferrable`s all of the following conditions apply:
  - Their nearest `MIReferrable`-parents are either the same object or variant siblings
  - Their containment feature paths below these nearest `MIReferrable`-parents is equal

Special use cases: When the specified object is not a member of the model tree (the object itself or one of its parents has no parent), it also has no siblings. In this case this method returns a collection containing the specified object only.

Remark concerning visibility: This method returns all siblings independent of the currently visible objects. This means that the returned collection probably contains objects which are not visible by the caller! It also means that the specified object itself doesn't need to be visible for the caller.

7.2.1.4 The Invariant model views

There are use cases which require to see the invariant model content only. One example are generators for modules which don't support variance at all.

There are two different invariant views currently defined:

- **Value based invariance** (values *are equal* in all variants):
  The `IPostBuildInvariantValuesView` contains objects were all variant siblings have the same value and exist in all variants.
- **Definition based invariance** (values which *shall be equal* in all variants):
  The `IPostBuildInvariantEcucDefView` contains objects which are not allowed to be variant according to the BSWMD rules.

All Invariant views derive from the same interface `IInvariantView`, so if you want to use an invariant view and not specifying the exact view, you could use the `IInvariantView` interface. The figure 7.5 describes the hierarchy.

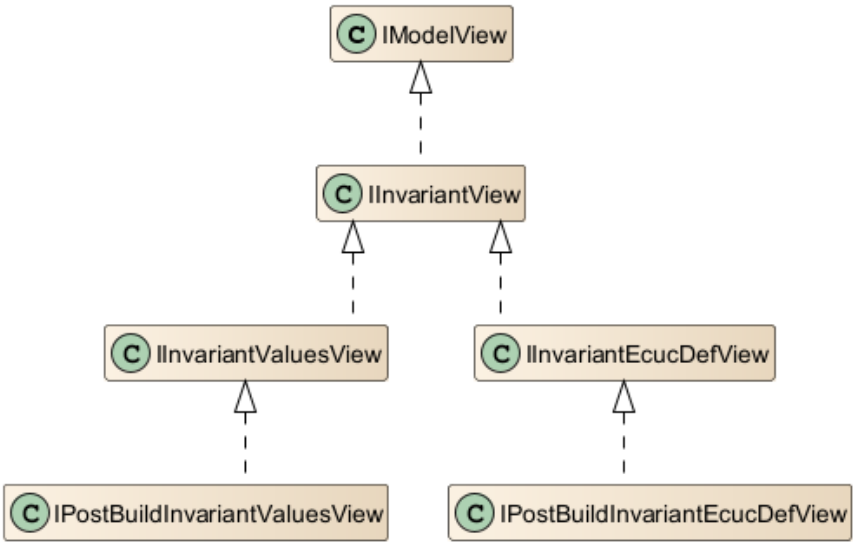


Figure 7.5: Invariant views hierarchy

The PostBuild InvariantValues model view The `IPostBuildInvariantValuesView` contains only elements which have **one** of the following properties:

- The element and no parent has any `MIVariationPoint` with a post-build condition
- All variant siblings have the same value and exist in all variants. Then one of the siblings is contained in the `IPostBuildInvariantValuesView`

So the semantic of the InvariantValues model view is that all values are equal in all variants.

You could retrieve an instance of `IPostBuildInvariantValuesView` by calling `IModelViewManager. getInvariantValuesView()`.

```
IModelViewManager viewMgr =...;
IPostBuildInvariantValuesView invariantView = viewMgr.
    getPostBuildInvariantValuesView();
// Use the invariantView like any other model view
```

Listing 7.6: Retrieving an InvariantValues model view

Example The figure 7.6 describes an example for a module with containers and the visibility in the `IPostBuildInvariantValuesView`.

- Container A is invisible because it is contained in variant 1 only
- Container B and C are visible because they are contained in all variants
- Parameter a is visible because it is contained in all variants with the same value
- Parameter b is invisible: It is contained in all variants but with different values
- Parameter c is invisible because it is contained in variant 3 only

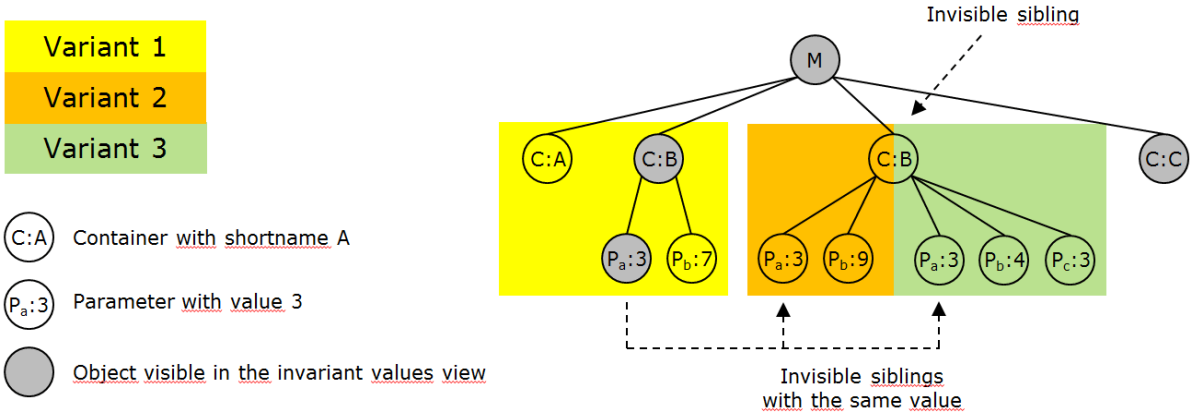


Figure 7.6: Example of a model structure and the visibility of the IInvariantValuesView

Specification See also the specification for details of the `IPostBuildInvariantValuesView`.

The PostBuild Invariant EcuC definition model view The `IPostBuildInvariantEcucDefView` contains the same objects as the invariant values view but additionally excludes all objects which, by (EcuC / BSWMD) definition, support variance. Using this view you can avoid dealing with objects which are accidentally equal by value (in your test configurations) but potentially can be different because they support variance.

More exact the `IPostBuildInvariantEcucDefView` will additionally exclude elements which have the following properties:

- If the parent module configuration specifies VARIANT-POST-BUILD-SELECTABLE as implementation configuration variant
  - All objects (`MIContainer`, `MINumericalValue`, ...) are *excluded*, which **support** variance according to their EcuC definition. (potentially variant objects)

- If the parent module configuration doesn't specify VARIANT-POST-BUILD-SELECTABLE as implementation configuration variant. All contained objects **do not** support variance, so the view actually shows the same objects as the `IPostBuildInvariantValuesView`.

The implementation configuration variant in fact overwrites the objects definition for elements in the `ModuleConfiguration`.

Reasons to Use the view The EcucDef view guarantees that you don't access potentially variant data without using variant specific model views. So it allows you to improve code quality in your generator.

When your test configuration for example contains equal values for a parameter which is potentially variant you will see this parameter in the invariant values view but not in the EcucDef view. Consequences if you access data in other module configurations: When the BSWMD file of this other module is being changed, e.g. a parameter now supports variance, objects can become invisible due to this change. You are forced to adapt your code then.

Usage You could retrieve an instance of `IPostBuildInvariantEcucDefView` by calling `IModelViewManager.getInvariantEcucDefView()`. And then use is as any other `IModelView`.

```
IModelViewManager viewMgr =...;
IInvariantEcucDefView invariantView = viewMgr.getInvariantEcucDefView();
// Use the invariantView like any other model view
```

Listing 7.7: Retrieving an InvariantEcucDefView model view

Specification See also the specification for details of the `IPostBuildInvariantEcucDefView`.

7.2.1.5 Accessing invisible objects

When you switch to a model view, objects which are not contained in the related variant become invisible. This means that access to their content leads to an `InvisibleVariantObjectFeatureException`.

To simplify handling of invisible objects, some model services provide model access even for invisible objects in variant projects. The affected classes and interfaces are:

- `com.vector.cfg.model.asr.ecuc.access.IEcucReferrableAccess`
- `com.vector.cfg.model.asr.ecuc.access.IEcucModelAccess`
- `com.vector.cfg.model.asr.ecuc.compare.IModelEquivalenceService`
- `com.vector.cfg.model.access.AsrPath`
- `com.vector.cfg.model.access.DefRef`
- `com.vector.cfg.model.asr.ecuc.access.ecucdefinition.IEcucDefinitionAccess` (all methods which deal with configuration side objects)

Only a subset of the methods in these services work with invisible objects (read the methods JavaDoc for details). The general policy to select exactly these methods was:

- Support access to type and object identity of MDF objects (definition and AUTOSAR path)

- Parameter value or other content related information must still be retrieved in a context the object is visible in
- Also not contained are methods which change model content. E.g. deleting invisible objects, set parameter values, ...

7.2.1.6 IViewedModelObject

The `IViewedModelObject` is a container for one `MIObject` and an `IModelView` that was used when viewing the `MIObject`.

The interface provides getter for the `MIObject`, and the `IModelView` which was active during creation of the `IViewedModelObject`. So the `IViewedModelObject` represents a tuple of `MIObject` and `IModelView`.

This could be used to preserve the state/tuple of a `MIObject` and `IModelView`, for later retrieval.

Examples:

- BswmdModel objects
- Elements for validation results, retrieved in a certain view
- Model Query API like ModelTraverser, to preserve `IModelView` information

Notes:
A `IViewedModelObject` is immutable and will not update any state. Especially not when the visibility of the `getMdfObject()`, is changed after the construction of the `IViewedModelObject`.

It is not guaranteed, that the `MIObject` is visible in the creation `IModelView`, after the model is changed. It is also possible to create an `IViewedModelObject` of a `MIObject` and a `IModelView`, where the `MIObject` is invisible.

The method `getCreationModelView()` returns the `IModelView` of the `IViewedModelObject`, which was active when the model object was viewed `IViewedModelObject`.

7.2.1.7 Default Model View

Default model view when nothing is set is the IPostBuildInvariantValuesView.

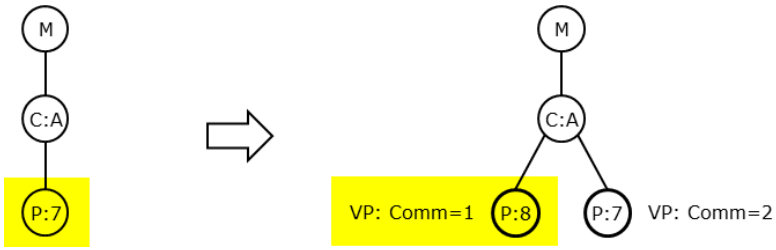
7.2.2 Change Modes

7.2.2.1 Variant Specific Model Changes

The data model provides an execution context which guarantees that only the selected variant is being modified. Objects which are visible in more than one variant are cloned automatically. The clones and the object which is being modified (or their parents) automatically get a variation point with the required post-build conditions.

The following picture shows how this execution context works:
See figure 7.7 on the next page.

- Before modifying the parameter, this instance is invariant. The same MDF instance is visible in all variants
- When the client code changes the parameter value, the model automatically clones the parameter first



Criterion	Left	Right
Comm	1	2

Figure 7.7: Variant specific change of a parameter value

- Only the parameter instance which is visible in the currently active view is being modified. The content of other variants stays untouched

Remark: This change mode is implicitly turned off when executing code in the `IInvariantView` or in an unfiltered context.

```
try (final IModelViewExecutionContext viewContext = viewMgr.executeWithModelView(
    variantView)) {
    try (final IModelViewExecutionContext modeContext = viewMgr.
        executeWithVariantSpecificModelChanges()) {
        ma.setAsString(parameter, "Vector-Informatik");
    }
}
```

Listing 7.8: Execute code with variant specific changes

7.2.2.2 Variant Common Model Changes

The data model provides an execution context which guarantees that model objects are modified in all variants.

The behavior of this mode depends on the mode flag parameter as follows:

- **mode == ALL** : All parameters and containers are affected
- **mode == DEFINITION_BASED** : Only those parameters and containers are affected which do not support variance (according to their definition in the BSWMD file and the implementation configuration variant of their module configuration)
- **mode == OFF** : Doesn't turn on this change mode (this value is used internally only)

Remark: This method doesn't allow to reduce the scope of this change mode. So if ALL is already set, this method doesn't permit to use DEFINITION_BASED (or OFF) to reduce the effective amount of objects. ALL will be still active then.

The following picture shows how this execution context works:
See figure 7.8 on the following page.

- We start with a variant model which contains one parameter in two instances - one per variant - with the values 3 and 7
- When the client code sets the parameter value in variant 1 to 4, the model automatically modifies the variant sibling in variant 2
- As a result, the parameter has the same value in all variants

This change mode works with parameters and containers. The following operations are supported:

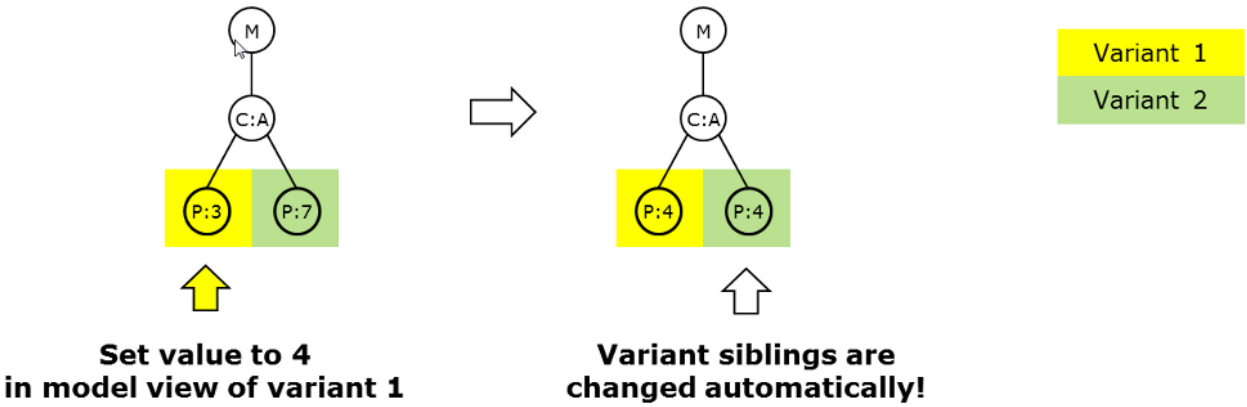


Figure 7.8: Variant common change of a parameter value

- **Container/parameter creation:** The created object afterwards exists in all variants the related parent exists in. Already existing objects are not modified. Missing objects are created
- **Container/parameter deletion:** The deleted object afterwards is being removed from all variants the related parent exists in. So actually all variant siblings are deleted
- **Parameter value change:** The parameter exists and has the same value in all variants the parent container exists in. If a parameter instance is missing in a variant, it is being created

Special behavior for multi-instance parameters:

- This mode guarantees that a set of multi-instance parameters is equal in all variants
- Only the values of multi-instance parameters are relevant. Their order can be different in different variants
- Beside the values, this change mode guarantees that all variants contain the same number of parameter instances. So, when a multi-instance set is being modified in a variant view, this change mode creates or deletes objects in other variants to guarantee an equal number of instances in all variant sibling sets

Remark: This change mode is implicitly turned on with the mode flag ALL when code is being executed in the `IInvariantView`. It is being ignored implicitly when executing code in an unfiltered context.

7.2.2.3 Default Change Mode

Default change modes when nothing is set are:

- EVariantSpecificMode.ON
- EVariantCommonMode.DEFINITION_BASED

7.3 BswmdModel details

7.3.1 BswmdModel - DefinitionModel

The BswmdModel provides a type safe and easy access to data of BSW modules (Ecu configuration elements).

