DaVinci Configurator AutomationInterface

Development Documentation of the AutomationInterface (AI)

DaVinci Configurator Team

September 19, 2016

Vector Informatik GmbH
Ingersheimerstr. 24
70499 Stuttgart

Contents

1 Introduction

1.1 General

The user of the DaVinci Configurator Pro can create scripts, which will be executed inside of the Configurator to:

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

The scripts are written by the *user* with the DaVinci Configurator AutomationInterface.

1.2 Facts

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

Languages The scripts are written in Groovy or Java. See 3.2 on page 15 for details.

Debugging Support The scripts can be debugged via IntelliJ IDEA. See 7.5 on page 140.

Documentation The AutomationInterface provides a comprehensive documentation:

- This document
- Javadoc HTML pages as class reference
- Script samples and templates
  - ScriptProject creation assistant in the DaVinci Configurator
- API documentation inside of an IDE
- Integrated Definition (BSWMD) description for all modules in the SIP

Code Completion You have code completion for Groovy and Java for the DaVinci Configurator AutomationInterface. You have to use IntelliJ IDEA for code completion.[1]

There is also a SIP based code completion for contained Module, Container and Parameter definitions. This eases the traversal through the AUTOSAR model.

[1] See chapter 7 on page 137 for details.

2 Getting started with Script Development

2.1 General

This chapter gives a short introduction of how to get started with script file or script project creation.

2.2 Automation Script Development Types

The DaVinci Configurator supports two types of automation scripts:

- Script files (`.dvgroovy` files)
- Script projects (`.jar` files)

Script File The script file 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 file proceed with chapter 2.3.

Script Project The script project is **more effort** to create and maintain, but provides IDE support for:

- Code completion
- Syntax highlighting
- API Documentation
- Debug support
- Build support

It is the **recommended way to develop** scripts, containing more tasks or multiple classes.

To create a script project proceed with chapter 2.4 on page 10.

2.3 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 need support by the tool during implementing the script and if debugging is not required.

Prerequisites Before you start, please make sure that you have a **SIP** containing a DaVinci Configurator 5 available on your system.

Creation Inside your SIP you find examples of automation script files. Create your own script folder and copy an example, e.g. `...ScriptSamples/SimpleScript.dvgroovy` to your folder.

Rename the script file and open it in any text editor. In case of `SimpleScript.dvgroovy` it consists of several tasks. One of the tasks will print a "HelloApplication" string to the console.

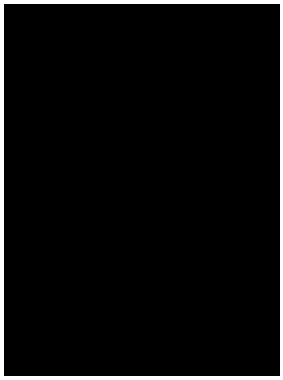


Figure 2.1: Script Samples location

Open the DaVinci Configurator inside your SIP. If not yet visible open the Views

- Script Locations
- Script Tasks

via the View menu.

In the **Script Locations** View select the location folder User@Machine. On its context menu you can **Add** a script location. Select your own script folder.

Figure 2.2: Script Locations View

Alternatively you could add the script location to the Session folder. In this case the script location would only be stored in the current session.

Switch to the **Script Tasks** View. It provides an overview over the tasks contained in your script.

Figure 2.3: Script Tasks View

Execute the SimpleAppTask by double-click or by the Execute Command contained in its context menu or by the Execute Button of the Task View and check that "HelloApplication" is printed in the console.

You can modify the implementation according to your needs. For the AutomationInterface API Reference see chapter 4 on page 19. It is sufficient to edit and save the modifications in your editor. The file is automatically reloaded by the DaVinci Configurator then and can be executed immediately.

Debugging It is not possible to debug a script file, if you want to debug, please migrate to a script project, see chapter 2.4.

2.4 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 a normal IDE project (IntelliJ IDEA recommended), with compile bindings to the DaVinci Configurator AutomationInterface.

The DaVinci Configurator 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.

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

- **SIP** containing a DaVinci Configurator 5
- **IDE**: For the script project development the *recommended* IDE is *IntelliJ IDEA*. Please install the IDE as described in chapter 2.4.3 on page 12.
- **Java SDK**: For the development with the IntelliJ IDEA a "Java SE Development Kit 8" (JDK 8) is required. Please install the JDK 8 as described in chapter 2.4.2 on page 12.
- **Build system**: To build the script project the build system Gradle is required. See chapter 2.4.4 on page 13 for details.

Creation Open the DaVinci Configurator inside your SIP. If not yet visible open the Views

- Script Locations
- Script Tasks

via the View menu.

Switch to the View **Script Tasks** and select the Button **Create New Script Project...**.

Figure 2.4: Create New Script Project... Button

A **Project Settings** dialog opens where you have to specify the

- **Script Project Name**

  - Define a name for your new project.
- **Project Location**
  - Select a folder in which your project shall be created.
- **Gradle Distribution URL**
  - Select one option:
    * **Gradle Default**
      · Use the Gradle build system provided by default. To use this option you need **internet access**.
    * **Custom URL**
      · Specify an URL to your own build system. A new dialog opens and supports you in specifying the Custom Gradle URL. To setup the Gradle build system see 2.4.4 on page 13.
- **Open IntelliJ IDEA**
  - Select this option if the project shall automatically be opened in IntelliJ IDEA. In case IntelliJ IDEA is not installed on your system but selected here, a warning will be issued.

Figure 2.5: Project Settings

Finish the dialog.

A new project will be created. Necessary tasks as setting up the IntelliJ IDEA and building the project are automatically initiated.

For detailled information to the **Build System** please see chapter 7.6 on page 140.

The View **Script Tasks** provides an overview over the scripts and tasks contained in the project. The newly created project already contains a sample script file `MyScript.groovy`. The sample

script file contains one task that prints a "HelloApplication" string to the console. Run and check it as already described in 2.3 on page 8.

The View **Script Locations** contains the path to the script project file which is a `.jar` file.

You can modify now the implementation according to your needs. For the AutomationInterface API Reference see chapter 4 on page 19. To edit and rebuild the project use IntelliJ IDEA.

After each build the project is automatically reloaded by the DaVinci Configurator and can be executed there.

IntelliJ IDEA Usage Ensure that the Gradle JVM and the Project SDK are set in the IntelliJ IDEA Settings. For details see 2.4.3.

Having modified and saved `MyScript.groovy` in the IntelliJ IDEA editor you can build the project by pressing the **Run Button provided in the toolbar**. The functionality of this Run Button is determined by the option selected in the Menu beneath this button. In this menu **<ProjectName> [build] shall be selected**.

Figure 2.6: Project Build

For more information to IntelliJ IDEA usage please see chapter 7.4 on page 138. If you have trouble with IntelliJ, see 7.4.4 on page 140.

Debugging To debug the script project follow the instructions in chapter 7.5 on page 140.

2.4.1 Script Project Development

For more details to the development of a script project see chapter 7 on page 137.

2.4.2 Java SDK Setup

Install a JDK 8 on your system. The Java SDK website provides download versions for different systems. Download an appropriate version.

The architecture is not relevant, both x86 and x64 are valid.

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

2.4.3 IntelliJ IDEA Setup

Install IntelliJ IDEA on your system. The IntelliJ IDEA website provides download versions for different applications. Download a version that supports Java and Groovy and that is in the list of supported versions (see list 7.4.1 on page 138).

Code completion and compilation additionally require that the Project SDK is set. Therefore open the File -> **Project Structure** Dialog in IntelliJ IDEA and switch to the settings dialog for **Project**. If not already available set an appropriate option for the **Project SDK**.

Figure 2.7: Project SDK Setting

To enable building of projects ensure that the Gradle JVM is set. Therefore open the File -> **Settings** Dialog in IntelliJ IDEA and find the settings dialog for **Gradle**. If not already available set an appropriate option for the **Gradle JVM**. If you do not have the Gradle settings, please make sure that the Gradle plugin inside of IntelliJ IDEA is installed. Open the File -> **Settings** Dialog then Plugins and select the Gradle plugin.

Figure 2.8: Gradle JVM Setting

2.4.4 Gradle Setup

If your system has internet access you can use the default Gradle Build System provided by the DaVinci Configurator. In this case you do not have to install your own build system.

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. Download at least version 2.13.

See chapter 7.6 on page 140 for more details to the Build System.

3 AutomationInterface Architecture

3.1 Components

The DaVinci Configurator 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 components calls.

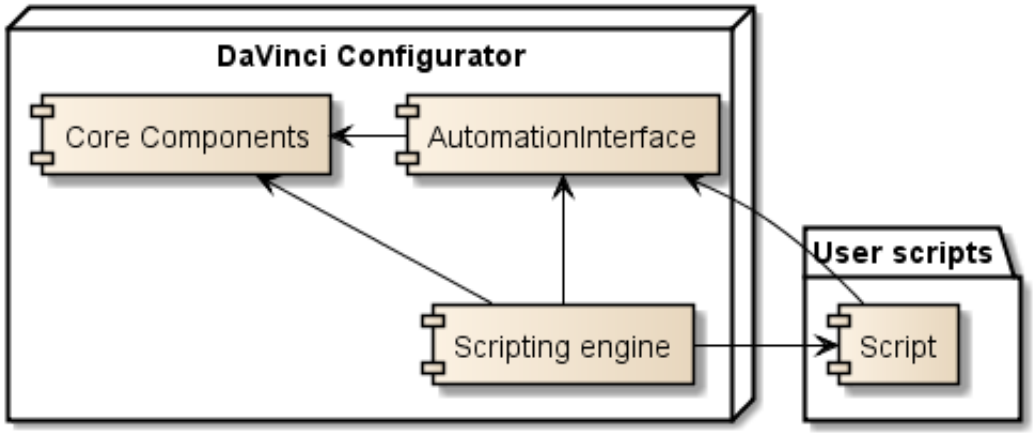


Figure 3.1: DaVinci Configurator components and interaction with scripts

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

- Stable API for script writters
  - Including checks, that the API will not break in following releases
- Well defined and documented API
- Abstraction from the internal heavy lifting
  - This ease 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 shall not be used by any client code, nor shall a client call methods via reflection or other runtime techniques.

You shall **not** access private or package private classes, methods or fields.

3.2 Languages

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

- Java[1]
- Groovy[2]

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

3.2.1 Why Groovy

Flat Learning Curve Groovy is concise, readable with an expressive syntax and is easy to learn for Java developers[3].

- Groovy syntax is 95%-compatible with Java[4]
- Any Java developer will be able to code in Groovy without having to know nor understand the subtleties of this language

This is very important for teams where there's not much time for learning a new language.

Domain-Specific Languages (DSL) Groovy has a flexible and malleable syntax, advanced integration and customization mechanisms, to integrate readable business rules in your applications.

The DSL features of Groovy are extensively used in DaVinci Automation API to provide simple and expressive syntax.

Powerful Features Groovy supports Closures, builders, runtime & compile-time meta-programming, functional programming, type inference, and static compilation.

Website The website of Groovy is http://groovy-lang.org. It provides a good documentation and starting guides for the Groovy language.

Groovy Book The book **"Groovy in Action, Second Edition"**[5] provides a comprehensive guide to Groovy programming language. It is written by the developers of Groovy.

3.3 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`.

[1] `http://http://www.java.com` [2016-05-09]

[2] `http://groovy-lang.org` [2016-05-09]

[3] Copied from `http://groovy-lang.org` [2016-05-09]

[4] Copied from `http://melix.github.io/blog/2010/07/27/experience_feedback_on_groovy.html` [2016-05-09]

[5] Groovy in Action, Second Edition by Dierk König, Paul King, Guillaume Laforge, Hamlet D'Arcy, Cédric Champeau, Erik Pragt, and Jon Skeet June 2015 ISBN 9781935182443
`https://www.manning.com/books/groovy-in-action-second-edition` [2016-05-09]

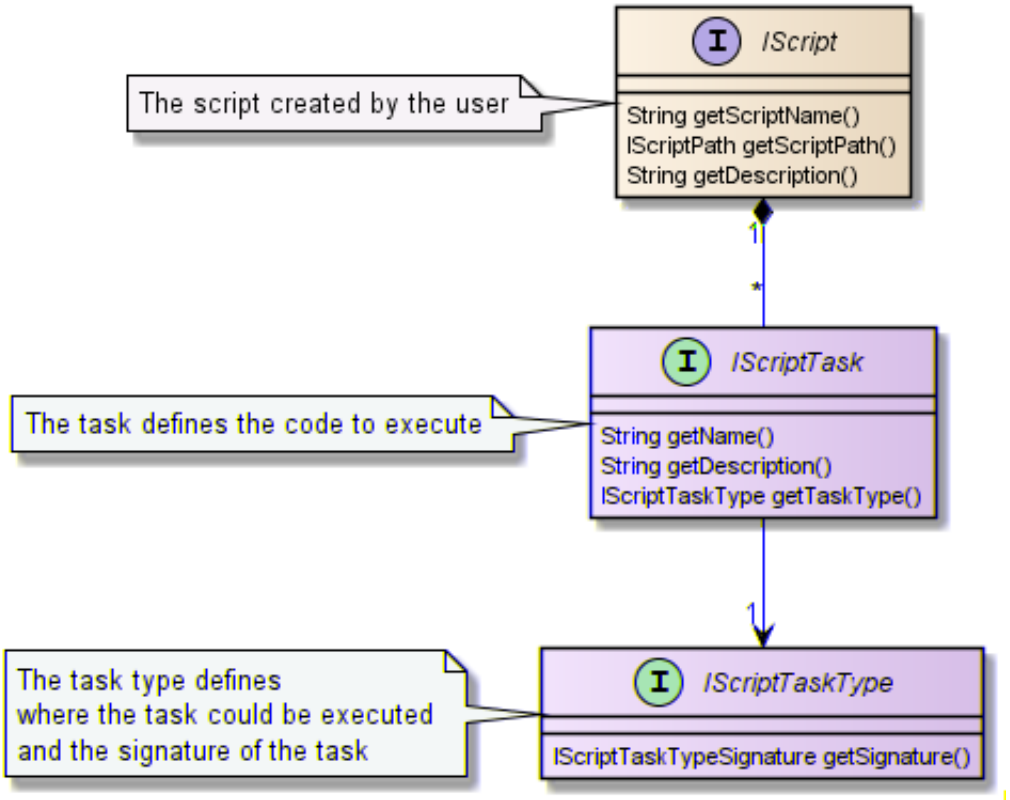


Figure 3.2: Structure of scripts and script tasks

You create the `IScript` and `IScriptTask` instance with the API described in chapter 4.2 on page 20.

The script task type (`IScriptTaskType`) defines where the task could be execute nand the signature of the task `code {}` block. See chapter 4.3 on page 24 for the available types.

3.3.1 Scripts

Script contain the tasks to execute and are loaded from the script locations specified in the DaVinci Configurator.

The DaVinci Configurator supports two types of automation scripts:

- Script files (`.dvgroovy` files)
- Script projects (`.jar` files)

For details to the script project, see chapter 7 on page 137.

3.3.2 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.

3.3.3 Script Locations

Script locations define where script files are loaded from. These locations are edited in the DaVinci Configurator Script Locations view. You can also start the Configurator with the option `-scriptLocations` to specify additional locations.

The DaVinci Configurator could load scripts from different script locations:

- SIP
- Project
- User-defined directories
- More

3.4 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: A jar file of 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.

3.5 Script editing

The DaVinci Configurator does not contain any editing support for scripts, like:

- Script editor
- Debugger
- REPL (Read-Eval-Print-Loop)

These tasks are delegated to other development tools:

- IntelliJ IDEA (recommended)
- EclipseIDE
- Notepad++

See chapter 7 on page 137 for script development and debugging with IntelliJ IDEA.

3.6 Licensing

The DaVinci Configurator requires certain license options to develop and/or execute script tasks.

The required license options differ between development and execution time. Normally you need more license options to develop scripts than you need to execute them.

The default license options are:

- .PRO option for execution
- .WF option for development and debugging

The license option .MD includes the option .WF for automation scripts. So you can also use .MD as replacement of .WF.

Some script task may require different options during development or execution. It is also possible that the execution does not require any license option. See chapter 4.3 on page 24 for details, which script task type requires which license.

4 AutomationInterface API Reference

4.1 Introduction

This chapter contains the description of the DaVinci Configurator AutomationInterface. The figure 4.1 shows the APIs and the containment structure of the different APIs.

