

# **DaVinci Configurator AutomationInterface**

**Development Documentation of the AutomationInterface (AI)**

DaVinci Configurator Team

July 21, 2020

© 2020

Vector Informatik GmbH  
Ingersheimerstr. 24  
70499 Stuttgart

# Contents

<b>1</b>	<b>Introduction</b>	<b>13</b>
1.1	General . . . . .	13
1.2	Facts . . . . .	13
<b>2</b>	<b>Getting started with Script Development</b>	<b>14</b>
2.1	General . . . . .	14
2.2	Automation Script Development Types . . . . .	14
2.3	Script File . . . . .	14
2.4	Script Project . . . . .	16
2.4.1	Script Project Development . . . . .	18
2.4.2	Java JDK Setup . . . . .	19
2.4.3	IntelliJ IDEA Setup . . . . .	19
2.4.4	Gradle Setup . . . . .	20
<b>3</b>	<b>AutomationInterface Architecture</b>	<b>21</b>
3.1	Components . . . . .	21
3.2	Languages . . . . .	22
3.2.1	Why Groovy . . . . .	22
3.3	Script Structure . . . . .	23
3.3.1	Scripts . . . . .	23
3.3.2	Script Tasks . . . . .	23
3.3.3	Script Locations . . . . .	24
3.4	Script loading . . . . .	24
3.4.1	Internal Script Reload Behavior . . . . .	24
3.5	Script editing . . . . .	25
3.6	Licensing . . . . .	25
3.7	Script Coding Conventions and Constraints . . . . .	25
3.7.1	Usage of static fields . . . . .	26
3.7.2	Usage of Outer Closure Scope Variables . . . . .	26
3.7.3	States over script task execution . . . . .	27
3.7.4	Usage of Threads . . . . .	27
3.7.5	Usage of DaVinci Configurator private Classes Methods or Fields . . . . .	27
<b>4</b>	<b>AutomationInterface API Reference</b>	<b>28</b>
4.1	Introduction . . . . .	28
4.2	Script Creation . . . . .	29
4.2.1	Script Task Creation . . . . .	29
4.2.1.1	Script Creation with IDE Code Completion Support . . . . .	30
4.2.1.2	Script Task isExecutableIf . . . . .	31
4.2.2	Description and Help . . . . .	31
4.3	Script Task Types . . . . .	33
4.3.1	Available Types . . . . .	33
4.3.1.1	Application Types . . . . .	33
4.3.1.2	Project Types . . . . .	34
4.3.1.3	UI Types . . . . .	35
4.3.1.4	Generation Types . . . . .	36
4.3.1.5	Workflow Types . . . . .	37

4.4	Script Task Execution . . . . .	39
4.4.1	Execution Context . . . . .	39
4.4.1.1	Code Block Arguments . . . . .	40
4.4.2	Task Execution Sequence . . . . .	40
4.4.3	Script Path API during Execution . . . . .	41
4.4.3.1	Path Resolution by Parent Folder . . . . .	42
4.4.3.2	Path Resolution . . . . .	42
4.4.3.3	Script Folder Path Resolution . . . . .	43
4.4.3.4	Project Folder Path Resolution . . . . .	43
4.4.3.5	SIP Folder Path Resolution . . . . .	44
4.4.3.6	Temp Folder Path Resolution . . . . .	44
4.4.3.7	Other Project and Application Paths . . . . .	45
4.4.4	Script logging API . . . . .	45
4.4.5	User Interactions and Inputs . . . . .	46
4.4.5.1	UserInteraction . . . . .	46
4.4.5.2	Progress Indication . . . . .	47
4.4.6	Script Error Handling . . . . .	49
4.4.6.1	Script Exceptions . . . . .	49
4.4.6.2	Script Task Abortion by Exception . . . . .	49
4.4.6.3	Unhandled Exceptions from Tasks . . . . .	50
4.4.7	User defined Classes and Methods . . . . .	51
4.4.8	Usage of Automation API in own defined Classes and Methods . . . . .	52
4.4.8.1	Access the Automation API like the Script code{} Block . . . . .	52
4.4.8.2	Access the Project API of the current active Project . . . . .	52
4.4.9	User defined Script Task Arguments . . . . .	53
4.4.9.1	User defined Argument Validators . . . . .	55
4.4.9.2	Call Script Task with Task Arguments from Commandline . . . . .	57
4.4.10	Stateful Script Tasks . . . . .	58
4.4.11	ScriptAccess - Calling ScriptTasks . . . . .	60
4.5	Project Handling . . . . .	61
4.5.1	Projects . . . . .	61
4.5.2	Accessing the active Project . . . . .	61
4.5.3	Accessing the project search . . . . .	63
4.5.4	Accessing Project Settings . . . . .	64
4.5.4.1	Target Project Settings . . . . .	64
4.5.4.2	UseCase Project Settings . . . . .	66
4.5.5	Creating a new Project . . . . .	67
4.5.5.1	Mandatory Settings . . . . .	68
4.5.5.2	General Settings . . . . .	68
4.5.5.3	Target Settings . . . . .	69
4.5.5.4	Post Build Settings . . . . .	70
4.5.5.5	Folders Settings . . . . .	70
4.5.5.6	DaVinci Developer Settings . . . . .	72
4.5.5.7	vVIRTUALtarget Settings . . . . .	73
4.5.5.8	TA Tool Suite Settings . . . . .	75
4.5.6	Opening an existing Project . . . . .	76
4.5.6.1	Parameterized Project Load . . . . .	76
4.5.6.2	Open Project Details . . . . .	77
4.5.7	Create Ecu Configuration Report . . . . .	77
4.5.8	Create Support Request Package . . . . .	79
4.5.9	Saving a Project . . . . .	80

4.5.10	Opening AUTOSAR Files as Project . . . . .	82
4.5.10.1	Raw AUTOSAR models as Project . . . . .	83
4.6	Model . . . . .	84
4.6.1	Introduction . . . . .	84
4.6.2	Getting Started . . . . .	84
4.6.2.1	Read the ActiveEcuc . . . . .	84
4.6.2.2	Write the ActiveEcuc . . . . .	88
4.6.2.3	Read the SystemDescription . . . . .	91
4.6.2.4	Write the SystemDescription . . . . .	92
4.6.3	BswmdModel in AutomationInterface . . . . .	94
4.6.3.1	BswmdModel Package and Class Names . . . . .	94
4.6.3.2	Reading with BswmdModel . . . . .	94
4.6.3.3	Writing with BswmdModel . . . . .	95
4.6.3.4	Sip DefRefs . . . . .	96
4.6.3.5	BswmdModel DefRefs . . . . .	96
4.6.3.6	Untyped Model with the DefRef API . . . . .	97
4.6.3.7	Switching from Domain Models to BswmdModel . . . . .	98
4.6.4	MDF Model in AutomationInterface . . . . .	99
4.6.4.1	Reading the MDF Model . . . . .	99
4.6.4.2	Reading the MDF Model by String . . . . .	102
4.6.4.3	Writing the MDF Model . . . . .	104
4.6.4.4	Simple Property Changes . . . . .	105
4.6.4.5	Creating single Child Members (0:1) . . . . .	105
4.6.4.6	Creating and adding Child List Members (0:*) . . . . .	106
4.6.4.7	Updating existing Elements . . . . .	108
4.6.4.8	Deleting Model Objects . . . . .	109
4.6.4.9	Duplicating Model Objects . . . . .	110
4.6.4.10	Special properties and extensions . . . . .	110
4.6.4.11	Reverse Reference Resolution - ReferencesPointingToMe . . . . .	112
4.6.4.12	Derived Containers . . . . .	113
4.6.4.13	AUTOSAR Root Object . . . . .	113
4.6.4.14	ActiveEcuC . . . . .	113
4.6.4.15	DefRef based Access to Containers and Parameters . . . . .	114
4.6.4.16	Ecuc Parameter and Reference Value Access . . . . .	115
4.6.5	SystemDescription Access . . . . .	117
4.6.6	Transactions . . . . .	118
4.6.6.1	Transactions API . . . . .	118
4.6.6.2	Operations . . . . .	120
4.6.7	Model Synchronization . . . . .	121
4.6.8	PreBuild and PostBuild Variance (Post-build selectable) . . . . .	122
4.6.8.1	Investigate Project Variance . . . . .	122
4.6.8.2	Variant Model Objects . . . . .	123
4.6.9	Additional Model API . . . . .	125
4.6.9.1	User Annotations . . . . .	125
4.7	Generation . . . . .	127
4.7.1	Code Generation . . . . .	127
4.7.1.1	Generation Settings . . . . .	127
4.7.1.2	Generation of Generation Steps . . . . .	131
4.7.1.3	Evaluate generation or validation results . . . . .	132
4.7.2	Generation Task Types . . . . .	133
4.7.3	Software Component Templates and Contract Phase Headers Generation	135

4.7.3.1	Swct Generation Settings . . . . .	135
4.7.3.2	Generation with default Project Settings . . . . .	135
4.7.3.3	Generation of all Software Components . . . . .	135
4.7.3.4	Generation of one Software Component . . . . .	136
4.7.3.5	Generation of multiple Software Components . . . . .	137
4.7.3.6	Evaluate generation results . . . . .	137
4.8	Validation . . . . .	138
4.8.1	Introduction . . . . .	138
4.8.2	Access Validation-Results . . . . .	139
4.8.3	Model Transaction and Validation-Result Invalidation . . . . .	139
4.8.4	Solve Validation-Results with Solving-Actions . . . . .	140
4.8.4.1	Solver API . . . . .	140
4.8.5	Advanced Topics . . . . .	142
4.8.5.1	Access Validation-Results of a Model Object . . . . .	142
4.8.5.2	Access Validation-Results of a DefRef . . . . .	142
4.8.5.3	Filter Validation-Results using an ID Constant . . . . .	143
4.8.5.4	Identification of a Particular Solving-Action . . . . .	143
4.8.5.5	Validation-Result Description as MixedText . . . . .	144
4.8.5.6	Further IValidationResultUI Methods . . . . .	144
4.8.5.7	IValidationResultUI in a variant (Post-Build selectable) Project	145
4.8.5.8	Erroneous CEs of a Validation-Result . . . . .	145
4.8.5.9	Examine Solving-Action Execution . . . . .	147
4.8.5.10	Create a Validation-Result in a Script Task . . . . .	148
4.8.5.11	Turn off auto-solving-action execution . . . . .	150
4.9	Workflow . . . . .	151
4.9.1	Update Workflow . . . . .	151
4.9.1.1	Prerequisites . . . . .	151
4.9.1.2	Method Overview . . . . .	151
4.9.1.3	Example: Content of Input Files has changed.	156
4.9.1.4	Example: List of input files shall be changed . . . . .	157
4.9.1.5	Example: Diagnostic Data as input file . . . . .	158
4.9.1.6	Example: Change the list of Input Files in an Postbuild-Selectable Variant project . . . . .	159
4.9.1.7	Example: File Preprocessing of Input Files and querying the result	159
4.9.2	Configure Variants . . . . .	160
4.9.2.1	Example . . . . .	161
4.9.2.2	Simple Mode APIs . . . . .	161
4.9.2.3	Variant API . . . . .	162
4.10	Domains . . . . .	163
4.10.1	Communication Domain . . . . .	163
4.10.1.1	CanControllers . . . . .	165
4.10.1.2	CanFilterMasks . . . . .	166
4.10.1.3	CanPdus . . . . .	166
4.10.2	Diagnostics Domain . . . . .	168
4.10.2.1	DemEvents . . . . .	170
4.10.3	Mode Management Domain . . . . .	172
4.10.3.1	BswM Auto Configuration . . . . .	172
4.10.4	Runtime System Domain . . . . .	175
4.10.4.1	Component Port Selection . . . . .	176
4.10.4.2	Signal Instance Selection . . . . .	182
4.10.4.3	Communication Element Selection . . . . .	186

4.10.4.4 Component Type Selection . . . . .	190
4.10.4.5 Event Selection . . . . .	192
4.10.4.6 Executable Entity Selection . . . . .	195
4.10.4.7 Port Interface Selection . . . . .	198
4.10.4.8 Origin Component Port Selection . . . . .	200
4.10.4.9 Component Port Connection . . . . .	203
4.10.4.10 Disconnect (unmap) Component Ports . . . . .	215
4.10.4.11 Terminating Component Ports . . . . .	216
4.10.4.12 Data Mapping . . . . .	221
4.10.4.13 Remove Data Mappings . . . . .	243
4.10.4.14 Create Component Prototypes . . . . .	248
4.10.4.15 Create Delegation Ports . . . . .	250
4.10.4.16 Task Mapping . . . . .	255
4.10.4.17 Bridge Between MDF and Model Abstractions . . . . .	279
4.10.4.18 Deleting Elements . . . . .	280
4.10.4.19 Variant Handling . . . . .	285
4.10.4.20 Retrieving Short Name Paths and Fully Qualified Names . . . . .	286
4.10.4.21 Best Practice And Further Examples . . . . .	287
4.11 Unresolved Reference API . . . . .	291
4.11.1 Active ECUC Unresolved Reference API . . . . .	291
4.11.1.1 Selecting unresolved references . . . . .	292
4.11.1.2 Set changeable unresolved references . . . . .	293
4.12 Reporting . . . . .	294
4.12.1 Custom Report . . . . .	294
4.13 Persistency . . . . .	296
4.13.1 Model Export . . . . .	296
4.13.1.1 Export ActiveEcuc . . . . .	296
4.13.1.2 Export PostBuild Variants (Post-build selectable) . . . . .	296
4.13.1.3 Export PreBuild Variants . . . . .	297
4.13.1.4 Export Module Configuration . . . . .	298
4.13.1.5 Advanced Exports . . . . .	298
4.13.2 Model Import . . . . .	300
4.13.2.1 Module Configuration Import . . . . .	300
4.13.2.2 Specify import mode and module filter . . . . .	301
4.14 Project Comparison . . . . .	303
4.14.1 Automatic merge . . . . .	303
4.14.1.1 Accessing the API . . . . .	303
4.14.2 Configure automatic merge . . . . .	303
4.14.2.1 The settings in detail . . . . .	303
4.14.2.2 Global conflict resolution . . . . .	304
4.14.2.3 Automatic merge result evaluation . . . . .	304
4.14.2.4 Comparison scope . . . . .	305
4.14.2.5 Platform function development . . . . .	305
4.14.3 Automatic merge result . . . . .	306
4.14.3.1 Example of result evaluation . . . . .	307
4.15 Utilities . . . . .	309
4.15.1 Constraints . . . . .	309
4.15.2 Converters . . . . .	310
4.16 Advanced Topics . . . . .	312
4.16.1 Java Development . . . . .	312
4.16.1.1 Script Task Creation in Java Code . . . . .	312

4.16.1.2 Java Code accessing Groovy API . . . . .	312
4.16.1.3 Java Code in dvgroovy Scripts . . . . .	313
4.16.2 Unit testing API . . . . .	314
4.16.2.1 JUnit4 Integration . . . . .	314
4.16.2.2 Execution of Spock Tests . . . . .	315
4.16.2.3 Registration of Unit Tests in Scripts . . . . .	316
4.16.2.4 Model TestInfrastructure . . . . .	316
4.16.2.5 Automation Project TestInfrastructure . . . . .	317
4.17 Generator Testing . . . . .	318
4.17.1 White-box tests . . . . .	318
4.17.1.1 Step-by-step . . . . .	318
4.17.1.2 Compilation . . . . .	318
4.17.1.3 GeneratorTestingApi . . . . .	319
4.17.1.4 Test OuputFile Generation . . . . .	320
4.17.1.5 Test GeneratorResults and ValidationResults . . . . .	321
<b>5 Data models in detail . . . . .</b>	<b>323</b>
5.1 MDF model - the raw AUTOSAR data . . . . .	323
5.1.1 Naming . . . . .	323
5.1.2 The models inheritance hierarchy . . . . .	323
5.1.2.1 MIOBJECT and MDFOBJECT . . . . .	324
5.1.3 The models containment tree . . . . .	324
5.1.4 The ECUC model . . . . .	326
5.1.5 Order of child objects . . . . .	326
5.1.6 AUTOSAR references . . . . .	326
5.1.7 Model changes . . . . .	327
5.1.7.1 Transactions . . . . .	327
5.1.7.2 Undo/redo . . . . .	327
5.1.7.3 Event handling . . . . .	327
5.1.7.4 Deleting model objects . . . . .	327
5.1.7.5 Access to deleted objects . . . . .	328
5.1.7.6 Set-methods . . . . .	328
5.1.7.7 Changing child list content . . . . .	328
5.1.7.8 Change restrictions . . . . .	328
5.2 Post-build selectable . . . . .	329
5.2.1 Model views . . . . .	329
5.2.1.1 What model views are . . . . .	329
5.2.1.2 The IMODELVIEWMANAGER project service . . . . .	329
5.2.1.3 Variant siblings . . . . .	331
5.2.1.4 The INVARIANT model views . . . . .	332
5.2.1.5 Accessing invisible objects . . . . .	334
5.2.1.6 IVIEWEDMODELOBJECT . . . . .	335
5.2.2 Variant specific model changes . . . . .	335
5.2.3 Variant common model changes . . . . .	336
5.3 BswmdModel details . . . . .	337
5.3.1 BswmdModel - DefinitionModel . . . . .	337
5.3.1.1 Types of DefinitionModels . . . . .	338
5.3.1.2 DefRef Getter methods of Untyped Model . . . . .	339
5.3.1.3 References . . . . .	341
5.3.1.4 Post-build selectable with BswmdModel . . . . .	342
5.3.1.5 Creation ModelView of the BswmdModel . . . . .	343
5.3.1.6 Lazy Instantiating . . . . .	344

5.3.1.7	Optional Elements . . . . .	344
5.3.1.8	Class and Interface Structure of the BswmdModel . . . . .	344
5.3.1.9	BswmdModel write access . . . . .	345
5.3.2	BswmdModel generation . . . . .	349
5.3.2.1	DerivativeMapping . . . . .	349
5.4	Model Utility Classes . . . . .	349
5.4.1	AutosarUtil . . . . .	349
5.4.2	AsrPath . . . . .	349
5.4.3	AsrObjectLink . . . . .	350
5.4.3.1	Restrictions of object links . . . . .	350
5.4.4	DefRefs . . . . .	350
5.4.4.1	TypedDefRefs . . . . .	352
5.4.4.2	DefRef Wildcards . . . . .	352
5.4.5	CeState . . . . .	353
5.4.5.1	Getting a CeState object . . . . .	353
5.4.5.2	IParameterStatePublished . . . . .	353
5.4.5.3	IContainerStatePublished . . . . .	354
5.5	Model Services . . . . .	355
5.5.1	EcucDefinitionAccess . . . . .	355
5.5.1.1	Post-build loadable . . . . .	356
5.5.1.2	Post-build selectable . . . . .	358
5.5.2	EcuConfigurationAccess . . . . .	359
5.5.2.1	Post-build loadable . . . . .	360
5.5.2.2	Post-build selectable . . . . .	362
<b>6</b>	<b>AutomationInterface Content</b>	<b>365</b>
6.1	Introduction . . . . .	365
6.2	Folder Structure . . . . .	365
6.3	Script Development Help . . . . .	365
6.3.1	AutomationInterfaceDocumentation PDF . . . . .	365
6.3.2	Javadoc HTML Pages . . . . .	366
6.3.3	Script Templates . . . . .	366
6.4	Libs and BuildLibs . . . . .	366
6.5	Beta API Usage . . . . .	366
<b>7</b>	<b>Automation Script Project</b>	<b>368</b>
7.1	Introduction . . . . .	368
7.2	Automation Script Project Creation . . . . .	368
7.3	Project File Content . . . . .	368
7.4	Deployment of the Jar File . . . . .	369
7.5	IntelliJ IDEA Usage . . . . .	369
7.5.1	Supported versions . . . . .	369
7.5.2	Show API Specifications (JavaDoc) . . . . .	369
7.5.3	Building Projects . . . . .	371
7.5.4	Debugging with IntelliJ . . . . .	372
7.5.5	Troubleshooting . . . . .	373
7.6	Project Usage in different DaVinci Configurator Versions . . . . .	374
7.7	Project Migration to newer DaVinci Configurator Version / SIP . . . . .	375
7.8	Debugging Script Project . . . . .	375
7.9	Build System . . . . .	376
7.9.1	Jar Creation and Output Location . . . . .	376
7.9.2	Gradle File Structure . . . . .	376

7.9.2.1	projectConfig.gradle File settings . . . . .	376
7.9.3	Advanced Build Topics . . . . .	377
7.9.3.1	Usage of external Libraries (Jars) in the AutomationProject . . . . .	377
7.9.3.2	Update IntelliJ IDEA project . . . . .	378
7.9.3.3	Static Compilation of Groovy Code . . . . .	379
7.9.3.4	Gradle Maven publishing of an AutomationProject . . . . .	380
7.9.3.5	Gradle dvCfgAutomation API Reference . . . . .	380
<b>8</b>	<b>AutomationInterface Changes between Versions</b>	<b>382</b>
8.1	Currently Supported Features . . . . .	382
8.2	Changes in MICROSAR AR4-R24 - Cfg5.21 . . . . .	385
8.2.1	General . . . . .	385
8.2.1.1	Groovy . . . . .	385
8.2.2	Project Creation TA Tool Suite workspace settings . . . . .	385
8.3	Changes in MICROSAR AR4-R23 - Cfg5.20 . . . . .	386
8.3.1	General . . . . .	386
8.3.2	Automation Script Project . . . . .	386
8.3.2.1	Supported IntelliJ IDEA Version . . . . .	386
8.3.3	Unit testing API . . . . .	386
8.3.4	Communication Domain . . . . .	386
8.3.4.1	FullCAN Flag of PDUs . . . . .	386
8.3.5	Mode Management Domain . . . . .	386
8.3.6	Runtime System Domain . . . . .	386
8.3.6.1	Changed Structure of Runtime System Domain Documentation . . . . .	386
8.3.6.2	Origin Context . . . . .	387
8.3.6.3	New Methods for Objects of Runtime System Domain . . . . .	387
8.3.6.4	TaskMapping . . . . .	388
8.3.6.5	Unmapping Component Ports . . . . .	388
8.3.6.6	Removing Data Mappings . . . . .	388
8.3.6.7	Select Elements by a Collection of Names . . . . .	388
8.3.6.8	Create Delegation Ports . . . . .	389
8.3.6.9	Create Selection Based On Existing Elements . . . . .	389
8.3.6.10	Connect Ports . . . . .	389
8.3.6.11	Map Communication Elements to System Signals . . . . .	389
8.3.6.12	Select Communication Elements . . . . .	389
8.3.6.13	Deleting Elements . . . . .	390
8.3.6.14	Variant Handling . . . . .	390
8.3.6.15	Performance . . . . .	390
8.3.7	Create Custom Report . . . . .	390
8.3.8	Automatic Merge . . . . .	390
8.3.9	Post-build Selectable . . . . .	390
8.3.9.1	Model Views . . . . .	390
8.3.10	mdfModel Api . . . . .	390
8.3.11	Generation . . . . .	391
8.3.11.1	Report settings . . . . .	391
8.3.12	Create Search Access . . . . .	391
8.4	Changes in MICROSAR AR4-R22 - Cfg5.19 . . . . .	392
8.4.1	General . . . . .	392
8.4.2	Automation Script Project . . . . .	392
8.4.2.1	Bootstrap file . . . . .	392
8.4.2.2	Supported IntelliJ IDEA Version . . . . .	392
8.4.2.3	Gradle . . . . .	392

8.4.3	Unresolved Reference API . . . . .	392
8.4.4	Update Workflow Settings . . . . .	392
8.4.5	Project Settings UseCase Api . . . . .	393
8.4.6	ECUC Unresolved Reference API . . . . .	393
8.5	Changes in MICROSAR AR4-R21 - Cfg5.18 . . . . .	394
8.5.1	General . . . . .	394
8.5.2	Automation Script Project . . . . .	394
8.5.2.1	Groovy . . . . .	394
8.5.2.2	Supported IntelliJ IDEA Version . . . . .	394
8.5.2.3	BuildSystem . . . . .	394
8.5.3	Unit testing API . . . . .	394
8.5.4	Model TestInfrastructure . . . . .	395
8.5.5	Automation Project TestInfrastructure . . . . .	395
8.5.6	Generator Testing . . . . .	395
8.5.7	Model changes . . . . .	395
8.5.8	Workflow . . . . .	396
8.5.9	Create Ecu Configuration Report . . . . .	396
8.5.10	Project Settings Target Api . . . . .	396
8.5.11	Runtime System Domain . . . . .	396
8.5.11.1	Component Port Connection . . . . .	396
8.5.11.2	Task Mapping . . . . .	396
8.5.11.3	Origin Component Port . . . . .	396
8.5.11.4	Origin Context Selection . . . . .	397
8.5.11.5	Component Port Selection . . . . .	397
8.6	Changes in MICROSAR AR4-R20 - Cfg5.17 . . . . .	398
8.6.1	General . . . . .	398
8.6.2	Automation Script Project . . . . .	398
8.6.2.1	BuildSystem . . . . .	398
8.6.2.2	Supported IntelliJ IDEA Version . . . . .	398
8.6.3	Persistency . . . . .	398
8.6.3.1	Model Module Import . . . . .	398
8.6.3.2	Model Export . . . . .	398
8.6.4	BswmdModel . . . . .	399
8.6.4.1	SIP DefRefs . . . . .	399
8.6.5	Runtime System Domain . . . . .	399
8.6.5.1	Component Port Selection . . . . .	399
8.6.5.2	Communication Element Selection . . . . .	399
8.6.5.3	Create Delegation Ports . . . . .	399
8.6.5.4	Task Mapping . . . . .	399
8.6.5.5	Simple API for connection between ports . . . . .	399
8.6.5.6	Simple API for data mapping . . . . .	400
8.6.5.7	Port Terminators . . . . .	400
8.6.5.8	Data Mapping . . . . .	400
8.6.5.9	Service Proxy Components . . . . .	400
8.6.5.10	Retrieve Short Name Paths and Fully Qualified Names . . . . .	400
8.6.5.11	More Examples . . . . .	400
8.7	Changes in MICROSAR AR4-R19 - Cfg5.16 . . . . .	401
8.7.1	General . . . . .	401
8.7.2	Automation Script Project . . . . .	401
8.7.2.1	Groovy . . . . .	401
8.7.2.2	BuildSystem . . . . .	401

8.7.2.3	Supported IntelliJ IDEA Version . . . . .	401
8.7.3	ScriptAccess . . . . .	401
8.7.4	UserInteraction - Progress Indication . . . . .	401
8.7.5	Project Handling . . . . .	402
8.7.6	Model Automation API . . . . .	402
8.7.6.1	Derived Containers . . . . .	402
8.7.6.2	Variance API . . . . .	402
8.7.6.3	CE State . . . . .	402
8.7.6.4	MDF Modification API . . . . .	402
8.7.7	Persistency . . . . .	402
8.7.7.1	Model Export . . . . .	402
8.7.8	Generation . . . . .	403
8.7.8.1	Generation Steps . . . . .	403
8.7.9	Runtime System Domain . . . . .	403
8.7.9.1	Component Port Selection . . . . .	403
8.7.9.2	Signal Instance Selection . . . . .	403
8.7.9.3	Bridge between mdf and model abstractions . . . . .	403
8.7.9.4	Create Component Prototypes . . . . .	403
8.7.9.5	Task Mapping . . . . .	403
8.8	Changes in MICROSAR AR4-R18 - Cfg5.15 . . . . .	404
8.8.1	General . . . . .	404
8.8.2	Automation Script Project . . . . .	404
8.8.2.1	Supported IntelliJ IDEA Version . . . . .	404
8.8.2.2	BuildSystem . . . . .	404
8.8.3	Script Execution . . . . .	404
8.8.3.1	User defined arguments . . . . .	404
8.8.4	Project Handling . . . . .	404
8.8.5	Project Creation vVIRTUALtarget settings . . . . .	404
8.8.6	Model changes . . . . .	405
8.8.7	Model Automation API . . . . .	406
8.8.7.1	IVarianceApi . . . . .	406
8.8.7.2	Access methods . . . . .	406
8.8.7.3	Reverse Reference Resolution - ReferencesPointingToMe . . . . .	406
8.8.7.4	Operations . . . . .	406
8.8.7.5	User Annotations . . . . .	406
8.8.7.6	Variance . . . . .	406
8.8.7.7	Model Synchronization . . . . .	406
8.8.8	Persistency . . . . .	406
8.8.9	Workflow . . . . .	407
8.8.10	Validation . . . . .	407
8.8.10.1	Validation-Result Access Methods . . . . .	407
8.8.11	Generation . . . . .	407
8.8.11.1	SWC Templates and Contract Headers Generation . . . . .	407
8.8.12	BswmdModel . . . . .	407
8.8.12.1	BswmdModel Groovy . . . . .	407
8.8.12.2	DerivativeMapping . . . . .	408
8.8.13	Mode Management Domain . . . . .	408
8.8.14	Runtime System Domain . . . . .	408
8.8.14.1	Data Mapping . . . . .	408
8.9	Changes in MICROSAR AR4-R17 - Cfg5.14 . . . . .	409
8.9.1	General . . . . .	409

---

8.9.2	Script Execution . . . . .	409
8.9.2.1	Stateful Script Tasks . . . . .	409
8.9.3	Automation Script Project . . . . .	409
8.9.3.1	Groovy . . . . .	409
8.9.3.2	Supported IntelliJ IDEA Version . . . . .	409
8.9.3.3	BuildSystem . . . . .	409
8.9.4	Converter Refactoring . . . . .	409
8.9.5	UserInteraction . . . . .	409
8.9.6	Project Load . . . . .	410
8.9.6.1	AUTOSAR Arxml Files . . . . .	410
8.9.7	Model . . . . .	410
8.9.7.1	Transactions . . . . .	410
8.9.7.2	MDF Model Read and Write . . . . .	410
8.9.7.3	SystemDescription Access . . . . .	411
8.9.7.4	ActiveEcuc . . . . .	411
8.9.8	Persistency . . . . .	411
8.9.9	Generation . . . . .	411
8.9.10	BswmdModel . . . . .	411
8.9.10.1	Writing with BswmdModel . . . . .	411
8.9.11	BswmdModel Groovy . . . . .	411
8.9.12	Diagnostics Domain . . . . .	412
8.9.13	Communication Domain . . . . .	412
8.9.14	Runtime System Domain . . . . .	412
8.10	Changes in MICROSAR AR4-R16 - Cfg5.13 . . . . .	413
8.10.1	General . . . . .	413
8.10.2	API Stability . . . . .	413
8.10.3	Beta Status . . . . .	413
<b>9</b>	<b>Appendix</b> . . . . .	<b>414</b>
	Nomenclature . . . . .	415
	Figures . . . . .	416
	Tables . . . . .	418
	Listings . . . . .	419
	ToDos . . . . .	428

# 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 access 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 22 for details.

**Debugging Support** The scripts can be debugged via IntelliJ IDEA. See 7.8 on page 375.

**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.<sup>1</sup>

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

---

<sup>1</sup>See chapter 7 on page 368 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.

**Attention:** You need at least one of the **License Options .WF or .MD** to develop scripts. The script project creation assistant will not be available otherwise. Please note that the execution of a script requires no specific license.

## 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 16.

## 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.

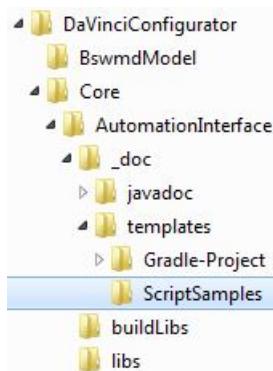


Figure 2.1: Script Samples location

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.

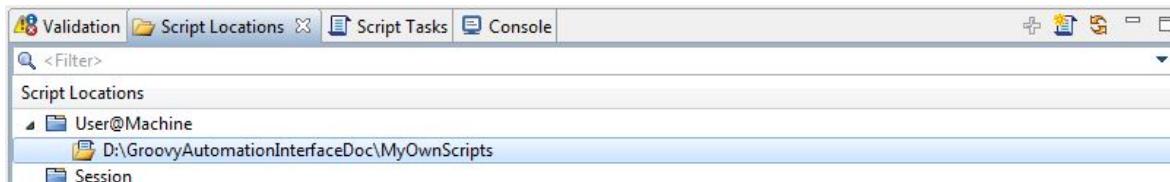


Figure 2.2: Script Locations View

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.

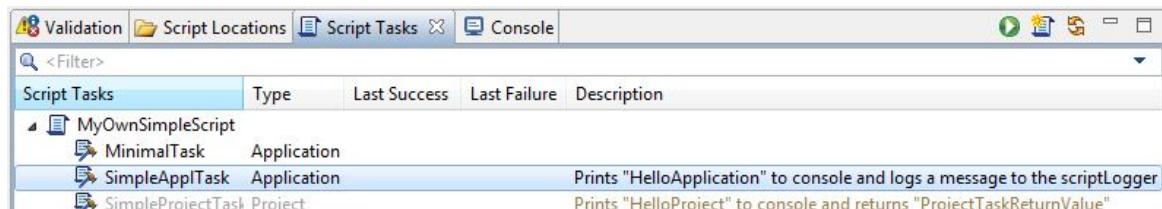


Figure 2.3: Script Tasks View

**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 28. 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. It is also called "Automation Script Project" throughout this document.

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 by the DaVinci Configurator.

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

- **SIP** containing a DaVinci Configurator 5
- **Java JDK:** 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 19.
- **IDE:** For the script project development the *recommended* IDE is *IntelliJ IDEA*. Please install IntelliJ IDEA as described in chapter 2.4.3 on page 19.
- **Build system:** To build the script project the build system Gradle is required. See chapter 2.4.4 on page 20 for installation instructions.

**Project Creation** Open the DaVinci Configurator inside your SIP. If not yet visible open the following Views via the View menu:

- Script Locations
- Script Tasks

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



Figure 2.4: Create New Script Project... Button

**Note:** If the button is not available, please make sure you have least one of the **License Options .WF or .MD** to develop scripts.

The **New Automation Script Project** dialog is opened. Click *Next* because you are reading the document.

On the second page first you have to select a Script template on which the new project shall be based on. Please select **Default Automation Project** and click *Next*.

On the third page **Project Settings**, please specify the following items:

- **Script Project Name**
  - Define a name for your new project.
- **Project Location**
  - Select a parent folder in which your project shall be created in.  
Note: A new folder with the project name is created in this folder.
- **Gradle Distribution URL**
  - Select one option:
    - \* **Gradle Default**
      - This will download the required Gradle build system. To use this option you need **internet access**.
    - \* **Custom URL**
      - Specify an URL to your own Gradle distribution.  
New settings are displayed to specify the path. To setup your own Gradle build system see 2.4.4 on page 20.
  - **Open IntelliJ IDEA**
    - Select this option if the project shall automatically be opened in IntelliJ IDEA after creation. In case IntelliJ IDEA is not installed on your system a warning will be issued.

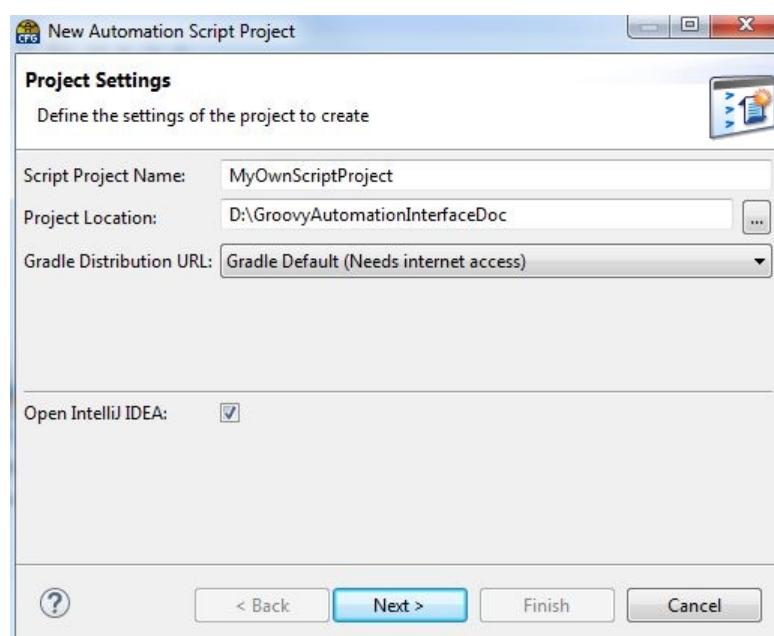


Figure 2.5: Project Settings

Proceed until the dialog is finished.

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

You can now modify the implementation according to your needs. For the AutomationInterface API Reference see chapter 4 on page 28. 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 on the following page.

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 7.5 on page 369. If you have trouble with IntelliJ, see 7.5.5 on page 373.

**Debugging** To debug the script project follow the instructions in chapter 7.8 on page 375.

**DaVinci Configurator views** 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 Default Automation Project sample script file contains one task that prints a "Hello-Application" string to the console. Run and check it as already described in 2.3 on page 14. If you have selected a different Script Sample the `MyScript.groovy` will contain the sample code.

The View **Script Locations** contains the path to the script project build folder containing the built `.jar` file.

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

### 2.4.1 Script Project Development

For more details to the development of a script project see chapter 7 on page 368.

### 2.4.2 Java JDK Setup

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

The DaVinci Tool Suite only supports 64 Bits, so make sure you get the x64 version.

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<sup>1</sup> a version that supports Java and Groovy and that is in the list of supported versions (see list 7.5.1 on page 369).

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**. Please set the value to a valid Java JDK ( see 2.4.2). **Do not** select a JRE.

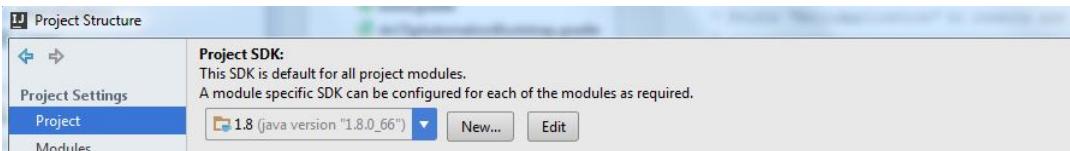


Figure 2.7: Project SDK Setting

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**. Please set the value to the same Java JDK as the Project SDK above. **Do not** select a JRE.

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.

<sup>1</sup> Vector-Internal: If you are inside of the Vector intranet, you could download it from: file:///vistrpesfs1/project2/DaVinci/Eclipse/Platform/CFG5/BuildComponents/IntelliJ

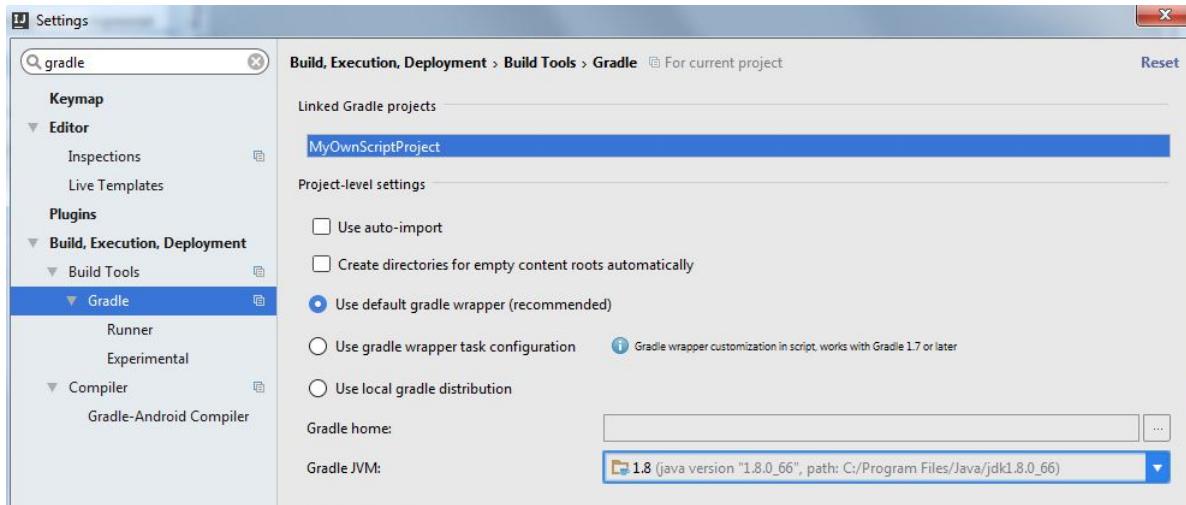


Figure 2.8: Gradle JVM Setting

#### 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 Gradle. If you are a Vector internal user you could also **skip** the Gradle installation.

If you want to use your own Gradle Build System install it on your system. The Gradle website provides the required download version for the Gradle Build System. Please **download the version 4.0.1**. See chapter 7.9 on page 376 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.

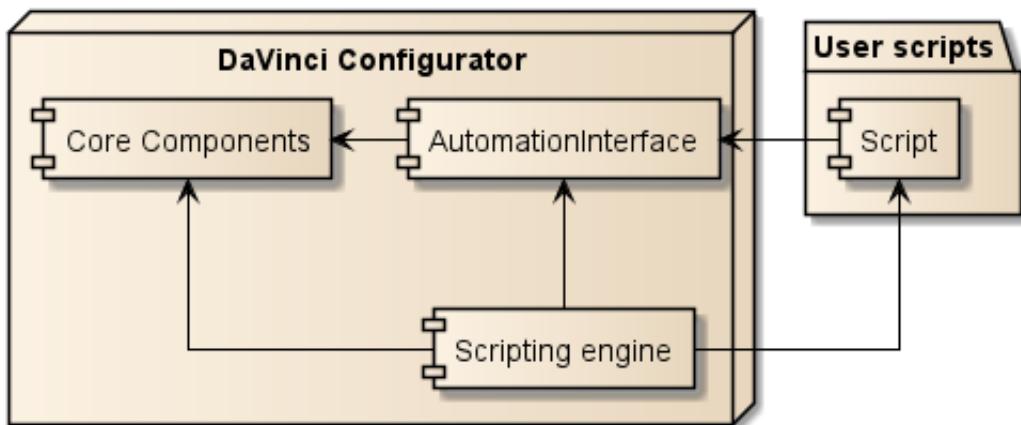


Figure 3.1: DaVinci Configurator components and interaction with scripts

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

- Stable API for script writers
  - 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 should not be used by any client code, nor shall a client call methods via reflection or other runtime techniques.

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

## 3.2 Languages

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

- Java<sup>1</sup>
- Groovy<sup>2</sup>

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<sup>3</sup>.

- Groovy syntax is 95%-compatible with Java<sup>4</sup>
- 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** The Groovy language 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**"<sup>5</sup> provides a comprehensive guide to Groovy programming language. It is written by the developers of Groovy.

<sup>1</sup><http://www.java.com> [2016-05-09]

<sup>2</sup><http://groovy-lang.org> [2016-05-09]

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

<sup>4</sup>Copied from [http://melix.github.io/blog/2010/07/27/experience\\_feedback\\_on\\_groovy.html](http://melix.github.io/blog/2010/07/27/experience_feedback_on_groovy.html) [2016-05-09]

<sup>5</sup>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]

### 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`.

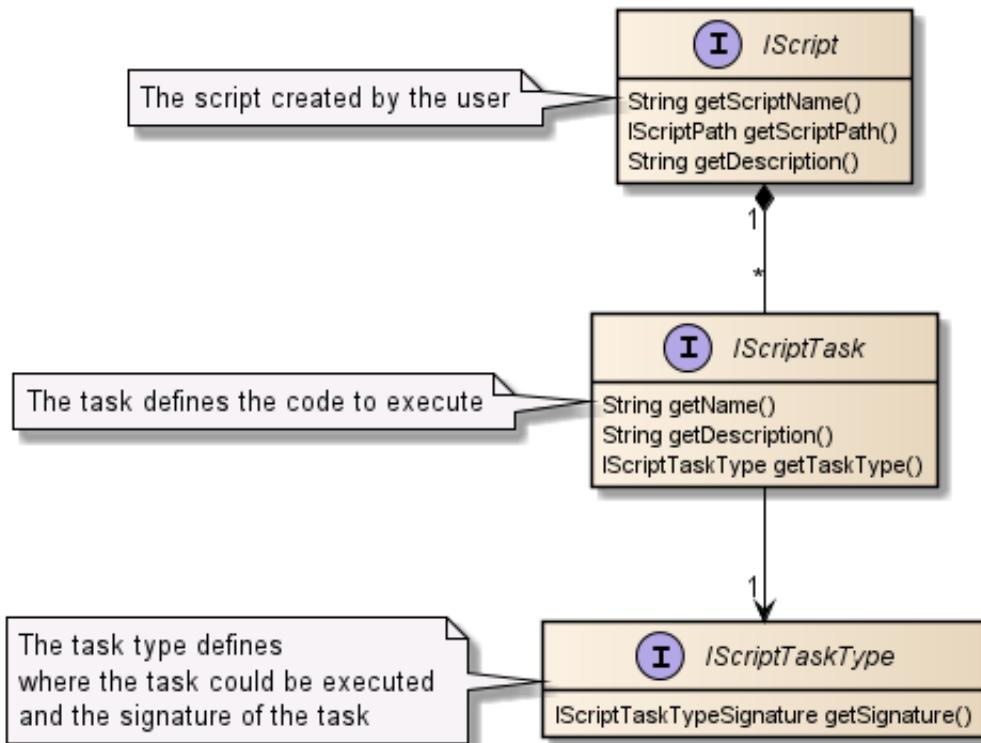


Figure 3.2: Structure of scripts and script tasks

You create the `IScript` and `IScriptTask` instance with the API described in chapter 4.2 on page 29.

The script task type (`IScriptTaskType`) defines where the task could be executed. It also defines the signature of the task's code `{}` block. See chapter 4.3 on page 33 for the available script task 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 368.

#### 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.

### 3.4.1 Internal Script Reload Behavior

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

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

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

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

## 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 368 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.

You need specific combinations of DaVinci Configurator licenses and options to develop and execute scripts.

For typical automation script tasks you need following licenses:

- Product license CFG PRO required for execution of script tasks
- Additional option .WF required for development and debugging

For generation script tasks for example you need a different license combination.

See chapter 4.3 on page 33 for details, which script task type requires which license.

Some script task may require different licenses or options during development or execution. It is also possible that the execution does not require any license at all. Normally you need more license options to develop scripts than you need to execute them.

## 3.7 Script Coding Conventions and Constraints

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

### Requirement Levels - Wording

- Shall: This word, or the terms "Mandatory", "Required" or "Must", mean that the rule or convention is an absolute requirement.
- Shall not: This word, or the terms "Must not" mean that the rule or convention is an absolute prohibition.
- Should: This word, or the adjective "Recommended", mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.

- Should not: This phrase, or the phrase "Not recommended" mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.
- May: This word, or the adjective "Optional", mean that an item is truly optional.

See also "RFC 2119: Key words for use in RFCs to Indicate Requirement Levels"<sup>6</sup>.

### 3.7.1 Usage of static fields

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

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

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

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

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

Listing 3.1: Static field memory leak

The use of static fields of the AutomationInterface is allowed.

### 3.7.2 Usage of Outer Closure Scope Variables

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

Example:

---

<sup>6</sup><https://www.ietf.org/rfc/rfc2119.txt>

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

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

Listing 3.2: Memory leak with closure variable

### 3.7.3 States over script task execution

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

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

If you need to cache data over multiple executions, see chapter 4.4.10 on page 58 for a solution.

### 3.7.4 Usage of Threads

A script task **shall not** create any Thread, Executor, ThreadPool or ForkJoinPool instances. If multithreading is required, the Automation API provides the corresponding methods.

A different thread will not provide any Automation APIs and will cause IllegalStateExceptions.

### 3.7.5 Usage of DaVinci Configurator private Classes Methods or Fields

A script task **should not** call or rely on any non published API or private (also package private) classes, methods or fields. You also should not use any reflection techniques to reflect about Configurator internal APIs. Otherwise it is not guaranteed that your script will work with other DaVinci Configurator versions. See 3.1 on page 21 for details about PublishedApi.

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

# 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.

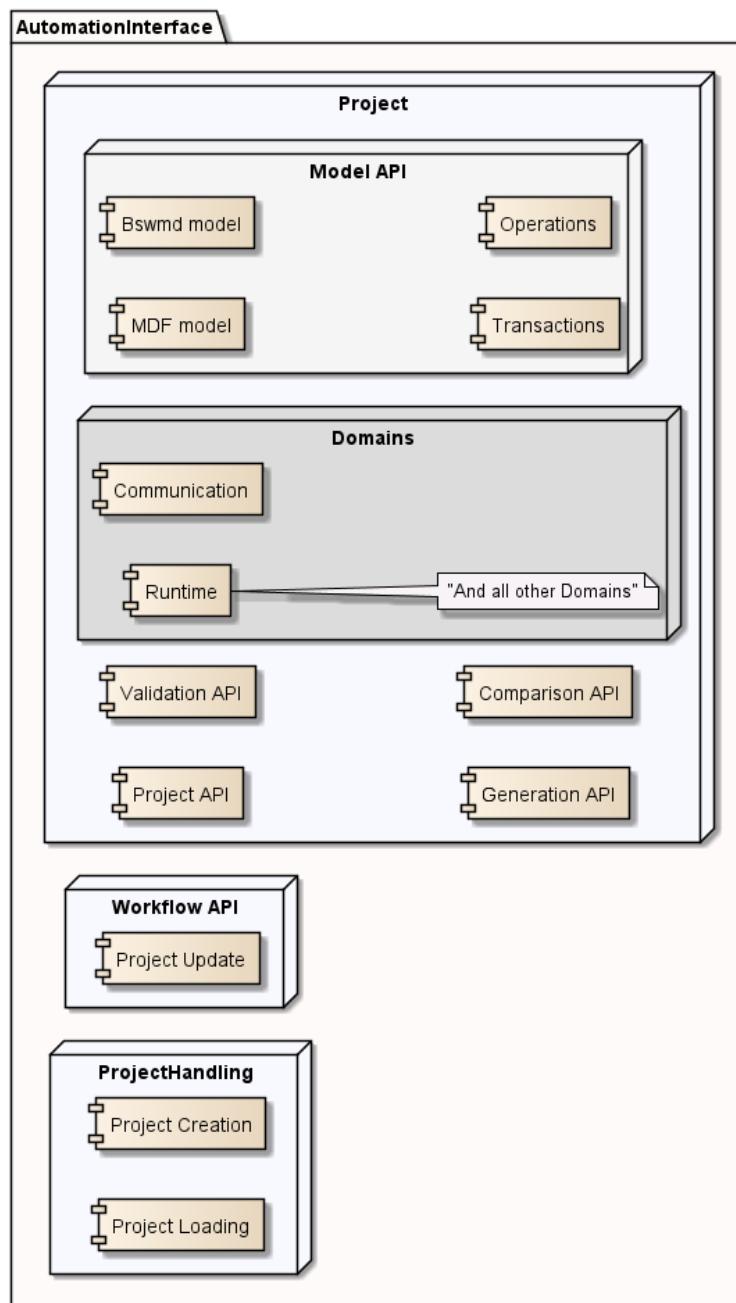


Figure 4.1: The API overview and containment structure

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 Project creation API is only usable outside of a loaded Project

## 4.2 Script Creation

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

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

### 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 33 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 33

```
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 in 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 25.

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

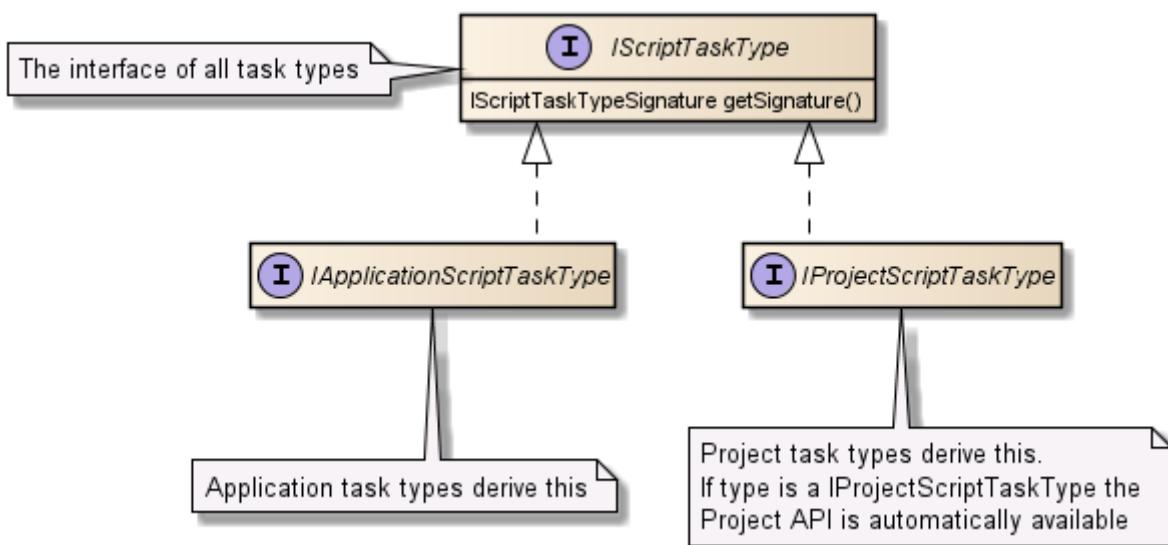


Figure 4.2: `IScriptTaskType` interfaces

### 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.

ScriptTaskType input parameters can get accessed via script by adding them to the code `{}` closure. See example 4.3 on the following page

#### 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

```

davinci {
    scriptTask(s: "SelectionTask", DV_EDITOR_MULTI_SELECTION) {
        code { List<MIOObject> selectedElements ->
            println selectedElements
        }
    }
}

```

Figure 4.3: ScriptTaskType input parameters in code closure

handling, or you want to handle multiple project at once.

<b>Name</b>	Application
<b>Code identifier</b>	DV_APPLICATION
<b>Task type interface</b>	IApplicationScriptTaskType
<b>Parameters</b>	None
<b>Return type</b>	None
<b>Execution</b>	Standalone
<b>Required license</b>	Development: .WF Execution: CFG 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.

<b>Name</b>	Project
<b>Code identifier</b>	DV_PROJECT
<b>Task type interface</b>	IProjectScriptTaskType
<b>Parameters</b>	None
<b>Return type</b>	None
<b>Execution</b>	Standalone
<b>Required license</b>	Development: .WF Execution: CFG PRO

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

<b>Name</b>	Module activation
<b>Code identifier</b>	DV_ON_MODULE_ACTIVATION
<b>Task type interface</b>	IProjectScriptTaskType
<b>Parameters</b>	MIModuleConfiguration moduleConfiguration
<b>Return type</b>	Void
<b>Execution</b>	Automatically during module activation
<b>Required license</b>	Development: .WF Execution: CFG PRO

**Module deactivation** The type DV\_ON\_MODULE\_DEACTIVATION allows the script to hook any Module Deactivation in a loaded project. Every DV\_ON\_MODULE\_DEACTIVATION task is automatically executed, when an "Deactivate Module" operation is executed. The script task is called before the module is deleted.

<b>Name</b>	Module deactivation
<b>Code identifier</b>	DV_ON_MODULE_DEACTIVATION
<b>Task type interface</b>	IProjectScriptTaskType
<b>Parameters</b>	MIModuleConfiguration moduleConfiguration
<b>Return type</b>	Void
<b>Execution</b>	Automatically during module deactivation
<b>Required license</b>	Development: .WF Execution: CFG 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.

<b>Name</b>	Editor selection
<b>Code identifier</b>	DV_EDITOR_SELECTION
<b>Task type interface</b>	IProjectScriptTaskType
<b>Parameters</b>	MIOBJECT selectedElement
<b>Return type</b>	Void
<b>Execution</b>	In context menu of an editor selection
<b>Required license</b>	Development: .WF Execution: CFG 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.

<b>Name</b>	Editor multiple selections
<b>Code identifier</b>	DV_EDITOR_MULTI_SELECTION
<b>Task type interface</b>	IProjectScriptTaskType
<b>Parameters</b>	List<MIOBJECT> selectedElements
<b>Return type</b>	Void
<b>Execution</b>	In context menu of an editor selection
<b>Required license</b>	Development: .WF Execution: CFG PRO

Those ScriptTaskTypes can be executed via selecting Configuration Elements in the editor.  
See usage: 4.4

Usage

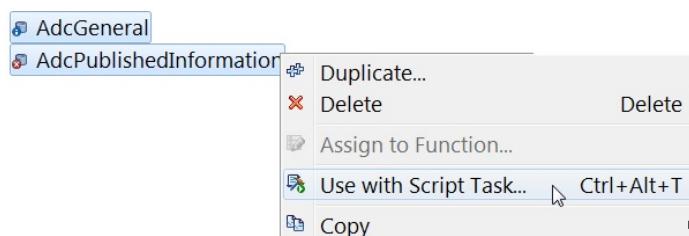


Figure 4.4: Access Editor Selection

Script

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

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

Listing 4.10: Usage of ScriptTaskType: DV\_EDITOR\_MULTI\_SELECTION

#### 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 Project Settings Editor, which references the script task.

Name	Generation Step
Code identifier	DV_GENERATION_STEP
Task type interface	IProjectScriptTaskType
Parameters	EGenerationPhaseType phase
	EGenerationProcessType processType
	IValidationResultSink resultSink
Return type	Void
Execution	Selected as GenerationStep in GenerationProcess
Required license	Development: .MD Execution: none

See chapter 4.7.2 on page 133 for usage samples.

**Custom Workflow Step** The type DV\_CUSTOM\_WORKFLOW\_STEP defines that the script task is executable as a CustomWorkflow step in the CustomWorkflow process. The user has to explicitly create an CustomWorkflow step in the Project Settings Editor, which references the script task.

Name	Custom Workflow Step
Code identifier	DV_CUSTOM_WORKFLOW_STEP
Task type interface	IProjectScriptTaskType
Parameters	None
Return type	Void
Execution	Selected as Custom Workflow Step in the Project Settings
Required license	Development: .WF Execution: CFG PRO

See chapter 4.7.2 on page 133 for usage samples.

**Generation Process Start** The type DV\_ON\_GENERATION\_START defines that the script task is automatically executed when the generation is started.

Name	Generation Process Start
Task type interface	IProjectScriptTaskType
Code identifier	DV_ON_GENERATION_START
Parameters	List<EGenerationPhaseType> generationPhases
	List<IGenerator> executedGenerators
Return type	Void
Execution	Automatically before GenerationProcess
Required license	Development: .MD Execution: none

See chapter 4.7.2 on page 133 for usage samples.

**Generation Process End** The type DV\_ON\_GENERATION\_END defines that the script task is automatically executed when the generation has finished.

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

See chapter 4.7.2 on page 133 for usage samples.

#### 4.3.1.5 Workflow Types

**Successful update workflow** The type DV\_ON\_SUCCESSFUL\_UPDATE\_WORKFLOW allows the script to access the loaded project during workflow after a successful update. Every DV\_ON\_SUCCESSFUL\_UPDATE\_WORKFLOW task is automatically executed, when the update has finished.

Because the update workflow is executed in a temporary environment you can access the path to the original project folder by reading the task argument.