Example:

- Access a single parameter `/MICROSAR/ComM/ComMGeneral/ComMUseRte`
  You can to write: `comM.getComMGeneral().getComMUseRte()`
- Access containers[0:*] `/MICROSAR/ComM/ComMChannel`
  You can to write:

```
for (ComMChannel channel : comM.getComMChannel()){
    int value = channel.getComMChannelId().getValue();
}
```

The DaVinci Configurator Classic internal Model (MDF model) has 1:1 relationship to your BswmdModel. The BswmdModel will retrieve all data from the underlying MDF model.

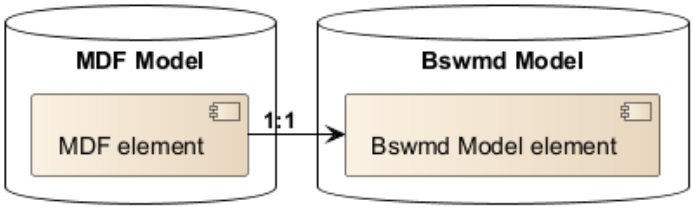


Figure 7.9: The relationship between the MDF model and the BswmdModel

DefinitionModel The DefinitionModel is the base implementation of every BswmdModel. Every BswmdModel class is a subclass of the DefinitionModel where the classes begin with `GI`, like `GIContainer`.

7.3.1.1 Types of DefinitionModels

There are two types of DefinitionModels:

1. **BswmdModel** (formally known as DefinitionTyped BswmdModel)
2. **DefRef API** (formally known as Untyped BswmdModel)

The **BswmdModel** consists of generated classes for the module definition elements like `ModuleDefinitions, Containers, Parameters` in bswmd files. The generated class contains getter methods for each child element. So you can access every child by the corresponding getter method with compile time safety of the sub type.

The **BswmdModel** derives from the **DefinitionModel DefRef API**, so the **BswmdModel** contains all functionalities of the **DefRef API**.

The **DefRef API** of the DefinitionModel provides a generic access to the Ecu configuration structure via `DefRefs`. There are **NO** generated classes for the Definition structure. The **DefRef API** uses the base classes of the DefinitionModel to provide this `DefRef` based access. Every interface in the DefinitionModel starts with an `GI`. The Ecu Configuration elements have corresponding base interfaces for each element:

- ModuleConfiguration - `GIModuleConfiguration`
- Container - `GIContainer`

- ChoiceContainer - `GIChoiceContainer`
- Parameter - `GIParameter<?>`
  - Integer Parameter - `GIParameter<BigInteger>`
  - Boolean Parameter - `GIParameter<Boolean>`
  - Float Parameter - `GIParameter<Double>`
  - String Parameter - `GIParameter<String>`
- Reference - `GIReference<?>`
  - Container Reference - `GIReferenceToContainer`
  - Foreign Reference- `GIReference<Class>`

So there are different classes for the different model types, e.g. all MDF classes start with MI, the Untyped start with GI and DefinitionTyped classes are generated. The table 7.1 contrasts the different model types and their corresponding classes.

AUTOSAR type	MDFModel	"Untyped" BswmdModel	"DefinitionTyped"
ModuleConfiguration	`MIModuleConfiguration`	`GIModuleConfiguration`	`CanIf` (generated)
Container	`MIContainer`	`GIContainer`	`CanIfPrivateCfg` (generated)
String Parameter	`MITextualValue`	`GIParameter<String>`	`GString`
Integer Parameter	`MINumericalValue`	`GIParameter<BigInteger>`	`GInteger`
Reference to Container	`MIReferenceValue`	`GIReferenceToContainer`	`CanIfCtrlDrvInitHohConfigRef` (generated)
Enum Parameter	`MITextualValue`	`GIParameter<String>`	`CanIfDispatchBusOffUL` (generated)

Table 7.1: Different Class types in different models

Note: The `GString` in the table is not the Groovy `GString` class.
It is `com.vector.cfg.gen.core.bswmdmodel.param.GString`.

7.3.1.2 DefRef Getter methods of Untyped Model

The DefRef API classes have no getter methods for the specific child types, but the children can be retrieved via the generic getter methods like:

- `GIContainer.getSubContainers()`
- `GIContainer.getParameters()`
- `GIContainer.getParameters(TypedDefRef)`
- `GIContainer.getParameter(TypedDefRef)`
- `GIContainer.getReferencesToContainer(TypedDefRef)`
- `GIModuleConfiguration.getSubContainer(TypedDefRef)`
- `GIParameter.getValueMdf()`

Additionally there are methods to retrieve other referenced elements, like parent of reference reverse lookup:

- `GIContainer.getParent()`
- `GIContainer.getParent(DefRef)`

- `GIContainer.getReferencesPointingToMe()`
- `GIContainer.getReferencesPointingToMe(DefRef)`

The following listings describe the usage of the untyped bswmd method in both models:

```
// Get the container from external method getCanIfInitConfigBswmd() ...
final GIContainer canIfInit = getCanIfInitConfigBswmd();

// Gets all subcontainers from a container CanIfRxPduConfig from the canIfInit
    instance
final List<GIContainer> subContainers = canIfInit.getSubContainers(
    CanIfRxPduConfig.DEFREF.castToTypedDefRef());
if (subContainers.isEmpty()) {
    // ERROR Handling
}
final GIContainer cont = subContainers.get(0);

// Gets exactly one CanIfCanRxPduHrhRef reference from the cont instance
final GIReference<MIContainer> child = cont.getReference(CanIfCanRxPduHrhRef.
    DEFREF.castToTypedDefRef());
```

Listing 7.9: Sample code to access element in an Untyped model with DefRefs

```
final GIReferenceToContainer ref = getCanIfCanRxPduHrhRefBswmd();

final GIContainer target = ref.getRefTarget();
```

Listing 7.10: Resolves a Reference target of a Reference Parameter

```
final GIParameter<BigInteger> param = getCanIfInitConfigBswmd().getParameter(
    CanIfInitConfiguration.CANIF_NUMBER_OF_CAN_TXPDU_IDS_DEFREF);

final BigInteger value = param.getValueMdf();
```

Listing 7.11: The value of a GIParameter

Figure 7.10 on the following page shows the available `DefRef` navigation methods for the Untyped model. There are more methods to navigate with the DefRef API through the DefinitionModel, please look into the Javadoc documentation of the `GI...` classes for more functionality.

Figure 7.10: SubContainer DefRef navigation methods

7.3.1.3 References

All references in the BswmdModel are subtypes of `GIReference`. The generated model contains generated DefintionTyped classes for references to container, for the other references there are only Untyped classes like `GInstanceReference`.

A `GIReference` has the method `getRefTargetMdf()`, this will always return the target in the MDF model as `MIReferrable`. For non `GIReferenceToContainer` this is the normal way to resolve references, but for references to a container you should always try to use the method `getRefTarget()`, which will not leave the BswmdModel.

Note: Try to use `getRefTarget()` as much as possible.

References to container The following references are references to a container (References pointing to container) and are subtypes of the `GIReferenceToContainer`.

- Normal Reference

- SymbolicNameReference
- ChoiceReference

References have the method `getRefTarget()`, which returns the target as BswmdModel object. If the type of the target is known at model generation time, the return type will be the generated type, otherwise the return type is `GIContainer`.

Note: It is always allowed to call `getRefTarget()`, also for references pointing to external types.

There is the other method `getPossibleRefTargets()`, which returns all possible target container as list. If the type of the targets is known at model generation time, the list type will be the generated type, e.G. `List<CanGeneral>`. Otherwise the return type is `List<GIContainer>`.

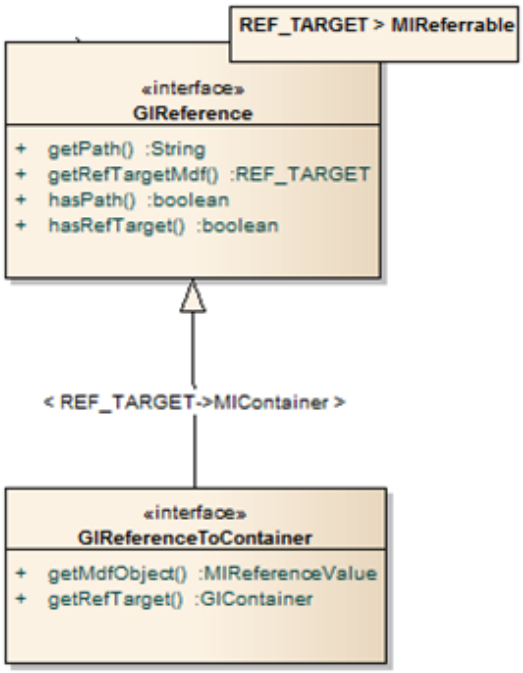


Figure 7.11: Untyped reference interfaces in the BswmdModel

SymbolicNameReferences SymbolicNameReferences have the same methods as `GIReference-ToContainer` and the additional methods `getRefTargetParameterMdf()`, which returns the target parameter as `MIObject` The method `getRefTargetParameter()` return a BswmdModel object, if the type is known at model generation time, the type will be the generated type. Otherwise the return type is `GIParameter`.

Note: It is always allowed to call `getRefTargetParameter()`, also for references pointing to external types.

7.3.1.4 Post-build selectable with BswmdModel

The BswmdModel supports the Post-build selectable use case, in respect that you do not have to switch nor cache the corresponding `IModelView`. The BswmdModel objects cache the so called Creation ModelView and switch transparently to that view when accessing the Model. So you don't have to switch to the correct view on access. See figure 7.12 on the next page. You only have to ensure, that the requested `IModelView` is active or passed as parameter, when you create an

instance at the `GIModelFactory`. Note: A lazy created object will inherit the view of the existing element.

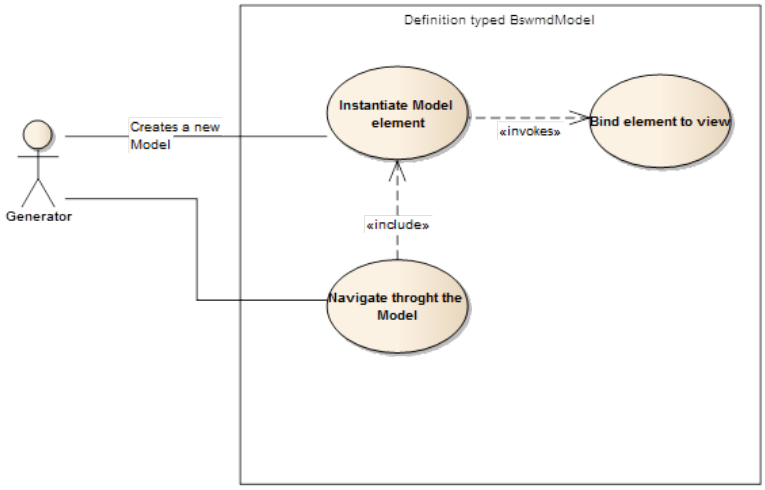


Figure 7.12: Creating a BswmdModel in the Post-build selectable use case

7.3.1.5 Creation ModelView of the BswmdModel

Every `GIModelObject` (BswmdModel object) has a creation `IModelView`. This is the `IModelView`, which was active or passed during creation of the BswmdModel. At every method call to the BswmdModel, the model will switch to this view.

Using the creation ModelView of the BswmdModel The method `getCreationModelView()` returns the `IModelView` of this `GIModelObject`, which was active during the creation of this BswmdModel.

The method `executeWithCreationModelView()` executes the code under visibility of the `getCreationModelView()` of this `GIModelObject`.

The returned `IModelViewExecutionContext` must be used within a Java "try-with-resources" block. It makes sure, that the old view is restored when the try is completed.

```
GIModelObject myModelObject = ...;

try (final IModelViewExecutionContext context = myModelObject.
    executeWithCreationModelView()) {
        // do some operations
        ...
}
```

Listing 7.12: Java: Execute code with creation IModelView of BswmdModel object

The method `executeWithCreationModelView(Runnable)` executes the `Runnable` code under visibility of the `getCreationModelView()` of this `GIModelObject`.

```
GIModelObject myModelObject = ...;

myModelObject.executeWithCreationModelView(()->{
 // do some operations
});
```

Listing 7.13: Java: Execute code with creation IModelView of BswmdModel object via runnable

The method `executeWithCreationModelView()` executes the `Supplier` code under visibility of the `getCreationModelView()` of this `GIModelObject`. You could use this method, if you want to return an object from this operation.

```
GIModelObject myModelObject = ...;

ReturnType returnVal = myModelObject.executeWithCreationModelView(()->{
  // do some operations
  return theValue;
 });
```

Listing 7.14: Java: Execute code with creation IModelView of BswmdModel object

7.3.1.6 Lazy Instantiating

The BswmdModel is instantiated lazily; this means when you create a ModuleConfiguration object only one object for the module configuration is created.

When you call a `getXXX()` method on the configuration it will create the requested sub element, if it exists. So you can start at any point in the model (e.g. a Subcontainer) and the model is built successively by your calls.

It is also allowed to call `getParent()` on a Subcontainer, if the parent was not created yet. The technique could be used in validations, when the creation of the full BswmdModel is too expensive. Then you can create only the needed container by an MDF model object.

7.3.1.7 Optional Elements

All elements (Container, Parameter ...) are considered as optional if they have a multiplicity of 0:1. The BswmdModel provides a special handling of optional elements. This shall support you to recognize optional elements during development (in most cases some kind of special handling is needed). An optional Element has other access methods as a required Element: The method `getXXX()` will not return the element, it will return a `GIOptional<Element>` object instead. You can query the `GIOptional` object if the element exists (`optElement.exists()`). Then `optElement.get()` can be called to retrieve the real object.

You also have the choice to use the method `existsXXX()` to check for element existence. This method is equivalent to `getXXX().exists()`. The difference is that you get a compile error, if you try to use the optional element without any check. When you are sure that the element must exist you can directly call `getXXXUnsafe()`. Note: If you use any of the get methods (`optElement.get()` or `getXXXUnsafe()`) and the element does not exist the normal `BswmdModelException` is thrown.

7.3.1.8 Class and Interface Structure of the BswmdModel

The upper part of the figure 7.13 on the following page shows the Untyped API (GI... interfaces). The bottom left part is an example of DefinitionTyped (generated) class for the CanIf module.

The bottom right part are the classes used by the DefinitionTyped model, but are not visible in the Untyped model.