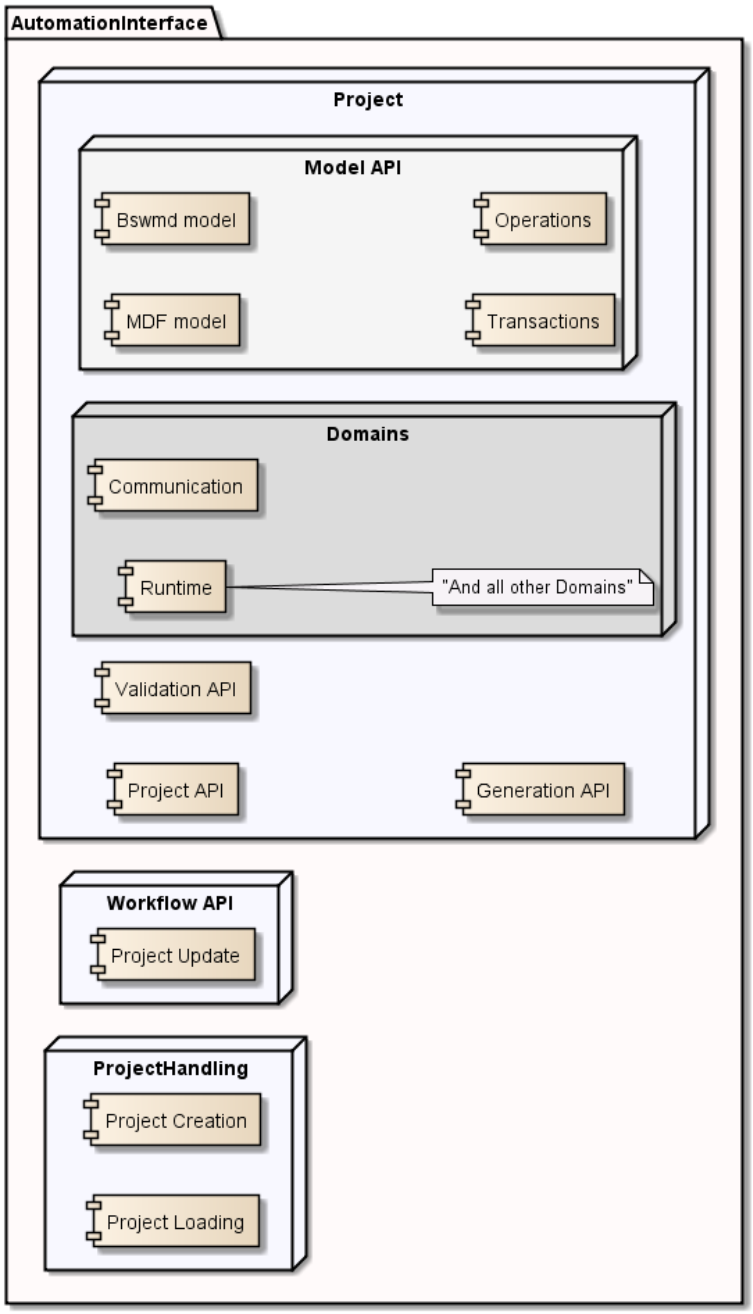


Figure 4.1: The API overview and containment structure

The components have an 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.

Usage examples:

- The Generation API is only usable inside of a loaded Project
- The Workflow Update API is only usable outside of a loaded Project

4.2 Script Creation API

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

4.2.1 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 4.3 on page 24 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 an script task for the default `IScriptTaskType DV_PROJECT`.

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

Listing 4.1: 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, Closure)`
- `scriptTask(String, IProjectScriptTaskType, Closure)`

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 4.3 on page 24


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

Listing 4.2: Task creation with TaskType Application

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

Listing 4.3: 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 4.4: Define two tasks is one script

4.2.1.1 Script Creation with IDE Code Completion Support

The IDE could not know which API is available inside of a script file. So a glue code is needed to tell the IDE, what API is callable inside of a script file.

The `ScriptApi.daVinci()` 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 4.5: 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 (`.dvgroovy`)

4.2.1.2 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 4.6: Task with isExecutableIf

4.2.2 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 4.7: 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 4.8: 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 4.9: Task with description and help text

4.3 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.

License Options For the common explanation of the required license options, see chapter 3.6 on page 18.

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

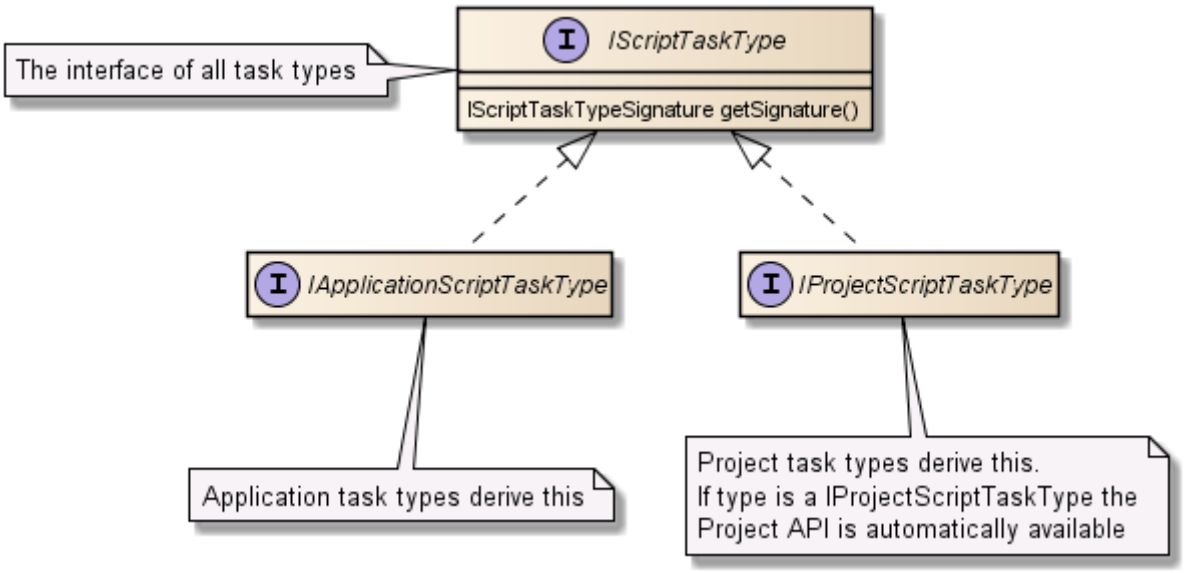


Figure 4.2: IScriptTaskType interfaces

4.3.1 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.

4.3.1.1 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
Required license option	Development: .WF Execution: .PRO

4.3.1.2 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
Required license option	Development: .WF Execution: .PRO

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

Name	Module activation
Code identifier	DV_MODULE_ACTIVATION
Task type interface	`IProjectScriptTaskType`
Parameters	`MIModuleConfiguration moduleConfiguration`
Return type	`Void`
Execution	Automatically during module activation
Required license option	Development: .WF Execution: .PRO

4.3.1.3 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
Required license option	Development: .WF Execution: .PRO

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
Required license option	Development: .WF Execution: .PRO

4.3.1.4 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 ProjectStettingsEditor, 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
Required license option	Development: .MD Execution: None

See chapter 4.7.2 on page 78 for usage samples.

Generation Process Start The type `DV_GENERATION_ON_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_GENERATION_ON_START
Parameters	`List<EGenerationPhaseType> generationPhases`
	`List<IGenerator> executedGenerators`
Return type	`Void`
Execution	Automatically before GenerationProcess
Required license option	Development: .MD Execution: None

See chapter 4.7.2 on page 78 for usage samples.

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

Name	Generation Process End
Code identifier	DV_GENERATION_ON_END
Task type interface	`IProjectScriptTaskType`
Parameters	`EGenerationProcessResult processResult`
	`List<IGenerator> executedGenerators`
Return type	`Void`
Execution	Automatically after GenerationProcess
Required license option	Development: .MD Execution: None

See chapter 4.7.2 on page 78 for usage samples.

4.4 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

4.4.1 Execution Context

Every `IScriptTask` could be 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 describes in their own chapters like `IProjectHandlingApiEntryPoint` or `IWorkflowApiEntryPoint`.

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"){
  code{  // The IScriptExecutionContext is automatically active here
     // Call methods of the IScriptExecutionContext
     def logger = scriptLogger
     def temp = paths.tempFolder

     // Use an automation API
     workflow{
         // Now the Workflow API is active
     }
  }
}
```

Listing 4.10: Access automation API in Groovy clients by the IScriptExecutionContext

In Groovy the `IScriptExecutionContext` is automatically activated inside of 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
IWorkflowApi workflow = scriptContext.getInstance(IWorkflowApiEntryPoint.class)
    .getWorkflow();
IWorkflowContext workflowCtx = workflow.getWorkflow();


// In groovy code it would be:
workflow{

}
```

Listing 4.11: 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`).

4.4.1.1 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 4.3 on page 24 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 4.12: Script task code block arguments

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

4.4.2 Task Execution Sequence

The figure 4.3 on the following page 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.

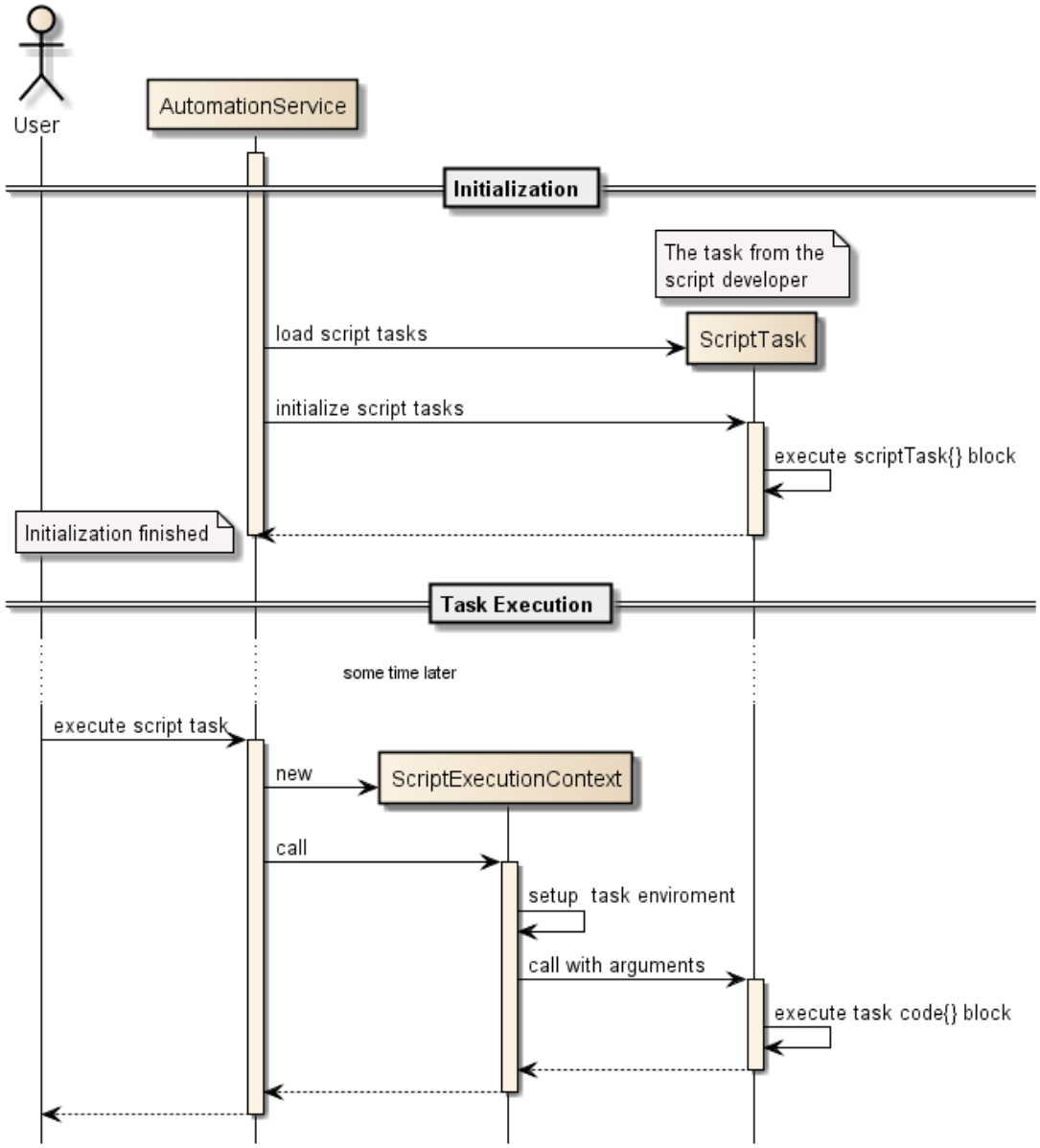


Figure 4.3: Script Task Execution Sequence

4.4.3 Script Path API during Execution

Script tasks could resolve relative and absolute file system paths with the `IAutomationPathsApi`.

As entry point call `paths` in a `code{ }` block (see `IScriptExecutionContext.getPaths()`).

There are multiple ways to resolve relative paths:

- by Script folder
- by Temp folder
- by SIP folder
- by Project folder
- by any parent folder

4.4.3.1 Path Resolution by Parent Folder

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{
       // Method resolvePath(Path, Object) resolves a path relative to the
           supplied folder
       Path parentFolder = Paths.get('.')
       Path p = paths.resolvePath(parentFolder, "MyFile.txt")

      /* The resolvePath(Path, Object) method will resolve
       * relative and absolute paths to a java.nio.file.Path object.
       */
    }
}
```

Listing 4.13: Resolves a path with the resolvePath() method

4.4.3.2 Path Resolution

The `resolvePath(Object)` method resolves the `Object` to a file path. Relative paths are preserved, so relative paths are not converted into absolute paths.

This method converts the supplied path same as the `resolvePath(Path, Object)` method. The return type is `java.nio.file.Path`. See 4.4.3.1. But it does **NOT** convert relative paths into absolute.

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

      /* The resolvePath() method will resolve
       * relative and absolute paths to a java.nio.file.Path object.
       * Is also preserves relative paths.
       */
    }
}
```

Listing 4.14: Resolves a path with the resolvePath() method

4.4.3.3 Script Folder Path Resolution

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 4.4.3.1 on the preceding page.

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

      /* The resolveScriptPath() method will resolve
       * relative and absolute paths to a java.nio.file.Path object.
       */
    }
}
```

Listing 4.15: Resolves a path with the resolveScriptPath() method

4.4.3.4 Project Folder Path Resolution

The `resolveProjectPath(Object)` method resolves a file path relative to the project directory (see `getDpaProjectFolder()`) 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 4.4.3.1 on the previous page.

There must be an active project to use this method. See chapter 4.5.2 on page 43 for details about active projects.

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

      /* The resolveProjectPath() method will resolve
       * relative and absolute paths to a java.nio.file.Path object.
       */
    }
}
```

Listing 4.16: Resolves a path with the resolveProjectPath() method

4.4.3.5 Sip Folder Path Resolution

The `resolveSipPath(Object)` method resolves a file path relative to the SIP directory (see `getSipRootFolder()`).

This method converts the supplied path same as the `resolvePath(Path, Object)` method. The return type is `java.nio.file.Path`. See 4.4.3.1 on page 31.

```
scriptTask("TaskName"){
    code{
       // Method resolveSipPath() resolves a path relative SIP folder
       Path p = paths.resolveSipPath("MyFile.txt")

      /* The resolveSipPath() method will resolve
       * relative and absolute paths to a java.nio.file.Path object.
       */
    }
}
```

Listing 4.17: Resolves a path with the resolveSipPath() method

4.4.3.6 Temp Folder Path Resolution

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 4.4.3.1 on page 31.

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

      /* The resolveTempPath() method will resolve
       * relative and absolute paths to a java.nio.file.Path object.
       */
    }
}
```

Listing 4.18: Resolves a path with the resolveTempPath() method

4.4.3.7 Other Project and Application Paths

The `IAutomationPathsApi` will also resolve any other Vector provided path variable like `$(EcucFile)`. The call would be `paths.ecucFile`, add the variable to resolve as a Groovy property.

Short list of available variables (not complete, please see DaVinci Configurator help for more details):

- EcucFile
- OutputFolder
- SystemFolder
- AutosarFolder
- more ...

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

Listing 4.19: Get the project output folder path

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

Listing 4.20: Get the SIP folder path

4.4.4 Script logging API

The script task execution (`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
- More...

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 4.21: 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.infoFormat("My script {0} with:{1}", scriptTask, argument)
    }
}
```

Listing 4.22: 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 4.23: Usage of the script logger with Groovy GString message formatting

4.4.5 Script Error Handling

4.4.5.1 Script Exceptions

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

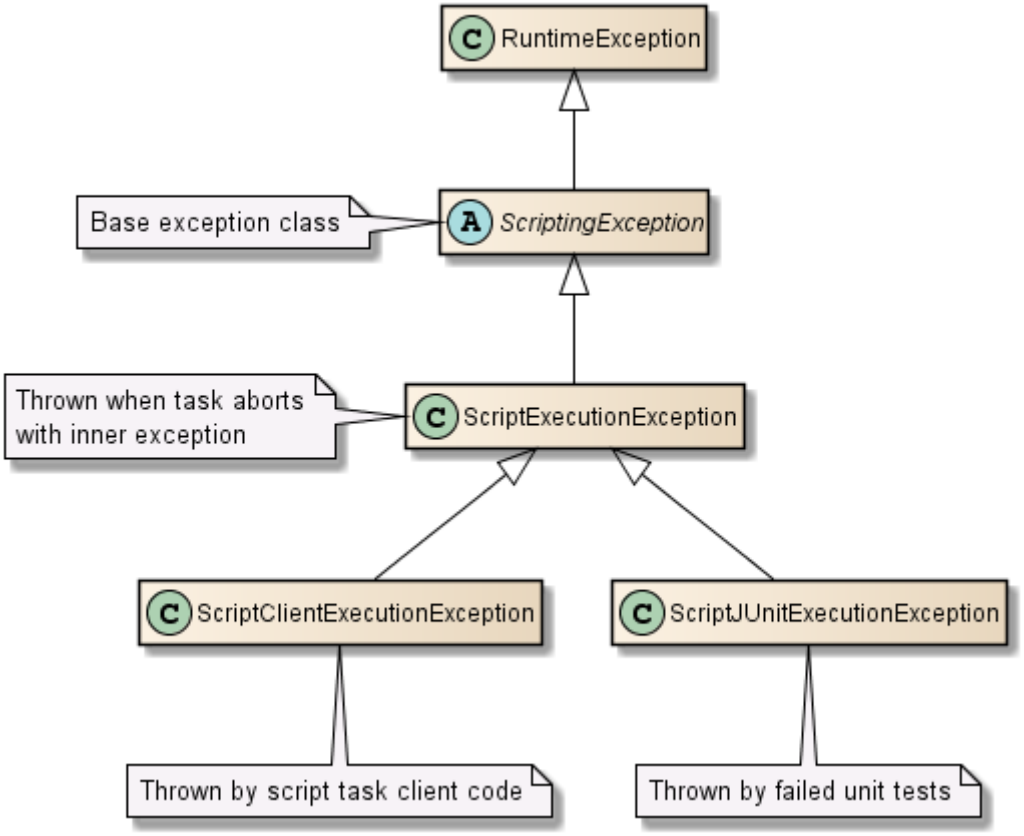


Figure 4.4: ScriptingException and sub types

4.4.5.2 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 4.24: 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 4.25: 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.