```
code { path ->
    String originalProjectPath = path
}
```

Name	On successful update workflow
Code identifier	DV_ON_SUCCESSFUL_UPDATE_WORKFLOW
Task type interface	IProjectScriptTaskType
Return type	Void
Execution	Automatically after successful update workflow
Required license	Development: .WF Execution: CFG PRO

**Result file of the FilePrePocessing** The type DV\_ON\_FILE\_PREPROCESSING\_RESULT allows the script to modify the result file created in the update workflow by the FilePreProcessing. Every DV\_ON\_FILE\_PREPROCESSING\_RESULT task is automatically executed at the end of the FilePreprocessing. Scripts with this ScriptTaskType have only access to the SystemExtract model.

Name	On FilePreProcessing result
Code identifier	DV_ON_FILE_PREPROCESSING_RESULT
Task type interface	IProjectScriptTaskType
Parameters	None
Return type	Void
Execution	Automatically within the update workflow.
Required license	Development: .WF Execution: CFG PRO

**FilePreProcessing input file** The type DV\_ON\_FILE\_PREPROCESSING\_INPUT\_FILE allows the script to modify every input file which is used in the update workflow except of ProjectStandardConfiguration and legacy diagnostic files. Legacy communication files (.dbc,...) are converted in the first step and afterwards the result will be used for the script execution. Every DV\_ON\_FILE\_PREPROCESSING\_INPUT\_FILE task is automatically executed within the FilePreProcessing. Scripts with this ScriptTaskType will be executed for all Autosar input files and have only access to the model.

<b>Name</b>	On Autosar input file
<b>Code identifier</b>	DV_ON_FILE_PREPROCESSING_INPUT_FILE
<b>Task type interface</b>	IProjectScriptTaskType
<b>Parameters</b>	None
<b>Return type</b>	Void
<b>Execution</b>	Automatically within the FilePrePocessing of the update workflow.
<b>Required license</b>	Development: .WF Execution: CFG PRO

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

The `IScriptExecutionContext` holds the context of the execution:

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

The `IScriptExecutionContext` is also the entry point into every automation API, and provide access to the different API classes. The classes are described in their own chapters like `IProjectHandlingApiEntryPoint` 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", DV_APPLICATION){
    code{ // The IScriptExecutionContext is automatically active here
        // Call methods of the IScriptExecutionContext
        def logger = scriptLogger
        def temp = paths.tempFolder

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

Listing 4.11: 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 4.12: Access to automation API in Java clients by the `IScriptExecutionContext`

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

#### 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 33 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 4.13: Script task code block arguments

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

#### 4.4.2 Task Execution Sequence

The figure 4.5 on the next 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.

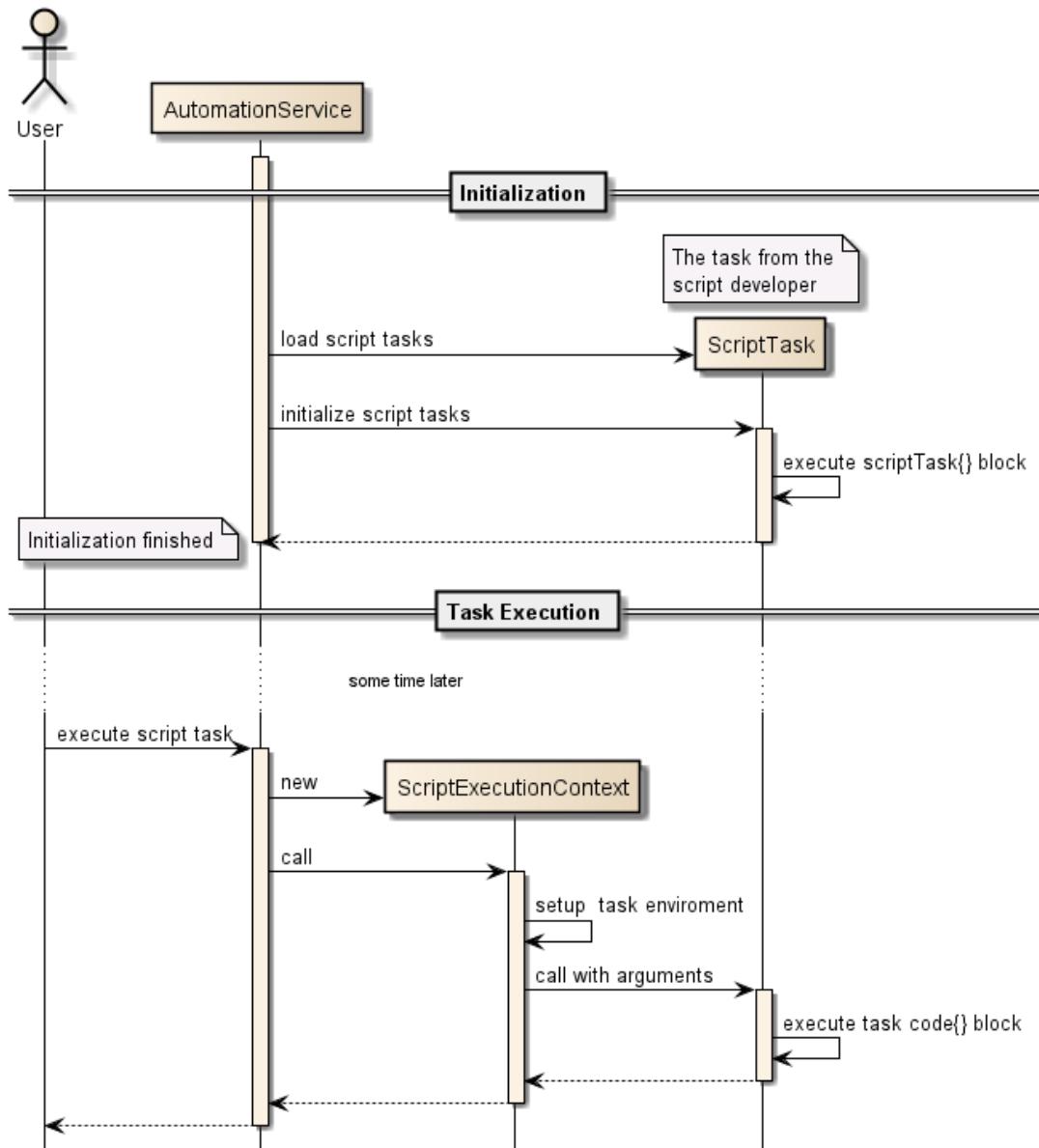


Figure 4.5: Script Task Execution Sequence

#### 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 4.14: 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 4.15: 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 previous 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 4.16: 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 preceding page.

There must be an active project to use this method. See chapter 4.5.2 on page 61 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 4.17: 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 42.

```
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 4.18: 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 42.

```
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 4.19: 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 4.20: Get the project output folder path

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

Listing 4.21: 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 4.22: Usage of the script logger

The `ILogger` also provides a formatting syntax for the format String. The syntax is `{Index-Number}` 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 4.23: 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 4.24: Usage of the script logger with Groovy GString message formatting

#### 4.4.5 User Interactions and Inputs

The `UserInteraction` and `UserInput` API provides methods to directly communicate with the user via `MessageBoxes`, `Input dialogs` or report progress of long running operations.

You should use the API only if you want do communicate directly with the user, because some API calls may block and wait for user interaction. So you should not use the API for batch jobs.

##### 4.4.5.1 UserInteraction

The `UserInteraction` API provides methods to display messages to the user directly. In UI mode the DaVinci Configurator will prompt a message box and will block until the user has acknowledged the message. In console (non UI) mode, the message is logged to the console in a user logger.

The user logger will display error, warnings and infos by default. The logger name will not be displayed.

The user interaction is good to display information where the user has to respond to immediately. Please use the feature sparingly, because users do not like to acknowledge multiple messages for a single script task execution.

The code block `userInteractions{}` provides the API inside of the block. The following methods can be used:

- `errorToUser()`
- `warnToUser()`
- `infoToUser()`
- `messageToUser(ELogLevel, Object)`

The severity (error, warning, info) will change the display (icons, text) of the message box. No other semantic is applied by the severity.

```
scriptTask("TaskName", DV_APPLICATION){
    code{

        userInteractions{
            warnToUser("Warning displayed to the user as message box")
        }

        // You could also write
        userInteractions.errorToUser("Error message for the user")
    }
}
```

Listing 4.25: UserInteraction from a script

#### 4.4.5.2 Progress Indication

If you perform long running operations in a script task, you should display some progress to the user, otherwise the user may cancel the whole execution. The progress API will display the progress of the currently running script task by the information provided by the script code.

The method `progress(String, Closure)` displays the passed message in progress information dialog and executed the code block. So the message is displayed until the code block has finished.

```
userInteractions.progress("The text for the user"){
    // Here the code of the long running operation
}
```

Listing 4.26: Display progress to the user

You could also nest multiple `progress()` calls. When a progress block is left, the parent progress text will be displayed again.

```
userInteractions{
    progress("The text for the user"){
        // Here the code of the long running operation
        progress("Inner operation"){
            // Here code of inner operation
        }
    }
    progress("Second operation"){
        //Code of the second operation
    }
}
```

Listing 4.27: Display progress to the user nested

The method `progress(String, int, Closure)` updates the progress information for the user with the message, during the code is running with work ticks.

It also indicates progress in the progress bar, but you have to set the total amount of work. The total work will be taken from the parent and sets the remaining work for the code block.

The root script task always starts with `totalWork` of 1000 ticks, so you have to consume 1000 ticks to fill the progress bar.

```
userInteractions{
    progress("The text for the user", 1000){
        worked(100)
        progress("Inner operation", 400){
            //100 ticks
            worked(200)
            //300 ticks
        }
        // half reached - 500 ticks
        progress("Inner operation", 200){
            worked(100)
            // 600 ticks reached
        }
        // 700 ticks reached
    }
    // All 1000 ticks done, the progress bar is now full!
}
```

Listing 4.28: Display progress to the user with progress bar work

**Eclipse API** You can also use the underlying Eclipse API to fine grain control the progress bar and information data. To do this use the `getProgressMonitor()` method to retrieve the Eclipse `SubMonitor`. See also the Eclipse API `SubMonitor.setWorkRemaining(int)` to scale your own work to different values (also more than 1000 ticks).

## 4.4.6 Script Error Handling

### 4.4.6.1 Script Exceptions

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

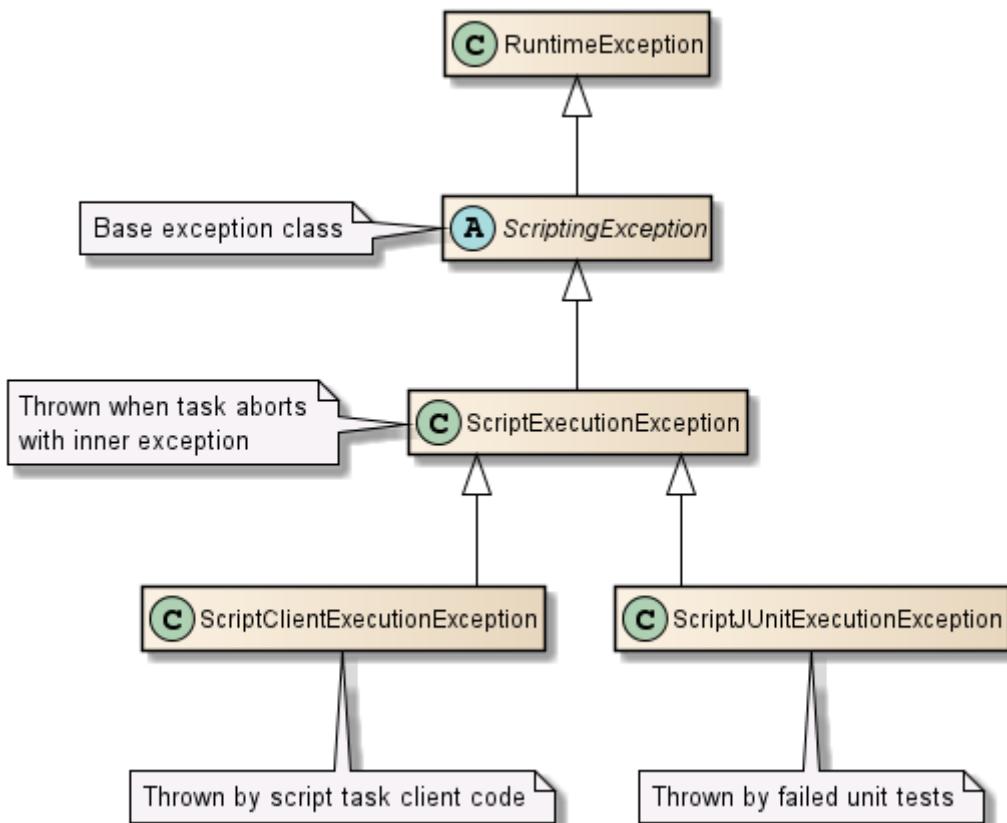


Figure 4.6: `ScriptingException` and sub types

### 4.4.6.2 Script Task Abortion by Exception

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

```

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

Listing 4.29: 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 4.30: 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.

#### 4.4.6.3 Unhandled Exceptions from Tasks

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

#### 4.4.7 User defined Classes and Methods

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

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

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

Listing 4.31: 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 4.32: 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 4.33: Using your own defined method with a daVinci block

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

block. The chapter 4.4.8 describes how to use the Automation API for your own defined classes and methods.

#### 4.4.8 Usage of Automation API in own defined Classes and Methods

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

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

##### 4.4.8.1 Access the Automation API like the Script code{} Block

The `ScriptApi.scriptCode(Closure)` method provides access to all automation APIs the same way as 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.

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

Listing 4.34: `ScriptApi.scriptCode()` usage in own method

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

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

Listing 4.35: `ScriptApi.scriptCode()` usage in own method

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

##### 4.4.8.2 Access the Project API of the current active Project

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

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

Listing 4.36: 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 4.37: ScriptApi.activeProject() usage in own method

#### 4.4.9 User defined Script Task Arguments

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

Example: `--count 25` or `-s`

Possible valueTypes are:

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

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

```
/*
Cmd Script Call Syntax
--scriptTask "MyScript:TaskName" --taskArgs "TaskName" "-p"
*/
scriptTask("TaskName"){
    def procArg = newUserDefinedArgument("p", Void, "Enables the processing of
        ...")
    code{
        if(procArg.hasValue){
            scriptLogger.info "The argument -p was defined"
        }
    }
}
```

Listing 4.38: Script task UserDefined argument with no value

```
/*
Cmd Script Call Syntax
--scriptTask "MyScript:TaskName" --taskArgs "TaskName" "--count 25 --name Test"
*/
scriptTask("TaskName"){
    def countArg = newUserDefinedArgument("count", Integer,
                                         "The amount of elements to create")

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

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

Listing 4.39: Define and use script task user defined arguments from commandline

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

Listing 4.40: Script task UserDefined argument with default value

```
/*
Cmd Script Call Syntax
--scriptTask "MyScript:TaskName" --taskArgs "TaskName" "--multiArg "argValueOne
    " --multiArg "argValueTwo""
*/
scriptTask("TaskName"){
    def multiArg = newUserDefinedArgument("multiArg", String, "Help text ...")

    code{

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

Listing 4.41: Script task UserDefined argument with multiple values

#### 4.4.9.1 User defined Argument Validators

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

There are also default validators available, like:

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

Please see chapter 4.15.1 on page 309 for more available validators.

```
/*
Cmd Script Call Syntax
--scriptTask "MyScript:TaskName" --taskArgs "TaskName" "-p InvalidAsrShortName"
*/
import com.vector.cfg.business.Constraints

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

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

Listing 4.42: Script task UserDefined argument with predefined validator

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

```
/*
Cmd Script Call Syntax
--scriptTask "MyScript:TaskName" --taskArgs "TaskName" "-p"
*/
scriptTask("TaskName"){

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

    code{
    }
}
```

Listing 4.43: Script task UserDefined argument with own validator

#### 4.4.9.2 Call Script Task with Task Arguments from Commandline

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>" "<ArgName ArgValue>"  
          |           | _____ ||  
          | _____ |
```

Examples:

Single Argument without value:

E.g. `--taskArgs "MyTask" "-s"`

Single Argument with value:

E.g. `--taskArgs "MyTask" "--count 25"`

Multiple Arguments with and without value:

E.g. `--taskArgs "MyTask" "-s --count 25"`

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

If there are multiple tasks apply for the following syntax:

```
Syntax: --taskArgs "<TaskName>" "<ArgName ArgValue>"  
          "<TaskName2>" "<ArgName ArgValue>"
```

```
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, you can specify the full qualified name with script name

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

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

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

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

```
--scriptTaskHelp taskName
```

#### 4.4.10 Stateful Script Tasks

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

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

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

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

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

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

Listing 4.44: `executionData` - Cache and retrieve data during one script task execution

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

**Caution:** The data is saved globally so the usage of the `sessionData` can lead to memory leaks or `OutOfMemoryErrors`. You have to take care not to store too much memory in the `sessionData`.

The DaVinci Configurator will also free the `sessionData`, when the system run low on free memory. So you have to deal with the fact, that the `sessionData` was freed, when the script task getting executed again. But the data is not deallocated during a running execution.

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

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

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

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

Listing 4.45: `sessionData` - Cache and retrieve data over multiple script task executions

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

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

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

The listing below describes the usage of the API:

```
scriptTask("TaskName"){  
    code{  
        def val  
        // The sessionData and executionData have the same API  
  
        // You have multiple ways to set a value  
        executionData.myCacheId = "VALUE"  
        executionData.set("myCacheId", "VALUE")  
        executionData["myCacheId"] = "VALUE"  
        // Or with classes for a service locator pattern  
        executionData.set(Integer.class, 50) // Possible for any Class  
        executionData[Integer] = 50  
  
        // There are the same ways to retrieve the values  
        val = executionData.myCacheId  
        val = executionData.get("myCacheId")  
        val = executionData["myCacheId"]  
        // Or with classes for a service locator pattern  
        val = executionData.get(Integer.class)  
        val = executionData[Integer]  
  
        // You can also ask if the property exists  
        boolean exists = executionData.has("myCacheId")  
    }  
}
```

Listing 4.46: `sessionData` and `executionData` syntax samples

#### 4.4.11 ScriptAccess - Calling ScriptTasks

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

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

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

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

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

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

Listing 4.47: Call another script task from a script task

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

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

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

Listing 4.48: Call another script task with arguments

## 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 4.49: 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 4.50: 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 4.7 describes the behavior to search for the active project of a script task.

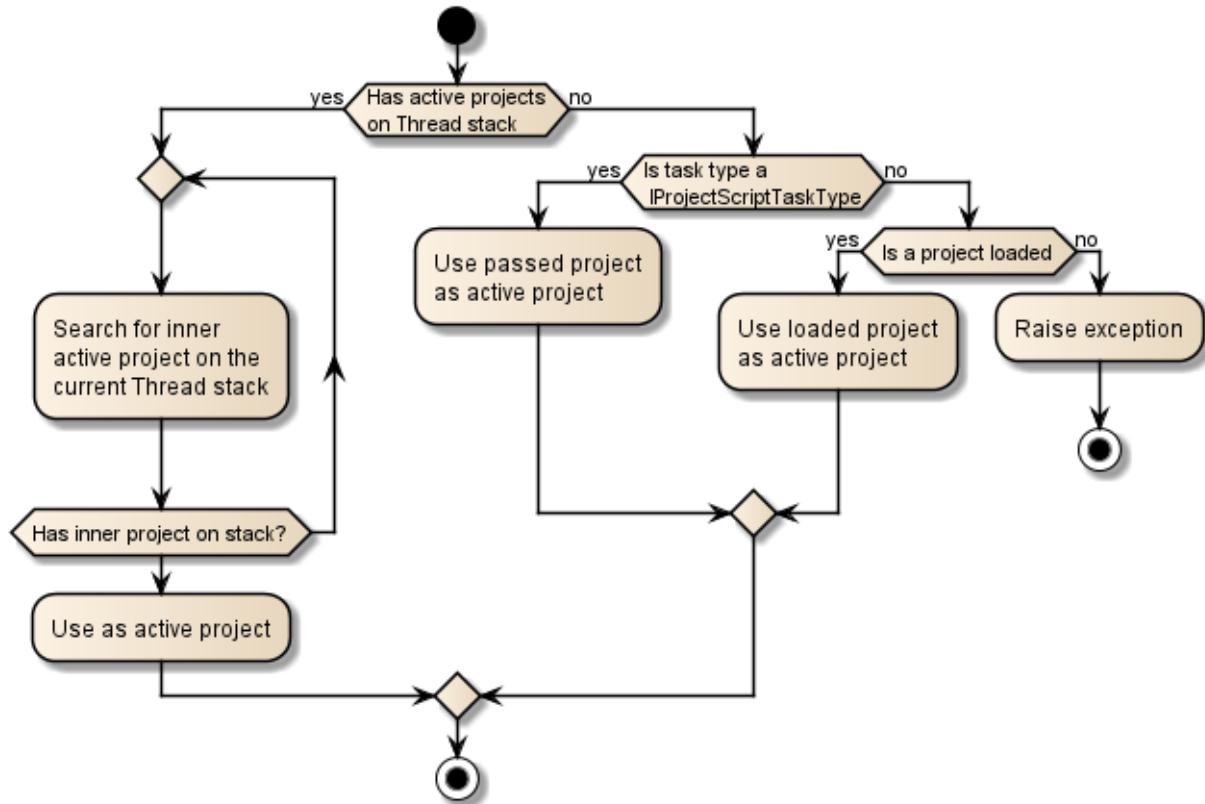


Figure 4.7: Search for active project in `getActiveProject()`

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 4.51: 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 4.52: 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 Accessing the project search

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

```

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

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

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

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

```

Listing 4.53: Using the search API

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

a `SearchApiException` will be thrown.

Example:

```
* What went wrong:  
> The task execution has thrown an exception.  
> com.vector.cfg.datamining.pai.impl.SearchApiException: Failure: -- line 1 col 25: EOF expected  
>  
> AutoCompletion fo [Query: Definition==EcucGeneral wrong Grammer]  
> - isEroneous: true  
> - isQuickSearch: false  
> - ErrorIndex: 24  
> - ValidPart: Definition==EcucGeneral  
> - suggestions:  
> <NONE>Possible extensions:  
> Tokens:  
> Class: TokenCondition , Kind: 63 , Value: Definition , Start: 0, End: 9  
> Class: TokenOperator , Kind: 35 , Value: == , Start: 10, End: 11  
> Class: TokenValue , Kind: 1 , Value: EcucGeneral , Start: 12, End: 22  
> Class: TokenError , Kind: -1 , Value: wrong Grammer , Start: 24, End: 36
```

Figure 4.8: SearchApi Exception Message

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

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

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

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

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

#### 4.5.4 Accessing Project Settings

##### 4.5.4.1 Target Project Settings

The `IProjectSettingsApiEntryPoint` enables querying or modifying the target project settings. The following section describes how the project settings of a project can be accessed and modified.

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

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

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

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

```

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

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

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

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

                }
            }
        }
    } // code
} // scriptTask

```

Listing 4.54: Access and modify Project Settings - Variant 1

**Module** The module which supported Derivatives shall be retrieved.

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

**Module** The module which the derivative shall be set for.

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

**Module** The module which derivative setting shall be retrieved.

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

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

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

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

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

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

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

#### 4.5.4.2 UseCase Project Settings

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

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

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

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

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

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

Listing 4.55: Access and modify Use Project Settings UseCases

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

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

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

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

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

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

#### 4.5.5 Creating a new Project

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

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

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

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

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

Listing 4.56: Creating a new project (mandatory parameters only)

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

```

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

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

            general {
                author 'projectAuthor'
                version '0.9'
            }

            postBuild {
                loadable true
                selectable true
            }

            folders.ecucFileStructure = ONE_FILE_PER_MODULE
            folders.moduleFilesFolder = 'App1/GenData'
            folders.templatesFolder = 'App1/Source'

            target.vVIRTUALtargetSupport = false
            daVinciDeveloper.createDaVinciDeveloperWorkspace = false
        }
    }
}

```

Listing 4.57: Creating a new project (with some optional parameters)

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

#### 4.5.5.1 Mandatory Settings

**Project Name** Specify the name newly created project with `setProjectName(String)`. The name given here is postfixed with ".dpa" for the new project's .dpa file.

The following constraints apply:

- `Constraints.IS_VALID_PROJECT_NAME` 4.15.1 on page 309

**Project Folder** Specify the folder in which to create the new project in with `setProjectFolder(Object)`. The value given here is converted to `Path` using the converter `ScriptConverters.TO_SCRIPT_PATH` 4.15.2 on page 310.

The following constraints apply:

- `Constraints.IS_CREATABLE_FOLDER` 4.15.1 on page 309

#### 4.5.5.2 General Settings

Use `getGeneral()` or `general(Closure)` to specify the new project's general settings. The provided settings are defined in ICreateProjectGeneralApi.

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

The following constraints apply:

- `Constraints.IS_NON_EMPTY_STRING` 4.15.1 on page 309

**Version** The version for the new project can be specified with `setVersion(Object)`. This is an optional parameter defaulting to "1.0" if the parameter is not provided explicitly. The value given here is converted to `IVersion` using `ScriptConverters.TO_VERSION` 4.15.2 on page 310.

The following constraints apply:

- `Constraints.IS_NOT_NULL` 4.15.1 on page 309

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

The following constraints apply:

- `Constraints.IS_NOT_NULL` 4.15.1 on page 309

**Start Menu Entries** `setCreateStartMenuEntries(boolean)` defines 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.

#### 4.5.5.3 Target Settings

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

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

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

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

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

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

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

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

**vVIRTUALtarget Support** `setvVIRTUALtargetSupport(boolean)` specifies 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. See also `ICreateProjectApi.getVirtualTarget()` and `ICreateProjectVirtualTargetApi` for specifying further details (path to vVIRTUALtarget project, ...).

The following constraints apply:

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

#### 4.5.5.4 Post Build Settings

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

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

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

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

#### 4.5.5.5 Folders Settings

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

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

**Module Files Folder** Set the module files folder for the new project with `setModuleFilesFolder(Object)`. 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 `ScriptConverters.TO_PATH` 4.15.2 on page 310. Normally a relative path (to be interpreted relative to the project folder) should be given here.

The following constraints apply:

- `Constraints.IS_CREATABLE_FOLDER` 4.15.1 on page 309

**Templates Folder** Set the templates folder for the new project with `setTemplatesFolder(Object)`. 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 `ScriptConverters.TO_PATH` 4.15.2 on page 310. Normally a relative path (to be interpreted relative to the project folder) should be given here.

The following constraints apply:

- `Constraints.IS_CREATABLE_FOLDER` 4.15.1 on page 309

**Service Components Folder** Set the service component files folder for the new project with `setServiceComponentFilesFolder(Object)`. 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 `ScriptConverters.TO_PATH` 4.15.2 on page 310. Normally a relative path (to be interpreted relative to the project folder) should be given here.

The following constraints apply:

- `Constraints.IS_CREATABLE_FOLDER` 4.15.1 on page 309

**Application Components Folder** Set the application component files folder for the new project with `setApplicationComponentFilesFolder(Object)`. 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 `ScriptConverters.TO_PATH` 4.15.2 on page 310. Normally a relative path (to be interpreted relative to the project folder) should be given here.

The following constraints apply:

- `Constraints.IS_CREATABLE_FOLDER` 4.15.1 on page 309

**Log Files Folder** Set the log files folder for the new project with `setLogFilesFolder(Object)`. 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 `ScriptConverters.TO_PATH` 4.15.2 on page 310. Normally a relative path (to be interpreted relative to the project folder) should be given here.

The following constraints apply:

- `Constraints.IS_CREATABLE_FOLDER` 4.15.1 on page 309

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

The folder object passed to the method is converted to Path using `ScriptConverters.TO_PATH` 4.15.2 on page 310. Normally a relative path (to be interpreted relative to the project folder) should be given here.

The following constraints apply:

- `Constraints.IS_CREATABLE_FOLDER` 4.15.1 on page 309

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

The value given here is converted to Path using `ScriptConverters.TO_PATH` 4.15.2 on page 310. Normally a relative path (to be interpreted relative to the project folder) should be given here.

The following constraints apply:

- `Constraints.IS_CREATABLE_FOLDER` 4.15.1 on page 309

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

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

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

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

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

#### 4.5.5.6 DaVinci Developer Settings

Use `getDaVinciDeveloper()` to specify the new project's DaVinci Developer settings.

`ICreateProjectDaVinciDeveloperApi` contains the methods for specifying the new project's DaVinci Developer settings.

**Create DEV Workspace** `setCreateDaVinciDeveloperWorkspace(boolean)` specifies 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** Set the DaVinci Developer executable for the new project with `setDaVinciDeveloperExecutable(Object)`. 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 `ScriptConverters.TO_SCRIPT_PATH` 4.15.2 on page 310.

The following constraints apply:

- The path must points to a compatible DaVinci Developer executable (DaVinciDEV.exe).

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

The value given here is converted to Path using `ScriptConverters.TO_PATH` 4.15.2 on page 310. Normally a relative path (to be interpreted relative to the project folder) should be given here.

The following constraints apply:

- `Constraints.IS_DCF_FILE` 4.15.1 on page 310
- `Constraints.IS_CREATABLE_FOLDER` 4.15.1 on page 309 (applies to the parent Path of the given Path to the DaVinci Developer executable)

**Import Mode Preset** `setUseImportModePreset(boolean)` specifies 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** `setLockCreatedObjects(boolean)` specifies 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.

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

#### 4.5.5.7 vVIRTUALtarget Settings

Use `getVirtualTarget()` to specify the new project's vVIRTUALtarget settings. The vVIRTUALtarget support may not be available depending on the purchased license.

```

scriptTask('ProjectCreation', DV_APPLICATION) {
    code {
        def prjFolder = paths.resolveTempPath('projectFolder')

        def newProject = projects.createProject {
            projectName 'tpVttFullyCustom'
            projectFolder prjFolder

            target {
                vVIRTUALtargetSupport = true
            }

            virtualTarget {
                createVirtualTargetProjectFile = true
                virtualTargetExecutable = getCustomVttExe()
                virtualTargetProject = new File(prjFolder.toFile(), "/MyVtt/custom.
                    vttproj")
            }
        }

        scriptLogger.info("Project created and saved")
    }
}

```

Listing 4.58: Creating a new project with custom VTT settings

**Create vVIRTUALtarget project file** `setCreateVirtualTargetProjectFile(boolean)` specifies whether or not to create a vVIRTUALtarget project file for the new project. This is an optional parameter defaulting to `true`. However the vVIRTUALtarget project file is only created when `ICreateProjectTargetApi.vVIRTUALtargetSupport(boolean)` evaluates to `true`.

**vVIRTUALtarget Project** Set the path to the vVIRTUALtarget project (\*.vttproj) for the new project with `setVirtualTargetProject(Object)`. This is an optional parameter defaulting to `'./Config/VTT/ProjectName.vttproj'` if the parameter is not provided explicitly. See also `ICreateProjectTargetApi.setvVIRTUALtargetSupport(boolean)` and `ICreateProjectVirtualTargetApi.setCreateVirtualTargetProjectFile(boolean)` at which both have to be `true` to force the creation of the vVIRTUALtarget project.

The value given here is converted to Path using `ScriptConverters.TO_SCRIPT_PATH` 4.15.2 on page 310.

**vVIRTUALtarget Executable** Set the vVIRTUALtarget executable (VttCmd.exe) for the new project with `setVirtualTargetExecutable(Object)`. This is an optional parameter defaulting to the location of the currently registered installation (if there is any) if the parameter is not provided explicitly.

The value given here is converted to Path using `ScriptConverters.TO_SCRIPT_PATH` 4.15.2 on page 310.

#### 4.5.5.8 TA Tool Suite Settings

Use `getTaToolSuite()` to specify the new project's TA Tool Suite settings.

```
scriptTask('taskName', DV_APPLICATION) {
    code {
        def prjFolder = paths.resolveTempPath('projectFolder')

        def newProject = projects.createProject {
            projectName 'tp'
            projectFolder prjFolder

            taToolSuite {
                createTaToolSuiteWorkspace true
                taToolSuiteExecutable getMyTaToolSuiteExe()
            }
        }

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

        return newProject.getDpaProjectFilePath()
    }
}
```

Listing 4.59: Creating a new TA Tool Suite workspace

**Create TA Tool Suite Workspace** `setCreateTaToolSuiteWorkspace(boolean)` specifies whether or not to create a TA Tool Suite workspace for the new project. This is an optional parameter defaulting to `false`.

**TA Tool Suite Workspace** Set the TA Tool Suite workspace for the new project with `setTaToolSuiteWorkspace(Object)`. This is an optional parameter defaulting to `"./Config/TA/"` if the parameter is not provided explicitly.

The value given here is converted to Path using `ScriptConverters.TO_SCRIPT_PATH` 4.15.2 on page 310. Normally a relative path (to be interpreted relative to the project folder) should be given here.

The following constraints apply:

- `Constraints.IS_CREATABLE_FOLDER` 4.15.1 on page 309 (applies to the given Path to the TA Tool Suite workspace)

**TA Tool Suite Executable** Set the TA Tool Suite executable for the new project with `setTaToolSuiteExecutable(Object)`.

The value given here is converted to Path using `ScriptConverters.TO_SCRIPT_PATH` 4.15.2 on page 310.

The following constraints apply:

- The path must point to a compatible TA Tool Suite executable (TA Tool Suite.exe).

### 4.5.6 Opening an existing Project

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

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

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

The `Object` given as .dpa file is converted to Path using `ScriptConverters.TO_SCRIPT_PATH` 4.15.2 on page 310

```
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 4.60: Opening a project from .dpa file

#### 4.5.6.1 Parameterized Project Load

You can also configure how a Dpa project is loaded, e.g. by disabling the generators.

The method `parameterizeProjectLoad(Closure)` returns a handle on the project specified by the given `Closure`. Using the `IOpenDpaProjectApi`, the `Closure` may further customize the project's opening procedure.

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

```
scriptTask('taskName', DV_APPLICATION) {
    code {
        def project = projects.parameterizeProjectLoad {
            // replace 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 4.61: Parameterizing the project open procedure

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

**DPA File** The method `setDpaFile(Object)` sets the .dpa file of the project to be opened. The value given here is converted to Path using `ScriptConverters.TO_SCRIPT_PATH` 4.15.2 on page 310.

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

**Validation** `setEnabledValidation(boolean)` specifies whether or not to activate validation for the opened project.

#### 4.5.6.2 Open Project Details

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

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

- `IProjectHandlingApi.createProject(Closure)` 4.5.5 on page 67
- `IProjectHandlingApi.parameterizeProjectLoad(Closure)` 4.5.6 on the preceding 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.

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

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

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

#### 4.5.7 Create Ecu Configuration Report

The `ICreateEcucReportApiEntryPoint` enables creating an Ecu Configuration report. The following section describes how an Ecu Configuration report can be created. If no report settings are set the defaults will take place.

Use `getCreateEcucReport()` or `createEcucReport(Closure)` to specify the ecuc report settings. The report gets created after the closure gets closed.

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

See examples below to create an Ecu Configuration report.

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

Listing 4.62: Create Ecu Configuration Report with default settings

```
scriptTask("TestEcucReportCreation", DV_PROJECT) {
    code { outputAsPath ->
        activeProject {

            // Use closure to get access to the ecuc report settings
            createEcucReport {

                // Set only those options for which to change the default value
                outputFilePath           outputAsPath
                // default: <ProjectFolder>/Log/EcucReport.html
                addAnnotations           true          // default: true
                addPlatformFunction      true          // default: true
                createXmlReportFile     false         // default: false
                overwriteExistingReportFile true         // default: true
                openReportAfterGeneration false        // default: true
            }
        }
    } // code
} // scriptTask
```

Listing 4.63: Create Ecu Configuration Report

**outputFilePath** *Default:* Log directory e.g. <ProjectFolder>\Log\EcucReport.html - Sets the output file path.

**addAnnotations** *Default:* true - Specify if user annotations should be shown in the report.

**addPlatformFunctions** *Default:* true - Specify if platform functions should be shown in the report.

**createXmlReportFile** *Default:* false - Specify if a xml report file should be created beside the html report file.

**overwriteExistingReportFile** *Default:* true - Specify if an existing report file should be overwritten.

**openReportAfterGeneration** *Default:* true - Specify if the created report file should be opened afterwards.

#### 4.5.8 Create Support Request Package

The `createSupportRequestPackage(Closure)` provides the possibility to create a support request package. It also provides options to configure the creation of the SRP, means to include required project informations.

The following example shows the minimal setup to create a SRP.

```
scriptTask("CreateSupportRequestPackageTest", DV_PROJECT) {
    code {
        activeProject {
            createSupportRequestPackage {
                setFirstName "Winnie"
                setLastName "Puh"
                setEMail("winnie.puh@honey.moon")

                // If no project information will be set,
                // it will likewise not include any project information in the
                .zip file.

                // If no saveLocation will be set, it will use the default one.
                // The default will be the project folder.
            }
        }
    }
}
```

Listing 4.64: Minimal Example of Create Support Request Package

Use `setFirstName(String)` or `firstName(String)` to set the first name.

**Mandatory**

Use `setLastName(String)` or `lastName(String)` to set the last name.

**Mandatory**

Use `setEMail(String)` or `eMail(String)` to set the email.

**Mandatory**

Use `setProjectInformation(Collection)` or `projectInformation(Collection)` to manually add the necessary project informations in the .zip. This is optional. If no `projectInformation(Collection)` is used, the system will only add the mandatory project informations.

**Optional**

- PC Info
- SIP Info
- Tool Version Info

Use this `setSaveLocation(Path)` or `saveLocation(Path)` to define the output location. If nothing is set, the system will use the project location.

**Optional**

```
scriptTask("CreateSupportRequestPackageTest", DV_PROJECT) {
code {
    activeProject {
        createSupportRequestPackage {
            setFirstName "Winnie"
            setLastName "Puh"
            setEMail "winnie.puh@honey.moon"

            setProjectInformation( [ EProjectInformationApi.
                ALL_PROJECT_RELEVANT_DATA ] )
            setSaveLocation Paths.get(saveLocString)
        }
    }
}
}
```

Listing 4.65: Create Support Request Package with all project informations

```
scriptTask("CreateSupportRequestPackageTest", DV_PROJECT) {
code {
    activeProject {
        createSupportRequestPackage {
            setFirstName "Winnie"
            setLastName "Puh"
            setEMail("winnie.puh@honey.moon")

            def projectInformations = [
                EProjectInformationApi.DAVINCI_WORKSPACE,
                EProjectInformationApi.PROJECT_SYSTEM_FILES,
                EProjectInformationApi.PROJECT_LOG_FILES,
                EProjectInformationApi.BSW_INTERNAL_BEHAVIOR_FILES,
                EProjectInformationApi.MEASUREMENT_AND_CALIBRATION_FILES,
                EProjectInformationApi.SCRIPT_TASKS,
                EProjectInformationApi.ECU_CONFIGURATION_FILES,
                EProjectInformationApi.PROJECT_INPUT_FILES,
                EProjectInformationApi.GENERATED_FILES,
                EProjectInformationApi.TOOL_LOG_FILES,
                EProjectInformationApi.APPLICATION_COMPONENT_FILES,
                EProjectInformationApi.SERVICE_COMPONENT_FILES ]

            setProjectInformation(projectInformations)
            setSaveLocation Paths.get(saveLocString)
        }
    }
}
}
```

Listing 4.66: Create Support Request Package with individual project informations

#### 4.5.9 Saving a Project

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

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

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

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

Listing 4.67: Opening, modifying and saving a project

### 4.5.10 Opening AUTOSAR Files as Project

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

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

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

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

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

- Update workflow: You can't update a non existing project
- Validation: The validation is disabled by default
- Generation: The generators are not loaded by default

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

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

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

Listing 4.68: Opening Arxml files as project

**Arxml Files** Add `arxml` files to load with the method `arxmlFiles(Collection)`. Multiple files and method calls are allowed. The given values are converted to `Path` instances using `ScriptConverters.TO_SCRIPT_PATH` 4.15.2 on page 310.

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

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

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

#### 4.5.10.1 Raw AUTOSAR models as Project

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

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

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

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

**Empty AUTOSAR model** The `emptyAutosarModel(String, Closure)` method creates a new empty AUTOSAR model, only containing one `MIARPackage` created by this method with the path `AsrPath`.

The passed AUTOSAR version defines the version of the AUTOSAR model, the version is specified in the format "4.2.1" or "4.0.3", ...

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

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

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

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

Listing 4.69: Create an empty AUTOSAR model

## 4.6 Model

### 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 323 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 based on the AUTOSAR MetaModel which can be found for example on the AUTOSAR website. In 5.1 on page 323 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 94 for more details.

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

```

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

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

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

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

```

Listing 4.70: Read with BswmdModel objects starting with a module DefRef (no type declaration)

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

```

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

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

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

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

```

Listing 4.71: Read with BswmdModel objects starting with a module class (strong typing)

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

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

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

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

Listing 4.72: Read with `BswmdModel` objects with closure argument

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

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

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

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

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

Listing 4.73: Read with `BswmdModel` object for an MDF model object

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

be used. See chapter 4.6.3.4 on page 96 and 4.6.3.6 on page 97 for more details:

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

scriptTask("TaskName"){
    code {

       .IContainer ecucGen = bswmdModel(sipDefRef.EcucGeneral).single

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

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

Listing 4.74: Read with BswmdModel objects with the untyped model (DefRefAPI)

#### 4.6.2.2 Write the ActiveEcuc

This section provides some typical examples as a brief introduction for writing the Ecuc by means of the BswmdModel. See chapter 4.6.3.3 on page 95 for more details.

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

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

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

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

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

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

Listing 4.75: Write with BswmdModel required/optional objects

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

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

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

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

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

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

Listing 4.76: Write with BswmdModel multiple objects

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

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

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

Listing 4.77: Write with BswmdModel - Duplicate a container

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

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

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

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

Listing 4.78: Write with BswmdModel - Delete elements

#### 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 99 for more details.

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

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

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

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

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

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

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

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

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

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

Listing 4.80: Read system description starting with an AUTOSAR path in property style

#### 4.6.2.4 Write the SystemDescription

Writing the system description looks quite similar to the reading, but you have to use methods like (see chapter 4.6.4.3 on page 104 for more details):

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

You have to open a transaction before you can modify the MDF model, see chapter 4.6.6 on page 118 for details.

The following samples show the different types of write API:

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

Listing 4.81: Changing a simple property of an MIVariableDataPrototype

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

Listing 4.82: Creating non-existing member by navigating into its content with OrCreate()

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

        adminData {
            sdg.createAndAdd(MISdg) {
                gid = "NewGidValue"
            }
        }

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

Listing 4.83: Creating new members of child lists with createAndAdd() by type

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

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

        def dataElement2 = dataList.byNameOrCreate("NewName")

        assert dataElement == dataElement2
    }
}
```

Listing 4.84: Updating existing members of child lists with byNameOrCreate() by type

### 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, Parameter, References). The BswmdModel is automatically generated from the SIP of the DaVinci Configurator.

**You should** use the BswmdModel whenever possible to access Ecuc elements of the AUTOSAR model. For accessing the Ecuc elements with the BswmdModel, see chapter 4.6.3.2.

For a detailed description of the BswmdModel, see chapter 5.3.1 on page 337.

#### 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 4.85: BswmdModel usage with import

#### 4.6.3.2 Reading with BswmdModel

The `bswmdModel()` 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:

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

- `bswmdModel(MIHasDefinition, WrappedTypedDefRef)` returns the BswmdModel instance for the provided MDF model instance.
- `bswmdModel(Class, String)` searches all objects with the specified class and the matching path, see `IMdfModelApi.mdfModel(String)` or chapter 4.6.4.2 on page 102 for details.

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

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

Examples:

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

Listing 4.86: Read with BswmdModel the EcuC module configuration

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

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

Listing 4.87: Read with BswmdModel the EcuC module configuration with DefRef

For more usage samples please see chapter 4.6.2.1 on page 84.

#### 4.6.3.3 Writing with BswmdModel

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

Example:

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

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

Listing 4.88: Write with BswmdModel the EcucGeneral container

For more usage samples please see chapter 4.6.2.2 on page 88.

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

See 5.3.1.9 on page 345 for more details to the BswmdModel write API.

#### 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 `DefRefs` 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"]

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

Listing 4.89: Usage of the `sipDefRef` API to retrieve `DefRefs` in script files

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

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

Listing 4.90: Check if a definition exists in the SIP

#### 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 94 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 4.91: Usage of generated DefRefs form the bswmd model

#### 4.6.3.6 Untyped Model with the DefRef API

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

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

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

scriptTask("TaskName"){
    code {

        GIModuleConfiguration ecuc = bswmdModel(sipDefRef.EcuC).single

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

       .IContainer ecucPduCollection = ecuc.getSubContainer(sipDefRef["/
            MICROSAR/EcuC/EcucPduCollection"])

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

       .IContainer pdu = pdus.get(0)

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

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

Listing 4.92: Usage of the untyped BswmdModel with SipDefRefs

See chapter 5.3.1.2 on page 339 for more details.

#### 4.6.3.7 Switching from Domain Models to BswmdModel

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

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

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

def canControllerBswmd = canDomainModel.bswmdModel(CanController.DefRef)

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

Listing 4.93: Switch from a domain model object to the corresponding BswmdModel object

#### 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 323.

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 Reading the MDF Model

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

The following types of entry points are provided here:

- `mdfModel(TypedAsrPath)` searches an object with the specified AUTOSAR path
- `mdfModel(TypedDefRef)` searches all objects with the specified definition
- `mdfModel(Class)` searches all objects with the specified model type (meta class)
- `mdfModel(String)` searches for model elements with by different properties, see 4.6.4.2 on page 102 for details.
- `mdfModel(MIObject, String)` searches for model elements by giving a root element and a relative path, see 4.6.4.2 on page 103 for details.

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

```

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

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

    // Or use the Closure syntax to navigate

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

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

    saveProject()
}

```

Listing 4.94: Navigate into an MDF object starting with an AUTOSAR path

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

Naming of the interface classes to create the type safe AUTOSAR path is described in chapter 5.1 on page 323.

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

```

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

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

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

```

Listing 4.95: Find an MDF object and retrieve some content data

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

```
mdfModel(asrPath) {
    int count = 0
    swDataDefProps {
        // 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 4.96: Navigating deeply into an MDF object with nested closures

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

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

Listing 4.97: Ignoring non-existing member closures

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

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

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

Listing 4.98: Get a MIReferrable child object by name

### Lists containing Referrables

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

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

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

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

// byName() with type and shortname
def data2 = prototype.dataElement.byName(MIVariableDataPrototype, "DeSignal2")

// getAt() with shortname
def data3 = prototype.dataElement["DeSignal3"]
```

Listing 4.99: Retrieve child from list with byName()

### Lists containing Parameters and Containers

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

#### 4.6.4.2 Reading the MDF Model by String

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

- AUTOSAR path, see `mdfModel(AsrPath)`, if the path begins with an '/' and the model element is no definition object (`MIParamConfMultiplicity`)
  - Example: `/ActiveEcuc/MyCan/MyContainer`
- ObjectLink, see `AsrObjectLink`, if the path begins with an '/' and the model element is no definition object (type `MIParamConfMultiplicity`)
  - Example: `/ActiveEcuc/MyCan/MyContainer[0:ParameterDef]`
- Definition path, see `mdfModel(DefRef)`, if the path begins with an '/'
  - Example: `/MICROSAR/Can2`
- Relative path, see `mdfModel(MIOBJECT, String)`, the relative path may not start with an '/'. See 4.6.4.2 on the following page for more details.
- MICROSAR QUERY, if the path begins with "`msrq:`". The defined Microsar Query, filters the configuration elements by the given arbitrary filter code. The filter must be evaluable to a `String`, `Boolean` or `Pattern`. The Microsar Query can be used for modules, containers and parameters. See 4.6.4.2 on page 104 for more details.
- AUTOSAR path relative to the ActiveEcuc package, if it does not begin with an '/'

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

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

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

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

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

Listing 4.100: Get elements with `mdfModel(String)`

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

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

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

Listing 4.101: Read definitions elements with a relative path using the `mdfModel`

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

scriptTask("mdfModel", DV_PROJECT){
    code {
        // Reading an activeEcuc element
        def asrPath = AsrPath.create("/ActiveEcuc/Can/CanConfigSet",
            MIContainer)
        def root = mdfModel(asrPath)
        def reqElem = mdfModel(root, "
            ECU_T_CTP_1_NWT_CTP_CANH_ak72ea5qpue3dstlfi5v43l2z_090525f9_Rx_Ext")
            .getFirst()
    }
}
```

Listing 4.102: Read activeEcuc elements with a relative path using the mdfModel

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

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

#### Examples:

- `/MICROSAR/Crc/CrcGeneral{ true }`
- `/MICROSAR/Crc/CrcGeneral{ ~"[\\w]*[1231]\\$" }`
- `/MICROSAR/Crc/CrcGeneral{ "CrcGeneral" }`
- `/MICROSAR/Crc/CrcGeneral{ it.getName() == "CrcGeneral" }`
- `/MICROSAR/Crc/CrcGeneral{ elem -> elem.getName() == "CrcGeneral" }`
- `/MICROSAR/Crc/CrcGeneral{ getName() == "CrcGeneral" }`
- `/MICROSAR/Crc/CrcGeneral{ it.getName().contains("CrcGeneral") }`
- `/[ANY]/Crc/CrcGeneral/{ true }`

#### 4.6.4.3 Writing the MDF Model

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

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

You have to open a transaction before you can modify the MDF model, see chapter 4.6.6 on page 118 for details about transactions.

#### 4.6.4.4 Simple Property Changes

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

- String
- Enums
- Integer
- Double

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

Listing 4.103: Changing a simple property of an MIVariableDataPrototype

#### 4.6.4.5 Creating single Child Members (0:1)

For single child members (0:1), the automation API provides and additional method for the getter `get<Element>OrCreate()` for convenient child object creation. The methods will create the element, instead of returning `null`.

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

Listing 4.104: Creating non-existing member by navigating into its content with OrCreate()

If the compile time child type is not instatiatable, you have to provide the concrete type by `get<Element>OrCreate(Class childType)`.

```
transaction{
    // The asrPath points to an MIVariableDataPrototype
    mdfModel(asrPath) {
        introductionOrCreate(MIBlockLevelContent) { docuBlock ->
            assert docuBlock instanceof MIBlockLevelContent
        }
    }
}
```

Listing 4.105: Creating child member by navigating into its content with OrCreate() with type

#### 4.6.4.6 Creating and adding Child List Members (0:\*)

For child list members, the automation API provides many `createAndAdd()` methods for convenient child object creation. These method will always create the element, regardless if the same element (e.g. same ShortName) already exists.

If you want to update element see the chapter 4.6.4.7 on page 108.

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

        adminData {
            sdg.createAndAdd(MISdg) {
                gid = "NewGidValue"
            }
        }

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

Listing 4.106: Creating new members of child lists with `createAndAdd()` by type

These methods are available — but be aware that not all of these methods are available for all child lists. Adding parameters, for example, is only permitted in the parameter child list of an `MIContainer` instance.

#### All Lists:

- The method `createAndAdd()` creates a new MDF object of the lists content type and appends it to this list. If the type is not instantiatable the method will thrown a `ModelException`. The new object is finally returned.
- The method `createAndAdd(Closure)` creates a new MDF object of the lists content type and appends it to this list. If the type is not instantiatable the method will thrown a `ModelException`. Then the closure is executed with the new object as closure parameter. The new object is finally returned.
- The method `createAndAdd(Class)` creates a new MDF object of the specified type and appends it to this list. The new object is finally returned.
- The method `createAndAdd(Class, Closure)` creates a new MDF object of the specified type and appends it to this list. Then the closure is executed with the new object as closure parameter. The new object is finally returned.
- The method `createAndAdd(Class, Integer)` creates a new MDF object of the specified type and inserts it to this list at the specified index position. The new object is finally returned.
- The method `createAndAdd(Class, Integer, Closure)` creates a new MDF object of the specified type and inserts it to this list at the specified index position. Then the closure is executed with the new object as closure parameter. The new object is finally returned.

### Lists containing Referrables

- The method `createAndAdd(String)` creates a new child with the specified shortname and appends it to this list. The new object is finally returned. The used type is the lists content type. If the type is not instantiatable the method will thrown a `ModelException`.
- The method `createAndAdd(String, Closure)` creates a new `MIReferrable` with the specified shortname and appends it to this list. Then the closure is executed with the new object as closure parameter. The new object is finally returned. The used type is the lists content type. If the type is not instantiatable the method will thrown a `ModelException`.
- The method `createAndAdd(Class, String)` creates a new `MIReferrable` with the specified type and shortname and appends it to this list. The new object is finally returned.
- The method `createAndAdd(Class, String, Closure)` creates a new `MIReferrable` with the specified type and shortname and appends it to this list. Then the closure is executed with the new object as closure parameter. The new object is finally returned.
- The method `createAndAdd(Class, String, Integer)` creates a new `MIReferrable` with the specified type and shortname and inserts it to this list at the specified index position. The new object is finally returned.
- The method `createAndAdd(Class, String, Integer, Closure)` creates a new `MIReferrable` with the specified type and shortname and inserts it to this list at the specified index position. Then the closure is executed with the new object as closure parameter. The new object is finally returned.

### Lists containing Parameters and Containers

- The method `createAndAdd(TypedDefRef)` creates a new Ecuc object (container or parameter) with the specified definition and appends it to this list. The new object is finally returned.
- The method `createAndAdd(TypedDefRef, Closure)` creates a new Ecuc object (container or parameter) with the specified definition and appends it to this list. Then the closure is executed with the new object as closure parameter. The new object is finally returned.
- The method `createAndAdd(TypedDefRef, Integer)` creates a new Ecuc object (container or parameter) with the specified definition and inserts it to this list at the specified index position. The new object is finally returned.
- The method `createAndAdd(TypedDefRef, Integer, Closure)` creates a new Ecuc object (container or parameter) with the specified definition and inserts it to this list at the specified index position. Then the closure is executed with the new object as closure parameter. The new object is finally returned.
- The method `byDefOrCreate(TypedDefRef)` retrieves the child with the passed definition, if the child exists and has a definition multiplicity of 0:1 or 1:1. Otherwise a new child is created. The definition and shortname (using the definition name) are automatically set before returning the new child. So this method will always create a new child if the upper multiplicity is greater than 1.

### Lists containing Containers

- The method `createAndAdd(TypedDefRef, String)` creates a new container with the specified definition and shortname and appends it to this list. The new container is finally returned.
- The method `createAndAdd(TypedDefRef, String, Closure)` creates a new container with the specified definition and shortname and appends it to this list. Then the closure is executed with the new container as closure parameter. The new container is finally returned.
- The method `createAndAdd(TypedDefRef, String, Integer)` creates a new container with the specified definition and shortname and inserts it to this list at the specified index position. The new container is finally returned.
- The method `createAndAdd(TypedDefRef, String, Integer, Closure)` creates a new container with the specified definition and shortname and inserts it to this list at the specified index position. Then the closure is executed with the new container as closure parameter. The new container is finally returned.

#### 4.6.4.7 Updating existing Elements

For child list members, the automation API provides many `byNameOrCreate()` methods for convenient child object update and creation on demand. These method will create the element if id does not exists, or return the existing element.

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

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

        def dataElement2 = dataList.byNameOrCreate("NewName")

        assert dataElement == dataElement2
    }
}
```

Listing 4.107: Updating existing members of child lists with `byNameOrCreate()` by type

These methods are available — but be aware that not all of these methods are available for all child lists. Updating container, for example, is only permitted in the parameter child list of an `MIContainer` instance.

#### Lists containing Referrables

- The method `byNameOrCreate(String)` retrieves the child with the passed shortname, or creates the child, if it does not exist. The shortname is automatically set before returning the new child.
- The method `byNameOrCreate(TypedDefRef, String, Closure)` retrieves the child with the passed shortname, or creates the child, if it does not exist. The shortname is automatically set before returning the new child. Then the closure is executed with the child as closure parameter. The child is finally returned.

- The method `byNameOrCreate(Class, String)` retrieves the child with the passed type and shortname, or creates the child, if it does not exist. The shortname is automatically set before returning the new child.
- The method `byNameOrCreate(Class, String, Closure)` retrieves the child with the passed type and shortname, or creates the child, if it does not exist. The shortname is automatically set before returning the new child. Then the closure is executed with the child as closure parameter. The child is finally returned.

### Lists containing Containers

- The method `byNameOrCreate(TypedDefRef, String)` retrieves the child with the passed definition and shortname, or creates the child, if it does not exist. The definition and shortname are automatically set before returning the new child.
- The method `byNameOrCreate(TypedDefRef, String, Closure)` retrieves the child with the passed definition and shortname, or creates the child, if it does not exist. The definition and shortname are automatically set before returning the new child. Then the closure is executed with the child as closure parameter. The child is finally returned.

#### 4.6.4.8 Deleting Model Objects

The method `delete(MIObject)` deletes the specified object from the model. This method must be called inside a transaction because it changes the model content.

Special case: If this method is being called on an active module configuration, it actually calls `IOperations.deactivateModuleConfiguration(MIModuleConfiguration)` to deactivate the module correctly.

```
// MIParameterValue param = ...

transaction {
    assert !param.isDeleted()
    param.delete()
    assert param.isDeleted()
}
```

Listing 4.108: Delete a parameter instance

The method `moRemove()` does the same as `delete()`. For details about model object deletion and access to deleted objects, read section 5.1.7.4 on page 327 ff.

**IsDeleted** The `isDeleted(MIObject)` method returns `true` if the specified object has been deleted (removed) from the MDF model, or is invisible in the current active `IModelView`.

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

Listing 4.109: Check is a model instance is deleted

Note: The return value is dependent on the current active thread and the current active `IModelView` in this thread!

The method `moIsRemoved()` does the same as `isDeleted()`.

#### 4.6.4.9 Duplicating Model Objects

The `duplicate(MIObject)` method copies (clones) a complete MDF model sub-tree and adds it as child below the same parent.

- The source object must have a parent. The clone will be added to the same MDF feature below the same parent then
- AUTOSAR UUIDs will not be cloned. The clone will contain new UUIDs to guarantee unambiguousness

This method can clone any model sub-tree, also see `IOperations.deepClone(MIObject, MIObject)` for details.

Note: This operation must be executed inside of a transaction.

```
// MIContainer container = ...
transaction {
    def newCont = container.duplicate()
    // The duplicated container newCont
}
```

Listing 4.110: Duplicates a container under the same parent

#### 4.6.4.10 Special properties and extensions

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

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

Listing 4.111: Get the AsrPath of an MIReferrable instance

See chapter 5.4.2 on page 349 for more details about AsrPaths.

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

```
MIPParameterValue param = ...
AsrObjectLink link = param.asrObjectLink
```

Listing 4.112: Get the AsrObjectLink of an AUTOSAR model instance

See chapter 5.4.3 on page 350 for more details about AsrObjectLinks.

**defRef** The `getDefRef()` method returns the `DefRef` of the model object.

```
MIPParameterValue param = ...
DefRef defRef = param.defRef
```

Listing 4.113: Get the DefRef of an Ecuc model instance

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

```
MIPParameterValue param = ...
DefRef newDefinition = ...
param.defRef = newDefinition
```

Listing 4.114: Set the DefRef of an Ecuc model instance

If the specified `DefRef` has a wildcard, the parameter must have a parent to calculate the absolute definition path - otherwise a `ModelCeHasNoParentException` will be thrown.

If it has no wildcard and no parent, the absolute definition path of the defRef will be used.

If the parameter has a parent or and parents definition does not match the defRefs parent definition, this method fails with `InconsistentParentDefinitionException`.

The `MIContainer.setDefRef(DefRef)` method sets the definition of this container to the defRef.

See chapter 5.4.4 on page 350 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.

```
MIPParameterValue param = ...
IParameterStatePublished state = param.ceState
```

Listing 4.115: Get the CeState of an Ecuc parameter instance

See chapter 5.4.5 on page 353 for more details about the CeState.

**ceState - User-defined Flag** The method `isUserDefined()` returns `true`, if the ecuc configuration element like parameters is flagged as user-defined.

```
MIPParameterValue param = ...
def flag = param.ceState.userDefined
```

Listing 4.116: Retrieve the user-defined flag of an Ecuc parameter in Groovy

The method `setUserDefined(boolean)` sets or removes the user-defined flag of a ecuc configuration element like parameters.

Note: This method must executed inside a transaction because it modifies the model state.

```
MIPParameterValue param = ...
transaction {
    param.ceState.userDefined = true
}
```

Listing 4.117: Set an Ecuc parameter instance to user defined

**EcuConfigurationAccess and EcucDefinitionAccess** The Groovy automation interface also provides special access methods for Ecuc elements (module configurations, container and parameter) to the

- EcuConfigurationAccess (see 5.5.1 on page 355)
- EcucDefinitionAccess (see 5.5.2 on page 359)

The `getEcucDefinition()` method returns the `IEcucDefinition` of the model object.

```
MIPParameterValue param = ...
IEcucDefinition definition = param.ecucDefinition
```

Listing 4.118: Get the `IEcucDefinition` of an Ecuc model instance

The `getEcuConfiguration()` method returns the `IEcucHasDefinition` of the model object.

```
MIPParameterValue param = ...
IEcucHasDefinition cfg = param.ecuConfiguration
```

Listing 4.119: Get the `IEcucHasDefinition` of an Ecuc model instance

These methods are the same as for bswmd model objects.

#### 4.6.4.11 Reverse Reference Resolution - `ReferencesPointingToMe`

You can resolve all references in the MDF model in the reverse direction, so you can start at a reference target and navigate to all references which point to the reference target.

**referencesPointingToMe** The `getReferencesPointingToMe()` method returns all reference parameters in the active ecuc pointing to specified target (`MIReferrable`) object. It returns an empty collection if the target object is invisible or removed.