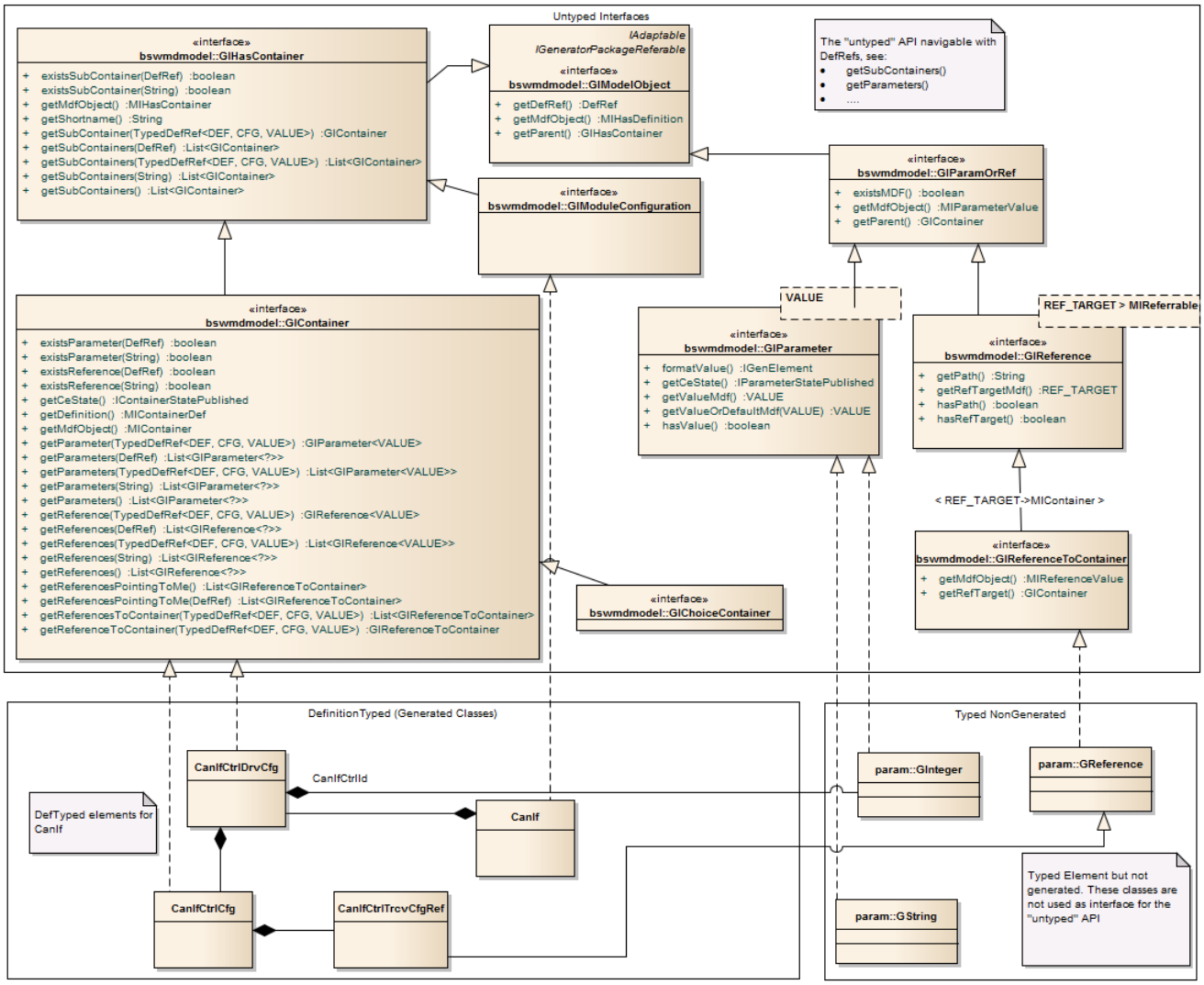


Figure 7.13: Class and Interface Structure of the BswmdModel

7.3.1.9 BswmdModel Write Access

The BswmdModel supports a write access for ECU configuration elements. This means new elements can be created and existing elements can be modified and deleted by the BswmdModel.

NOTE: The model is in read-only state by default, so no objects can be created. For this reason all calls to an API which creates or deletes elements has to be executed within a transaction.

Optional and required Elements (0:1/1:1 Multiplicity) For optional or required elements, the following additional methods are generated, if BswmdModelWriteAccess is enabled:

- `get...OrNull()`: Returns the requested element or `null` if it is missing.
- `get...OrCreate()`: Returns the existing requested element or implicitly creates a new one if it is missing.

E.g. `EcucGeneral`:

```
Ecuc ecuc = getEcucModuleConfig();

//Gets the EcucGeneral container or null if it is missing.
EcucGeneral ecucGeneralOrNull = ecuc.getEcucGeneralOrNull();

//Gets the existing EcucGeneral container or creates a new one if it is missing.
EcucGeneral ecucGeneralOrCreate = ecuc.getEcucGeneralOrCreate();
```

Listing 7.15: Additional write API methods for EcucGeneral

Multiple elements (Upper Multiplicity > 1) For each multiple element, the return type for these elements is changed from `List<>` to `GIPList<>` for parameter and `GICList<>` for container, if BswmdModelWriteAccess is enabled. These new interfaces provide methods which allow creating and adding new children for the corresponding elements:

- `createAndAdd()`: Creates a new child element, appends it to the list and returns the new element.
- `createAndAdd(int index)`: Creates a new child element, inserts it to the list at the specified index position and returns the new element.
- For `GICList<>` only:
  - `createAndAdd(String shortName)`: Creates a new child element with the specified shortName, appends it to the list and returns the new element.
  - `createAndAdd(String shortName, int index)`: Creates a new child element with the specified `shortName`, inserts it to the list at the specified index position and returns the new element.
  - `byName(String shortName)`: Gets the container by specified shortName or throws an exception if it is missing.
  - `byNameOrNull(String shortName)`: Gets the container by specified shortName or `null` if it is missing.
  - `byNameOrCreate(String shortName)`: Gets the container by specified shortName or implicitly creates a new one if it is missing.
  - `exists(String shortname)`: Returns `true` if the container exists, otherwise `false`.

The following example creates `EcucCoreDefinition` via the Write API:

```
Ecuc ecuc = getEcucModuleConfig();

//Gets the EcucCoreDefinition list (create EcucHardware container if it is missing
    )
GICList<EcucCoreDefinition> ecucCores = ecuc.getEcucHardwareOrCreate().
    getEcucCoreDefinition();

//Adds two EcucCores
EcucCoreDefinition core0 = ecucCores.createAndAdd("EcucCore0");
EcucCoreDefinition core1 = ecucCores.createAndAdd("EcucCore1");

if(ecucCores.exists("EcucCore0")){
    //Sets EcucCoreId from EcucCore0 to 0
    ecucCores.byName("EcucCore0").getEcucCoreId().setValue(0);
}

//Creates a new EcucCore by method byNameOrCreate
EcucCoreDefinition core2 = ecucCores.byNameOrCreate("EcucCore2");

...
```

Listing 7.16: EcucCoreDefinition as GICList<EcucCoreDefinition>

Other write API

- **Deleting model objects**: It is also possible to delete objects from the model.
  - `moRemove`: Deletes the specified object from the model.
  - `moIsRemoved`: Returns `true`, if the object was removed from repository, or is invisible in the current active `IModelView`.

```
//Deletes the container 'EcucGeneral' from the model.
ecucGeneral.moRemove();

//Deletes the parameter 'EcuCSafeBswChecks' from the model.
ecucGeneral.getEcuCSafeBswChecks.moRemove();

//Deletes the child container 'EcucCoreDefinition' with shortname 'EcucCore0' from
     the model.
ecucCores.byName("EcucCore0").moRemove();

// Checks if the container 'EcucGeneral' was removed from repository, or is
    invisible in the current active `IModelView`.
if(ecucGeneral.moIsRemoved()){
    ...
}
```

Listing 7.17: Deleting model objects

- **Duplication of containers**: The method `duplicate()` copies a container with all its children and appends it to the same parent.

```
//Duplicates the container 'EcucGeneral'
EcucGeneral duplicatedEcucGeneral = ecucGeneral.duplicate();

//Duplicates the child container 'EcucCoreDefinition' with shortname 'EcucCore0'
var duplicatedEcucCore0 = ecucCores.byName("EcucCore0").duplicate();
```

Listing 7.18: Duplication of containers

- **Parameter values**: The method `setValue(VALUE)` sets the value of a parameter. This method checks if the specified parameters configuration object is available and sets the new value. If the parameter object is missing it is implicitly created in the model.

```
//Sets the value of the parameter 'EcuCSafeBswChecks' to 'true'
ecucGeneral.getEcuCSafeBswChecks.setValue(true);
```

Listing 7.19: Set parameter values with the BswmdModel Write API

- **Reference targets**: The method `setRefTarget(REF_TARGET)` sets the target of a reference. This method sets the specified target object as reference target of the specified reference parameter. If the reference parameter object is missing it is implicitly created in the model.

```
//Gets the container 'OsCounter' with shortname 'SystemTimer'
OsCounter osCounterTarget = os.getOsCounters.byName("SystemTimer");

//Sets the reference target of the parameter 'CanCounterRef'
can.getCanGeneral().getCanCounterRef().setRefTarget(osCounterTarget);
```

Listing 7.20: Set reference targets with the BswmdModel Write API

7.3.1.10 BswmdModel Declaration API

The BswmdModel supports declaration API to declare an AUTOSAR ECU configuration structure in code, which is then synchronized with the existing structure to create elements in a declarative way.

Note: The model is in read-only state by default, so no objects can be created or synchronized. You must have an open transaction running, when using the Declaration API, because it will change the model.

Entrypoint into Declaration You can enter the Declaration API on every module configuration or container with the method `declare{}`. Inside the `declare{}` block you use the Declaration API.

```
can.declare {
    //Inside here you can use the Declaration API
}
```

Listing 7.21: Start declaration API on a Module

Usage of the Declaration API on an existing container:

```
CanGeneral canGeneral = can.canGeneral
canGeneral.declare { CanGeneralDeclaration decl ->
    //Inside here you can use the Declaration API
}
```

Listing 7.22: Start declaration API on any existing container

API Structure Every module or container class has a corresponding `<ElementName>Declaration` class, which is used to declare the structure of the module or container tree.

Every `<Name>Declaration` class defines the methods to declare its direct child containers and child parameters:

- `<ContainerName>(Action)` method: For `0:1` or `1:1` child containers
- `<ContainerName>(String shortname, Action)` method: For `0:*` child containers
- `<ParameterName>(value)` method: For `0:1` or `1:1` child parameters
- `<ParameterName>(values...)` method: For `0:*` child parameters
- `<ReferenceName>(referenceTarget)` method: For `0:1` or `1:1` child references
- `<ReferenceName>(referenceTargets...)` method: For `0:*` child references

The declaration methods will return the created or existing BswmdModel element (`GIxxx`), which was used by the called declaration. So the `CanGeneral{}` method returns the `CanGeneral` container. This can be useful, if you need the element as a reference target for a reference pointing to that container.

Semantics The Declaration API does **not** clean the underlying AUTOSAR model, it tries to synchronize the existing model with your declared structure. If you want to declare a new structure, you have to delete (with `moRemove()`) the model element before declaring elements.

So a call to the `CanGeneral{}` in the `Can` module with multiplicity `1:1` will use the existing container. If no container exists, a new container will be created with the name `CanGeneral`.

```
can.declare {
    CanGeneral {
        //Declares the CanGeneral container
    }
}
```

Listing 7.23: Container declaration with 0:1 or 1:1 multiplicity

For the `0:*` multiplicity, the Declaration API tries to find the container with the given shortname, otherwise it will create the container with the shortname. That is the reason why `0:*` containers need a shortname in the API.

```
can.declare {
    CanConfigSet("Config1") {
        //Declares "Config1" CanConfigSet container
    }
    CanConfigSet("Config2") {
        //Declares "Config2" CanConfigSet container
    }
}
```

Listing 7.24: Container declaration with upper multiplicity > 1

All `0:1` and `1:1` parameters and references are automatically created, when their values are set. And also existing parameters are synchronized with the new values.

```
can.declare {
    CanGeneral {
        //Declares 0:1 or 1:1 parameters
        CanDevErrorDetection(true)
        CanInterruptLock(ECanInterruptLock.APPL)
        CanGenericConfirmation(false)
    }
}
```

Listing 7.25: Parameter declaration with 0:1 or 1:1 multiplicity

For `0:*` parameters and references, the Declaration API will synchronize the existing parameters with the new values by index. If there is no parameter for that index, a new parameter will be created. But existing ones are not deleted. The `0:*` parameters have a vararg parameter, where the index in the vararg array is the index of the resulting parameter.

```
can.canGeneral.declare {
    CanMainFunctionRWPeriods {
        //Declares 3 parameters of type CanMainFunctionReadPeriod
        CanMainFunctionReadPeriod(10, 30, 50)
    }
}
```

Listing 7.26: Parameter declaration with upper multiplicity > 1

Interop with BswmdModel and MDF Model The Declaration API provides interop methods to switch into the normal BswmdModel or the MDF model for each element. You can use the methods `getBswmdModelObject()` or `getMdfObject()` to switch to them.

```
can.declare {
    CanGeneral {
        //Switch into the normal BswmdModel API
        CanGeneral theBswmdObject = bswmdModelObject
        def theObjectLink = bswmdModelObject.objectLink
        //Switch into the MDF model
        MIContainer theMdfObj = mdfObject
    }
}
```

Listing 7.27: Declaration API interop

You can also interleave other model operation code with the Declaration API code. The Declaration API will synchronize the model directly at the method call and the BswmdModel will reflect the change immediately.

The method `setShortname()` can be used to rename an object.

```
can.declare {
    CanGeneral {
        //Get the current shortName
        String theShortName = shortname
        //Set shortName of the container
        shortname = theShortName + "New"
    }
}
```

Listing 7.28: Container shortname API

The Declaration API uses the underlying BswmdModel Write Access to synchronize the model, so the same post-build selectable write semantics apply.

Usage Sample The following sample declares a structure on the `Can` module with the Declaration API.

```
can.declare {

    CanGeneral {
        CanDevErrorDetection(true)
        CanGetStatus(false)
        CanIdenticalIdCancellation(false)
        CanInterruptLock(ECanInterruptLock.APPL)
        CanIndex(5)
        CanGenericPreTransmit(false)
    }

    CanConfigSet("CanConfigSet") {
        CanController("Controller1") {
            CanControllerId(1)
            CanBusName("CanBus1")

            def baud1 = CanControllerBaudrateConfig("Baud1") {
                CanSamplingMode(ECanSamplingMode.OneSample)
            }
            //Second Baudrate config
            CanControllerBaudrateConfig("Baud2") {
                CanSamplingMode(ECanSamplingMode.ThreeSamples)
            }

            CanControllerDefaultBaudrate(baud1) //Assign a refTarget to reference
        }

        CanController("Controller2") {
            CanControllerId(2)
        }
    }
}
```

Listing 7.29: Example BswmdModel Declaration API on a Can module

7.3.2 BswmdModel generation

The BswmdModel for the automation interface is generated automatically by the DaVinci Configurator.

7.3.2.1 DerivativeMapping

If the BSW Package contains one or more modules with a DerivativeMapping, the BswmdModel classes for these modules can only be generated for one certain derivative. By default, the first derivative is selected, sorted by UUID.

If a other derivative shall be selected for BswmdModel generation a `Settings.xml` file can be defined in the BSW Package at `<SIP-ROOT-PATH>/DaVinciConfigurator/Generators`.

Sample file:

```
<Settings>
    <Settings Name="com.vector.cfg.bswmdmgen.BswmdAutomationModelSettings">
              <!--Selects the derivative with the name or UUID specified by Value
                  -->
              <Setting Name="SelectedDerivative" Value="SPX546B"/>
    </Settings>
</Settings>
```

Listing 7.30: Settings.xml sample for DerivativeMapping

7.4 Model Utility Classes

7.4.1 AutosarUtil

The class `AutosarUtil` is a static utility class. Its methods are not directly related to the MDF model but are useful when client code deals with AUTOSAR paths and shortnames on string basis. Some of these methods are

- `isValidShortname(String)`: Checks if this shortname is valid according the rules, the AUTOSAR standard defines (character set for example)
- `getLastShortname(String)`: Returns the last shortname of the specified AUTOSAR path
- `getFirstShortname(String)`: Returns the first shortname of the specified AUTOSAR path
- `getAllShortnames(String)`: Returns all shortnames of the specified AUTOSAR path

7.4.2 AsrPath

The `AsrPath` class represents an AUTOSAR path without a connection to any model.