4.4.5.3 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.

4.4.6 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 4.26: 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 4.27: 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 of the `daVinci{ }` block.

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

def userMethod(){}

class UserClass{}
```

Listing 4.28: Using your own defined method with a daVinci block

4.4.7 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`.

4.4.7.1 Access to API like the Script code{} Block

The `ScriptApi.scriptCode(Closure)` method provides access to all automation APIs defined in the `IScriptExecutionContext` (inside of the normal script `code{}` block. This is useful, when you want to call script code API inside of your own methods and classes.

The method will call the passed `Closure` with the running `IScriptExecutionContext` as delegate.

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

Listing 4.29: ScriptApi.scriptCode{} usage in own method

The `ScriptApi.scriptCode()` without a `Closure` method returns the currently running `IScriptExecutionContext`. This can be used to call API in Java style.

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

Listing 4.30: ScriptApi.scriptCode() usage in own method

4.4.7.2 Access to 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, when you want to call project API inside of your own methods and classes.

The method will call the passed `Closure` with the current active `IProject` as delegate.

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

Listing 4.31: 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 4.32: ScriptApi.activeProject() usage in own method

4.4.8 User defined Script Task Arguments in Commandline

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.

Possible valueTypes are:

- `String`
- `Boolean`
- `Void`: For parameter where only the existence is relevant.
- `File`: The existence of the file is checked. To reference non existing paths use `String` instead.
- `Path`: Same as `File`
- `Integer`
- `Long`
- `Double`

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

```
scriptTask("TaskName"){
  /*
   * newUserDefinedArgument(String argName, Class<T> valueType, String help)
   */
  def countArg = newUserDefinedArgument("count", Integer,
                                        "The amount of elements to create")

  def nameArg = newUserDefinedArgument("name", String,
                                       "The element name to create")
  code{

      int count = countArg.value
      String name = nameArg.value

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

Listing 4.33: Define and use script task user defined arguments from commandline

```
scriptTask("TaskName"){
  /*
   * newUserDefinedArgument(String argName, Class<T> valueType, String help)
   */
  def procArg = newUserDefinedArgument("p", Void,
                                       "Enables the processing")
  code{

      if(procArg.hasValue){
        scriptLogger.info  "The argument -p was defined"
      }
  }
}
```

Listing 4.34: Script task UserDefined argument with no value

```
scriptTask("TaskName"){
  /*
   * newUserDefinedArgument(String argName, Class<T> valueType,
   *                        T defaultValue, String help)
   */
  def procArg = newUserDefinedArgument("p", Double, 25.0,
                                       "Help text ...")
  code{

      double value = procArg.value
      scriptLogger.info  "The argument -p was $value"
  }
}
```

Listing 4.35: Script task UserDefined argument with default value

```
scriptTask("TaskName"){
  /*
   * newUserDefinedArgument(String argName, Class<T> valueType, String help)
   */
  def multiArg = newUserDefinedArgument("multiArg", String, "Help text ...")

  /*
   * The client calls the task with arguments:
   *  --multiArg "ArgOne" --multiArg "ArgTwo"
   */
  code{

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

Listing 4.36: Script task UserDefined argument with multiple values

4.4.8.1 Call Script Task with Task Arguments

The commandline option `taskArgs` is used to specify the arguments passed to a script task to execute:

```
--taskArgs <TASK_ARGS>   Passes arguments to the specified script tasks.
```

The arguments have the following syntax:

```
Syntax: --taskArgs "<TaskName>" "<Arguments to Task>"
   E.g. --taskArgs "MyTask" "-s --projectCfg MyFile.cfg"
```

If only one task is executed, the `"<TaskName>"` can be omitted.

For multiple task arguments the following syntax apply:

```
Syntax: --taskArgs "<TaskName>" "<Arguments to Task>"
                   "<TaskName2>" "<Arguments to Task2>"

   E.g. --taskArgs "MyTask" "-s --projectCfg MyFile.cfg"
                   "Task2" "-d --saveTo saveFile.txt"
```

Note: The newlines in the listing are only for visualization.

If the task name is not unique, your can specify the full qualified name with script name

```
  --taskArgs "MyScript:MyTask" "-s --projectCfg MyFile.cfg"
```

Arguments with spaces inside the script task argument could be quoted with ""

```
  --taskArgs "MyScript:MyTask" "-s --projectCfg \"Path to File\MyFile.cfg\" -d"
```

The task help of a task will print the possible arguments of a script task.

```
  --scriptTaskHelp taskName
```

4.5 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 4.37: Accessing IProjectHandlingApi as a property

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

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

Listing 4.38: Accessing IProjectHandlingApi in a scope-like way

4.5.1 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 `Closure` block at `IProjectHandlingApi` or `IProjectRef` class. This shall prevent memory leaks, by not closing open projects.

4.5.2 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 `Closure` block
  - Stack-based - so multiple opened projects are possible and the last (inner) `Closure` 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 4.5 describes the behavior to search for the active project of a script task.

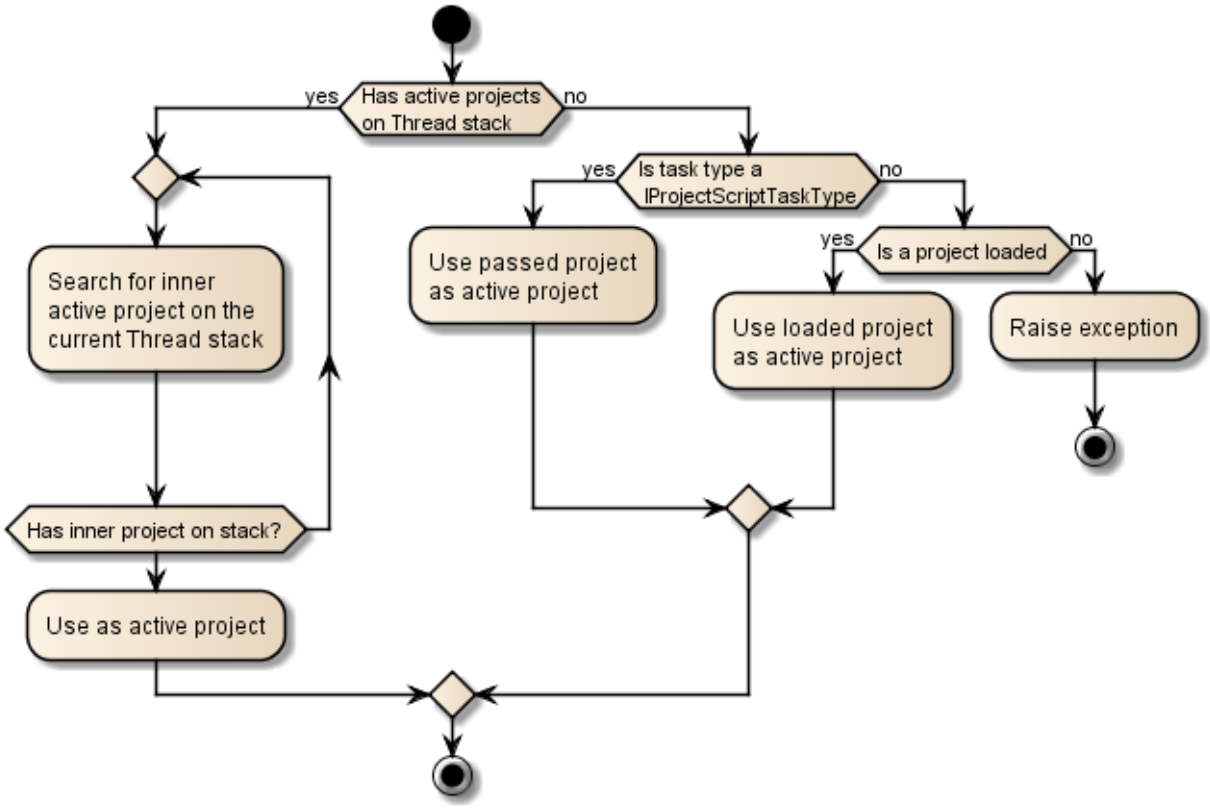


Figure 4.5: Search for active project in getActiveProject()

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

You can switch the active project, by calling the `with(Closure)` 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 closure
}
```

Listing 4.39: Switch the active project

To access the active project you can use the `activeProject(Closure)` 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 4.40: 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(Closure)` allows accessing the active `IProject` in a scope-like way. This will enable the project specific API inside of the `Closure`.

4.5.3 Creating a new Project

`createProject(Closure)` creates a new project as specified by the given `Closure` which is delegated to the `ICreateProjectApi`. The new project is not opened and usable until `IProjectRef.openProject(Closure)` is called on the `IProjectRef` returned by this method.

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

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

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

Listing 4.41: Creating a new project (mandatory parameters only)

The next is a more sophisticated example of creating a project:

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

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

      general {
        author 'projectAuthor'
        version '0.9'
      }

      postBuild {
        loadable true
        selectable true
      }

      folders.ecucFileStructure = ONE_FILE_PER_MODULE
      folders.moduleFilesFolder = 'Appl/GenData'
      folders.templatesFolder = 'Appl/Source'

      target.vVIRTUALtargetSupport = false

      daVinciDeveloper.createDaVinciDeveloperWorkspace = false
    }
  }
}
```

Listing 4.42: Creating a new project (with some optional parameters)

`ICreateProjectApi` provides means for parameterizing the creation of a new project.

4.5.3.1 Mandatory Settings

Project Name Using `setProjectName(String)` the name for the newly created project can be specified. The name given here is postfixed with ".dpa" for the new project's .dpa file. The following constraints apply:

- `Constraints.IS_VALID_PROJECT_NAME` 4.11.1 on page 98

Project Folder Using `setProjectFolder(Object)` the folder in which to create the new project can be specified. The value given here is converted to `Path` using the converter `Converters.TO_SCRIPT_PATH` 4.11.2 on page 99. The following constraints apply:

- `Constraints.IS_CREATABLE_FOLDER` 4.11.1 on page 99

4.5.3.2 General Settings

Using `getGeneral()` the `ICreateProjectGeneralApi` for specifying the new project's general settings can be retrieved like a property.

Using `general(Closure)` the `ICreateProjectGeneralApi` for specifying the new project's general settings can be access in a scope-like way.

`ICreateProjectGeneralApi` provides means for specifying the new project's general settings.

Author Using `setAuthor(String)` the author for the new project can be specified. 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` 4.11.1 on page 98

Version Using `setVersion(Object)` the version for the new project can be specified. 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 `Converters.TO_VERSION` 4.11.2 on page 99. The following constraints apply:

- `Constraints.IS_NOT_NULL` 4.11.1 on page 98

Description Using `setDescription(String)` the description for the new project can be specified. This is an optional parameter defaulting to "" if the parameter is not provided explicitly. The following constraints apply:

- `Constraints.IS_NOT_NULL` 4.11.1 on page 98

Start Menu Entries Using `setCreateStartMenuEntries(boolean)` it can be specified whether or not to create start menu entries for the new project. This is an optional parameter defaulting to `false` if the parameter is not provided explicitly.

4.5.3.3 Target Settings

Using `getTarget()` the `ICreateProjectTargetApi` for specifying the new project's target settings can be retrieved like a property.

Using `target(Closure)` the `ICreateProjectTargetApi` for specifying the new project's target settings can be access in a scope-like way.

`ICreateProjectTargetApi` provides means for specifying the new project's target settings.

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

Derivative Using `setDerivative(DerivativeInfo)` the derivative for the new project can be specified. This is an optional parameter defaulting to the first element in the collection returned by `getAvailableDerivatives()` (or `null` if the collection is empty). The value given here must be one of the values returned by `getAvailableDerivatives()`.

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 Using `setCompiler(ImplementationProperty)` the compiler for the new project can be specified. This is an optional parameter defaulting to the first element in the collection returned by `getAvailableCompilers()` (or `null` if the collection is empty). The value given here must be one of the values returned by `getAvailableCompilers()`.

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 Using `setPinLayout(ImplementationProperty)` the pin layout for the new project can be specified. This is an optional parameter defaulting to the first element in the collection returned by `getAvailablePinLayouts()` (or `null` if the collection is empty). The value given here must be one of the values returned by `getAvailablePinLayouts()`.

vVIRTUALtarget Support Using `setvVIRTUALtargetSupport(boolean)` it can be specified whether or not to support the vVIRTUALTarget for the new project. This is an optional parameter defaulting to `false` if the parameter is not provided explicitly. The following constraints apply:

- vVIRTUALtarget support may not be available depending on the purchased license

4.5.3.4 Post Build Settings

Using `getPostBuild()` the `ICreateProjectPostBuildApi` for specifying the new project's post build settings can be retrieved like a property.

Using `postBuild(Closure)` the `ICreateProjectPostBuildApi` for specifying the new project's post build settings can be access in a scope-like way.

`ICreateProjectPostBuildApi` provides means for specifying the new project's post build settings.

Post Build Loadable Support Using `setLoadable(boolean)` it can be specified 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 Using `setSelectable(boolean)` it can be specified 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.

4.5.3.5 Folders Settings

Using `getFolders()` the `ICreateProjectFolderApi` for specifying the new project's folders settings can be retrieved like a property.

Using `folders(Closure)` the `ICreateProjectFolderApi` for specifying the new project's folders settings can be access in a scope-like way.

ICreateProjectFolderApi provides means for specifying the new project's folders settings.

Module Files Folder Using setModuleFilesFolder(Object) the module files folder for the new project can be specified. This is an optional parameter defaulting to ".\Appl\GenData" if the parameter is not provided explicitly. The value given here is converted to Path using Converters.TO_PATH 4.11.2 on page 99. Normally a relative path (to be interpreted relative to the project folder) should be given here. The following constraints apply:

- Constraints.IS_CREATABLE_FOLDER 4.11.1 on page 99

Templates Folder Using setTemplatesFolder(Object) the templates folder for the new project can be specified. This is an optional parameter defaulting to ".\Appl\Source" if the parameter is not provided explicitly. The value given here is converted to Path using Converters.TO_PATH 4.11.2 on page 99. Normally a relative path (to be interpreted relative to the project folder) should be given here. The following constraints apply:

- Constraints.IS_CREATABLE_FOLDER 4.11.1 on page 99

Service Components Folder Using setServiceComponentFilesFolder(Object) the service component files folder for the new project can be specified. This is an optional parameter defaulting to ".\Config\ServiceComponents" if the parameter is not provided explicitly. The value given here is converted to Path using Converters.TO_PATH 4.11.2 on page 99. Normally a relative path (to be interpreted relative to the project folder) should be given here. The following constraints apply:

- Constraints.IS_CREATABLE_FOLDER 4.11.1 on page 99

Application Components Folder Using setApplicationComponentFilesFolder(Object) the application component files folder for the new project can be specified. This is an optional parameter defaulting to ".\Config\ApplicationComponents" if the parameter is not provided explicitly. The value given here is converted to Path using Converters.TO_PATH 4.11.2 on page 99. Normally a relative path (to be interpreted relative to the project folder) should be given here. The following constraints apply:

- Constraints.IS_CREATABLE_FOLDER 4.11.1 on page 99

Log Files Folder Using setLogFilesFolder(Object) the log files folder for the new project can be specified. This is an optional parameter defaulting to ".\Config\Log" if the parameter is not provided explicitly. The value given here is converted to Path using Converters.TO_PATH 4.11.2 on page 99. Normally a relative path (to be interpreted relative to the project folder) should be given here. The following constraints apply:

- Constraints.IS_CREATABLE_FOLDER 4.11.1 on page 99

Measurement And Calibration Files Folder Using setMeasurementAndCalibrationFilesFolder(Object) the measurement and calibration files folder for the new project can be specified. This is an optional parameter defaulting to ".\Config\McData" if the parameter is not provided explicitly. The folder object passed to the method is converted to Path using Converters.TO_PATH 4.11.2 on page 99. Normally a relative path (to be interpreted relative to the project folder) should be given here. The following constraints apply:

- `Constraints.IS_CREATABLE_FOLDER` 4.11.1 on page 99

AUTOSAR Files Folder Using `setAutosarFilesFolder(Object)` the AUTOSAR files folder for the new project can be specified. 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 `Converters.TO_PATH` 4.11.2 on page 99. Normally a relative path (to be interpreted relative to the project folder) should be given here. The following constraints apply:

- `Constraints.IS_CREATABLE_FOLDER` 4.11.1 on page 99

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.

Using `setEcucFileStructure(EEcucFileStructure)` the ECUC file structure for the new project can be specified.
This is an optional parameter defaulting to `EEcucFileStructure.SINGLE_FILE` if the parameter is not provided explicitly.

4.5.3.6 DaVinci Developer Settings

Using `getDaVinciDeveloper()` the `ICreateProjectDaVinciDeveloperApi` for specifying the new project's DaVinci Developer settings can be retrieved like a property.

Using `daVinciDeveloper(Closure)` the `ICreateProjectDaVinciDeveloperApi` for specifying the new project's DaVinci Developer settings can be access in a scope-like way.

`ICreateProjectDaVinciDeveloperApi` provides means for specifying the new project's DaVinci Developer settings.

Create DEV Workspace Using `setCreateDaVinciDeveloperWorkspace(boolean)` it can be specified whether or not to create a DaVinci Developer workspace for the new project. This is an optional parameter defaulting to `true` if and only if a compatible DaVinci Developer installation can be detected and the parameter is not provided explicitly.

DEV Executable Using `setDaVinciDeveloperExecutable(Object)` the DaVinci Developer executable for the new project can be specified. This is an optional parameter defaulting to the location of a compatible DaVinci Developer installation (if there is any) if the parameter is not provided explicitly. The value given here is converted to `Path` using `Converters.TO_SCRIPT_PATH` 4.11.2 on page 99. The following constraints apply:

- `Constraints.IS_COMPATIBLE_DA_VINCI_DEV_EXECUTABLE` 4.11.1 on page 99

DEV Workspace Using `setDaVinciDeveloperWorkspace(Object)` the DaVinci Developer workspace for the new project can be specified. This is an optional parameter defaulting to ".\Config\Developer\<ProjectName>.dcf" if the parameter is not provided explicitly. The value given here is converted to `Path` using `Converters.TO_PATH` 4.11.2 on page 99. Normally a relative path (to be interpreted relative to the project folder) should be given here. The following constraints apply:

- `Constraints.IS_DCF_FILE` 4.11.1 on page 99
- `Constraints.IS_CREATABLE_FOLDER` 4.11.1 on page 99 (applies to the parent `Path` of the given `Path` to the DaVinci Developer executable)

Import Mode Preset Using `setUseImportModePreset(boolean)` it can be specified whether or not to use the import mode preset for the new project. This is an optional parameter defaulting to `true` if the parameter is not provided explicitly.

Object Locking Using `setLockCreatedObjects(boolean)` it can be specified whether or not to lock created objects for the new project. This is an optional parameter defaulting to `true` if the parameter is not provided explicitly.

Selective Import The literals of `ESelectiveImport` define the alternative modes for the selective import into the DaVinci Developer workspace during project updates. The following alternatives are supported:

`ALL` results in selective import for all elements.

`COMMUNICATION_ONLY` results in selective import for communication elements only.

Using `setSelectiveImport(ESelectiveImport)` the selective import mode for the new project can be specified. This is an optional parameter defaulting to `ESelectiveImport.ALL` if the parameter is not provided explicitly.

4.5.4 Opening an existing Project

`openProject(Object, Closure)` opens the project at the given .dpa file location, delegates the given code to the opened `IProject` and closes the project. The `Object` given as .dpa file is converted to `Path` using `Converters.TO_SCRIPT_PATH` 4.11.2 on page 99

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

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

Listing 4.43: Opening a project from .dpa file

`parameterizeProjectLoad(Closure)` returns a handle on the project specified by the given `Closure`. Using the `IOpenProjectApi`, the `Closure` may further customize the project's open-

ing procedure. The project is not opened until `IProjectRef.openProject(Closure)` is called on the `IProjectRef` returned by this method.