The `getReferencesPointingToMe(DefRef)` method returns all reference parameters in the active ecuc with the specified definition (`DefRef`) pointing to the specified target (`MIReferrable`) object. It returns an empty collection if the target object is invisible, removed or the specified definition is `null`.

```
List<MIReferenceValue> refs = container.referencesPointingToMe
//Or
DefRef refDefRef = // DefRef to reference parameter
def refByDef = container.getReferencesPointingToMe(refDefRef)
```

Listing 4.120: `referencesPointingToMe` sample

**systemDescriptionObjectsPointingToMe** The method `getSystemDescriptionObjectsPointingToMe()` returns all objects located in the system description which are parent objects of references pointing to the specified target. It returns an empty collection if the object is invisible or removed.

```
List<MIObject> references =
systemDescElement.systemDescriptionObjectsPointingToMe
```

Listing 4.121: `systemDescriptionObjectsPointingToMe` sample

#### 4.6.4.12 Derived Containers

The `MIHasContainer.getDerived()` method provides access to derived container information. The method returns a `IDerivedElementInfo` object corresponding to the model element.

The `IDerivedElementInfo` can be used to retrieve information about element or delete it:

- `getRemovedDerivedSubContainers()`: Retrieves the removed children, which could be used to restore them the children
- `isDerived()`: returns `true` if the element is derived
- `delete()`: deletes the element regardless if it is derived or not

```
container.derived.isDerived()

// Or
container.derived{
    boolean isDerivedFlag = isDerived()
    def removedList = getRemovedDerivedSubContainers()
}
```

Listing 4.122: Derived Container API access

**Deletion of Derived Containers** The method `delete()` deletes the `MIContainer` regardless, if it is derived or not.

This method behaves as follows:

- If the container is a derived container it calls the derived container deletion operation to delete it.
- All other containers will be deleted by means of `moRemove()`

```
transaction{
    container.derived.delete()
}
```

Listing 4.123: Delete a derived container unconditionally

#### 4.6.4.13 AUTOSAR Root Object

The `getAUTOSAR()` method returns the AUTOSAR root object (the root object of the MDF model tree of AUTOSAR data).

```
MIAUTOSAR root = AUTOSAR
```

Listing 4.124: Get the AUTOSAR root object

#### 4.6.4.14 ActiveEcuC

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

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

Listing 4.125: Get the active Ecuc and all module configurations

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

Listing 4.126: Iterate over all module configurations

```
activeEcuc {
    // Parameter type is IActiveEcuc
    ecuc ->

    def defRef = DefRef.create(EDefRefWildcard.AUTOSAR, "EcuC")

    // Get the modules as Collection<MIConfiguration>
    Collection foundModules = ecuc.modules(defRef)
    assert !foundModules.empty
}
```

Listing 4.127: Get module configurations by definition

#### 4.6.4.15 DefRef based Access to Containers and Parameters

The Groovy automation interface for the MDF model provides some overloaded access methods for

- MIConfiguration.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<MIPParameterValue>
    def cpuType = containers.first.parameter(CPUType.DefRef)

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

Listing 4.128: Get subContainers and parameters by definition

#### 4.6.4.16 Ecuc Parameter and Reference Value Access

The Groovy automation interface also provides special access methods for Ecuc parameter values. These methods are implemented as extensions of the Ecuc parameter and value types and can therefore be called directly at the parameter or reference instance.

##### Value Checks

- `hasValue(MIPParameterValue)`  
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 4.129: 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 `AritheticException` 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 `ArithmaticException` 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 4.130: 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 4.131: Get reference parameter value

#### 4.6.5 SystemDescription Access

The systemDescription API provides methods to retrieve system description data like the path to the flat extract or the flat map instance.

It is grouped by the AUTOSAR version. So the `getAutosar4()` methods provides access to AUTOSAR 4 model elements.

The `getPaths()` provides common paths to elements like:

- FlatMap path
- FlatExtract path
- FlatCompositionType path

```
AsrPath flatExtractPath = systemDescription.paths.flatExtractPath
AsrPath flatMapPath = systemDescription.paths.flatMapPath
```

Listing 4.132: Get the FlatExtract and FlatMap paths by the SystemDescription API

```
systemDescription{
    autosar4{
        flatExtract.ifPresent{ theFlatExtract ->
            // do something with the flatMap
        }
    }
}
// Or in property style
def theFlatExtractOpt = systemDescription.autosar4.flatExtract
if(theFlatExtractOpt){
    def theFlatExtract = theFlatExtractOpt.get()
}
```

Listing 4.133: Get FlatExtract instance by the SystemDescription API

## 4.6.6 Transactions

Model changes must always be executed within a transaction. The automation API provides some simple means to execute transactions.

For details about transactions read 5.1.7 on page 327.

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

Listing 4.134: Execute a transaction

```
scriptTask("TaskName", DV_PROJECT){
    code {
        transaction("Transaction name") {
            // The transactionName property is available inside a transaction
            String name = transactionName
        }
    }
}
```

Listing 4.135: 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.

**Nested Transactions** If you open a transaction inside of a transaction the inner transaction is ignored and it is as no transaction call was done. So be aware that nested transactions are no real transaction, which leads to the fact the these nested transactions can not be undone.

If you want to know whether a transaction is already running, see the transactions API below.

### 4.6.6.1 Transactions API

The Transactions API with the keyword `transactions` provides access to running transactions or the transaction history.

You can use method `isTransactionRunning()` to check if a transaction is currently running. The method returns `true`, if a transaction is running in the current `Thread`.

```

scriptTask("TaskName", DV_PROJECT){
    code {
        // Switch to the transactions API
        transactions{

            //Check if a transaction is running
            assert isTransactionRunning() == false

            // Open a transaction
            transaction{
                // Now a transaction is running
                assert isTransactionRunning() == true
            }
        }
        // Or the short form
        transactions.isTransactionRunning()
    }
}

```

Listing 4.136: Check if a transaction is running

**TransactionHistory** The transaction history API provides some methods to handle transaction undo and redo. This way, complex model changes can be reverted quite easily.

- The `undo()` method executes an undo of the last transaction. If the last transaction frame cannot be undone or if the undo stack is empty this method returns without any changes.
- The `undoAll()` method executes undo until the transaction stack is empty or an undoable transaction frame appears on the stack.
- The `redo()` method executes an redo of the last undone transaction. If the last undone transaction frame cannot be redone or if the redo stack is empty this method returns without any changes.
- The `canUndo()` method returns `true` if the undo stack is not empty and the next undo frame can be undone. This method changes nothing but you can call it to find out if the next `undo()` call would actually undo something.
- The `canRedo()` method returns `true` if the redo stack is not empty and the next redo frame can be redone. This method changes nothing but you can call it to find out if the next `redo()` call would actually redo something.

```

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

        transactions{
            assert transactionHistory.canUndo()

            transactionHistory.undo()

            assert !transactionHistory.canUndo()
        }
    }
}

```

Listing 4.137: Undo a transaction with the transactionHistory

```

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

        transactions{
            transactionHistory.undo()

            assert transactionHistory.canRedo()

            transactionHistory.redo()

            assert !transactionHistory.canRedo()
        }
    }
}

```

Listing 4.138: Redo a transaction with the transactionHistory

#### 4.6.6.2 Operations

The model operations implement convenient means to execute complex model changes like AUTOSAR module activation or cloning complete model sub-trees. The operations API is available inside of a transaction with the keyword `operation`. The class `IOperations` defines the available methods.

- The method `activateModuleConfiguration(DefRef)` activates the specified module configuration. This covers:
  - Creation of the module including the reference in the ActiveEcuC (the ECUC-VALUE-COLLECTION)
  - Creation of mandatory containers and parameters (lower multiplicity > 0)
  - Applying the recommended configuration
  - Applying the pre-configuration values

Note: If the `DefRef` has a wildcard, `activateModuleConfiguration(DefRef)` tries to activate the most specific module definition matching the wildcard, if unique. If it is not unique the method will throw an exception. For example the `DefRef /[ANY]/Dio` will activate the `/MICROSAR/Dio` instead of `/AUTOSAR/EcucDefs/Dio`.

```

transaction{
    // Activates the Dio module
    operations.activateModuleConfiguration(sipDefRef.Dio)
}

```

Listing 4.139: Activation of the ModuleConfiguration Dio

- The method `deactivateModuleConfiguration(MIModuleConfiguration)` deletes the specified module configuration from the model. In case of a split configuration, the related persistency location is being removed from the project settings. In XML file base configurations, the related file is being deleted during the next project save if it doesn't contain configuration objects anymore.

If the module configuration is referenced from the active-ECUC this link is being removed too.

- The method `changeBswImplementation(MIModuleConfiguration, MIBswImplementation)` changes the BSW-implementation of a module configuration including the definition of all contained containers and parameters.
- `setConfigurationVariantOfAllModuleConfigurations(EEcucConfigurationVariant)` sets the implementation configuration variant of all active `MIModuleConfiguration`. If a module configuration does not support the requested variant it is ignored.

Supported enum values are:

- `com.vector.cfg.model.access.ecuconfiguration.EEcucConfigurationVariant`
  - \* `VARIANT_PRE_COMPILE`
  - \* `VARIANT_LINK_TIME`
  - \* `VARIANT_POST_BUILD_LOADABLE`

This is for *post-build loadable* only! See the method `setConfigurationVariant()` in class `IEcucModuleConfiguration` for details.

- The `deepClone(MIOBJECT, MIOBJECT)` operation copies (clones) a complete MDF model sub-tree and adds it as child below the specified parent.
  - The source object must have a parent. The clone will be added to the same MDF feature below the destination parent then
  - AUTOSAR UUIDs will not be cloned. The clone will contain new UUIDs to guarantee unambiguousness
- The method `createModelObject(Class)` creates a new element of the passed `modelClass` (meta class). The `modelObject` must be added to the whole AUTOSAR model, before finishing the transaction.
- The method `createUniqueMappedAutosarPackage()` can be used to create new MIAR-Packages in new arxml files. It creates an new instance of the specified AUTOSAR package and adds it to the model tree. All non-existing parent packages will be created too.

The new package (including new created parent packages) will be mapped uniquely to the specified location (Path and AUTOSAR version).

#### 4.6.7 Model Synchronization

The Model synchronization provides operation to solve and synchronize common model related items. The model synchronization API is available inside of an active project with the keyword `modelSynchronization`. The class `IModelSynchronizationApi` defines the available methods.

The method `synchronize()` synchronizes the model for all registered model synchronization elements like validations and other operations. The method will open a transaction, if `isSynchronizationRequired()` returns `true`, otherwise this method does nothing.

```
// Execute the model synchronization
modelSynchronization.synchronize()

//Or more elaborated, but means the same
modelSynchronization{
    if(synchronizationRequired){
        synchronize()
    }
}
```

Listing 4.140: Model synchronization inside an open project

#### 4.6.8 PreBuild and PostBuild Variance (Post-build selectable)

The variance access API is the entry point for convenient access to variant AUTOSAR model content. It provides means to filter variant model content and access variant specific data.

The DaVinci Configurator supports two types of variance:

- PostBuild variance (Post-build selectable)
- PreBuild variance

For details about PostBuild variance variance and model views read 5.2 on page 329.

##### 4.6.8.1 Investigate Project Variance

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

- The method `getCurrentlyActiveView()` returns the currently active model view.
- The method `variantView(String)` returns the `IPredefinedVariantView` with the given name. This may be a PreBuild or PostBuild view.

```
scriptTask("TaskName", DV_PROJECT){
    code{
        // Activates the DoorLeftFront variant
        variance.variantView("DoorLeftFront").activeWith{
            // Now all MDF model accesses are executed in the variant "DoorLeftFront"
        }
    }
}
```

Listing 4.141: Retrieve and use a variant view by name

```

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

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

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

```

Listing 4.142: The default view is the IPostBuildInvariantValuesView

### Investigate Project Variance - PostBuild

- The method `hasPostBuildVariance()` returns `true` if the active project contains post-build variants.
- The method `getPostBuildInvariantValuesView()` returns the PostBuild invariant values view. This view contains objects which are not variant (Object or parent have no VariationPoint) or the values in all variants are equal.
- The method `getPostBuildInvariantEcucDefView()` returns the PostBuild invariant Ecuc definition view. This view contains the same objects as the invariant values view but excludes all objects which, by (Ecuc / BSWMD) definition, support variance. Using this view you can avoid dealing with objects which are accidentally equal by value (in your test configurations) but potentially can be different because they support variance.
- The method `getAllPostBuildVariantViews()` returns the model views of all PostBuild predefined variants defined in the evaluated variant set. It never returns `null`. If the project contains no PostBuild variants, the result will be an empty list.

The order of variant views returned is deterministic. It is the natural order of the names of the variants defined in the evaluated variant set.

- The method `getAllPostBuildVariantViewsOrInvariant()` returns the same as the method `getAllPostBuildVariantViews()` if the project contains PostBuild variants. If the project contains no PostBuild variants (see `hasPostBuildVariance()`) the method returns a list containing only the `IPostBuildInvariantValuesView`.

This helps to create code working with both variant and non-variant projects.

#### 4.6.8.2 Variant Model Objects

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

- The method `activeWith(IModelView, Closure)` 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 `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 `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 331.
- The method `getVariantSiblingsWithoutMyself(MIObject)` returns the same collection as `getVariantSiblings(MIObject)` but without the specified object.

```
// IPostBuildPredefinedVariantView viewDoorLeftFront = ...
// MIParameterValue variantParameter = ...

viewDoorLeftFront.activeWith {
    assert variance.currentlyActiveView == viewDoorLeftFront

    // The parameter instance is not visible in all variants ...
    assert !variantParameter.isModelInvariant()

    // ... but all variants have a sibling with the same value
    assert variantParameter.isPostBuildValueInvariant()
}
```

Listing 4.143: Execute code in a model view

### Variant Model Objects - PostBuild

- The method `isPostBuildValueInvariant(MIObject)` returns `true` if the object has the same value in all PostBuild variants. For details about invariant views see 5.2.1.4 on page 332.
- The method `isPostBuildEcucDefInvariant(MIObject)` returns `true` if the object is invariant according to its Ecuc definition.
- The method `isNeverPostBuildVisible(MIObject)` returns `true` if the object is *invisible* in all variant views.
- The method `getVisiblePostBuildVariantViews(MIObject)` returns all variant views the specified object is visible in.
- The method `getVisiblePostBuildVariantViewsOrInvariant(MIObject)` For semantic details see `IModelViewManager.getVisiblePostBuildVariantViewsOrInvariant(MIObject)`.

## 4.6.9 Additional Model API

### 4.6.9.1 User Annotations

In DaVinci Configurator the user can add AUTOSAR annotations to configuration elements. You can create, modify, read and delete these annotations like in the UI editors.

All sub types of `MIHasAnnotation` elements support annotations like:

- `MIModuleConfigurations`
- `MIContainers`
- `MIParstringValue`s
- `MIIdentifiables`

Although annotations are stored in the data model, their `changeable` state is independent of the configuration element `changeable` state. Annotations can be added/changed/deleted on every existing configuration element with valid definition, except the project was opened in read-only mode.

The `IUserAnnotation` interface provide methods like:

- `getLabel()` - Returns the label of the annotation, like `getName()` of a container
- `setLabel()` - Changes the label
- `getText()` - Returns the text of the annotation.
- `setText()` - Changes the text
- `isChangeable()` - Returns `true`, if the annotation is changeable
- `delete()` - Deletes the annotation

**Access User Annotations** The `getUserAnnotations(MIARObject)` method returns the `IUserAnnotations` for the model element. The returned list provides additional methods defined in `IUserAnnotationList`.

```
// We already have the container "cont" or any other model element
def myContainer = cont

def annos = myContainer.userAnnotations // Retrieve the list of annotations
def anno = annos.byId("MyLabel")        // Select the annotation with "MyLabel"
def text = anno.text                   // Get the Text

// Or short
text = myContainer.userAnnotations["MyLabel"].text
```

Listing 4.144: Get a UserAnnotation of a container

**Creation and Modification of User Annotations** You can create new User Annotations with the methods:

- `createAndAdd(label)`
- `byLabelOrCreate(label)`

```
transaction{
    // We already have the container "cont"
    def anno = cont.userAnnotations.createAndAdd("MyAnno")
    anno.text = "My Text"
}
```

Listing 4.145: Create a new UserAnnotation

```
transaction{
    // We already have the container "cont"
    def anno = cont.userAnnotations.byIdOrCreate("MyAnno")
    anno.text = "My Text"
}
```

Listing 4.146: Create or get the existing UserAnnotation by label name

**Notes** The `IUserAnnotationList` is updated, when the underlying model changes.

The `IUserAnnotationList` is read only list and does not permit any modify operations defined in `java.util.List`, but certain operations like `createAndAdd(String)` will affect the list content. If you delete a contained `IUserAnnotation` the list will not be updated.

## 4.7 Generation

The Automation Interface provides generation API for different generation use cases:

- Normal code generation, see 4.7.1
  - Including external generation steps
- SWC Templates and Contract Phase Headers generation, see 4.7.3 on page 135

### 4.7.1 Code Generation

The block **generation** encapsulates all settings and commands which are related to code generation of BSW modules:

The basic structure is the following:

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

Listing 4.147: Basic structure

#### 4.7.1.1 Generation Settings

The class `IGenerationSettingsApi` encapsulates all settings which belong to a generation process.

E.g.

- Select the generators to execute
- Select the target type (Real, VTT)
- Select the external generation steps
- If the module supports multiple module configurations, select the configurations which shall be generated

The following chapters show samples for the standard use cases.

**Generation with default Project Settings** The following snippet executes a validation with the default project settings.

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

Listing 4.148: 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 4.149: Generate with standard project settings

**Generation with Reporting Settings** `IGenerationReportApi` is the entry point for generation report settings. When the settings are set and generation has been finished, the report output path can be seen in the "Configurator Console View".

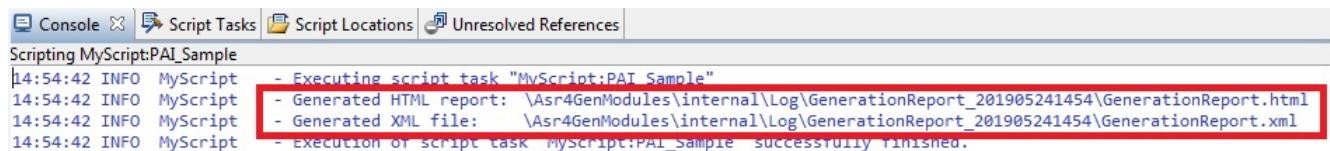


Figure 4.9: Report Output Path

The following snippet sets the report settings and executes a generation.

```

scriptTask("generate_components_with_report"){
    code{
        generation{
            settings {
                selectGeneratorsByDefRef("/MICROSAR/Aaa")
                selectGeneratorsByDefRef("/MICROSAR/Hhh")

                // Open the report closure to get access to the report settings
                report {
                    // If no settings set, the configurator settings get used
                    createHtmlReport true
                    createXmlFile true
                }
            }

            // After generation the output paths can be found in the console
            view
        generate()
    }
}

```

Listing 4.150: Generation of components with a result report

**createXmlFile** If not set, project settings will be used.

`setCreateXmlFile(Boolean)` defines if XML report file should be generated.

**createHtmlReport** If not set, project settings will be used.

`setCreateHtmlReport(Boolean)` defines if HTML report should be generated.

**Generation of one Module** This sample selects one specific module and starts the generation. There are two ways to open 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 4.151: 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 4.152: Generate one module

**Generation of multiple Modules** To select more than one generator the following snippet can be used.

```
scriptTask("generate_two_modules"){
    code{
        generation{
            settings{
                selectGeneratorsByDefRef ("/MICROSAR/Aaa", "/MICROSAR/Bbb")
            }
            generate()
        }
    }
}
```

Listing 4.153: Generate two modules

**Generation of Multi Instance Modules** Some module definitions have a upper multiplicity greater than one. (E.g. [0:5] or [0:\*)] This means it is allowed to create more than one module configuration from this module definition. If the related generator is started with the default API, all available module configurations are selected for generation. The following API can be used to generate only a subset of all related module configurations.

```
scriptTask("generate_one_module_with_two_configs"){
    code{
        generation{
            settings{
                def gen = generatorByDefRef ("/MICROSAR/MultiInstModule")
                // clear default selection
                gen.deselectAllModuleInstances()
                // Select the module configurations to generate
                gen.selectModuleInstance(AsrPath.create("/ActiveEcuC/
                    MultiInstModule1"))

                // Instead of the full qualified path, the module configuration
                // short name can also be used
                gen.selectModuleInstance("MultiInstModule2")
            }
            generate()
        }
    }
}
```

Listing 4.154: Generate one module with two configurations

#### 4.7.1.2 Generation of Generation Steps

Besides the internal generators, which are covered by the topics above, there are also generation steps which can be executed with the following API. A new block `externalGenerationSteps` within the `settings` block encapsulates all settings related to external generation scripts.

```
scriptTask("generate_ext_gen_step"){
    code{
        generation{
            settings{
                externalGenerationSteps{
                    // To take the project settings as template use
                    // externalGenerationStepsFromProject={}
                    selectStep("ExtGen1")
                    selectStep("ExtGen2")
                }
            }
            generate()
        }
    }
}
```

Listing 4.155: Execute an external generation step

**Retrieval of TargetType (REAL, VTT) of Generation Steps** You can query the `EEnvironmentTargetType` of the generation step. This will give you the information if the step can be executed in REAL, VTT or both modes.

```
generation.settings.externalGenerationSteps{
    def step = stepByName("ExtGen1")
    def targetType = step.generationStep.targetType

    if(targetType.isRealAvailable()){
        // Real use case
    }else if(targetType.isVttAvailable()){
        // VTT use case
    }else{
        // None selected
    }
}
```

Listing 4.156: Retrieval of the TargetType of a Generation Step

#### 4.7.1.3 Evaluate generation or validation results

Each validation and generation process has an overall result which states if the execution has been successfully or not. Additionally to the overall state, the state of one specific generator can also be of interest. To provide a possibility to access this information all methods for `validate` and `generate` return an `IGenerationResultModel`.

```
scriptTask("generate_with_default_settings"){
    code{
        generation{
            def result = generate()
            println "Overall result : " + result.result
            println "Duration       : " + result.formattedDuration

            // Access results of each generator or generation step
            result.generationResultRoot.allGeneratorAndStepElements.each {
                println "Generator name : " + it.name
                println "Result          : " + it.currentState
            }
        }
    }
}
```

Listing 4.157: Evaluate the generation result

### 4.7.2 Generation Task Types

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

- Generation Step: `DV_GENERATION_STEP`
- Custom Workflow Step: `DV_CUSTOM_WORKFLOW_STEP`
- Generation Process Start: `DV_ON_GENERATION_START`
- Generation Process End: `DV_ON_GENERATION_END`

The general description of the type is in chapter 4.3.1.4 on page 36. The following code samples show the usage of these task types:

**Generation Step** A sample for the `DV_GENERATION_STEP` type:

```
scriptTask("GenStepTask", DV_GENERATION_STEP){
    taskDescription "Task is executed as Generation Step"

    def myArg = newUserDefinedArgument(
        "myArgument",
        String,
        "Defines a user argument for the GenerationStep")

    code{ phase, generationType, resultSink ->

        def myArgVal = myArg.value
        // The value myArgVal was passed from the generation step in the
        // project settings editor

        scriptLogger.info "MyArg is: $myArgVal"
        scriptLogger.info "GenerationType is: $generationType"

        if(phase.calculation){
            // Execute code before / after calculation

            transaction {
                // Modify the Model in the calculation phase
            }
        }

        if(phase.validation){
            // Execute code before / after validation
        }

        if(phase.generation){
            // Execute code before / after generation
        }
    }
}
```

Listing 4.158: 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 148 for a sample how to create an validation-result and report it.

The `generationType` defines if the current generation is for the `REAL` or `VTT` platform.

**Custom Workflow Step** A sample for the DV\_CUSTOM\_WORKFLOW\_STEP type:

```
scriptTask("GenStepTask", DV_CUSTOM_WORKFLOW_STEP){
    taskDescription "Task is executed as custom workflow step"

    def myArg = newUserDefinedArgument(
        "myArgument",
        String,
        "User argument for the step")
    code{
        def myArgVal = myArg.value
        // The value myArgVal was passed from the custom workflow step in the
        // project settings editor
        scriptLogger.info "MyArg is: $myArgVal"
    }
}
```

Listing 4.159: Use a script task as custom workflow step

**Generation Process Start** A sample for the DV\_ON\_GENERATION\_START type:

```
scriptTask("GenStartTask", DV_ON_GENERATION_START){
    taskDescription "The task is automatically executed at generation start"

    code{ phasesToExecute, generators ->

        scriptLogger.info "Phases are: $phasesToExecute"
        scriptLogger.info "Generators to execute are: $generators"

        // Execute code before the generation will start
    }
}
```

Listing 4.160: Hook into the GenerationProcess at the start with script task

**Generation Process End** A sample for the DV\_ON\_GENERATION\_END type:

```
scriptTask("GenEndTask", DV_ON_GENERATION_END){
    taskDescription "The task is automatically executed at generation end"

    code{ generationResult, generators ->

        scriptLogger.info "Process result was: $generationResult"
        scriptLogger.info "Executed Generators: $generators"

        // Execute code after the generation process was finished
    }
}
```

Listing 4.161: Hook into the GenerationProcess at the end with script task

### 4.7.3 Software Component Templates and Contract Phase Headers Generation

The Software Component Templates and Contract Phase Headers (Swct) generation automation API provides access to configure and start the Swct generation.

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

The basic structure is the following:

```
generation.swct{
    settings{
        // Settings like the selection of components to generate
    }
    // The execution of the generation can be started here
    generate()
}
```

Listing 4.162: Basic Swct structure

#### 4.7.3.1 Swct Generation Settings

The class **IGenerationSwctSettingsApi** encapsulates all settings which belong to a Swct generation process.

Examples:

- Select the software components to execute
- Retrieve the available software components

The following chapters show samples for the standard use cases.

#### 4.7.3.2 Generation with default Project Settings

To execute the Swct generation with the standard project settings the following snippet can be used:

```
scriptTask("generate_with_default_settings"){
    code{
        generation.swct{
            generate()
        }
    }
}
```

Listing 4.163: SWC Templates and Contract Headers generation with standard project settings

#### 4.7.3.3 Generation of all Software Components

To execute the Swct generation for all available software components the following snippet can be used:

```
scriptTask("generate_with_default_settings"){
    code{
        generation.swct{
            settings.selectAll()
            generate()
        }
    }
}
```

Listing 4.164: SWC Templates and Contract Headers generation of all components

#### 4.7.3.4 Generation of one Software Component

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

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

```
scriptTask("generate_one_component"){
    code{
        generation.swct{
            settings{
                selectSoftwareComponent("MyApplType")
            }

            generate()
        }
    }
}
```

Listing 4.165: SWC Templates and Contract Headers generation of one selected component

Instead of selecting the software component directly by its **Name**, there is also the possibility to fetch the software component object and **select()** this object for execution.

```
scriptTask("generate_one_component"){ code{
    generation.swct{
        settings{
            def sw = softwareComponentByName("MyApplType")
            // Select the software component
            sw.select()

            // You could also retrieve information about the component
            def asrPath = sw.asrPath
            if(sw.selected){ /* Do something */ }
        }
        generate()
    }
}}
```

Listing 4.166: Swct generation get component and select component

#### 4.7.3.5 Generation of multiple Software Components

To select more than one Software Component the following snippet can be used.

```
scriptTask("generate_one_component"){
    code{
        generation.swct{
            settings{
                // Select the tow software components
                selectSoftwareComponent("MyApplType", "MySecondApplType")
            }

            generate()
        }
    }
}
```

Listing 4.167: Swct generation of multiple components

#### 4.7.3.6 Evaluate generation results

The same API is used as for the normal generation, see chapter 4.7.1.3 on page 132 for details.

## 4.8 Validation

### 4.8.1 Introduction

All examples in this chapter are based on the situation of the figure 4.10. 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 later solving-action is marked as preferred-solving-action.

There is also a **TP00011** result that stands for any other result. The examples will not touch it.

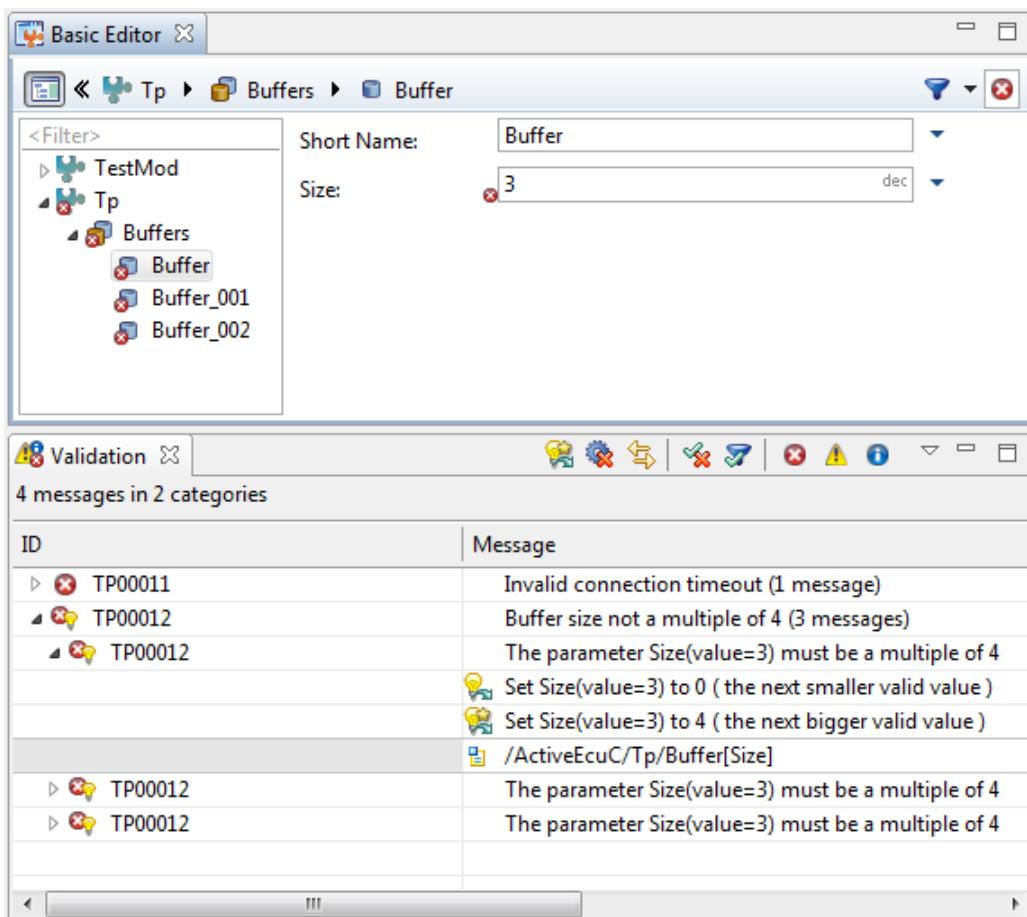


Figure 4.10: example situation with the GUI

---

Remark:

The validation-results to solve are identified via their ID. This ID is case-sensitive: that means it has to be used exactly as seen in the Validation-View. Validation-Result-IDs of MICROSAR Bsw modules are usually in capital letters (e.g. **COM02325**). Other validation-results may use validation-IDs in camel-case style (e.g. **Cfg00022**).

### 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. The returned collection has no deterministic order, especially it is not the same order as in the GUI.

```
scriptTask("CheckValidationResults_filterByOriginId", DV_PROJECT){
    code{
        validation{
            // access all validation-results
            def allResults = validationResults
            assert allResults.size() > 3

            // filter based on methods of IValidationResultUI e.g. isId()
            def tp12Results = validationResults.filter{it.isId("TP", 12)}
            assert tp12Results.size() == 3
        }

        // alternative access to validation-results without a validation block
        assert validation.validationResults.size() > 3
    }
}
```

Listing 4.168: 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 the method `ISolver.solve(Closure)` which is the most preferred way of solving validation-results with solving-actions.

See chapter 4.8.4.1 on the next page for details.

#### 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", DV_PROJECT) {
    code{
        validation{
            def tp12Results = validationResults.filter{it.isId("TP", 12)}
            assert tp12Results.size() == 3

            // Take first (any) validation-result and filter its solving-
            // actions based on methods of ISolvingActionUI
            tp12Results.first.solvingActions.filter{
                it.description.contains("next bigger valid value")

            }.single.solve() // reduce the collection to a single
            ISolvingActionUI and call solve()

            assert validationResults.filter{it.isId("TP", 12)}.size() == 2
            // One TP12 validation-result solved
        }
    }
}
```

Listing 4.169: 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)` first waits for background-validation-idle in order to have reproducible results.

The closure may contain multiple statements like:

```
result{specify result predicate}.withAction{select solving action}
```

All statements together will be used as a mapper from any solvable validation-result to a particular solving-action. The order of these statements does not affect the solving action execution order. The statement order might only be relevant if multiple statements match on a particular result, but would select a different solving-action. In that case, the first statement that successfully selects a solving-action wins.

```

scriptTask("SolveMultipleResults", DV_PROJECT){
    code{
        validation{
            assert validationResults.size() == 4
            solver.solve{
                // Call result() and pass a closure that works as filter
                // based on methods of IValidationResultUI.
                result{
                    isId("TP", 12)
                }.withAction{
                    containsString("next bigger valid value")
                }

                // On the return value, call withAction() and pass a closure that
                // selects a solving-action based on methods
                // of IValidationResultForSolvingActionSelect

                // multiple result() 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 4.170: Fast solve multiple results within one transaction

**Solve all PreferredSolvingActions** `ISolver.solveAllWithPreferredSolvingAction()` solves all validation-results with its preferred solving- action of each validation-result (solving- action return by `IValidationResultUI.getPreferredSolvingAction()`). Validation-results without a preferred solving-action are skipped.

This method first waits for background-validation-idle in order to have reproducible results.

```

scriptTask("SolveAllWithPreferred", DV_PROJECT){
    code{
        validation{
            assert validationResults.size() == 4

            solver.solveAllWithPreferredSolvingAction()

            assert validationResults.size() == 1

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

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

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

```

Listing 4.171: 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

You can retrieve validation-results also from any model object (MDF, Domain or BswmdModel).

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

```
scriptTask("CheckValidationResultsOfObject", DV_PROJECT){
    code{
        // sampleDefRefs contains DefRef constants just for this example.
        // Please use the real DefRefs from your SIP

        // a Buffer container
        def buffer002 = mdfModel(AsrPath.create("/ActiveEcuC/Tp/Buffer_002"))
        // the Size parameter
        def sizeParam = buffer002.parameter(sampleDefRefs.tpBufferSizeDefRef).
            single

        // the result exists for the Size parameter, not for the Buffer
        // container
        assert sizeParam.validationResults.size() == 1
        assert buffer002.validationResults.size() == 0
    }
}
```

Listing 4.172: Access all validation-results of a particular object

`MIOBJECT.getValidationResultsRecursive()` returns the validation-results of an `MIOBJECT` and all its children. So this will return all results of the whole subtree, like an editor displays results at parent objects.

`IViewedModelObject.getValidationResults()` returns the validation-results for the element matching the model object and the model view, like `BswmdModel` objects.

`IViewedModelObject.getValidationResultsRecursive()` returns the validation-results of an `MIOBJECT` for the elements like `BswmdModel` objects all its children. This will also filter for the correct `IModelView`. So this will return all results of the whole subtree, like an editor displays results at parent objects.

### 4.8.5.2 Access Validation-Results of a DefRef

`DefRef.getValidationResults()` returns all validation-results which match the passed definition. So every configuration element which matches the validation-result and is an instance of definition.

The used project for this call is the active project, see `ScriptApi.getActiveProject()`.

```
scriptTask("CheckValidationResultsOfDefRef", DV_PROJECT){
    code{
        // sampleDefRefs contains DefRef constants just for this example.
        // Please use the real DefRefs from your SIP

        assert sampleDefRefs.tpBufferSizeDefRef.validationResults.size() == 3
    }
}
```

Listing 4.173: 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", DV_PROJECT){
    code{
        validation{
            def tp12Const = ["TP", 12]

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

Listing 4.174: 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", DV_PROJECT){
    code{
        validation{
            assert validationResults.size() == 4

            solver.solve{
                result{isId("TP", 12)}
                    .withAction{
                        byGroupId(SA_GROUP_ID_TP12_NEXT_BIGGER_VALID_VALUE)
                    }
                    // instead of .withAction{containsString("next bigger valid value")}

                assert validationResults.size() == 1
                // Three TP12 validation-results solved.
            }
        }
    }
}
```

Listing 4.175: 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 IValidationResultUI.

```

scriptTask("IValidationResultUIApiOverview", DV_PROJECT){
    code{
        validation{
            def r = validationResults.filter{it.isId("TP", 12)}.first
            assert r.id.origin == "TP"
            assert r.id.id == 12
            assert r.description.toString().contains("must be a multiple of")
            assert r.severity == EValidationSeverityType.ERROR
            assert r.solvingActions.size() == 2
            assert r.getSolvingActionByGroupId(2).description.contains("next bigger
                valid value")

            // this result has a preferred-solving-action
            assert r.preferredSolvingAction == r.getSolvingActionByGroupId(2)

            // results with lower severity than ERROR can be acknowledged in the GUI
            assert r.acknowledgement.isPresent() == false

            // if the cause was an exception, r.cause.get() returns it
            assert r.cause.isPresent() == false

            // an ERROR result gets reduced to WARNING if one of its erroneous CEs is
                user-defined (user-overridden)
            assert r.isReducedSeverity() == false

            // on-demand results are visualized with a gear-wheel icon
            assert r.isOnDemandResult() == false
        }
    }
}

```

Listing 4.176: IValidationResultUI overview

#### 4.8.5.7 IValidationResultUI in a variant (Post-Build selectable) Project

```

scriptTask("IValidationResultUIInAVariantProject", DV_PROJECT){
    code{
        validation{
            def r = validationResults.filter{it.isId("TP", 12)}.first
            assert r.isGeneralVariantContext() // either it is a general result
            ...
            assert r.predefinedVariantContexts.size() == 0 // or it is assigned
                to one or more (but never all) variants
            // If a validator assigns a result to all variants, it will be a
                general result at UI-side.
        }
    }
}

```

Listing 4.177: 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.

To check for a certain model element is affected by the result please use the methods, which return `true`, if a model is affected by the validation-result:

- IValidationResultUI.matchErroneousCE(MIObject)
- IValidationResultUI.matchErroneousCE(IHasModelObject)
- IValidationResultUI.matchErroneousCE(MIHasDefinition, DefRef)

```
scriptTask("IValidationResultUIErroneousCEs", DV_PROJECT){
    code{
        validation{
            // sampleDefRefs contains DefRef constants just for this example.
            // Please use the real DefRefs from your SIP

            def result = validationResults.filter{it.isId("TP", 12)}.first

            // Retrieve the model element to check
            def modelElement // = retrieveElement ...

            // Check if the model object is affected by the validation-result
            assert result.matchErroneousCE(modelElement)

        }
    }
}
```

Listing 4.178: CE is affected by (matches) an IValidationResultUI

**Advanced Descriptor Details** An IDescriptor is a construct that can be used to "point to" some location in the model. A descriptor can have several kinds of aspects to describe where it points to. Aspect kinds are e.g. IMdfObjectAspect, IDefRefAspect, IMdfMetaClassAspect, IMdfFeatureAspect.

getAspect(Class) gets a particular aspect if available, otherwise null.

A descriptor has a parent descriptor. This allows to describe a hierarchy.  
E.g. if you want to express that something with definition X is missing as a child of the existing MDF object Y. In this example you have a descriptor with an IDefRefAspect containing the definition X. This descriptor that has a parent descriptor with an IMdfObjectAspect containing the object Y.

The term descriptor refers to a descriptor together with its parent-descriptor hierarchy.

```

import com.vector.cfg.model.cedescriptor.aspect.*

scriptTask("IValidationResultUIErroneousCEs", DV_PROJECT){
    code{
        validation{
            // sampleDefRefs contains DefRef constants just for this example.
            // Please use the real DefRefs from your SIP

            def result = validationResults.filter{it.isId("TP", 12)}.first
            def descriptor = result.erroneousCEs.single // this result in this
                // example has only a single erroneous-CE descriptor
            def defRefAspect = descriptor.getAspect(IDefRefAspect.class)
            assert defRefAspect != null; // this descriptor in this example has
                // an IDefRefAspect
            assert defRefAspect.defRef.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 4.179: Advanced use case - Retrieve Erroneous CEs with descriptors 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", DV_PROJECT){
    code{
        validation{
            assert validationResults.size() == 4
            // In this example, three validation-results have a preferred
            // solving action.
            // One of the three cannot be solved because a parameter is user-
            // defined.
            def summaryResult = solver.solveAllWithPreferredSolvingAction()
            assert validationResults.size() == 2 // Two have been solved, one
            // with a preferred solving-action is left.
            assert summaryResult.executionResult == EExecutionResult.WARNING

            // DemoAsserts is just for this example to show what kind of sub-
            // results the summaryResult contains.
            DemoAsserts.summaryResultContainsASubResultWith("OK",summaryResult)
            //two such sub-results for the validation-results with preferred-
            //solving-action that could be solved

            DemoAsserts.summaryResultContainsASubResultWith(["invalid
                modification","not changeable","Reason","is user-defined"],
                summaryResult)
            // such a sub-result for the failed preferred solving action due to
            // the user-defined parameter

            DemoAsserts.summaryResultContainsASubResultWith("Maximum solving
                attempts reached for the validation-result of the following
                solving-action",summaryResult)
            // 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 4.180: Examine an ISolvingActionResult

#### 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 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`).

Or you have to activate the MD license option for development during script task creation by calling the method `requiresMDDevelopmentLicense()`, then you could retrieve an `IValidationResultSink` from the method `getResultsink()`.

**Reporting ValidationResult in Task providing a ResultSink** This sample applies to task types providing a `ResultSink` in the Task API, like `DV_GENERATION_STEP`.

```

scriptTask("ScriptTaskCreationResult" /* Insert with task type providing
    resultSink */ ){
    code{
        validation{
            resultCreation{
                // The ValidationResultId group multiple results
                def valId = createValidationResultIdForScriptTask(
                    /* ID */ 1234,
                    /* Description */ "Summary of the ValidationResultId",
                    /* Severity */ EValidationSeverityType.ERROR)
                // Create a new resultBuilder
                def builder = newResultBuilder(valId, "Description of the Result")

                // You can add multiple elements as error objects to mark them
                builder.addErrorObject(sipDefRef.EcucGeneral.bswmdModel().single)
                // Add more calls when needed

                // Create the result from the builder
                def valResult = builder.buildResult()

                // You need to report the result to a resultSink
                // You have to get the sink from the context, e.g. script task args
                // a sample line would be
                resultSinkForTask.reportValidationResult(valResult)
            }
        }
    }
}

```

Listing 4.181: Create a ValidationResult

**Reporting ValidationResult with MD License Option for Development** This sample can be used in every task types but you need a MD license option for development to retrieve the ResultSink.

```

scriptTask("ScriptTaskCreationResult", DV_PROJECT){

    // Result reporting requires an MD license for development
    requiresMDDevelopmentLicense()

    code{
        validation{
            resultCreation{
                // The ValidationResultId group multiple results
                def valId = createValidationResultIdForScriptTask(
                    /* ID */ 1234,
                    /* Description */ "Summary of the ValidationResultId",
                    /* Severity */ EValidationSeverityType.ERROR)
                // Create a new resultBuilder
                def builder = newResultBuilder(valId, "Description of the Result")

                // Create the result from the builder
                def valResult = builder.buildResult()

                // When MD license is enabled you can access a resultSink
                resultSink.reportValidationResult(valResult)
            }
        }
    }
}

```

Listing 4.182: Report a ValidationResult when MD license option is available

#### 4.8.5.11 Turn off auto-solving-action execution

Auto-solving-action execution is a feature to simplify configuration by automatically adjusting dependent data after a change was made by the user. This feature runs synchronous to the user change and may have impact on UI responsiveness. If UI response time is not acceptable, this should be reported to Vector.

Using `setEnabled(boolean)`, auto-solving-action execution can be disabled to find out if this is the cause and as an interim workaround.

If auto-solving-action execution is disabled, data might get out of sync after a user change, E.g. Vtt dual target sync, BSW Internal Behavior, ... . In that case, these have to be solved manually with the corresponding validation-result's solving action.

This setting is stored as user-independent project setting.

This setting can only be changed if `isChangeable()` returns true (false e.g. due to read-only project), otherwise an `IllegalStateException` is thrown.

```
scriptTask("SolvingReturnValue", DV_PROJECT){  
    code{  
        validation{  
            settings{  
                if (autoSolvingActionExecution.changeable) {  
                    autoSolvingActionExecution.enabled = false  
                }  
            }  
        }  
    }  
}
```

Listing 4.183: Turn off auto solving action execution

## 4.9 Workflow

### 4.9.1 Update Workflow

The Update Workflow derives the initial EcuC from the input files and updates the project accordingly. The Update Workflow API comprises modification of variants, modification of the input files list and execution of an update workflow.

#### 4.9.1.1 Prerequisites

The Update Workflow can't be executed while the Project to update is open. E.g. in a `IProjectRef.openProject` closure block or in a ScriptTask with the `DV_PROJECT` ScriptTaskType. Because the update workflow has to close and open the project during update, which would cause strange behavior in your client code.

#### 4.9.1.2 Method Overview

- **workflow:** The `workflow` closure is the central entry point for the Workflow API.
  - `updateProject`: Contains all settings for the Update Workflow and executes the update after leaving the closure block.
  - `filePreprocessingProject`: Contains all settings for the file preprocessing part of the Update Workflow. Only the file preprocessing is executed after leaving the closure block and the file paths to the results are returned.
  - `isUpdateRequired`: Checks if an complete Update Workflow is required for the given project.
  - `isFilePreprocessingRequired`: Checks if the file preprocessing is required for the given project.

**File Sets** Use `getAvailableFileSets()` to get a list with all available file sets. For non-Variant projects, only one file set is available.

Use `getFileSetByName(String)` to get the file set with the given name. This method can be used to get the according file set of one specific variant.

```

scriptTask("UpdateExistingProject", DV_APPLICATION) {
    code {
        workflow {

            if(isFilePreprocessingRequired(pathToDpaFile)) {
                // File Preprocessing execution
            }

            if(isUpdateRequired(pathToDpaFile)) {

                updateProject(pathToDpaFile) {

                    fileSet() {
                        // for non variant projects
                    }

                    fileSet("VariantLeft") {
                        // for variant projects
                    }
                }
            }
        }
    }
}

```

Listing 4.184: "Selecting the according file set"

**Input Files** Use `createInputFile(IPersistablePath, Closure)` to create a new input file for the given path.

Use `clear()` to remove all existing input file from the input file list.

Use `removeByPath(IPersistablePath)` to remove the input file with the given path from the input file list. If the input file is not part of this input file list no operation will be executed.

**Input File** The `IInputFileApi` is the entry point to define and modify an input file.

Use `getAvailableEcuInstances()` to get all available ecu instances for this input file.

Use `setEcuInstance()` to set the ecu instance for this input file.

Use `setFileCategory(EInputFileCategory)` to set one file category. This overwrites all already set file categories.

Use `setFileCategory(List)` to set one or multiple file categories. This overwrites all already set file categories.

Use `getAvailableFileCategories()` to get a all available `EInputFileCategory` for this input file.

Use `setImportDidsAndRids(Boolean)` to import DIDs and RIDs as single signal for DCM (backward compatibility). This settings is only necessary for diagnostic data input files.

Use `getAvailableDiagEcuInstances()` to get all available ecu instances for this diagnostic data input file.

Use `setDiagDataEcuInstance(String, Closure)` to specify the Diagnostic Ecu Instance and the variant setting.

Use `setDiagDataEcuInstance(String)` to specify the Diagnostic Ecu Instance.

Following input file categories are generally available. (To retrieve all available file categories for a specific input file, use `IInputFileApi.getAvailableFileCategories()`)

- `EInputFileCategory.LEGACY_DIAGNOSTIC_DATA`
- `EInputFileCategory.LEGACY_DIAGNOSTIC_STATE_DESCRIPTION`
- `EInputFileCategory.LEGACY_DIAGNOSTIC_PATCH_FILE`
- `EInputFileCategory.LEGACY_COMMUNICATION_DATA`
- `EInputFileCategory.LEGACY_COMMUNICATION_MODIFICATION_SCRIPT`
- `EInputFileCategory.PROJECT_STANDARD_CONFIGURATION`
- `EInputFileCategory.DEFINITION_RESTRICTIONS`
- `EInputFileCategory.DIAGNOSTIC_COMMON_DATA`
- `EInputFileCategory.DIAGNOSTIC_SYSTEM_EXTRACT`
- `EInputFileCategory.DIAGNOSTIC_SYSTEM_DESCRIPTION`
- `EInputFileCategory.COMMUNICATION_SYSTEM_EXTRACT`
- `EInputFileCategory.COMMUNICATION_SYSTEM_DESCRIPTION`
- `EInputFileCategory.AUTOSAR_COMPLEMENTARY_DATA`

**Diagnostic Data** Use `getAvailableVariants()` to get all available variants for this diagnostic data input file.

Use `setVariant(String)` to set the according variant.

**Update Workflow Settings** The `updateSettings()` is the entry point for accessing and modification of the update settings.

```

scriptTask("TestUpdateReportSettings", DV_APPLICATION) {
    code {
        workflow.updateProject(dpaProjectFile) {

            updateSettings{

                if (isLegacyDiagOnlyUpdateExecutable()) {
                    updateMode = LEGACY_DIAG_ONLY
                } else if(isEcucAndDevWsUpdateExecutable()) {
                    updateMode = ECUC_AND_DEV_WS
                } else if(isEcucOnlyUpdateExecutable()) {
                    updateMode = ECUC_ONLY
                }

                uuidUsageInSystemDescriptionEnabled      true
                uuidUsageInStandardConfigurationEnabled true
                def sysDescEnabled =      uuidUsageInSystemDescriptionEnabled
                def sysDescStdEnabled =  uuidUsageInStandardConfigurationEnabled

                // Executes the given scripts
                // If no API call gets specified, the .DPA file content will be
                // used.
                executeSysDescModificationScript(scriptPathAsString)

                // Disables all System Description Modification scripts
                disableSysDescModificationScript()
            }

        } // workflow.updateProject
    } // code
} // scriptTask

```

Listing 4.185: Access and modify the Workflow Update settings

**Update Mode** Use `setUpdateMode` or `updateMode` to set the update mode.

- `EUpdateMode.ECUC_ONLY`
- `EUpdateMode.LEGACY_DIAG_ONLY`
- `EUpdateMode.ECUC_AND_DEVELOPER_WORKSPACE`

**UUID Usage in System Description Enabled** Use `uuidUsageInSystemDescriptionEnabled` or `setUuidUsageInSystemDescriptionEnabled` to set the UUID Usage in System Description setting. Project update will identify objects of the system description based on their UUID.

**Is UUID Usage in System Description Enabled** Use `isUuidUsageInSystemDescriptionEnabled` to get the value of the UUID Usage in System Description setting.

**UUID Usage in Standard Configuration Enabled** Use `uuidUsageInStandardConfigurationEnabled` or `setUuidUsageInStandardConfigurationEnabled` to set the UUID Usage in Standard Configuration setting. Project update will identify objects of the standard configuration based on their UUID.

**Is UUID Usage in Standard Configuration Enabled** Use `isUuidUsageInStandardConfigurationEnabled` to get the value of the UUID Usage in Standard Configuration setting.

**Execute System Description Modification Script** `executeSysDescModificationScript(String)`  
Sets the script to execute during the update workflow.

**Disable System Description Modification Script** `disableSysDescModificationScript()`  
Disables the execution of the script during the update workflow.

**Is Legacy Diagnostic only Update executable** Use `isLegacyDiagOnlyUpdateExecutable` to get the information if an Update with only the Legacy Diagnostic files is executable. If this check returns `true` the `EupdateMode` can be set to `EUpdateMode.LEGACY_DIAG_ONLY`.

**Is Ecuc and Developer Workspace Update executable** Use `isEcucAndDevWsUpdateExecutable` to get the information if an Ecuc and Developer Workspace Update is executable. If this check returns `true` the `EupdateMode` can be set to `EUpdateMode.ECUC_AND_DEV_WS`.

**Is Ecuc only Update executable** Use `isEcucAndDevWsUpdateExecutable` to get the information if an Ecuc only Update is executable. If this check returns `true` the `EupdateMode` can be set to `EUpdateMode.ECUC_ONLY`.

**Report Setting** Use `getReport()` or `report(Closure)` to specify the update report settings.

The `IUpdateReportApi` is the entry point for accessing and modifying the update workflow report settings.

```
scriptTask("TestUpdateReportSettings", DV_APPLICATION) {
    code { dpaProjectFile ->

        workflow.updateProject(dpaProjectFile)  {

            updateSettings{

                report {
                    // Sets the report options
                    createHtmlReport      true
                    createXmlFile         true

                    // If no report API call gets specified, the .DPA file
                    // content will be used.
                }
            }

            } // workflow.updateProject
        } // code
    } // scriptTask
```

Listing 4.186: Access and modify the update report settings

**Create HTML Report** `setCreateHtmlReport(Boolean)` sets the create html report option for the update workflow.

**Create XML file** `setCreateXmlFile(Boolean)` sets the create xml file report option for the update workflow.

**Selective update** Configuration of the selective update. This settings can only affect the currently loaded input files and must therefore be used after the `inputFiles` closure.

Use `includeSWC(boolean)` to include all SWC's.

Use `includeLegacyDiagnosticData(boolean)` to include all legacy diagnostic data.

Use `getAvailableCluster()` to get a list of AsrPath with all allowed clusters.

Use `selectCluster(AsrPath)` to select one cluster.

Use `selectCluster(List)` to select multiple clusters.

```
scriptTask("TestEnableSelectiveUpdateWithSettings", DV_APPLICATION) {
    code { dpaProjectFile, extractFile ->

        def extractFilePath = paths.resolvePath(extractFile)

        workflow.updateProject(dpaProjectFile) {
            fileSet() {
                inputFiles {
                    createInputFile(extractFilePath.asPersistablePath())
                }
            }
        }

        selectiveUpdate(true){
            def cluster = getAvailableCluster()

            selectCluster(cluster)

            includeLegacyDiagnosticData(false)

            includeSWC(false)
        }
    }
}
```

Listing 4.187: Enable the selective update

This example shows the API to enable the selective update.

#### 4.9.1.3 Example: Content of Input Files has changed.

In case of changed content of input files, the update workflow can be started with the `workflow.updateProject(dpaProjectFilePath)` method. This will start the Update Workflow, with the input files as selected in the "DaVinci Configurator GUI". The parameter `dpaProjectFilePath` accepts the same types and has the same semantic as `resolvePath` described in 4.4.3.1 on page 42.

```
scriptTask("UpdateExistingProject", DV_APPLICATION) {
    code {
        workflow.updateProject pathToDpaFile
    }
}
```

Listing 4.188: "Update existing project"

The update workflow is started at the end of the updateProject-closure.

#### 4.9.1.4 Example: List of input files shall be changed

In case of new input files must be added or replaced, the update workflow can be start in the same way like in previous example. Use the FileSet- and InputFiles API to define the input files. This will start the Update Workflow, with the selected input files via the InputFile API.

```
scriptTask("DemonstrateUpdateWorkflow", DV_APPLICATION) {
    code {

        def extractFilePath = paths.resolvePath(extractFile)
        def diagExtractPath = paths.resolvePath(diagExtract)

        workflow.updateProject(dpaProjectFile) {

            fileSet() {

                inputFiles {
                    // Remove all existing input files
                    clear()

                    // Create a new input file
                    createInputFile(extractFilePath.asPersistablePath()) {
                        // Set the according file category
                        setFileCategory(EInputFileCategory.
                            COMMUNICATION_SYSTEM_EXTRACT)
                        def ecuInstance = getAvailableEcuInstances().get(0)
                        // Set the ecu instance
                        setEcuInstance(ecuInstance)
                    }
                    // The input file validation is started at the end of the
                    // inputFile-closure.

                    // create a second input file
                    createInputFile(diagExtractPath.asPersistablePath()) {
                        setFileCategory(EInputFileCategory.
                            DIAGNOSTIC_SYSTEM_EXTRACT)
                        def ecuInstance = getAvailableEcuInstances().get(0)
                        setEcuInstance(ecuInstance)
                    }
                }
            }
        }
    }
}
```

Listing 4.189: Exchange all input files and start update

This example shows the complete replacement of the current list of input files. The update workflow is executed after the update closure block is left.

#### 4.9.1.5 Example: Diagnostic Data as input file

In case the input file is a diagnostic data input file, the diagnostic ecu instance and the variant might need to be set. Use the DiagData Api to set the ecu instance and the variant.

```
scriptTask("DemonstrateDiagUpdateWorkflow", DV_APPLICATION){  
  
    code{  
        def diagDataPath = paths.resolvePath(diagData)  
  
        workflow.updateProject(dpaProjectFile) {  
            fileSet {  
                inputFiles {  
                    clear()  
  
                    // Create a new diagnostic data input file  
                    createInputFile(diagDataPath.asPersistablePath()){  
                        // Set the according file category  
                        setFileCategory(EInputFileCategory.  
                            LEGACY_DIAGNOSTIC_DATA)  
  
                        def ecuInstance = getAvailableDiagEcuInstances().get(0)  
  
                        // Set the according diagnostic ecu instance only  
                        setDiagDataEcuInstance(ecuInstance)  
  
                        // Set the according diagnostic ecu instance and  
                        // specify the variant  
                        setDiagDataEcuInstance(ecuInstance) {  
                            // The available variants depends on the selected  
                            // ecu instance  
                            def myVariant = getAvailableVariants().get(0)  
                            variant = myVariant  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

Listing 4.190: Set diag data as input file

#### 4.9.1.6 Example: Change the list of Input Files in an Postbuild-Selectable Variant project

```

scriptTask("DemonstrateVariantUpdateWorkflow", DV_APPLICATION) {

    code {
        def extractFilePathLeft = paths.resolvePath(extractFileLeft)
        def extractFilePathRight = paths.resolvePath(extractFileRight)

        workflow.updateProject(dpaProjectFile) {
            // Get the according file set by the name of the variant
            def variantLeft = getFileSetByName("Left")
            // Select the file set
            fileSet(variantLeft){

                inputFiles {
                    // Remove all existing input files from this file set
                    clear()
                    // Create a new InputFile and set EcuInstance and
                    // FileCategory
                    createInputFile(extractFilePathLeft.asPersistablePath())
                        {
                            setFileCategory(EInputFileCategory.
                                COMMUNICATION_SYSTEM_DESCRIPTION)

                            def ecuInstance = getAvailableEcuInstances().get(1)
                            setEcuInstance(ecuInstance)
                        }
                }
            }

            def variantRight = getFileSetByName("Right")

            fileSet(variantRight) {

                inputFiles{
                    clear()
                    createInputFile(extractFilePathRight.asPersistablePath())
                        {
                            setFileCategory(EInputFileCategory.
                                COMMUNICATION_SYSTEM_DESCRIPTION)
                            def ecuInstance = getAvailableEcuInstances().get(1)
                            setEcuInstance(ecuInstance)
                        }
                }
            }
        }
        // Leaving the closure will start the Update Workflow
    }
}

```

Listing 4.191: Change list of communication extracts and update

This example shows the complete replacement of the current list of input files in a Postbuild-Selectable Variant project. The according file set for each variant can be retrieved by name.

#### 4.9.1.7 Example: File Preprocessing of Input Files and querying the result

The file preprocessing can be executed separately from the configuration update.