`AsrPath`s are constant; their values cannot be changed after they are created. This class is immutable!

```
// String based APIs
final AsrPath genericPath = AsrPath.create("/a/b");
final AsrPath relativeChildPathWithParent = AsrPath.create(genericPath, "e");
final AsrPath absoluteChildPathWithParent = AsrPath.create(genericPath, "/a/b/e");

// Error tolerant APIs
final String pathMayBeInvalid = "??/invalid/??";
final AsrPath mayBeInvalidPath = AsrPath.tryCreate(pathMayBeInvalid);
if (mayBeInvalidPath != null) {
    // work on valid path
}

// retrieve the model element: connect to loaded model via project context
final MIReferrable referrableAutosarObject = genericPath.getAutosarObject(
    getProjectContext());
if (referrableAutosarObject != null) {
    // work with autosar object
}
```

Listing 7.31: AsrPath methods

7.4.3 TypedAsrPath

Typed version of an `AsrPath` to represent an AUTOSAR path and its expected metamodel class.

```
// String based APIs
final TypedAsrPath<MIContainer> containerAsrPath = AsrPath.create("/ActiveEcuC/
    PduR/PduRRoutingTables/PduRRoutingPathGroup", MIContainer.class);
final TypedAsrPath<MIModuleConfiguration> moduleAsrPath = AsrPath.create("/
    ActiveEcuC/PduR", MIModuleConfiguration.class);
final TypedAsrPath<MIContainer> pduRRoutingTables = AsrPath.create(moduleAsrPath,
    "PduRRoutingTables", MIContainer.class);

// Error tolerant APIs
final String pathMayBeInvalid = "??/invalid/??";
final TypedAsrPath<MIContainer> mayBeInvalidPath = AsrPath.tryCreate(
    pathMayBeInvalid, MIContainer.class);
if (mayBeInvalidPath != null) {
    // work on valid path
}

// check model elements: connect to loaded model via project context
final MIReferrable referrable = getReferrable();
if (containerAsrPath.isPathOf(referrable)) {
    // referrable has correct path and type
    final MIContainer container = (MIContainer) referrable;
}
```

Listing 7.32: TypedAsrPath methods

7.4.4 AsrObjectLink

This class implements an immutable identifier for AUTOSAR objects.

An `AsrObjectLink` can be created for each object in the MDF AUTOSAR model tree. The main use case of object links is to identify an object unambiguously at a specific point in time for logging reasons. Additionally and under specific conditions it is also possible to find the related

MDF object using its AsrObjectLink instance. But this search-by-link cannot be guaranteed after model changes (details and restrictions below).

7.4.4.1 Restrictions of object links

- They are immutable and will therefore become invalid when the model changes
- So they don't guarantee that the related MDF object can be retrieved after the model has been changed. Search-by-link may even find another object or throw an exception in this case

7.4.5 DefRefs

The `DefRef` class represents an AUTOSAR definition reference (e.g. `/MICROSAR/CanIf`) without a connection to any model. A `DefRef` replaces the `String` which represents a definition reference. You shall always use a `DefRef` instance, when you want to reference something by it's definition.

The class abstracts the behavior of definition references in the AUTOSAR model (e.g. AUTOSAR 3 and AUTOSAR 4 handling).

DefRefs are constant; their values can not be changed after they are created. All `DefRef` classes are immutable.

A `DefRef` represents the definition reference as two parts:

- Package part - e.g. `/MICROSAR`
- Definition without the package part - e.g. `CanIf/CanIfGeneral`

This is used to navigate through the AUTOSAR model with refinements and wildcards. So you have to create a `DefRef` with the two parts separated.

The figure 7.14 shows the structure of the `DefRef` class and its sub classes.

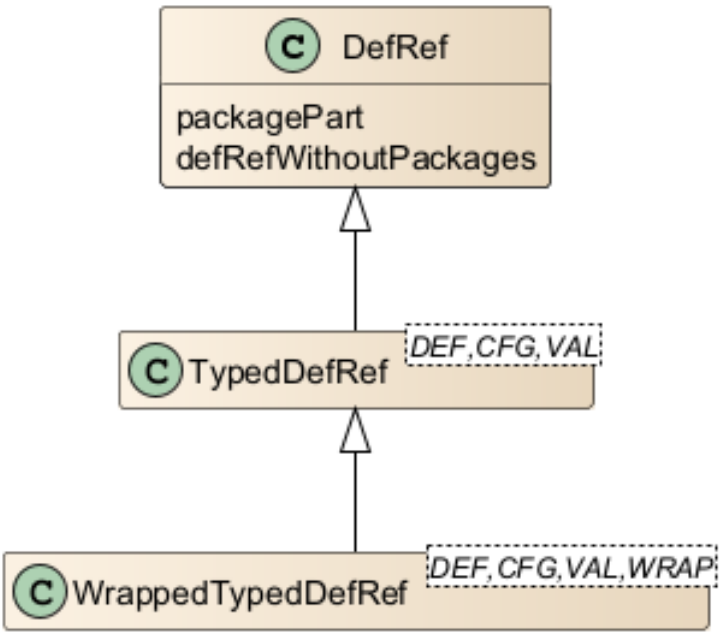


Figure 7.14: DefRef class structure

Creation You can create a DefRef object with following public static methods (partial):

- `DefRef.create(DefRef, String)` - Parent DefRef, Child name
- `DefRef.create(IDefRefWildcard, String)` - Wildcard, Definition without package
- `DefRef.create(MIHasDefinition)` - Model object
- `DefRef.create(MIHasDefinition, String)` - Parent object, Child name
- `DefRef.create(MIParamConfMultiplicity)` - Definition object
- `DefRef.create(String, String)` - Package part, Definition without package

Wildcards `DefRef` instances can also have a wildcard instead of a package `String` (`IDefRefWildcard`). The wildcard is used to match on multiple packages. See chapter 7.4.5.2 on the following page for details.

Useful Methods This section describes some useful methods (Please look at the javadoc of the DefRef class for a full documentation):

- `defRef.isDefinitionOf(MIHasDefinition)` - Checks the definition of the configuration element and returns true if the element has the definition. The "defRef" object is e.g. from the Constants class.
  - Note: The method `isDefinitionOf()` returns `false`, if the element is removed or invisible.
- `defRef.asDefinitionOf(MIHasDefinition, Class)` - Checks the definition of the configuration element and returns the element casted to the configuration subtype, or null.
  - Note: The method `asDefinitionOf()` returns `null`, if the element is removed or invisible.

```
MIObject yourObject = ...;
DefRef yourDefRef = ...;

if(yourDefRef.isDefinitionOf(yourObject){
     //It is the correct instance
     //Do something
}

//Or with an integrated cast in the TypedDefRef case
final MIContainer container = yourDefRef.asDefinitionOf(yourObject);
if(container  != null){
     //Do something
}
```

Listing 7.33: DefRef isDefinitionOf methods

7.4.5.1 TypedDefRefs

The `TypedDefRef` class represents an AUTOSAR definition reference with the type of the AUTOSAR (MDF) model. So every `TypedDefRef` knows which Definition, Configuration and Value element is correct for the Definition path.

The `DEF_TYPE`, `CONFIG_TYPE` and `VALUE_TYPE` are Java generics and are used many APIs to return the specific type of a request.

In addition the most TypedDefRefs also provide additional TypeInfo data, like the Multiplicity of the element. See TypeInfo javadoc for more details.

7.4.5.2 DefRef Wildcards

The DefRef class supports so called wildcards, which could be used to match on multiple packages at once, like the /[MICROSAR] wildcard matches on any DefRef package starting with /MICROSAR. E.g. /MICROSAR, /MICROSAR/S12x, ...

Every wildcard is of type IDefRefWildcard. An IDefRefWildcard instance could be passed to the DefRef.create(IDefRefWildcard, String) method to create a DefRef with wildcard information.

Predefined DefRef Wildcards The class EDefRefWildcard contains the predefined IDefRefWildcards for the DefRef class. These IDefRefWildcards could be used to create DefRefs, without creating your own wildcard for the standard use cases

The DefRef.create(String, String) method will parse the first String to find a wildcard matching the EDefRefWildcards.

Predefined wildcards: The class EDefRefWildcard defines the following wildcards, with the specified semantic:

- EDefRefWildcard.ANY /[ANY]: Matches on any package path. It is equal to any package and any packages refines from ANY wildcard.
- EDefRefWildcard.AUTOSAR /[AUTOSAR]: Matches on the AUTOSAR3 and AUTOSAR4 packages (see DefRef class). It is equal to the AUTOSAR packages, but not to refined packages e.g. /MICROSAR. Any packages which refined from AUTOSAR also refines from AUTOSAR wildcard.
- EDefRefWildcard.NOT_AUTOSAR_STMD /[!AUTOSAR_STMD]: Matches on any package except the AUTOSAR packages. It is equal to any package, except AUTOSAR packages. Any package refines from NOT_AUTOSAR_STMD wildcard, except AUTOSAR packages.
- EDefRefWildcard.MICROSAR /[MICROSAR]: Matches on any package stating with /MICROSAR (also /MICROSAR/S12x). It is equal to any package stating with /MICROSAR. Any package starting with /MICROSAR refines from MICROSAR wildcard.
- EDefRefWildcard.NOT_MICROSAR /[!MICROSAR]: Matches on any package path not starting with /MICROSAR. It is equal to any package not starting with /MICROSAR. Any package, which does not start with /MICROSAR, refines from NOT_MICROSAR wildcard. Also the AUTOSAR packages refine from NOT_MICROSAR wildcard.

Creation of the DefRef with Wildcard The elements of EDefRefWildcard could be passed to the DefRef constructor:

```
DefRef myDefRef = DefRef.create(EDefRefWildcard.MICROSAR, "CanIf");
```

Listing 7.34: Creation of DefRef with wildcard from EDefRefWildcard

Custom DefRef Wildcards You could create your own wildcard by implementing the interface `IDefRefWildcard`. Please choose a good name for your wildcard, because this could be displayed to the user, e.g. in Validation results. The `matches(DefRef)` method shall return true, if the passed DefRef matches the wildcard constraints.

Every wildcard string shall have the notation `/[NameOfWildcard]`.
E.g. `/[MICROSAR]`, `/[!MICROSAR]`.

7.4.6 CeState

The `CeState` is an object which allows to retrieve different states of a configuration entity.

The most important APIs for generator and script code are:

- `IParameterStatePublished`
- `IContainerStatePublished`

7.4.6.1 Getting a CeState object

The BSWMD models implement methods to get the CeState for a specific CE as the following listing shows (the types `GIParameter` and `GIContainer` are interface base types in the BSWMD models):

```
GIParameter parameter = ...;
IParameterStatePublished parameterState = parameter.getCeState();

GIContainer container = ...;
IContainerStatePublished containerState = container.getCeState();
```

Listing 7.35: Getting CeState objects using the BSWMD model

7.4.6.2 IParameterStatePublished

The `IParameterStatePublished` specifies a type-safe published API for parameter states. It mainly covers the following state information

- Does this parameter have a pre-configuration value? What is this value? The same information is being provided for recommended and initial (derived) values
- Is this parameter user-defined?
- Is value change or deletion allowed in the current configuration phase (post-build loadable use case)?
- What is the configuration class of this parameter

The figure 7.15 on the next page shows the inheritance hierarchy of the `IParameterStatePublished` class and its sub classes.

Parameters have different types of state information:

- **Simple state retrieval**
  Example: The method `isUserDefined()` returns true when the parameter has a user-defined flag.
- **States and values** (pre-configuration, recommended configuration and inital (derived) values)

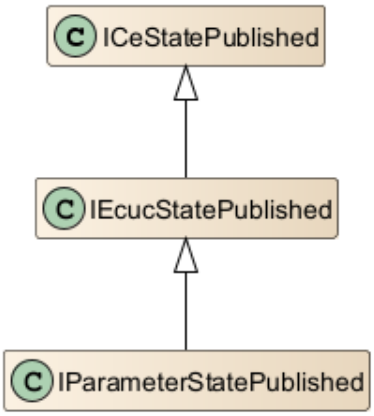


Figure 7.15: IParameterStatePublished class structure

Example: The method `hasPreConfigurationValue()` returns true when the parameter has a pre-configured value. `getPreConfigurationValue()` returns this value.

- **States and reasons**
  Example: The method `isDeletionAllowedAccordingToCurrentConfigurationPhase()` returns true if the parameter can be deleted in the current configuration phase (post-build loadable projects only). `getNotDeletionAllowedAccordingToCurrentConfigurationPhaseReasons()` returns the reasons if deletion is not allowed.

7.4.6.3 IContainerStatePublished

The `IContainerStatePublished` specifies a type-safe published API for container states. It mainly covers the following state information

- Does this container have a pre-configuration container (includes access to this container)? The same information is being provided for recommended and initial (derived) values
- Is change or deletion allowed in the current configuration phase (post-build loadable use case)?
- In which configuration phase has this container been created in (post-build loadable use case)?
- What is the configuration class of this container

The figure 7.16 on the following page shows the inheritance hierarchy of the `IContainerStatePublished` class and its sub classes.

This API provides state information similar to IParameterStatePublished. Some of the states are container-specific, of course. `getCreationPhase()`, for example, which returns the phase a container in a post-build loadable configuration has been created in.

7.5 Model Services

7.5.1 EcucDefinitionAccess

The `IEcucDefinitionAccess` provides convenient and typesafe access to definition objects (module, container, parameter and reference definitions). The contained `def()` methods take MDF

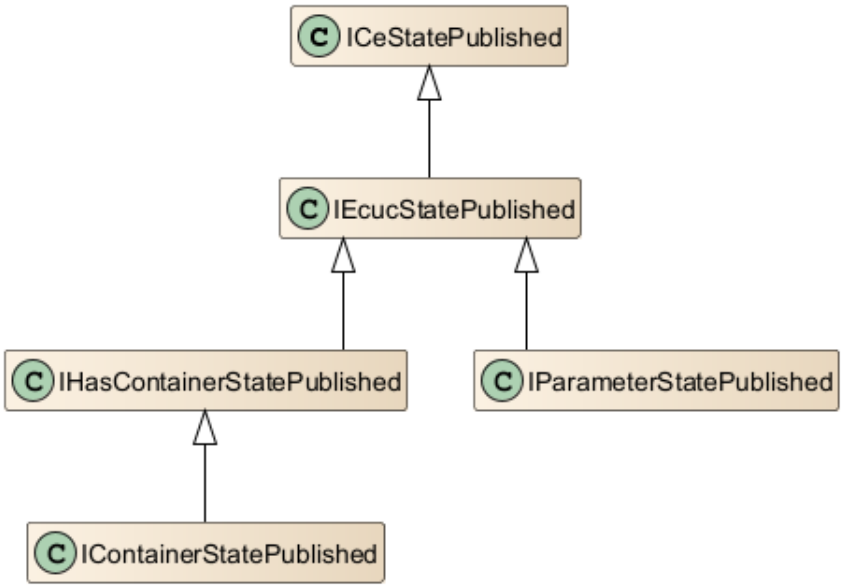


Figure 7.16: IContainerStatePublished class structure

definition objects and return wrappers which can be used to retrieve specific characteristics of definitions.

Example:

```
IEcucDefinitionAccess eda;
MIIntegerParamDef intParamDef;

// Get the integer definition wrapper
IEcucIntegerDefinition def = eda.def(intParamDef);

// Get the (optional) default value
Optional<BigInteger> defaultOpt = def.getDefault();
boolean hasDefault = defaultOpt.isPresent();
   BigInteger defaultValue = defaultOpt.get();

// Get the multiplicity
IEcucDefMultiplicity multiplicity = def.getMultiplicity();
BigInteger lower = multiplicity.getLower();
BigInteger upper = multiplicity.getUpper();
```

Listing 7.36: Integer parameter definition access examples

7.5.1.1 Post-build loadable

EcucModuleDefinition `IEcucModuleDefinition` is the interface of the module definition wrapper. It provides the following method(s):

getSupportedConfigurationVariants()
The `getSupportedConfigurationVariants()` method returns a collection of supported configuration variants. Never returns `null` but an empty collection if no supported config variants are specified.

The returned collection never contains the following literals:

- `EEcucConfigurationVariant.PRECONFIGURED_CONFIGURATION`

- EEcucConfigurationVariant.RECOMMENDED_CONFIGURATION

This method is for post-build loadable only!