```
scriptTask('taskName', DV_APPLICATION) {
  code {
    def project = projects.parameterizeProjectLoad {
      // replace getDpaFileToLoad() with the path to the .dpa file to be loaded
      dpaFile getDpaFileToLoad()
      // 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 4.44: Parameterizing the project open procedure

`IOpenProjectApi` provides means for parameterizing the process of opening a project.

DPA File Using `setDpaFile(Object)` the .dpa file of the project to be opened can be specified. The value given here is converted to `Path` using `Converters.TO_SCRIPT_PATH` 4.11.2 on page 99.

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

Validation Using `setEnableValidation(boolean)` is can be specified whether or not to activate validation for the opened project.

4.5.4.1 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)` 4.5.3 on page 45
- `IProjectHandlingApi.parameterizeProjectLoad(Closure)` 4.5.4 on the previous page

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

4.5.5 Saving a Project

`IProject.saveProject()` saves the current state 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(sipDefRef.EcuC)
      }

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

Listing 4.45: Opening, modifying and saving a project

4.6 Model API

4.6.1 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 5 on page 106 may additionally be useful to understand detailed concepts and as a reference to handle special use cases.

4.6.2 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 base on the AUTOSAR MetaModel. In 5.1 on page 106 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.

4.6.2.1 Read the ActiveEcuc

This section provides some typical examples as a brief introduction for reading the Ecuc by means of the BswmdModel. See chapter 4.6.3.2 on page 58 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 {
        // Get the module DefRef searching all definitions of this SIP
        def moduleDefRef = sipDefRef.EcuC

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

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

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

Listing 4.46: 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 {
        // Get all instances with this definition
        List<EcuC> ecucModules = bswmdModelRead(EcuC.DefRef)

        // Get the EcucGeneral container of the first found module instance
        EcuC ecuc = ecucModules.single
        EcucGeneral ecucGeneral = ecuc.ecucGeneral

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

Listing 4.47: Read with BswmdModel objects starting with a module DefRef (strong typing)

The `bswmdModelRead()` 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 {
        // Execute the closure with all instances of this definition
        bswmdModelRead(EcuC.DefRef) {
            // The related BswmdModel instance is parameter of this closure
            ecuc ->

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

Listing 4.48: 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 {
        // Get the MDF model instance of the Ecuc General container
        def container = mdfRead(EcucGeneral.DefRef).single

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

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

Listing 4.49: Read with BswmdModel object for an MDF model object

4.6.2.2 Write the ActiveEcuc

In Release 16 (Cfg5.13) the BswmdModel does not yet support write access to the Ecuc model. Please use the MDF model (with `mdfWrite()`) instead.

4.6.2.3 Read the SystemDescription

This section contains only one example for reading the SystemDescription by means of the MDF model. See chapter 4.6.4.1 on page 60 for more details.

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

scriptTask("mdfRead", 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
        mdfRead(asrPath) {
            // Parameter type is MIVariableDataPrototype:
            prototype ->

            // Traverse down to the swDataDefProps
            prototype.swDataDefProps {
                // Parameter type is MISwDataDefProps:
                swDataDefPropsParam ->

                // swDataDefPropsVariant is a List<MISwDataDefPropsConditional>
                // Execute the following for ALL elements of this List
                swDataDefPropsParam.swDataDefPropsVariant {
                    // Parameter type is 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 4.50: Read system description starting with an AUTOSAR path

4.6.2.4 Write the SystemDescription

Writing the system description looks quite similar but uses `mdfWrite()` instead. See chapter 4.6.4.2 on page 62 for more details.

4.6.3 BswmdModel in AutomationInterface

The AutomationInterface contains a generated BswmdModel. The BswmdModel provides classes for all Ecuc elements of the AUTOSAR model (ModuleConfigurations, Containers, Pa-

rameter, 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 4.6.3.2.

For a detailed description of the BswmdModel, see chapter 5.3.1 on page 120.

4.6.3.1 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 4.51: BswmdModel usage with import

4.6.3.2 Reading with BswmdModel

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

The following types of entry points are provided here:

- `bswmdModelRead(WrappedTypedDefRef)` searches all objects with the specified definition and returns the BswmdModel instances
- `bswmdModelRead(MIHasDefinition, WrappedTypedDefRef)` returns the BswmdModel instance for the provided MDF model instance.

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

The `bswmdModelRead()` 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.

For usage sample please see chapter 4.6.2.1 on page 54.

4.6.3.3 Writing with BswmdModel

In Release 16 (Cfg5.13) the BswmdModel does not yet support write access to the Ecuc model. Please use the MDF model (with `mdfWrite()`) instead.

4.6.3.4 Sip 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 4.6.3.5 for details.

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

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

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

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

Listing 4.52: Usage of the sipDefRef API to retrieve DefRefs in script files

4.6.3.5 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 4.6.3.1 on page 58 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 4.53: Usage of generated DefRefs form the bswmd model

4.6.4 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 5.1 on page 106.

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

4.6.4.1 mdfRead

The `mdfRead()` 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:

- `mdfRead(TypedAsrPath)` searches an object with the specified AUTOSAR path

- `mdfRead(TypedDefRef)` searches all objects with the specified definition

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

```
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
    mdfRead(asrPath) {
        // Parameter type is MIVariableDataPrototype:
        param ->

        // Traverse down to the swDataDefProps
        param.swDataDefProps {
            println "Do something ..."
        }
    }
}
```

Listing 4.54: Navigate into an MDF object starting with an AUTOSAR path

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

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 = mdfRead(asrPath)      // Type MIVariableDataPrototype
def props = obj.swDataDefProps  // Type MISwDataDefProps

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

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

Listing 4.55: Find an MDF object and retrieve some content data

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

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

Listing 4.56: 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:

```
mdfRead(asrPath) {
    int count = 0
    assert adminData == null
    adminData {
        count++
    }
    assert count == 0
}
```

Listing 4.57: Ignoring non-existing member closures

4.6.4.2 mdfWrite

The mdfWrite 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:

- `mdfWrite(TypedAsrPath)` searches an object with the specified AUTOSAR path
- `mdfWrite(TypedDefRef)` searches all objects with the specified definition

This behavior is similar to mdfRead so far. But there are two differences:

1. mdfWrite opens a transaction if there is no transaction opened yet (see chapter 4.6.5 on page 70 for details about transactions)
2. When a member object doesn't exist during navigation into nested closures, it is being created automatically

Non existing members are created automatically when navigating into an MDF model tree with mdfWrite():

```
mdfWrite(asrPath) {
    int count = 0
    assert adminData == null
    adminData {
        count++
    }
    assert count == 1
    assert adminData != null
}
```

Listing 4.58: Creating non-existing member by navigating into its content

Creating and adding child list members For child list members, the automation API provides overloaded add() methods for convenient child object creation.

```
mdfWrite(asrPath) {
    assert adminData.sdg.empty
    adminData {
        sdg.add(MISdg) {
            gid = "NewGidValue"
        }
    }
    assert adminData.sdg.first.gid == "NewGidValue"
}
```

Listing 4.59: Creating new members of child lists 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.

- The method `add(Class)` creates a new MDF object of the specified type and appends it to this list. The new object is finally returned.
- The method `add(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 `add(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 `add(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.
- The method `add(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 `add(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 `add(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 `add(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.

- The method `add(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 `add(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 `add(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 `add(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 `add(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 `add(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 `add(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 `add(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.

4.6.4.3 Deleting model objects

The method `moRemove(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 `IModelOperationsPublished.deleteModuleConfiguration(MIModuleConfiguration)` to deactivate the module correctly.

```
// MIParameterValue param = ...

transaction {
    assert !param.moIsRemoved()
    param.moRemove()
    assert param.moIsRemoved()
}
```

Listing 4.60: Delete a parameter instance

For details about model object deletion and access to deleted objects, read section 5.1.7.4 on page 111 ff.

4.6.4.4 Special properties and extensions

childByName The `childByName(MIARObject, String)` method returns the child `MIReferrable` below the specified object which has this relative AUTOSAR path (not starting with '/').

```
MIContainer canGeneral = ...
MIContainer mainFunctionRWPeriods
    = canGeneral.childByName("CanMainFunctionRWPeriods")
```

Listing 4.61: Get a MIReferrable child object by name

asrPath The `getAsrPath(MIReferrable)` method returns the AUTOSAR path of the specified object.

```
MIContainer canGeneral = ...
AsrPath path = canGeneral.asrPath
```

Listing 4.62: Get the AsrPath of an MIReferrable instance

See chapter 5.4.1 on page 128 for more details about AsrPaths.

asrObjectLink The `getAsrObjectLink(MIARObject)` method returns the `AsrObjectLink` of the specified object.

```
MIParameterValue param = ...
AsrObjectLink link = param.asrObjectLink
```

Listing 4.63: Get the AsrObjectLink of an AUTOSAR model instance

See chapter 5.4.2 on page 128 for more details about AsrObjectLinks.

defRef The `getDefRef(MIHasDefinition)` method returns the `DefRef` of the specified object.

```
MIParameterValue param = ...
DefRef defRef = param.defRef
```

Listing 4.64: Get the DefRef of an Ecuc model instance

The `setDefRef(MIParameterValue, DefRef)` method sets the definition of this parameter to the specified defRef.

```
MIParameterValue param = ...
DefRef newDefinition = ...
param.defRef = newDefinition
```

Listing 4.65: 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 is 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 `setDefRef(MIContainer, DefRef)` method sets the definition of this container to the specified defRef. For details see `setDefRef(MIParameterValue, DefRef)`.

See chapter 5.4.3 on page 129 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 4.66: Get the CeState of an Ecuc parameter instance

See chapter 5.4.4 on page 132 for more details about the CeState.

moIsRemoved The `getMoIsRemoved(MIObject)` method returns `true` if the specified object has been removed (deleted) from the MDF model.

```
MIObject obj = ...
if (!obj.moIsRemoved()) {
    work with obj ...
}
```

Listing 4.67: Check is a model instance is deleted

See chapter 5.1.7.4 on page 111 for more details about deleting model objects and access to them.

4.6.4.5 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 4.68: Get the AUTOSAR root object

4.6.4.6 ActiveEcuC

The activeEcuc access methods provide access to the module configurations of the Ecuc model as follows.


```
// Get the modules as Collection<MIModuleConfiguration>
Collection modules = activeEcuc.allModules
```

Listing 4.69: Get the active Ecuc and all module configurations

```
// Iterate over all module configurations
activeEcuc {
    int count = 0
    allModules.each {
        count++
    }
    assert count > 1
}
```

Listing 4.70: 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 4.71: Get module configurations by definition

4.6.4.7 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 4.72: Get sub-containers and parameters by definition

4.6.4.8 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

- `hasValue(MIParameterValue)`
  returns `true` if the parameter (or reference) has a value.
- `containsBoolean(MINumericalValue)`
  returns `true` if the parameter value contains a valid boolean with the same semantic as `IModelAccess.containsBoolean(MINumericalValue)`.

  Call this method in advance
  to guarantee that `getAsBoolean(MINumericalValueVariationPoint)` doesn't lead to errors.
- `containsInteger(MINumericalValue)`
  returns `true` if the parameter value contains a valid integer with the same semantic as `IModelAccess.containsInteger(MINumericalValue)`.

  Call this method in advance
  to guarantee that `getAsInteger(MINumericalValueVariationPoint)` doesn't lead to errors.
- `containsDouble(MINumericalValue)`
  returns `true` if the parameter value contains a valid double (AUTOSAR float) with the same semantic as
  `IModelAccess.containsFloat(MINumericalValue)`.

  Call this method in advance
  to guarantee that `getAsDouble(MINumericalValueVariationPoint)` 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 4.73: Check parameter values

Parameters

- `getAsLong(MINumericalValueVariationPoint)` 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`.
- `getAsInteger(MINumericalValueVariationPoint)` 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`.

- `getAsBigInteger(MINumericalValueVariationPoint)` returns the value as `BigInteger`.

  Throws `NumberFormatException` if the value string doesn't represent an integer value.

- `getAsDouble(MINumericalValueVariationPoint)` returns the value as `Double`.

  Throws `NumberFormatException` if the value string doesn't represent a float value.

- `getAsBigDecimal(MINumericalValueVariationPoint)` returns the value as `BigDecimal`.

  Note: This method will possibly return `MBigDecimal.POSITIVE_INFINITY`, `MBigDecimal.NEGATIVE_INFINITY` or `MBigDecimal.NaN`.

  If it is necessary to do computations with these special numbers,
  use `getAsDouble(MINumericalValueVariationPoint)` instead.

  Throws `NumberFormatException` if the value string doesn't represent a float value.

- `getAsBoolean(MINumericalValueVariationPoint)` returns the value as `Boolean`.

  Throws `NumberFormatException` if the value string doesn't represent a boolean value.

- `asCustomEnum(MITextualValue, Class)` returns the value of the enum parameter as a custom enum literal. If the Class destClass implements the `IModelEnum` interface, the literals are mapped via these information form the `IModelEnum` interface. Read the JavaDoc of `IModelEnum` for more details.

```
// 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 4.74: Get integer parameter value

References

- `getAsAsrPath(MIARRef)` returns the reference value as AUTOSAR path.
- `getAsAsrPath(MIReferenceValue)` returns the reference parameters value as AUTOSAR path.
- `getRefTarget(MIReferenceValue)` 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 4.75: Get reference parameter value

4.6.5 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 5.1.7 on page 110.

```
scriptTask("TaskName", DV_PROJECT){
    code {
        transaction{
            // Your transaction code here
        }
    }
}
```

Listing 4.76: Execute a transaction

```
scriptTask("TaskName", DV_PROJECT){
    code {
        transaction("Transaction name") {

            // The transactionName proptery is available inside a transaction
            String name = transactionName

        }
    }
}
```

Listing 4.77: Execute a transaction with a name

The transaction name has no additional semantic. It is only be used for logging and to improve error messages.

4.6.5.1 Nested transactions

Be aware that nested transactions are not supported. Opening a transaction when there is already a transaction running, leads to a `TransactionException`.

4.6.5.2 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.

```
scriptTask("TaskName", DV_PROJECT){
    code {
        transaction("TransactionName") {
            // Your transaction code here
        }

        assert transactionHistory.canUndo()

        transactionHistory.undo()

        assert !transactionHistory.canUndo()
    }
}
```

Listing 4.78: Undo a transaction with the transactionHistory

```
scriptTask("TaskName", DV_PROJECT){
    code {
        transaction("TransactionName") {
            // Your transaction code here
        }
        transactionHistory.undo()

        assert transactionHistory.canRedo()

        transactionHistory.redo()

        assert !transactionHistory.canRedo()
    }
}
```

Listing 4.79: Redo a transaction with the transactionHistory

4.6.5.3 Operations

The model operations implement convenient means to execute complex model changes like AUTOSAR module activation or cloning complete model sub-trees.

- 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
- The method `changeBswImplementation(MIModuleConfiguration, MIBswImplementation)` changes the BSW-implementation of a module configuration including the definition of all contained containers and parameters.
- 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

4.6.6 Post-build selectable variance

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.

For details about post-build selectable variance and model views read 5.2 on page 112.

4.6.6.1 Investigate project variance

The projects variance can be analyzed using the `variance` keyword. These methods can be called then:

- The method `hasPostBuildVariance()` returns `true` if the active project contains post-build variants.
- The method `getInvariantValuesView()` returns the invariant values view.
- The method `getInvariantEcucDefView()` returns the invariant Ecuc definition view.
- The method `getCurrentlyActiveView()` returns the currently active model view.
- The method `getAllVariantViews()` returns all available variant model views (one `IPredefinedVariantView` per predefined varient).