```

scriptTask("DemonstrateFilePreprocessing", DV_APPLICATION) {
    code {
        def inputFileResolved = paths.resolvePath(inputFile)
        def result

        workflow {
            result = filePreprocessingProject(dpaProjectFile) {

                fileSet {
                    inputFiles {
                        // Remove all existing input files from this file set
                        clear()

                        createInputFile(inputFileResolved.asPersistablePath()) {
                        }
                    } // inputFiles
                } // fileSet
            } // filePreprocessingProject

        } // workflow

        // path to System Extract and Diagnostic Ecuc Export available in the
        // result
        def extractResult = result.getSystemExtractResult()
        def diagEcuc = result.getDiagnosticEcucExportResult()

        println "System Extract result file path: ${extractResult}"
        println "Diagnostic Ecuc result file path: ${diagEcuc}"

    } // code
} // scriptTask

```

Listing 4.192: Executes the file preprocessing without updating the configuration

The same API for file preprocessing is available like in the update project API to configure the Update Workflow. Additionally the results of the file preprocessing are available. Depending on the configured input files there will be one resulting System Extract or one Diagnostic Ecuc file. The results can be accessed by their file paths.

#### 4.9.2 Configure Variants

This API is in beta status.

`IConfigureVariantsApiEntryPoint` is the entry point for accessing the variant configuration.

Use `IConfigureVariantsApi` to access the variant creation.

`ISimpleModeApi` is the entry point for accessing the simple mode of creating variants.

`IVariantApi` can be used to create and modify variants.

#### 4.9.2.1 Example

```

import com.vector.cfg.workflow.evs.groovy.IVariantApi

scriptTask("TestsProjectSettings", DV_APPLICATION) {
    code {
        workflow {

            configureVariants( dpaFilePath ) {
                simpleMode {

                    // The first getOrCreateVariant("Variant1") creates a
                    // variant
                    IVariantApi variant1a = getOrCreateVariant("Variant1")
                    // The second getOrCreateVariant("Variant1") returns
                    // the already existing variant.
                    IVariantApi variant1b = getOrCreateVariant("Variant1")
                    if ( variant1a == variant1b ) {
                        println "Same variant object"
                    }

                    getOrCreateVariant("Variant2")
                    IVariantApi variant3 = getOrCreateVariant("Variant3")
                    removeVariant(variant3)

                    List<IVariantApi> variants = getAvailableVariants()

                    existsVariantByName("Variant1")
                    existsVariantByName(variant1a.getName())

                } // simpleMode
            } // configureVariants

        } // workflow
    } // code
} // scriptTask

```

Listing 4.193: Accessing the API for creating variants

#### 4.9.2.2 Simple Mode APIs

**Create or get variant** ISimpleModeApi.getOrCreateVariant(String) returns a new IVariantApi or an already existing one.

**Available variants** ISimpleModeApi.getAvailableVariants() returns the available variants.

**Remove variant** ISimpleModeApi.removeVariant(IVariantApi) removes the given variant.

**Exist variant** ISimpleModeApi.existsVariantByName(String) checks if given variant exists.

#### 4.9.2.3 Variant API

**Name** `IVariantApi.getName()` returns the name of the variant.

## 4.10 Domains

The domain APIs are specifically designed to provide high convenience support for typical domain use cases.

The domain API is the entry point for accessing the different domain interfaces. It is available in opened projects in the form of the `IDomainApi` interface.

`IDomainApi` provides methods for accessing the different domain-specific APIs. Each domain's API is available via the domain's name. For an example see the communication domain API 4.10.1.

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

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

Listing 4.194: 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 4.195: 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 4.196: 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 4.197: 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.

**Accessing Can Pdus** `getCanPdus()` returns a list of all `ICanPdus` in the configuration. Can Pdus of a certain Can Controller can also be accessed via `ICanController.getCanPdus()`. 4.10.1.3 on page 166

#### 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('OptimizeAcceptanceFilters') {
    code {
        transaction {
            domain.communication {
                // open acceptance filters of all CanControllers
                canControllers*.openAcceptanceFilters()

                // open acceptance filters of first CanController
                canControllers.first.openAcceptanceFilters()
                canControllers[0].openAcceptanceFilters() // same as above

                // open acceptance filters of second CanController
                // (if there is a second CanController)
                canControllers[1]?.openAcceptanceFilters()

                // open acceptance filters of a dedicated CanController
                canControllers.filter { it.name.contains 'CHO' }.single.
                    openAcceptanceFilters()

                // accessing a dedicated CanController
                def ch0 = canControllers.filter { it.name.contains 'CHO' }.single

                // assert: ch0's first CanFilterMask value is XXXXXXXXXXXX
                assert 'XXXXXXXXXX' == ch0.canFilterMasks[0].filter

                // set CanFilterMask value to 0111111111
                ch0.canFilterMasks[0].filter = '0111111111'
                assert '0111111111' == ch0.canFilterMasks[0].filter

                // automatic acceptance filter optimization
                ch0.optimizeFilters { fullCan = true }
            }
        }
        scriptLogger.info('Successfully optimized Can acceptance filters.')
    }
}
```

Listing 4.198: 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 previous page. The following use cases are supported:

**Filter Types** ECanAcceptanceFilterType lists the possible values for an ICanFilterMask's filter type.

STANDARD results in a standard Can acceptance filter value with length 11.

EXTENDED results in an extended Can acceptance filter value with length 29.

MIXED results in a mixed Can acceptance filter value with length 29.

**Accessing a CanFilterMask's Filter Type** `getFilterType()` returns this ICanFilterMask's filter type.

**Specifying a CanFilterMask's Filter Type** Using `setFilterType(ECanAcceptanceFilterType)` this ICanFilterMask's filter type can be specified.

**Accessing a CanFilterMask's Filter Value** `getFilter()` returns this ICanFilterMask's filter value. A CanFilterMask's filter value is a String containing the characters '0', '1' and 'X' (don't care). For determining if a given Can ID passes the filter it is matched bit for bit against the String's characters. The character at index 0 is matched against the most significant bit. The character at index `length() - 1` is matched against the least significant bit. The length of the String corresponds to the CanFilterMask's filter type.

**Specifying a CanFilterMask's Filter Value** Using `setFilter(String)` this ICanFilterMask's filter value can be specified.

**Accessing a CanFilterMask's MIContainer** `getMdfObject()` returns the MIContainer represented by this ICanFilterMask.

#### 4.10.1.3 CanPdus

An ICanPdu represents a Pdu of the CanIf module.

- `disableFullCan()` disables the FullCAN feature for this `ICanPdu`. In other words the corresponding FullCAN hardware object will be deleted and the references redirected to a suitable BasicCAN hardware object.

To enable the FullCAN feature please use `enableFullCan()`.

- `enableFullCan()` enables the FullCAN feature for this `ICanPdu`. In other words a corresponding FullCAN hardware object will be created and the references of the relevant objects redirected to it from the previously referenced BasicCAN hardware object.

To disable the FullCAN feature please use `disableFullCan()`. If the FullCAN feature is already enabled can be checked using `isFullCanEnabled()`. This might be useful to avoid multiple hardware objects for the same PDU, when using `enableFullCan()` more than once on the same PDU.

- `setFullCanLocked(boolean)` switches whether the filter optimization algorithm is allowed to change the configured CAN handle type (BasicCAN/FullCAN) or not.

If locked the selected CAN handle type (FullCAN/BasicCAN) of the corresponding Rx-PDU is NOT touched and NOT changed by the filter optimization algorithm.

Tx `ICanPdus` are ignored by this method, since the lock is only available for Rx `ICanPdus`.

- `isFullCanEnabled()` determines whether the configured CAN handle type is FullCAN or BasicCAN .

The CAN handle type can be changed between FullCAN and BasicCAN using `enableFullCan()` and `disableFullCan()` methods.

- `isFullCanLocked()` determines whether the CAN handle type is locked for the filter optimization algorithm.

Note: Only Rx `ICanPdus` can be locked, so returns always false for Tx `ICanPdus`.

**Accessing a CanController's CanPdus** `getCanPdus()` returns all of this `ICanController`'s `ICanPdus`.

**Accessing a CanPdu's MIContainer** `getMdfObject()` returns the `MIContainer` represented by this `ICanPdu`.

```

import com.vector.cfg.model.view.IModelViewExecutionContext

scriptTask("enableFullCAN", DV_PROJECT){
    code {
        transaction {
            domain.communication {
                variance {
                    getAllPostBuildVariantViewsOrInvariant().each {
                        if (it.getName().equals("Left")) {
                            final IModelViewExecutionContext context = it.executeWithThisView()
                            context.withCloseable {
                                canPdus.filter {
                                    it.getName().equals("RxFullCanDisabled1_0a94227e_Rx")
                                }*.enableFullCan()
                            }
                        }
                    }
                }
            }
        }
    }
}

```

Listing 4.199: Enable FullCAN feature for PDU

```

import com.vector.cfg.model.view.IModelViewExecutionContext

scriptTask("disableFullCAN", DV_PROJECT){
    code {
        transaction {
            domain.communication {
                variance {
                    getAllPostBuildVariantViewsOrInvariant().each {
                        if (it.getName().equals("Left")) {
                            final IModelViewExecutionContext context = it.executeWithThisView()
                            context.withCloseable {
                                canControllers.filter {
                                    it.getName().equals("Controller_MyEcu_1_2514e902")
                                }*.canPdus
                                .flatten()
                                *.disableFullCan()
                            }
                        }
                    }
                }
            }
        }
    }
}

```

Listing 4.200: Disable FullCAN feature for all PDUs of CanController

#### 4.10.2 Diagnostics Domain

The diagnostics domain API is specifically designed to support diagnostics related use cases. It is available from the [IDomainApi 4.10](#) on page 163 in the form of the [IDiagnosticsApi](#) interface.

`getDiagnostics()` allows accessing the `IDiagnosticsApi` like a property.

```
scriptTask('taskName') {  
    code {  
        // IDiagnosticsApi is available as "diagnostics" property  
        def diagnostics = domain.diagnostics  
    }  
}
```

Listing 4.201: Accessing `IDiagnosticsApi` as a property

`diagnostics(Closure)` allows accessing the `IDiagnosticsApi` in a scope-like way.

```
scriptTask('taskName') {  
    code {  
        domain.diagnostics {  
            // IDiagnosticsApi is available here  
        }  
    }  
}
```

Listing 4.202: Accessing `IDiagnosticsApi` in a scope-like manner

The following use cases are supported:

**Dem Events** The API provides access and creation of `IDemEvents` in the configuration. See chapter 4.10.2.1 on the next page for more details.

**Check for OBD II** `isObd2Enabled()` checks, if OBD II is available in the configuration.

**Enable OBD II** `setObd2Enabled(boolean)` enables or disables OBD II in the configuration. Note, that OBD II can only be enabled, if a valid SIP license was found.

**Check for WWH-OBD** `isWwhObdEnabled()` checks, if WWH-OBD is available in the configuration.

**Enable WWH-OBD** `setWwhObdEnabled(boolean)` enables or disables WWH-OBD in the configuration. Note, that WWH-OBD can only be enabled, if a valid SIP license was found.

#### 4.10.2.1 DemEvents

An IDemEvent instance represents a diagnostic event and provides usecase centric functionalities to modify and query diagnostic events.

**Accessing Dem Events** `getDemEvents()` returns a list of all IDemEvents in the configuration.

**Creating Dem Events** `createDemEvent(Closure)` is used to create diagnostic events of different kinds.

The method can be configured to create different types of DTCs/Events:

1. **UDS Event:** This is the default type of event, when only an 'eventName' and a 'dtc' number is specified. A new DemEventParameter container with the given shortname and a new DemDTCClass with the given DemUdsDTC is created.

```
scriptTask('taskName') {
    code {
        transaction {
            domain.diagnostics {

                def udsEvent = createDemEvent {
                    eventName = "NewUdsEvent"
                    dtc = 0x30
                }
            }}}}
```

Listing 4.203: Create a new UDS DTC with event

2. **OBD II Event:** If OBD II is enabled for the loaded configuration, and a 'obd2Dtc' is specified instead of a 'dtc', the method will create an OBD II relevant event. The difference is, that it will set the parameter DemObdDTC instead of DemUdsDTC. It is also possible to specify 'dtc' as well as 'obd2dtc', which will result in both DTC parameters are set.

```
scriptTask('taskName') {
    code {
        transaction {
            domain.diagnostics {
                // OBD must be enabled and legislation must be OBD2
                // Enable OBD2
                obd2Enabled = true

                def obd2Event = createDemEvent {
                    eventName = 'NewOBD2Event'
                    obd2Dtc = 0x40
                }

                def obd2CombinedEvent = createDemEvent {
                    eventName = 'UDS_OBD2_Combined_Event'
                    dtc = 0x31
                    obd2Dtc = 0x41
                }
            }}}}
```

Listing 4.204: Enable OBD II and create a new OBD related DTC with event

3. **WWH-OBD Event:** If WWH-OBD is enabled for the loaded configuration, and a 'wwhObdDtcClass' with a value other than 'NO\_CLASS' is specified, the method will create a WWH-OBD relevant event. Note that WWH-OBD relevant events usually du reference the so called MIL indicator, thus this reference will be set by default in the newly created DemEventParameter.

```
scriptTask('taskName') {
    code {
        transaction {
            domain.diagnostics {
                // OBD must be enabled, and legislation must be WWH-OBD
                // The parameter '/Dem/DemGeneral/DemMILIndicatorRef' must
                // be set
                wwhObdEnabled = true

                def wwhObdEvent = createDemEvent {
                    eventName = 'WWHOBD_Event'
                    dtc = 0x50
                    // wwhObdClass != NO_CLASS indicates WWH-OBD event
                    wwhObdDtcClass = CLASS_A
                }
            }
        }
    }
}
```

Listing 4.205: Enable WWH-OBD and create a new OBD related DTC with event

4. **J1939 Event:** The last type of event is a J1939 related event, which can be created when J1939 is licensed and available for the loaded configuration. This is done in a similar way as for UDS events, but additionally specifying 'spn', 'fmi' values as well as the name of the referenced 'nodeAddress'.

```
scriptTask('taskName') {
    code {
        def nodeAddressContainer = mdfModel(AsrPath.create("/ActiveEcuC/
            Dem/DemConfigSet/DemJ1939NodeAddress", MIContainer))

        transaction {
            domain.diagnostics {
                // J1939 Event creation
                // J1939 must be enabled and License must be available.
                j1939Enabled = true

                def j1939Event = createDemEvent {
                    eventName 'J1939_Event'
                    dtc 0x30
                    spn 90
                    fmi 13
                    nodeAddress nodeAddressContainer
                }
            }
        }
    }
}
```

Listing 4.206: Open a project, enable J1939 and create a new J1939 DTC with event

#### Important Note:

For every DTC numbers apply the rule, that if there are already DemDTCClasses with the given number, they will be used. In such a case, no new DemDTCClass container is created.

### 4.10.3 Mode Management Domain

The mode management domain API is specifically designed to support mode management related use cases. It is available from the `IDomainApi` 4.10 on page 163 in the form of the `IModeManagementApi` interface.

`getModeManagement()` allows accessing the `IModeManagementApi` like a property.

```
scriptTask('taskName') {
    code {
        // IModeManagementApi is available as "modeManagement" property
        def modeManagement = domain.modeManagement
    }
}
```

Listing 4.207: Accessing `IModeManagementApi` as a property

`modeManagement(Closure)` allows accessing the `IModeManagementApi` in a scope-like way.

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

Listing 4.208: Accessing `IModeManagementApi` in a scope-like way

#### 4.10.3.1 BswM Auto Configuration

The `IBswMAutoConfigurationApi` allows for semi-automatic creation of dedicated parts of the BswM configuration. The BswM auto configuration takes an input consisting of "features" and "parameters" to be provided via the `IBswMAutoConfigurationApi`. Each feature may have zero, one or more sub-features and zero, one or more parameters.

The corresponding BswM configuration content is derived based on the (de)activation of features and the values assigned to the parameters.

The available features and parameters depend strongly on the project's input data and general project setup. They can be addressed by `String` identifiers. These identifiers are best obtained from the corresponding auto configuration assistant of the BSW management editor in the Cfg5 GUI.

```

scriptTask('EcuStateHandlingAutoConfiguration', DV_PROJECT) {
    code {
        // In projects with post-build selectable variance switching to an
        // IPredefinedVariantView for performing auto configuration is mandatory
        variance.variantView('Left').activeWith {

            domain.modeManagement.bswMAutoConfig('Ecu State Handling') {
                activate '/ECU State Machine/Support ComM'
                set '/ECU State Machine/Self Run Request Timeout' to 0.2
                set '/ECU State Machine/Number of Run Request User' to 4
                overrides {
                    if (addition || removal) {
                        keepOverride
                    } else if (BswMArgumentRef.DefRef.isDefinitionOf(element)
                               && feature('/ECU State Machine/Support ComM/CAN00_f26020e5').
                                   enabled
                               && parameter('/ECU State Machine/Number of PostRun Request
                                         User').value == 4) {
                        discardOverride
                    } else {
                        keepOverride
                    }
                }
            }
        }
    }
}

```

Listing 4.209: ECU State Handling Auto Configuration

**Executing the BswM Auto Configuration** `IModeManagementApi.bswMAutoConfig(String, Closure)` delegates the given code to the `IBswMAutoConfigurationApi` of the given BswM auto configuration domain. Also see overload `bswMAutoConfig(MIContainer, String, Closure)` for using this API in multi partition use case.

**Activating BswM Auto Configuration Features** `activate(String)` activates the BswM auto configuration feature with the given identifier. All enabled sub-features of the specified feature are also activated. Imagine the features displayed in a tree structure (like in Cfg5 GUI) where checking a tree node automatically checks all children.

**Deactivating BswM Auto Configuration Features** `deactivate(String)` deactivates the BswM auto configuration feature with the given identifier. All enabled sub-features of the specified feature are also deactivated. Imagine the features displayed in a tree structure (like in Cfg5 GUI) where unchecking a tree node automatically unchecks all children.

**Assigning Values to BswM Auto Configuration Parameters** `set(String)` sets the parameter with the given identifier to the specified value. Supported value types are `boolean`, `BigInteger`, `BigDecimal`, `String` and `MIReferrable` (reference parameters).

**Manually Adapting the BswM Auto Configuration Content** The BswM auto configuration mechanism is useful for creating large parts of the BswM configuration based on certain built-in heuristics. Where these heuristics fail to fulfill detailed project specific requirements manual adaptations to the auto-generated configuration content become necessary.

Per default manual adjustments are kept in the configuration. But subsequent BswM auto configuration runs may render previously applied adjustments obsolete or dysfunctional. Using `overrides(Closure)` a callback can be registered to be called for each detected adaptation. The callback can decide for each adjustment if it is to remain in the configuration or if it is to be overwritten by the BswM auto configuration. For details on which information is provided to this callback please refer to the javadoc provided with `IBswMAutoConfigurationOverride`.

**Inspecting BswM Auto Configuration Domains** The `getBswMAutoConfigDomains()` method of the `IModeManagementApi` interface provides read-access to all available BswM auto configuration domains. Available features and parameters can be inspected for various properties. See javadoc of `IBswMAutoConfigurationDomain`, `IBswMAutoConfigurationFeature` and `IBswMAutoConfigurationParameter` for details. Also see overload `getBswMAutoConfigDomains(MIContainer)` for using this API in multi partition use case.

```

domain.modeManagement {
    // In projects with post-build selectable variance switching to an
    // IPredefinedVariantView for inspecting auto configuration is mandatory
    variance.variantView('Left').activeWith {

        // get all BswM auto configuration domains
        def ecuStateHandlingDomain = bswMAutoConfigDomains.foreach {
            scriptLogger.info it.identifier
        }

        def isEnabled = bswMAutoConfigDomain 'Ecu State Handling' feature '/ECU
            State Machine/Support Comm' enabled
        def isActivated = bswMAutoConfigDomain 'Ecu State Handling' feature '/ECU
            State Machine/Support Comm' activated
        if (isEnabled && isActivated) {
            // activation state can be toggled at enabled features only
            bswMAutoConfig('Ecu State Handling') {
                deactivate '/ECU State Machine/Support Comm'
            }
        }

        bswMAutoConfigDomain('Ecu State Handling') {
            // this code is delegated to the 'Ecu State Handling'
            // auto configuration domain
            def p1 = parameter '/ECU State Machine/Self Run Request Timeout' value
            scriptLogger.info 'Self Run Request Timeout = ' + p1
            def p2 = parameter '/ECU State Machine/Number of Run Request User' value
            scriptLogger.info 'Number of Run Request User = ' + p2

            // get all root features
            rootFeatures.foreach { scriptLogger.info it.identifier }

            // get all sub-features of a feature
            feature '/ECU State Machine/Support Comm' subFeatures.foreach {
                scriptLogger.info it.identifier
            }

            // get all parameters of a feature
            feature '/ECU State Machine' parameters.foreach {
                scriptLogger.info it.identifier
            }
        }
    }
}

```

Listing 4.210: Inspecting Auto Configuration Elements

#### 4.10.4 Runtime System Domain

The runtime system domain API is specifically designed to support runtime system related use cases. It is available from the `IDomainApi` (see 4.10 on page 163) in the form of the `IRuntimeSystemApi` interface.

`getRuntimeSystem()` allows accessing the `IRuntimeSystemApi` like a property.

```
scriptTask('taskName') {
    code {
        // IRuntimeSystemApi is available as "runtimeSystem" property
        def runtimeSystem = domain.runtimeSystem
    }
}
```

Listing 4.211: Accessing IRuntimeSystemApi as a property

`runtimeSystem(Closure)` allows accessing the `IRuntimeSystemApi` in a scope-like way.

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

Listing 4.212: Accessing IRuntimeSystemApi in a scope-like way

The access point for elements of the runtime system domain are the selections. They are most of the time your starting point and offer predicates to filter for elements which are relevant. These selections can be used to get the elements for further work, but they also offer direct methods for actions that can be performed for them.

As default the selection APIs select always from all elements, but you can also put elements into most selections and use predicates to filter them. The `put` methods helps you to transfer the elements from selection to selection and put newly created elements into the next selection to perform the next step of your workflow.

We will start by introducing each selection API and will then have a look on common use cases. Before starting to implement a use case it might be helpful to have a quick look into the chapters of the selections that are required for it. The objects such as communication element or component port used in the runtime system domain API are explained in the chapter of the corresponding selection API.

#### 4.10.4.1 Component Port Selection

A component port (`IComponentPort`) represents a port prototype and its corresponding component prototype, and in case of a delegation port the corresponding top level composition type (ECU Composition).

`selectComponentPorts(Closure)` allows the selection of `IComponentPorts` using predicates.

The component port selection can be used to select and filter component ports and either do further operations on them, such as connecting to other ports, terminating them or to just return a list of component ports with which you can continue working.

`getComponentPorts()` allows access to the single component ports in the `IComponentPortSelection`.

**Component Port Predicates** To select component ports predicates can be provided to narrow down the result.

Per default the predicates are combined via logical AND. To realize other combinations, use the 'or', 'not' and 'and' predicates.

- `unconnected()` matches unconnected component ports.
- `connected()` matches connected component ports.
- `completed()` matches component ports which are completed.

A delegation port is completed if and only if the port is connected and each of the port's communication elements is data mapped to a system signal or a system signal group.

An inner port is completed if the port is connected or each of the port's communication elements is data mapped to a system signal or a system signal group. This means an inner port which is connected and data mapped is also completed.

- `notCompleted()` matches component ports which are not completed.

See `completed()` for the conditions a port has to meet to be a completed port.

- `terminated()` matches terminated component ports.
- `notTerminated()` matches non-terminated component ports.
- `senderReceiver()` matches component ports whose port has a sender/receiver port interface.
- `clientServer()` matches component ports whose port has a client/server port interface.
- `modeSwitch()` matches component ports whose port has a mode-switch port interface.
- `nvData()` matches component ports whose port has a NvData port interface.
- `parameter()` matches component ports whose port has a parameter (calibration) port interface.
- `trigger()` matches component ports whose port has a trigger port interface.
- `provided()` matches provided component ports (p-port).
- `required()` matches required component ports (r-port).
- `providedRequired()` matches provided-required component ports (pr-port).
- `delegation()` matches delegation ports (ports of the Ecu composition).
- `application()` matches component ports whose port interface is an application port interface.
- `service()` matches component ports whose port interface is an service port interface.
- `applicationComponent()` matches component ports whose component type is an application component type. Application component types are all component types which are not service component types, as displayed in the ECU Software Components Editor, not ApplicationSwComponentTypes as defined by AUTOSAR.
- `serviceComponent()` matches component ports whose component type is a service component type.
- `parameterComponent()` matches component ports whose component type is a parameter component type.

- `nvBlockComponent()` matches component ports whose component type is a nv block component type.
- `sensorActuatorComponent()` matches component ports whose component type is a sensor actuator component type.
- `ioHwAbstractionComponent()` matches component ports whose component type is a I/O hardware abstraction component type, also called EcuAbstractionSwComponentType.
- `complexDeviceDriverComponent()` matches component ports whose component type is a complex device driver component type.
- `serviceProxyComponent()` matches component ports whose component type is a service proxy component type.
- `name(String)` matches component ports with the given port name.
- `names(Collection)` matches component ports with the given port names. The order of the names is not relevant in any kind.
- `name(Pattern)` matches component ports with the given port name pattern.
- `componentPortName(String)` matches component ports with the given component port name.

The component name and port name are separated by a dot, e.g. '`MySwc.MyApplicationPort`', '`ECU Composition.MyDelegationPort`'. See also `IComponentPort.getName()`.

- `componentPortNames(Collection)` matches component ports with the given component port names.  
The component name and port name are separated by a dot, e.g. '`MySwc.MyApplicationPort`', '`ECU Composition.MyDelegationPort`'. See also `IComponentPort.getName()`.  
The order of the names is not relevant in any kind.
- `asrPath(String)` matches component ports with the given port autosar path.
- `asrPath(Pattern)` matches component ports with the given port autosar path pattern.
- `component(String)` matches component ports with the given component name.
- `components(Collection)` matches component ports with the given component names.  
The order of the names is not relevant in any kind.
- `component(Pattern)` matches component ports with the given component name pattern.
- `componentAsrPath(String)` matches the component ports with the given component autosar path.
- `componentAsrPath(Pattern)` matches component ports with the given component autosar path pattern.
- `componentType(String)` matches component ports whose component type's name equals the given component type name.
- `componentType(Pattern)` matches component ports whose component type's name matches the given component type name pattern.
- `componentTypeAsrPath(String)` matches the component ports whose component type's autosar path equals the given component type autosar path.
- `componentTypeAsrPath(Pattern)` matches component ports whose component type's autosar path matches the given component type autosar path pattern.

- `portInterfaceMapping(String)` matches component ports for whose port interfaces a port interface mapping with the given port interface mapping name exists.
- `portInterfaceMapping(Pattern)` matches component ports for whose port interfaces a port interface mapping with the given port interface mapping name pattern exists.
- `portInterfaceMappingAsrPath(String)` matches component ports for whose port interfaces a port interface mapping with the given port interface mapping autosar path exists.
- `portInterfaceMappingAsrPath(Pattern)` matches component ports for whose port interfaces a port interface mapping with the given port interface mapping autosar path pattern exists.
- `originComponentPortName(String)` matches component ports having an origin component port with the given originPortName. That means the port has an incomplete delegation connection in the structured extract to a composition port with the given originPortName.
- `originComponentPortNames(Collection)` matches component ports having an origin component port with one of the given originPortNames. That means the port has an incomplete delegation connection in the structured extract to a composition port with one of the given originPortNames. The order of the names is not relevant in any kind.
- `originComponentPortNamePattern(Pattern)` matches component ports having an origin component port with the given origin port name pattern. That means the port has an incomplete delegation connection in the structured extract to a composition port with the given origin port name pattern.
- `originComponentPortComponent(String)` matches component ports having an origin component port with the given originComponentName. That means the port has an incomplete delegation connection in the structured extract to a composition port whose composition owner instance has the given originComponentName.
- `originComponentPortComponent(Pattern)` matches component ports having an origin component port with the given origin component name pattern. That means the port has an incomplete delegation connection in the structured extract to a composition port whose composition owner instance has the given origin component name pattern.
- `hasInnerTopLevelDelegationOriginComponentPort()` matches component ports which have an inner top level origin component port. That means the port in the structured extract has an incomplete delegation connection which ends at a composition that is instantiated directly inside the top level composition (ECU Composition).
- `filterAdvanced(Closure)` matches component ports for which the given closure results to true.
- `and(Closure)` combines the predicates inside the closure with a logical AND.
- `or(Closure)` combines the predicates inside the closure with a logical OR.
- `not(Closure)` negates the combination of predicates inside the closure.
- `put(List)` can be used to set `IComponentPorts` into the selection. This is the most efficient way to create a selection from existing objects. If further predicates are specified the predicates will be applied only on the component ports that were given into this `put(List)` method. The iteration order is relevant for getting deterministic results on further usage of the selection API. This method should only be called once.

## Examples

```
scriptTask("selectAllPorts", DV_PROJECT){  
    code {  
        transaction {  
            domain.runtimeSystem {  
                def selectedPorts =  
                    selectComponentPorts {  
                        // no predicates: select ALL component ports  
                        } getComponentPorts()  
                scriptLogger.infoFormat("Selected {0} component ports.", selectedPorts.  
                    size())  
            }  
        }  
    }  
}
```

Listing 4.213: Selects all component ports

```
scriptTask("selectAllUnconnectedPorts", DV_PROJECT){  
    code {  
        transaction {  
            domain.runtimeSystem {  
                def selectedPorts =  
                    selectComponentPorts {  
                        unconnected() // select all unconnected component ports  
                        } getComponentPorts()  
                scriptLogger.infoFormat("Selected {0} component ports.", selectedPorts.  
                    size())  
            }  
        }  
    }  
}
```

Listing 4.214: Selects all unconnected component ports

```

scriptTask("selectAllUnconnectedSRAAndConnectedModePorts", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def selectedPorts =
                    selectComponentPorts {
                        // start with logical OR
                        or {
                            and { // unconnected sender/receiver ports
                                unconnected()
                                senderReceiver()
                            }
                            and { // connected modeSwitch ports
                                connected()
                                modeSwitch()
                            }
                        }
                    } getComponentPorts()
                scriptLogger.infoFormat("Selected {0} component ports.", selectedPorts.size())
            }
        }
    }
}

```

Listing 4.215: Select all unconnected sender/receiver or connected mode-switch component ports

```

scriptTask("selectNotCompletedPorts", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def selectedPorts =
                    selectComponentPorts {
                        // this predicate filters for ports
                        // which needs to be connected and/or a data mapping
                        notCompleted()
                    } getComponentPorts()
                scriptLogger.infoFormat("Selected {0} component ports.", selectedPorts.size())
            }
        }
    }
}

```

Listing 4.216: Selects not completed component ports

```

scriptTask ("selectComponentPortsUsingOriginContextPredicates", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {

                // we want to use information from the structured extract to
                // select flat extract ports

                def selectedComponentPorts = selectComponentPorts {
                    // use component port related predicates
                    senderReceiver()
                    required()

                    // combine them with origin context related predicates
                    // but remember that we are still selecting flat extract
                    // ports here
                    // we only use the origin context ports as additional
                    // criteria

                    // we want only ports for which the connection ends at the
                    // highest composition level inside the top level
                    // composition
                    // of the structured extract
                    hasInnerTopLevelDelegationOriginComponentPort()

                    // and only ports whose origin context has a special name
                    // pattern for their composition owner instance
                    originComponentPortComponent(~"Origin.*")
                }.getComponentPorts()

                scriptLogger.infoFormat("Selected {0} component ports using
                    their origin context as additional selection criteria.",
                    selectedComponentPorts.size())
            }
        }
    }
}

```

Listing 4.217: Use origin context predicates for selecting component ports

#### 4.10.4.2 Signal Instance Selection

The system signals and system signal groups to be data-mapped are represented by a signal instance (`IAbstractSignalInstance`). `ISignalInstance` represents a system signal, `ISignalGroupInstance` represents a system signal group. 'Signal instance' means that the system signal or system signal group is at least referenced by one `ISignal` or `ISignalGroup`. System signals or system signal groups which are not referenced by an `ISignal` or `ISignalGroup` are not represented as signal instance and so are not available for data mapping.

`selectSignalInstances(Closure)` allows the selection of `IAbstractSignalInstances` using predicates.

The signal instance selection can be used to select and filter signal instances and either do further operations on them, such as map them to communication elements or to just return a list of signal instances with which you can continue working.

`getSignalInstances()` allows access to the single signal instances in the `ISignalInstanceSelection`.

**Signal Instance Predicates** To select signal instances predicates can be provided to narrow down the result.

Per default the predicates are combined via logical AND. To realize other combinations, use the 'or', 'not' and 'and' predicates.

- `unmapped()` matches signal instances which are not data-mapped.
- `mapped()` matches signal instances which are data-mapped.
- `signalGroup()` matches signal instances which are a signal group instance.
- `groupSignal()` matches signal instances which are a group signal.
- `transformed()` matches signal instances which are transformation signals.
- `tx()` matches signal instances whose direction is compatible to `EDirection.Tx`.
- `rx()` matches signal instances whose direction is compatible to `EDirection.Rx`.
- `name(String)` matches signal instances with the given name.
- `names(Collection)` matches signal instances with the given names. The order of the names is not relevant in any kind.
- `name(Pattern)` matches signal instances with the given name pattern.
- `asrPath(String)` matches signal instances with the given autosar path.
- `asrPaths(Collection)` matches signal instances with the given autosar paths. The order of the names is not relevant in any kind.
- `asrPath(Pattern)` matches signal instances with the given autosar path pattern.
- `iSignal(String)` matches signal instances which are referenced at least by one `ISignal/I-SignalGroup` with the given name.
- `iSignal(Pattern)` matches signal instances which are referenced at least by one `ISignal/I-SignalGroup` with the given name pattern.
- `iSignalAsrPath(String)` matches signal instances which are referenced at least by one `ISignal/I-SignalGroup` with the given autosar path.
- `iSignalAsrPath(Pattern)` matches signal instances which are referenced at least by one `ISignal/I-SignalGroup` with the given autosar path pattern.
- `physicalChannel(String)` matches signal instances which are referenced by at least an `ISignal/I-SignalGroup` for which an `ISignalTriggering` exists for a `PhysicalChannel` with the given name.
- `physicalChannel(Pattern)` matches signal instances which are referenced by at least an `ISignal/I-SignalGroup` for which an `ISignalTriggering` exists for a `PhysicalChannel` with the given name pattern.
- `physicalChannelAsrPath(String)` matches signal instances which are referenced by at least an `ISignal/I-SignalGroup` for which an `ISignalTriggering` exists for a `PhysicalChannel` with the given autosar path.
- `physicalChannelAsrPath(Pattern)` matches signal instances which are referenced by at least an `ISignal/I-SignalGroup` for which an `ISignalTriggering` exists for a `PhysicalChannel` with the given autosar path pattern.

- `communicationCluster(String)` matches signal instances which are referenced by at least an ISignal/ISignalGroup which is sent via a PhysicalChannel of a CommunicationCluster with the given name.
- `communicationCluster(Pattern)` matches signal instances which are referenced by at least an ISignal/ISignalGroup which is sent via a PhysicalChannel of a CommunicationCluster with the given name pattern.
- `communicationClusterAsrPath(String)` matches signal instances which are referenced by at least an ISignal/ISignalGroup which is sent via a PhysicalChannel of a CommunicationCluster with the given autosar path.
- `communicationClusterAsrPath(Pattern)` matches signal instances which are referenced by at least an ISignal/ISignalGroup which is sent via a PhysicalChannel of a CommunicationCluster with the given autosar path pattern.
- `pdu(String)` matches signal instances which are referenced by at least an ISignal/ISignalGroup for which an ISignalToIPduMapping exists for a Pdu with the given name.
- `pdu(Pattern)` matches signal instances which are referenced by at least an ISignal/ISignalGroup for which an ISignalToIPduMapping exists for a Pdu with the given name pattern.
- `pduAsrPath(String)` matches signal instances which are referenced by at least an ISignal/ISignalGroup for which an ISignalToIPduMapping exists for a Pdu with the given autosar path.
- `pduAsrPath(Pattern)` matches signal instances which are referenced by at least an ISignal/ISignalGroup for which an ISignalToIPduMapping exists for a Pdu with the given autosar path pattern.
- `frame(String)` matches signal instances which are referenced by at least an ISignal/ISignalGroup which is sent via a Pdu for that a PduToFrameMapping exists for a Frame with the given name.
- `frame(Pattern)` matches signal instances which are referenced by at least an ISignal/ISignalGroup which is sent via a Pdu for that a PduToFrameMapping exists for a Frame with the given name pattern.
- `frameAsrPath(String)` matches signal instances which are referenced by at least an ISignal/ISignalGroup which is sent via a Pdu for that a PduToFrameMapping exists for a Frame with the given autosar path.
- `frameAsrPath(Pattern)` matches signal instances which are referenced by at least an ISignal/ISignalGroup which is sent via a Pdu for that a PduToFrameMapping exists for a Frame with the given autosar path pattern.
- `filterAdvanced(Closure)` matches signal instances for which the given closure results to true.
- `and(Closure)` combines the predicates inside the closure with a logical AND.
- `or(Closure)` combines the predicates inside the closure with a logical OR.
- `not(Closure)` negates the combination of predicates inside the closure.
- `put(List)` can be used to set `IAbstractSignalInstances` into the selection. This is the most efficient way to create a selection from existing objects. If further predicates are specified the predicates will be applied only on the signal instances that were given into

this put(List) method. The iteration order is relevant for getting deterministic results on further usage of the selection API. This method should only be called once.

## Examples

```
scriptTask("SelectAllUnmappedSignalInstances", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def signalInstances =
                    selectSignalInstances {
                        unmapped() // select all signal instances which are not yet
                        data mapped
                    } getSignalInstances()
                scriptLogger.infoFormat("Selected {0} signal instances.",
                    signalInstances.size())
            }
        }
    }
}
```

Listing 4.218: Select all unmapped signal instances

```
scriptTask("SelectAllUnmappedRxOrTransformedSignalInstances", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def signalInstances =
                    selectSignalInstances {
                        // the signal instances should not be data-mapped yet
                        unmapped()
                        or { // and should either be a rx signal or a transformation
                            signal
                            rx()
                            transformed()
                        }
                    } getSignalInstances()
                scriptLogger.infoFormat("Selected {0} signal instances.",
                    signalInstances.size())
            }
        }
    }
}
```

Listing 4.219: Select all unmapped rx or transformed signal instances

```

import com.vector.cfg.model.sysdesc.api.com.IAbstractSignalInstance
scriptTask("SelectSignalInstancesUsingAdvancedFilter", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def signalInstances =
                    selectSignalInstances {
                        filterAdvanced { IAbstractSignalInstance signalInstance ->
                            // implement own custom filter
                            def mdfObject = signalInstance.getMdfObject()
                            // work on directly on autosar model level ...
                            // select signal instance only which has admin data
                            def select = false
                            mdfObject.adminData {
                                select = true
                            }
                            select
                        }
                        } getSignalInstances()
                        scriptLogger.infoFormat("Selected {0} signal instances.", signalInstances.size())
                    }
                }
            }
        }
    }
}

```

Listing 4.220: Select signal instances using an advanced filter

#### 4.10.4.3 Communication Element Selection

A data element, an operation or a trigger to be data-mapped is represented by an **ICommunicationElement**. A data element is represented by the subtype **IDataCommunicationElement**, an operation is represented by the subtype **IOperationCommunicationElement** and a trigger is represented by the subtype **ITriggerCommunicationElement**. A communication element contains the full context information (component prototype, port prototype, data type hierarchy) necessary for data mapping.

**selectCommunicationElements(Closure)** allows the selection of **ICommunicationElements** using predicates.

The communication element selection can be used to select and filter communication elements and either do further operations on them, such as map them to signal instances or to just return a list of communication elements with which you can continue working.

**getCommunicationElements()** allows access to the single communication elements in the **ICommunicationElementSelection**. Please make sure you are using the correct expansion mode, see **ICommunicationElementSelector.selectFullyExpanded()** and **ICommunicationElementSelector.selectFullyExpandedButPrimitiveArraysAsLeafs()**.

**Communication Element Predicates** To select communication elements predicates can be provided to narrow down the result.

Per default the predicates are combined via logical AND. To realize other combinations, use the 'or', 'not' and 'and' predicates.

- **unconnected()** matches communication elements whose component port is unconnected.

- `connected()` matches communication elements whose component port is connected.
- `senderReceiver()` matches communication elements whose port has a sender/receiver port interface.
- `clientServer()` matches communication elements whose port has a client/server port interface.
- `trigger()` matches communication elements whose port has a trigger port interface.
- `provided()` matches communication elements whose port is a provided port (p-port).
- `required()` matches communication elements whose port is a required port (r-port).
- `delegation()` matches communication elements whose port is delegation port.
- `unmapped()` matches communication elements whose are not data-mapped.
- `mapped()` matches communication elements whose are data-mapped.
- `ownerPortTerminated()` matches communication elements whose component port is terminated.
- `ownerPortNotTerminated()` matches communication elements whose component port is not terminated.
- `name(String)` matches communication elements with the given data element or operation name.
- `names(Collection)` matches communication elements with the given data element or operation names. The order of the names is not relevant in any kind.
- `name(Pattern)` matches communication elements with the given data element or operation name pattern.
- `fullyQualifiedName(String)` matches communication elements with the given full qualified communication element name. E.g. 'App1.Port1.DataElement1', 'ECU Composition.DelegationPort2.DataElement2'. See also `ICommunicationElement.getFullyQualifiedName()`.
- `fullyQualifiedNames(Collection)` matches communication elements with the given full qualified communication element names. E.g. 'App1.Port1.DataElement1', 'ECU Composition.DelegationPort2.DataElement2'. See also `ICommunicationElement.getFullyQualifiedName()`. The order of the names is not relevant in any kind.
- `asrPath(String)` matches communication elements with the given data element or operation autosar path.
- `asrPath(Pattern)` matches communication elements with the given data element or operation autosar path pattern.
- `component(String)` matches communication elements with the given component name.
- `components(Collection)` matches communication elements with the given component names. The order of the names is not relevant in any kind.
- `component(Pattern)` matches communication elements with the given component name pattern.
- `componentAsrPath(String)` matches communication elements with the given component name autosar path.

- `componentAsrPath(Pattern)` matches communication elements with the given component name autosar path pattern.
- `port(String)` matches communication elements with the given component port name.
- `ports(Collection)` matches communication elements with the given component port names. The order of the names is not relevant in any kind.
- `port(Pattern)` matches communication elements with the given component port name pattern.
- `portAsrPath(String)` matches communication elements with the given component port autosar path.
- `portAsrPath(Pattern)` matches communication elements with the given component port autosar path pattern.
- `filterAdvanced(Closure)` Add a custom predicated which matches communication elements for which the given closure results to true.
- `and(Closure)` combines the predicates inside the closure with a logical AND.
- `or(Closure)` combines the predicates inside the closure with a logical OR.
- `not(Closure)` negates the combination of predicates inside the closure.
- `put(List)` can be used to set `ICommunicationElements` into the selection. This is the most efficient way to create a selection from existing objects. If further predicates are specified the predicates will be applied only on the communication elements that were given into this `put(List)` method. The iteration order is relevant for getting deterministic results on further usage of the selection API. This method should only be called once.
- `selectFullyExpanded()` modifies the behavior of the selection. Using this option you are not only selecting the root communication elements (as you know from the data mapping assistant in GUI), but also the leafs. See `IDataCommunicationElement.getLeafsFullExpanded()` for more details.

@exception SelectionException if both `selectFullyExpanded()` and `selectFullyExpandedButPrimitiveArraysAsLeafs()` are called.
- `selectFullyExpandedButPrimitiveArraysAsLeafs()` modifies the behavior of the selection. Using this option you are not only selecting the root communication elements (as you know from the data mapping assistant in GUI), but also the leafs. Arrays of primitives (e.g. uint8 arrays) will not be expanded. See `IDataCommunicationElement.getLeafsFullExpandedExceptPrimitiveArrays()` for more details.

@exception SelectionException if both `selectFullyExpanded()` and `selectFullyExpandedButPrimitiveArraysAsLeafs()` are called.

## Examples

```
scriptTask("SelectAllUnmappedDelPortComElements", DV_PROJECT){  
    code {  
        transaction {  
            domain.runtimeSystem {  
                def comElements =  
                    selectCommunicationElements {  
                        // select all unmapped delegation communication elements  
                        delegation()  
                        unmapped()  
                    } getCommunicationElements()  
                scriptLogger.infoFormat("Selected {0} communication elements.",  
                    comElements.size())  
            }  
        }  
    }  
}
```

Listing 4.221: Select all unmapped delegation port communication elements

```
scriptTask("SelectComplexFullyExpanded", DV_PROJECT){  
    code {  
        transaction {  
            domain.runtimeSystem {  
                def comElements =  
                    selectCommunicationElements {  
                        // expand all complex communication elements except  
                        // primitive arrays  
                        // and select all communication elements with their leafs  
                        // e.g. for a record we select both here, the record and its  
                        // record elements  
                        selectFullyExpandedButPrimitiveArraysAsLeafs()  
                    } getCommunicationElements()  
                scriptLogger.infoFormat("Selected {0} communication elements.",  
                    comElements.size())  
            }  
        }  
    }  
}
```

Listing 4.222: Select all communication elements with their leafs

```

import com.vector.cfg.model.sysdesc.api.com.ICommunicationElement
import com.vector.cfg.model.sysdesc.api.com.IDataCommunicationElement
scriptTask("SelectComElementsUsingAdvancedFilter", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def comElements =
                    selectCommunicationElements {
                        // advanced filter:
                        // only select communication elements
                        // which represent data elements of a specific data type
                        filterAdvanced { ICommunicationElement comElement ->
                            if (comElement instanceof IDataCommunicationElement) {
                                def mdfDataElement = comElement.
                                    getDataElementOrOperationMdfObject()
                                // check directly on autosar model level
                                return mdfDataElement.type.refTarget.name.equals(
                                    "myCustomDataType")
                            }
                            false
                        }
                    } getCommunicationElements()
                scriptLogger.infoFormat("Selected {0} communication elements.",
                    comElements.size())
            }
        }
    }
}

```

Listing 4.223: Select communication elements using an advanced filter

#### 4.10.4.4 Component Type Selection

```

public interface IComponentType extends IIIdentifiable<IComponentTypeAnchor>,
    com.vector.cfg.model.sysdesc.api.component.IComponentType {
}

```

Listing 4.224: Example of the IComponentType model abstraction

`selectComponentTypes(Closure)` allows the selection of `IComponentTypes` using predicates.

The component type selection can be used to select and filter component types and either do further operations on them, such as instantiating them by creating new component prototypes or to just return a list of component types with which you can continue working.

`getComponentTypes()` allows access to the single component types in the `IComponentTypeSelection`.

**Component Type Predicates** To select the component types predicates can be provided to narrow down the result.

Per default the predicates are combined via logical AND. To realize other combinations, use the 'or', 'not' and 'and' predicates.

- `name(String)` matches component types with the given component type name.

- `names(Collection)` matches component types with the given component type names. The order of the names is not relevant in any kind.
- `name(Pattern)` matches component types with the given component type name pattern.
- `asrPath(String)` matches component types with the given component type autosar path.
- `asrPath(Pattern)` matches component types with the given component type autosar path pattern.
- `component(String)` matches component types for which a component prototype with the given component name exists.
- `component(Pattern)` matches component types for which a component prototype with the given component name pattern exists.
- `application()` matches component types which are application component types. Application component types are all component types which are not service component types, as displayed in the ECU Software Components Editor, not ApplicationSwComponentTypes as defined by AUTOSAR.
- `service()` matches component types which are service component types.
- `parameter()` matches component types which are parameter (calibration) component types.
- `nvBlock()` matches component types which are nv block component types.
- `sensorActuator()` matches component types which are sensor actuator component types.
- `ioHwAbstraction()` matches component types which are I/O hardware abstraction component types, also called EcuAbstractionSwComponentType.
- `complexDeviceDriver()` matches component types which are complex device driver component types.
- `serviceProxy()` matches component types which are service proxy component types.
- `instantiated()` matches component types that are already instantiated. In other words matches if a component prototype of that component type already exists.
- `supportsMultipleInstantiation()` matches component types which support multiple instantiation.
- `filterAdvanced(Closure)` matches component types for which the given closure results to true.
- `and(Closure)` combines the predicates inside the closure with a logical AND.
- `or(Closure)` combines the predicates inside the closure with a logical OR.
- `not(Closure)` negates the combination of predicates inside the closure.
- `put(List)` can be used to set IComponentTypes into the selection. This is the most efficient way to create a selection from existing objects. If further predicates are specified the predicates will be applied only on the component types that were given into this `put(List)` method. The iteration order is relevant for getting deterministic results on further usage of the selection API. This method should only be called once.

## Examples

```

scriptTask ("selectComponentTypeByName", DV_PROJECT ){
    code {
        domain.runtimeSystem {
            def selectedComponentTypes = selectComponentTypes {
                name "App1"
            }.getComponentTypes()

            scriptLogger.infoFormat("Selected '{0}' component types.",
                selectedComponentTypes.size())
        }
    }
}

```

Listing 4.225: Select component type by name

```

scriptTask ("selectNotInstantiatedComponentTypes", DV_PROJECT ){
    code {
        domain.runtimeSystem {
            def selectedComponentTypes = selectComponentTypes {
                not {
                    instantiated()
                }
            }.getComponentTypes()

            scriptLogger.infoFormat("Selected '{0}' component types.",
                selectedComponentTypes.size())
        }
    }
}

```

Listing 4.226: Select not instantiated component types

#### 4.10.4.5 Event Selection

An event `IEvent` (called `AbstractEvent` in AUTOSAR) represents a `RTEEvent` or a `BswEvent`. Events are raised on different conditions and are used to implement application or basic software in AUTOSAR. (Sometimes they are also called triggers.)

A task mapping (`ITaskMapping`) represents an `IEvent` (also called trigger) that is mapped to a task in the context of a component prototype or a module configuration. It corresponds to the task mapping container in the RTE configuration.

`selectEvents(Closure)` allows the selection of `IEvents` using predicates.

The event selection can be used to select and filter events and either do further operations on them, such as mapping the executable entities they trigger to tasks or to just return a list of events or the task mappings for them with which you can continue working.

`getEvents()` allows access to the single events in the `IEventSelection`.

`getTaskMappings()` retrieves all `ITaskMappings` for the selected events (see `getEvents()`).

Note:

1. In case of multi instantiation of component prototypes, the different instances share the same events, since the event is part of the internal behavior of the component type. Therefore if the event is selected, `getTaskMappings()` will always return the task mappings for all component prototypes.

2. Since this method can be run outside of a transaction, there might be selected events for which no task mapping container does exist yet. The container cannot be created by calling `getTaskMappings()`, so no task mapping can be returned. This happens if the system description is not synchronized, after changes in the structured extract were done (see Automation Interface Documentation, chapter about Model Synchronization for examples how to synchronize).

**Event Predicates** To select the events predicates can be provided to narrow down the result.

Per default the predicates are combined via logical AND. To realize other combinations, use the 'or', 'not' and 'and' predicates.

- `name(String)` matches events (triggers) with the given event name.
- `names(Collection)` matches events (triggers) with the given event names. The order of the names is not relevant in any kind.
- `name(Pattern)` matches events (triggers) with the given event name pattern.
- `asrPath(String)` matches events (triggers) with the given event autosar path.
- `asrPath(Pattern)` matches events (triggers) with the given event autosar path pattern.
- `component(String)` matches events (triggers) which belong to components with the given component name.
- `components(Collection)` matches events (triggers) which belong to components with the given component names. The order of the names is not relevant in any kind.
- `component(Pattern)` matches events (triggers) which belong to components which matches the given component name pattern.
- `componentType(String)` matches events (triggers) which are part of the internal behavior of component types with the given component type name.
- `componentType(Pattern)` matches events (triggers) which are part of the internal behavior of component types which matches the given component type name pattern.
- `componentTypeAsrPath(String)` matches events (triggers) which are part of the internal behavior of component types with the given component type autosar path.
- `componentTypeAsrPath(Pattern)` matches events (triggers) which are part of the internal behavior of component types whose autosar path matches the given component type autosar path pattern.
- `moduleConfiguration(String)` matches events (triggers) which belong to module configurations with the given module configuration name.
- `moduleConfigurations(Collection)` matches events (triggers) which belong to module configurations with the given module configuration names. The order of the names is not relevant in any kind.
- `moduleConfiguration(Pattern)` matches events (triggers) which belong to module configurations which matches the given module configuration name pattern.
- `moduleConfigurationAsrPath(String)` matches events (triggers) which belong to module configurations with the given module configuration autosar path.

- `moduleConfigurationAsrPath(Pattern)` matches events (triggers) which belong to module configurations whose autosar path matches the given module configuration autosar path pattern.
- `task(String)` matches events (triggers) which are mapped to a task with the given task name.
- `task(Pattern)` matches events (triggers) which are mapped to a task whose name matches the given task name pattern.
- `bswEvent()` matches events (triggers) which are bsw events.
- `rteEvent()` matches events (triggers) which are rte events.
- `unmapped()` matches unmapped events (triggers). In case of multi instantiated components/modules matches if unmapped at least in one context. Use `fullyUnmapped()` to determine whether an event is unmapped in all contexts.
- `fullyUnmapped()` matches events (triggers) which are not mapped in any context. If no multi instantiation is used, the result is the same as for `unmapped()`.
- `mapped()` matches mapped events (triggers). In case of multi instantiated components/-modules matches if mapped at least in one context. Use `fullyMapped()` to determine whether an event is mapped in all contexts.
- `fullyMapped()` matches events (triggers) which are mapped in every context. If no multi instantiation is used, the result is the same as for `mapped()`.
- `timing()` matches events which are timing events (triggers).
- `timing(BigDecimal)` matches events (triggers) which are timing events with the given period (seconds).
- `init()` matches events (triggers) which are init events.
- `dataReception()` matches events (triggers) which are data received events.
- `dataReceptionError()` matches events (triggers) which are data receive error events.
- `dataSendCompletion()` matches events (triggers) which are data send completed events.
- `operationInvoked()` matches events (triggers) which are operation invoked events.
- `operationInvoked(String)` matches operation invoked events (triggers) which are invoked by an operation with the given operationName.
- `serverCallReturns()` matches events (triggers) which are asynchronous server call returns events.
- `modeSwitch()` matches events (triggers) which are mode switch events.
- `modeEntry()` matches events (triggers) which are mode switch events with activation kind ON-ENTRY.
- `modeExit()` matches events (triggers) which are mode switch events with activation kind ON-EXIT.
- `modeTransition()` matches events (triggers) which are mode switch events with activation kind ON-TRANSITION.
- `modeSwitchedAck()` matches events (triggers) which are mode switched acknowledgement events.

- `externalTrigger()` matches events (triggers) which are external trigger occurred events.
- `internalTrigger()` matches events (triggers) which are internal trigger occurred events.
- `background()` matches events (triggers) which are background events.
- `mandatory()` matches events (triggers) which must be mapped. (The mapping of operation invoked events is optional, so this predicate filters all operation invoked events.)
- `filterAdvanced(Closure)` matches events (triggers) for which the given closure results to true.
- `and(Closure)` combines the predicates inside the closure with a logical AND.
- `or(Closure)` combines the predicates inside the closure with a logical OR.
- `not(Closure)` negates the combination of predicates inside the closure.
- `put(List)` can be used to set `IEvents` into the selection. This is the most efficient way to create a selection from existing objects. If further predicates are specified the predicates will be applied only on the events that were given into this `put(List)` method. The iteration order is relevant for getting deterministic results on further usage of the selection API. This method should only be called once.

### Examples

```
scriptTask ("selectEvents", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def selectedEvents = selectEvents {
                    // select all unmapped events of component 'App1'
                    unmapped()
                    component("App1")
                }.getEvents()

                scriptLogger.infoFormat("Selected '{0}' events.",
                    selectedEvents.size())
            }
        }
    }
}
```

Listing 4.227: Select events example

#### 4.10.4.6 Executable Entity Selection

An executable entity (`IExecutableEntity`) represents a `RunnableEntity` or a `BswSchedulableEntity`. Both are abstractions of executable code in AUTOSAR. (Sometimes they are also called functions.)

A task mapping (`ITaskMapping`) represents an `IEvent` (also called trigger) that is mapped to a task in the context of a component prototype or a module configuration. It corresponds to the task mapping container in the RTE configuration.

`selectExecutableEntities(Closure)` allows the selection of `IExecutableEntitys` using predicates.

The executable entity selection can be used to select and filter executable entities and either do further operations on them, such as mapping them to tasks or to just return a list of executable entities or the task mappings for them with which you can continue working.

`getExecutableEntities()` allows access to the single executable entities in the `IExecutableEntitySelection`.

`getTaskMappings()` retrieves all `ITaskMappings` for the selected executable entities (see `getExecutableEntities()`).

**Executable Entity Predicates** To select the executable entities predicates can be provided to narrow down the result.

Per default the predicates are combined via logical AND. To realize other combinations, use the 'or', 'not' and 'and' predicates.

- `symbol(String)` matches runnable entities with the given symbol and bsw schedulable entities whose corresponding bsw module entry short name matches the given symbol.
- `symbol(Pattern)` matches runnable entities whose symbol matches the given symbol pattern and bsw schedulable entities whose corresponding bsw module entry short name matches the given symbol pattern.
- `name(String)` matches executable entities (functions) with the given name.
- `names(Collection)` matches executable entities (functions) with the given names. The order of the names is not relevant in any kind.
- `name(Pattern)` matches executable entities (functions) with the given name pattern.
- `asrPath(String)` matches executable entities (functions) with the given autosar path.
- `asrPath(Pattern)` matches executable entities (functions) with the given autosar path pattern.
- `component(String)` matches executable entities (functions) which belong to components with the given component name.
- `components(Collection)` matches executable entities (functions) which belong to components with the given component names. The order of the names is not relevant in any kind.
- `component(Pattern)` matches executable entities (functions) which belong to components which matches the given component name pattern.
- `componentType(String)` matches executable entities (functions) which are part of the internal behavior of component types with the given component type name.
- `componentType(Pattern)` matches executable entities (functions) which are part of the internal behavior of component types which matches the given component type name pattern.
- `componentTypeAsrPath(String)` matches executable entities (functions) which are part of the internal behavior of component types with the given component type autosar path.
- `componentTypeAsrPath(Pattern)` matches executable entities (functions) which are part of the internal behavior of component types whose autosar path matches the given component type autosar path pattern.

- `moduleConfiguration(String)` matches executable entities (functions) which belong to module configurations with the given module configuration name.
- `moduleConfigurations(Collection)` matches executable entities (functions) which belong to module configurations with the given module configuration names. The order of the names is not relevant in any kind.
- `moduleConfiguration(Pattern)` matches executable entities (functions) which belong to module configurations which matches the given module configuration name pattern.
- `moduleConfigurationAsrPath(String)` matches executable entities (functions) which belong to module configurations with the given module configuration autosar path.
- `moduleConfigurationAsrPath(Pattern)` matches executable entities (functions) which belong to module configurations whose autosar path matches the given module configuration autosar path pattern.
- `task(String)` matches executable entities (functions) which have at least one event (trigger) that is mapped to a task with the given task name.
- `task(Pattern)` matches executable entities (functions) which have at least one event (trigger) that is mapped to a task whose name matches the given task name pattern.
- `bswSchedulableEntity()` matches executable entities (functions) which are bsw schedulable entities.
- `runnableEntity()` matches executable entities (functions) which are runnable entities.
- `unmapped()` matches executable entities (functions) with at least one unmapped event (trigger).
- `fullyUnmapped()` matches executable entities (functions) with all of its events (triggers) being not mapped in any context to a task.
- `mapped()` matches executable entities (functions) with at least one mapped event (trigger).
- `fullyMapped()` matches executable entities (functions) with all of its events (triggers) being mapped in each context to a task.
- `filterAdvanced(Closure)` matches executable entities (functions) for which the given closure results to true.
- `and(Closure)` combines the predicates inside the closure with a logical AND.
- `or(Closure)` combines the predicates inside the closure with a logical OR.
- `not(Closure)` negates the combination of predicates inside the closure.
- `put(List)` can be used to set `IExecutableEntity`s into the selection. This is the most efficient way to create a selection from existing objects. If further predicates are specified the predicates will be applied only on the executable entities that were given into this `put(List)` method. The iteration order is relevant for getting deterministic results on further usage of the selection API. This method should only be called once.

## Examples

```

scriptTask ("selectExecutableEntities", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def selectedExecutables = selectExecutableEntities {
                    // select all runnables with symbol 'MySymbol'
                    symbol("MySymbol")
                    runnableEntity()
                }.getExecutableEntities()

                scriptLogger.infoFormat("Selected '{0}' executable entities.",
                    selectedExecutables.size())
            }
        }
    }
}

```

Listing 4.228: Select executable entities example

#### 4.10.4.7 Port Interface Selection

A `IPortInterface` represents a port interface according to AUTOSAR. A port interface is an interface that is either provided or required by a port of a software component.

`selectPortInterfaces(Closure)` allows the selection of `IPortInterfaces` using predicates.

The port interface selection can be used to select and filter port interfaces and either do further operations on them, such as instantiating them by creating new delegation port prototypes or to just return a list of port interfaces with which you can continue working.

`getPortInterfaces()` allows access to the single port interfaces in the `IPortInterfaceSelection`.

**Port Interfaces Predicates** To select the port interfaces predicates can be provided to narrow down the result.

Per default the predicates are combined via logical AND. To realize other combinations, use the 'or', 'not' and 'and' predicates.