Remark about AUTOSAR versions: Prior to AUTOSAR 4.2.1 the module definitions used the following valid values:

- VARIANT-PRE-COMPILE
- VARIANT-LINK-TIME
- VARIANT-POST-BUILD-LOADABLE
- VARIANT-POST-BUILD-SELECTABLE

VARIANT-POST-BUILD was invalid! With AUTOSAR 4.2.1 and later, the following values are valid (because the loadable and selectable specifications have been separated):

- VARIANT-PRE-COMPILE
- VARIANT-LINK-TIME
- VARIANT-POST-BUILD

VARIANT-POST-BUILD-LOADABLE and VARIANT-POST-BUILD-SELECTABLE are invalid!

This method takes the AUTOSAR version into account and returns the post-build loadable relevant specification only.

EcucContainerDefinition IEcucContainerDefinition is the interface of the container definition wrapper. It provides the following method(s):

getMultiplicityConfigurationClass()
The getMultiplicityConfigurationClass(EEcucConfigurationVariant) method returns the multiplicity configuration class for the specified module implementation variant. The returned value defines in which configuration phase the number of container instances latest may change if the module implements the specified variant.

Supported values for the variant are

- EEcucConfigurationVariant.VARIANT_PRE_COMPILE
- EEcucConfigurationVariant.VARIANT_LINK_TIME
- EEcucConfigurationVariant.VARIANT_POST_BUILD_LOADABLE

Other values lead to an IllegalArgumentException.

This method doesn't take the multiplicity into account. It only investigates the multiplicity configuration class as specified in the related container definition. So it still may return EEcucConfigurationClass.POST_BUILD even if the multiplicity is 1:1 for example. The post-build loadable use case differs here from post-build selectable (see supportsVariantMultiplicity()) because the changeability in the post-build phase is being inherited from parent objects. So, if you want to find out if a container actually permits changes in the post-build phase, you should use IContainerStatePublished.

This method is for post-build loadable only!

Remark about AUTOSAR versions: Prior to AUTOSAR 4.2.1 the container definitions contained the postBuildChangeable flag to define post-build loadable support. This method inter-

nally investigates the `postBuildChangeable` flag in this case but the `multiplicityConfigClass` table for AUROSAR 4.2.1 and newer versions.

EcucCommonAttributes `IEcucCommonAttributes` is the base interface of all parameter and reference definition wrappers. It provides the following method(s):

getMultiplicityConfigurationClass()
The `getMultiplicityConfigurationClass(EEcucConfigurationVariant)` method returns the multiplicity configuration class for the specified module implementation variant. The returned value defines in which configuration phase the number of parameter instances latest may change if the module implements the specified variant.

Supported values for the variant are

- `EEcucConfigurationVariant.VARIANT_PRE_COMPILE`
- `EEcucConfigurationVariant.VARIANT_LINK_TIME`
- `EEcucConfigurationVariant.VARIANT_POST_BUILD_LOADABLE`

Other values lead to an `IllegalArgumentException`.

This method doesn't take the multiplicity into account. It only investigates the multiplicity configuration class as specified in the related parameter definition. So it still may return `EEcucConfigurationClass.POST_BUILD` even if the multiplicity is 1:1 for example. The post-build loadable use case differs here from post-build selectable (see `supportsVariantMultiplicity()`) because the changeability in the post-build phase is being inherited from parent objects. So, if you want to find out if a parameter actually permits changes in the post-build phase, you should use `IParameterStatePublished`.

This method is for post-build loadable only!

Remark about AUTOSAR versions: Prior to AUTOSAR 4.2.1 the parameter definitions contain the `implementationConfigClass` table to define post-build loadable support. This method internally investigates the `implementationConfigClass` in this case but the `multiplicityConfigClass` table for AUROSAR 4.2.1 and newer versions.

getValueConfigurationClass()
The `getValueConfigurationClass(EEcucConfigurationVariant)` method returns the value configuration class for the specified module implementation variant. The returned value defines in which configuration phase the value of parameter instances latest may change if the module implements the specified variant.

Supported values for the variant are

- `EEcucConfigurationVariant.VARIANT_PRE_COMPILE`
- `EEcucConfigurationVariant.VARIANT_LINK_TIME`
- `EEcucConfigurationVariant.VARIANT_POST_BUILD_LOADABLE`

Other values lead to an `IllegalArgumentException`.

This method never returns `EEcucConfigurationClass.LINK`.

This method is for post-build loadable only!

Remark about AUTOSAR versions: Prior to AUTOSAR 4.2.1 the parameter definitions contain the `implementationConfigClass` table to define post-build loadable support. This method

internally investigates the `implementationConfigClass` in this case but the `valueConfigClass` table for AUROSAR 4.2.1 and newer versions.

7.5.1.2 Post-build selectable

EcucModuleDefinition `IEcucModuleDefinition` is the interface of the module definition wrapper. It provides the following method(s):

supportsPostBuildVariance()
The `supportsPostBuildVariance()` method returns `true` if this module configuration supports post-build selectable.

Remark about AUTOSAR versions: Prior to AUTOSAR 4.2.1 the module definitions `supportedSupportedConfigurationVariants` defined both, post-build loadable and selectable support. With AUTOSAR 4.2.1 the `supportedSupportedConfigurationVariants` specifies post-build loadable only and this method returns the value of the new `postBuildVariantSupport` flag.

EcucCommonAttributes `IEcucContainerDefinition` is the interface of the container definition wrapper. It provides the following method(s):

supportsVariantMultiplicity()
The `supportsVariantMultiplicity()` method returns `true` if this container type supports variant multiplicity. If `true` is returned this means that different variants may contain different number of instances of this container type.

This method takes the multiplicity into account. So, if the container definition specifies the multiplicity with lower == upper, it always returns `false`. Concerning post-build selectable it never makes sense to permit variance if lower and upper multiplicity are equal.

This method is for post-build selectable only!

Remark about AUTOSAR versions: Prior to AUTOSAR 4.2.1 the container definitions contained the `postBuildChangeable` flag to define post-build loadable support. This method internally investigates the `postBuildChangeable` flag in this case but the `postBuildVariantMultiplicity` flag for AUROSAR 4.2.1 and newer versions.

supportsVariantShortname()
The `supportsVariantShortname()` method returns `true` if one of the following conditions apply.

- `supportsVariantMultiplicity()` returns `true`
- The ADMIN-DATA flag postBuildSelectableChangeable is `true`

The use case for this specification are 1:1 containers. When this method returns `true`, 1:1 containers may have different shortnames in different variants. This is a Vector specific semantic which is not provided by AUTOSAR.

EcucCommonAttributes `IEcucCommonAttributes` is the base interface of all parameter and reference definition wrappers. It provides the following method(s):

supportsVariantMultiplicity()
The `supportsVariantMultiplicity()` method returns `true` if this parameter type supports variant multiplicity. If `true` is returned this means that different variants may contain different number of instances of this parameter type.

This method takes the multiplicity into account. So, if the parameter definition specifies the multiplicity with lower == upper, it always returns `false`. Concerning post-build selectable it never makes sense to permit variance if lower and upper multiplicity are equal.

This method is for post-build selectable only!

Remark about AUTOSAR versions: Prior to AUTOSAR 4.2.1 the parameter definitions contain the `implementationConfigClass` table to define post-build selectable support. This method internally investigates the `implementationConfigClass` in this case but the `postBuildVariantMultiplicity` flag for AUROSAR 4.2.1 and newer versions.

supportsVariantValue()
The `supportsVariantValue()` method returns `true` if this parameter type supports a variant value. If `true` is returned this means that different variants may contain different values in instances of this parameter type.

This method is for post-build selectable only!

Remark about AUTOSAR versions: Prior to AUTOSAR 4.2.1 the parameter definitions contain the `implementationConfigClass` table to define post-build selectable support. This method internally investigates the `implementationConfigClass` in this case but the `postBuildVariantValue` flag for AUROSAR 4.2.1 and newer versions.

7.5.2 EcuConfigurationAccess

The `IEcuConfigurationAccess` provides convenient and typesafe access to configuration objects (modules, containers, parameters and references). The contained `cfg()` methods take MDF (ECU configuration) objects and return wrappers which can be used to retrieve specific characteristics of the configuration content.

Example:

```
IEcuConfigurationAccess eca;
MINumericalValue intParam;

// Get the parameter wrapper
IEcucNumericalParameter numCfg = eca.cfg(intParam);

// Check if this is an integer parameter
if (numCfg instanceof IEcucIntegerParameter) {
    IEcucIntegerParameter intCfg = (IEcucIntegerParameter) numCfg;

    // Get the parameter value
    boolean hasValue = intCfg.hasValue();
    BigInteger value = intCfg.getValue();

    // Get the related definition wrapper
    IEcucIntegerDefinition def = intCfg.getEcucDefinition();
}
```

Listing 7.37: Integer parameter configuration access examples

7.5.2.1 Post-build loadable

EcucModuleConfiguration `IEcucModuleConfiguration` is the base interface of all module configuration wrappers. It provides the following method(s):

getConfigurationVariant()
The `getConfigurationVariant()` method returns the modules configuration variant.

This method never returns `null`. If the module has no value specified, this method returns a default value as follows:

- `EEcucConfigurationVariant.VARIANT_PRE_COMPILE`, if it is contained in the supported config variants of the related module definition
- otherwise `EEcucConfigurationVariant.VARIANT_LINK_TIME`, if it is contained in the supported config variants of the related module definition
- otherwise `EEcucConfigurationVariant.VARIANT_POST_BUILD_LOADABLE`, if it is contained in the supported config variants of the related module definition
- otherwise `EEcucConfigurationVariant.VARIANT_PRE_COMPILE`, even if not contained in the supported config variants of the related module definition or if the definition is not available

Remark about AUTOSAR versions: Prior to AUTOSAR 4.2.1 the module configurations implementation config variant defined if this module implements post-build loadable and/or selectable. With AUTOSAR 4.2.1 the implementation config variant defines only if the module implements post-build loadable. The post-build selectable aspect has been separated from this definition. This method handles the loadable semantic, independent of the AUTOSAR version.

This is for post-build loadable only!

setConfigurationVariant()
The `setConfigurationVariant(EEcucConfigurationVariant)` method sets the specified implementation configuration variant.

This is for post-build loadable only!

Supported values are

- `EEcucConfigurationVariant.VARIANT_PRE_COMPILE`
- `EEcucConfigurationVariant.VARIANT_LINK_TIME`
- `EEcucConfigurationVariant.VARIANT_POST_BUILD_LOADABLE`

Remarks concerning AUTOSAR versions:

- If the modules definition has schema version 4.2.1 or higher, the specified value is being written directly to the model
- If the modules definition has a schema version lower than 4.2.1, the modules implementation configuration variant in the MDF model encodes both, post-build loadable and post-build selectable. The following behavior is being implemented in this case:

Current model value	Parameter	Result in the model
PRE_COMPILE	PRE_COMPILE	PRE_COMPILE
	LINK_TIME	LINK_TIME
	POST_BUILD_LOADABLE	POST_BUILD_LOADABLE
LINK_TIME	PRE_COMPILE	PRE_COMPILE
	LINK_TIME	LINK_TIME
	POST_BUILD_LOADABLE	POST_BUILD_LOADABLE
POST_BUILD_LOADABLE	PRE_COMPILE	PRE_COMPILE
	LINK_TIME	LINK_TIME
	POST_BUILD_LOADABLE	POST_BUILD_LOADABLE
POST_BUILD_SELECTABLE	PRE_COMPILE	POST_BUILD_SELECTABLE
	LINK_TIME	POST_BUILD_SELECTABLE
	POST_BUILD_LOADABLE	POST_BUILD
POST_BUILD	PRE_COMPILE	POST_BUILD_SELECTABLE
	LINK_TIME	POST_BUILD_SELECTABLE
	POST_BUILD_LOADABLE	POST_BUILD

EcucContainer `IEcucContainer` is the base interface of all container wrappers. It provides the following method(s):

getEffectiveMultiplicityConfigurationClass()
The `getEffectiveMultiplicityConfigurationClass()` method walks up the model tree to find the related module configuration. Then it uses the module implementation configuration variant to return the selected configuration class as specified in the container definition.

This method never returns `null`. In case the detection of the configuration class fails (e.g. if the related module configuration cannot be detected), this method returns `EEcucConfigurationClass.PRE_COMPILE` by default. It also never returns `EEcucConfigurationClass.LINK`.

This method is for post-build loadable only!

getEffectiveMultiplicityConfigurationClassDefRef()
The `getEffectiveMultiplicityConfigurationClass(DefRef)` method walks up the model tree to find the related module configuration. Then it uses the module implementation configuration variant to return the selected configuration class of the specified parameter definition.

This method never returns `null`. In case the detection of the configuration class fails (e.g. if the related module configuration cannot be detected), this method returns `EEcucConfigurationClass.PRE_COMPILE` by default. It also never returns `EEcucConfigurationClass.LINK`.

This method is for post-build loadable only!

getEffectiveValueConfigurationClass()
The `getEffectiveValueConfigurationClass(DefRef)` method walks up the model tree to find the related module configuration. Then it uses the module implementation configuration variant to return the selected configuration class of the specified parameter definition.

This method never returns `null`. In case the detection of the configuration class fails (e.g. if the related module configuration cannot be detected), this method returns `EEcucConfigurationClass.PRE_COMPILE` by default. It also never returns `EEcucConfigurationClass.LINK`.

This method is for post-build loadable only!

EcucParameter `IEcucParameter` is the base interface of all parameter and reference wrappers. It provides the following method(s):

getEffectiveMultiplicityConfigurationClass()
The `getEffectiveMultiplicityConfigurationClass()` method walks up the model tree to find the related module configuration. Then it uses the module implementation configuration variant to return the selected configuration class as specified in the parameter definition.

This method never returns `null`. In case the detection of the configuration class fails (e.g. if the related module configuration cannot be detected), this method returns `EEcucConfigurationClass.PRE_COMPILE` by default.

This is for post-build loadable only!

getEffectiveValueConfigurationClass()
The `getEffectiveValueConfigurationClass()` method walks up the model tree to find the related module configuration. Then it uses the module implementation configuration variant to return the selected configuration class as specified in the parameter definition.

This method never returns `null`. In case the detection of the configuration class fails (e.g. if the related module configuration cannot be detected), this method returns `EEcucConfigurationClass.PRE_COMPILE` by default.

This is for post-build loadable only!

7.5.2.2 Post-build selectable

EcucModuleConfiguration `IEcucModuleConfiguration` is the base interface of all module configuration wrappers. It provides the following method(s):

supportsPostBuildVariance()
The `supportsPostBuildVariance()` method returns `true` if this module configuration supports post-build selectable.

This is for post-build selectable only!

What this method actually does:

- It checks if the related definition specifies post-build selectable as supported
- It checks if the module configuration implements post-build variance. That's `true` in the following cases
  - If the modules definition has schema version 4.4 or higher: Check if the `PostBuildVariantUsed` is `true`
  - If the modules definition has schema version 4.2.1 or higher: Check if the modules ADMIN-DATA flag "postBuildVariantSupport" is `true` (false is default if this flag is missing)
  - If the modules definition has a schema version lower than 4.2.1: Check if the modules implementation configuration variant contains one of the following values VARIANT_POST_BUILD_SELECTABLE or VARIANT_POST_BUILD

It returns `true` if both conditions are `true`.

setPostBuildVarianceSupport()
The `setPostBuildVarianceSupport(boolean)` method sets the post-build support flag in the module configuration.

This is for post-build selectable only!

Remarks concerning AUTOSAR versions:

- If the modules definition has schema version 4.4.0 or higher, this method sets the `PostBuildVariantUsed` to the specified value.