```
scriptTask("TaskName", DV_PROJECT){
    code{
        def activeView1 = variance.currentlyActiveView
        assert activeView1 instanceof IInvariantValuesView

        // ... or with a closure
        variance {
                def activeView2 = currentlyActiveView
                assert activeView1 == activeView2
                assert activeView1 == invariantValuesView

                // Get number of variants
                int num = allVariantViews.size()
                assert num == 4
        }
    }
}
```

Listing 4.80: The default view is the IInvariantValuesView

4.6.6.2 Variant model objects

The following model object extensions provide convenient means to investigate model object variance in detail.

- The method `activeWith(IModelView, Callable)` executes code under visibility of the specified model view.
- The method `isModelInvariant(MIObject)` 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 views.
- The method `isValueInvariant(MIObject)` returns `true` if the object has the same value in all variants.

  For details about invariant views see 5.2.1.4 on page 115.
- The method `isEcucDefInvariant(MIObject)` returns `true` if the object is invariant according to its EcuC definition.

  See `IInvariantEcucDefView` for more details to the concept.
- The method `isVisible(MIObject)` returns `true` if the object is visible in the current model view.
- The method `isVisibleInModelView(MIObject, IModelView)` returns `true` if the object is visible in the specified model view.
- The method `isNeverVisible(MIObject)` returns `true` if the object is *invisible* in all variant views.
- The method `getVisibleVariantViews(MIObject)` returns all variant views the specified object is visible in.
- The method `getVisibleVariantViewsOrInvariant(MIObject)` For semantic details see `IModelViewManager.getVisibleVariantViewsOrInvariant(MIObject)`.
- The method `asViewedModelObject(MIObject)` returns a new `IViewedModelObject` instance using the currently active view.

- The method `getVariantSiblings(MIObject)` returns MDF object instances representing the same object but in all variants.

  For details about the sibling semantic see 5.2.1.3 on page 114.

- The method `getVariantSiblingsWithoutMyself(MIObject)` returns the same collection as `getVariantSiblings(MIObject)` but without the specified object.

```
// IPredefinedVariantView
```

4.7 Generation API

The following chapter describes the module generation automation API.

The block **generation** encapsulates all settings and commands which are related to this use case.

The basic structure is the following:

```
generation{
    settings{
        // Settings like the selection of generators for execution can be done
            here
        externalGenerationSteps{
            // Settings related to externalGenerationSteps can be done here
        }
    }
    // The execution of the generation or validation can be started here
}
```

Listing 4.82: Basic structure

4.7.1 Settings

This class encapsulates all settings which belong to a generation process.
E.g.

- Select the generators to execute
- Select the target type
- 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.

4.7.1.1 Default Project Settings

The following snippet executes a validation with the default project settings.

```
scriptTask("validate_with_default_settings"){
    code{
        generation{
            validate()
        }
    }
}
```

Listing 4.83: 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 4.84: Generate with standard project settings

4.7.1.2 Generate One Module

This sample selects one specific module and starts the generation. There are two ways to open an settings block:

- **settings**
  - This keyword creates empty settings. E.g. no module is selected for execution.
- **settingsFromProject**
  - This keyword takes the project settings as template. E.g. modules from the project settings are initially activated and can optionally be refined by explicit selections.

```
scriptTask("generate_one_module"){
    code{
        generation{
            settings{
            // To take the project settings as template use
            // settingsFromProject{
                selectGeneratorsByDefRef("/MICROSAR/Aaa")
            }
            generate()
        }
    }
}
```

Listing 4.85: Generate one module

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 4.86: Generate one module

4.7.1.3 Generate 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 4.87: Generate two modules

4.7.1.4 Generate 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 4.88: Generate one module with two configurations

4.7.1.5 Generate External Generation Step

Besides the internal generators, which are covered by the topics above, there are also external 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 4.89: Execute an external generation step

4.7.2 Generation Task Types

There are three types of `IScriptTaskTypes`for the generation process:

- Generation Step: DV_GENERATION_STEP
- Generation Process Start: DV_GENERATION_ON_START
- Generation Process End: DV_GENERATION_ON_END

The general description of the type is in chapter 4.3.1.4 on page 26. The following code samples show the usage of these task types:

Sample for *Generation Step*:

```
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 4.90: Use a script task as generation step during generation

The *Generation Step* can also report validation-results into the passed `resultSink`. See chapter 4.8.5.10 on page 90 for a sample how to create an validation-result and report it.

Sample for *Generation Process Start*:

```
scriptTask("GenStartTask", DV_GENERATION_ON_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 4.91: Hook into the GenerationProcess at the start with script task

Sample for *Generation Process End*:

```
scriptTask("GenEndTask", DV_GENERATION_ON_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 4.92: Hook into the GenerationProcess at the end with script task

4.8 Validation API

4.8.1 Introduction

All examples in this chapter are based on the situation of the figure 4.6. The module and the validators are not from the real MICROSAR stack, but just for the examples. 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.
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 letter 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.

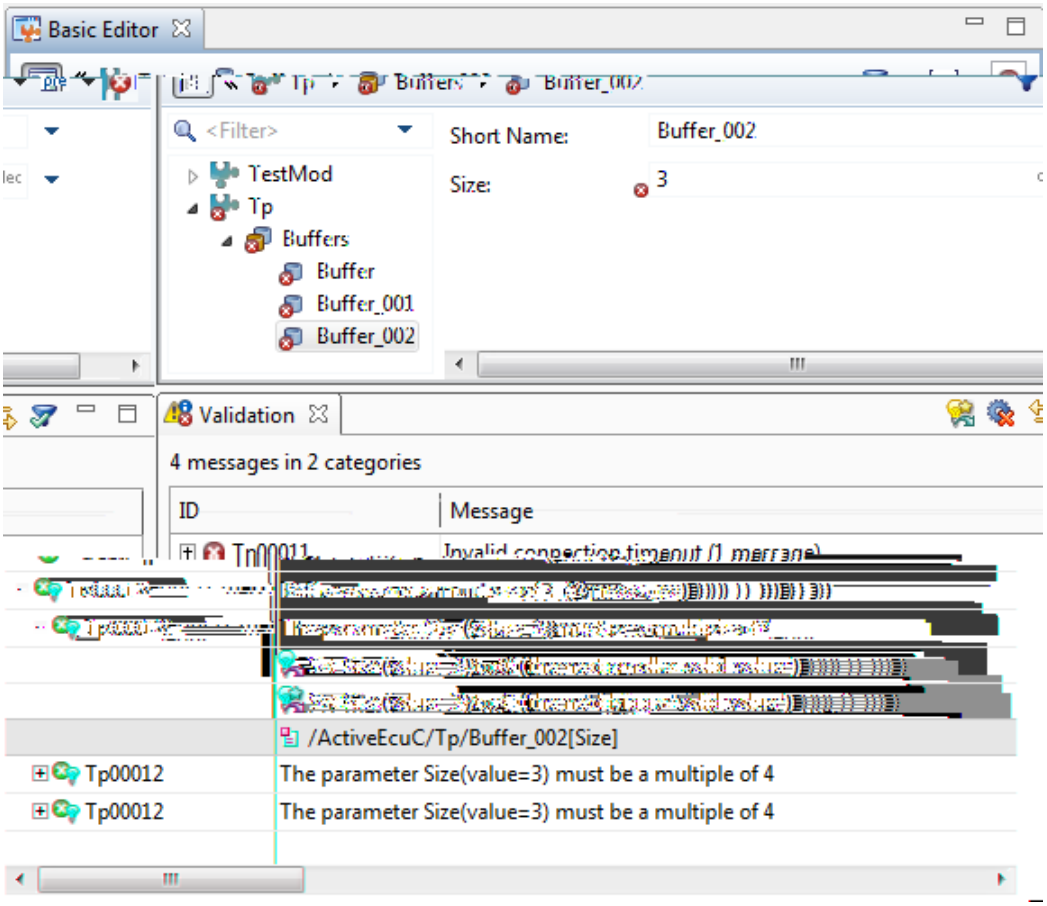


Figure 4.6: example situation with the GUI

4.8.2 Access Validation-Results

A `validation{}` block gives access to the validation API of the consistency component. That means accessing the validation-results which are shown in the GUI in the validation view, and solving them by executing solving-actions which are also shown in the GUI beneath each validation-result (with a bulb icon).

`getValidationResults()` waits for background-validation-idle and returns all validation-results

of any kind.

```
scriptTask("CheckValidationResults_filterByOriginId"){
    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 direct access to validation-results without a validation
             block
        assert validation.validationResults.size() > 3
    }
}
```

Listing 4.93: Access all validation-results and filter them by ID

4.8.3 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 `ISolver.solve(Closure)` which is the most preferred way of solving validation-results with solving-actions. See 4.8.4.1 on the next page.

4.8.4 Solve Validation-Results with Solving-Actions

A single validation-result can be solved by calling `solve()` on one of its solving-actions.


```
scriptTask("SolveSingleResultWithSolvingAction"){
    code{
        validation{
            def tp12Results = validationResults.filter{it.isId("Tp", 12)}
            assert tp12Results.size() == 3

            // Take first 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 4.94: Solve a single validation-result with a particular solving-action

4.8.4.1 Solver API

`getSolver()` gives access to the `ISolver` API, which has advanced methods for bulk solutions.

`ISolver.solve(Closure)` 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(Closure)` 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(Closure)` waits for background-validation-idle in order to have reproducible results.

```
scriptTask("SolveMultipleResults"){
    code{
        validation{
            assert validationResults.size() == 4

            solver.solve{
                result{isId("Tp", 12)}
                    .withAction{containsString("next bigger valid value")}
                // Call result() and pass a closure that works as filter based
                    on methods of IValidationResultUI.
                // On the returned, call withAction() and pass a closure that
                    selects a solving-action based on methods of
                    IValidationResultForSolvingActionSelect

                // multiple such calls can be placed in one solve() call.
                result{isId("Com", 34)}.withAction{containsString("recalculate"
                    )}
            }

            assert validationResults.size() == 1
            // Three Tp12 and zero Com34 (didn't exist) results solved. One
                other left
        }
    }
}
```

Listing 4.95: Fast solve multiple results within one transaction

Solve all PreferredSolvingActions `ISolver.solveAllWithPreferredSolvingAction()` solves all validation-results with its preferred solving- action (`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"){
    code{
        validation{
            assert validationResults.size() == 4

            solver.solveAllWithPreferredSolvingAction()

            assert validationResults.size() == 1

            // this would do the same
            transactionHistory.undo()
            assert validationResults.size() == 4

            solver.solve{
                result{true}.withAction{preferred}
            }

            assert validationResults.size() == 1
        }
    }
}
```

Listing 4.96: Solve all validation-results with its preferred solving-action (if available)

4.8.5 Advanced Topics

4.8.5.1 Access Validation-Results of a Model-Object

`MIObject.getValidationResults()` returns the validation-results of an `MIObject`. These are those results for which `IValidationResultUI.matchErroneousCE(MIObject)` returns true.

```
scriptTask("CheckValidationResultsOfObject"){
    code{
        // sampleDefRefs contains DefRef constants just for this example.
           Please use the real DefRefs from your SIP
        def buffer002 = mdfRead(AsrPath.create("/ActiveEcuC/Tp/Buffer_002")) //
            a Buffer container
        def sizeParam = buffer002.parameter(sampleDefRefs.tpBufferSizeDefRef).
           single // the Size parameter
        assert sizeParam.validationResults.size() == 1
        assert buffer002.validationResults.size() == 0 // the validation-result
            exists for the Size parameter, not for the Buffer container
    }
}
```

Listing 4.97: Access all validation-results of a particular object

4.8.5.2 Access Validation-Results of a BSWMD Definition

`DefRef.getValidationResults()` returns the validation-results for which `IValidationResultUI.matchErroneousCE(MIHasDefinition, DefRef)` returns true.
The context project for this call is the active project, see `ScriptApi.getActiveProject()`.

```
scriptTask("CheckValidationResultsOfDefRef"){
    code{
        // sampleDefRefs contains DefRef constants just for this example.
           Please use the real DefRefs from your SIP
        assert sampleDefRefs.tpBufferSizeDefRef.validationResults.size() == 3
    }
}
```

Listing 4.98: Access all validation-results of a particular DefRef

4.8.5.3 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"){
    code{
        validation{
            def tp12Const = ["Tp",12]

            assert validationResults.size() > 3
            assert validationResults.filter{it.isId(*tp12Const)}.size() == 3
        }
    }
}
```

Listing 4.99: Filter validation-results using an ID constant

4.8.5.4 Identification of a Particular Solving-Action

A so called solving-action-group-ID identifies a solving-action uniquely within one validation-result. In other words, two solving-actions, which do semantically the same, from two validation-results of the same result-ID (origin + number), belong to the same solving-action-group. This semantical group may have an optional solving-action-group-ID, that can be used for solving-action identification within one validation-result.

Keep in mind that the solving-action-group-ID is only unique within one validation-result-ID, and that the group-ID assignment is optional for a validator implementation.

In order to find out the solving-action-group-IDs, press CTRL+SHIFT+F9 with a selected validation-result to copy detailed information about that result including solving-action-group-IDs (if assigned) to the clipboard.

If group-IDs are assigned, it is much safer to use these for solving-action identification than description-text matching, because a description-text may change.

```
final int SA_GROUP_ID__TP12__NEXT_BIGGER_VALID_VALUE = 2

scriptTask("SolveMultipleResultsByGroupId"){
    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 4.100: Fast solve multiple validation-results within one transaction using a solving-action-group-ID

4.8.5.5 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.

4.8.5.6 Further IValidationResultUI Methods

The following listing gives an overview of other "properties" of an IValidatonResultUI.

```
scriptTask("IValidationResultUIApiOverview"){
    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(2).description.contains("next
                bigger valid value")
            assert r.preferredSolvingAction == r.getSolvingActionByGroupId(2)
                // this result has a preferred-solving-action
            assert r.acknowledgement.isPresent() == false // results with lower
                 severity than ERROR can be acknowledged in the GUI
            assert r.cause.isPresent() == false // if the cause was an
                exception, r.cause.get() returns it
            assert r.isReducedSeverity() == false // an ERROR result gets
                reduced to WARNING if one of its erroneous CEs is user-defined (
                user-overridden)
            assert r.isOnDemandResult() == false // on-demand results are
                visualized with a gear-wheel icon
        }
    }
}
```

Listing 4.101: IValidationResultUI overview

4.8.5.7 IValidationResultUI in a variant (Post Build Selectable) Project

```
scriptTask("IValidationResultUIInAVariantProject"){
    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 4.102: IValidationResultUI in a variant (post build selectable) project

4.8.5.8 Erroneous CEs of a Validation-Result

`IValidationResultUI.getErroneousCEs()` returns a collection of `IDescriptor`, each describing a CE that gets an error annotation in the GUI.

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"){
    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.equals(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 4.103: Erroneous CEs of an IValidationResultUI

4.8.5.9 Examine Solving-Action Execution

The easiest and most reliable option for verifying solving-action execution is to check the presence of validation-results afterwards.

This is also the feedback strategy of the GUI. After multiple solving-actions have been solved, the GUI does not show the execution result of each individual solving-action, but just the remaining validation-results after the operation. Only if a single solving-action is to be solved, and that fails, the GUI shows the message of that failure including the reason.

The following describes further 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(Closure)` 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"){
    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)
            // Cfg5 takes multiple attempts to solve a result because other
                changes may eliminate a blocking reason, but stops after an
                execution limit is reached.
        }
    }
}
```

Listing 4.104: Examine an ISolvingActionSummaryResult

4.8.5.10 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 The `IValidationResultSink` must be obtained by the context and is not provided by the creation API. E.g. some script tasks pass an `IValidationResultSink` as argument (like DV_GENERATION_STEP). If you don't get a sink from the script task, you can not report any `IValidationResults`.

```
scriptTask("ScriptTaskCreationResult"){
  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 are error objects to mark them
        builder.addErrorObject(sipDefRef.EcucGeneral.bswmdModelRead().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
        resultSink.reportValidationResult(valResult)
      }
    }
  }
}
```

Listing 4.105: Create a ValidationResult

4.9 Update Workflow

The Update Workflow derives the initial EcuC from the input files and updates the project

4.9.3 Example: List of Input Files shall be changed

```
scriptTask("ChangeListOfComExtractsAndUpdate", DV_APPLICATION) {
code {
    def extractPath = paths.resolvePath(extractFile)
    workflow {
        update(dpaProjectFile){
            input{
                communication{
                   extract{
                        extractFiles{exFilePathList->
                            // clear the list of communication extracts
                            exFilePathList.clear()
                            // adds an communication extract
                            exFilePathList.add(extractPath)
                        }

                        // change the selection of the ecuInstance
                        // Note: this closure is deferred executed.
                        ecuInstanceSelection{
                            return availableEcuInstances[0]
                        }
                    }
                }
            }
        }
    }
}
}
```

Listing 4.107: Change list of communication extracts and update

Note: The code in the `ecuInstanceSelection` closure is deferred executed. The access to variables, declared outside of this closure is not allowed.

This example shows the complete replacement of the current list of communication extracts with one extract and the selection of the first ecuInstance in the new extract. The update workflow is executed after the update closure block is left.

4.9.4 Prerequisites

The Update Workflow API can not be used while the Project to update is opened. E.g. in a `IProjectRef.openProject` closure block or in a ScriptTask with the `DV_PROJECT` ScriptTask-Type.

4.10 Domain APIs

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 4.10.1.

`getDomain()` allows accessing the `IDomainApi` like a property.

```
scriptTask('taskName') {
  code {
    // IDomainApi is available as "domain" property
    def domainApi = domain
  }
}
```

Listing 4.108: Accessing IDomainApi as a property

`domain(Closure)` allows accessing the `IDomainApi` in a scope-like way.

```
scriptTask('taskName') {
  code {
    domain {
      // IDomainApi is available inside this Closure
    }
  }
}
```

Listing 4.109: Accessing IDomainApi in a scope-like way

4.10.1 Communication Domain

The communication domain API is specifically designed to support communication related use cases. It is available from the `IDomainApi` 4.10 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 4.110: Accessing ICommunicationApi as a property

`communication(Closure)` allows accessing the `ICommunicationApi` in a scope-like way.

```
scriptTask('taskName') {
  code {
    domain.communication {
      // ICommunicationApi is available inside this Closure
    }
  }
}
```

Listing 4.111: 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 4.10.1.1 on the next page.