- `name(String)` matches port interfaces with the given port interface name.
- `names(Collection)` matches port interfaces with the given port interface names. The order of the names is not relevant in any kind.
- `name(Pattern)` matches port interfaces with the given port interface name pattern.
- `asrPath(String)` matches port interfaces with the given port interface autosar path.
- `asrPath(Pattern)` matches port interfaces with the given port interface autosar path pattern.
- `service()` matches port interfaces which are service interfaces.
- `application()` matches port interfaces which are application interfaces.
- `senderReceiver()` matches port interfaces which are sender receiver interfaces.
- `clientServer()` matches port interfaces which are client server interfaces.

- `modeSwitch()` matches port interfaces which are mode switch interfaces.
- `nvData()` matches port interfaces which are NvData interfaces.
- `trigger()` matches port interfaces which are trigger interfaces.
- `parameter()` matches port interfaces which are parameter interfaces.
- `componentType(String)` first matches all component types with the given component type name, then retrieves all port interfaces of the component type's port prototypes.
- `componentType(Pattern)` first matches all component types with the given component type name pattern, then retrieves all port interfaces of the component type's port prototypes.
- `componentTypeAsrPath(String)` first matches all component types with the given component type asr path, then retrieves all port interfaces of the component type's port prototypes.
- `componentTypeAsrPath(Pattern)` first matches all component types with the given component type asr path pattern, then retrieves all port interfaces of the component type's port prototypes.
- `component(String)` first matches all components with the given component name, then retrieves all port interfaces of the component's ports.
- `components(Collection)` first matches all components with the given component names, then retrieves all port interfaces of the component's ports. The order of the names is not relevant in any kind.
- `component(Pattern)` first matches all components with the given component name pattern, then retrieves all port interfaces of the component's ports.
- `componentPort(String)` first matches all `IComponentPorts` with the given name, then retrieves all port interfaces of the component ports.
- `componentPorts(Collection)` first matches all `IComponentPorts` with the given names, then retrieves all port interfaces of the component ports. The order of the names is not relevant in any kind.
- `componentPort(Pattern)` first matches all `IComponentPorts` with the given name pattern, then retrieves all port interfaces of the component ports.
- `filterAdvanced(Closure)` matches port interfaces for which the given closure results to true.
- `and(Closure)` combines the predicates inside the closure with a logical AND.
- `or(Closure)` combines the predicates inside the closure with a logical OR.
- `not(Closure)` negates the combination of predicates inside the closure.
- `put(List)` can be used to set `IPortInterfaces` into the selection. This is the most efficient way to create a selection from existing objects. If further predicates are specified the predicates will be applied only on the port interfaces that were given into this `put(List)` method. The iteration order is relevant for getting deterministic results on further usage of the selection API. This method should only be called once.

## Examples

```

scriptTask("selectPortInterfacesByName", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def selectedPortInterfaces =
                    selectPortInterfaces {
                        // selects all sender receiver application port interfaces with
                        // the name 'MyPortInterface'
                        senderReceiver()
                        application()
                        name "MyPortInterface"

                        // getPortInterfaces() will filter all port interfaces for the
                        // given predicates
                        // so in our example we will receive
                        // all sender receiver application port interfaces with the short
                        // name 'MyPortInterface'
                    } getPortInterfaces()
                scriptLogger.infoFormat("Selected {0} port interfaces.",
                    selectedPortInterfaces.size())
            }
        }
    }
}

```

Listing 4.229: Select PortInterface by name and type

```

scriptTask("selectPortInterfacesByComponentPorts", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def selectedPortInterfaces =
                    selectPortInterfaces {
                        // selects the port interface of the port 'pCSPort1' of
                        // component 'App1'
                        componentPort "App1.pCSPort1"
                    } getPortInterfaces()
                scriptLogger.infoFormat("Selected {0} port interfaces.",
                    selectedPortInterfaces.size())
            }
        }
    }
}

```

Listing 4.230: Select PortInterfaces by component ports

#### 4.10.4.8 Origin Component Port Selection

**Origin Context** According to AUTOSAR the flat extract is created out of the structured extract. During the flattening process, the inner compositions and their ports are lost. However sometimes the information about these objects is helpful to perform actions on the flat extract, such as for example connecting ports or doing the data mapping. We call the objects of the structured extract, which are related to the flat extract objects, origin contexts.

The component port connection provides an option to use the origin context's names as additional mapping criteria. This will be introduced below (see 4.10.4.9 on page 210).

**Origin Component Port** There is an own model element for the component ports of the structured extract. A component port (see also `IComponentPort`) represents a port prototype and its corresponding component prototype. The `IOriginComponentPort` represents a port in context of a component prototype for the upstream model (structured extract).

Further we want to clarify some of the terminology which is used in context of the origin component port. The term delegation port is pretty clear for the flat extract, since there are no other compositions beside the top level composition. However it is possible to instantiate also compositions inside the top level composition and inside other compositions in the structured extract. We call all ports whose owner is a composition, delegation ports.

Another term used here is the inner top level. Since a project can have only one top level composition, the first hierarchy level defining the rough structure of the software components is directly inside the top level composition. So everything directly inside the top level composition is called the inner top level.

For example if a composition 'MyComposition' is instantiated directly in the top level composition and has a port named 'SendData', so we call the origin component port 'MyComposition.SendData' an inner top level delegation port. Let's assume the composition named 'OtherComposition' has a port named 'ReceiveData' and is instantiated in 'MyComposition'. The origin component port 'OtherComposition.ReceiveData' is not an inner top level delegation port, since we use this term only for the hierarchy level inside the top level composition.

`selectOriginComponentPorts(Closure)` allows the selection of `IOriginComponentPorts` using predicates. The origin component ports which are selected here, are ends of incomplete delegation connections in the structured extract.

The origin component port selection can be used to select and filter origin component ports and either do further operations on them, such as creating new delegation ports in the flat extract for them or to just return a list of origin component ports with which you can continue working.

`getOriginComponentPorts()` allows access to the single origin component ports in the `IOriginComponentPortSelection`.

**Origin Component Port Predicates** To select the origin component ports predicates can be provided to narrow down the result.

Per default the predicates are combined via logical AND. To realize other combinations, use the 'or', 'not' and 'and' predicates.

- `name(String)` matches origin component ports with the given port name.
- `names(Collection)` matches origin component ports with the given port names. The order of the names is not relevant in any kind.
- `name(Pattern)` matches origin component ports with the given port name pattern.
- `asrPath(String)` matches origin component ports with the given port autosar path.
- `asrPath(Pattern)` matches origin component ports with the given port autosar path pattern.
- `component(String)` matches origin component ports with the given component name.
- `components(Collection)` matches origin component ports with the given component names. The order of the names is not relevant in any kind.

- `component(Pattern)` matches origin component ports with the given component name pattern.
- `componentAsrPath(String)` matches the origin component ports with the given component autosar path.
- `componentAsrPath(Pattern)` matches origin component ports with the given component autosar path pattern.
- `componentType(String)` matches origin component ports whose component type's name equals the given component type name.
- `componentType(Pattern)` matches origin component ports whose component type's name matches the given component type name pattern.
- `componentTypeAsrPath(String)` matches origin component ports whose component type's autosar path equals the given component type autosar path.
- `componentTypeAsrPath(Pattern)` matches origin component ports whose component type's autosar path matches the given component type autosar path pattern.
- `provided()` matches provided origin component ports (p-port).
- `required()` matches required origin component ports (r-port).
- `providedRequired()` matches provided-required origin component ports (pr-port).
- `innerTopLevelDelegation()` matches origin component ports on the highest hierarchy level. In other words the component of the matched ports is instantiated directly inside the top level composition (ECU Composition).
- `ofFlatExtractPort()` retrieves the origin component ports of the given flatComponent-Port.
- `senderReceiver()` matches origin component ports whose port has a sender/receiver port interface.
- `clientServer()` matches origin component ports whose port has a client/server port interface.
- `trigger()` matches origin component ports whose port has a trigger port interface.
- `filterAdvanced(Closure)` matches origin component ports for which the given closure results to true.
- `and(Closure)` combines the predicates inside the closure with a logical AND.
- `or(Closure)` combines the predicates inside the closure with a logical OR.
- `not(Closure)` negates the combination of predicates inside the closure.
- `completed()` matches origin component ports which are completed.

An `IOOriginComponentPort` is completed if an only if all of its flat extract ports and additionally all of the delegation ports which are connected to these flat extract ports are completed. (See `IComponentPort` for definition of completed state for flat extract ports.)

- `notCompleted()` matches origin component ports which are not completed.

See `completed()` for the conditions an origin port has to meet to be a completed port.

- `put(List)` can be used to set `IOriginComponentPorts` into the selection. This is the most efficient way to create a selection from existing objects. If further predicates are specified the predicates will be applied only on the origin component ports that were given into this `put(List)` method. The iteration order is relevant for getting deterministic results on further usage of the selection API. This method should only be called once.

### Examples

```
scriptTask("selectOriginComponentPorts", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def selectedOriginPorts = selectOriginComponentPorts {
                    innerTopLevelDelegation()
                    provided()
                } getOriginComponentPorts()

                scriptLogger.infoFormat("Selected '{0}' origin component ports.",
                    selectedOriginPorts.size())
            }
        }
    }
}
```

Listing 4.231: Select ends of incomplete connections

```
scriptTask("originPortsForFlatExtractPort", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                // first we need to find our flat extract port
                // we will use the component port selection for this example
                def componentPort = selectComponentPorts {
                    name "pDataSend"
                    component "App1"
                } getComponentPorts().iterator().next()

                def selectedOriginPorts = selectOriginComponentPorts {
                    ofFlatExtractPort(componentPort)
                } getOriginComponentPorts()

                scriptLogger.infoFormat("Selected '{0}' origin component ports for {1}.
                    ",
                    selectedOriginPorts.size(),
                    componentPort.getName())
            }
        }
    }
}
```

Listing 4.232: Select the ends of incomplete connections for a specific flat extract component port

#### 4.10.4.9 Component Port Connection

This chapter is about connecting (a.k.a. mapping) component ports to other component ports. There are two ways to do that.

The first way is using a the component port selection API (see 4.10.4.1 on page 176) and calling

a method to connect the selected ports to other ports. It is possible to filter the targets and to evaluate and change by the auto-mapper suggested connections. So this way is similar to the component connection assistant from the GUI.

The second way is to use a simple API that requires already prepared data structures e.g. two lists of component ports that are sorted applying certain custom rules and to map them via index matching. To initially find the appropriate component ports you can use the common selection APIs.

**Auto-Mapping** The use case of auto-mapping component ports is based on the selection of component ports. The auto-mapper matches component ports using their names.

`autoMap()` tries to auto-map the selection of component ports according the component connection assistant default rules.

### Examples for `autoMap()`

```
scriptTask("automapAll", DV_PROJECT) {
    code {
        transaction {
            domain.runtimeSystem {
                def mappedConnectors =
                    selectComponentPorts {
                        // no predicates: select ALL component ports
                    } autoMap()
                scriptLogger.infoFormat("Created {0} mappings.", mappedConnectors.size
                    ())
            }
        }
    }
}
```

Listing 4.233: Tries to auto-map all ports

```
scriptTask("automapAllUnconnected", DV_PROJECT) {
    code {
        transaction {
            domain.runtimeSystem {
                def mappedConnectors =
                    selectComponentPorts {
                        unconnected() // select all unconnected component ports
                    } autoMap()
                scriptLogger.infoFormat("Created {0} mappings.", mappedConnectors.size
                    ())
            }
        }
    }
}
```

Listing 4.234: Tries to auto-map all unconnected component ports

```

scriptTask("autoMapUnconnectedSRCS", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def mappedConnectors =
                    selectComponentPorts {
                        // select all unconnected client/server and unconnected
                        // sender/receiver ports
                        unconnected()
                        or {
                            clientServer()
                            senderReceiver()
                        }
                    } autoMap()
                scriptLogger.infoFormat("Created {0} mappings.", mappedConnectors.size())
            }
        }
    }
}

```

Listing 4.235: Tries to auto-map all unconnected sender/receiver and client/server ports

```

import com.vector.cfg.model.sysdesc.api.port.IComponentPort

scriptTask("autoMapAdvancedfilter", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def mappedConnectors =
                    selectComponentPorts {
                        // select component port by own custom filter predicate
                        filterAdvanced {IComponentPort port ->
                            "MyUUID".equals(port.getMdfPort().getUuid2())
                        }
                    } autoMap()
                scriptLogger.infoFormat("Created {0} mappings.", mappedConnectors.size())
            }
        }
    }
}

```

Listing 4.236: Tries to auto-map port determined by advanced filter

**autoMapTo(Closure)** tries to auto-map the selection of component ports according the component connection assistant rules but offers more control for the auto-mapping: Inside the closure additional predicates for narrowing down the target component ports can be defined and code to evaluate and change the auto-mapper results can be provided.

Narrowing down the target component ports may be useful to gain better matches for the auto-mapper: In case several target component ports match equally, no auto-mapping is performed. So reducing the target component ports may improve the results of the auto-mapping.

The component port selection will produce trace, info and warning logs. To see them, activate the 'com.vector.cfg.dom.runtimesys.groovy.api.IComponentPortSelection' logger with the appropriate log level.

#### Control the auto-mapping in autoMapTo(Closure)

**selectTargetPorts(Closure)** allows to define predicates to narrow down the target ports for

the auto-mapping. The predicates are used to filter the possible target component ports which were computed from the source component port selection.

```
scriptTask("autoMapUnconnectedToComponentPrototype", DV_PROJECT){  
    code {  
        transaction {  
            domain.runtimeSystem {  
                def mappedConnectors =  
                    selectComponentPorts {  
                        unconnected() // select all unconnected ports  
                    } autoMapTo {  
                        selectTargetPorts {  
                            component "App1" // and auto-map them to all ports of  
                            component "App1"  
                        }  
                    }  
                scriptLogger.infoFormat("Created {0} mappings.", mappedConnectors.size())  
            }  
        }  
    }  
}
```

Listing 4.237: Tries to auto map all unconnected ports to the ports of one component prototype

`evaluateMatches(Closure)` allows to evaluate and change the results of the auto-mapping. It corresponds to the confirm page of the component connection assistant.

For each source component port the provided closure is called: Parameters are the source component port, the optional matched target component port (or null), and a list of all potential target component ports (respecting the `selectTargetPorts(Closure)` predicates). The return value must be a list of target component ports.

```
import com.vector.cfg.dom.runtimesys.api.assistant.connection.  
    ISourceComponentPort  
import com.vector.cfg.dom.runtimesys.api.assistant.connection.  
    ITargetComponentPort  
  
scriptTask("automapAllUnconnectedAndEvaluateMatches", DV_PROJECT){  
    code {  
        transaction {  
            domain.runtimeSystem {  
                def mappedConnectors =  
                    selectComponentPorts {  
                        unconnected()  
                    } autoMapTo {  
                        evaluateMatches {  
                            ISourceComponentPort sourcePort,  
                            ITargetComponentPort optionalMatchedTargetPort,  
                            List<ITargetComponentPort> potentialTargetPorts ->  
                                if (sourcePort.getPortName().equals("MyExceptionalPort")) {  
                                    // example for excluding a port from auto-mapping  
                                    // by having a close look  
                                    // sourcePort.getMdfPort()....  
                                    return null  
                                }  
                                // default: do not change the auto-matched port  
                                [optionalMatchedTargetPort]  
                            }  
                        }  
                    }  
                scriptLogger.infoFormat("Created {0} mappings.",mappedConnectors.size())  
            }  
        }  
    }  
}
```

Listing 4.238: Tries to auto-map all unconnected ports and evaluate matches

```
import com.vector.cfg.dom.runtimesys.api.assistant.connection.  
    ISourceComponentPort  
import com.vector.cfg.dom.runtimesys.api.assistant.connection.  
    ITargetComponentPort  
  
scriptTask("anotherExampleForUsingEvaluateMatches", DV_PROJECT){  
    code {  
        transaction {  
            domain.runtimeSystem {  
                def mappedConnectors =  
                    selectComponentPorts {  
                        unconnected()  
                            } autoMapTo {  
                                evaluateMatches {  
                                    ISourceComponentPort sourcePort,  
                                    ITargetComponentPort optionalMatchedTargetPort,  
                                    List<ITargetComponentPort> potentialTargetPorts ->  
  
                                    // iterate over potential target ports to find the  
                                    // correct target  
  
                                    // like in java you can use a for loop  
                                    for (ITargetComponentPort targetCP :  
                                        potentialTargetPorts) {  
                                        if (targetCP.getPortName().startsWith("MyPort_"))  
                                            {  
                                                return [targetCP]  
                                            }  
                                        }  
  
                                    // or you can use a stream  
                                    def myTargets = potentialTargetPorts.findAll {  
                                        it.getPortName().startsWith("OtherPort_")  
                                    }  
                                    return myTargets  
                                }  
                            }  
                scriptLogger.infoFormat("Created {0} mappings.",mappedConnectors.  
                    size())  
            }  
        }  
    }  
}
```

Listing 4.239: Another example for using evaluate matches

```

import com.vector.cfg.dom.runtimesys.api.assistant.connection.
ISourceComponentPort
import com.vector.cfg.dom.runtimesys.api.assistant.connection.
ITargetComponentPort

scriptTask("automap1ToN", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def mappedConnectors =
                    selectComponentPorts {
                        // select single delegation port
                        delegation()
                        name "rDelegationSRPort1"
                    } autoMapTo {
                        selectTargetPorts {
                            // select a collection of target ports (names start with
                            // "rSRPort")
                            name ~"rSRPort.*"
                        }
                        evaluateMatches {
                            ISourceComponentPort sourcePort,
                            ITargetComponentPort optionalMatchedTargetPort,
                            List<ITargetComponentPort> potentialTargetPorts ->
                                // return all potentialTargetPorts for 1:n
                                // connections, not only the one matched best
                                potentialTargetPorts
                        }
                    }
                    scriptLogger.infoFormat("Created {0} mappings.",mappedConnectors.size())
                }
            }
        }
    }
}

```

Listing 4.240: Auto-map a component port and realize 1:n connection by using evaluate matches

`forceConnectionWhen1To1()` allows to force a mapping even the usual auto-mapping rules will not match. Precondition is that the collections of source component ports and target component ports only contain one component port each. Otherwise no mapping is done.

```

scriptTask("autoMapTwoNonMatchingPorts", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def mappedConnectors =
                    selectComponentPorts {
                        // select a single source component port
                        name "prNVPo1"
                        component "NvApp1"
                    } autoMapTo {
                        selectTargetPorts {
                            // select a single target component port
                            name "rSRPort2"
                            component "App2"
                        }
                        // force the connection even names do not match at all
                        forceConnectionWhen1To1()
                    }
                scriptLogger.infoFormat("Created {0} mappings.", mappedConnectors.size())
            }
        }
    }
}

```

Listing 4.241: Create mapping between two ports which names do not match.

`useOriginContextForMatch()` uses another algorithm to match the component ports. The standard algorithm matches only the names of the ports (delegation port of flat extract or port of a SWC of the flat extract). This option uses also the origin context's name for the name matching.

#### Incomplete Connections:

Below we will talk about complete and incomplete connections. A connection is complete if the port of a SWC is connected to another SWC port or to a delegation port of the top level ECU composition. A connection is incomplete if a SWC port is connected to a delegation port of a composition which is not the top level ECU composition and the connection stops at this port (the delegation port is unconnected on the other side).

#### Origin Context:

Origin contexts of an inner port are delegation ports of the structured extract which are connected to this port and are the outermost ports of an incomplete connection. Delegation ports of the flat extract cannot have any origin contexts.

#### Example:

We want to map the port 'pData' of 'App1' of the flat extract. The corresponding component in the structured extract is instantiated inside the composition 'Comp1'. 'pData' is connected to the port 'pOriginContext' of 'Comp1'. 'pOriginContext' has no further ports connected to it.

When not using `useOriginContextForMatch()` option, only 'pData' would be used for the port name matching.

When using the `useOriginContextForMatch()` option, not only (but also) the name 'pData' is used for the port name matching, but also the origin context 'pOriginContext'.

```

scriptTask("mapUsingOriginContext", DV_PROJECT) {
    code {
        transaction {
            domain.runtimeSystem {
                def createdConnections = selectComponentPorts {
                    component("App1OriginContextMatch")
                    name("A")
                } autoMapTo {
                    // this option will not only match the name of the port 'A'
                    // but also the delegation ports' names of incomplete connections
                    // of the structured extract
                    useOriginContextForMatch()
                }

                scriptLogger.infoFormat("Created '{0}' connections.",
                    createdConnections.size())
            }
        }
    }
}

```

Listing 4.242: Use the origin context for the port name matching

**Simple API for connection between ports** This API covers the use case when the names of the components and ports which should be connected are known, but the naming rules are too specific or if you just want to increase the level of control by giving exactly the pairs that should be mapped into the API. There is one method for connecting exactly one component port to another and one API for doing multiple connections at once requiring a list of source and a list of target ports.

`componentPort(String, String)` allows to select an `IComponentPort` and to do further operations on it, e.g. connecting the port to another port.

`ISelectedComponentPort` represents a selection of exactly one `IComponentPort` and provides further actions on it.

`connectTo(String, String, Optional)` creates an `IConnector` between the `IComponentPort` which is represented by this `ISelectedComponentPort` and the `IComponentPort` which is retrieved by the given component and port name. It is possible to connect the component port to a delegation port by using 'COMPOSITIONTYPE' as componentName.

The API provides only the very basic checks.

- Direction of the ports is checked. E.g. it is not allowed to connect two PPorts within an AssemblySwConnector.
- Connecting incompatible types of port interfaces is not allowed. E.g. it is not allowed to connect a mode switch with a sender receiver port.
- Connecting two delegation ports is not allowed.
- The port interface mapping has to reference the port interfaces of the selected component ports.
- Connecting a terminated port is not allowed. Please remove the port terminator first. Use `ISelectedComponentPort.removePortTerminator()`.
- Creating redundant connections is not allowed. The ports cannot be connected by a second connector.

If you want to use some internal rules other than name matching to connect ports you can apply this rules to sort a list of source and another list of target ports and then use the simple API below that connects two lists of ports via index.

`connectTo(List)` creates `IConnectors` between the `IComponentPorts` which are represented by this `ISelectedComponentPorts` with the given `targetPorts`. The ports are connected using the index. The first port of this `ISelectedComponentPorts` will be connected to the first target port, the second to the second target port and so on.

In case you want to ignore already connected port pairs please use `assureConnectedTo(List)`.

If you want to connect ports using name matching please use `IRuntimeSystemApi.selectComponentPorts(group)`.

`assureConnectedTo(List)` checks whether the `IComponentPorts` represented by this `ISelectedComponentPorts` are already connected to the given `targetPorts` matching them via index. In contrast to `connectTo(List)` this method does nothing if the ports are already connected. If the ports are not connected a new connector will be created.

## Examples

```
import java.util.Optional

scriptTask ("assemblyConnectionExample", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {

                // enter component and port name of a component port and the
                // component port to connect it to
                // optionally you can add a port interface mapping
                def createdConnector = componentPort("App1", "pDataSend").
                    connectTo("App2", "rSRPort2", Optional.empty())

                scriptLogger.infoFormat("Created a connection between '{0}' and
                    '{1}'.",
                    createdConnector.getProvidedPort().getName(),
                    createdConnector.getRequiredPort().getName())
            }
        }
    }
}
```

Listing 4.243: Example how to create a simple assembly connection

```

import java.util.Optional

scriptTask ("delegationConnectionExample", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                // to select a delegation port the name of the ECU Composition
                // is used as component name
                def createdConnector = componentPort("COMPOSITIONTYPE", "rDelegationSRPort1").connectTo("App2", "rSRPort2", Optional.empty())

                // the provided and required port getter work also for
                // delegation connections
                // you can find more info in the java doc of IConnector
                scriptLogger.infoFormat("Created a connection between '{0}' and
                    '{1}'.",
                    createdConnector.getProvidedPort().getName(),
                    createdConnector.getRequiredPort().getName())
            }
        }
    }
}

```

Listing 4.244: Example how to create a simple delegation connection

```

import java.util.Optional

scriptTask ("delegationConnectionExample", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def portInterfaceMappingPath = "/ComponentTypes/
                    DataTypeMappingSets/PortInterfaceMappingSet1/Mapping1"

                // to add a port interface mapping to the connection, use an
                // optional with the AUTOSAR path to the port interface mapping
                def createdConnector = componentPort("COMPOSITIONTYPE", "pDelegationSRPort2").connectTo("App1", "pSRPort1", Optional.of(portInterfaceMappingPath))

                scriptLogger.infoFormat("Created a connection between '{0}' and
                    '{1}'.",
                    createdConnector.getProvidedPort().getName(),
                    createdConnector.getRequiredPort().getName())
            }
        }
    }
}

```

Listing 4.245: Create connector with port interface mapping

```

import com.vector.cfg.model.sysdesc.api.connection.IConnector
import com.vector.cfg.model.sysdesc.api.port.IComponentPort

scriptTask("UseSimpleAPIToCreateMultipleConnections", DV_PROJECT) {
    code {
        transaction {
            domain.runtimeSystem {
                // in this example we know for each port the target port (for
                // example stored in some external file)
                // and we know that the names of source and target ports do not
                // always match
                // so we use the simple API instead of the auto mapping
                List<String> sourcePortNames = ["App1.pDataSend",
                    "App1.pNvPort1",
                    "App1.pCSPort1"]
                List<String> targetPortNames = ["ECU Composition.
                    pDelegationSRPort2",
                    "App2.rNVPor1",
                    "App3.rOtherPort1CS"]

                // retrieve the component ports for the names
                List<IComponentPort> sourcePorts = selectComponentPorts {
                    componentPortNames(sourcePortNames)
                }.getComponentPorts()

                // since our used predicate does not guarantee any order, we
                // have to sort our ports to assure they are in correct order
                // because the connections below will be created matching index
                // of source and target ports
                sourcePorts.sort{a,b -> sourcePortNames.indexOf(a.getName())
                    <=> sourcePortNames.indexOf(b.getName())}

                // do the same for the target ports
                List<IComponentPort> targetPorts = selectComponentPorts {
                    componentPortNames(targetPortNames)
                }.getComponentPorts()

                targetPorts.sort{a,b -> targetPortNames.indexOf(a.getName())
                    <=> targetPortNames.indexOf(b.getName())}

                // we just want to make sure that the ports are connected
                // so we use assureConnectedTo(...) instead of connectTo(...)
                // in other words we do not care, which ports are already
                // connected
                List<IConnector> newConnectors = componentPorts(sourcePorts).
                    assureConnectedTo(targetPorts)

                // finally do some reporting for the port pairs that were
                // unconnected
                for (IConnector connector in newConnectors) {
                    scriptLogger.infoFormat("Connected {0} to {1}.",
                        connector.getProvidedPort().getName(),
                        connector.getRequiredPort().getName())
                }
            }
        }
    }
}

```

Listing 4.246: Connect ports using simple API

#### 4.10.4.10 Disconnect (unmap) Component Ports

The previous chapter was about connecting component ports. Now we want to have a look how to remove such connections again. We call it unmapping component ports.

This can be done using the component port selection API (see 4.10.4.1 on page 176) and calling a method to unmap the selected ports from other ports. It is possible to filter and evaluate the targets, so that you can have the control also for 1:n or n:m connected ports.

**Unmapping Component Ports** The use case of unmapping component ports is based on the selection of component ports. The targets are the ports which are connected to the selected port.

`unmap()` unmaps the selected component ports of this selection from ALL connected ports. In case that not all connections shall be removed the targets can be narrowed down using `unmapFrom(Closure)`.

##### Examples for `unmap()`

```
import com.vector.cfg.dom.runtimesys.groovy.api.IUnmappedPortsResult

scriptTask("unmapComponentPorts", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def unmappedPorts =
                    selectComponentPorts {
                        // select the component ports to be unmapped
                        connected()
                        componentPortName("App1.pDataSend")
                    } unmap()

                    // the simple 'unmap()' removes all connectors connecting the selected
                    // component ports to any other ports

                    // now print info of for the component ports that were unmapped from
                    // each other
                    // the data structure is a pair representing the source and the target
                    // port which were unmapped
                    for (final IUnmappedPortsResult unmappedPortPair : unmappedPorts) {
                        scriptLogger.infoFormat("Removed connector between {0} -> {1}.",
                                unmappedPortPair.getSourcePort().getName(),
                                unmappedPortPair.getTargetPort().getName())
                    }
                }
            }
        }
    }
}
```

Listing 4.247: Remove Connectors between Component Ports

##### Control unmapping in `unmapFrom(Closure)`

`selectTargetPorts(Closure)` allows to define predicates to narrow down the target ports which shall be disconnected from the in previous step selected ports.

`evaluateMatches(Closure)` allows to evaluate and change the results of the unmapped component ports.

For each selected component port the provided closure is called: Parameters are the current handled component port and a list of all connected target component ports (respecting the `selectTargetPorts(Closure)` predicates). The return value must be a list of component ports which are connected to the current handled source port.

```

import com.vector.cfg.dom.runtimesys.groovy.api.IUnmappedPortsResult
import com.vector.cfg.model.sysdesc.api.port.IComponentPort

scriptTask("unmapComponentPorts", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def unmappedPorts =
                    selectComponentPorts {
                        // select the component ports to be unmapped
                        connected()
                        componentPortName("App1OriginContextMatch.
                            CompletedOriginPortTest")
                    } unmapFrom {
                        // the connected (or target) ports can be filtered in this
                        closure
                        selectTargetPorts {
                            componentPortNames(["App1.ConnectedToCompletedOriginTest",
                                "App1_1.ConnectedToCompletedOriginTest"])
                        }
                    }
                    // additionally the target matches to be unmapped can be
                    evaluated
                    // the 'targetPorts' below are all ports connected to the '
                    // sourcePort' considering the 'selectTargetPorts' call above
                    evaluateMatches { IComponentPort sourcePort, List<
                        IComponentPort> targetPorts ->
                        // in our example we want to unmap the source ports only
                        // from ports of the component 'App1_1'
                        targetPorts.each {
                            if (it.getComponentName().equals("App1_1")) {
                                return [it]
                            }
                        }
                    }
                }
            }
            // print info for newly unmapped ports
            for (final IUnmappedPortsResult unmappedPortPair : unmappedPorts) {
                scriptLogger.infoFormat("Removed connector between {0} -> {1}.",
                    unmappedPortPair.getSourcePort().getName(),
                    unmappedPortPair.getTargetPort().getName())
            }
        }
    }
}

```

Listing 4.248: Remove Connectors between Component Ports Filtering Targets

#### 4.10.4.11 Terminating Component Ports

Port terminators can be used to acknowledge the fact, that the port is not connected yet. This will prevent validation rules to produce validation results reporting these ports as unconnected or missing data mappings for these ports. It also allows you to filter such ports very easily using

the according predicates of the selection APIs.

Starting point is the component port selection (see 4.10.4.1 on page 176).

`terminate()` terminates all selected `IComponentPorts`. If one of the selected ports is already terminated or connected to another component port, the port will be ignored.

The termination of component ports disables the validation of these ports. In other words, these ports are acknowledged as not connected yet. This should give a better overview of the open ports, which still have to be connected or need a data mapping.

`removePortTerminators()` removes the port terminators of all selected component ports. If one of the selected component ports is not terminated the method will ignore that port.

See `terminate()` for more information about the purpose of terminating ports.

It is also possible to create and remove port terminators via the simple API starting with the selection of one component port (`componentPort(String, String)`).

`terminate()` simple API to terminate the selected `IComponentPort` if it is not already terminated or connected to another component port. You can use `IComponentPort.isTerminated()` to check if a port is terminated and `IComponentPort.isConnected()` if a port is connected to other ports.

The termination of component ports disables the validation of these ports. In other words, these ports are acknowledged as not connected yet. This should give a better overview of the open ports, which still have to be connected or need a data mapping.

`removePortTerminator()` simple API to remove the port terminator of the selected component port.

See `terminate()` for more information about the purpose of terminating ports.

## Examples

```

import com.vector.cfg.model.sysdesc.api.port.IComponentPort

scriptTask("terminatePort", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                // the statements below will return us a collection with all component
                // ports which will be terminated
                def terminatedPorts =
                    selectComponentPorts {
                        delegation()
                        name("pDelegationCSPort1")

                        // use terminate() to terminate all selected component ports
                        // if a selected port is already terminated or connected, it will
                        // not be terminated (again)
                    } terminate()

                // this result may contain less ports than the actual selected ports,
                // if some of the selected ports were connected or terminated
                for (final IComponentPort terminatedPort : terminatedPorts) {
                    scriptLogger.infoFormat("Terminated component port '{0}'.",
                        terminatedPort.getName())
                }
            }
        }
    }
}

```

Listing 4.249: Terminate port using the component port selection API

```

import com.vector.cfg.model.sysdesc.api.port.IComponentPort

scriptTask("removePortTerminator", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                // the statements below will return us a collection with all component
                // ports for which a port terminator was removed
                def removedPortTerminators =
                    selectComponentPorts {
                        // filter for all terminated delegation ports
                        delegation()
                        // there are predicates to filter for terminated() and
                        // notTerminated() ports
                        terminated()

                        // if a port is not terminated it will be skipped
                    } removePortTerminators()

                // the result may contain less component ports than the selection,
                // if some of the ports were not terminated
                for (final IComponentPort componentPort : removedPortTerminators) {
                    scriptLogger.infoFormat("Removed port terminator of component port
                        '{0}'.", componentPort.getName())
                }
            }
        }
    }
}

```

Listing 4.250: Remove port terminator using the component port selection API

```

scriptTask("createPortTerminator", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                // select the component port via the simple API and call terminate()
                // this API is more strict and throws an exception if the selected port
                // cannot be terminated
                def terminatedPort = componentPort("COMPOSITIONTYPE", "rDelegationSRPort1").terminate()

                scriptLogger.infoFormat("Terminated component port '{0}'.",
                    terminatedPort.getName())
            }
        }
    }
}

```

Listing 4.251: Create a port terminator using the simple API

```

scriptTask("removePortTerminator", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                // select the component port via the simple API and call terminate()
                // this API is more strict and throws an exception if the selected port
                // has no port terminator to remove
                def removedTerminatorPort = componentPort("App1_1", "pSRPort1").removePortTerminator()

                scriptLogger.infoFormat("Terminated component port '{0}'.",
                    removedTerminatorPort.getName())
            }
        }
    }
}

```

Listing 4.252: Remove a port terminator using the simple API

**Terminating Component Ports of Communication Elements** As already mentioned above port terminators can be used to acknowledge the fact, that the port is not connected yet. It is possible to terminate owner ports of communication elements using the communication element selection.

`terminateOwnerPorts()` terminates the `IComponentPorts` of all selected `ICommunicationElements`. If one of the selected ports is already terminated or connected to another component port, the port will be ignored.

The termination of component ports disables the validation of these ports. In other words, these ports are acknowledged as not connected yet. This should give a better overview of the open ports, which still have to be connected or need a data mapping.

`removePortTerminatorsForOwnerPorts()` removes the port terminators of the `IComponentPorts` of all selected `ICommunicationElements`.

See `terminateOwnerPorts()` for more information about the purpose of terminating ports.

## Examples

```

import com.vector.cfg.model.sysdesc.api.port.IComponentPort

scriptTask("terminatePort", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                // the statements below will return us a collection with all
                // component ports which will be terminated
                def terminatedPorts =
                    selectCommunicationElements {
                        // also for the communication elements there are predicates to
                        // filter for terminated owner ports
                        ownerPortNotTerminated()
                        port("pSRPort1")

                        // this call will terminate the component port owner of each
                        // selected communication element
                    } terminateOwnerPorts()

                // if multiple communication elements has the same component port owner
                // the component port will be terminated and returned only once
                for (final IComponentPort terminatedPort : terminatedPorts) {
                    scriptLogger.infoFormat("Terminated component port '{0}'.",
                        terminatedPort.getName())
                }
            }
        }
    }
}

```

Listing 4.253: Terminate port using the communication element selection API

```

import com.vector.cfg.model.sysdesc.api.port.IComponentPort

scriptTask("removePortTerminator", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                // the statements below will return us a collection with all component
                // ports for which a port terminator was removed
                def removedPortTerminators =
                    selectCommunicationElements {
                        component("App1_1")
                        port("pSRPort1")

                        // if a port is not terminated it will be skipped
                    } removePortTerminatorsForOwnerPorts()

                for (final IComponentPort componentPort : removedPortTerminators) {
                    scriptLogger.infoFormat("Removed port terminator of component port
                        '{0}'.", componentPort.getName())
                }
            }
        }
    }
}

```

Listing 4.254: Remove port terminator using the communication element selection API

#### 4.10.4.12 Data Mapping

The data mapping use case allows to connect signal instances and data elements / operations / triggers. We will introduce two ways for that.

The first one will be using a selection API allowing to define predicates to filter communication elements/signals for the data mapping and calling a method to filter the target signals/communication elements. This is the way to map communication elements to system signals in a way like the data mapping assistant from the GUI. See 4.10.4.3 on page 186 and 4.10.4.2 on page 182 for the selection starting points.

The second way is to use a simple API that requires already prepared data structures e.g. a list of communication elements and a list of signal instances. To initially find the appropriate communication elements and signals you can use the common selection APIs.

**Mapping signal instances** The use case of auto-mapping signal instances is based on the selection of signal instances.

`autoMap()` tries to auto-map the selection of `IAbstractSignalInstances` (`ISignalInstance` or `ISignalGroupInstance`) according the data mapping assistant default rules. Therefore the selection of possible target communication elements is computed and tried to match to the selected signal instances.

##### Examples for `autoMap()`

```
scriptTask("autoDataMapAllUnmappedSignalInstances", DV_PROJECT) {
    code {
        transaction {
            domain.runtimeSystem {
                def dataMappings =
                    selectSignalInstances {
                        unmapped()
                    } autoMap()
                scriptLogger.infoFormat("Created {0} data mappings.",
                    dataMappings.size())
            }
        }
    }
}
```

Listing 4.255: Auto data map all unmapped signal instances

`autoMapTo(Closure)` tries to auto-map the selection of signal instances according the data mapping assistant rules but offers more control for the auto-mapping: Inside the closure additional predicates for narrowing down the target communication elements can be defined and code to evaluate and change the auto-mapper results can be provided.

`autoMapTo(Closure)` will produce trace, info and warning logs. To see them, activate the '`com.vector.cfg.dom.runtimesys.groovy.api.ISignalInstanceSelection`' logger with the appropriate log level.

##### Control the auto-mapping in `autoMapTo(Closure)`

`selectTargetCommunicationElements(Closure)` allows to define predicates to narrow down the target communication elements for the auto-mapping. The predicates are used to filter the possible target communication elements which were computed from the signal instance selection.

`evaluateMatches(Closure)` allows to evaluate and change the results of the auto-mapping. It corresponds to the confirm page of the data mapping assistant.

For each signal instance the provided closure is called: Parameters are the signal instance, the optional matched target communication element (or null), and a list of all potential target communication elements (respecting the `selectTargetCommunicationElements(Closure)` predicates). The return value must be a communication element or null.

```

import com.vector.cfg.model.sysdesc.api.com.IAbstractSignalInstance
import com.vector.cfg.model.sysdesc.api.com.ICommunicationElement

scriptTask("autoDataMapAllUnmappedSignalInstancesAndEvaluate", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def dataMappings =
                    selectSignalInstances {
                        unmapped()
                    } autoMapTo {
                        selectTargetCommunicationElements {
                            unmapped()
                        }
                    } evaluateMatches {
                        IAbstractSignalInstance signal,
                        ICommunicationElement optionalMatchedComElement,
                        List<ICommunicationElement> potentialComElements
                        ->
                        // evaluate
                        optionalMatchedComElement
                    }
                scriptLogger.infoFormat("Created {0} data mappings.", dataMappings.size())
            }
        }
    }
}

```

Listing 4.256: Auto data map all unmapped signal instances to unmapped communication elements and evaluate

**Nested Array of Primitives** `expandNestedArraysOfPrimitive(boolean)` allows to control the expansion of nested arrays of primitive globally. Per default, arrays are fully expanded (allowing to data map each array element). By setting the value to 'false', all nested arrays of primitive are not expanded and can be directly data-mapped to a signal.

```
import com.vector.cfg.model.sysdesc.api.com.IAbstractSignalInstance
import com.vector.cfg.model.sysdesc.api.com.ICommunicationElement

scriptTask("autoDatamapAllSignalInstancesAndDoNotExpandNestedArrayElements",
    DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def dataMappings =
                    selectSignalInstances {
                        } autoMapTo {
                            // do not expand nested array elements
                            expandNestedArraysOfPrimitive false
                            evaluateMatches {
                                IAbstractSignalInstance signal,
                                ICommunicationElement optionalMatchedComElement,
                                List<ICommunicationElement> potentialComElements ->
                                    // perform manual mapping to a signal group
                                    if (signal.getName().equals("elemB_c255f5e38fd8b21d")) {
                                        for (final ICommunicationElement comElement :
                                            potentialComElements) {
                                                if (comElement.getFullyQualifiedName().equals("App2.
                                                    rSRPort1.Element_2")) {
                                                        return comElement
                                                    }
                                                }
                                            }
                                        // now check: for the group signal the the record element
                                        // representing an array is not expanded
                                        if (signal.getName().equals("fieldA_f1d3783e235e88d3")) {
                                            // group signal
                                            for (final ICommunicationElement comElement :
                                                potentialComElements) {
                                                    if (comElement.getFullyQualifiedName().equals("App2.
                                                        rSRPort1.Element_2.RecordElement")) {
                                                            // do some direct mapping here
                                                        }
                                                    }
                                                }
                                            optionalMatchedComElement
                                        }
                                    }
                                scriptLogger.infoFormat("Created {0} data mappings.",dataMappings.size
                                    ())
                            }
                        }
                    }
    }
}
```

Listing 4.257: Auto data map all signal instances and do not expand nested array elements

`expandNestedArraysOfPrimitive(String,boolean)` allows to control the expansion of nested arrays of primitive for single nested arrays. Per default, the `expandNestedArraysOfPrimitive(boolean)` applies. For the given fully qualified communication element name, the global setting can be overridden.

```

import com.vector.cfg.model.sysdesc.api.com.IAbstractSignalInstance
import com.vector.cfg.model.sysdesc.api.com.ICommunicationElement

scriptTask("autoDatamapAllSignalInstancesAndDoExpandSpecificNestedArrayElement"
    , DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def dataMappings =
                    selectSignalInstances {
                        } autoMapTo {
                            // do not expand nested array elements
                            expandNestedArraysOfPrimitive false
                            expandNestedArraysOfPrimitive( "App2.rSRPort1.Element_2.
                                RecordElement",true)
                evaluateMatches {
                    IAbstractSignalInstance signal,
                    ICommunicationElement optionalMatchedComElement,
                    List<ICommunicationElement> potentialComElements ->
                    // perform manual mapping to a signal group
                    if (signal.getName().equals("elemB_c255f5e38fd8b21d")) {
                        for (final ICommunicationElement comElement :
                            potentialComElements) {
                            if (comElement.getFullyQualifiedName().equals("App2.
                                rSRPort1.Element_2")) {
                                return comElement
                            }
                        }
                    }
                    // now check: for the group signal the the record element
                    // representing an array is expanded:
                    // the single array elements can be mapped
                    if (signal.getName().equals("fieldA_f1d3783e235e88d3")) {
                        // group signal
                        for (final ICommunicationElement comElement :
                            potentialComElements) {
                            if (comElement.getFullyQualifiedName().equals("App2.
                                rSRPort1.Element_2.RecordElement[0]")) {
                                // do some direct mapping to array element here
                            }
                        }
                    }
                    optionalMatchedComElement
                }
            }
            scriptLogger.infoFormat("Created {0} data mappings.",dataMappings.size
                ())
        }
    }
}
}

```

Listing 4.258: Auto data map all signal instances and expand specific nested array element

`evaluateMatchesWithCompatibility(Closure)` allows to evaluate and change the results of the auto-mapping. It corresponds to the confirm page of the data mapping assistant. In contrast to `evaluateMatches(Closure)` this method also provides the compatibility of the optional match.

For each signal instance the provided closure is called: Parameters are the signal instance, the optional matched target communication element (or null), their compatibility and a list of

all potential target communication elements (respecting the `selectTargetCommunicationElements(Closure)` predicates). The return value must be a communication element or null.

```

import com.vector.cfg.model.sysdesc.api.com.ECompatibility
import com.vector.cfg.model.sysdesc.api.com.IAbstractSignalInstance
import com.vector.cfg.model.sysdesc.api.com.ICommunicationElement
import com.vector.cfg.model.sysdesc.api.com.IDataMapping

scriptTask("evaluateCommunicationElementsByCompatibility", DV_PROJECT) {
    code {
        transaction {
            domain.runtimeSystem {

                // in this example we want to accept full matches for data
                // mapping
                // and apply some custom rules for non-full matches

                List<IDataMapping> createdMappings = selectSignalInstances {
                    unmapped()
                } autoMapTo {
                    evaluateMatchesWithCompatibility {
                        IAbstractSignalInstance signal,
                        ICommunicationElement
                        optionalMatchedCommunicationElement,
                        ECompatibility compatibility,
                        List<ICommunicationElement> potentialComElements ->

                        if (compatibility == ECompatibility.FULL) {
                            return optionalMatchedCommunicationElement
                        }
                        // for non-full matches we return the first potential
                        // match if present
                        if (potentialComElements.size() > 0) {
                            return potentialComElements.get(0)
                        }
                        return null
                    }
                }
                scriptLogger.infoFormat("Created {0} data mappings.",
                    createdMappings.size())
            }
        }
    }
}

```

Listing 4.259: Evaluate matched communication elements using compatibility

`confirmByCompatibility(Closure)` allows to evaluate the mappings which should be created.

For each signal instance the provided closure is called: Parameters are the signal instance, the optional matched target communication element (or null) and the compatibility of them (`ECompatibility.NULL` if no optional match present) respecting all previous evaluations. So this is the final verifying step to confirm or reject the mapping. The return value must be true if the mapping should be created or false if you want to reject the mapping.

```

import com.vector.cfg.model.sysdesc.api.com.ECompatibility
import com.vector.cfg.model.sysdesc.api.com.IAbstractSignalInstance
import com.vector.cfg.model.sysdesc.api.com.ICommunicationElement
import com.vector.cfg.model.sysdesc.api.com.IDataMapping

scriptTask("confirmMappingToComElementByCompatibility", DV_PROJECT) {
    code {
        transaction {
            domain.runtimeSystem {

                // in this example we want only to accept data mappings with a
                // full match
                // and report all other

                List<IDataMapping> createdMappings = selectSignalInstances {
                    unmapped()
                } autoMapTo {
                    confirmByCompatibility {
                        IAbstractSignalInstance signal,
                        ICommunicationElement
                        optionalMatchedCommunicationElement,
                        ECompatibility compatibility ->

                        if (compatibility == ECompatibility.FULL) {
                            // accept full match
                            return true
                        }

                        if (optionalMatchedCommunicationElement != null) {
                            // report non-full matches
                            scriptLogger.infoFormat("Compatibility {0} between
                                signal {1} and communication element {2}.",
                                compatibility,
                                signal.getName(),
                                optionalMatchedCommunicationElement.
                                    getFullyQualifiedName())
                        } else {
                            // report signals for which the auto mapper could
                            // not find a match
                            scriptLogger.infoFormat("No match for signal {0}.",
                                signal.getName())
                        }
                        return false
                    }
                }
                scriptLogger.infoFormat("Created {0} data mappings.",
                    createdMappings.size())
            }
        }
    }
}

```

Listing 4.260: Decide which mappings should be created by compatibility

**Mapping communication elements** `autoMap()` tries to auto-map the selection of `ICommunicationElements` (`IDataCommunicationElement` or `IOperationCommunicationElement`) according the data mapping assistant default rules. Therefore the selection of possible target signal instances is computed and tried to match to the selected communication elements. You do not have to expand the communication elements in the previous selection step. They will

be expanded after the root is mapped automatically as you know from the data mapping assistant.

### Examples for autoMap()

```
scriptTask("autoDataMapAllUnmappedSRDelPortComElements", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def dataMappings =
                    selectCommunicationElements {
                        // select all unmapped sender/receiver delegation ports
                        delegation()
                        unmapped()
                        senderReceiver();
                    } autoMap()
                scriptLogger.infoFormat("Created {0} data mappings.", dataMappings.size
                    ())
            }
        }
    }
}
```

Listing 4.261: Auto data map all unmapped sender/receiver delegation port communication elements

**autoMapTo(Closure)** tries to auto-map the selection of communication elements according the data mapping assistant rules but offers more control for the auto-mapping: Inside the closure additional predicates for narrowing down the target signal instances can be defined and code to evaluate and change the auto-mapper results can be provided. You do not have to expand the communication elements in the previous selection step. They will be expanded after the root is mapped automatically as you know from the data mapping assistant.

**autoMapTo(Closure)** will produce trace, info and warning logs. To see them, activate the `'com.vector.cfg.dom.runtimesys.groovy.api.ICommunicationElementSelection'` logger with the appropriate log level.

### Control the auto-mapping in autoMapTo(Closure)

**selectTargetSignalInstances(Closure)** allows to define predicates to narrow down the target signal instances for the auto-mapping. The predicates are used to filter the possible target signal instances which were computed from the communication element selection.

**evaluateMatches(Closure)** allows to evaluate and change the results of the auto-mapping. It corresponds to the confirm page of the data mapping assistant.

For each communication element the provided closure is called: Parameters are the communication element, the optional matched target signal instance (or null), and a list of all potential target signal instances (respecting the `selectTargetSignalInstances(Closure)` predicates). The return value must be a signal instance or null.

```

import com.vector.cfg.model.sysdesc.api.com.IAbstractSignalInstance
import com.vector.cfg.model.sysdesc.api.com.ICommunicationElement

scriptTask("autoDatamapAllUnmappedComElementsAndEvaluate", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def dataMappings =
                    selectCommunicationElements {
                        unmapped() // only unmapped communication elements
                    } autoMapTo {
                        selectTargetSignalInstances {
                            // only map to unmapped rx signal instances
                            unmapped()
                            rx()
                        }
                    }

                    // we selected only the root communication elements, but for
                    // performing the data mapping the complex data elements are
                    // expanded
                    // this is a behavior which you can also notice in the data
                    // mapping assistant in the GUI
                    // that means the evaluateMatches method will also offer
                    // child communication elements and the corresponding matched
                    // group signals
                evaluateMatches {
                    ICommunicationElement communicationElement,
                    IAbstractSignalInstance optionalMatchedSignalInstance,
                    List<IAbstractSignalInstance> potentialSignalInstances ->
                        // evaluate the match here
                        if (optionalMatchedSignalInstance != null) {
                            def mdfSystemSignal =
                                optionalMatchedSignalInstance.getMdfObject()
                            ;
                            // check more specific ...
                        }
                        optionalMatchedSignalInstance
                }
            }
            scriptLogger.infoFormat("Created {0} data mappings.", dataMappings.size
                ())
        }
    }
}
}

```

Listing 4.262: Auto data map all unmapped communication elements to unmapped rx signal instances and evaluate

**Nested Array of Primitives** `expandNestedArraysOfPrimitive(boolean)` allows to control the expansion of nested arrays of primitive globally. Per default, arrays are fully expanded (allowing to data map each array element). By setting the value to 'false', all nested arrays of primitive are not expanded and can be directly data-mapped to a signal.

```

import com.vector.cfg.model.sysdesc.api.com.IAbstractSignalInstance
import com.vector.cfg.model.sysdesc.api.com.ISignalGroupInstance
import com.vector.cfg.model.sysdesc.api.com.ICommunicationElement

scriptTask("autoDatamapDoNotExpandNestedArrayElements", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def dataMappings =
                    selectCommunicationElements {
                } autoMapTo {
                    expandNestedArraysOfPrimitive false // do not expand nested arrays
                    of primitive
                    evaluateMatches {
                        ICommunicationElement communicationElement,
                        IAbstractSignalInstance optionalMatchedSignalInstance,
                        List<IAbstractSignalInstance> potentialSignalInstances ->
                        if ("App2.rSRPort1.Element_2".equals(communicationElement.
                            getFullyQualifiedName())) {
                            // manual matching: map to first signal group
                            for (IAbstractSignalInstance potentialSignal:
                                potentialSignalInstances) {
                                if (potentialSignal instanceof ISignalGroupInstance
                                    ) {
                                    return potentialSignal
                                }
                            }
                        }
                        if ("App2.rSRPort1.Element_2.RecordElement".equals(
                            communicationElement.getFullyQualifiedName())) {
                            // now the RecordElement which represents an array is
                            // directly offered to map
                            // ....
                        }
                    optionalMatchedSignalInstance
                }
            }
            scriptLogger.infoFormat("Created {0} data mappings.", dataMappings.size
                ())
        }
    }
}
}

```

Listing 4.263: Autodatamap and do not expand nested array elements

`expandNestedArraysOfPrimitive(String,boolean)` allows to control the expansion of nested arrays of primitive for single nested arrays. Per default, the `expandNestedArraysOfPrimitive(boolean)` applies. For the given fully qualified communication element name, the global setting can be overridden.

The fully qualified communication element name is e.g. determinable when using the data mapping assistant, performing an arbitrary signal group mapping of the root data element, and using the right-mouse menu its 'Copy fully qualified name' action on the nested array element.

```

import com.vector.cfg.model.sysdesc.api.com.IAbstractSignalInstance
import com.vector.cfg.model.sysdesc.api.com.ISignalGroupInstance
import com.vector.cfg.model.sysdesc.api.com.ICommunicationElement

scriptTask("autoDatamapDoExpandSpecificNestedArrayElement", DV_PROJECT) {
    code {
        transaction {
            domain.runtimeSystem {
                def dataMappings =
                    selectCommunicationElements {
                } autoMapTo {
                    // do not generally expand nested arrays of primitive
                    expandNestedArraysOfPrimitive false
                    // but expand the following specific record element
                    expandNestedArraysOfPrimitive("App2.rSRPort1.Element_2.
                        RecordElement",true)
                evaluateMatches {
                    ICommunicationElement communicationElement,
                    IAbstractSignalInstance optionalMatchedSignalInstance,
                    List<IAbstractSignalInstance> potentialSignalInstances ->
                    if ("App2.rSRPort1.Element_2".equals(
                        communicationElement.getFullyQualifiedName())) {
                        // manual matching: map to first signal group
                        for (IAbstractSignalInstance potentialSignal:
                            potentialSignalInstances) {
                            if (potentialSignal instanceof
                                ISignalGroupInstance) {
                                return potentialSignal
                            }
                        }
                    }
                    if ("App2.rSRPort1.Element_2.RecordElement[0]".equals(
                        communicationElement.getFullyQualifiedName())) {
                        // the RecordElement (representing an array of
                        primitive) is expanded to map the single array
                        elements
                        // ....
                    }
                    optionalMatchedSignalInstance
                }
            }
            scriptLogger.infoFormat("Created {0} data mappings.",dataMappings.size
                ())
        }
    }
}
}

```

Listing 4.264: Autodatamap and do expand a specific nested array element

`evaluateMatchesWithCompatibility(Closure)` allows to evaluate and change the results of the auto-mapping. In contrast to `evaluateMatches(Closure)` this method also provides the compatibility of the optional match.

For each communication element the provided closure is called: Parameters are the communication element, the optional matched target signal instance (or null), the compatibility of them and a list of all potential target signal instances (respecting the `selectTargetSignalInstances(Closure)` predicates). The return value must be a signal instance or null.

```

import com.vector.cfg.model.sysdesc.api.com.ECompatibility
import com.vector.cfg.model.sysdesc.api.com.IAbstractSignalInstance
import com.vector.cfg.model.sysdesc.api.com.ICommunicationElement
import com.vector.cfg.model.sysdesc.api.com.IDataMapping

scriptTask("evaluateSignalsByCompatibility", DV_PROJECT) {
    code {
        transaction {
            domain.runtimeSystem {

                // in this example we want to accept full matches for data
                // mapping
                // and apply some custom rules for non-full matches

                List<IDataMapping> createdMappings =
                    selectCommunicationElements {
                        unmapped()
                        senderReceiver()
                    } autoMapTo {
                        evaluateMatchesWithCompatibility {
                            ICommunicationElement communicationElement,
                            IAbstractSignalInstance optionalMatchedSignal,
                            ECompatibility compatibility,
                            List<IAbstractSignalInstance> potentialSignals ->

                            if (compatibility == ECompatibility.FULL) {
                                return optionalMatchedSignal
                            }
                            // for non-full matches we return the first potential
                            // match if present
                            if (potentialSignals.size() > 0) {
                                return potentialSignals.get(0)
                            }
                            return null
                        }
                    }
                }

                scriptLogger.infoFormat("Created {0} data mappings.",
                    createdMappings.size())
            }
        }
    }
}

```

Listing 4.265: Evaluate matched signal instances using compatibility

`confirmByCompatibility(Closure)` allows to evaluate the mappings which should be created.

For each communication element the provided closure is called: Parameters are the communication element, the optional matched target signal instance (or null) and the compatibility of them (`ECompatibility.NULL` if no optional match present) respecting all previous evaluations. So this is the final verifying step to confirm or reject the mapping. The return value must be true if the mapping should be created or false if you want to reject the mapping.

```

import com.vector.cfg.model.sysdesc.api.com.ECompatibility
import com.vector.cfg.model.sysdesc.api.com.IAbstractSignalInstance
import com.vector.cfg.model.sysdesc.api.com.ICommunicationElement
import com.vector.cfg.model.sysdesc.api.com.IDataMapping

scriptTask("confirmMappingToSignalByCompatibility", DV_PROJECT) {
    code {
        transaction {
            domain.runtimeSystem {

                // in this example we want only to accept data mappings with a
                // full match
                // and report all other


                List<IDataMapping> createdMappings =
                    selectCommunicationElements {
                        unmapped()
                        senderReceiver()
                    } autoMapTo {
                        confirmByCompatibility {
                            ICommunicationElement communicationElement,
                            IAbstractSignalInstance optionalMatchedSignal,
                            ECompatibility compatibility ->

                                if (compatibility == ECompatibility.FULL) {
                                    return true
                                }
                                if (optionalMatchedSignal != null) {
                                    // report non-full matches
                                    scriptLogger.infoFormat("Compatibility {0} between
                                        signal {1} and communication element {2}.",
                                        compatibility,
                                        optionalMatchedSignal.getName(),
                                        communicationElement.getFullyQualifiedName())
                                } else {
                                    // report signals for which the auto mapper could
                                    // not find a match
                                    scriptLogger.infoFormat("No match for communication
                                        element {0}.",
                                        communicationElement.getName())
                                }
                                return false
                            }
                        }
                    }

                    scriptLogger.infoFormat("Created {0} data mappings.",
                        createdMappings.size())
                }
            }
        }
    }
}

```

Listing 4.266: Decide which mappings should be created by compatibility

**Compatibility Between Communication Elements and Signal Instances** `ECompatibility` represents the compatibility between an `IAbstractSignalInstance` and an `ICommunicationElement` for a potential or existing `IDataMapping`. This enum helps to determine how good the elements are matching in terms of names and types and is available for example at the data mapping assistant or the data mapping automation API.

**FULL** For a **FULL** match the **IAbstractSignalInstance** and the **ICommunicationElement** have to be compatible regarding their types (see **TYPE\_INCOMPATIBLE**) and the name of the signal has to match either the owner port name of the communication element, the communication element name itself or a combination of the two. All names are normalized (e.g. removing some common signal group and group signal suffixes and convert capital to small letters) before performing the match.

**FULL** matches will be mapped automatically by the auto-mapper when not using further evaluation.

**Examples:**

Fully qualified **ICommunicationElement** names on the left mapped to fully qualified **IAbstractSignalInstance** names on the right.

- 'MyPort.MyData' -> 'MyData': full match, signal and data element names are equal.
- 'MyData.DataElement' -> 'MyData': full match, signal and port names are equal.
- 'MyData\_Record' -> 'MyData\_SignalGroup': full match, the suffix is recognized as marker of signal group and marker of record without further meaning.
- 'ParkingBrake\_Status' -> 'ParkingBrake\_Position': no match, the suffixes are no obvious markers which can be ignored, compatibility **NONE**.
- 'MyPort.MyStructure.MyData1' -> 'MyStructure.MyData1': full match, group signal and record element have equal names.
- 'ParkingBrake.ParkingBrakeStatus' -> 'MyStatus1': no full match, compatibility **NONE**.
- 'ParkingBrake.ParkingBrakeStatus' -> 'BrakeStatus': no full match, but a **PARTIAL\_NAME** match because signal name is contained in data element name.