- If the modules definition has schema version 4.2.1 or higher but less than 4.4.0, this method sets the modules ADMIN-DATA flag "postBuildVariantSupport" to the specified value.

- If the modules definition has a schema version lower than 4.2.1, the modules implementation configuration variant in the MDF model encodes both, post-build loadable and post-build selectable. The following behavior is being implemented in this case:

Current model value	Parameter	Result in the model
PRE_COMPILE	true	POST_BUILD_SELECTABLE
	false	PRE_COMPILE
LINK_TIME	true	POST_BUILD_SELECTABLE
	false	LINK_TIME
POST_BUILD_LOADABLE	true	POST_BUILD
	false	POST_BUILD_LOADABLE
POST_BUILD_SELECTABLE	true	POST_BUILD_SELECTABLE
	false	PRE_COMPILE
POST_BUILD	true	POST_BUILD
	false	POST_BUILD_LOADABLE

EcucContainer `IEcucContainer` is the base interface of all container wrappers. It provides the following method(s):

supportsVariantMultiplicity()
The `supportsVariantMultiplicity()` method returns `true` if the related module configuration supports variance and this containers definition support variant multiplicity. If `true` is returned this means that different variants may contain different number of instances of this container.

If the container has no definition, this method returns `false`.

This method is for post-build selectable only!

EcucParameter `IEcucParameter` is the base interface of all parameter and reference wrappers. It provides the following method(s):

supportsVariantMultiplicity()
The `supportsVariantMultiplicity()` method returns `true` if the related module configuration supports variance and this parameters definition support variant multiplicity. If `true` is returned this means that different variants may contain different number of instances of this parameter.

If the parameter has no definition, this method returns `false`.

This is for post-build selectable only!

supportsVariantValue()

The `supportsVariantValue()` method returns `true` if the related module configuration supports variance and this parameters definition support variant values. If `true` is returned this means that different variants may contain different values in instances of this parameter.

If the parameter has no definition, this method returns `false`.

This is for post-build selectable only!

8 AutomationInterface Content

8.1 Introduction

This chapter describes the content of the DaVinci Configurator AutomationInterface.

8.2 Folder Structure

The AutomationInterface consists of the following files and folders:

- **dvcfg**
  - **dvcfgpai**
    - **__doc** (find more details to its content in chapter 8.3)
      - **DVCfg__AutomationInterfaceDocumentation.pdf:** this document
      - **javadoc:** Javadoc HTML pages
      - **templates:** script file and script project templates for a simple start of script development
    - **libs:** compile bindings to Groovy and to the DaVinci Configurator AutomationInterface, used by IntelliJ IDEA and Gradle

8.3 Script Development Help

The help for the AutomationInterface script development is distributed among the following sources:

- DVCfg__AutomationInterfaceDocumentation.pdf (this document)
- Javadoc HTML Pages
- Script Templates

8.3.1 AutomationInterfaceDocumentation PDF

You find this document as described in chapter 8.2. It provides a good overview of architecture, available APIs and gives an introduction of how to get started in script development. The focus of the document is to provide an overview and not to be complete in API description. To get a complete and detailed description of APIs and methods use the Javadoc HTML Pages as described in 8.3.2.

8.3.2 Javadoc HTML Pages

You find this documentation as described in chapter 8.2. Open the file `index.html` to access the complete DaVinci Configurator AutomationInterface API reference. It contains descriptions of all classes and methods that are part of the AutomationInterface.

The Javadoc is also accessible at your source code in the IDE for script development.

8.3.3 Script Templates

You find the Script Templates as described in chapter 8.2 on the previous page. You may copy them for a quick startup in script development.

8.4 Libs and BuildLibs

The AutomationInterface contains libraries to build projects, see **buildLibs** in 8.2 on the preceding page . And it contains other libraries which are described in **libs** in 8.2 on the previous page.

8.5 Beta API Usage

The beta annotation is exempt from any compatibility guarantees made by its containing library. Note that the presence of this annotation implies nothing about the quality or performance of the API in question, only the fact that it is not "API-frozen".

Note that the client which uses this API must upgrade to each product release, to guarantee, that the used API is still available.

Beta API is annotated with:

```
@PublishedBeta
```

Listing 8.1: Beta API Annotation

To use Beta APIs in script projects, see the following example:

```
scriptProject {
    allowBetaApiUsage = true
}
```

Listing 8.2: allowBetaApiUsage flag enables beta API usage

8.6 Introduction

An automation script project is a normal Java/Groovy development project, where the built artifact is a single `.jar` file. The jar file is created by the build system, see chapter 8.14 on page 384.

It is the recommended way to develop scripts, containing more tasks or multiple classes.

The project provides IDE support for:

- Code completion
- Syntax highlighting
- API Documentation
- Debug support
- Build support

The recommended IDE is IntelliJ IDEA.

8.7 Automation Script Project Creation

To create a new script project please follow the instructions in chapter 3.6 on page 13.

8.8 Project File Content

An automation project will at least contain the following files and folders:

- Folders
  - `.gradle` - Gradle temp folder - **DO NOT** commit it into a version control system
  - `build` - Gradle build folder - **DO NOT** commit it into a version control system
  - `gradle` - Gradle bootstrap folder - Please commit it into your version control system
  - `src` - Source folder containing your Groovy, Java, Kotlin sources and resource files
  - `testCfg` - Test folder containing tests written with the [@sec:AutomationTestingFramework:Automatic Testing Framework] for your script project.
- Files
  - Gradle files - see 8.14.2 on page 384 for details
    - `gradlew.bat`
    - `build.gradle`
    - `settings.gradle`
  - IntelliJ Project files (optional) - **DO NOT** commit it into a version control system
    - `ProjectName.iws`
    - `ProjectName.iml`
    - `ProjectName.ipr`

The IntelliJ Project files (`*.iws, *.iml, *.ipr`) can be recreated with the command in the windows command shell (`cmd.exe`): `gradlew idea`

8.9 Deployment of the Jar File

To deploy your automation script project you only need to deploy the built jar file located in `<ProjectDir>/build/libs/<ProjectName>-<Version>.jar`. All other files in your automation script project are **not required** for the script **execution**.

So if you want to use your script project in an DaVinci Configurator project, copy the jar file into the DaVinci Configurator project and add the folder containing the jar file in the Script Locations view with the Project scope.

8.10 IntelliJ IDEA Usage

8.10.1 Show API Specifications (JavaDoc)

In newer IntelliJ versions the automatic download of the source files and their respective javadocs has been disabled. In order to benefit from the API-Specifications during coding it is necessary to

download the source files. This has to be done manually.

If source files are not yet downloaded, it looks like 8.1.

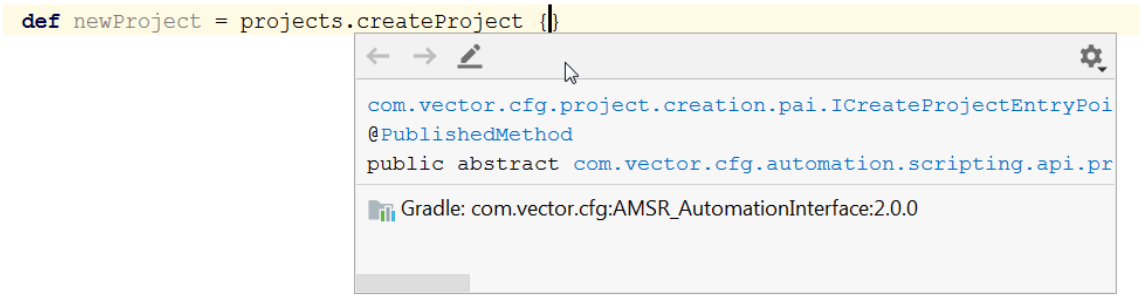


Figure 8.1: No JavaDoc

To download source files enter with F3 the API and click on "Download Sources" 8.11 on page 380.

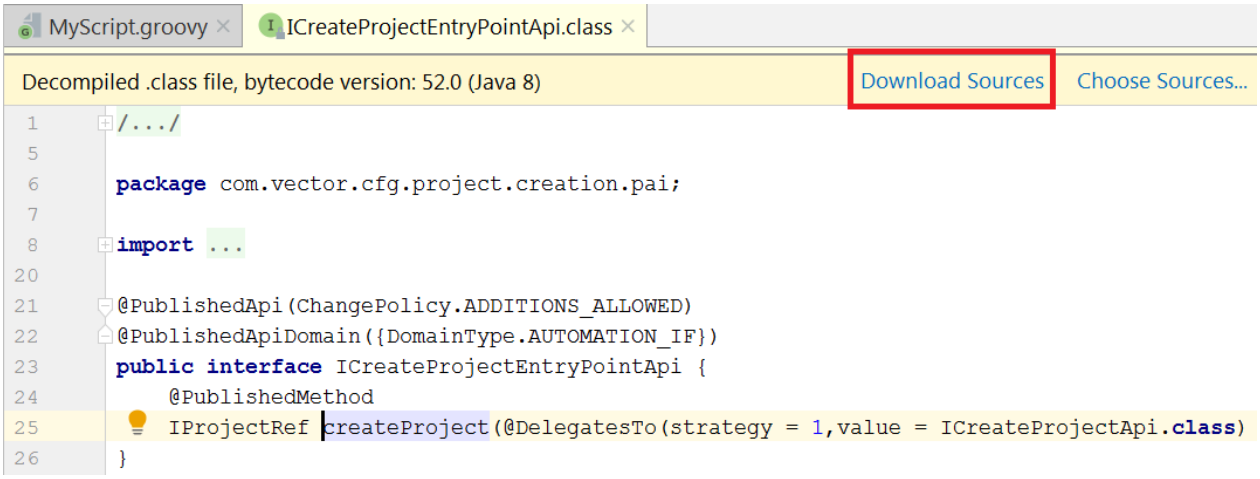


Figure 8.2: No JavaDoc

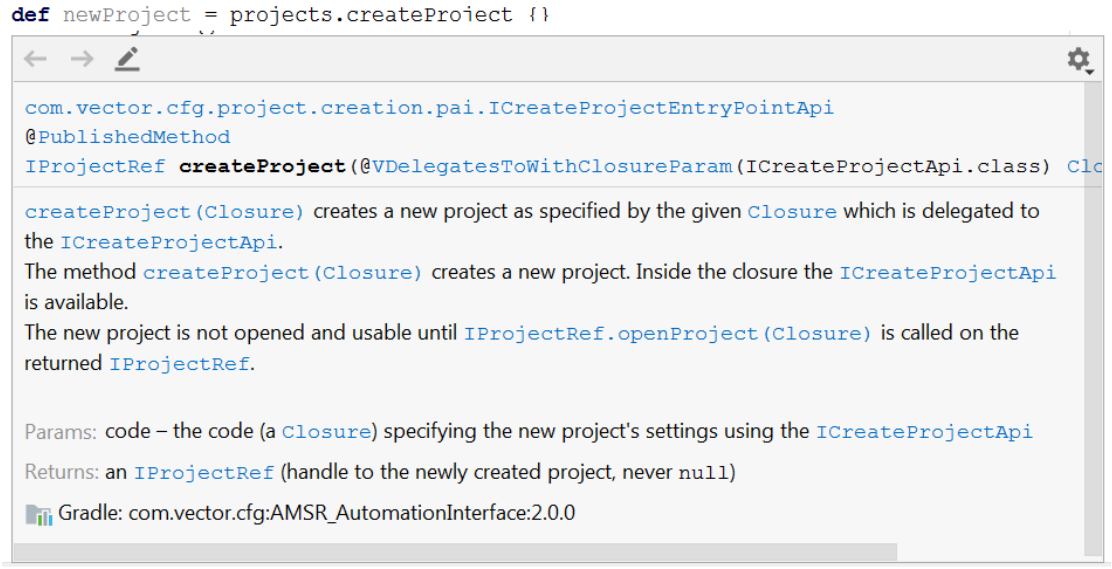


Figure 8.3: JavaDoc

8.10.2 Building Projects

Project Build The standard way to build projects is to choose the option **<ProjectName> [build]** in the Run Menu in the toolbar and to press the Run Button beneath that menu.

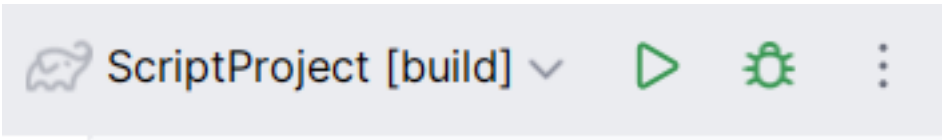


Figure 8.4: Project Build

8.10.3 Debugging with IntelliJ

1. Access Debug Options
   - Navigate to the script project's remote debug options
   - Copy the remote debug options
2. Set Up Shell Environment
   - Open a shell terminal and navigate to the core directory of the DaVinci Configurator installation
   - Set the `DVCFG_JVM_ARGS` environment variable in the shell with your copied remote debug options. Note: Modify the suspend flag to **suspend=y** to enable the wait-for-debugger mode.
   - Verify the correct shell configuration: `echo $env:DVCFG_JVM_ARGS`
3. Start Debugging the Script Task
   - Use the opened shell terminal to start the script task as usual.
   - Set the desired breakpoints in the script code

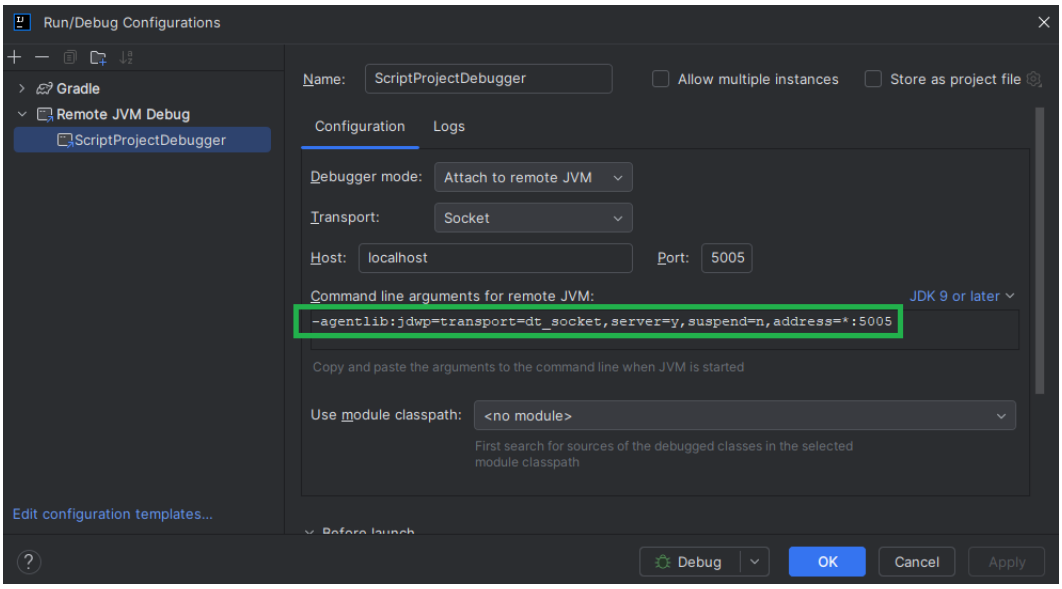


Figure 8.5: Remote Debug Options

Figure 8.6: Shell Debug Variable

- After the script task is started, the DaVinci Configurator will wait for a debugger to connect.