4.10.1.1 CanControllers

An `ICanController` instance represents a CanController `MIContainer` providing support for use cases exceeding those supported by the model API.

```
scriptTask('taskName', DV_APPLICATION) {
    code {
        // replace $dpaFile with the path to your project
        def theProject = projects.openProject("$dpaFile") {
            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.name.contains 'CH0' }.single.
                        openAcceptanceFilters()

                    // accessing a dedicated CanController
                    def ch0 = canControllers.filter { it.name.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 4.112: Optimizing Can Acceptance Filters

Opening Acceptance Filters `openAcceptanceFilters()` opens all of this `ICanController`'s acceptance filters.

Optimizing Acceptance Filters `optimizeFilters(Closure)` 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`.

4.10.1.2 CanFilterMasks

An `ICanFilterMask` instance represents a CanFilterMask `MIContainer` providing support for use cases exceeding those supported by the model API.

For example code see 4.10.1.1 on the preceding 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`.

4.11 Utilities

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

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_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_DPA_FILE Ensures that the given `Path` points to a DaVinci project file (.dpa file).

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

IS_SYSTEM_DESCRIPTION_FILE Ensures that the given `Path` points to a system description input file (.arxml, .dbc, .ldf, .xml or .vsde file).

IS_COMPATIBLE_DA_VINCI_DEV_EXECUTABLE Ensures that the given `Path` points to a compatible DaVinci Developer executable (DaVinciDEV.exe).

4.11.2 Converters

`Converters` provides general purpose `Functions` performing value conversions used throughout the automation interface. 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:

TO_PATH Attempts to convert arbitrary `Objects` to `Paths` using `IAutomationPathsApi.resolvePath(Object)` 4.4.3.2 on page 31.

TO_SCRIPT_PATH Attempts to convert arbitrary `Objects` to `Paths` using `IAutomationPathsApi.resolveScriptPath(Object)` 4.4.3.3 on page 32.

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()`.

For thrown `Exceptions` see the used functions described above.

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 `getMdfModelElement()` method.
- `IViewedModelObjects` are converted using their `getMdfObject()` method.
- For all other `Objects` `ClassCastExceptions` are thrown.

For thrown `Exceptions` see the used functions described above.

4.12 Advanced Topics

This chapter contains advanced use cases and classes for special tasks. For a normal script these items are not relevant.

4.12.1 Java Development

It is also possible to write automation scripts is 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.

4.12.1.1 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(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 4.113: 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.

4.12.1.2 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 4.114: 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 4.115: 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]

4.12.1.3 Java Code in dvgroovy Scripts

It is not possible to write Java classes when using the `.dvgroovy` script file. You have to create an automation script project, see chapter 7 on page 137.

[1]Last accessed 2016-05-24

4.12.2 Unit testing API

The Automation Interface provides an connector to execute unit tests as script task. This is helpful, if you want to write tests for:

- Generators
- Validations
- Workflow rules
- ...

Normally a script task executes it's code block, but the unit test task will execute all contained unit tests instead.

4.12.2.1 JUnit4 Integration

The AutomationInterface can execute JUnit 4 test cases and test suites.

Execution of JUnit Test Classes A simple unit test class will look like:

```
import org.junit.Test;

public class ScriptJUnitTest {

    @Test
    public void testYourLogic() {
        // Write your test code here
    }
```

Listing 4.116: Run all JUnit tests from one class

You can access the Automation API with the `ScriptApi` class. See chapter 4.4.7 on page 39 for details.

Execution of multiple Tests with JUnit Suite To execute multiple tests you have to group the tests into a test suite.

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({
        // Two test classes
        ScriptJUnitTest.class,
        ScriptSpockTest.class,

        // Another JUnit suite
        InnerSuiteScriptJUnitTests.class,
})
public class AllMyScriptJUnitTests {
```

Listing 4.117: Run all JUnit tests using a Suite

You can also group test suites in test suites and so on.

4.12.2.2 Execution of Spock Tests

The AutomationInterface can also execute Spock tests. See:

- Homepage: https://github.com/spockframework/spock[2]
- Documentation: http://spockframework.github.io/spock/docs/1.0/index.html[3]

It is also possible to group multiple Spock test into a JUnit Test Suite.

Usage sample:

```
import spock.lang.Specification

class ScriptSpockTest extends Specification {

    def "Simple Spock test"(){
        when:
        //Add your test logic here
        def myExpectedString = "Expected"

        then:
        myExpectedString == "Expected"
    }
}
```

Listing 4.118: Run unit test with the Spock framework

You can access the Automation API with the `ScriptApi` class. See chapter 4.4.7 on page 39 for details.

You have to add a Spock dependency in your `build.gradle` file:

```
dependencies {
```

dep51l S q1 0 0 1compile4920fg0."3rg[]spod 0 G00e398 w(0909n 3f3824099150qTd0[(ScriptApi)]20/F185 1CGPT9I 4f

```
project.ext.automationClasses = [
        "sample.MyScript",
        "sample.MyUnitTestSuite", // This is a test suite
        // Add here your test or suite class with full qualified name
]
```

Listing 4.120: The projectConfig.gradle file content for unit tests

5 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 4.6 on page 54. So, whenever possible use the Groovy API and read this chapter only to get background information when required.

5.1 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.

5.1.1 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`.

5.1.2 The models inheritance hiearchy

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 5.1 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`

5.1.2.1 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

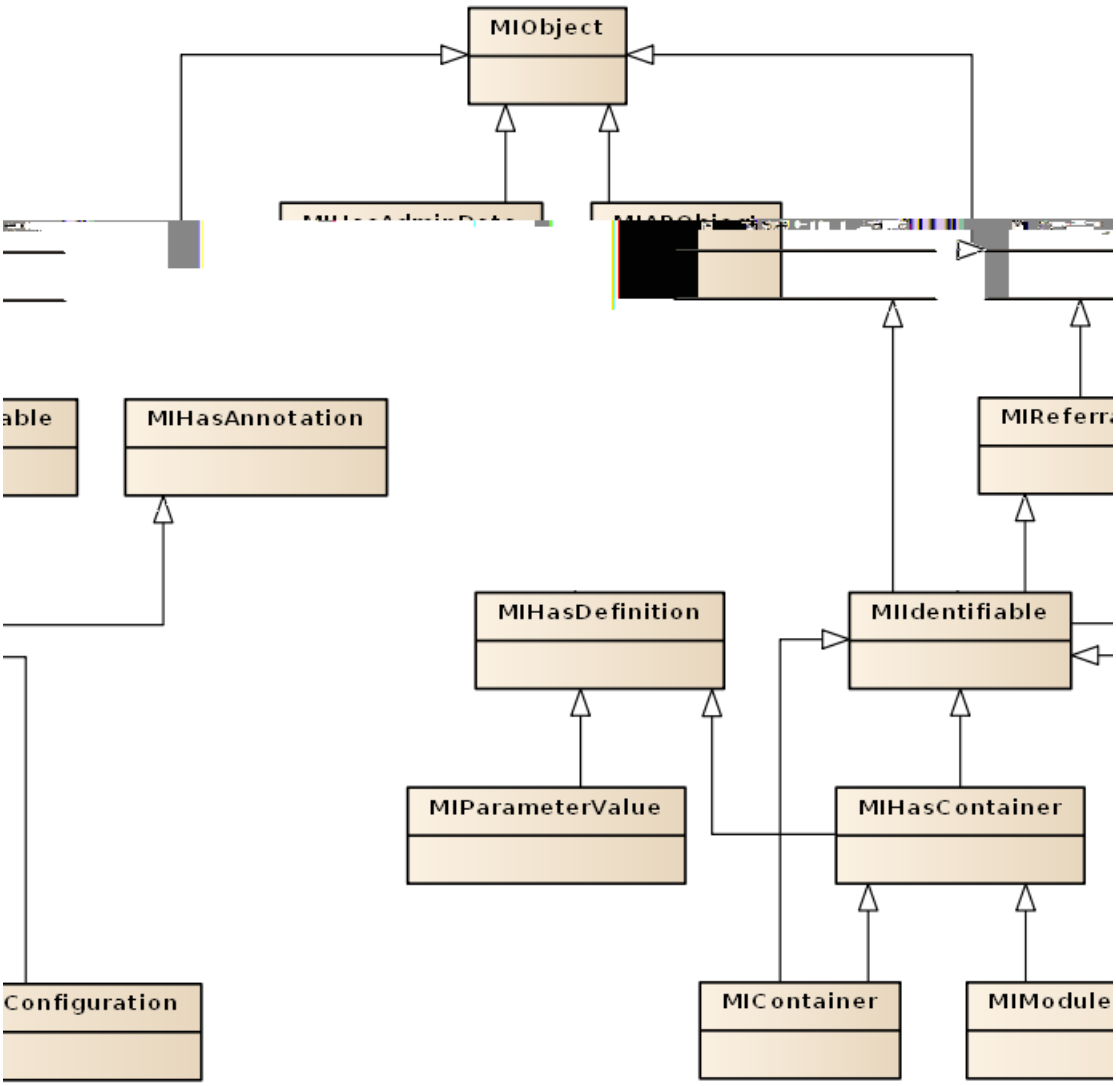


Figure 5.1: ECUC container type inheritance

client code shall always use `MIObject`, when AUTOSAR model objects are used, instead of `MDFObject`.

The figure 5.2 on the following page describes the class hierarchy of the `MIObject`.

5.1.3 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 5.3 on the next page shows a simple example of an MDF object containment hierarchy. This example contains two AUTOSAR packages with module configurations below.

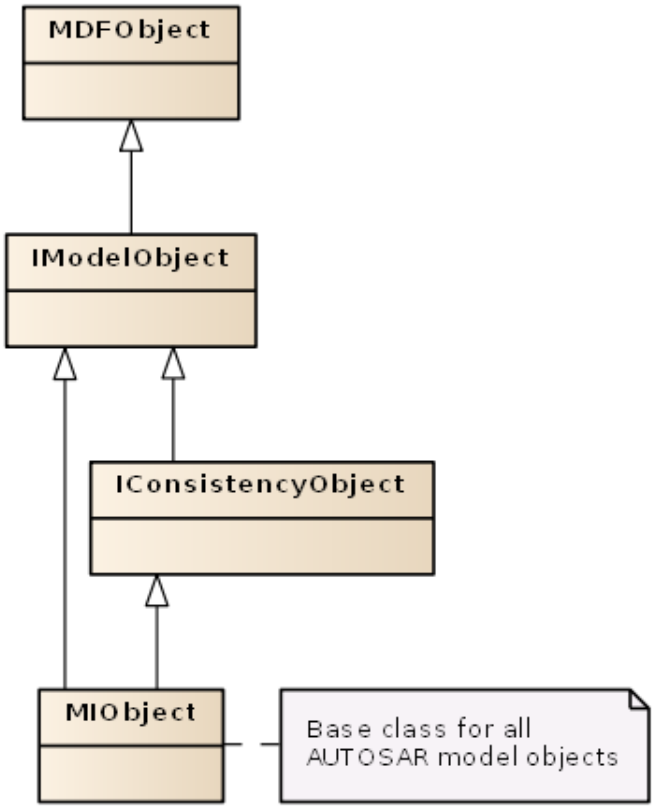


Figure 5.2: MIObject class hierarchy and base interfaces

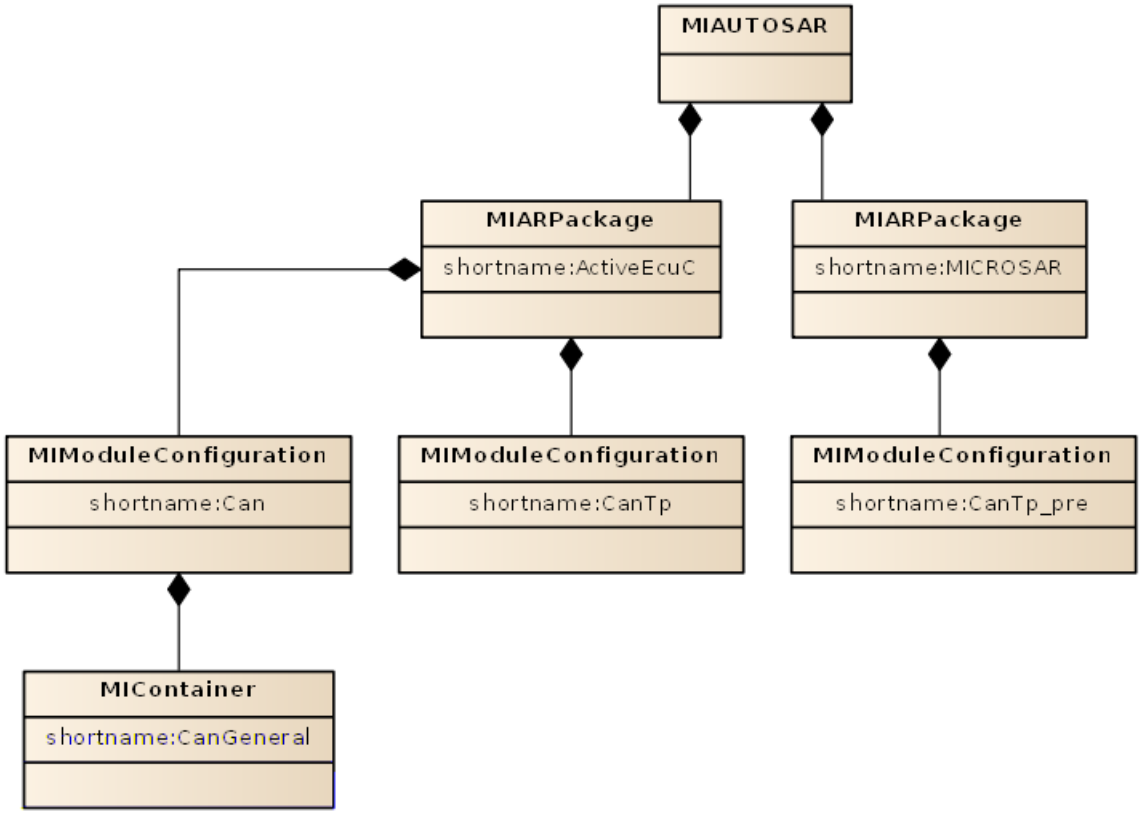


Figure 5.3: Autosar package containment

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

5.1.4 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.

5.1.5 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.

5.1.6 AUTOSAR references

All AUTOSAR reference objects in the MDF model have the base interface `MIARRef`.

Figure 5.4 on the following page shows this type hierarchy for the definition reference of an ECUC container.

In ARXML, such a reference can be specified as:

```
<DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">
    /MIRCOSAR/Com/ComGeneral
</DEFINITION-REF>
```

- `MIARRef.getValue()` returns the AUTOSAR path of the object, the reference points to (as specified in the ARXML file). In the example above `"/MIRCOSAR/Com/ComGeneral"` would be this value
- `MIContainerDefARRef.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 `"/MIRCOSAR/Com/ComGeneral"` exists in the model, this method will return it

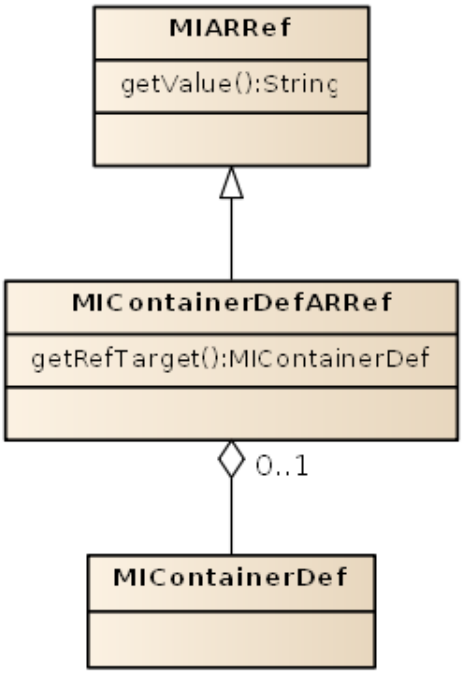


Figure 5.4: The ECUC container definition reference

5.1.7 Model changes

5.1.7.1 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.

5.1.7.2 Undo/redo

On basis of model change transactions, MDF provides means to undo and redo all changes made within one transaction. The tools GUI allows the user to execute undo/redo on this granularity.

5.1.7.3 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 listens to events to update its editors and views when model content changes.

5.1.7.4 Deleting model objects

Model objects must be deleted by a special service API. In Java code that's: `IModelOperationsPublished.deleteFromModel(MDFObject)`.

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

5.1.7.5 Access to deleted objects

All subsequent access to content of deleted objects throws a runtime exception. Reading the shortname of an `MIContainer`, for example.

5.1.7.6 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

5.1.7.7 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

5.1.7.8 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).

5.2 Post-build selectable

5.2.1 Model views

5.2.1.1 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 = ModelAccessUtil.isVisible(t.paramVariantA);
```

Listing 5.1: Check object visibility

5.2.1.2 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<IPredefinedVariantView> variants = viewMgr.getAllVariantViews();
```

Listing 5.2: Get all available variants

```
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 5.3: 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<IPredefinedVariantView> visibleVariants = viewMgr.
    getVisibleVariantViews(t.paramInvariant);
assertThat(visibleVariants.size(), equalTo(2));
assertThat(visibleVariants, containsInAnyOrder(t.variantViewA, t.variantViewB))
    ;

visibleVariants = viewMgr.getVisibleVariantViews(t.paramVariantA);
assertThat(visibleVariants.size(), equalTo(1));
assertThat(visibleVariants, containsInAnyOrder(t.variantViewA));
```

Listing 5.4: Get all variants, a specific object is visible in

5.2.1.3 Variant siblings

Variant siblings of an MDF object are MDF object instances which represent the same object but in other variants.

The method `IModelVarianceAccessPublished.getVariantSiblings()` 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 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.