**PARTIAL\_NAME** For a **PARTIAL\_NAME** match the **IAbstractSignalInstance** and the **ICommunicationElement** have to be compatible regarding their types (see **TYPE\_INCOMPATIBLE**). Additionally the name of the signal should be contained in the owner port name of the communication element or in the communication element name itself, but also vice versa, that means if the owner port name or the communication element name is contained in the signal name. All names are normalized (e.g. removing some common signal group and group signal suffixes and convert capital to small letters) before performing the match.

**PARTIAL\_NAME** matches will be mapped automatically by the auto-mapper when not using further evaluation.

**Examples:**

Fully qualified **ICommunicationElement** names on the left mapped to fully qualified **IAbstractSignalInstance** names on the right.

- 'ParkingBrake.BrakeStatus' -> 'ParkingBrakeStatus': partial name match, data element name is part of signal name.
- 'ParkingBrake.ParkingBrakeStatus' -> 'BrakeStatus': partial name match, signal name is part of data element name.
- 'SendBrakeStatus.ParkingBrake' -> 'Status': partial name match, signal name is part of port name.
- 'SendStatus.ParkingBrake' -> 'ParkingBrake\_SendStatus': partial name match, port name is part of signal name.

**TYPE\_INCOMPATIBLE** An **IAbstractSignalInstance** is **TYPE\_INCOMPATIBLE** to an **ICommunicationElement** if they match **FULL** or by **PARTIAL\_NAME** and the values of dynamic length attribute of the signal and variable size of the communication element's data type do not match, but also if the signal is an **ISignalGroupInstance** and the communication element's data type uses variable size. Signals using data transformation are never **TYPE\_INCOMPATIBLE**.

**TYPE\_INCOMPATIBLE** matches will be mapped automatically by the auto-mapper when not using further evaluation.

Hint: At first it sounds strange that the auto-mapper accepts **TYPE\_INCOMPATIBLE** matches. The reason is that the auto-mapper is strongly based on name matching and since the names match (as it is the case here), the auto-mapper will already do the mapping, but you have probably to correct some attributes at the signal or the data type which do not match some expectations of our validations.

**STRUCTURE\_INCOMPATIBLE** Compatibility **STRUCTURE\_INCOMPATIBLE** is only used for **ISignalGroupInstances**. It is used if the compatibility between the signal group and the root communication element is either **FULL** or **PARTIAL\_NAME** and at the same time there is at least one communication element leaf for which neither a group signal that matches **FULL** nor by **PARTIAL\_NAME** exist below the signal group.

#### Examples:

A signal group named 'MyComplexData' has the group signals 'SubData1' and 'OtherGroupSignal'. The communication element of port 'MyPort' to be matched is named 'MyComplexData' and is representing a record with the record elements 'SubData1' and 'OtherRecordElement'. So that the roots will match by names and types, but no match for the record element 'OtherRecordElement' does exist at the signal group, since group signal 'OtherGroupSignal' is not matching by name.

The auto-mapper will produce the following result for that:

```
MyPort.MyComplexData -> MyComplexData
MyPort.MyComplexData.SubData1 -> MyComplexData.SubData1
MyPort.MyComplexData.OtherRecordElement -> unmapped
```

**STRUCTURE\_INCOMPATIBLE** matches will NOT be mapped automatically by the auto-mapper.

**NONE** Compatibility **NONE** is used in the following cases:

1. **IAbstractSignalInstance** and **ICommunicationElement** names are neither matching **FULL**, nor by **PARTIAL\_NAME**.
2. EDirections of **IAbstractSignalInstance** and **ICommunicationElement** are incompatible.
3. If the **IAbstractSignalInstance** does not use data transformation but the communication element is an **IOperationCommunicationElement**.
4. The length of the signal is not 0 but the communication element is an **ITriggerCommunicationElement**.
5. The **IAbstractSignalInstance** is not a signal group and does not use data transformation but the communication element represents a record, a union or an array of non-primitives.
6. If the **IAbstractSignalInstance** is a group signal but the communication element is not a leaf of a complex **IDataCommunicationElement**.
7. **IAbstractSignalInstance** is a signal group but the communication element is neither representing a record nor an array.

**NONE** matches will NOT be mapped automatically by the auto-mapper.

**NULL** The compatibility **NULL** is only used in case the auto-mapper did not find any matching communication elements.

**Simple API for data mapping** This API can be used when the names of the communication elements, which should be data mapped, are known, as well as the AUTOSAR paths to the system signal groups and system signals or if you want to use custom rules which you apply to sort the elements and want to map a list of communication elements and a list of signals via index which helps you to increase the level of control. Possible entry points are the selection of a communication element and potentially also its child communication elements by using the fully qualified names or using the communication and signal selections introduced above to select and later sort a list of communication elements and a list of signals. For complex data mappings there is a comfort function. If you define the root mapping only (e.g. mapping a record to a system signal group) the auto-mapper will try to match the children (e.g. record elements and group signals) by naming. This comfort function does not expand primitive arrays.

`communicationElement(String...)` allows to select an `ICommunicationElement` and to do further operations on it, e.g. performing a data mapping. For complex communication elements also the children can be selected. In such case the first `communicationElementName` has to be the name of the root element.

`ISelectedCommunicationElement` represents a selection of exactly one `ICommunicationElement` and provides further actions on it. For complex communication elements, also a subset of the child elements is represented.

`mapTo(String...)` creates an `IDataMapping` for the `ICommunicationElement` which is represented by this `ISelectedCommunicationElement`. The mapped signal is the given `abstractSignalInstance`.

In case of a complex mapping first the system signal group has to be referenced. The child mappings are created considering the given order or in other words, the first communication element is mapped to the first signal, the second element to the second signal and so on.

For a client server to signal mapping, select the operation communication element and enter the paths to two serialized signals. Which signal is the call and which one the return signal is determined by the direction of the signals.

Hint: Use 'ECU Composition' or 'COMPOSITIONTYPE' as component name to select communication elements of delegation ports.

There are a few checks also for the simple API.

- Before creating the data mapping check if an equal mapping already exists. Do not create redundant mappings.
- Check that the amount of the selected communication elements is equal to the amount of the selected signals. For the client server use case there will be one communication element for the call direction and one for the return direction.
- Checks that the direction of the signal and communication element are compatible. Checks also the type compatibility of signal (group) and communication element.  
An operation communication element can only be mapped to a serialized signal.

Records can only be mapped to serialized signals or signal groups.

Unions can only be mapped to serialized signals.

Arrays with complex array element can only be mapped to serialized signals or signal groups.

Trigger communication elements cannot be mapped to signal groups.

- For the client server data mappings, check that exactly two signals are selected and that both signals are serialized signals.
- Checks for complex mappings, that the first abstract signal instance is a signal group and the other instances are group signals of this signal group.
- Check that the selected communication elements are suitable for data mapping. That means they should belong to a sender receiver, client server or a trigger port, which is not a service port and not a PRPort.
- Check the hierarchy of the selected communication elements for complex mappings. First selected element should be the root element and all further selected elements children of the root element.
- Check for the client server use case, that both communication elements, one for the call and one for the return direction can be found. The communication elements have to be operation communication elements.
- Check for a complex mapping that the first selected communication element is really a complex root element.
- Check if the owner port of the communication element is terminated. Terminated ports should not be data mapped. Please remove the port terminator first.

If you want to use some internal rules other than name matching to map communication elements to system signals you can apply this rules to sort a list of communication elements and a list of abstract signal instances and then use the simple API below that maps them via index.

`communicationElements(List)` wraps `ICommunicationElements` to do further operations on them, e.g. map them to system signals.

`mapTo(List)` maps the `ICommunicationElements` which are represented by this `ISelectedCommunicationElements` to the given `IAbstractSignalInstances` via index matching. So the first communication element will be mapped to the first abstract signal instance, the second to the second abstract signal instance and so on.

In case you want to ignore communication element and system signal pairs for which a mapping does already exist please use `assureMappedTo(List)`.

If you want to map your communication elements to system signals using name matching please use `IRuntimeSystemApi.selectCommunicationElements(groovy.lang.Closure)`.

If you want to create complex mappings (currently only `SenderReceiverToSignalGroupMappings`) the given `abstractSignalInstances` should contain the group signals right after the parent signal group instances.

For a client server to signal mapping, select call and return signal instance directly after each other.

`assureMappedTo(List)` does the same as `mapTo(List)` but ignores communication element and abstract signal instance pairs for which a data mapping does already exist.

If you want to create complex mappings (currently only `SenderReceiverToSignalGroupMappings`) the given `abstractSignalInstances` should contain the group signals right after the parent signal group instances.

For a client server to signal mapping, select call and return signal instance directly after each other.

## Examples

```
scriptTask ("createSimpleMapping", DV_PROJECT ){  
    code {  
        transaction {  
            domain.runtimeSystem {  
                // specify path to the system signal  
                def signalPath = "/VectorAutosarExplorerGeneratedObjects/  
                    SYSTEM_SIGNALS/Element_1_b16df82332bcf915"  
  
                // enter the fully qualified communication element name  
                // that means ComponentName.PortName.DataElementName  
                def communicationElementName = "App1.pDataSend.Element"  
  
                def createdMapping = communicationElement(  
                    communicationElementName).mapTo(signalPath)  
  
                scriptLogger.infoFormat("Mapped '{0}' to '{1}'.",  
                    createdMapping.getMappedCommunicationElement().  
                        getFullyQualifiedName(),  
                    createdMapping.getMappedSystemSignal().getAutosarPath())  
            }  
        }  
    }  
}
```

Listing 4.267: Create sender receiver to signal mapping

```

scriptTask ("mapDelegationPort", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def signalPath = "/VectorAutosarExplorerGeneratedObjects/
                    SYSTEM_SIGNALS/Element_1_b16df82332bcf915"

                // for delegation ports use 'ECU Composition' instead of the
                // component name
                def communicationElementName = "ECU Composition.
                    pDelegationSRPort2.Element"

                def createdMapping = communicationElement(
                    communicationElementName).mapTo(signalPath)

                scriptLogger.infoFormat("Mapped '{0}' to '{1}'.",
                    createdMapping.getMappedCommunicationElement().
                        getFullyQualifiedName(),
                    createdMapping.getMappedSystemSignal().getAutosarPath())
            }
        }
    }
}

```

Listing 4.268: Create data mapping for delegation port

```

scriptTask ("createClientServerMapping", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def callSignalPath = "/VectorAutosarExplorerGeneratedObjects/
                    SYSTEM_SIGNALS/rSRPort2_d4aecc362f1feef3"
                def returnSignalPath = "/VectorAutosarExplorerGeneratedObjects/
                    SYSTEM_SIGNALS/pSRPort3_2264a06bc04fc81d"

                // if an operation communication element is selected, a client
                // server to signal mapping will be created
                // the assignment of call and return signal role is depending
                // on the direction of the signal
                def communicationElementName = "ECU Composition.
                    pDelegationCSPort1.Operation"

                def createdMapping = communicationElement(
                    communicationElementName).mapTo(callSignalPath,
                    returnSignalPath)

                scriptLogger.infoFormat("Mapped '{0}' to '{1}' as call signal
                    and '{2}' as return signal.",
                    createdMapping.getMappedCommunicationElement().
                        getFullyQualifiedName(),
                    createdMapping.getMappedSystemSignal().getAutosarPath(),
                    createdMapping.getMappedReturnSignal().getAutosarPath())
            }
        }
    }
}

```

Listing 4.269: Create client server to signal mapping

```

import com.vector.cfg.model.sysdesc.api.com.ISenderReceiverDataMapping

scriptTask ("mapRecordToSignalGroup", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                // specify the path to the signal group
                def signalGroup = "/VectorAutosarExplorerGeneratedObjects/
                    SYSTEM_SIGNAL_GROUPS/Element_2_de8db6949370c6b4"

                // specify the paths to the group signals of the signal group
                def groupSignal1 = "/VectorAutosarExplorerGeneratedObjects/
                    SYSTEM_SIGNALS/ay1_48523fe229ba8c99"
                def groupSignal2 = "/VectorAutosarExplorerGeneratedObjects/
                    SYSTEM_SIGNALS/ay2_071a3305d39fcca4"
                def groupSignal3 = "/VectorAutosarExplorerGeneratedObjects/
                    SYSTEM_SIGNALS/ay3_84eba37e401eacd1"

                // the name of the root element
                def record = "ECU Composition.pDelegationSRPort1.Element_2"

                // the names of the child elements
                def recordElement1 = "ECU Composition.pDelegationSRPort1.
                    Element_2.RecordElement"
                def recordElement2 = "ECU Composition.pDelegationSRPort1.
                    Element_2.RecordElement_1"
                def recordElement3 = "ECU Composition.pDelegationSRPort1.
                    Element_2.RecordElement_2"

                // create the mapping, first argument should be the root
                // element, followed by the leaf elements for the child
                // mappings
                // for the signals, first argument should be the signal group,
                // followed by the group signals

                // the mapping will be done using the given order
                // e.g. the first element (record) will be mapped to the signal
                // group (signalGroup),
                // the last record element (recordElement3) will be mapped to
                // the last group signal (groupSignal3)
                def createdMapping = communicationElement(record,
                    recordElement1, recordElement2, recordElement3)
                    .mapTo(signalGroup, groupSignal1, groupSignal2,
                        groupSignal3)

                scriptLogger.infoFormat("Mapped '{0}' to '{1}'.",
                    createdMapping.getMappedCommunicationElement().
                        getFullyQualifiedName(),
                    createdMapping.getMappedSystemSignal().getAutosarPath())

                // print info for the child mappings
                for (final ISenderReceiverDataMapping childMapping :
                    createdMapping.getChildDataMappings()) {
                    scriptLogger.infoFormat("Mapped '{0}' to '{1}'.",
                        childMapping.getMappedCommunicationElement().
                            getFullyQualifiedName(),
                        childMapping.getMappedSystemSignal().getAutosarPath())
                }
            }
        }
    }
}

```

Listing 4.270: Map record to signal group

```

import com.vector.cfg.model.sysdesc.api.com.ISenderReceiverDataMapping

scriptTask ("mapArrayToSignalGroup", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                // specify the path to the signal group
                def signalGroup = "/VectorAutosarExplorerGeneratedObjects/
                    SYSTEM_SIGNAL_GROUPS/Element_2_de8db6949370c6b4"

                // specify the paths to the group signals of the signal group
                def groupSignal1 = "/VectorAutosarExplorerGeneratedObjects/
                    SYSTEM_SIGNALS/ay1_48523fe229ba8c99"
                def groupSignal2 = "/VectorAutosarExplorerGeneratedObjects/
                    SYSTEM_SIGNALS/ay2_071a3305d39fcca4"
                def groupSignal3 = "/VectorAutosarExplorerGeneratedObjects/
                    SYSTEM_SIGNALS/ay3_84eba37e401eacd1"

                // the name of the root element
                def array = "ECU Composition.pDelegationSRPort2.Element_1"

                // select the element of the array using the position index of
                // the element
                def arrayElement1 = "ECU Composition.pDelegationSRPort2.
                    Element_1[0]"
                def arrayElement2 = "ECU Composition.pDelegationSRPort2.
                    Element_1[1]"
                def arrayElement3 = "ECU Composition.pDelegationSRPort2.
                    Element_1[2]"

                def createdMapping = communicationElement(array, arrayElement1,
                    arrayElement2, arrayElement3)
                    .mapTo(signalGroup, groupSignal1, groupSignal2,
                        groupSignal3)

                scriptLogger.infoFormat("Mapped '{0}' to '{1}'.",
                    createdMapping.getMappedCommunicationElement().
                        getFullyQualifiedName(),
                    createdMapping.getMappedSystemSignal().getAutosarPath())

                // print info for the child mappings
                for (final ISenderReceiverDataMapping childMapping :
                    createdMapping.getChildDataMappings()) {
                    scriptLogger.infoFormat("Mapped '{0}' to '{1}'.",
                        childMapping.getMappedCommunicationElement().
                            getFullyQualifiedName(),
                        childMapping.getMappedSystemSignal().getAutosarPath())
                }
            }
        }
    }
}

```

Listing 4.271: Map array to signal group

```
import com.vector.cfg.model.sysdesc.api.com.ISenderReceiverDataMapping

scriptTask("completeComplexDataMapping", DV_PROJECT) {
    code {
        transaction {
            domain.runtimeSystem {

                // in this example we want to map a record to a system signal
                // group
                // we will map only the roots
                // a comfort function will try to find matches in record
                // elements and group signals via naming

                String recordName = "ECU Composition.pDelegationSRPort1.
                    Element_2"
                String signalGroupPath = "/"
                    VectorAutosarExplorerGeneratedObjects/SYSTEM_SIGNAL_GROUPS/
                    Element_2_de8db6949370c6b4"

                ISenderReceiverDataMapping createdDataMapping =
                    communicationElement(recordName).mapTo(signalGroupPath)

                scriptLogger.infoFormat("Mapped {0} -> {1}.",
                    createdDataMapping.getMappedCommunicationElement().
                        getFullyQualifiedName(),
                    createdDataMapping.getMappedSystemSignal().getName())

                // we want also to print info for the child mappings
                for (ISenderReceiverDataMapping childMapping in
                    createdDataMapping.getLeafDataMappings()) {
                    scriptLogger.infoFormat("Mapped {0} -> {1}.",
                        childMapping.getMappedCommunicationElement().
                            getFullyQualifiedName(),
                        childMapping.getMappedSystemSignal().getName())
                }
            }
        }
    }
}
```

Listing 4.272: Map complex data element to system signal group and let the auto-mapper complete the mapping if possible

```

import com.vector.cfg.model.sysdesc.api.com.ICommunicationElement
import com.vector.cfg.model.sysdesc.api.com.IDataMapping
import com.vector.cfg.model.sysdesc.api.com.IClientServerDataMapping
import com.vector.cfg.model.sysdesc.api.com.IAbstractSignalInstance

scriptTask("UseSimpleAPIToCreateMultipleDataMappings", DV_PROJECT) {
    code {
        transaction {
            domain.runtimeSystem {

                // in this example we know for each communication element the
                // name of the system signal to map it to
                // (for example stored in some external file)
                // so we use the simple API instead of the auto mapping

                List<String> comElementNames =
                    ["ECU Composition.TriggerDataMappingZeroLengthSignal.
                        SomeTrigger",
                     // com element below for client server to signal mapping
                     "App1_1.pCSPort1.Operation",
                     "App2.rSRPort1.Element",
                     // com elements below are a record and its record elements
                     "App2.rSRPort1.Element_2",
                     "App2.rSRPort1.Element_2.RecordElement",
                     "App2.rSRPort1.Element_2.RecordElement_2"]
                List<String> signalNames =
                    ["TriggerDataMappingZeroLengthSignal_896bde67e5a0f5b4",
                     // the two signals below are used for a client server to
                     // signal mapping
                     // one call and one return signal
                     "rSRPort2_d4aecc362f1feef3", "pSRPort3_2264a06bc04fc81d",
                     "RElement_1_c07c9ba68bc545ba",
                     // com elements below is a signal group and its group
                     // signals
                     "elemB_c255f5e38fd8b21d",
                     "fieldA_f1d3783e235e88d3",
                     "fieldB_344fdc16e87cfdaa"]

                // we use the communication element selection to retrieve the
                // communication elements
                // for the client server to signal mapping the selection will
                // find two com elements for the one name
                // one for the call direction and one for the return direction
                List<ICommunicationElement> comElements =
                    selectCommunicationElements {
                        fullyQualifiedNames(comElementNames)
                        // we need to select also unmapped record elements, so use
                        // fully expanded selection
                        // another way would be to select only the root
                        // and then access the leafs e.g. via
                        IDataCommunicationElement.
                        getLeafsFullExpandedExceptPrimitiveArrays()
                        selectFullyExpanded()
                    }.getCommunicationElements()

                // since our used predicate does not guarantee any order, we
                // have to sort our communication elements to assure they are
                // in correct order
                comElements.sort{a,b -> comElementNames.indexOf(a.
                    getFullyQualifiedName()) <= comElementNames.indexOf(b.
                    getFullyQualifiedName())}
            }
        }
    }
}

```

Listing 4.273: Map communication elements to system signals using simple API Part1

```
// now select and sort the signal instances
List<IAbstractSignalInstance> signals = selectSignalInstances {
    names(signalNames)
}.getSignalInstances()

signals.sort{a,b -> signalNames.indexOf(a.getName()) <=>
    signalNames.indexOf(b.getName())}

// map communication elements via index to the system signals
// for complex mappings the children need to be inserted directly
// after the parent

// for client server mappings call and return elements right
// after each other
// call first or return first does not matter

// mapTo(...) would fail if one of the mappings does already
// exist
// but in this example we just want to make sure that the mapping
// exist, if not assureMappedTo(...) will create it, if it does
// exist it will do nothing
List<IDataMapping> dataMappings = communicationElements(
    comElements).assureMappedTo(signals)

// finally do some reporting for the pairs that were previously
// not mapped to each other
for (IDataMapping dataMapping in dataMappings) {
    scriptLogger.infoFormat("Mapped {0} to {1}.",
        dataMapping.getMappedCommunicationElement(),
        getFullyQualifiedName(),
        dataMapping.getMappedSystemSignal().getName())

    if (dataMapping instanceof IClientServerDataMapping) {
        scriptLogger.infoFormat("Return signal is {0}.",
            ((IClientServerDataMapping) dataMapping).
            getMappedReturnSignal().getName())
    }
}
}
```

Listing 4.274: Map communication elements to system signals using simple API Part2

#### 4.10.4.13 Remove Data Mappings

The previous chapter was about mapping communication elements and signal instances. Now we want to have a look how to remove such mappings again.

This can be done using the communication element selection API (see 4.10.4.3 on page 186) or the signal instance selection API (see 4.10.4.2 on page 182) and calling a method to unmap the selected communication elements / signal instances.

**Unmapping by Selecting Communication Elements** The use case of unmapping communication elements is based on the selection of communication elements. The targets to be unmapped

are signal instances, which can be also narrowed down by further closures.

`unmap()` unmaps the selected communication elements from all mapped system signals. In case that not all data mappings of the selected communication elements shall be removed the mapped signal instances can be narrowed down using `unmapFrom(Closure)`.

For `SenderReceiverToSignalGroupMappings` (complex mappings) it is already enough to select one of the communication elements (for example the root element), all mappings belonging to the complex mapping will be removed. In other words removing the root mapping also removes the child mappings, removing a child mapping also removes the root mapping and the other child mappings of the same root.

#### Examples for `unmap()`

```

import com.vector.cfg.dom.runtimesys.groovy.api.
    IUnmappedComElementAndSignalResult

scriptTask("UnmapCommunicationElements", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {

                // we want to remove all data mappings of delegation ports in this
                // example
                def result = selectCommunicationElements {
                    mapped()
                    delegation()
                } unmap()

                // now print a detailed info for all removed data mappings
                // we want also detailed info for client server to signal mappings and
                // sender receiver to signal group mappings
                scriptLogger.infoFormat("Removed {0} (root) mappings.", result.size())

                for (IUnmappedComElementAndSignalResult unmappedResult : result) {
                    scriptLogger.infoFormat("{0} -> {1}",
                        unmappedResult.getCommunicationElement(),
                        getFullyQualifiedName(),
                        unmappedResult.getSignalInstance().getName());

                    // additional info for return signal mapping
                    if (unmappedResult.getReturnSignalCommunicationElement() != null) {
                        scriptLogger.infoFormat("{0} -> {1}",
                            unmappedResult.getReturnSignalCommunicationElement(),
                            getFullyQualifiedName(),
                            unmappedResult.getReturnSignalInstance().getName());
                    }

                    // additional info for mapped record and array elements
                    if (!unmappedResult.getChildCommunicationElements().isEmpty()) {

                        for (int i = 0; i < unmappedResult.
                            getChildCommunicationElements().size(); i++) {
                            scriptLogger.infoFormat("{0} -> {1}",
                                unmappedResult.getChildCommunicationElements().get(
                                    i).getFullyQualifiedName(),
                                unmappedResult.getGroupSignalInstances().get(i).
                                    getName());
                        }
                    }
                }
            }
        }
    }
}

```

Listing 4.275: Remove Data Mapping of Communication Element

### Control unmapping in unmapFrom(Closure)

`selectTargetSignalInstances(Closure)` allows to define predicates to narrow down the target signal instances to be unmapped from the previously selected communication elements.

`evaluateMatches(Closure)` allows to evaluate and change the results of the communication

elements which are about to be unmapped from signal instances.

For each selected communication element the provided closure is called: Parameters are the current handled communication element and a list of all mapped signal instances (respecting the `selectTargetSignalInstances(Closure)` predicates). The return value must be a list of signal instances which are mapped to the current handled communication element.

### Examples for `unmapFrom(Closure)`

```
scriptTask("UnmapCommunicationElementsAdvanced", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {

                // we want to remove only data mappings for delegation ports to signals
                // of a special frame
                def result = selectCommunicationElements {
                    mapped()
                    delegation()
                } unmapFrom {
                    // we have already filtered the communication elements
                    // in this closure we can now additionally filter also for
                    // the signals of the data mapping to be removed
                    selectTargetSignalInstances {
                        frame("MyFrame")
                    }
                }

                // see the simple example above how to print more detailed info
                scriptLogger.infoFormat("Removed {0} (root) mappings.", result.size())
            }
        }
    }
}
```

Listing 4.276: Remove Data Mapping of Communication Element Considering Signals

**Unmapping by Selecting Signal Instances** The use case of unmapping system signals is based on the selection of signal instances. The targets to be unmapped are communication elements, which can be also narrowed down by further closures.

`unmap()` unmaps the selected signal instances from all mapped communication elements. In case that not all data mappings of the selected signal instances shall be removed the mapped communication elements can be narrowed down using `unmapFrom(Closure)`.

For `SenderReceiverToSignalGroupMappings` (complex mappings) it is already enough to select the signal group or one of the group signals, all mappings belonging to the complex mapping will be removed. In other words removing the root mapping also removes the child mappings, removing a child mapping also removes the root mapping and the other child mappings of the same root.

### Examples for `unmap()`

```

import com.vector.cfg.dom.runtimesys.groovy.api.
    IUnmappedComElementAndSignalResult

scriptTask("UnmapSignalInstances", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {

                // we want to remove all data mappings of all mapped tx signals in this
                // example
                def result = selectSignalInstances {
                    mapped()
                    tx()
                } unmap()

                // now print a detailed info for all removed data mappings
                // we want also detailed info for client server to signal mappings and
                // sender receiver to signal group mappings
                scriptLogger.infoFormat("Removed {0} (root) mappings.", result.size())

                for (IUnmappedComElementAndSignalResult unmappedResult : result) {
                    scriptLogger.infoFormat("{0} -> {1}",
                        unmappedResult.getCommunicationElement(),
                        getFullyQualifiedName(),
                        unmappedResult.getSignalInstance().getName());

                    // additional info for return signal mapping
                    if (unmappedResult.getReturnSignalCommunicationElement() != null) {
                        scriptLogger.infoFormat("{0} -> {1}",
                            unmappedResult.getReturnSignalCommunicationElement(),
                            getFullyQualifiedName(),
                            unmappedResult.getReturnSignalInstance().getName());
                    }

                    // additional info for mapped record and array elements
                    if (!unmappedResult.getChildCommunicationElements().isEmpty()) {

                        for (int i = 0; i < unmappedResult.
                            getChildCommunicationElements().size(); i++) {
                            scriptLogger.infoFormat("{0} -> {1}",
                                unmappedResult.getChildCommunicationElements().get(
                                    i).getFullyQualifiedName(),
                                unmappedResult.getGroupSignalInstances().get(i).
                                    getName());
                        }
                    }
                }
            }
        }
    }
}

```

Listing 4.277: Remove Data Mapping of Signal

### Control unmapping in unmapFrom(Closure)

`selectTargetCommunicationElements(Closure)` allows to define predicates to narrow down the target communication elements to be unmapped from the previously selected signal instances.

`evaluateMatches(Closure)` allows to evaluate and change the results of the signal instances

which are about to be unmapped from communication elements.

For each selected signal instance the provided closure is called: Parameters are the current handled signal instance and a list of all mapped communication elements (respecting the `selectTargetCommunicationElements(Closure)` predicates). The return value must be a list of communication elements which are mapped to the current handled signal instance.

#### Examples for `unmapFrom(Closure)`

```
scriptTask("UnmapSignalInstancesAdvanced", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {

                // we want to remove only data mappings for transformed signals which
                // are mapped to communication elements of component 'App1'
                def result = selectSignalInstances {
                    mapped()
                    transformed()
                } unmapFrom {
                    // we have already filtered the signal instances
                    // in this closure we can now additionally filter also for
                    // communication elements of the data mapping to be removed
                    selectTargetCommunicationElements {
                        component("App1")
                    }
                }

                // see the simple example above how to print more detailed info
                scriptLogger.infoFormat("Removed {0} (root) mappings.", result.size())
            }
        }
    }
}
```

Listing 4.278: Remove Data Mapping of Signals Considering Communication Elements

#### 4.10.4.14 Create Component Prototypes

In the create component prototypes use case, components can be instantiated after a component type was selected. So the entry point is the component type selection (see 4.10.4.4 on page 190).

**Instantiate Components** `createPrototype()` creates a `SwComponentPrototype` in the `FltExtract` for each selected component type. The names of the created `SwComponentPrototypes` are derived from the selected component types.

#### Examples for `createPrototype()`

```

scriptTask ("createComponentPrototypesForNotInstantiatedTypes", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def createdComponents = selectComponentTypes {
                    not {
                        instantiated()
                    }
                }.createPrototype()

                scriptLogger.infoFormat("Created '{0}' component prototypes.",
                    createdComponents.size())
            }
        }
    }
}

```

Listing 4.279: Create component prototypes for not instantiated types

### Specify the component prototype instantiation in createPrototypeWith(Closure)

`IComponentPrototypeCreator` provides an API to control some aspects, e.g. the naming, of newly created components.

- `name(Closure)` computes a name for the component prototypes that should be created for, by the `IComponentTypeSelection` provided, component types.
- `count(int)` defines how many component prototypes should be created for each selected component type. The default is 1.

### Examples for customizing the instantiation

```

import com.vector.cfg.model.sysdesc.api.component.IComponentType

scriptTask ("specifyNameOfCreatedComponent", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def createdComponents = selectComponentTypes {
                    application()
                }.createPrototypeWith {
                    name {
                        // define the naming of new created prototypes
                        IComponentType type -> type.getName() + "_postfix"
                    }
                }

                scriptLogger.infoFormat("Created '{0}' component prototypes.",
                    createdComponents.size())
            }
        }
    }
}

```

Listing 4.280: Specify name of created component

```

import com.vector.cfg.model.sysdesc.api.component.IComponentType

scriptTask ("specifyNameOfCreatedComponent", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def createdComponents = selectComponentTypes {
                    name ~ "App.*"
                }.createPrototypeWith {
                    name {
                        // you can still define a naming pattern
                        IComponentType type -> type.getName() + "_CP"
                    }

                    // and at the same time define how many prototypes should
                    // be created for each component type
                    count(3)
                }

                scriptLogger.infoFormat("Created '{0}' component prototypes.",
                    createdComponents.size())
            }
        }
    }
}

```

Listing 4.281: Create more than 1 component prototype

#### 4.10.4.15 Create Delegation Ports

The automation interface offers a possibility to create delegation ports at the ECU Composition of the FlatExtract. Therefore a port interface needs to be selected and a direction specified. Alternatively the component port and the origin port selections offer also APIs for that use case. For the origin context based creation of delegation ports see 4.10.4.15 on page 253 below, the component port based creation of delegation ports can be found in the examples at the end of following chapter ( 4.10.4.15 on page 252).

The entry points are the port interface selection (see 4.10.4.7 on page 198), the component port selection (see 4.10.4.1 on page 176) or the origin component port selection (see 4.10.4.8 on page 201).

**Instantiate Delegation Ports** `createPrototype(Closure)` creates a PortPrototype on the ecu composition of the FlatExtract for each selected port interface. If the naming is not specified, the names of the created delegation ports are derived from the selected port interfaces. The direction of the ports has to be specified.

##### Specify the delegation port prototype instantiation

`IDelegationPortCreator` provides an Api to control some aspects, e.g. the naming or the direction, of newly created delegation ports.

- `name(Closure)` computes a name for the delegation port prototypes that should be created for port interfaces, which are provided by the used selection API.
- `direction(EDirection)` defines the direction of the port prototype that should be created. `EDirection.Tx` will create provided ports, `EDirection.Rx` will create required ports. Delegation provided-required ports are not supported.

- `count(int)` defines how many delegation port prototypes should be created for each selected port interface. The default is 1.

For the origin and component port selection APIs the naming can be also done based on the selected ports.

- `nameFromOriginPort(Closure)` computes a name for the delegation port prototypes that should be created for origin ports, which are provided by the used `IOriginComponentPortSelection`.
- `nameFromComponentPort(Closure)` computes a name for the delegation port prototypes that should be created for component ports, which are provided by the used `IComponentPortSelection`.

### Examples

```

import com.vector.cfg.model.sysdesc.api.com.EDirection

scriptTask("createDelegationPortSimple", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def createdComponentPorts =
                    selectPortInterfaces {
                        // select all client server application port interfaces of
                        component 'App3'
                        componentType "App3"
                        clientServer()
                        application()
                } createPrototype {
                    // EDirection.Tx to create a PPort and EDirection.Rx to create
                    // a RPort
                    direction(EDirection.Tx)
                    // if no name is specified, the name of the port interface will
                    // be taken for the port
                }
                scriptLogger.infoFormat("Created {0} delegation ports.",
                    createdComponentPorts.size())
            }
        }
    }
}

```

Listing 4.282: Create delegation port simple

```
import com.vector.cfg.model.sysdesc.api.com.EDirection
import com.vector.cfg.model.sysdesc.api.port.IPortInterface

scriptTask("createDelegationPortCustomized", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def createdComponentPorts =
                    selectPortInterfaces {
                        // select the port interface of 'App1.ppFirst' and the port
                        // interface named 'Second'
                        or {
                            componentPort "App1.ppFirst"
                            name "Second"
                        }
                } createPrototype {
                    name {
                        // specify the naming of the new ports
                        IPortInterface portInterface -> "pp" + portInterface.
                            getName() + "_new"
                    }
                    // EDirection.Rx is leading to the creation of required ports
                    direction(EDirection.Rx)
                    // for each selected port interface two delegation ports will
                    // be created
                    count(2)
                }
                scriptLogger.infoFormat("Created {0} delegation ports.",
                    createdComponentPorts.size())
            }
        }
    }
}
```

Listing 4.283: Create delegation port customized

```

import com.vector.cfg.model.sysdesc.components.IComponentPort

scriptTask("createDelegationPortFromComponentPort", DV_PROJECT){

    // in this example we want to create a delegation port for an existing SWC
    // port

    code {
        transaction {
            domain.runtimeSystem {
                def createdComponentPorts =
                    selectComponentPorts {
                        component "App3"
                        name "rOtherSR1Port"
                    } createDelegationPorts {

                        // we can use the selected component port to specify the name
                        // of the new port
                        // also we could have used the name(Closure) method from
                        // examples above and use the port interface for naming
                        nameFromComponentPort {
                            // specify the naming of the new ports
                            IComponentPort componentPort -> componentPort.getPortName()
                                + "_new"
                        }
                        // since we have selected a component port previously, we can
                        now use the direction of it
                        // of course it is also possible to use direction(EDirection)
                        // here as in the examples above
                        useDefaultDirection()
                    }
                    scriptLogger.infoFormat("Created {0} delegation ports.", createdComponentPorts.size())
                }
            }
        }
    }
}

```

Listing 4.284: Create delegation port based on existing component port

**Create Delegation Ports Using Origin Context** We have already learned how to create new delegation ports in the flat extract using the port interface selection (see 4.10.4.15 on page 250). Sometimes a delegation connection was not completed in the structured extract because the delegation port was missing and is not flattened out. To complete this connection in the flat extract we need a new delegation port and want to use exactly the same port interfaces as in the structured extract. For this use case there is a shortcut directly at the origin component port selection, that simplifies the transition from origin port to its port interface. To specify the name of the new port, the API offers to do it using the port interface or the origin component port itself (see examples below).

For more details about origin context see 4.10.4.8 on page 200.

`createFlatExtractDelegationPorts(Closure)` creates a PortPrototype on the ecu composition of the FlatExtract for each selected origin component port using their port interfaces. If the naming is not specified, the names of the created delegation ports are derived from the port interfaces. The direction of the ports has to be specified.

## Examples

```
import com.vector.cfg.model.sysdesc.api.com.EDirection

scriptTask("createPortFromOriginContext", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {

                // in this example we create a delegation port which is missing in the
                // flat extract
                // the port interface for our new port is provided by an origin context
                // port

                def selectedOriginPorts
                def createdDelegationPorts = selectOriginComponentPorts {
                    provided()
                    name "OriginContext"

                    // remember the selected ports to print better info later
                    selectedOriginPorts = getSelectedOriginComponentPorts()

                }.createFlatExtractDelegationPorts {
                    // this call retrieves the port interfaces of the selected origin
                    // component ports
                    // and creates for each origin component port a delegation port in
                    // the flat extract

                    // specify the name and direction of the new port
                    name {
                        // optionally you could use the port interface as help for
                        // specifying the name here
                        // IPortInterface portInterfaceOfOriginPort -> ...
                        "PortWithInterfaceOfOriginContext"
                    }
                    direction(EDirection.Tx)
                }

                for (int i = 0; i < selectedOriginPorts.size(); i++) {
                    scriptLogger.infoFormat("Created new delegation port {0} for origin
                        port {1}.",
                        createdDelegationPorts.get(i).getName(),
                        selectedOriginPorts.get(i).getName())
                }
            }
        }
    }
}
```

Listing 4.285: Create a delegation port using the port interface of an origin context port

```

import com.vector.cfg.model.sysdesc.api.origin.IOriginComponentPort

scriptTask("createPortFromOriginContextUsingPort", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {

                // in this example we create a delegation port which is missing in the
                // flat extract
                // the port interface for our new port is provided by an origin context
                // port
                // and we use the original port for naming and direction this time

                def selectedOriginPorts
                def createdDelegationPorts = selectOriginComponentPorts {
                    provided()
                    name "OriginContext"

                    // remember the selected ports to print better info later
                    selectedOriginPorts = getSelectedOriginComponentPorts()

                }.createFlatExtractDelegationPorts {

                    // specify the name and direction of the new port
                    nameFromOriginPort { IOriginComponentPort originPort ->
                        originPort.getPortName() + "_new"
                    }
                    // for the origin component port selection we can use the direction
                    // of the previously selected origin ports
                    useDefaultDirection()
                }

                for (int i = 0; i < selectedOriginPorts.size(); i++) {
                    scriptLogger.infoFormat("Created new delegation port {0} for origin
                        port {1}.",
                        createdDelegationPorts.get(i).getName(),
                        selectedOriginPorts.get(i).getName())
                }
            }
        }
    }
}

```

Listing 4.286: Create a delegation port using the origin context port to specify name and direction

#### 4.10.4.16 Task Mapping

The task mapping use case allows to map executable entities (also called functions) directly or using their events (also called triggers) to tasks.

The entry point for the task mapping is either to select events (see 4.10.4.5 on page 192) or executable entities (see 4.10.4.6 on page 195). After that a task can be selected and the task mappings customized.

##### Mapping to a Task

**Event selection** `mapToTask(Closure)` tries to perform a task mapping for the selection of events (triggers). Inside the closure the task mapping can be controlled, e.g. selecting the task

to which the events should be mapped to and order the event's positions. Does not consider events (triggers) which do not reference an executable entity (function).

**ExecutableEntity selection mapToTask(Closure)** tries to perform a task mapping for the selection of executable entities (functions). Inside the closure the task mapping can be controlled, e.g. selecting the task to which the events (triggers) of the selected executable entities should be mapped to and order the event's positions.

**Select a task** Exactly one task has to be selected to perform a task mapping. Since the task selection is only available for the task mapping use case there is no own chapter for it.

`selectTask(Closure)` allows to define predicates to select a task for the task mapping.

Per default the predicates are combined via logical AND. To realize other combinations, use the 'or', 'not' and 'and' predicates.

### Task Predicates

- `name(String)` matches tasks with the given task name.
- `name(Pattern)` matches tasks with the given task name pattern.
- `core(String)` matches tasks running on a core with the given name / number (whether a core name or a core number is used, depends on the OS, if core number is used the String to be matched is 'Core<number>', e.g. 'Core1').
- `core(Pattern)` matches tasks running on a core with the given name pattern / number pattern (whether a core name or a core number is used, depends on the OS, if core number is used the String to be matched is 'Core<number>', e.g. 'Core1').
- `application(String)` matches tasks which belong to an application with the given name.
- `application(Pattern)` matches tasks which belong to an application whose name matches the given name pattern.
- `numberOfTaskMappings(int)` matches tasks which already have the given number of task mappings. The predicate can also be used to search for empty tasks with '0' as argument.
- `priority(BigInteger)` matches tasks with the given priority value.
- `filterAdvanced(Closure)` matches tasks for which the given closure results to true.
- `and(Closure)` combines the predicates inside the closure with a logical AND.
- `or(Closure)` combines the predicates inside the closure with a logical OR.
- `not(Closure)` negates the combination of predicates inside the closure.

### Examples for mapToTask(Closure)

```
import com.vector.cfg.model.sysdesc.api.taskmapping.ITaskMapping

scriptTask ("doTaskMappingOfApp1", DV_PROJECT ) {
    code {
        transaction {
            domain.runtimeSystem {
                def taskMappings = selectEvents {
                    // select all events of component App1
                    component("App1")
                } mapToTask {
                    selectTask {
                        // select a task
                        name("OsTask")
                    }
                }

                scriptLogger.infoFormat("Created '{0}' task mappings.",
                    taskMappings.size())

                // let's print more information to check the created task
                // mappings
                for (ITaskMapping taskMapping : taskMappings) {
                    scriptLogger.infoFormat("Mapped '{0}' triggered by '{1}' to
                        position '{2}' on task '{3}'.",
                        taskMapping.getExecutableEntity().getName(),
                        taskMapping.getEvent().getName(),
                        taskMapping.getPositionInTask(),
                        taskMapping.getMappedTask().getName())
                }
            }
        }
    }
}
```

Listing 4.287: Perform task mapping example

```

import com.vector.cfg.model.sysdesc.api.taskmapping.IEvent
import com.vector.cfg.model.mdf.ar4x.swcomponenttemplate.swcinternalbehavior.
    rteevents.MIDataReceivedEvent

scriptTask ("advancedFilterForEvents", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def taskMappings = selectEvents {
                    // use advanced filter if you cannot find a suitable
                    // predicate
                    filterAdvanced { IEvent event ->
                        // use the mdf model if the IEvent does not offer
                        // required methods
                        def mdfEvent = event.getMdfObject()

                        // for example, filter for data received events
                        // with special criteria
                        if (mdfEvent instanceof M IDataReceivedEvent) {
                            // filter here for the special criteria
                            return true
                        }

                        // do not select other events
                        return false
                    }
                } mapToTask {
                    selectTask {
                        name("OtherName")
                    }
                }

                scriptLogger.infoFormat("Created '{0}' task mappings.",
                    taskMappings.size())
            }
        }
    }
}

```

Listing 4.288: Advanced Filter for Events

**Additional Comfort Functions** The API provides some comfort functions listed below.

**Combine via Symbol** `combineViaSymbol(boolean)` determines whether the BswModuleEntities and the RunnableEntities should be combined using their symbol. That means they will be mapped to the same position on the same task. It is enough to select only the RunnableEntity or only the BswModuleEntity, when using this option both will be mapped. The default is true.

The condition is that the symbol of a RunnableEntity and the BswModuleEntry short name of a BswModuleEntity are equal.

### Example

```
scriptTask ("combineViaSymbol", DV_PROJECT ){  
    code {  
        transaction {  
            domain.runtimeSystem {  
                def taskMappings = selectEvents {  
                    component("Service1")  
                    timing()  
                } mapToTask {  
                    selectTask {  
                        name("OtherName")  
                    }  
                    // the default is true  
                    // call this if you do not want to combine runnables and  
                    // bsw module entities via their symbol  
                    combineViaSymbol(false)  
                }  
  
                scriptLogger.infoFormat("Created '{0}' task mappings.",  
                    taskMappings.size())  
            }  
        }  
    }  
}
```

Listing 4.289: Do not combine runnable and bsw module entity via symbol

**Map Events of a Runnable Entity Together** `mapAllEventsOfRunnableEntity(boolean, boolean)` is a possibility to map all events of a RunnableEntity to the same position on a task. In case of the selection of events, the task mapping will be extended, by all events (triggers) of runnable entities (functions) for which at least one event (trigger) is selected.

With help of the two boolean arguments, the behavior of ignoring already mapped events and ignoring events whose mapping is optional can be controlled.

### Example

```

scriptTask ("mapAllEventsOfRunnable", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def taskMappings = selectEvents {
                    name("background_event")
                    component("App1_1")
                } mapToTask {
                    selectTask {
                        name("OsTask")
                    }
                    // decide whether to consider only unmapped events
                    // and whether to consider only events whose mapping is
                    mandatory
                } mapAllEventsOfRunnableEntity(true, false)
            }
            scriptLogger.infoFormat("Created '{0}' task mappings.",
                taskMappings.size())
        }
    }
}

```

Listing 4.290: Map all events of a runnable together

**Order Task Mappings by Defining Successor Mappings** If you do not care about the absolute position value of the task mappings, but want to define an order, there is an option to specify successor relationships between the selected task mappings. This is the preferred way to define an order to using `order(Closure)` which will be introduced below, since it is easier to use in most cases. One exception is for example if you already read in a sorted list of executable entity names from external files.

`defineSuccessors(Closure)` allows to specify the order of the selected task mappings by defining successor relationships between the task mappings. The order defined by this method may still be overridden by `order(Closure)` and `queue()` if used.

First you have to specify one task mapping as the starting point.

`forTaskMapping(Closure)` allows to select a starting task mapping for which successors can be defined.

After that you can define a direct successor or a (logical) successor for the previously selected task mapping.

`successor(Closure)` allows to select a successor for the previously selected task mapping of the sequence. The sequence can be continued by another `successor(Closure)` or `directSuccessor(Closure)` call. To start a new sequence call `ITaskMappingSuccessorDefinition.forTaskMapping(Closure)`.

`directSuccessor(Closure)` allows to select a direct successor for the previously selected task mapping of the sequence. The sequence can be continued by another `successor(Closure)` or `directSuccessor(Closure)` call. To start a new sequence call `ITaskMappingSuccessorDefinition.forTaskMapping(Closure)`.

Within these calls you can select task mappings using task mapping predicates (see 4.10.4.16 on page 270). If the task mappings cannot be ordered to match the defined rules (for example if cycles were defined), an exception is thrown.

## Example

```

import com.vector.cfg.model.sysdesc.api.taskmapping.ITaskMapping

scriptTask ("defineSuccessorsSimple", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {

                // you can use both the event selection or the executable
                // entity selection API here
                def taskMappings = selectEvents {
                    component("App1")
                } mapToTask {
                    selectTask {
                        name("OsTask")
                    }
                    defineSuccessors {
                        forTaskMapping {
                            // define task mapping predicates to select the
                            // start of your sequence
                            // here we really use the predicates for task
                            // mappings, not for events nor executable entities
                            // but the task mapping selection offers us a lot
                            // of predicates
                            // it allow us to filter e.g. for the events which
                            // the task mappings reference or the executable
                            // entity which is triggered by the referenced
                            // event
                            executableEntity "Runnable1"
                        } successor {
                            // now define predicates to select successors for
                            // Runnable1
                            executableEntity "Runnable2"
                        } successor {
                            // we can continue by defining the successors for
                            // Runnable2 now
                            externalTrigger()
                        }
                    }
                }
                // so the result on OsTask will be Runnable1 -> Runnable2 ->
                // all runnables triggered by external trigger events

                scriptLogger.infoFormat("Created '{0}' task mappings.",
                    taskMappings.size())

                for (final ITaskMapping taskMapping : taskMappings) {
                    scriptLogger.infoFormat("Mapped runnable {0} to position
                        {1}.",
                        taskMapping.getExecutableEntity().getName(),
                        taskMapping.getPositionInTask())
                }
            }
        }
    }
}

```

Listing 4.291: Order the task mappings by defining successor relationships

```

import com.vector.cfg.model.sysdesc.api.taskmapping.ITaskMapping

scriptTask ("successorForGroup", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {

                def taskMappings = selectExecutableEntities {
                    runnableEntity()
                } mapToTask {
                    selectTask {
                        name("OsTask")
                    }
                    defineSuccessors {
                        // you do NOT have to narrow the selection down to one
                        // task mapping
                        // we want to order the task mappings by their SWC
                        // owner in this example

                        // we want the runnables of App1 be executed before the
                        // runnables of App2 and App3
                        // but we do not care about any order for the runnables
                        // of App2 and App3 or want define them later

                        // execute runnables of App1 before runnables of App2
                        forTaskMapping {
                            component "App1"
                        } successor {
                            component "App2"
                        }

                        // you can define multiple sequences for the same task
                        // mappings
                        // execute runnables of App1 before runnables of App3
                        forTaskMapping {
                            component "App1"
                        } successor {
                            component "App3"
                        }
                    }
                }

                // the script will map the runnables to OsTask and guarantees
                // that the runnables of App1 are mapped before the runnables
                // of App2 and App3

                scriptLogger.infoFormat("Created '{0}' task mappings.",
                    taskMappings.size())

                for (final ITaskMapping taskMapping : taskMappings) {
                    scriptLogger.infoFormat("Mapped runnable {0} to position
                        {1}.",
                        taskMapping.getExecutableEntity().getName(),
                        taskMapping.getPositionInTask())
                }
            }
        }
    }
}

```

Listing 4.292: Order groups by using successor calls

```
// in this example we have a task on which periodically 50ms trigger are mapped  
// after periodically 10ms trigger  
// additionally we want the new task mappings will be insert always at the  
// bottom of each group  
// so new 50ms triggered task mappings shall be inserted below the other 50ms  
// triggered task mappings  
// and the new 10ms below the other 10ms triggered task mappings  
  
scriptTask("DefineSuccessorsAndInsertBelowAlreadyMapped", DV_PROJECT){  
    code {  
        transaction {  
            domain.runtimeSystem {  
  
                def result = selectEvents {  
                    or {  
                        timing(0.01)  
                        timing(0.05)  
                    }  
                }.mapToTask {  
                    selectTask {  
                        name("OsTask")  
                    }  
                    mapAllEventsOfRunnableEntity(false, true)  
  
                    // example continues on next page  
                }  
            }  
        }  
    }  
}
```

Listing 4.293: Insert new task mappings always below existing - Part 1

```
defineSuccessors {

    // you can define it in one forTaskMapping sequence if you want to
    // we use in this example multiple sequences to demonstrate that it is
    // possible to split one complex rule into a few simple rules
    // Note: we use task("OsTask") and the negation instead of mapped() /
    // unmapped(), to eventually correct 10ms and 50ms trigger mapped to other
    // tasks by accident

    // at first define that 10ms trigger shall be mapped ahead of 50ms trigger
    forTaskMapping {
        timing(0.01)
    }.successor {
        timing(0.05)
    }

    // already to 'OsTask' mapped 10ms trigger shall be mapped ahead of the
    // other 10ms trigger
    forTaskMapping {
        timing(0.01)
        task("OsTask")
    }.successor {
        timing(0.01)
        not {
            task("OsTask")
        }
    }

    // and already to 'OsTask' mapped 50ms trigger ahead of the other 50ms
    // trigger
    forTaskMapping {
        timing(0.05)
        task("OsTask")
    }.successor {
        timing(0.05)
        not {
            task("OsTask")
        }
    }
}

scriptLogger.infoFormat("Created {0} task mappings.", result.
    size())
}
```

Listing 4.294: Insert new task mappings always below existing - Part 2

**Additionally Sort Successors** When you automate the task mapping using successor definition, there might be use cases where you still need the mappings in a 'well human readable' form. For example if you look them up in the task mapping editor from time to time. In that case you can sort the task mappings using the methods below so you can find them faster in the GUI. The sorting algorithm will respect the logical structure that is defined by the successor calls, so that these constraints are still guaranteed.

Quick example, if you define that all periodical 50ms trigger shall be mapped after the periodical 10ms trigger, all 50ms trigger will be sorted among the other 50ms trigger and all 10ms among

the other 10ms trigger, without messing up the successor constraint.

`sortSuccessorsInternally(Comparator)` allows to define an additional comparator to increase the overview. Therefore the defined successors will be analyzed and divided in logical groups. The given comparator is applied on each logical group separately, so that the defined successor relationships will not be violated.

Hint: You can use only one comparator at the same time. So defining a custom comparator and using a default comparator at the same time will cause an exception.

The following default comparators can be used instead:

`sortSuccessorsInternallyByExecutableEntity()` - sorts the task mappings alphabetically by their executable entity names.

`sortSuccessorsInternallyByExecutableEntity()` sorts the task mappings of the defined successors alphabetically by their executable entity names. See `sortSuccessorsInternally(Comparator)` for more details.

To use a custom comparator use `sortSuccessorsInternally(Comparator)` instead.

### Example

```

// in this example we have a task were all mode exit events shall be mapped
// ahead of all mode entry events
// but additionally we want to sort them alphabetically without breaking this
// constraint

// the result will be a task on which all mode exit events are mapped first,
// sorted alphabetically
// followed by as well alphabetically sorted mode entry events

scriptTask("DefineSuccessorsAndSort", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {

                def result = selectEvents {
                    or {
                        modeExit()
                        modeEntry()
                    }
                }.mapToTask {
                    selectTask {
                        name("OsTask")
                    }
                    mapAllEventsOfRunnableEntity(false, true)

                    defineSuccessors {
                        forTaskMapping {
                            modeExit()
                        }.successor {
                            modeEntry()
                        }

                        // you can use an own comparator calling
                        // sortSuccessorsInternally(Comparator<ITaskMapping>)
                        // in our example we sort alphabetically by executable entity
                        // names with a default comparator
                        sortSuccessorsInternallyByExecutableEntity()
                    }
                }
            }

            scriptLogger.infoFormat("Created {0} task mappings.", result.size())
        }
    }
}
}

```

Listing 4.295: Define successors and sort elements for better overview

**Specify an Order** The order of the task mappings can be specified also with the help of an internal structural element, the so called position in task entry.

An **IPositionInTaskEntry** represents a position in task for the task mapping. The entry is able to combine several events that are mapped to one position (e.g. needed when mapping a main function of a service component and its corresponding schedulable entity).

**order(Closure)** allows to evaluate and change the order of the task mappings. The received **IPositionInTaskEntry**s are already sorted respecting the defined successors by **defineSuccessors(Closure)** code. If used, the **queue()** option may override the order defined by this **order(Closure)** call.

It provides a possibility to order the already existing task mappings of the selected task and the new task mappings that should be created.

### Example

```

import com.vector.cfg.model.sysdesc.api.taskmapping.IPositionInTaskEntry
import com.vector.cfg.model.sysdesc.api.taskmapping.ITaskMapping

scriptTask ("orderTaskMappingsSimple", DV_PROJECT ) {
    code {
        transaction {
            domain.runtimeSystem {
                def taskMappings = selectEvents {
                    component("App1")
                } mapToTask {
                    selectTask {
                        name("OsTask")
                    }
                    order {
                        List<IPositionInTaskEntry> entries ->

                            for (IPositionInTaskEntry entry : entries) {
                                // identify by executable entity name and set the
                                // position
                                if (entry.getTriggeredExecutableEntity().equals("DataSendComp")) {
                                    // Runnable 'DataSendComp' will be mapped
                                    // to position 0 on task 'OsTask'
                                    entry.setPosition(0)
                                    continue
                                } else if (entry.getTriggeredExecutableEntity().equals("Runnable1")) {
                                    entry.setPosition(1)
                                    continue
                                } else if (entry.getTriggeredExecutableEntity().equals("Runnable2")) {
                                    entry.setPosition(2)
                                    continue
                                }
                            }
            }
        }
        // print info to the console which runnable was mapped to which
        // position
        for (final ITaskMapping taskMapping : taskMappings) {
            scriptLogger.infoFormat("Mapped runnable {0} to position
                {1}.",
                taskMapping.getExecutableEntity().getName(),
                taskMapping.getPositionInTask())
        }
    }
}
}

```

Listing 4.296: Simple example of ordering the task mappings

```
import com.vector.cfg.model.sysdesc.api.taskmapping.IPositionInTaskEntry

scriptTask ("orderTaskMappings", DV_PROJECT ) {
    code {
        transaction {
            domain.runtimeSystem {
                def taskMappings = selectEvents {
                    component("App1")
                } mapToTask {
                    selectTask {
                        name("OtherName")
                    }
                    order {
                        List<IPositionInTaskEntry> entries ->
                        int mappedIndex = 0
                        int index = 10

                        for (IPositionInTaskEntry entry : entries) {
                            // identify by executable entity name
                            if (entry.getTriggeredExecutableEntity().equals("DataSendComp")) {
                                entry.setPosition(9)
                                continue;
                            }

                            // already mapped on task
                            def alreadyMapped = entry.getAssociatedTaskMappings()
                                ().find {
                                    taskMapping -> taskMapping.getMappedTask() !=
                                        null
                                }
                            if (alreadyMapped != null) {
                                entry.setPosition(mappedIndex)
                                mappedIndex++
                                continue;
                            }

                            // newly mapped
                            entry.setPosition(index)
                            index++
                        }
                    }
                }
            }
        }
    }

    scriptLogger.infoFormat("Created '{0}' task mappings.", taskMappings.size())
}
}
```

Listing 4.297: Manually order the task mappings

```

import com.vector.cfg.model.sysdesc.api.taskmapping.IPositionInTaskEntry

scriptTask ("orderTaskMappingsOfOsTask", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def taskMappings = selectEvents {
                    task("OsTask")
                } mapToTask {
                    filterTaskMappings {
                        task("OsTask")
                    }
                } selectTask {
                    name("OsTask")
                } order {
                    List<IPositionInTaskEntry> entries ->

                    // in this example runnables of App1, App2 and App3 (
                    // with only 1 task mapping) are mapped on OsTask
                    // sort the runnables by owner
                    int runnablesOfApp1 = 0
                    int runnablesOfApp2 = 0
                    for (IPositionInTaskEntry entry : entries) {
                        if (entry.getOwner().equals("Component App1")) {
                            runnablesOfApp1++;
                        }
                        if (entry.getOwner().equals("Component App2")) {
                            runnablesOfApp2++;
                        }
                    }

                    // we sort in this example first runnables of 'App1'
                    // followed by the runnables of 'App2'
                    // and last but not least the runnable of 'App3'
                    int maxIndex = entries.size() - 1
                    int indexForApp1 = 0
                    int indexForApp2 = runnablesOfApp1

                    for (IPositionInTaskEntry entry : entries) {
                        // the runnable of App3 should be mapped to the
                        // last position on OsTask
                        if (entry.getOwner().equals("Component App3")) {
                            entry.setPosition(maxIndex);
                        }
                        if (entry.getOwner().equals("Component App1")) {
                            entry.setPosition(indexForApp1);
                            indexForApp1++
                        }
                        if (entry.getOwner().equals("Component App2")) {
                            entry.setPosition(indexForApp2);
                            indexForApp2++
                        }
                    }
                }
            }
            scriptLogger.infoFormat("Created '{0}' task mappings.",
                taskMappings.size())
        }
    }
}

```

Listing 4.298: Order task mappings on OsTask

**Filter Task Mappings** There is a way to narrow down the selected task mappings after selecting events or executable entities. This might be helpful especially in case you use multi-instantiation of software components. Since the selection of task mappings is only available for the task mappings use case, there is no own chapter for it.

`filterTaskMappings(Closure)` allows to filter the task mappings that should be created. This might be especially helpful to narrow down the task mappings after selecting events or executable entities when using multi instantiation (e.g. to filter the task mappings for only one instance of a multi instantiated component prototype).

Per default the predicates are combined via logical AND. To realize other combinations, use the 'or', 'not' and 'and' predicates.

### Task Mapping Predicates

- `component(String)` matches task mappings whose event is part of the internal behavior of a component with the given component name.
- `component(Pattern)` matches task mappings whose event is part of the internal behavior of a component with the given component name pattern.
- `moduleConfiguration(String)` matches task mappings whose event is part of the internal behavior of a module configuration with the given module configuration name.
- `moduleConfiguration(Pattern)` matches task mappings whose event is part of the internal behavior of a module configuration with the given module configuration name pattern.
- `moduleConfigurationAsrPath(String)` matches task mappings whose event is part of the internal behavior of a module configuration with the given module configuration autosar path.
- `moduleConfigurationAsrPath(Pattern)` matches task mappings whose event is part of the internal behavior of a module configuration with the given module configuration autosar path pattern.
- `unmapped()` matches task mappings which are not mapped to a task.
- `mapped()` matches task mappings which are mapped to a task.
- `task(String)` matches task mappings which are mapped to a task with the given task name.
- `task(Pattern)` matches task mappings which are mapped to a task whose name matches the given task name pattern.
- `componentType(String)` matches task mappings which belongs to component types with the given component type name.
- `componentType(Pattern)` matches task mappings which belongs to component types matching the given component type name pattern.
- `componentTypeAsrPath(String)` matches task mappings which belongs to component types with the given component type autosar path.
- `componentTypeAsrPath(Pattern)` matches task mappings which belongs to component types matching the given component type autosar path pattern.