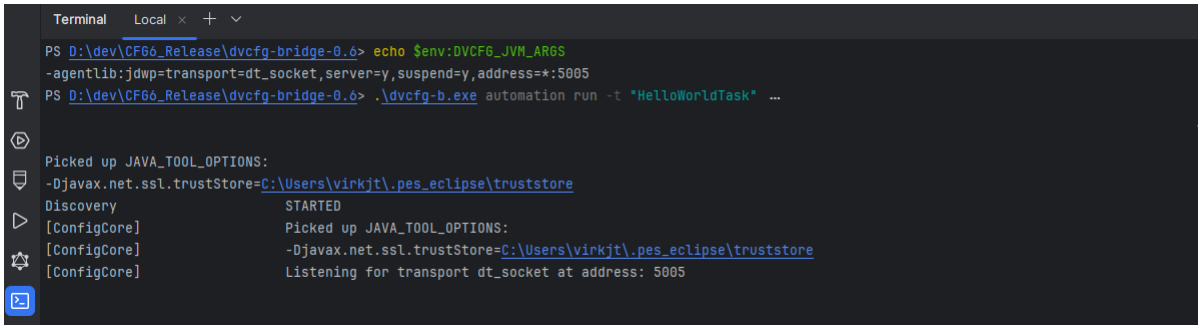


Figure 8.7: Run Script Task

- Now attach your script to the DaVinci Configurators script task execution process, by activating the remote debugging in the script project.

8.10.4 Troubleshooting

Code completion, Compilation If the code completion or compilation does not work, please verify that the Java JDK settings in the IntelliJ IDEA are correct. You have to set the Project JDK and the Gradle JDK setting. See 3.6.2 on page 14.

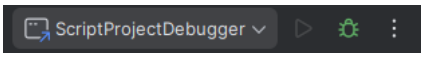


Figure 8.8: Attach for Debugger

Gradle build, build button If the Gradle build does nothing after start or the build button is grayed, please verify that the Java JDK settings in the IntelliJ IDEA are correct. You have to set the Gradle JDK setting. See 3.6.2 on page 14.

If the build button is marked with an error, please make sure that the Gradle plugin inside of IntelliJ IDEA is installed. Open File->Settings...->Plugins and select the Gradle plugin.

IntelliJ Build You shall not use the IntelliJ menu "Build" or the context menu entries "Make Project", "Make Module", "Rebuild Project" or "Compile". The project shall be build with Gradle not with IntelliJ IDEA. So you have to select one of the Run Configuration (Run menu) to build the project as described in chapter 8.10 on page 371.

Groovy SDK not configured If you get the message 'Groovy SDK is not configured for ...' in IntelliJ IDEA you probably have to migrate your project as described in chapter 8.12 on the following page.

No JavaDoc Shown If you don't see a javadoc description for the APIs. See chapter 8.10.1 on page 371.

Compile errors - Could not find com.vector.cfg:DVCfgAutomationInterface If you get compile errors inside of the IntelliJ IDEA, after updating the DaVinci Configurator or moving projects.

Please execute the **Project Migration to newer DaVinci Configurator Version** step, see 8.12 on the following page.

Download of Gradle Distribution Error If you get an error when you start the `gradlew` like:

```
Downloading
http://vistrcfgci1.vi.vector.int/buildcomponents/Gradle/distributions/gradle-9.3.1-bin.zip

Exception in thread "main" java.io.FileNotFoundException:
http://vistrcfgci1.vi.vector.int/buildcomponents/Gradle/distributions/gradle-9.3.1-bin.zip
at sun.net.www.protocol.http.HttpURLConnection.getInputStream0(HttpURLConnection.java:1836)
```

The problem is you can't connect to the server, where the Gradle installation is located[1] . To change the location, you have to open the file `<YourProject>/gradle/wrapper/gradle-wrapper.properties` and change the line `distributionUrl=`.

You have multiple options for the content of the `distributionUrl`:

- Change the URL to the Gradle default (needs internet access):
  - `https://services.gradle.org/distributions/gradle-9.3.1-bin.zip`

[1] The vector internal server `http://vistrcfgci1.vi.vector.int` is not accessible from outside of the vector network and shall only be used by internal projects. If you have a project with the internal server and your are not inside the network, please change it to another location.

- Change the URL to a Server location of your choice. E.g inside your company.
- Download Gradle manually and change the URL to a local file system location like:
  - `file:/D:/YourFolder/gradle-9.3.1-bin.zip`

Caution: You have to escape a `:` with `\:` so an HTTP address would start with `http\://` and the local filesystem would start with `file\:/`.

So the default line in the file 'gradle-wrapper.properties' for the default Gradle server would be:
`distributionUrl=https\://services.gradle.org/distributions/gradle-9.3.1-bin.zip`

8.11 Project Usage in different DaVinci Configurator Versions

You can execute the script tasks of a script project in different versions of the DaVinci Configurator as long as the following conditions are met:

- The script was compiled with the oldest DaVinci Configurator version in which it should be used

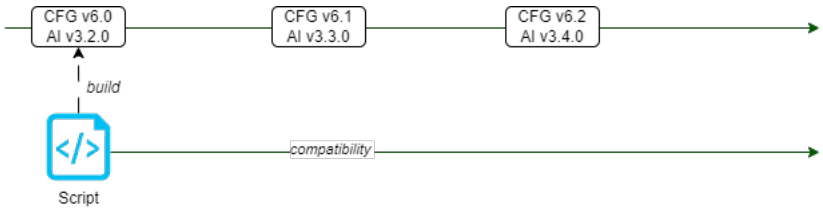


Figure 8.9: Shows the script compatibility

- The DaVinci Configurator version span must not contain a breaking change.
- If you use the BswmdModel, you have to use a compatible BSW package.
  - The used BSW module definitions (BSWMD files) must have compatible names and multiplicities.

8.12 Script Project Update to a newer Configurator/AutomationInterface version

If updating the script project to a newer DaVinci Configurator respectively a newer AutomationInterface version, it is necessary to rebuild the script project.

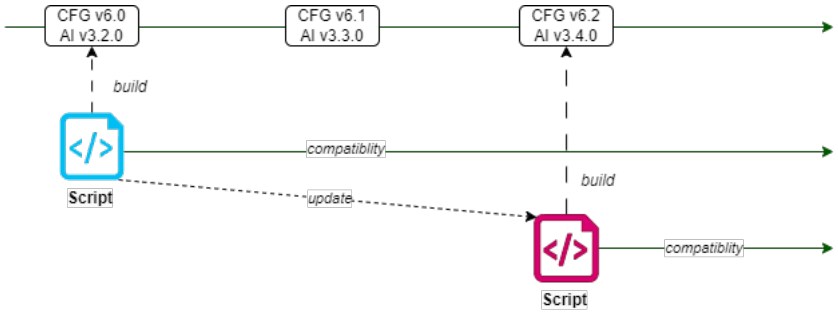


Figure 8.10: Shows the Script Update

Steps to execute:

1. Verify the correct reference in the `build.gradle` file to the DaVinci Configurator installation folder.

```
scriptProject {
    cfgPath = file("<PATH_TO_CFG6>")
}
```

Listing 8.3: cfgPath variable stores the path to the CFG installation

2. Rebuild the script project with the `gradlew build` command.

This will update the compile time dependencies of your Script Project according to the new DaVinci Configurator version.

8.13 Build System

The build system uses Gradle[2] to build a single Jar file. It also setups the dependencies to the DaVinci Configurator and create the IntelliJ IDEA project.

To set up the Gradle installation, see chapter 3.6.2 on page 15.

8.13.1 Jar Creation and Output Location

The call to `gradlew build` in the root directory of your automation script project will create the jar file. The *.jar output location is `<ProjectRoot>/build/libs/<ProjectName>.jar`.

8.13.2 Gradle File Structure

The default automation project contains the following Gradle build files:

- `gradlew.bat`
  - Gradle batch file to start Gradle (Gradle Wrapper[3])
- `build.gradle`
  - General build file - You can modify it to adapt the build to your needs
- `settings.gradle`
  - General build project settings - See Gradle documentation[4]

8.13.2.1 build.gradle

The file contains three essential parts of the build:

- `plugins{}` Defines the used Gradle plugins
- `dependencies{}` Defines the dependencies 8.14.2.2 on page 384 of the project
- `scriptProject{}` Defines the configuration of the Automation Build Gradle Plugin. (See Automation Build Gradle Plugin Documentation)

8.13.2.2 dependencies

We assume we have a jar from a Maven repository like `Apache Commons IO` (the identifier would be `'commons-io:commons-io:2.5'`, See MavenCentral).

```
dependencies {
    // Change the identifier to your library to use
    implementation 'commons-io:commons-io:2.5'
    // You could add multiple libraries with additional implementation lines
}
```

Listing 8.4: build.gradle - Add dependencies to a script project

- Optional: if you are behind a proxy or firewall:

[2] `https://gradle.org/` [2025-09-23]

[3] `https://docs.gradle.org/current/userguide/gradle_wrapper.html` [2025-09-23]

[4] `https://docs.gradle.org/current/dsl/org.gradle.api.initialization.Settings.html` [2025-09-23]

- You must either set proxy options for gradle [5]
- **Preferred way**: use a Maven repository inside your network: To set a repository, add before the dependencies block:

```
repositories {
    // URL to your repository
    // The URL below is the Vector internal network server
    // Please change the URL to your server
    maven { url 'https://vistrpesart1.vi.vector.int/artifactory/pes-davinciall-
        maven' }
    // Or reference MavenCentral server
    mavenCentral()
}
```

Usage of external Libraries (Jars) in the AutomationProject You could reference external libraries (Jar files) in your AutomationProject. But you have to configure the libraries in the Gradle build files. **DO NOT** add a dependency in IntelliJ, this will not work.

The easiest and preferred way is the use a library from any Maven repository like MavenCentral or JCenter. This will also handle versions, and transitive dependencies automatically.

8.13.2.3 Static Compilation of Groovy Code

The AutomationInterface contains a Groovy compiler extension. In some use cases performance has priority, and therefore it is possible to use Groovy with static compilation.

Groovy is a dynamic JVM language using dynamic dispatch for its method calls. Dynamic dispatch in Groovy is approximately three times slower compared to a normal Java method call. Groovy has added the static compilation feature via @CompileStatic annotation, which allows to compile most of Groovy method calls into direct JVM bytecode method calls, thus avoiding all the dynamic dispatch overhead. ([6])

Mark your classes or methods with:

```
@CompileStatic
def myMethod() {

}

@CompileStatic
class MyClass {

}
```

Listing 8.5: @CompileStatic with Automation API

The same applies, if you want to use the `@TypeChecked` annotation:

```
@TypeChecked
def myMethod() {

}
```

Listing 8.6: @TypeChecked with Automation API

[5] Gradle and Java online documentation for details how to set proxy settings

[6] `http://java-performance.info/static-code-compilation-groovy-2-0/` [2018-11-29]

8.13.2.4 Gradle Maven publishing of an AutomationProject

The Gradle scriptProject automatically adds a Gradle `MavenPublication` instance for all signed `Jar` files. The publication is named `signedJar`, so the `publish` and `publishToMavenLocal` will publish the signed `Jar` files.

To download source files enter with F3 the API and click on "Download Sources" 8.11.

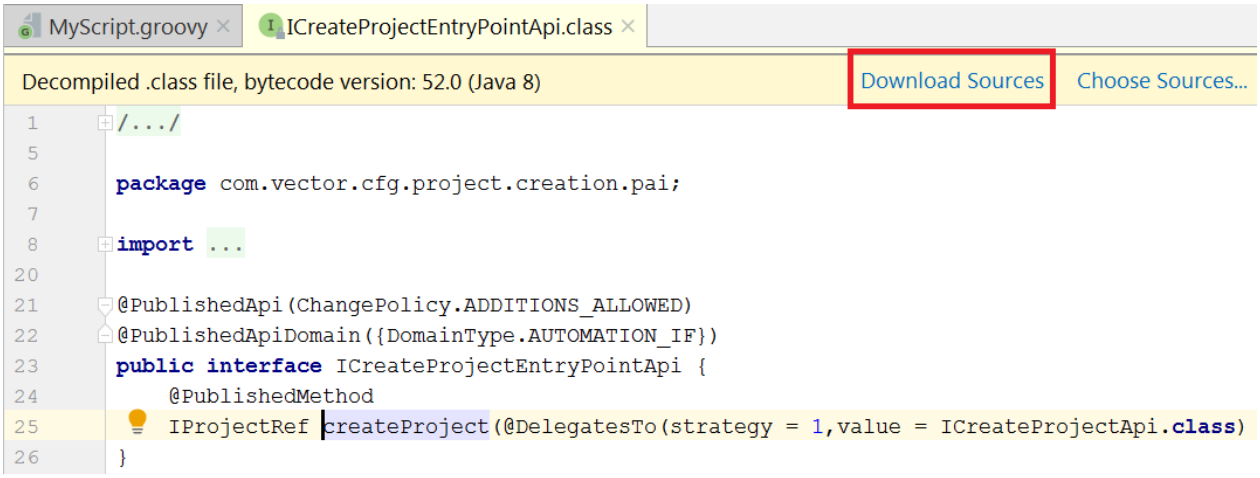


Figure 8.11: No JavaDoc

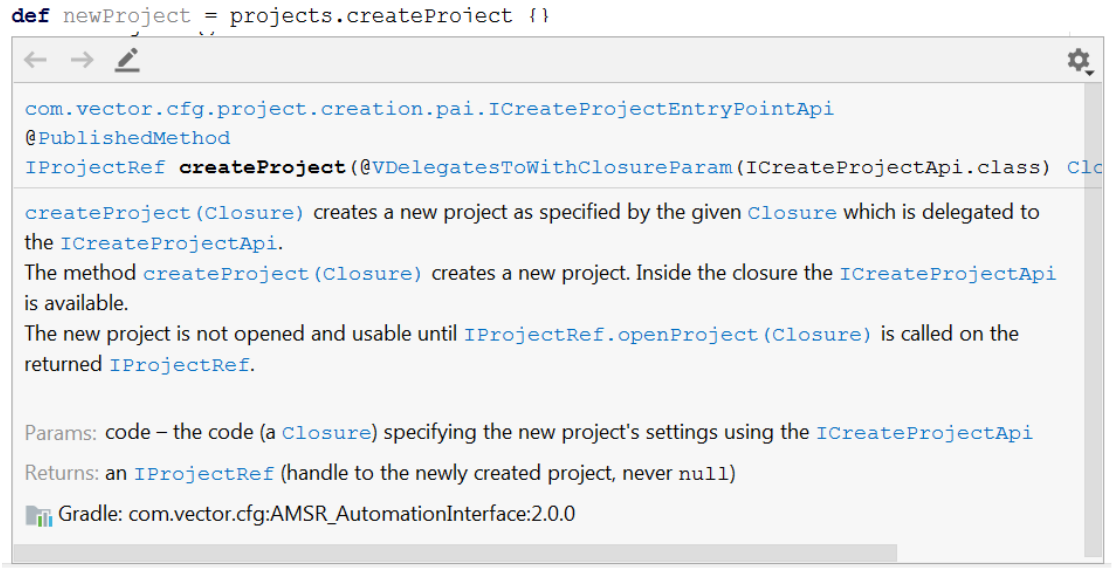


Figure 8.12: JavaDoc

8.13.2.5 Building Projects

Project Build The standard way to build projects is to choose the option **<ProjectName> [build]** in the Run Menu in the toolbar and to press the Run Button beneath that menu.