5.2.1.4 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 `IInvariantValuesView` contains objects were all variant siblings have the same value and exist in all variants. One of the siblings is contained
- **Definition based invariance** (values which *shall be equal* in all variants):
  The `IInvariantEcucDefView` contains objects which are not allowed to be variant according to the BSWMD rules. One of the siblings is contained

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 5.5 describes the hierarchy.

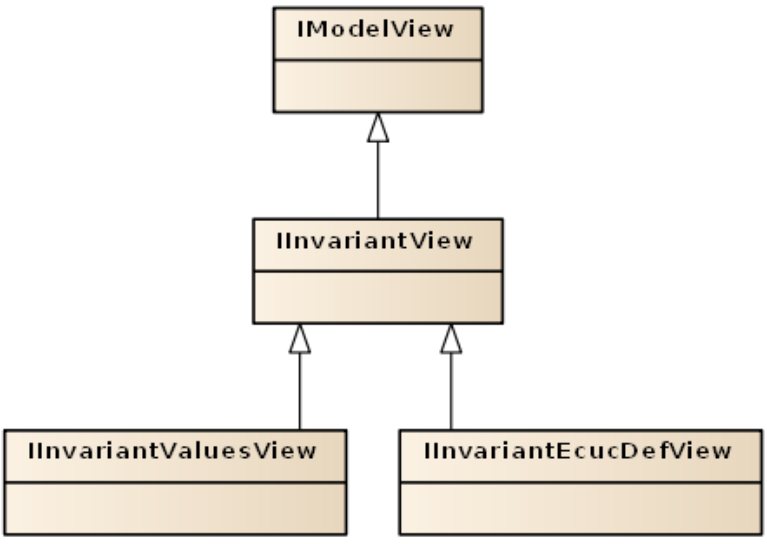


Figure 5.5: Invariant views hierarchy

The InvariantValues model view The `IInvariantValuesView` 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 `IInvariantValuesView`

So the semantic of the InvariantValues model view is that all values are equal in all variants.

You could retrieve an instance of `IInvariantValuesView` by calling `IModelViewManager.getInvariantValuesView()`.

```
IModelViewManager viewMgr =...;
IInvariantValuesView invariantView = viewMgr.getInvariantValuesView();
// Use the invariantView like any other model view
```

Listing 5.5: Retrieving an InvariantValues model view

Example The figure 5.6 describes an example for a module with containers and the visibility in the `IInvariantValuesView`.

- 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

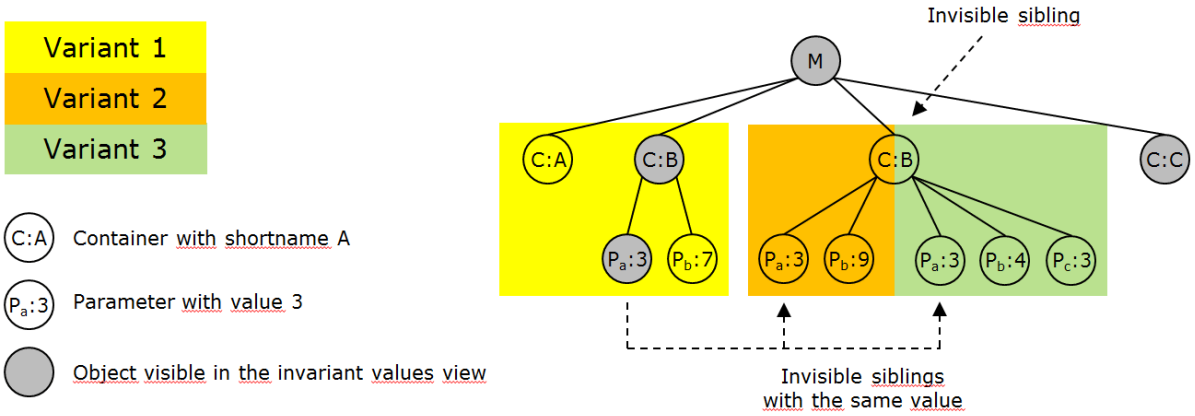


Figure 5.6: Example of a model structure and the visibility of the IInvariantValuesView

Specification See also the specification for details of the `IInvariantValuesView`.

The Invariant EcuC definition model view The `IInvariantEcucDefView` 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 `IInvariantEcucDefView` 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 `IInvariantValuesView`.

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 `IInvariantEcucDefView` 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 5.6: Retrieving an InvariantEcucDefView model view

Specification See also the specification for details of the `IInvariantEcucDefView`.

5.2.1.5 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:

- `ModelUtil`
- `ModelAccessUtil`
- `IReferrableAccess`
- `IModelAccess`
- `IModelCompareService`
- `DefRef`
- `AsrPath`
- `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, ...

5.2.1.6 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.

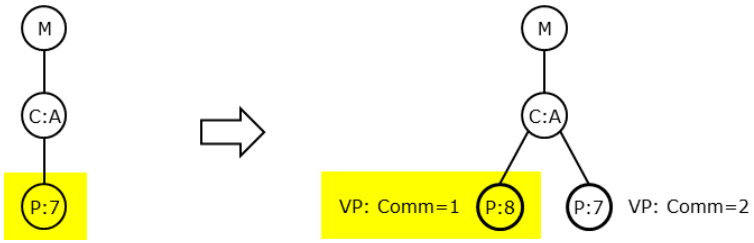
The method `getCreationModelView()` returns the `IModelView` of the `IViewedModelObject`, which was active when the model object was viewed `IViewedModelObject`.

5.2.2 Variant specific model changes

The CFG5 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 5.7 on the next page.

- Before modifying the parameter, this instance is invariant. The same MDF instance is visible in all variants



Criterion	Left	Right
Comm	1	2

Figure 5.7: Variant specific change of a parameter value

- When the client code changes the parameter value, the model automatically clones the parameter first
- 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 5.7: Execute code with variant specific changes

5.2.3 Variant common model changes

The CFG5 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 5.8 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

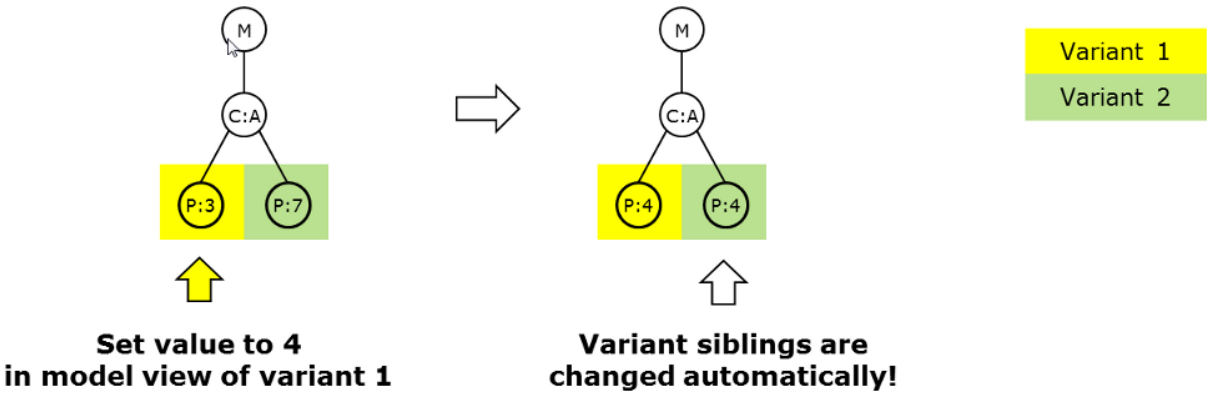


Figure 5.8: Variant common change of a parameter value

This change mode works with parameters and containers. The following operations are supported:

- **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.

5.3 BswmdModel details

5.3.1 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();
}
```

Bswmd model is instantiated at the beginning of generation process and is restricted to read only access. The DaVinci Configurator internal Model (MDF model) has 1:1 relationship to your BswmdModel.

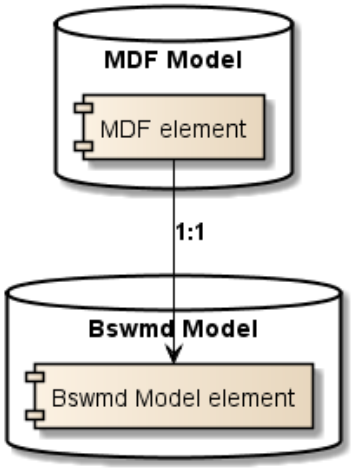


Figure 5.9: The relationship between the MDF model and the BswmdModel

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`.

5.3.1.1 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 an 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`<BigDecimal>
  - 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 5.1 contrasts the different model types and their corresonding 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 5.1: 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`.

5.3.1.2 DefRef Getter methods of Untyped Model

The DefRef API classes have no getter methods for the specific child types, but the children could be retrieve 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 listing 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 5.8: Sample code to access element in an Untyped model with DefRefs

```
final GIReferenceToContainer ref = getCanIfCanRxPduHrhRefBswmd();

final GIContainer target = ref.getRefTarget();
```

Listing 5.9: Resolves a Refference traget of an Reference Parameter

```
final GIParameter<BigInteger> param = getCanIfInitConfigBswmd().getParameter(
    CanIfInitConfiguration.CANIF_NUMBER_OF_CAN_TXPDU_IDS_DEFREF);