- `event(String)` matches task mappings for events with the given event name.
- `event(Pattern)` matches task mappings for events matching the given event name pattern.
- `eventAsrPath(String)` matches task mappings for events with the given event autosar path.
- `eventAsrPath(Pattern)` matches task mappings for events matching the given event autosar path pattern.
- `bswEvent()` matches task mappings for bsw events.
- `rteEvent()` matches task mappings for rte events.
- `timing()` matches task mappings for timing events.
- `timing(BigDecimal)` matches task mappings for timing events with the given period (seconds).
- `init()` matches task mappings for init events.
- `dataReception()` matches task mappings for data received events.
- `dataReceptionError()` matches task mappings for data receive error events.
- `dataSendCompletion()` matches task mappings for data send completed events.
- `operationInvoked()` matches task mappings for operation invoked events.
- `operationInvoked(String)` matches task mappings for operation invoked events which are invoked by an operation with the given `operationName`.
- `serverCallReturns()` matches task mappings for events which are asynchronous server call returns events.
- `modeSwitch()` matches task mappings for mode switch events.
- `modeEntry()` matches task mappings for mode switch events with activation kind ON-ENTRY.
- `modeExit()` matches task mappings for mode switch events with activation kind ON-EXIT.
- `modeTransition()` matches task mappings for mode switch events with activation kind ON-TRANSITION.
- `modeSwitchedAck()` matches task mappings for mode switched acknowledgement events.
- `externalTrigger()` matches task mappings for external trigger occurred events.
- `internalTrigger()` matches task mappings for internal trigger occurred events.
- `background()` matches task mappings for background events.
- `mandatory()` matches task mappings for events which must be mapped. (The mapping of operation invoked events is optional, so this predicate filters all operation invoked events.)
- `symbol(String)` matches task mappings for runnable entities with the given symbol and bsw schedulable entities whose corresponding bsw module entry short name matches the given symbol.

- `symbol(Pattern)` matches task mappings runnable entities whose symbol matches the given symbol pattern and bsw schedulable entities whose corresponding bsw module entry short name matches the given symbol pattern.
- `executableEntity(String)` matches task mappings for executable entities (functions) with the given name.
- `executableEntity(Pattern)` matches task mappings for executable entities (functions) with the given name pattern.
- `executableEntityAsrPath(String)` matches task mappings for executable entities (functions) with the given autosar path.
- `executableEntityAsrPath(Pattern)` matches task mappings for executable entities (functions) with the given autosar path pattern.
- `filterAdvanced(Closure)` matches task mappings for which the given closure results to true.

### Example

```

scriptTask ("firstTaskMappings", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def taskMappings = selectExecutableEntities {
                    componentType("App1")
                } mapToTask {
                    selectTask {
                        name("OsTask")
                    }
                }
                // in this example two components ('App1' and 'App1_1') are
                // of component type 'App1'
                // do the task mapping only for 'App1_1'
                filterTaskMappings {
                    component("App1_1")
                }
            }
            scriptLogger.infoFormat("Created '{0}' task mappings.",
                taskMappings.size())
        }
    }
}

```

Listing 4.299: Filter task mappings

**Apply Execution Order Constraints** An `ExecutionOrderConstraint` restricts the execution order of a set of `ExecutableEntities`. Therefore successor and direct successor relationships can be defined for executable entities (functions), but also for events (triggers).

Since the selection of execution order constraints is available only for the task mapping use case, there is no own chapter for it.

`selectExecutionOrderConstraints(Closure)` allows to define predicates to select execution order constraints that should be applied.

Per default the predicates are combined via logical AND. To realize other combinations, use the 'or', 'not' and 'and' predicates.

### Execution Order Constraint Predicates

- `name(String)` matches execution order constraints with the given execution order constraint name.
- `name(Pattern)` matches execution order constraints with the given execution order constraint name pattern.
- `filterAdvanced(Closure)` matches execution order constraints for which the given closure results to true.
- `and(Closure)` combines the predicates inside the closure with a logical AND.
- `or(Closure)` combines the predicates inside the closure with a logical OR.
- `not(Closure)` negates the combination of predicates inside the closure.

### Example

```

import com.vector.cfg.model.sysdesc.api.eoc.IExecutionOrderConstraint

scriptTask ("applyEOC", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {
                def selectedConstraints
                def taskMappings = selectExecutableEntities {
                    component("App1_1")
                } mapToTask {
                    selectTask {
                        name("OsTask")
                    }

                    // select execution order constraints that should be
                    // applied
                    selectExecutionOrderConstraints {
                        name("App1ExecutionOrderConstraint")
                        selectedConstraints =
                            getSelectedExecutionOrderConstraints()
                    }
                }
            }

            scriptLogger.infoFormat("Created '{0}' task mappings.",
                taskMappings.size())

            for (IEvolutionOrderConstraint eoc : selectedConstraints) {
                scriptLogger.infoFormat("Applied evolution order constraint
                    '{0}'.",
                    eoc.getName())
            }
        }
    }
}

```

Listing 4.300: Use execution order constraints for the task mapping

**Check Current Task Mapping** The event and the executable entity selections offer getters to retrieve task mappings. Therefore first the given predicate is evaluated to identify which events are selected, then all task mappings which references the selected events are collected.

`getTaskMappings()` retrieves all `ITaskMappings` for the selected events (see `getEvents()`).

Note:

1. In case of multi instantiation of component prototypes, the different instances share the same events, since the event is part of the internal behavior of the component type. Therefore if the event is selected, `getTaskMappings()` will always return the task mappings for all component prototypes.
2. Since this method can be run outside of a transaction, there might be selected events for which no task mapping container does exist yet. The container cannot be created by calling `getTaskMappings()`, so no task mapping can be returned. This happens if the system description is not synchronized, after changes in the structured extract were done (see Automation Interface Documentation, chapter about Model Synchronization for examples how to synchronize).

`getTaskMappings()` retrieves all `ITaskMappings` for the selected executable entities (see `getExecutableEntities()`).

## Example

```

import com.vector.cfg.model.sysdesc.api.taskmapping.ITaskMapping

scriptTask("checkTaskMapping", DV_PROJECT){
    code {

        // because we use only a getter method and no transaction below,
        // make sure that your system description is synchronized,
        // otherwise task mapping container may be missing or obsolete.
        // since Cfg 5.18 this call is enough to make sure your task mapping
        // containers are up to date
        modelSynchronization.synchronize()

        domain.runtimeSystem {
            def taskId = "OsTask"

            def taskMappings = selectEvents {
                task(taskId)
                // getTaskMappings will apply the predicate and return all task
                // mappings for the selected events
            } getTaskMappings()

            // be careful in case of multi-instantiated component prototypes,
            // since events and executable entities are part of the internal
            // behavior of the component type,
            // you will receive always the task mappings for all instances here
            // (even if the task mappings of the other component prototype
            // instances are not mapped to "OsTask")

            // print info to the console with owner of the task mapping and the
            // mapped task
            for (final ITaskMapping taskMapping : taskMappings) {
                scriptLogger.infoFormat("TaskMapping of '{1}' for event '{0}' is
                    mapped to '{2}'.",
                    taskMapping.getEvent().getName(),
                    taskMapping.getOwnerDescription(),
                    taskMapping.getMappedTask() == null ? "no task" : taskMapping.
                        getMappedTask().getName())
            }
        }
    }
}

```

Listing 4.301: Check which events are currently mapped to OsTask

**Keep Existing Task Mappings on Current Position** `queue()` is an option that allows to keep the task mappings which are already mapped to the selected task on the current position. The new task mappings will be placed in the defined order into existing gaps. That means they are mapped to the lowest free position. This method may override the order defined in `defineSuccessors(Closure)` and `order(Closure)`.

Example: The selected task 'Task1' has already a TaskMappingA at position 1 and a TaskMappingB at position 3. The task mappings TaskMappingC, TaskMappingD and TaskMappingE (in that order) should be mapped to the same task using the `queue()` option. The new task mappings will be assigned in the defined order to the next free position on 'Task1'.

So the result will be:

0 -> TaskMappingC (new1)  
 1 -> TaskMappingA (old)  
 2 -> TaskMappingD (new2)

3 -> TaskMappingB (old)  
 4 -> TaskMappingE (new3)

If the options `mapAllEventsOfRunnableEntity(boolean, boolean)` or `combineViaSymbol(boolean)` needs to combine an existing mapping with other mappings, the combined task mapping is seen as a new mapping. Its position will be assigned according to the rules of a new mapping explained above. The old mapping which was not combined will be removed. In other words combined task mappings are preferred over single task mappings.

### Example

```
scriptTask ("queueExample", DV_PROJECT ){
  code {
    transaction {
      domain.runtimeSystem {
        def taskMappings = selectExecutableEntities {
          componentType("App1")
        } mapToTask {
          selectTask {
            name("OsTask")
          }
          // this option will consider the existing mappings on the
          // task
          // if an existing task mapping is not combined for another
          // new incoming mapping, it will remain on its current
          // position
          queue()
        }

        // if the existing task mapping was combined for another new
        // incoming mapping, it is considered as a new mapping and will
        // appear in the taskMappings below
        scriptLogger.infoFormat("Created '{0}' task mappings.",
                               taskMappings.size())
      }
    }
  }
}
```

Listing 4.302: Use the queue option example

**Set Activation Offset** You can set the value of the activation offset for the events/executable entities which will be newly mapped.

`setActivationOffset(BigDecimal)` allows to set the activation offset at the created task mappings. The offset will be set for all created task mappings to the given `offsetInSeconds` value.

It is also possible to set the activation offset parameter value of a task mapping using the event or the executable entity selection without mapping the events/executable entities to a task. You can use this option for example if you just want to set the activation offset and do not care whether the event/executable is already mapped to a task or not.

`setActivationOffset(BigDecimal)` sets the activation offset at the task mapping containers for the selected events. If the symbol of a schedulable entity matches a runnable name, the activation offset will be set for both, even if only one of them is selected. (Matching works as for `ITaskMapper.combineViaSymbol(boolean)`.)

`setActivationOffset(BigDecimal)` sets the activation offset at the task mapping containers for the selected executable entities.

### Examples

```
import com.vector.cfg.model.sysdesc.api.taskmapping.ITaskMapping

scriptTask("setActivationOffset", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def taskMappings = selectEvents {
                    // define predicates to select your event
                    component("App1")
                    timing()

                    // set the activation offset parameter value
                    // for the task mapping to 10 ms
                    }.setActivationOffset(0.01)

                    // print info to the console with the runnable name which is
                    // triggered by the event
                    // and the activation offset of the task mapping
                    for (final ITaskMapping taskMapping : taskMappings) {
                        scriptLogger.infoFormat("TaskMapping of runnable {0} has the
                            activation offset {1}.",
                            taskMapping.getExecutableEntity().getName(),
                            taskMapping.getActivationOffset())
                    }
                }
            }
        }
    }
}
```

Listing 4.303: Set the activation offset using the event selection

```
import com.vector.cfg.model.sysdesc.api.taskmapping.ITaskMapping

scriptTask("setActivationOffset", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def taskMappings = selectExecutableEntities {
                    // define predicates to select your runnable
                    component("App1")
                    name("Runnable1")

                    // set the activation offset parameter value
                    // for the task mapping to 100 ms
                    }.setActivationOffset(0.1)

                    // print info to the console with runnable name and the activation
                    // offset
                    for (final ITaskMapping taskMapping : taskMappings) {
                        scriptLogger.infoFormat("TaskMapping of runnable {0} has the
                            activation offset {1}.",
                            taskMapping.getExecutableEntity().getName(),
                            taskMapping.getActivationOffset())
                    }
                }
            }
        }
    }
}
```

Listing 4.304: Set the activation offset using the executable entity selection

```

import com.vector.cfg.model.sysdesc.api.taskmapping.ITaskMapping

scriptTask("mapAndSetActivationOffset", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {
                def taskMappings = selectEvents {
                    // define predicates to select your event
                    component("App1")
                    init()
                } mapToTask {
                    // map the event to task 'OsTask'
                    selectTask {
                        name("OsTask")
                    }
                    // set the activation offset parameter value
                    // for the task mapping to 10 ms
                    setActivationOffset(0.01)
                }

                // print info to the console which runnable was mapped and the
                // activation offset
                for (final ITaskMapping taskMapping : taskMappings) {
                    scriptLogger.infoFormat("Mapped runnable {0} to position {1}"
                        with activation offset {2}.",
                        taskMapping.getExecutableEntity().getName(),
                        taskMapping.getPositionInTask(),
                        taskMapping.getActivationOffset())
                }
            }
        }
    }
}

```

Listing 4.305: Set the activation offset while mapping to a task

#### 4.10.4.17 Bridge Between MDF and Model Abstractions

The Runtime System Domain uses model abstractions to simplify the structure of the AUTOSAR model. All objects which you select using the selection APIs are model abstraction.

**IModelAbstraction** is the common super interface for all model abstractions (as e.g. **Object** for all java classes). It defines common functionality which all model abstractions provide for generic handling of model abstractions.

On MDF level the base interface for AUTOSAR model objects is the **MIOObject**.

It is possible to switch between model abstractions and MDF objects. This might be helpful for advanced script tasks that extend the current scope of the model abstractions.

**getModelAbstractionsForMdfObjects(Collection)** is a method for an arbitrary access to all model abstractions which correspond to the given collection of MDF objects.

**getMdfObject()** is a bridge from the **IModelAbstraction** to the underlying MDF object. For compound model abstractions, the main object will be returned, e.g. returns the port for a component port.

#### Example for navigating between MDF model and model abstractions

```

import com.vector.cfg.model.access.IReferrableAccess
import java.util.Collections
import com.vector.cfg.model.abstraction.api.IModelAbstraction
import com.vector.cfg.model.sysdesc.api.com.IAbstractSignalInstance

scriptTask ("switchBetweenMdfAndModelAbstraction", DV_PROJECT ){
    code {
        transaction {
            domain.runtimeSystem {

                // -----
                // get a model abstraction object for your MDF object
                // -----
                def referrableAccess = ScriptApi.activeProject.getInstance(
                    IReferrableAccess)

                // get some MDF objects by e.g. using the referrable access
                def mdfSystemSignal = referrableAccess.getReferrableByPath("/$VectorAutosarExplorerGeneratedObjects/SYSTEM_SIGNALS/Element_1_b16df82332bcf915")

                def mdfObjects = Collections.singletonList(mdfSystemSignal)

                // get the model abstractions for the MDF objects
                def modelAbstractions = getModelAbstractionsForMdfObjects(
                    mdfObjects)

                // for the system signal an IAbstractSignalInstance is returned
                // , if it is referenced by at least one ISignal
                // so there will be exactly one model abstraction in the
                // collection in this example
                def signalInstanceModelAbstraction
                for (IModelAbstraction modelAbstraction : modelAbstractions) {
                    if (modelAbstraction instanceof IAbstractSignalInstance) {
                        signalInstanceModelAbstraction = modelAbstraction
                    }
                }

                if (signalInstanceModelAbstraction == null) {
                    scriptLogger.infoFormat("System Signal '{0}' is not
                        referenced by any ISignals",
                        mdfSystemSignal.getName())
                }

                // -----
                // get a MDF object for your model abstraction object
                // -----
                def mdfObject = signalInstanceModelAbstraction.getMdfObject()
                // now the system signal can be used on MDF level
            }
        }
    }
}

```

Listing 4.306: Switch between MDF and model abstraction example

#### 4.10.4.18 Deleting Elements

Removing elements is not covered by the runtime system API yet. So we have to use the MDF model for that use case for now (see chapter for MDF model). You can of course use

the selection APIs to find the correct elements first (e.g. for the data mappings by selecting the signals and call `getDataMappings()` method) and then get their MDF objects by calling `getMdfObject()` which is supported for all objects of the runtime system domain model.

You can find examples for some common use cases below.

### Example

```

import com.vector.cfg.model.sysdesc.api.port.IComponentPort
import com.vector.cfg.model.sysdesc.api.connection.IConnector

scriptTask("diconnectDelegationPorts", DV_PROJECT) {
    code {
        transaction {
            domain.runtimeSystem {

                // in this example we want to delete the connectors to all
                // delegation ports

                // select the component ports which should be disconnected
                // first
                List<IComponentPort> connectedDelegationPorts =
                    selectComponentPorts {
                        connected()
                        delegation()
                    }.getComponentPorts()

                // get the connectors to those component ports
                List<IConnector> connectorsToDelegationPorts = []
                connectedDelegationPorts.each {
                    connectorsToDelegationPorts.addAll(it.
                        getConnectedConnectors())
                }

                // we want to do a report for the disconnected ports
                // therefore we will remember them
                List<IComponentPort> disconnectedPPorts = []
                List<IComponentPort> disconnectedRPorts = []

                // now delete the connectors
                connectorsToDelegationPorts.each { IConnector connector ->
                    // we need to check whether we did not already have deleted
                    // the connector
                    // and we need to check if the connector is deletable
                    // connectors which were not created in CFG5 cannot be
                    // deleted
                    if (!connector.getMdfObject().isDeleted()
                        && connector.getMdfObject().getCeState().
                            isDeletable()) {
                        // add the component ports for a final report
                        disconnectedPPorts.add(connector.getProvidedPort())
                        disconnectedRPorts.add(connector.getRequiredPort())
                        // finally delete the connector
                        connector.getMdfObject().delete()
                    }
                }

                for (int i = 0; i < disconnectedPPorts.size(); i++) {
                    scriptLogger.infoFormat("Deleted connector between {0} and
                        {1}.",
                        disconnectedPPorts.get(i).getName(),
                        disconnectedRPorts.get(i).getName())
                }
            }
        }
    }
}

```

Listing 4.307: Delete connectors to delegation ports

```
import com.vector.cfg.model.sysdesc.api.com.IDataMapping
import com.vector.cfg.model.sysdesc.api.com.IAbstractSignalInstance
import com.vector.cfg.model.sysdesc.api.com.ICommunicationElement
import com.vector.cfg.model.sysdesc.api.com.ISenderReceiverDataMapping

scriptTask("deleteDataMappingsForDelPorts", DV_PROJECT) {
    code {
        transaction {
            domain.runtimeSystem {

                // in this example we want to delete the data mappings of
                // sender receiver delegation ports

                // select the communication elements first
                List<ICommunicationElement> communicationElements =
                    selectCommunicationElements {
                        delegation()
                        senderReceiver()
                        mapped()
                    }.getCommunicationElements()

                // get the data mappings of these communication elements
                List<IDataMapping> dataMappingsToDelete = []
                communicationElements.each {
                    dataMappingsToDelete.addAll(it.getDataMappings())
                }
                // example continues on next page
            }
        }
    }
}
```

Listing 4.308: Delete data mappings of sender receiver delegation ports - Part 1

```
// we want to do a report for the removed data mappings
// therefore we will remember communication element and signal
List<ICommunicationElement> comElementsOfDeletedMapping = []
List<IAbstractSignalInstance> signalsOfDeletedMappings = []

// now delete the data mappings
dataMappingsToDelete.each { IDataMapping dataMapping ->
    // we need to check if the data mapping is deletable
    // data mappings which were not created in CFG5 cannot be
    // deleted
    if (!dataMapping.getMdfObject().isDeleted()
        && dataMapping.getMdfObject().getCeState().
            isDeletable()) {
        // add the component ports for a final report
        comElementsOfDeletedMapping.add(dataMapping.
            getMappedCommunicationElement())
        signalsOfDeletedMappings.add(dataMapping.
            getMappedSystemSignal())

        // we want to extend the reporting for sender receiver
        // to signal group mappings
        // report not only the signal group but also the group
        // signal mappings
        if (dataMapping instanceof ISenderReceiverDataMapping) {
            ((ISenderReceiverDataMapping) dataMapping).
                getLeafDataMappings().each {
                    ISenderReceiverDataMapping childMapping ->
                        comElementsOfDeletedMapping.add(
                            childMapping.
                                getMappedCommunicationElement())
                        signalsOfDeletedMappings.add(childMapping.
                            getMappedSystemSignal())
                }
        }

        // finally delete the data mappings
        // for sender receiver to signal group mappings it is
        // enough to delete the root mapping
        dataMapping.getMdfObject().delete()
    }
}

for (int i = 0; i < comElementsOfDeletedMapping.size(); i++) {
    scriptLogger.infoFormat("Deleted data mapping for {0} to
        signal {1}",
        comElementsOfDeletedMapping.get(i).
            getFullyQualifiedName(),
        signalsOfDeletedMappings.get(i).getName())
}
```

Listing 4.309: Delete data mappings of sender receiver delegation ports - Part 2

#### 4.10.4.19 Variant Handling

The only use case that supports PostBuild selectable variance in the runtime system domain is the mapping between communication elements and signals (data mapping). If you have a variant project the data mapping can only be done in an active model view (see chapter about model views).

There is no edit variance function for the data mapping in the automation API yet. But if a signal is visible in exactly one variant, the created data mapping will automatically be created only for the variant in which the signal is visible. For invariant signals the created data mappings will also be invariant.

So a good approach for PostBuild variant configurations is to loop over the model views and run your script logic for each view.

```

import com.vector.cfg.model.view.IModelViewExecutionContext

// this example runs also successfully for project with no PostBuild variance
// because there just be only one model view - the invariant model view

scriptTask("dataMapVariant", DV_PROJECT) {
    code {
        transaction {
            domain.runtimeSystem {

                for (def modelView in variance.allPostBuildVariantViews) {
                    final IModelViewExecutionContext context = modelView.
                        executeWithThisView()

                    // make sure to close the view when finish (even if
                    // exceptions occur) to not run further actions still in
                    // this view by accident
                    context.withCloseable {

                        // do the data mapping inside this closure, we will
                        // keep the example simple here
                        // remember: IAbstractSignalInstances may also be
                        // variant
                        // so they should be selected also with an active model
                        // view
                        selectCommunicationElements {
                            // the auto mapper will not create mappings which
                            // are real duplicates
                            // but it is better in terms of performance to
                            // filter for unmapped here
                            unmapped()
                        }.autoMap()
                    }
                }
            }
        }
    }
}

```

Listing 4.310: Create variant data mappings

#### 4.10.4.20 Retrieving Short Name Paths and Fully Qualified Names

The runtime system automation API requires/allows to use short name paths and fully qualified names to select elements. This chapter describes how these can be retrieved.

In general you can find a 'Copy' -> 'Copy Short Name Path' command in the context menu for most elements in the GUI.

For the automation interface you can use appropriate getters depending on your use case.

**Path of Port Interface Mapping** If you have already connected two ports and use a port interface mapping for the connection, you can search for your connector in the ECU Software Components Editor. You can find it in the Application Ports grid or the Service Mappings grid under the ECU Composition node or under the Application or Service Ports node of your application or service component.

'Copy' -> 'Copy Short Name Path' in the context menu available in the Port Interface Mapping column for a given connection copies the short name path of the connection's port interface mapping to the clip board.

If you have no connection yet which uses the port interface mapping, search for the port interface mapping in another way. Expand the Port Interface Mapping Sets node of the ECU Software Components Editor and the node of the set which contains your mapping. Now you can do the 'Copy' -> 'Copy Short Name Path' command in the context menu of the tree node which belongs to the port interface mapping you were looking for.

**Paths of System Signals and System Signal Groups** GUI: If you have already used your signal / signal group for a data mapping, you can find the data mapping in the Data Mapping or the Application Ports grid under the ECU Composition node. Once you found your mapping you can retrieve the short name path of the signal / signal group via the 'Copy' -> 'Copy Short Name Path' context menu on the cell of the Signal column in the Data Mapping grid or the Mapped Signal column of the Application Ports grid.

AI: The following getter methods might be useful (see Javadoc of the method for more details):

- `IAbstractSignalInstance.getAutosarPath()`
- `IAbstractSignalInstance.getFullyQualifiedName()` - This getter might help you identifying group signals.

**Fully Qualified Name of Communication Elements** GUI: Communication elements are used for data mappings. Their fully qualified name is build out of three parts (in general, children of complex communication elements might have more). The name of the component, the name of the port and the name of the data element/operation/trigger itself. In case of a delegation port you can use 'ECU Composition' or 'COMPOSITIONTYPE' as component name.

#### Examples

Communication element of non-delegation port:  
'ComponentName.PortName.DataElementName'

Communication element of delegation port:  
'ECU Composition.DelegationPortName.DataElementName'

Complex communication element (record element of a record):  
'ComponentName.PortName.RecordName.RecordElement1Name'

Complex communication element (array element at certain index):  
'ComponentName.PortName.ArrayName[0]'

Open the data mapping assistant. You can find it in the navigation view under Runtime System -> Add Data Mapping or as hyperlink above grids which shows data mappings. Select the direction 'Find matching signals for the communication elements' on the first page and after that your communication elements on the second page. Now on the third page (Confirm page) you can open a context menu on the cell in the Communication Element column of your communication element and select copy fully qualified name. The name will be copied into the clip board. If you need the name of a child communication element which is not shown yet, you might have map the parent to a signal group first.

You can also just use the examples above and replace the names with the names of your component, port and communication element.

AI: The following getter methods might be useful (see Javadoc of the method for more details):

- `ICommunicationElement.getFullyQualifiedName()`

#### 4.10.4.21 Best Practice And Further Examples

**Create Selection Based on Existing Elements** Most selection APIs offer a put method at the selector. This method allows us to put elements into a selection and continue working with elements we already have created or selected previously. The purpose of that is to optimize performance, avoid defining the same predicates over and over again and increase the level of control.

```

import com.vector.cfg.model.sysdesc.api.origin.IOriginComponentPort
import com.vector.cfg.model.sysdesc.api.connection.IConnector
import com.vector.cfg.model.sysdesc.api.port.IComponentPort

scriptTask("createAndConnectDelegationPort", DV_PROJECT){
    code {
        transaction {
            domain.runtimeSystem {

                List<IComponentPort> createdDelegationPorts =
                    selectOriginComponentPorts {
                        // select some origin context ports
                        provided()
                        innerTopLevelDelegation()
                    }.createFlatExtractDelegationPorts {

                        // create a delegation port for each selected origin port
                        nameFromOriginPort { IOriginComponentPort originPort ->
                            originPort.getPortName()
                        }
                        useDefaultDirection()
                    }

                // now use the new ports to put them into a selection
                // we want to use the auto mapper (using the origin context names) to
                // connect them to inner SWC ports
                List<IConnector> createdConnections = selectComponentPorts {
                    put(createdDelegationPorts)
                    // we can still use predicates here, they would be applied only to
                    // our new ports which we put into the selection
                    // for example we could connect only the provided new delegation
                    // ports here using additionally the predicate provided()
                } autoMapTo {
                    useOriginContextForMatch()
                }

                // and finally do some reporting
                for (final IConnector connector : createdConnections) {
                    final IComponentPort pPort = connector.getProvidedPort()
                    final IComponentPort rPort = connector.getRequiredPort()

                    scriptLogger.infoFormat("Created new delegation port {0} and
                        connected it to {1}.",
                        pPort.isDelegation() ? pPort.getName() : rPort.getName(), // 
                            our new created delegation port
                        pPort.isDelegation() ? rPort.getName() : pPort.getName()) // 
                            the connected inner application port
                }

            }
        }
    }
}

```

Listing 4.311: Create delegation ports for selected origin ports and connect them

**Optimizing Performance** This chapter will give some advice on what may help if the script execution times get too high. In general we recommend to optimize the readability of your code in first place, but in some cases when processing a large number of objects restructure the code according to the points below may reduce the execution time of the script code.

**Check complexity of the script code:** Scripts may get slow if the code iterate over big collections of elements and while doing that performs performance critical operations or iterates over another big collection ( $n^2$  complexity). This may be also hidden somewhere between the lines. For example if the script needs to do many data mappings or connectors make sure you use the auto map call directly at the selection APIs and as few times as possible or use the simple API inside which you can put lists of elements at once instead of doing the calls for one single object after another.

In general the script performs better, when suitable data structures are prepared first so that the total amount of modification calls (e.g. autoMapTo(Closure) at the component port selection) at the selection APIs can be reduced.

**Do not open unnecessary transactions and not over extend usage of transactions:** Each transaction has an overhead, for example the validation results needs to be updated. So avoid opening multiple transactions if there is no benefit for you by doing that. A good example is if you want to create 500 data mappings. You can do every single mapping in an own transaction or all in one transaction. If you do not need the undo operation for every single mapping separately, you should always prefer to do all 500 mappings in one transaction.

**Narrow down selections where possible without significant effort:** For example when you connect ports and you have many already connected ports which you do not care about, use the unmapped predicate to not selecting them at all. The auto mapper will have to match less elements.

**Prefer reusing elements over selecting elements:** Let's assume you have created new delegation ports and want to connect them to other component ports. After creating the new ports use the returned list and put it into the component port selection to connect them. This will be more efficient then defining predicates that match to that newly created ports.

**Use @CompileStatic for advanced filters:** If your scripts use advanced filters whose closures need to be evaluated a large number of times, the code should be refactored so that the closure is built by an own method which uses @CompileStatic. That makes the advanced filter faster, since the closure needs to be compiled only one single time.

**Avoid switching very often between selecting and creating elements** We use internal caches for many attributes and dependencies between elements which needs to be invalidated every time when model changes are performed. So if your selection does not depend on the model changes which you want to do in next steps, prefer to select all elements you need first and then start modifying the model.

So for example instead of:

- select communication elements for sender receiver data mappings
- create sender receiver data mappings
- select operation communication elements for client server data mappings
- create client server data mappings

The following workflow will use the caches more efficient:

- select communication elements for sender receiver data mappings
- select operation communication elements for client server data mappings
- create sender receiver data mappings
- create client server data mappings

## 4.11 Unresolved Reference API

The Unresolved Reference APIs are specifically designed to manage unresolved references in specific scopes.

The Unresolved Reference API is the entry point for accessing the unresolved references. It is available in form of the `IUnresolvedReferenceApiEntryPoint` Interface.

The `IUnresolvedReferenceApi` provides the methods to access the different APIs which are designed to manage unresolved references of specific scopes. For an example see the `IEcucUnresolvedReferenceApi` 4.11.1.

`getUnresolvedReferences()` allows accessing the `IUnresolvedReferenceApi` like a property.

```
scriptTask('AccessAsPropertyTask') {
    code {
        // IUnresolvedReferenceApi is available as unresolvedReferences property
        def unresolvedReferencesApi = unresolvedReferences
    }
}
```

Listing 4.312: Accessing `IUnresolvedReferenceApi` as a property

`unresolvedReferences(Closure)` allows accessing the `IUnresolvedReferenceApi` in a scope-like way.

```
scriptTask('AccessLikeScopeTask') {
    code {
        unresolvedReferences {
            // IUnresolvedReferenceApi is available inside this Closure
        }
    }
}
```

Listing 4.313: Accessing `IUnresolvedReferenceApi` in a scope-like way

### 4.11.1 Active ECUC Unresolved Reference API

The ECUC Unresolved Reference Api is specifically designed to read and edit unresolved references of the active ECUC.

`getActiveEcuc()` allows accessing the `IEcucUnresolvedReferenceApi` like a property.

```
scriptTask('AccessApiAsPropertyTask') {
    code {
        def ecucUnresolvedReferenceApi = unresolvedReferences.activeEcuc
    }
}
```

Listing 4.314: Accessing `IEcucUnresolvedReferenceApi` as a property.

`activeEcuc(Closure)` allows accessing the `IEcucUnresolvedReferenceApi` in a scope-like way.

```
scriptTask('AccessApiLikeScopeTask') {
    code {
        unresolvedReferences.activeEcuc {
            // IEcucUnresolvedReferenceApi is available inside this Closure
        }
    }
}
```

Listing 4.315: Accessing `IEcucUnresolvedReferenceApi` in a scope-like way.

#### 4.11.1.1 Selecting unresolved references

`select(Closure)` allows the selection of all unresolved references of the active ECUC using predicates. The returned selection is read only. Use `selectChangeable(Closure)` for changes. The selection is returned by an `IEcucUnresolvedReferenceSelection`.

`selectChangeable(Closure)` allows the selection of the changeable unresolved references of the active ECUC using predicates. The selection is returned by an `IChangeableEcucUnresolvedReferenceSelection`.

#### Examples

```
scriptTask('SelectAllReferencesTask') {
    code {
        def selection = unresolvedReferences.activeEcuc.select {
            // filters can be set here
        }
    }
}
```

Listing 4.316: Get a filtered set of all unresolved references at the ECUC configuration.

```
scriptTask('SelectAllReferencesExampleTask') {
    code {
        def selection = unresolvedReferences.activeEcuc.select {
            owner(~ ".*/Rte/Com/.*")
        }
    }
}
```

Listing 4.317: Get a filtered set of all unresolved references at the ECUC configuration with a specific pattern at the owner path.

```
scriptTask('SelectChangeableTask') {
    code {
        def selection = unresolvedReferences.activeEcuc.selectChangeable {
            // filters can be set here
        }
    }
}
```

Listing 4.318: Get a filtered set of the changeable unresolved references at the ECUC configuration.

**Predicates** Predicates can be set by several methods. These can be used to filter the unresolved references.

- `reference(String)` filters the unresolved references by the reference path as String.
- `reference(Pattern)` filters the unresolved references by the reference path as a pattern.
- `owner(MIObject)` filters the unresolved references by the owner object.
- `owner(String)` filters the unresolved references by the exact owner path string.
- `owner(Pattern)` filters the unresolved references by the owner as a pattern.
- `container(MIContainer)` filters the unresolved references by a container. This container contains the referenced elements of the filtered references.

#### 4.11.1.2 Set changeable unresolved references

`setCommonReferencePath(AsrPath)` sets a common reference for all filtered unresolved references.

`setReferencesByFunction(Function)` sets a common reference for all filtered unresolved references by using a `String -> String` function.

`replaceInReferences(String, String)` sets a new reference for all unresolved references of the selection by replacing an included string.

`replacePrefixInReferences(String, String)` sets a new reference for all unresolved references of the selection by replacing a prefix of the unresolved references.

#### Examples

```
scriptTask('SetChangeableTask'){
    code{
        def selection = unresolvedReferences.activeEcuc.selectChangeable {
            // filters can be set here
        }
        transaction{
            // this path is a created AsrPath as an example. Use a valid path instead.
            def path = AsrPath.create("/Changed/Reference")
            selection.setCommonReferencePath(path)
        }
    }
}
```

Listing 4.319: Set changeable unresolved references.

```
scriptTask('ReplaceInChangeableTask'){
    code{
        def selection = unresolvedReferences.activeEcuc.selectChangeable {
            // filters can be set here
        }
        transaction{
            selection.replaceInReferences("MICROSAR", "OTHER")
        }
    }
}
```

Listing 4.320: Replace 'MICROSAR' with 'OTHER' in all changeable unresolved references.

```
scriptTask('ReplacePrefixInChangeableTask'){
    code{
        def selection = unresolvedReferences.activeEcuc.selectChangeable {
            // filters can be set here
        }
        transaction{
            selection.replacePrefixInReferences("/MICROSAR", "/OTHER")
        }
    }
}
```

Listing 4.321: Replace the prefix '/MICROSAR' with '/OTHER' in all changeable unresolved references.

## 4.12 Reporting

### 4.12.1 Custom Report

This API enables the user to create a report with a content which is free to determine. The columns and rows of the report are fully customizable. A practical use case would be: Look for particular parameter values and report only the values of those parameters.

```
scriptTask("TestCustomReportCreation", DV_PROJECT) {
code {
    activeProject {

        createCustomReport("My Customized Report", outputPathAsString) {

            addTable("Ecuc General Parameters") {
                // Configure the column names
                configure {
                    columns( "Shortname"
                            , "Init Phase"
                            , "Header"
                            , "Configuration Pointer Name" )
                }

                // Loop and print parameter into the report
                EcuC ecuC = bswmdModel(EcuC).single
                ecuC.ecucGeneral.bswInitialization.first.initFunction.each {
                    // Print on report
                    InitFunction initFunction ->
                        addRow( initFunction.shortname.toString()
                                , initFunction.initPhase.toString()
                                , initFunction.header.toString()
                                , initFunction.configPtrName.toString())
                } // End loop

            } // addTable
        } // createCustomReport
    } // activeProject
}
```

Listing 4.322: Create Custom Report

**ICreateCustomReportApi** This API is the entry point for creating the custom report.

Use `getCreateCustomReport(String, String)` or `createCustomReport(String, String, Closure)` to create a custom report. The report gets created after the closure gets closed.

**reportName** Sets the report name

**location** Sets the location (e.g. D:\Report\report.html).

Use `createCustomReport(String, String, Closure)` to create a custom report. The report gets created after the closure gets closed.

**reportName** Sets the report name

**location** Sets the location

Use `createCustomReport(String, Closure)` to create a custom report with the default location. The default location is the LOG folder. The report gets created after the closure gets closed.

**reportName** Sets the report name

**ICreateCustomReportApi** The API allows to create the custom report tables. It is possible to add multiple tables on a report.

Use `getAddTable(String)` or `addTable(String, Closure)` to add a new a custom report table.

**tableName** Sets the table name

**IConfigureTableApi** The API allows to configure a custom report table.

Use `columns(String...)` to create the column headers.

**cols** Set the column headers.

**ITableApi** The API allows to setup a custom report table.

Use `getConfigure()` to configure the report columns.

Use `configure(Closure)` to configure the report columns.

Use `addRow(String...)` to print the content in a table row.

NOTE: The given parameters have to match the table columns.

## 4.13 Persistency

The persistency API provides methods which allow to import and export model data from and to files. The files are normally in the AUTOSAR .arxml format.

### 4.13.1 Model Export

The `modelExport` allows to export MDF model data into .arxml files.

To access the export functionality use one of the `getModelExport()` or `modelExport(Closure)` methods.

```
// You can access the API in every active project
def exportApi = persistency.modelExport

//Or you use a closure
persistency.modelExport {
```

Listing 4.323: Accessing the model export persistency API

#### 4.13.1.1 Export ActiveEcuc

The method `exportActiveEcucToFile(Object)` exports the whole ActiveEcuC configuration into a single file of type Path specified by the user.

```
scriptTask('taskName') {
    code {
        def destinationFile // Define the file to export into...
        persistency.modelExport.exportActiveEcucToFile(destinationFile)
    }
}
```

Listing 4.324: Export the ActiveEcuc to a file

The method `exportActiveEcuc(Object)` exports the whole ActiveEcuC configuration into a single file of type Path.

```
scriptTask('taskName') {
    code {
        def tempExportFolder = paths.resolveTempPath(".")
        def resultFile = persistency.modelExport.exportActiveEcuc(
            tempExportFolder)
    }
}
```

Listing 4.325: Export the ActiveEcuc into a folder

#### 4.13.1.2 Export PostBuild Variants (Post-build selectable)

The method `exportPostBuildVariants(Object)` exports the PostBuild variants info. This will export the ActiveEcuc and miscellaneous data. The ActiveEcuC is exported into one file (even for split DPA-projects) per variant into <project-name>.<variant-name>.ecuc.arxml.

Miscellaneous data is exported into one file per variant. The files contain all data of the project except:

- ModuleConfigurations, ModuleDefinitions
- BswImplementations, EcuConfigurations
- Variant information like EvaluatedVariantSet

The created files are `<project-name>.<variant-name>.misc.arxml`.

The method returns a `List<Path>` of exported files.

```
scriptTask('taskName') {
    code {
        persistency.modelExport {
            def tempExportFolder = paths.resolveTempPath(".")
            def fileList = exportPostBuildVariants(tempExportFolder)
        }
    }
}
```

Listing 4.326: Export a PostBuild project into files per predefined variant

#### 4.13.1.3 Export PreBuild Variants

The method `exportPreBuildVariants(Object)` exports the PreBuild variants info. This will export the ActiveEcuc and miscellaneous data. The ActiveEcuc is exported into one file (even for split DPA-projects) per variant into `<project-name>.<variant-name>.ecuc.arxml`.

Miscellaneous data is exported into one file per variant. The files contain all data of the project except:

- ModuleConfigurations, ModuleDefinitions
- BswImplementations, EcuConfigurations

The created files are `<project-name>.<variant-name>.misc.arxml`.

The method returns a `List<Path>` of exported files.

```
scriptTask('taskName') {
    code {
        persistency.modelExport {
            def tempExportFolder = paths.resolveTempPath(".")
            def fileList = exportPreBuildVariants(tempExportFolder);
        }
    }
}
```

Listing 4.327: Export a PreBuild project into files per predefined variant

#### 4.13.1.4 Export Module Configuration

```
scriptTask('taskName') {
    code {

        Path location = paths.resolveScriptPath(".")
        def moduleList = mdfModel("EcuC")
        MIModuleConfiguration ecuC = moduleList.getFirst()
        persistency.modelExport.exportModelTree(location, ecuC)

    }
}
```

Listing 4.328: Exports a module configuration

#### 4.13.1.5 Advanced Exports

The advanced export use case provides access to multiple `IModelExporter` for special export use cases like export the system description for the RTE.

Normally you would retrieve an `IModelExporter` by its ID via `getExporter(String)`. On this exporter you can call:

- `IModelExporter.export(Object)` to export the model
- `IModelExporter.exportAsPostBuildVariants(Object)` to export the model divided into files per PostBuild predefined variant.

You can retrieve a list of supported exporters from `getAvailableExporter()`. The list can differ from data loaded in your project.

```
scriptTask('taskName') {
    code {
        def tempExportFolder = paths.resolveTempPath(".")

        // Export with an exporter in one line
        persistency.modelExport["activeEcuc"].export(tempExportFolder)
    }
}
```

Listing 4.329: Export the project with an exporter into a folder

```
scriptTask('taskName') {
    code {
        def tempExportFolder = paths.resolveTempPath(".")

        def fileList
        //Switch to the persistency export API
        persistency.modelExport{
            // The getAvailableExporter() returns all exporters in the system
            def exporterList = getAvailableExporter()

            // Select an exporter by its ID
            def exporterOpt = getExporter("activeEcuc")

            exporterOpt.ifPresent { exporter ->
                // Export into folder, when exporter exists
                fileList = exporter.export(tempExportFolder)
            }
        }
    }
}
```

Listing 4.330: Export the project with an exporter and checks

**Export an Model Tree** The method `exportModelTreeToFile(Object, MIOBJECT)` exports the specified model object and the subtree into a single file of type `Path` specified by the user.

```
scriptTask('taskName') {
    code {
        def destinationFile // Define the file to export into...
        MIARPackage autosarPkg = mdfModel(AsrPath.create("/MICROSAR"))

        persistency.modelExport{
            exportModelTreeToFile(destinationFile, autosarPkg)
        }
    }
}
```

Listing 4.331: Export an AUTOSAR package into a file

The method `exportModelTree(Object, MIOBJECT)` exports the specified model object and the subtree into a single file of type `Path`.

```
scriptTask('taskName') {
    code {
        def exportFolder = paths.resolveTempPath(".")

        MIARPackage autosarPkg = mdfModel(AsrPath.create("/MICROSAR"))

        def resultFile = persistency.modelExport.exportModelTree(exportFolder,
            autosarPkg)
    }
}
```

Listing 4.332: Export an AUTOSAR package into a folder

**Export an Model Tree including all referenced Elements** You could also export model trees including all referenced elements with the exporter `modelTreeClosure`:

```
scriptTask('taskName') {
    code {
        def exportFolder = paths.resolveTempPath(".")
        MIARPackage microsarPkg = mdfModel(AsrPath.create("/MICROSAR"))
        MIARPackage autosarPkg = mdfModel(AsrPath.create("/AUTOSAR"))

        persistency.modelExport["modelTreeClosure"].export(exportFolder,
            autosarPkg, microsarPkg)
    }
}
```

Listing 4.333: Exports two elements and all references elements

**Usage of Exporter Arguments** You can use `withExporterArgs(Map, Closure)` to specify exporter arguments like in the command line with `-exporterArgs` argument. The key is the exporter ID, the value are the arguments to the exporter. See command line help for details.

```
persistency.modelExport{
    // Specify the arguments with exporterId: "arguments"
    withExporterArgs(modelTree: "--element /MICROSAR"){
        // Call any export code with the active arguments.
        getExporter("modelTree").get().exportToFile(destinationFile)
    }
}
```

Listing 4.334: Use exporter arguments like in the commandline

### 4.13.2 Model Import

To access the import functionality use one of the `getModelImport()` or `modelImport(Closure)` methods.

```
// You can access the API in every active project
def importApi = persistency.modelImport

//Or you use a closure
persistency.modelImport {
```

Listing 4.335: Accessing the model import persistency API

#### 4.13.2.1 Module Configuration Import

To access the module import functionality use one of the `importModuleConfigurations(List)` methods.

```
// You can access the API inside the closure

persistency.modelImport {
    importModuleConfigurations(importFile)
}
```

Listing 4.336: Accessing the import module configuration persistency API

The method `importModuleConfigurations(Path, Closure)` imports `MIModuleConfiguration` from the specified `.arxml` file into the current ActiveEcuC. The Closure can be used to specify the import mode and filter if necessary.

The method `importModuleConfigurations(Path)` imports `MIModuleConfiguration` from the specified `.arxml` file into the current ActiveEcuC.

The method `importModuleConfigurations(List)` imports `MIModuleConfiguration` from the specified `.arxml` files into the current ActiveEcuC.

The method `importModuleConfigurations(List, Closure)` imports `MIModuleConfiguration` from the specified `.arxml` files into the current ActiveEcuC. The Closure can be used to specify the import mode and filter if necessary.

Note: If a module configuration is contained in several files a `PersistencyImportException` is thrown.

#### 4.13.2.2 Specify import mode and module filter

To specify an import mode use the method `addToModel(Closure)` to add the imported modules to the ActiveEcuC or use the method `replaceInModel(Closure)` (*this is the default mode*) to replace already existing module configurations with the imported one.

To specify a filter for the module configurations to import use one of the methods:

- `module(DefRef)` Use a single `DefRef`
- `module(List)` Use a list of `AsrPath`, `DefRef` or Definition Refs as `String`
- `module(AsrPath)` Use a single `AsrPath`
- `module(String)` Use a single Definition Ref as `String`

```
// You can access the API inside the closure

def importFile = paths.resolvePath("./ImportFile.arxml")

persistency.modelImport {

    importModuleConfigurations(importFile) {

        addToModel(){

            // Use 'addToModel()' to add the imported module configuration to the
            // current ActiveEcuC. If a module configuration with the same name
            // already exists, the module configuration is ignored.

            module("/MICROSAR/LinIf") // -> DefRef as String
            def linNmAsrPath = AsrPath.create("/ActiveEcuC/LinNm")
            module(linNmAsrPath) // -> Autosar path as AsrPath instance

        }

        replaceInModel(){

            // Use 'replaceInModel()' to replace already existing module
            // configurations in the current ActiveEcuC.

            List<String> modulesToImport = Arrays.asList("/MICROSAR/LinSM")

            module(modulesToImport)

        }
    }
}
```

Listing 4.337: Specify the module configuration import mode and filter

## 4.14 Project Comparison

The Automation Interface provides an API for the comparison of DaVinci projects.

### 4.14.1 Automatic merge

The project comparison provides an API for the automatic merge of 3 DaVinci projects.

These projects are described in the following by these names:

- **MINE** identifies the currently loaded project. This is the project which receives the changes
- **OTHER** identifies the project which is compared with **MINE**
- **BASE** is the base of the projects **MINE** and **OTHER**

#### 4.14.1.1 Accessing the API

**IAutoMergeApi** is the entry point for the automatic merge API. It offers interfaces for the configuration and execution of the whole workflow.

```
activeProject {
    projectCompare {
        autoMerge {
            configure {
                projectOther = pathToOther
                projectBase = pathToBase

                comparisonScope = ECUC_ECUEX_ONLY

                // We only want to merge the 'EcuC' module configuration
                moduleToMerge("EcuC")
            }
        }
    }
}
```

Listing 4.338: Accessing the automatic merge API

### 4.14.2 Configure automatic merge

To configure the automatic merge related settings the **IConfigureApi** is used. It allows to make detailed settings for the comparison. Also it allows to set the conflict resolution strategy when resolving the differences.

#### 4.14.2.1 The settings in detail

The path to **MINE**, **OTHER** and **BASE** projects are mandatory. They have no default value.

**OTHER** `setProjectOther(Path)` is used to set the absolute path to project OTHER which is used for comparison with project MINE.

**BASE** `setProjectBase(Path)` is used to set the absolute path to project BASE which is used for comparison and to determine how a possible difference can be auto resolved.

**UUID object identification** `setDisableUuidsForObjectIdentification(Path)` disables or enables the UUID usage for the object identification during comparison. In case of `true` the UUID usage will be disabled (this is the default behavior). If `false` the UUID will be used for object identification.

**Project Settings** `setCompareProjectSettings(boolean)` activates or disables the comparison of project settings. `true` when project settings should be compared. Otherwise `false`.

**Module selection for the automatic merge** The default selection behavior of the modules to be considered is as follows.

(1) A module configuration contained in MINE **and** OTHER is compared regardless of whether it is also included in BASE.

(2) A module configuration which has been added in OTHER is imported.

However the selection can be restricted (only certain modules are to be considered) using the following method.

**Module configurations to merge** `moduleToMerge(String)` adds the module configuration identified by the given short name to the collection of the modules to be considered for the merge. This will restrict the default selection behavior.

#### 4.14.2.2 Global conflict resolution

In case a difference can't be resolved automatically a global conflict resolution approach can be specified. The following values are supported:

**DONT\_SOLVE** describes a global conflict resolution approach where no conflicted differences are resolved.

**USE\_MINE** describes a global conflict resolution approach where conflicted differences are resolved by using the value of project MINE.

**USE\_OTHER** describes a global conflict resolution approach where conflicted differences are resolved by using the value of project OTHER.

**Conflict resolution approach** `setConflictResolutionApproach(EConflictResolutionApproach)` is used to set the global conflict resolution approach.

#### 4.14.2.3 Automatic merge result evaluation

The individual automatic merge steps allow to evaluate the result and, if necessary, cancel the process. For this reason, there are two entry points:

**DaVinci Developer merge result** `evaluateMergeResultDaVinciDeveloper(Predicate)` specifies a `Predicate` to evaluate the merge result of the DaVinci Developer.

**DaVinci Configurator merge result** `evaluateMergeResultDaVinciConfigurator(Predicate)` specifies a `Predicate` to evaluate the merge result of the DaVinci Configurator.

For a detailed API reference see chapter 4.14.2.5 on the following page.

#### 4.14.2.4 Comparison scope

The automatic merge supports several scopes for merging the projects.

**ECUC\_ECUEX\_ONLY** describes the scope in which only ECUC and Extract of System Description relevant data is compared (no comparison of DaVinci Developer Workspace).

**ECUC\_ECUEX\_DEV\_WS** describes the scope in which ECUC, Extract of System Description and DaVinci Developer Workspace relevant data is compared.

**Comparison Scope** `setComparisonScope(EComparisonScope)` sets the comparison scope to use.

#### 4.14.2.5 Platform function development

In the case of the project comparison, the platform functions serve as filters for correspondingly annotated elements. To apply such a filter the definition of the function to use must already be part of project **MINE**. This import is also done during the automatic merge workflow. Please note that this import only has to be done once.

Once the definition has become a fixed part of the configuration, the function can also be declared as a filter.

**Importing a platform function definition** The method `importPlatformFunctionDefinition(String)` is used to add the platform function definition identified by the given name to the definitions which will be imported during the auto-merge process.

```
activeProject {
    projectCompare.autoMerge {
        configure {
            projectOther = pathToOther
            projectBase = pathToBase

            comparisonScope = ECUC_ECUEX_ONLY

            // Imports the definition of platform function 'FuncA' which is part
            // of project OTHER
            importPlatformFunctionDefinition("FuncA")
        }
    }
}
```

Listing 4.339: Importing a platform function definition

**Filter for a platform function** The method `filterPlatformFunction(String)` is used to extend the platform function filter by the given name of the function.

```
activeProject {
    projectCompare.autoMerge {
        configure {
            projectOther = pathToOther
            projectBase = pathToBase

            moduleToMerge("EcuC")

            comparisonScope = ECUC_ECUEX_ONLY

            // Activates the platform filter for function 'FuncA'
            filterPlatformFunction("FuncA")
        }
    }
}
```

Listing 4.340: Filter for a platform function

**Example for the first integration of a platform function into project MINE** The following example shows the first integration of a platform function.

```
activeProject {
    projectCompare.autoMerge {
        configure {
            projectOther = pathToOther
            projectBase = pathToBase

            moduleToMerge("EcuC")

            // Import the definition of platform function (only has to be done
            // once)
            importPlatformFunctionDefinition("FuncA")

            // Activate platform function filter
            filterPlatformFunction("FuncA")

            comparisonScope = ECUC_ECUEX_ONLY
        }
    }
}
```

Listing 4.341: Integrating a platform function

#### 4.14.3 Automatic merge result

The DaVinci Configurator and Developer allow the respective result of the merge to be examined more closely by a corresponding API.

`IMergeResult` is the entry point for the automatic merge result to retrieve the not resolved differences and the path to the report file. **Note** that the structure of the report is not stable and may change when another version of the DaVinci Configurator is used. This API instead is kept stable.

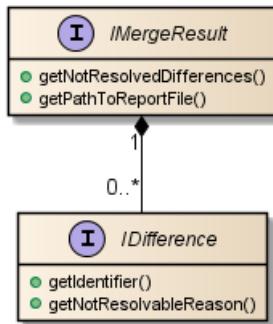


Figure 4.11: The structure of a merge result

The method `getNotResolvedDifferences()` is used to get all differences which haven't been resolved during the automatic merge workflow.

The method `getPathToReportFile()` is used to get the Path to the result file of the automatic merge workflow.

`IDifference` represents a difference within the `IMergeResult` API.

The method `getIdentifier()` is used to get the identifier of the CE represented by this `IDifference`.

The method `getNotResolvableReason()` is used to get the reason (if available) why the `IDifference` is not resolvable. If none is available `null` is returned.

#### 4.14.3.1 Example of result evaluation

The following shows how the merge result can be evaluated.

```

activeProject {
    projectCompare {
        autoMerge {
            configure {
                conflictResolutionApproach = DONT_SOLVE
                projectOther = pathToOther
                projectBase = pathToBase

                moduleToMerge("EcuC")

                comparisonScope = ECUC_ECUEX_ONLY

                evaluateMergeResultDaVinciConfigurator{
                    def notResolvableDiffs = getNotResolvedDifferences()
                    // Cancel if something is left
                    return !(notResolvableDiffs.size() > 0)
                }
            }
        }
    }
}

```

Listing 4.342: Evaluating the automatic merge result (cancel if something is left)

```
activeProject {  
    projectCompare {  
        autoMerge {  
            configure {  
                conflictResolutionApproach = DONT_SOLVE  
                projectOther = pathToOther  
                projectBase = pathToBase  
  
                moduleToMerge("EcuC")  
  
                comparisonScope = ECUC_ECUEX_ONLY  
  
                evaluateMergeResultDaVinciConfigurator{  
  
                    // Cancel if PduLength couldn't be merged  
                    def notResolvableDiffs = getNotResolvedDifferences()  
                    return (notResolvableDiffs.size() == 1 && notResolvableDiffs.  
                        get(0).getIdentifier() == "/ActiveEcuC/EcuC/  
                        EcucPduCollection/PduB[0:PduLength]")  
                }  
            }  
        }  
    }  
}
```

Listing 4.343: Evaluating the automatic merge result (abort at a certain identifier)

## 4.15 Utilities

### 4.15.1 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\_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).

#### 4.15.2 Converters

General purpose converters (`java.util.Functions`) for performing value conversions throughout the automation interface are provided. These converters are referenced from the AutomationInterface documentation wherever they apply. The AutomationInterface is typed strongly. In some cases, however, e.g. when specifying file locations, it is desirable to allow for a range of possibly parameter types. This is achieved by accepting parameters of type `Object` and converting the given parameters to the desired type.

The following converters are provided:

**ScriptConverters.TO\_PATH** Attempts to convert arbitrary `Objects` to `Paths` using `IAutomationPathsApi.resolvePath(Object)` 4.4.3.2 on page 42.

**ScriptConverters.TO\_SCRIPT\_PATH** Attempts to convert arbitrary `Objects` to `Paths` using `IAutomationPathsApi.resolveScriptPath(Object)` 4.4.3.3 on page 43.

**ScriptConverters.TO\_VERSION** Attempts to convert arbitrary `Objects` to `IVersions`. The following conversions are implemented:

- For `null` or `IVersion` arguments the given argument is returned. No conversion is applied.
- Strings are converted using `Version.valueOf(String)`.
- Numbers are converted by converting the `int` obtained from `Number.intValue()` using `Version.valueOf(int)`.
- All other `Objects` are converted by converting the `String` obtained from `Object.toString()`.

**ScriptConverters.TO\_BIG\_INTEGER** Attempts to convert arbitrary `Objects` to `BigIntegers`. The following conversions are implemented:

- For `null` or `BigInteger` arguments the given argument is returned. No conversion is applied.
- `Integers, Longs, Shorts and Bytes` are converted using `BigInteger.valueOf(long)`.
- All other types of objects are interpreted as `Strings` (`Object.toString()`) and passed to `BigInteger.BigInteger(String)`.

**ScriptConverters.TO\_BIG\_DECIMAL** Attempts to convert arbitrary Objects to BigDecimals. The following conversions are implemented:

- For null or BigDecimal arguments the given argument is returned. No conversion is applied.
- Floats and Doubles, are converted using BigDecimal.valueOf(double).
- Integers, Longs, Shorts and Bytes are converted using BigDecimal.valueOf(long).
- All other types of objects are interpreted as Strings (Object.toString()) and passed to BigDecimal.BigDecimal(String).

**ModelConverters.TO\_MDF** Attempts to convert arbitrary Objects to MDFObjects. The following conversions are implemented:

- For null or MDFObject arguments the given argument is returned. No conversion is applied.
- IHasModelObjects are converted using their getMdfModelElement() method.
- IViewedModelObjects are converted using their getMdfObject() method.
- For all other Objects ClassCastException are thrown.

For thrown Exceptions see the used functions described above.

## 4.16 Advanced Topics

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

### 4.16.1 Java Development

It is also possible to write automation scripts in plain Java code, but this is not recommended. There are some items in the API, which need a different usage in Java code.

This chapter describes the differences in the Automation API when used from Java code.

#### 4.16.1.1 Script Task Creation in Java Code

Java code could not use the Groovy syntax to provide script tasks. So another way is needed for this. The `IScriptFactory` interface provides the entry point that Java code could provide script tasks. The `createScript(IScriptCreationApi)` method is called when the script is loaded.

This interface is **not** necessary for Groovy clients.

```
public class MyScriptFactoryAsJavaCode implements IScriptFactory {
    @Override
    public void createScript(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 4.344: 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.

#### 4.16.1.2 Java Code accessing Groovy API

Most of the Automation API is usable from both languages Java and Groovy, but some methods are written for Groovy clients. To use it from Java you have to write some glue code.

Differences are:

- Accessing Properties
- Using API entry points.
- Creating Closures

**Accessing Properties** Properties are not supported by Java so you have to use the getter/setter methods instead.

**API Entry Points** Most of the Automation API is added to the object by so called DynamicObjects. This is not available in Java, so you have to call `IScriptExecutionContext.getInstance(Class)` instead. So if you want to access The IWorkflowApi you have to write:

```
// Java code:
IScriptExecutionContext scriptCtx = ...;
IWorkflowApi workflow = scriptCtx.getInstance(IWorkflowApiEntryPoint.class).
    getWorkflow()

// Instead of Groovy code:
workflow{
}
```

Listing 4.345: 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 4.346: 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<sup>1</sup>](http://melix.github.io/blog/2010/04/19/coding_a_groovy_closure_in.html)

#### 4.16.1.3 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 368.

---

<sup>1</sup>Last accessed 2016-05-24

## 4.16.2 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 (see chapter 4.17 on page 318 for details)
- Validations
- Workflow rules
- ...

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

### 4.16.2.1 JUnit4 Integration

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

For this you have to add a `JUnit4` dependency in your `build.gradle` file:

```
dependencies{
    compileOnly("junit:junit:4.12")
}
```

Listing 4.347: Additional JUnit4 dependency in Gradle

**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 4.348: Run all JUnit tests from one class

You can access the Automation API with the `ScriptApi` class. See chapter 4.4.8 on page 52 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 4.349: Run all JUnit tests using a Suite

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

#### 4.16.2.2 Execution of Spock Tests

The AutomationInterface can also execute Spock tests. Please note that for Spock tests, you need a Automation Scripting Project and simple Script File is not sufficient.

See:

- Homepage: <https://github.com/spockframework/spock><sup>2</sup>
- Documentation: <http://spockframework.org/spock/docs/1.3/index.html><sup>3</sup>

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 4.350: Run unit test with the Spock framework

You can access the Automation API with the `ScriptApi` class. See chapter 4.4.8 on page 52 for details.

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

---

<sup>2</sup>Last accessed 2019-03-25

<sup>3</sup>Last accessed 2019-03-25

```
dependencies {
    compileOnly("org.spockframework:spock-core:1.3-groovy-2.5") {
        exclude group: 'org.codehaus.groovy', module: 'groovy-all'
    }
}
```

Listing 4.351: Additional Spock dependency in Gradle

Note: after the change you have to call Gradle to update the IntelliJ IDEA project.