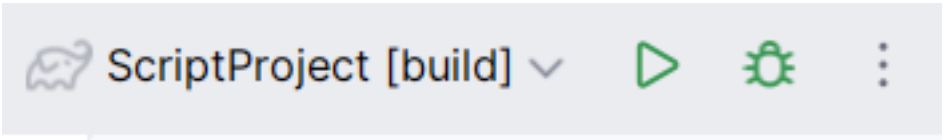


Figure 8.13: Project Build

8.13.2.6 Debugging with IntelliJ

1. Access Debug Options
   - Navigate to the script project's remote debug options
   - Copy the remote debug options

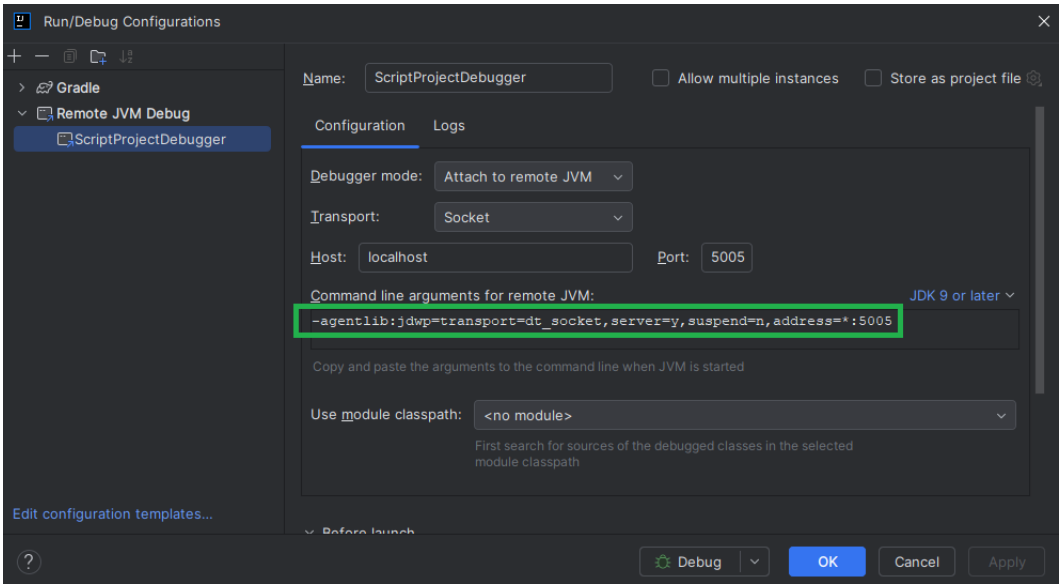


Figure 8.14: Remote Debug Options

2. Set Up Shell Environment
   - Open a shell terminal and navigate to the core directory of the DaVinci Configurator installation
   - Set the `DVCFG_JVM_ARGS` environment variable in the shell with your copied remote debug options. Note: Modify the suspend flag to **suspend=y** to enable the wait-for-debugger mode.
   - Verify the correct shell configuration: `echo $env:DVCFG_JVM_ARGS`

Figure 8.15: Shell Debug Variable

3. Start Debugging the Script Task

- Use the opened shell terminal to start the script task as usual.
- Set the desired breakpoints in the script code
- After the script task is started, the DaVinci Configurator will wait for a debugger to connect.

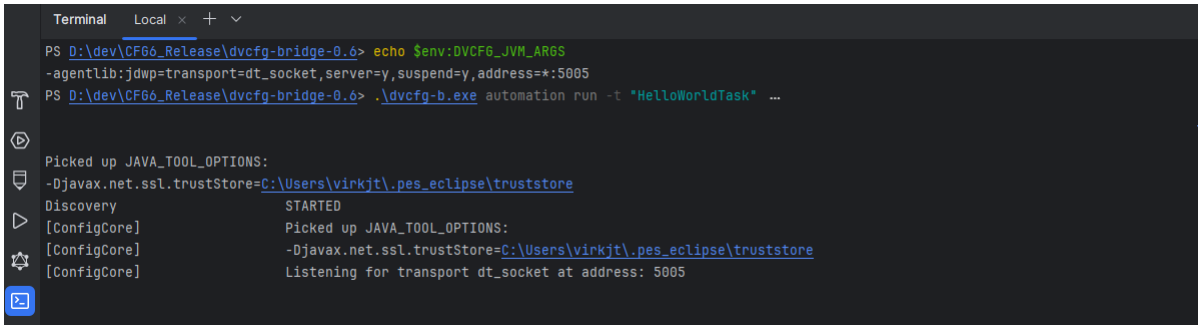


Figure 8.16: Run Script Task

- Now attach your script to the DaVinci Configurators script task execution process, by activating the remote debugging in the script project.

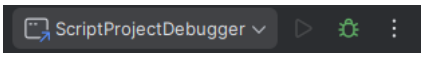


Figure 8.17: Attach for Debugger

8.13.2.7 Script Project Update to a newer Configurator/AutomationInterface version

If updating the script project to a newer DaVinci Configurator respectively a newer AutomationInterface version, it is necessary to rebuild the script project.

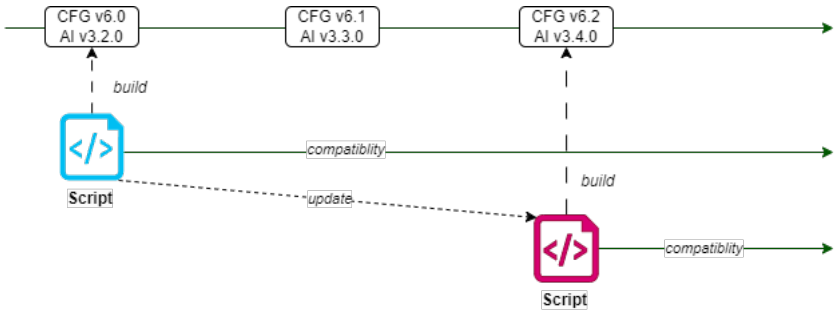


Figure 8.18: Shows the Script Update

Steps to execute:

1. Verify the correct reference in the `build.gradle` file to the DaVinci Configurator installation folder.

```
scriptProject {
    cfgPath = file("<PATH_TO_CFG6>")
}
```

Listing 8.7: cfgPath variable stores the path to the CFG installation

2. Rebuild the script project with the `gradlew build` command.

This will update the compile time dependencies of your Script Project according to the new DaVinci Configurator version.

8.14 Build System

The build system uses Gradle[7] to build a single Jar file. It also setups the dependencies to the DaVinci Configurator and create the IntelliJ IDEA project.

To set up the Gradle installation, see chapter 3.6.2 on page 15.

8.14.1 Jar Creation and Output Location

The call to `gradlew build` in the root directory of your automation script project will create the jar file. The *.jar output location is `<ProjectRoot>/build/libs/<ProjectName>.jar`.

8.14.2 Gradle File Structure

The default automation project contains the following Gradle build files:

- `gradlew.bat`
  - Gradle batch file to start Gradle (Gradle Wrapper[8])
- `build.gradle`
  - General build file - You can modify it to adapt the build to your needs
- `settings.gradle`
  - General build project settings - See Gradle documentation[9]

8.14.2.1 build.gradle

The file contains three essential parts of the build:

- `plugins{}` Defines the used Gradle plugins
- `dependencies{}` Defines the dependencies 8.14.2.2 of the project
- `scriptProject{}` Defines the configuration of the Automation Build Gradle Plugin. (See Automation Build Gradle Plugin Documentation)

8.14.2.2 dependencies

We assume we have a jar from a Maven repository like `Apache Commons IO` (the identifier would be `'commons-io:commons-io:2.5'`, See MavenCentral).

```
dependencies {
    // Change the identifier to your library to use
    implementation 'commons-io:commons-io:2.5'
    // You could add multiple libraries with additional implementation lines
}
```

Listing 8.8: build.gradle - Add dependencies to a script project

- Optional: if you are behind a proxy or firewall:

[7] `https://gradle.org/` [2025-09-23]

[8] `https://docs.gradle.org/current/userguide/gradle_wrapper.html` [2025-09-23]

[9] `https://docs.gradle.org/current/dsl/org.gradle.api.initialization.Settings.html` [2025-09-23]

- You must either set proxy options for gradle [10]
- **Preferred way**: use a Maven repository inside your network: To set a repository, add before the dependencies block:

```
repositories {
    // URL to your repository
    // The URL below is the Vector internal network server
    // Please change the URL to your server
    maven { url 'https://vistrpesart1.vi.vector.int/artifactory/pes-davinciall-
        maven' }
    // Or reference MavenCentral server
    mavenCentral()
}
```

Usage of external Libraries (Jars) in the AutomationProject You could reference external libraries (Jar files) in your AutomationProject. But you have to configure the libraries in the Gradle build files. **DO NOT** add a dependency in IntelliJ, this will not work.

The easiest and preferred way is the use a library from any Maven repository like MavenCentral or JCenter. This will also handle versions, and transitive dependencies automatically.

8.14.2.3 Static Compilation of Groovy Code

The AutomationInterface contains a Groovy compiler extension. In some use cases performance has priority, and therefore it is possible to use Groovy with static compilation.

Groovy is a dynamic JVM language using dynamic dispatch for its method calls. Dynamic dispatch in Groovy is approximately three times slower compared to a normal Java method call. Groovy has added the static compilation feature via @CompileStatic annotation, which allows to compile most of Groovy method calls into direct JVM bytecode method calls, thus avoiding all the dynamic dispatch overhead. ([11])

Mark your classes or methods with:

```
@CompileStatic
def myMethod() {

}

@CompileStatic
class MyClass {

}
```

Listing 8.9: @CompileStatic with Automation API

The same applies, if you want to use the `@TypeChecked` annotation:

```
@TypeChecked
def myMethod() {

}
```

Listing 8.10: @TypeChecked with Automation API

[10] Gradle and Java online documentation for details how to set proxy settings

[11] `http://java-performance.info/static-code-compilation-groovy-2-0/` [2018-11-29]

8.14.2.4 Gradle Maven publishing of an AutomationProject

The Gradle scriptProject automatically adds a Gradle `MavenPublication` instance for all signed `Jar` files. The publication is named `signedJar`, so the `publish` and `publishToMavenLocal` will publish the signed `Jar` files.

9 Kotlin Listings

This chapter contains Kotlin code listings that are equivalent to the Groovy listings presented in the previous chapters. Please note that not all Groovy listings have a corresponding Kotlin equivalent in this chapter, as some could not be automatically transpiled due to language-specific features or transpilation limitations.

```
scriptTask("taskName") {
  code {
    // ICommunicationApi is available as "communication" property
    var communication = domain.communication
  }
}
```

Accessing ICommunicationApi as a property - Kotlin

```
scriptTask("taskName") {
  code {
    domain.communication {
      // ICommunicationApi is available inside this Closure

    }
  }
}
```

Accessing ICommunicationApi in a scope-like way - Kotlin

```
scriptTask("taskName") {
  code {
    // IDiagnosticsApi is available as "diagnostics" property
    var diagnostics = domain.diagnostics
  }
}
```

Accessing IDiagnosticsApi as a property - Kotlin

```
scriptTask("taskName") {
  code {
    domain.diagnostics {
      // IDiagnosticsApi is available here

    }
  }
}
```

Accessing IDiagnosticsApi in a scope-like manner - Kotlin

```
scriptTask("taskName") {
  code {
    // IModeManagementApi is available as "modeManagement" property
    var modeManagement = domain.modeManagement
  }
}
```

Accessing IModeManagementApi as a property - Kotlin

```
scriptTask("taskName") {
  code {
    domain.modeManagement {
      // IModeManagementApi is available inside this Closure

    }
  }
}
```

Accessing IModeManagementApi in a scope-like way - Kotlin

```
scriptTask("taskName") {
  code {
    // IRuntimeSystemApi is available as "runtimeSystem" property
    var runtimeSystem = domain.runtimeSystem
  }
}
```

Accessing IRuntimeSystemApi as a property - Kotlin

```
scriptTask("taskName") {
  code {
    domain.runtimeSystem {
      // IRuntimeSystemApi is available inside this Closure

    }
  }
}
```

Accessing IRuntimeSystemApi in a scope-like way - Kotlin

```
scriptTask("createPortTerminator", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        // select the component port via the simple API and call terminate()
        // this API is more strict and throws an exception if the selected port
            cannot be terminated
        var terminatedPort = componentPort("COMPOSITIONTYPE", "rDelegationSRPort1"
            ).terminate()
        scriptLogger.info("Terminated component port '{0}'.", terminatedPort.
            getName())
      }
    }
  }
}
```

Create a port terminator using the simple API - Kotlin

```
scriptTask("removePortTerminator", DV_PROJECT){
  code {
    transaction {
      domain.runtimeSystem {
        // select the component port via the simple API and call terminate()
        // this API is more strict and throws an exception if the selected port
            has no port terminator to remove
        var removedTerminatorPort = componentPort("App1_1", "pSRPort1").
            removePortTerminator()
        scriptLogger.info("Terminated component port '{0}'.",
            removedTerminatorPort.getName())
      }
    }
  }
}
```

Remove a port terminator using the simple API - Kotlin

```
scriptTask("taskName") {
  code {
    // IDomainApi is available as "domain" property
    var domainApi = domain
  }
}
```

Accessing IDomainApi as a property - Kotlin

```
scriptTask("taskName") {
  code {
    domain {
      // IDomainApi is available inside this Closure

    }
  }
}
```

Accessing IDomainApi in a scope-like way - Kotlin

```
scriptTask("taskName") {
  code {
    // IProjectHandlingApi is available as "projects" property
    var projectHandlingApi = projects
  }
}
```

Accessing IProjectHandlingApi as a property - Kotlin

```
scriptTask("taskName") {
  code {
    projects {
      // IProjectHandlingApi is available inside this Closure

    }
  }
}
```

Accessing IProjectHandlingApi in a scope-like way - Kotlin

```
scriptTask("TaskName"){
    code{
        // Task execution code here

    }
}
```

Task creation with default type - Kotlin

```
scriptTask("TaskName"){
    taskDescription("The description of the task")
    code {

    }
}
```

Task with description - Kotlin

```
scriptTask("TaskName", DV_APPLICATION){
    code{
        // Task execution code here

    }
}
```

Task creation with TaskType Application - Kotlin

```
scriptTask("TaskName", DV_PROJECT){
    code{
        // Task execution code here

    }
}
```

Task creation with TaskType Project - Kotlin

```
scriptTask("TaskName"){
    code{
        // Use the scriptLogger to log messages
        scriptLogger.info("My script is running")
        scriptLogger.warn("My Warning")
        scriptLogger.error("My Error")
        scriptLogger.debug("My debug message")
        scriptLogger.trace("My trace message")
        scriptLogger.error("My Error", )
        // Also log an Exception as second argument
    }
}
```

Usage of the script logger - Kotlin

```
scriptTask("TaskName"){
    code{ argument ->
        // Use the format methods to insert data
        scriptLogger.info("My script {0} with:{1}", scriptTask, argument)
    }
}
```

Usage of the script logger with message formatting - Kotlin

```
scriptTask("TaskName", DV_PROJECT){
    code {
        // Switch to the transactions API
        transactions {

            //Check if a transaction is running
          transaction {

            // Open a transaction
          }
                // Now a transaction is running
        }
        transactions.isTransactionRunning()
    }
        // Or the short form
}
```

Check if a transaction is running - Kotlin

```
// Execute the model synchronization
// Execute the model synchronization
modelSynchronization.synchronize()
//Or more elaborated, but means the same

//Or more elaborated, but means the same
modelSynchronization{
  if(synchronizationRequired){
    synchronize()
  }
}
modelSynchronization.synchronize()
modelSynchronization {
  if (synchronizationRequired) {
    synchronize()
  }
}
```

Model synchronization inside an open project - Kotlin

```
scriptTask("AccessAsPropertyTask") {
  code {
    // IUnresolvedReferenceApi is available as unresolvedReferences property
    var unresolvedReferencesApi = unresolvedReferences
  }
}
```

Accessing IUnresolvedReferenceApi as a property - Kotlin

```
scriptTask("AccessLikeScopeTask") {
  code {
    unresolvedReferences {
      // IUnresolvedReferenceApi is available inside this Closure

    }
  }
}
```

Accessing IUnresolvedReferenceApi in a scope-like way - Kotlin