final BigInteger value = param.getValueMdf();
```

Listing 5.10: The value of a GIParameter

The figure 5.10 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 a DefinitionModel, please look into the Javadoc documentation of the `GI...` classes for more functionality.

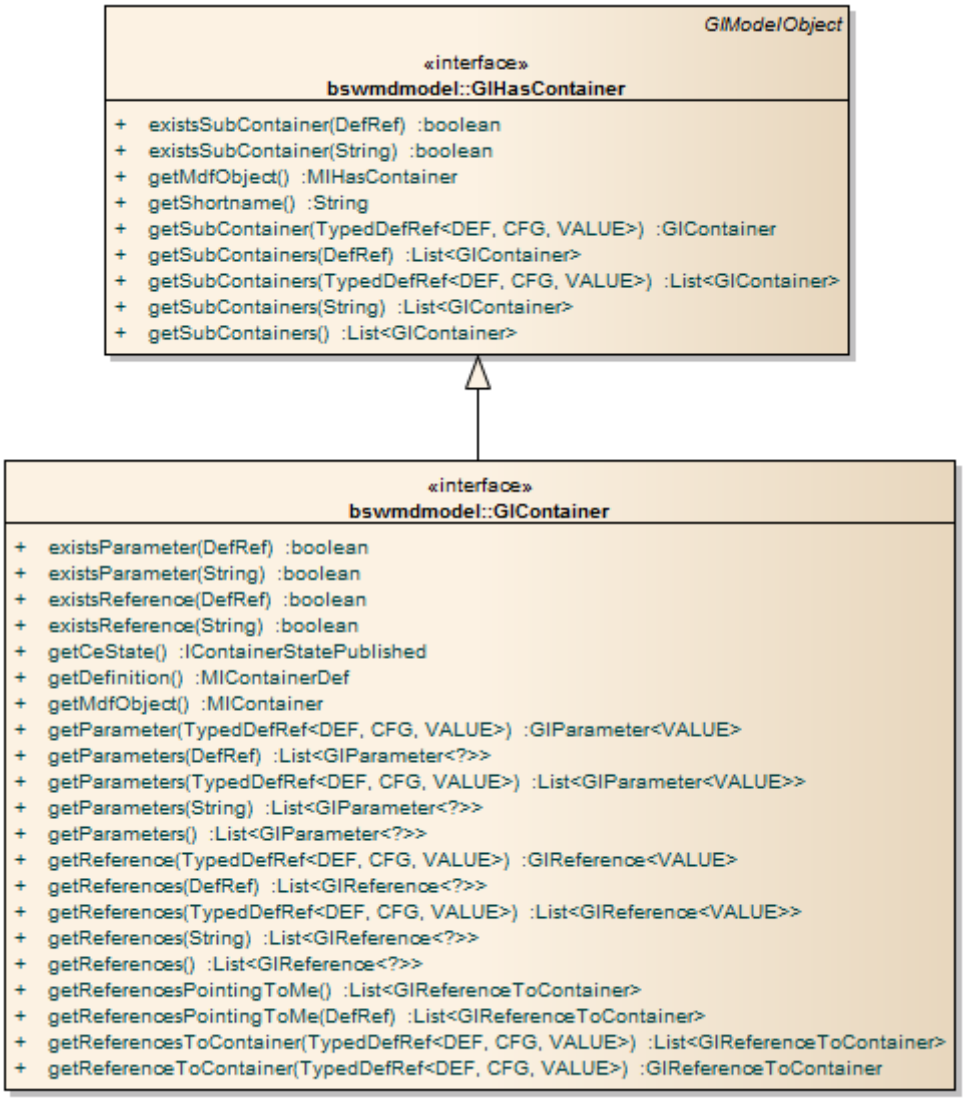


Figure 5.10: SubContainer DefRef navigation methods

5.3.1.3 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 their 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 reference to 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 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 is known at model generation time, the 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.

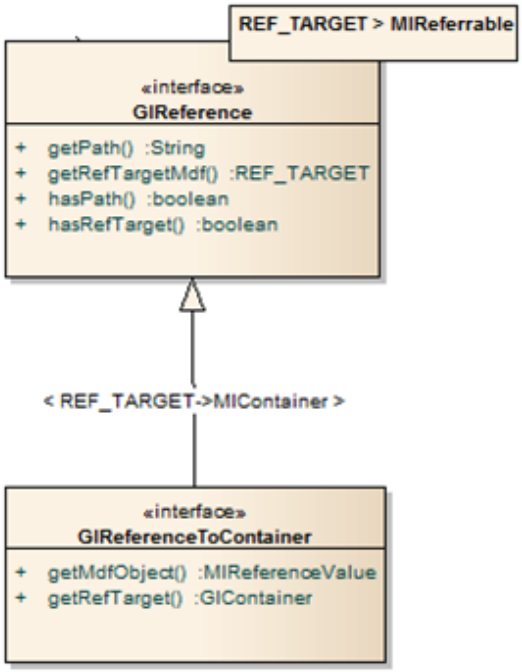


Figure 5.11: Untyped reference interfaces in the BswmdModel

SymbolicNameReferences SymbolicNameReferences have the same methods as `GIReferenceToContainer` 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.

5.3.1.4 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 5.12 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.

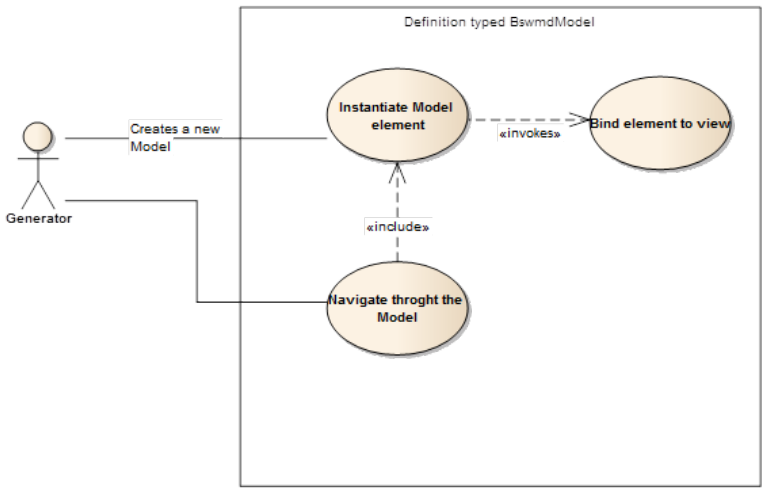


Figure 5.12: Creating a BswmdModel in the Post-build selectable use case

5.3.1.5 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" feature. 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 5.11: 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 5.12: 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 5.13: Java: Execute code with creation IModelView of BswmdModel object

5.3.1.6 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 build successively, by your calls.

It is also allowed to call a 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.

5.3.1.7 Optional Elements

All elements (Container, Parameter ...) are considered as optional if they have a multiplicity of 0:1. The BswmdModel provide a special handling of optional elements. This shall support you to recognize optional element during development (in the 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 ask the `GIOptional` object if the element exists (`optElement.exists()`). Then you can call `optElement.get()` to retrieve the real object.

You also have the choice to use the method `existsXXX()`. 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.

5.3.1.8 Class and Interface Structure of the BswmdModel

The upper part of the figure 5.13 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.

5.4 Model utility classes

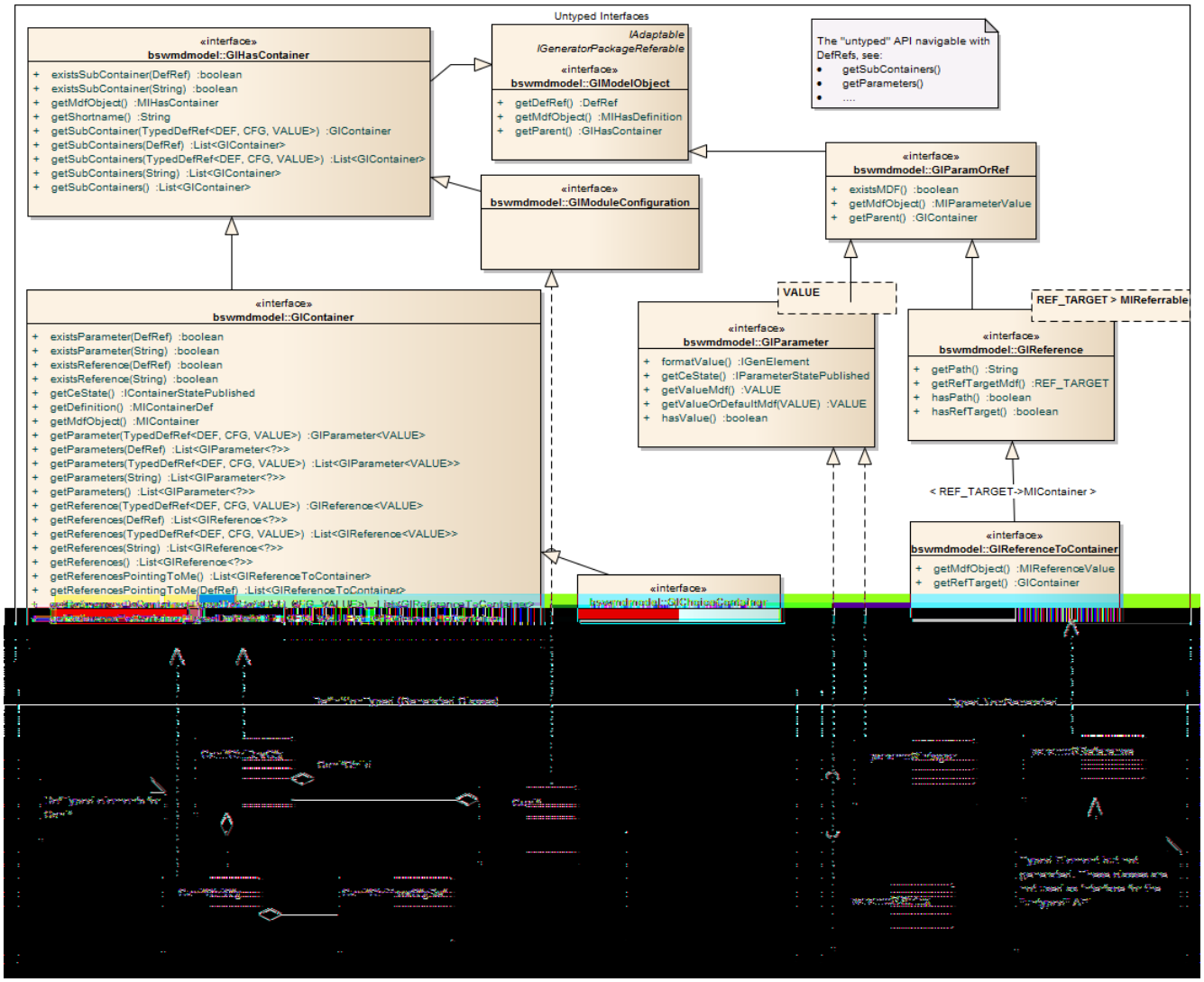


Figure 5.13: Class and Interface Structure of the BswmdModel

5.4.1 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!

5.4.2 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 releated MDF object using its AsrObjectLink instance. But this search-by-link cannot be guaranteed for each object type and after model changes (details and restrictions below).

5.4.2.1 Object links depend on the MDF object type

- **Referrables**
  The object link is actually identical with the AUTOSAR path
- **Ecuc objects with a definition** (module, container and parameter)
  The object link additionally stores the DefRef
- **Ecuc parameters**
  The object link additionally stores the parameters index. This is the index of all parameters with the same definition below the same parent container instance in the unfiltered model view

5.4.2.2 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

5.4.2.3 Examples for object link strings

The method `getObjectLinkString()` returns for example the following strings:

- For a container or module configuration object, the AUTOSAR path is returned: `"/ActiveEcuC/Can/CanGeneral"`
- For a parameter, the parents AUTOSAR path, the last shortname of its definition and a positional index in the list of parameters with the same definition is used: `"/ActiveEcuC/Can/CanGeneral[2:SomeDefName]"`
- In case of variant objects, all variants, this object is visible in, are added: `/ActiveEcuC/Can/CanConfigSet/CanHardwareObject[0:CanControllerRef]{VariantA, VariantB}`

5.4.3 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 5.14 shows the structure of the `DefRef` class and its sub classes.

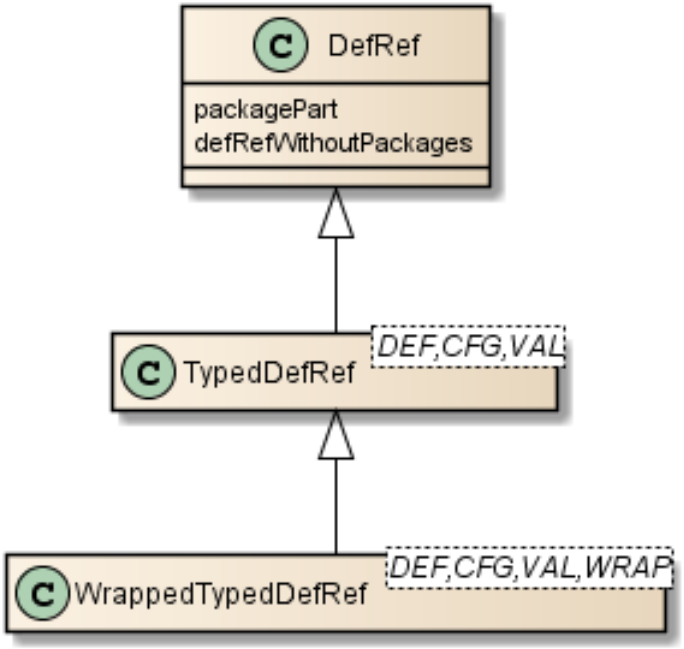


Figure 5.14: DefRef class structure

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 5.4.3.2 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 5.14: DefRef isDefinitionOf methods

5.4.3.1 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.

5.4.3.2 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 5.15: 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]`.

5.4.4 CeState

The `CeState` is an object which allows to retrieve different states of a configuration entity (typically containers or parameters).

The most important APIs for generator and script code are:

- `IParameterStatePublished`
- `IContainerStatePublished`

5.4.4.1 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 5.16: Getting CeState objects using the BSWMD model

5.4.4.2 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 5.15 shows the inheritance hierarchy of the `IParameterStatePublished` class and its sub classes.

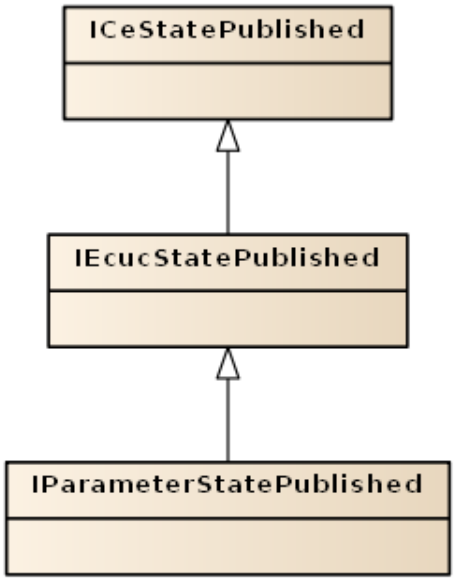


Figure 5.15: IParameterStatePublished class structure

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)
  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.

5.4.4.3 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 5.16 shows the inheritence hierarchy of the `IContainerStatePublished` class and its sub classes.

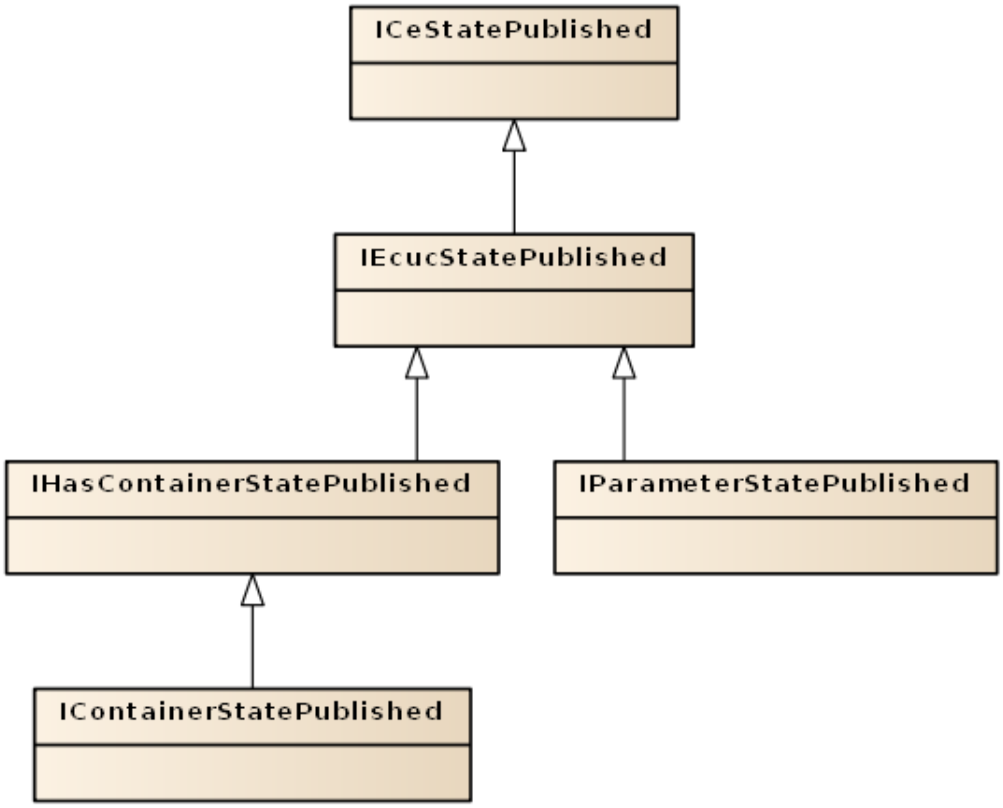


Figure 5.16: IContainerStatePublished class structure

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.

6 AutomationInterface Content

6.1 Introduction

This chapter describes the content of the DaVinci Configurator AutomationInterface.

6.2 Folder Structure

The AutomationInterface consists of the following files and folders:

- **BswmdModel:** contains the generated BswmdModel that is automatically created by the DaVinci Configurator during startup
- **Core**
  - **AutomationInterface**
    - **__doc** (find more details to its content in chapter 6.3)
      - **DVCfg__AutomationInterfaceDocumentation.pdf:** this document
      - **javadoc:** Javadoc HTML pages
      - **templates:** script file and script project templates for a simple start of script development
    - **buildLibs:** AutomationInterface Gradle Plugin to provide the build logic to build script projects, see also 7.6 on page 140
    - **libs:** compile bindings to Groovy and to the DaVinci Configurator AutomationInterface, used by IntelliJ IDEA and Gradle
    - **licenses:** the licenses of the used open source libraries

6.3 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

6.3.1 DVCfg__AutomationInterfaceDocumentation.pdf

You find this document as described in chapter 6.2. 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 6.3.2 on the following page.

6.3.2 Javadoc HTML Pages

You find this documentation as described in chapter 6.2 on the previous page. 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.

6.3.3 Script Templates

You find the Script Templates as described in chapter 6.2 on the preceding page. You may copy them for a quick startup in script development.

6.4 Libs and BuildLibs

The AutomationInterface contains libraries to build projects, see **buildLibs** in 6.2 on the previous page . And it contains other libraries which are described in **libs** in 6.2 on the preceding page.

7 Automation Script Project

7.1 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 7.6 on page 140.

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.

7.2 Automation Script Project Creation

To create a new script project please follow the instructions in chapter 2.4 on page 10.

7.3 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 sources and resource files
- Files
  - Gradle files - see 7.6.2 on page 141 for details
    - `gradlew.bat`
    - `build.gradle`
    - `settings.gradle`
    - `projectConfig.gradle`
    - `dvCfgAutomationBootstrap.gradle`
  - IntelliJ Project files (optional)

* ProjectName.iws
* ProjectName.iml
* ProjectName.ipr

7.4 IntelliJ IDEA Usage

7.4.1 Supported versions

The supported IntelliJ IDEA versions are:

- 2016.1 (.0-3)
- 2016.2

Please use one of the versions above. With other versions, there could be problems with the editing, code completion and so on.

7.4.2 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 7.1: Project Build

Project Continuous Build A further option is provided for the case you prefer an automatic project building each time you save your implementation. If you choose the menu option **<ProjectName> continuous** [**build**] in the toolbar the Run Button has to be pressed only one time to start the continuous building. Hence forward each saving of your implementation triggers an automatic building of the script project.

But be aware that the continuous build option is available for .java and .groovy files only. In case of changes in e.g. .gradle files you still have to press the Run Button in order to build the project.

Figure 7.2: Project Continuous Build

The Continuous Build process can be stopped with the Stop Button in the Run View.

Figure 7.3: Stop Continuous Build

If you want to exit the IntelliJ IDEA while the Continuous Build process is still running, you will be asked to disconnect from it. Having disconnected you are allowed to exit the IDE.

Figure 7.4: Disconnect from Continuous Build Process

7.4.3 Debugging with IntelliJ

Be aware that only script projects and not script files are debuggable.

To enable debugging you must start DaVinci Configurator application with the `enableDebugger` option as described in 7.5 on the following page.

In the IntelliJ IDEA choose the option **<ProjectName> [debug]** in the Run Menu located in the toolbar. Pressing the Debug Button starts a debug session.

Figure 7.5: Project Debug

Set your breakpoints in IntelliJ IDEA and execute the task. To stop the debug session press the Stop Button in the Debugger View.

Figure 7.6: Stop Debug Session

If you want to exit the IntelliJ IDEA while the Debug process is still running, you will be asked to disconnect from it. Having disconnected you are allowed to exit the IDE.

Figure 7.7: Disconnect from Debug Process

7.4.4 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 2.4.3 on page 12.

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 2.4.3 on page 12.

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 7.4 on page 138.

7.5 Debugging Script Project

Be aware that only script projects and not script files are debuggable.

To debug a script project, any java debugger could be used. Simply add the `enableDebugger` parameter to the commandline of the DaVinci Configurator and attach your debugger.

```
DVCfgCmd -s MyApplScriptTask --enableDebugger
```

You could attach a debugger at port `8000` (default).

```
DVCfgCmd -s MyApplScriptTask --enableDebugger 12345 --waitForDebugger
```

You could attach a debugger at port `12345` and the `DVCfgCmd` process will wait until the debugger is attached. You could also use these commandline parameters with the `DaVinciCFG.exe` to debug a script project with the DaVinci Configurator UI.

7.6 Build System

The build system uses Gradle[1] to build a single Jar file. It also setups the dependencies to the DaVinci Configurator and create the IntelliJ IDEA project.

To setup the Gradle installation, see chapter 2.4.4 on page 13.

[1] `http://gradle.org/` [2016-05-25]

7.6.1 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 file is then located in:

- `<ProjectRoot>\build\libs\<ProjectName>-<ProjectVersion>.jar`

7.6.2 Gradle File Structure

The default automation project contains the following Gradle build files:

- `gradlew.bat`
  - Gradle batch file to start Gradle (Gradle Wrapper[2])
- `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[3]
- `projectConfig.gradle`
  - Contains automation project specific settings - You can modify it to adapt the build to your needs
- `dvCfgAutomationBootstrap.gradle`
  - This is the internal bootstrap file. **DO NOT** change the file content.

7.6.2.1 projectConfig.gradle File settings

The file contains two essential parts of the build:

- Names of the scripts to load (automationClasses)
- The path to the DaVinci Configurator installation (dvCfgInstallation)

automationClasses You **have to add your classes** to the list of `automationClasses` to make them loadable.

The syntax of `automationClasses` is a list of `Strings`, of all classes as full qualified Class names.
Syntax: `"javaPkg.subPkg.ClassName"`

```
// The property project.ext.automationClasses defines the classes to load
project.ext.automationClasses = [
        "sample.MyScript",
        "otherPkg.MyOtherScript",
        "javapkg.ClassName"
]
```

Listing 7.1: The automationClasses list in projectConfig.gradle

[2] `https://docs.gradle.org/current/userguide/gradle_wrapper.html`

dvCfgInstallation The `dvCfgInstallation` defines the path to the DaVinci Configurator installation in your SIP. The installation is needed to retrieve the build dependencies and the generated model.

You **can** change the path to any location containing the correct version of the DaVinci Configurator.

You could also evaluate SystemEnv variables, other project properties or Gradle settings to define the path dependent of the development machine, instead of encoding an absolute path. This will help, when the project is committed to a version control system. But this is project dependent and out of scope of the provided template project.

7.6.3 Advanced Build Topics

7.6.3.1 Gradle dvCfgAutomation API Reference

The DaVinci Configurator build system provides a Gradle DSL API to set properties of the build. The entry point is the keyword `dvCfgAutomation`

```
dvCfgAutomation {
    classes project.ext.automationClasses
}
```

Listing 7.2: DaVinci Configurator build Gradle DSL API

The following methods are defined inside of the `dvCfgAutomation` block:

- `classes` (Type `List<String>`) - Defines the automation classes to load
- `useBswmdModel` (Type `boolean`) - Enables or disables the usage of the BswmdModel inside of the script project.

useBswmdModel The `useBswmdModel` enables or disables the usage of the BswmdModel inside of the project. This is helpful, if you want to create a project, which shall run with **different SIPs**. This prevent the inclusion of the BswmdModel. The default is `true` (Use the BswmdModel) if nothing is specified.

```
dvCfgAutomation {
   useBswmdModel false
}
```

Listing 7.3: DaVinci Configurator build Gradle DSL API - useBswmdModel

8 AutomationInterface Changes between Versions

This chapter describes all API changes between different MICROSAR releases.

8.1 Changes in MICROSAR AR4-R16 - Cfg5.13

8.1.1 General

This is the **first** version of the DaVinci Configurator AutomationInterface.

8.1.2 API Stability

The API is not stable yet and could still be changed in later releases. So it could be necessary to migrate your code when you update to later versions of the DaVinci Configurator.

8.1.3 Beta Status

Some features of the AutomationInterface are have beta status. This will change for later versions of the AutomationInterface. Which means that some features:

- Are not fully tested
- Missing documentation
- Missing functionality

9 Appendix

Nomenclature

AI Automation Interface

AUTOSAR AUTomotive Open System ARchitecture

CE Configuration Entity (typically a container or parameter)

Cfg DaVinci Configurator

Cfg5 DaVinci Configurator

DV DaVinci

IDE Integrated Development Environment

JAR Java Archive

JDK Java Development Kit

JRE Java Runtime Environment

MDF Meta-Data-Framework

MSN ModuleShortName

Figures

Tables

Listings

Todo list