```
gradlew idea
```

#### 4.16.2.3 Registration of Unit Tests in Scripts

A test or the root suite class has to be registered in a script to be executable. The first argument is the `taskName` for the `UnitTests` the second is the class of the tests.

```
// You can add a unit test inside a script
unitTestTask("MyUnitTest", AllMyScriptJUnitTests.class)
```

Listing 4.352: Add a UnitTest task with name MyUnitTest

It is also possible to reference the test/suite class directly as a script inside of a script project. So you don't have to create a script as a wrapper.

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

Listing 4.353: The projectConfig.gradle file content for unit tests

#### 4.16.2.4 Model TestInfrastructure

The `ITransactionUndoAllRule` class provides a `JUnit TestRule`, which will undo all transaction after the `JUnit` test was executed. This can be used as a `ClassRule` or `Rule` to undo only for the whole test class or for each test case. This could also be used with `Spock`.

```
public class TransactionUndoAllRuleTest {
    @Rule
    public ITransactionUndoAllRule rule = ITransactionUndoAllRule.
        forActiveProject();

    @Test
    public void testcase() {
        //After this test case all transactions are reverted by the
        TransactionUndoAllRule
    }
}
```

Listing 4.354: Usage of the `ITransactionUndoAllRule` in a `JUnit` test

#### 4.16.2.5 Automation Project TestInfrastructure

The `IProjectLoadRule` class provides a `JUnit TestRule`, which will load the passed `IProjectRef` before test execution and close it afterwards. This can be used as a `ClassRule` or `Rule` and it could also be used with `Spock`.

```
public class ProjectLoadRuleTest {  
    @ClassRule  
    static IProjectLoadRule rule = IProjectLoadRule.forProject(ScriptApi.  
        scriptCode.projects.parameterizeProjectLoad{  
            dpaFile "YourProject.dpa"  
        })  
  
    @Test  
    public void testcase() {  
        //In this test the project was loaded and will be closed after the test  
    }  
}
```

Listing 4.355: Usage of the `IProjectLoadRule` in a `JUnit` test

For details about the how to load a project, see chapter 4.5.6 on page 76. Or if you want to load only an `.arxml` file, see chapter 4.5.10 on page 82.

## 4.17 Generator Testing

You can test Generators with an Automation Interface ScriptProject. There are two categories of generator tests, which can be performed:

- Black-box tests
- White-box tests

Both test categories are achieved by unit tests written in the ScriptProject, see chapter 4.16.2 on page 314 for how to write unit tests with the Automation Interface.

**Black-box tests** The Black-box tests test the functionality without knowing the internal structure of the generator. You can achieve this by using the Code Generation (see chapter 4.7.1 on page 127)) and verify that the generated code is correct in your unit test.

### 4.17.1 White-box tests

The white-box tests test the functionality with knowledge about the internal structure of the generator. This allows you to test single classes and internal algorithms especially error cases of the generator.

You need the license option .MD for the white-box tests development and execution.

This is not available for script files, you need a script project.

**Technical description:** It will allow you to access and use all classes of the Generator including package-private classes and methods. The generator is loaded into your script during unit test execution.

#### 4.17.1.1 Step-by-step

You need to do the following steps:

1. Add the generator to your script project as compile dependency, see 4.17.1.2
2. Create your unit tests for your generator, see 4.16.2 on page 314
  - If you need a GeneratorPackage, see 4.17.1.3 on the next page
  - If you use transactions, see 4.16.2.4 on page 316
  - If you want to load projects, see 4.16.2.5 on the previous page
3. Create a TestSuite, which references your unit test(s), see 4.16.2.1 on page 314. This is mandatory for generator white-box tests.
4. Execute your tests inside of a running DaVinci Configurator

#### 4.17.1.2 Compilation

To enable the access to the generator classes you have to specify the generator under test in your `build.gradle` file of your script project.

The `generatorTests` allows to configure generators under test. You can add multiple generators with `generator(<PATH-TO-GENERATOR-JAR-FILE>)`.

```
dvCfgAutomation {
    generatorTests{
        generator("<PATH-TO-GENERATOR-JAR-FILE>")
    }
}
```

Listing 4.356: DaVinci Configurator build Gradle DSL API - generatorTests

The specified generators are automatically added as compile dependency to your script project.

**Note:** You have to update the IntelliJ IDEA project, see 7.9.3.2 on page 378.

If you want to access GenDevKit classes or you reference GenDevKit classes indirectly, you have to specify a GenDevKit version:

```
dvCfgAutomation {
    generatorTests{
        generator("<PATH-TO-GENERATOR>")
        genDevKit("<VERSION>") //E.g. 21.00.10
    }
}
```

Listing 4.357: DaVinci Configurator build Gradle DSL API - generatorTests - genDevKit

The `genDevKit` method accepts the version as `String` or as path to the GenDevKit folder.

#### 4.17.1.3 GeneratorTestingApi

The `IGeneratorTestingApi` provides methods to test code with white box tests which require a real `IGeneratorPackage` or an `IGeneratorResultSink` etc. of the generator under test.

You can use the `IGeneratorTestingApi` with the call: `ScriptApi.scriptCode.genTesting`

The `createGenerator(Class)` method creates an `IGeneratorTestingAccess` for the passed class of the `IGeneratorPackage` under test. The method will use the currently active project for the generator. This will always create a new generator instance.

You can specify the used project with the method `createDeployablePackage(IProjectContext, Class)`.

The `withGenerator(Class, Closure)` method creates an `IGeneratorTestingAccess` and automatically activates it inside of the `Closure` code block.

**IGeneratorTestingAccess** The `IGeneratorTestingAccess` holds one created `IGeneratorPackage` under test and provides the following accessors:

- `getGeneratorPackage()`
- `getGenerationProcessor()`
- `getResultSink()`

The `IGeneratorPackage` needs to be activated before it is used in the test case. The best way to do it is with an JUnit `TestRule`, see listing below. You could also use the `with(Closure)` method. Or you do it manually with `activate()` and `deactivate()`, but you have to use a `try/finally` construct.

**Examples** The following sample use the API to execute a Spock test:

```
class SpockGeneratorTest extends Specification {
    @ClassRule
    @Shared
    IGeneratorTestingAccess<TestGeneratorPackage> genUnderTest = ScriptApi.
        scriptCode.
    //We test the TestGeneratorPackage class
    genTesting.createGenerator(TestGeneratorPackage)

    def "GeneratorName" (){
        when:
        def gen = genUnderTest.generatorPackage

        then:
        gen.getGeneratorName() == "TestGenerator"
    }
}
```

Listing 4.358: Spock test using a GeneratorPackage

The following sample use the API to execute a JUnit test:

```
public class JunitGeneratorTest {
    @ClassRule
    public static IGeneratorTestingAccess<TestGeneratorPackage> genUnderTest =
        ScriptApi.scriptCode.
    //We test the TestGeneratorPackage class
    genTesting.createGenerator(TestGeneratorPackage)

    @Test
    public void testGeneratorName(){
        def gen = genUnderTest.generatorPackage
        assert gen.getGeneratorName() == "TestGenerator"
    }
}
```

Listing 4.359: JUnit test using a GeneratorPackage

#### 4.17.1.4 Test OuputFile Generation

You can also test the generation of `IOutputFile` classes with the `IGeneratorTestingApi`.

**Examples** The following sample use the API to execute a Spock test:

```
def "Content of TestOutputFile" (){
    when: "Generate the output file"
    def gen = genUnderTest.generatorPackage

    def dataRep = new TestDataRep(gen)
    def content = new TestOutputFile(gen, dataRep).generateFileContent()

    then:
    content == "TestFileContent" + NL
}
```

Listing 4.360: Spock test case which generates an OutputFile and verifies the content

The following sample use the API to execute a JUnit test:

```

@Test
public void testContentOfTestOutputFile(){
    def gen = genUnderTest.generatorPackage

    def dataRep = new TestDataRep(gen)
    def content = new TestOutputFile(gen, dataRep).generateFileContent()

    assert content == "TestFileContent" + NL
}

```

Listing 4.361: JUnit test case which generates an OutputFile and verifies the content

#### 4.17.1.5 Test GeneratorResults and ValidationResults

The `IGeneratorTestingAccess` provides a special `IGenTestingGeneratorResultSink` for testing purpose. You can retrieve it with `IGeneratorTestingAccess.getResultSink()`. The `IGenTestingGeneratorResultSink` can be used to call code under test which requires an `IGeneratorResultSink`.

The `IGenTestingGeneratorResultSink` provides access to assert that a certain code has created the expected `IValidationResult` objects. You can use the access API for `IValidationResultUI` objects provided by the validation , see chapter 4.8.2 on page 139 for details.

**Examples** The following sample use the result sink to execute a Spock test:

```

def "DataRep creates Error ValidationResult" (){
    when: "Execute the code under test"
    def gen = genUnderTest.generatorPackage
    def dataRep = new TestDataRep(gen)
    dataRep.createData()

    then: "Retrieve the validation result created by the code under test"
    def results = genUnderTest.resultSink.
        validationResults.filter{ it.isId("TestGen", 123) }

    results.size() == 1
    results.first.description.toString() == "DescriptionText"
}

```

Listing 4.362: Spock test case which verifies a ValidationResult

The following sample use the result sink to execute a JUnit test:

```
@Test
public void dataRepCreatesErrorValidationResult(){
    def gen = genUnderTest.generatorPackage
    //Execute the code under test
    def dataRep = new TestDataRep(gen)
    dataRep.createData()

    //Retrieve the validation result created by the code under test
    def results = genUnderTest.resultSink.
        validationResults.filter{ it.isId("TestGen", 123) }

    assert results.size() == 1
    assert results.first.description.toString() == "DescriptionText"
}
```

Listing 4.363: JUnit test case which verifies a ValidationResult

# 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 84. 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 `AR-Package` (AUTOSAR package in the top-level structure of the meta-model) is `MIARPackage`. The AUTOSAR meta model can be found for example on the AUTOSAR website.

### 5.1.2 The models inheritance hierarchy

The MDF model therefore implements (nearly) the same inheritance hierarchy and associations as defined by the AUTOSAR model. These interfaces provide access to the data stored in the model.

See figure 5.1 on the following 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 `MIPParameterValue` therefore has the same base type
- All `MIIdentifiables` can hold ADMIN-DATA and ANNOTATIONS
- All MDF objects in the AUTOSAR model tree inherit from `MIOBJECT` which is again an `MIOBJECT`

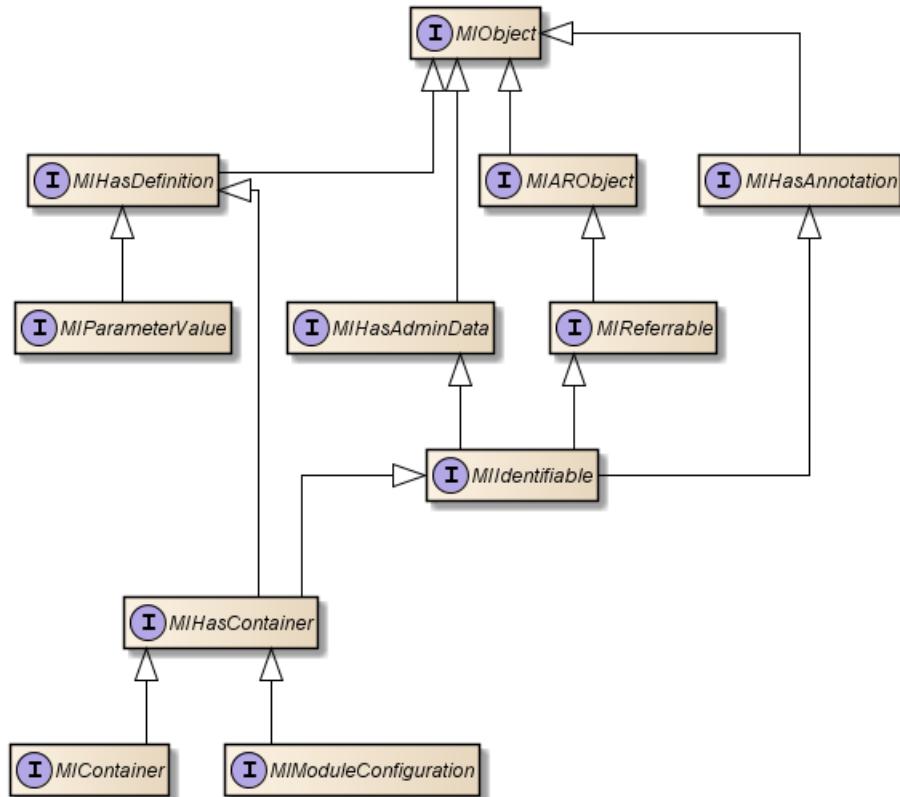


Figure 5.1: ECUC container type inheritance

### 5.1.2.1 MIOObject and MDFOObject

The `MIOObject` is the base interface for all AUTOSAR model objects in the DaVinci Configurator data model. It extends `MDFOObject` which is the base interface of all model objects. Your client code shall always use `MIOObject`, when AUTOSAR model objects are used, instead of `MDFOObject`.

The figure 5.2 on the next page describes the class hierarchy of the `MIOObject`.

### 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 following page shows a simple example of an MDF object containment hierarchy. This example contains two AUTOSAR packages with module configurations below.

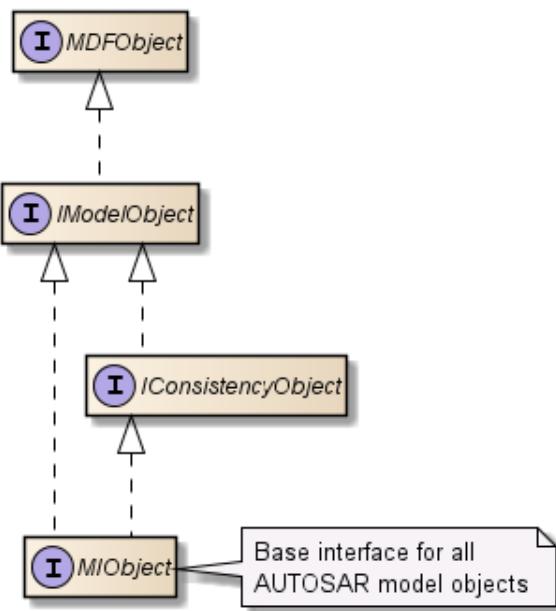


Figure 5.2: MIOBJECT class hierarchy and base interfaces

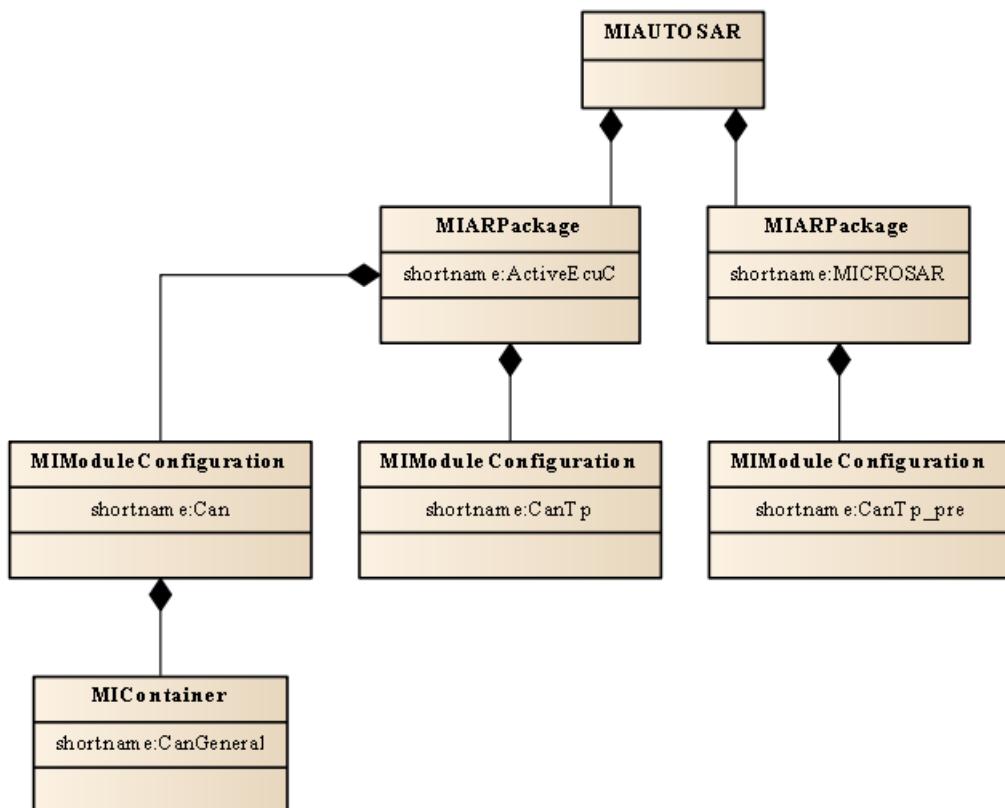


Figure 5.3: Autosar package containment

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 `MIPParameterValue` 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 ARXML 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 shows this type hierarchy for the definition reference of an ECUC container.

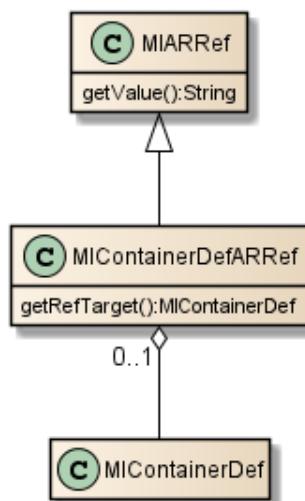


Figure 5.4: The ECUC container definition reference

In ARXML, such a reference can be specified as:

```
<DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">
  /MICROSAR/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

## 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 you 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<IPostBuildPredefinedVariantView> variants = viewMgr.
    getAllPostBuildVariantViews();
```

Listing 5.2: Get all available variants

```
final List<IPostBuildView> allVariantsOrInvariantView = viewMgr.
    getAllPostBuildVariantViewsOrInvariant();
```

Listing 5.3: Get all available variants or if project does not contain variants the invariant view

```
try (final IMModelViewExecutionContext 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 5.4: Execute code with variant visibility

**Important remark:** It is essential that the `execute...()` methods are used exactly as implemented in the listing above. The `try (...){...}` construct is a new Java 7 feature which guarantees that resources are closed whenever (and how ever) the try block is being left. For details read:

<http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html>

The try-with-resources feature introduced in Java 7 will be featured with Groovy Version 3.0 chapter ARM Try with resources <http://groovy-lang.org/releasenotes/groovy-3.0.html>

Or use constructs like this:

```

InputStream is = null
try {
    is = new InputStream();
} catch() { ... } finally {
    if (is != null) {
        try {
            is.close();
        } catch () {
            // Nothing
        }
    }
} // finally

```

```

Collection<IPostBuildPredefinedVariantView> visibleVariants = viewMgr.
    getVisiblePostBuildVariantViews(t.paramInvariant);
assertThat(visibleVariants.size(), equalTo(2));
assertThat(visibleVariants, containsInAnyOrder(t.variantViewA, t.variantViewB))
;

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

```

Listing 5.5: 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.getPostBuildVariantSiblings()` provides access to these sibling objects:

This method returns MDF object instances representing the same object but in all variants. The collection returned contains the object itself including all siblings from other PostBuild variants.

The calculation of siblings depends on the object-type as follows:

- **Ecuc Module Configuration:**

Since module configurations are never variant, this method always returns a collection which contains the specified object only

- **Ecuc Container:**

For siblings of a container all of the following conditions apply:

- They have the same AUTOSAR path
- They have the same definition path (containers with the same AUTOSAR path but different definitions may occur in variant models - but they are not variant siblings because they differ in type)

- **Ecuc Parameter:**

For siblings of a parameter all of the following conditions apply:

- The parent containers have the same AUTOSAR path
- The parameter siblings have the same definition path

The parameter values are **not** relevant so parameter siblings may have different values. Multi-instance parameters are special. In this case the method returns all multi-instance siblings of all variants.

- **System description object:**

For siblings of `MIReferrables` all of the following conditions apply:

- They have the same meta-class
- They have the same AUTOSAR path

For siblings of non-`MIReferrables` 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 **IPostBuildInvariantValuesView** 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 **IPostBuildInvariantEcucDefView** 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.

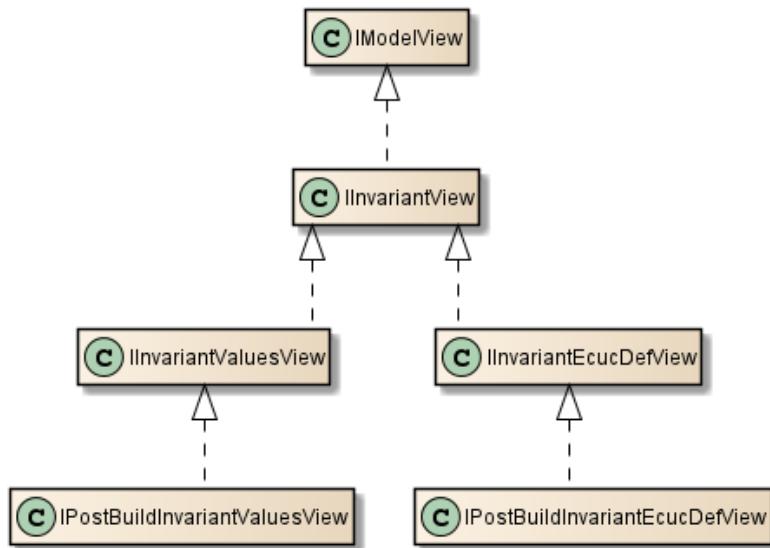


Figure 5.5: Invariant views hierarchy

**The PostBuild InvariantValues model view** The **IPostBuildInvariantValuesView** contains only elements which have **one** of the following properties:

- The element and no parent has any **MIVariationPoint** with a post-build condition

- All variant siblings have the same value and exist in all variants. Then one of the siblings is contained in the `IPostBuildInvariantValuesView`

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

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

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

Listing 5.6: Retrieving an InvariantValues model view

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

- Container A is invisible because it is contained in variant 1 only
- Container B and C are visible because they are contained in all variants
- Parameter a is visible because it is contained in all variants with the same value
- Parameter b is invisible: It is contained in all variants but with different values
- Parameter c is invisible because it is contained in variant 3 only

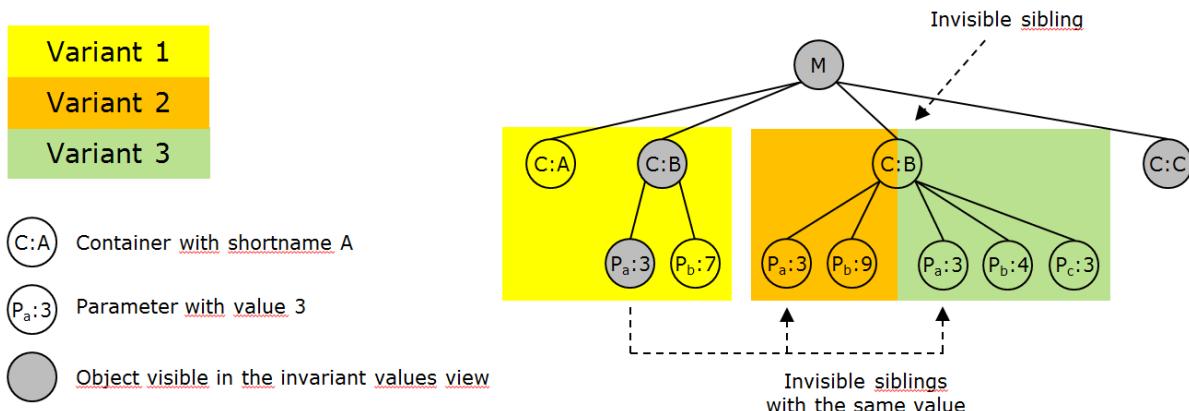


Figure 5.6: Example of a model structure and the visibility of the `IInvariantValuesView`

**Specification** See also the specification for details of the `IPostBuildInvariantValuesView`.

**The PostBuild Invariant EcuC definition model view** The `IPostBuildInvariantEcucDefView` contains the same objects as the invariant values view but additionally excludes all objects which, by (EcuC / BSWMD) definition, support variance. Using this view you can avoid dealing with objects which are accidentally equal by value (in your test configurations) but potentially can be different because they support variance.

More exact the `IPostBuildInvariantEcucDefView` will additionally exclude elements which have the following properties:

- If the parent module configuration specifies VARIANT-POST-BUILD-SELECTABLE as implementation configuration variant
  - All objects ( `MIContainer`, `MINumericalValue`, ... ) are *excluded*, which **support** variance according to their EcuC definition. (potentially variant objects)
- If the parent module configuration doesn't specify VARIANT-POST-BUILD-SELECTABLE as implementation configuration variant. All contained objects **do not** support variance, so the view actually shows the same objects as the `IPostBuildInvariantValuesView`.

The implementation configuration variant in fact overwrites the objects definition for elements in the `ModuleConfiguration`.

**Reasons to Use the view** The EcucDef view guarantees that you don't access potentially variant data without using variant specific model views. So it allows you to improve code quality in your generator.

When your test configuration for example contains equal values for a parameter which is potentially variant you will see this parameter in the invariant values view but not in the EcucDef view. Consequences if you access data in other module configurations: When the BSWMD file of this other module is being changed, e.g. a parameter now supports variance, objects can become invisible due to this change. You are forced to adapt your code then.

**Usage** You could retrieve an instance of `IPostBuildInvariantEcucDefView` by calling `IModelViewManager.getInvariantEcucDefView()`. And then use it as any other `IModelView`.

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

Listing 5.7: Retrieving an `InvariantEcucDefView` model view

**Specification** See also the specification for details of the `IPostBuildInvariantEcucDefView`.

### 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`
- `AsrPath`

- `DefRef`
- `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 following page.

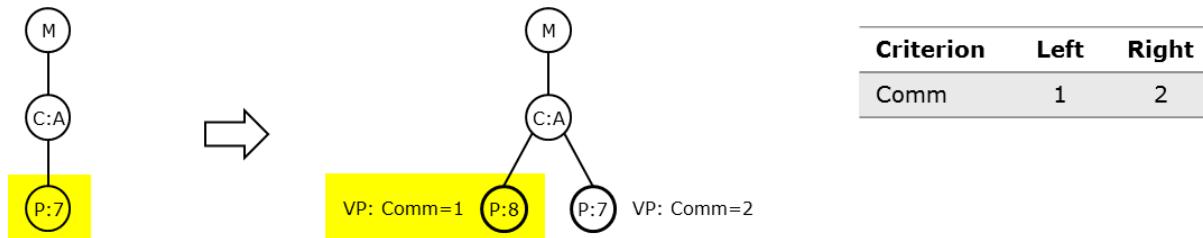


Figure 5.7: Variant specific change of a parameter value

- Before modifying the parameter, this instance is invariant. The same MDF instance is visible in all variants
- When the client code changes the parameter value, the model automatically clones the parameter first
- 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.setString(parameter, "Vector-Informatik");
    }
}
```

Listing 5.8: 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 next 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

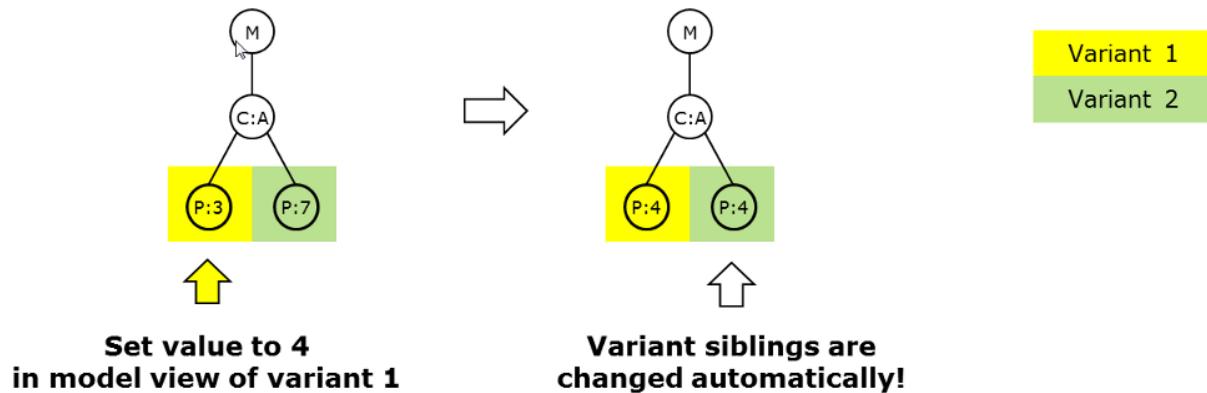


Figure 5.8: Variant common change of a parameter value

- As a result, the parameter has the same value in all variants

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

- 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 write: `comM.getComMGeneral().getComMUseRte()`

- Access containers[0:\*) /MICROSAR/ComM/ComMChannel

You can write:

```
for (ComMChannel channel : comM.getComMChannel()){
    int value = channel.getComMChannelId().getValue();
}
```

The DaVinci Configurator internal Model (MDF model) has 1:1 relationship to your BswmdModel. The BswmdModel will retrieve all data from the underlying MDF model.

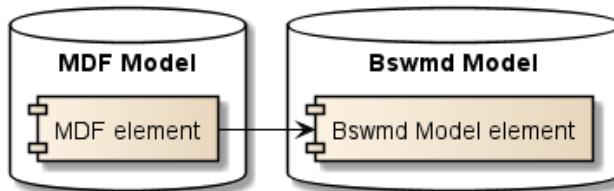


Figure 5.9: The relationship between the MDF model and the BswmdModel

**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 - `G.IContainer`
- ChoiceContainer - `GIChoiceContainer`
- Parameter - `GIPParameter<?>`

- 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 corresponding classes.

<b>AUTOSAR type</b>	<b>MDFModel</b>	<b>“Untyped” BswmdModel</b>	<b>“DefinitionTyped”</b>
ModuleConfiguration	<code>MIModuleConfiguration</code>	<code>GIModuleConfiguration</code>	<code>CanIf</code> (generated)
Container	<code>MIContainer</code>	<code>GIContainer</code>	<code>CanIfPrivateCfg</code> (generated)
String Parameter	<code>MITextualValue</code>	<code>GIParameter&lt;String&gt;</code>	<code>GString</code>
Integer Parameter	<code>MINumericalValue</code>	<code>GIParameter&lt;BigInteger&gt;</code>	<code>GInteger</code>
Reference to Container	<code>MIRefERENCEValue</code>	<code>GIReferenceToContainer</code>	<code>CanIfCtrlDrvInitHohConfigRef</code> (generated)
Enum Parameter	<code>MITextualValue</code>	<code>GIParameter&lt;String&gt;</code>	<code>CanIfDispatchBusOffUL</code> (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 retrieved via the generic getter methods like:

- `GIContainer.getSubContainers()`
- `GIContainer.getParameters()`
- `GIContainer.getParameters(TypedDefRef)`
- `GIContainer.getParameter(TypedDefRef)`
- `GIContainer.getReferencesToContainer(TypedDefRef)`
- `GIModuleConfiguration.getSubContainer(TypedDefRef)`
- `GIParameter.getValueMdf()`

Additionally there are methods to retrieve other referenced elements, like parent of reference reverse lookup:

- `GIContainer.getParent()`
- `GIContainer.getParent(DefRef)`
- `GIContainer.getReferencesPointingToMe()`
- `GIContainer.getReferencesPointingToMe(DefRef)`

The following 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 5.9: Sample code to access element in an Untyped model with DefRefs

```
final GIReferenceToContainer ref = getCanIfCanRxPduHrhRefBswmd();

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

Listing 5.10: 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 5.11: The value of a GIPparameter

The figure 5.10 on the next 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.

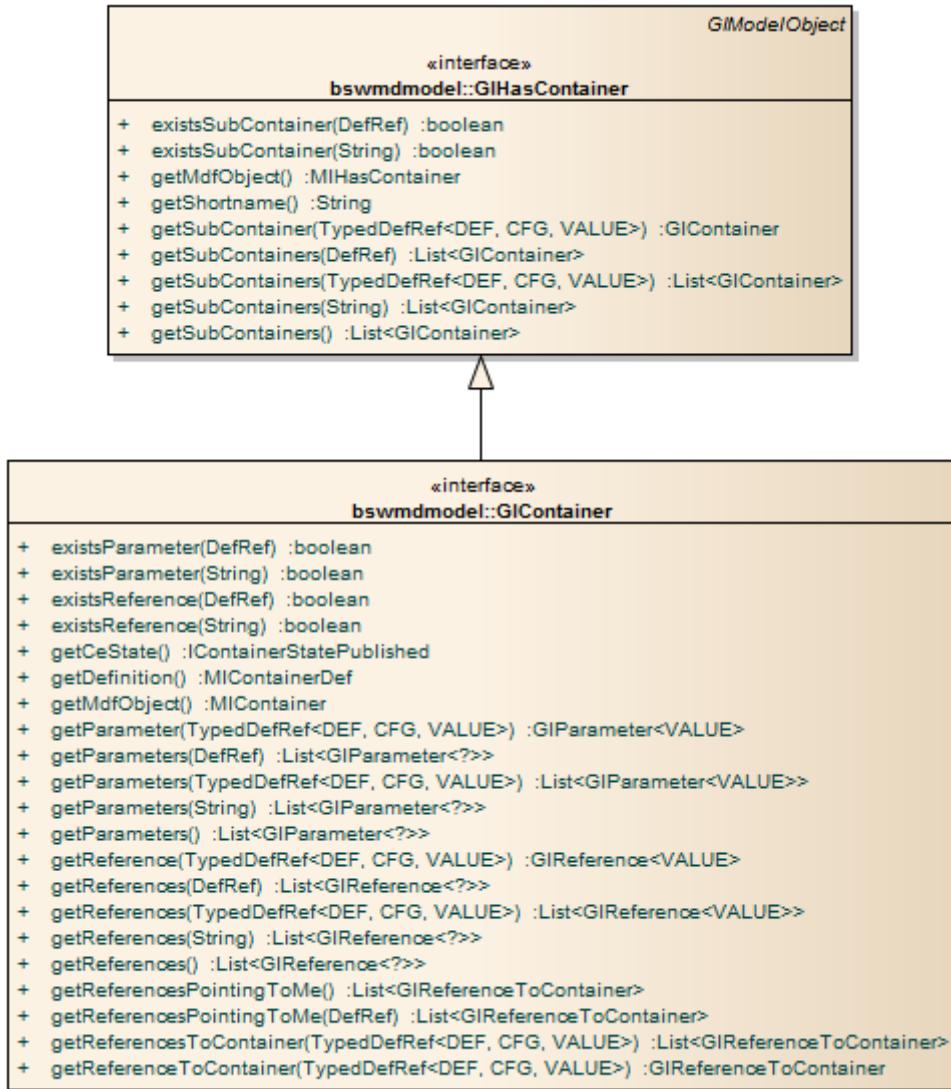


Figure 5.10: SubContainer DefRef navigation methods

### 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 of the target is known at model generation time, the return type will be the generated type, otherwise the return type is `GIContainer`.

**Note:** It is always allowed to call `getRefTarget()`, also for references pointing to external types.

There is the other method `getPossibleRefTargets()`, which returns all possible target container as list. If the type of the targets is known at model generation time, the list type will be the generated type, e.G. `List<CanGeneral>`. Otherwise the return type is `List<GIContainer>`.

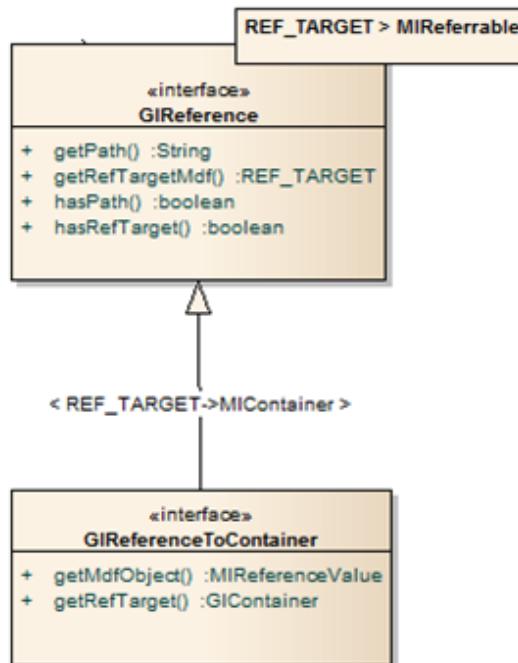


Figure 5.11: Untyped reference interfaces in the `BswmdModel`

**SymbolicNameReferences** `SymbolicNameReferences` have the same methods as `GIReferenceToContainer` and the additional methods `getRefTargetParameterMdf()`, which returns the target parameter as `MIOobject`. 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 `GIPparameter`.

**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 following 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.

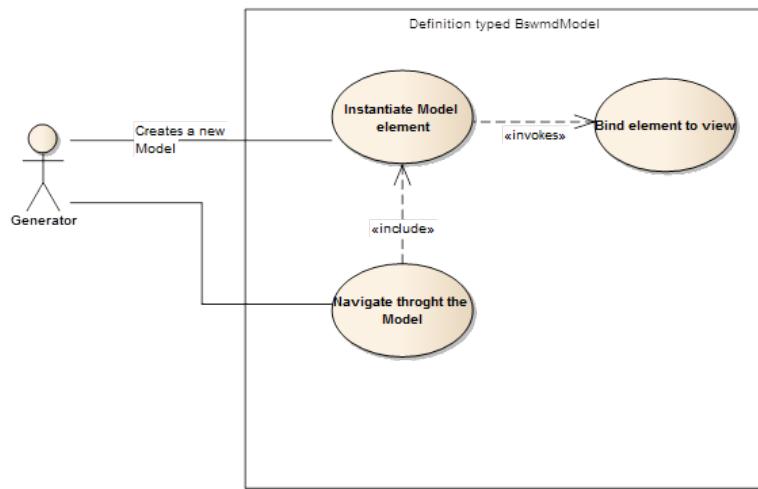


Figure 5.12: Creating a BswmdModel in the Post-build selectable use case

### 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 5.12: Java: Execute code with creation `IModelView` of `BswmdModel` object

The method `executeWithCreationModelView(Runnable)` executes the `Runnable` code under visibility of the `getCreationModelView()` of this `GIModelObject`.

```
GIModelObject myModelObject = ...;

myModelObject.executeWithCreationModelView(() ->{
    // do some operations
});
```

Listing 5.13: Java: Execute code with creation IMModelView 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 5.14: Java: Execute code with creation IMModelView 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 `BswmdModelError` is thrown.

### 5.3.1.8 Class and Interface Structure of the BswmdModel

The upper part of the figure 5.13 on the next page shows the Untyped API (GI... interfaces). The bottom left part is an example of DefinitionTyped (generated) class for the CanIf module.

The bottom right part are the classes used by the DefinitionTyped model, but are not visible in the Untyped model.

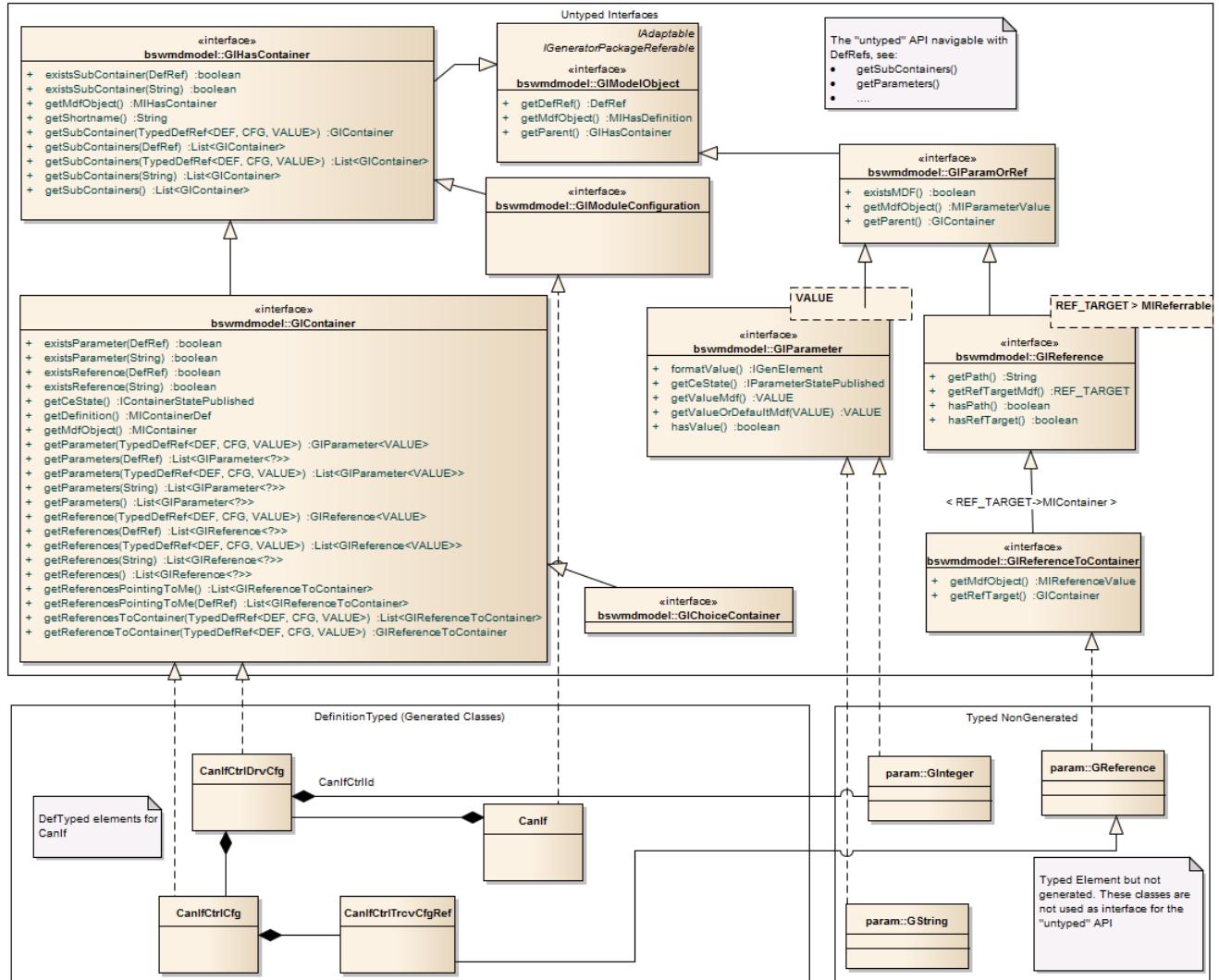


Figure 5.13: Class and Interface Structure of the BswmdModel

### 5.3.1.9 BswmdModel write access

The BswmdModel supports a write access for ecu configuration elements. This means new elements can be created and existing elements can be modified and deleted by the BswmdModel.

NOTE: The model is in read-only state by default, so no objects could be created. For this reason all calls to an API which creates or deletes elements has to be executed within a transaction. See *ModelDocumentation* chapter "Model changes" for more details.

**Optional and required Elements (0:1/1:1 Multiplicity)** For optional or required elements, the following additional methods are generated, if BswmdModelWriteAccess is enabled:

- `get...OrNull()`: Returns the requested element or `null` if it is missing.

- `get...OrCreate()`: Returns the existing requested element or implicitly creates a new one if it is missing.

E.g. `EcucGeneral`:

```
Ecuc ecuc = getEcucModuleConfig();

//Gets the EcucGeneral container or null if it is missing.
EcucGeneral ecucGeneralOrNull = ecuc.getEcucGeneralOrNull();

//Gets the existing EcucGeneral container or creates a new one if it is
//missing.
EcucGeneral ecucGeneralOrCreate = ecuc.getEcucGeneralOrCreate();
```

Listing 5.15: Additional write API methods for `EcucGeneral`

**Multiple elements (Upper Multiplicity > 1)** For each multiple element, the return type for these elements is changed from `List<>` to `GIList<>` for parameter and `GICList<>` for container, if `BswmdModelWriteAccess` is enabled. These new interfaces provide methods which allow creating and adding new children for the corresponding elements:

- `createAndAdd()`: Creates a new child element, appends it to the list and returns the new element.
- `createAndAdd(int index)`: Creates a new child element, inserts it to the list at the specified index position and returns the new element.
- For `GICList<>` only:
  - `createAndAdd(String shortName)`: Creates a new child element with the specified `shortName`, appends it to the list and returns the new element.
  - `createAndAdd(String shortName, int index)`: Creates a new child element with the specified `shortName`, inserts it to the list at the specified index position and returns the new element.
  - `byName(String shortName)`: Gets the container by specified `shortName` or throws an exception if it is missing.
  - `byNameOrNull(String shortName)`: Gets the container by specified `shortName` or `null` if it is missing.
  - `byNameOrCreate(String shortName)`: Gets the container by specified `shortName` or implicitly creates a new one if it is missing.
  - `exists(String shortname)`: Returns `true` if the container exists, otherwise `false`.

E.g. `EcucCoreDefinition`:

```

Ecuc ecuc = getEcucModuleConfig();

//Gets the EcucCoreDefinition list (create EcucHardware container if it is
//missing)
GICList<EcucCoreDefinition> ecucCores = ecuc.getEcucHardwareOrCreate().
    getEcucCoreDefinition();

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

if(ecucCores.exists("EcucCore0")) {
    //Sets EcucCoreId from EcucCore0 to 0
    ecucCores.byName("EcucCore0").getEcucCoreId().setValue(0);
}

//Creates a new EcucCore by method byNameOrCreate
EcucCoreDefinition core2 = ecucCores.byNameOrCreate("EcucCore2");

...

```

Listing 5.16: EcucCoreDefinition as GICList&lt;EcucCoreDefinition&gt;

## Other write API

- **Deleting model objects:** It is also possible to delete objects from the model.
  - `moRemove`: Deletes the specified object from the model.
  - `moIsRemoved`: Returns `true`, if the object was removed from repository, or is invisible in the current active `IModelView`.

```

//Deletes the container 'EcucGeneral' from the model.
ecucGeneral.moRemove();

//Deletes the parameter 'EcuCSafeBswChecks' from the model.
ecucGeneral.getEcuCSafeBswChecks.moRemove();

//Deletes the child container 'EcucCoreDefinition' with shortname 'EcucCore0'
//from the model.
ecucCores.byName("EcucCore0").moRemove();

// Checks if the container 'EcucGeneral' was removed from repository, or is
// invisible in the current active `IModelView`.
if(ecucGeneral.moIsRemoved()) {
    ...
}

```

Listing 5.17: Deleting model objects

- **Duplication of containers:** The method `duplicate()` copies a container with all its children and appends it to the same parent.

```
//Duplicates the container 'EcucGeneral'
EcucGeneral duplicatedEcucGeneral = ecucGeneral.duplicate();

//Duplicates the child container 'EcucCoreDefinition' with shortname '
//EcucCore0'
EcucCoreDefinition duplicatedEcucCore0 = ecucCores.byName("EcucCore0").
duplicate();
```

Listing 5.18: Duplication of containers

- **Parameter values:** The method `setValue(VALUE)` sets the value of a parameter. This method checks if the specified parameters configuration object is available and sets the new value. If the parameter object is missing it is implicitly created in the model.

```
//Sets the value of the parameter 'EcuCSafeBswChecks' to 'true'
ecucGeneral.getEcuCSafeBswChecks.setValue(true);
```

Listing 5.19: Set parameter values with the BswmdModel Write API

- **Reference targets:** The method `setRefTarget(REF_TARGET)` sets the target of a reference. This method sets the specified target object as reference target of the specified reference parameter. If the reference parameter object is missing it is implicitly created in the model.

```
//Gets the container 'OsCounter' with shortname 'SystemTimer'
OsCounter osCounterTarget = os.getOsCounters.byName("SystemTimer");

//Sets the reference target of the parameter 'CanCounterRef'
can.getCanGeneral().getCanCounterRef().setRefTarget(osCounterTarget);
```

Listing 5.20: Set reference targets with the BswmdModel Write API

### 5.3.2 BswmdModel generation

The BswmdModel for the automation interface is generated automatically by the DaVinciConfigurator.

#### 5.3.2.1 DerivativeMapping

If the SIP contains one or more modules with a DerivativeMapping, the BswmdModel classes for these modules can only be generated for one certain derivative. By default, the first derivative is selected, sorted by UUID.

If a other derivative shall be selected for BswmdModel generation a `Settings.xml` file can be defined in the SIP at `<SIP-ROOT-PATH>/DaVinciConfigurator/Generators`.

Sample file:

```

<Settings>
    <Settings Name="com.vector.cfg.gen.bswmdmodelgenerator.
        BswmdAutomationModelSettings">
        <!--Selects the derivative with the name or UUID specified by
            Value-->
        <Setting Name="SelectedDerivative" Value="SPX546B"/>
    </Settings>
</Settings>

```

Listing 5.21: Settings.xml sample for DerivativeMapping

## 5.4 Model Utility Classes

### 5.4.1 AutosarUtil

The class `AutosarUtil` is a static utility class. Its methods are not directly related to the MDF model but are useful when client code deals with AUTOSAR paths and shortnames on string basis.

Some of these methods are

- `isValidShortname(String)`: Checks if this shortname is valid according the rules, the AUTOSAR standard defines (character set for example)
- `getLastShortname(String)`: Returns the last shortname of the specified AUTOSAR path
- `getFirstShortname(String)`: Returns the first shortname of the specified AUTOSAR path
- `getAllShortnames(String)`: Returns all shortnames of the specified AUTOSAR path

### 5.4.2 AsrPath

The `AsrPath` class represents an AUTOSAR path without a connection to any model.

`AsrPaths` are constant; their values cannot be changed after they are created. This class is immutable!

### 5.4.3 AsrObjectLink

This class implements an immutable identifier for AUTOSAR objects.

An **AsrObjectLink** can be created for each object in the MDF AUTOSAR model tree. The main use case of object links is to identify an object unambiguously at a specific point in time for logging reasons. Additionally and under specific conditions it is also possible to find the related MDF object using its **AsrObjectLink** instance. But this search-by-link cannot be guaranteed after model changes (details and restrictions below).

#### 5.4.3.1 Restrictions of object links

- They are immutable and will therefore become invalid when the model changes
- So they don't guarantee that the related MDF object can be retrieved after the model has been changed. Search-by-link may even find another object or throw an exception in this case

### 5.4.4 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 its 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 on the following page shows the structure of the **DefRef** class and its sub classes.

**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

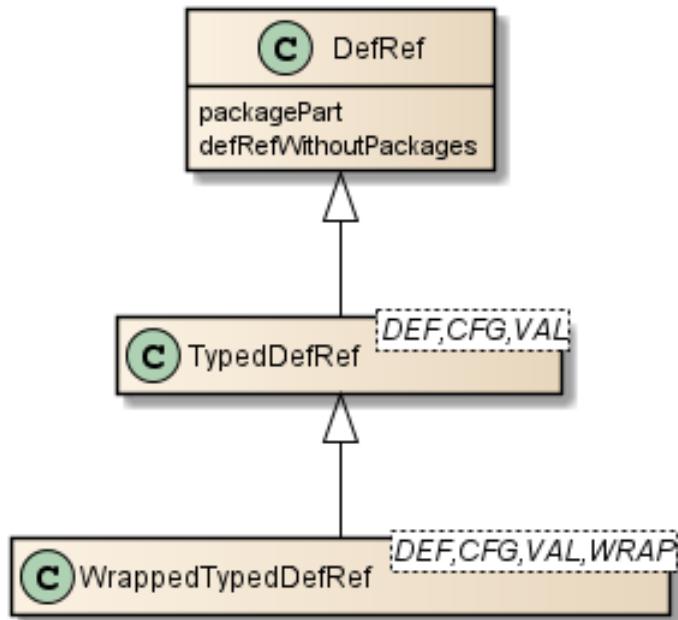


Figure 5.14: DefRef class structure

**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 5.4.4.2 on the next 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 5.22: DefRef isDefinitionOf methods

#### 5.4.4.1 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.

#### 5.4.4.2 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 5.23: 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]`.

## 5.4.5 CeState

The `CeState` is an object which allows to retrieve different states of a configuration entity.

The most important APIs for generator and script code are:

- `IParameterStatePublished`
- `IContainerStatePublished`

### 5.4.5.1 Getting a CeState object

The BSWMD models implement methods to get the `CeState` for a specific CE as the following listing shows (the types `GIPparameter` and `GIContainer` are interface base types in the BSWMD models):

```
GIPparameter parameter = ...;
IParameterStatePublished parameterState = parameter.getCeState();

GIContainer container = ...;
IContainerStatePublished containerState = container.getCeState();
```

Listing 5.24: Getting CeState objects using the BSWMD model

### 5.4.5.2 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.

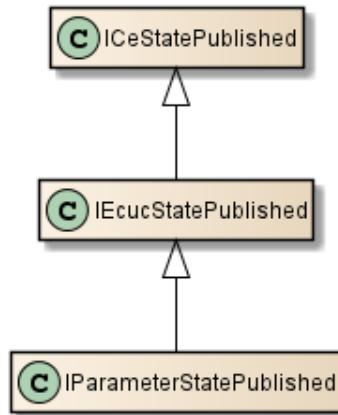


Figure 5.15: `IParameterStatePublished` class structure

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 initial (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.

#### 5.4.5.3 `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 on the next page shows the inheritance hierarchy of the `IContainerStatePublished` class and its sub classes.

This API provides state information similar to `IParameterStatePublished`. Some of the states are container-specific, of course. `getCreationPhase()`, for example, which returns the phase a

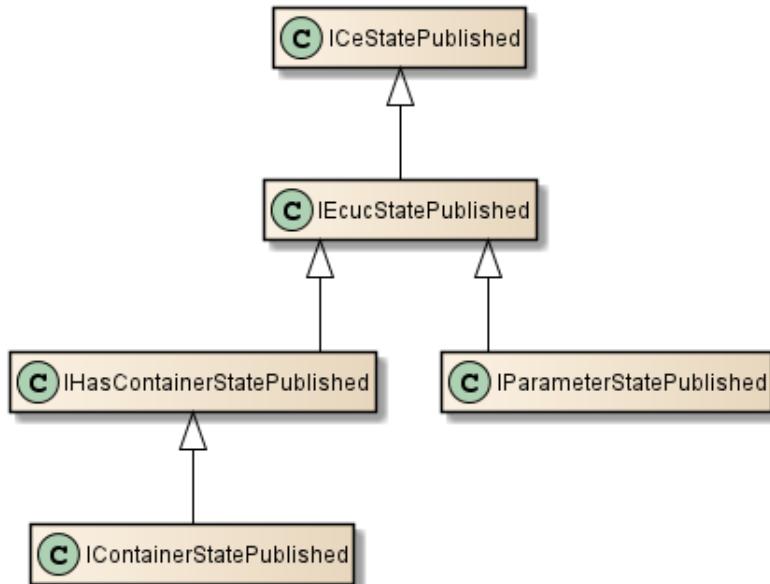


Figure 5.16: IContainerStatePublished class structure

container in a post-build loadable configuration has been created in.

## 5.5 Model Services

### 5.5.1 EcucDefinitionAccess

The `IEcucDefinitionAccess` provides convenient and typesafe access to definition objects (module, container, parameter and reference definitions). The contained `def()` methods take MDF definition objects and return wrappers which can be used to retrieve specific characteristics of definitions.

Example:

```

IEcucDefinitionAccess eda;
MIIIntegerParamDef intParamDef;

// Get the integer definition wrapper
IEcucIntegerDefinition def = eda.def(intParamDef);

// Get the (optional) default value
Optional<BigInteger> defaultOpt = def.getDefault();
boolean hasDefault = defaultOpt.isPresent();
BigInteger defaultValue = defaultOpt.get();

// Get the multiplicity
IEcucDefMultiplicity multiplicity = def.getMultiplicity();
BigInteger lower = multiplicity.getLower();
BigInteger upper = multiplicity.getUpper();
  
```

Listing 5.25: Integer parameter definition access examples

### 5.5.1.1 Post-build loadable

**EcucModuleDefinition** `IEcucModuleDefinition` is the interface of the module definition wrapper. It provides the following method(s):

#### `getSupportedConfigurationVariants()`

The `getSupportedConfigurationVariants()` method returns a collection of supported configuration variants. Never returns `null` but an empty collection if no supported config variants are specified.

The returned collection never contains the following literals:

- `EEcucConfigurationVariant.PRECONFIGURED_CONFIGURATION`
- `EEcucConfigurationVariant.RECOMMENDED_CONFIGURATION`

This method is for post-build loadable only!

**Remark about AUTOSAR versions:** Prior to AUTOSAR 4.2.1 the module definitions used the following valid values:

- `VARIANT-PRE-COMPIL`
- `VARIANT-LINK-TIME`
- `VARIANT-POST-BUILD-LOADABLE`
- `VARIANT-POST-BUILD-SELECTABLE`

`VARIANT-POST-BUILD` was invalid! With AUTOSAR 4.2.1 and later, the following values are valid (because the loadable and selectable specifications have been separated):

- `VARIANT-PRE-COMPIL`
- `VARIANT-LINK-TIME`
- `VARIANT-POST-BUILD`

`VARIANT-POST-BUILD-LOADABLE` and `VARIANT-POST-BUILD-SELECTABLE` are invalid!

This method takes the AUTOSAR version into account and returns the post-build loadable relevant specification only.

**EcucContainerDefinition** `IEcucContainerDefinition` is the interface of the container definition wrapper. It provides the following method(s):

#### `getMultiplicityConfigurationClass()`

The `getMultiplicityConfigurationClass(EEcucConfigurationVariant)` method returns the multiplicity configuration class for the specified module implementation variant. The returned value defines in which configuration phase the number of container instances latest may change if the module implements the specified variant.

Supported values for the variant are

- `EEcucConfigurationVariant.VARIANT_PRE_COMPILE`
- `EEcucConfigurationVariant.VARIANT_LINK_TIME`
- `EEcucConfigurationVariant.VARIANT_POST_BUILD_LOADABLE`

Other values lead to an `IllegalArgumentException`.

This method doesn't take the multiplicity into account. It only investigates the multiplicity configuration class as specified in the related container definition. So it still may return `EEcucConfigurationClass.POST_BUILD` even if the multiplicity is 1:1 for example. The post-build loadable use case differs here from post-build selectable (see `supportsVariantMultiplicity()`) because the changeability in the post-build phase is being inherited from parent objects. So, if you want to find out if a container actually permits changes in the post-build phase, you should use `IContainerStatePublished`.

This method is for post-build loadable only!

**Remark about AUTOSAR versions:** Prior to AUTOSAR 4.2.1 the container definitions contained the `postBuildChangeable` flag to define post-build loadable support. This method internally investigates the `postBuildChangeable` flag in this case but the `multiplicityConfigClass` table for AUROSAR 4.2.1 and newer versions.

**EcucCommonAttributes** `IEcucCommonAttributes` is the base interface of all parameter and reference definition wrappers. It provides the following method(s):

#### `getMultiplicityConfigurationClass()`

The `getMultiplicityConfigurationClass(EEcucConfigurationVariant)` method returns the multiplicity configuration class for the specified module implementation variant. The returned value defines in which configuration phase the number of parameter instances latest may change if the module implements the specified variant.

Supported values for the variant are

- `EEcucConfigurationVariant.VARIANT_PRE_COMPILE`
- `EEcucConfigurationVariant.VARIANT_LINK_TIME`
- `EEcucConfigurationVariant.VARIANT_POST_BUILD_LOADABLE`

Other values lead to an `IllegalArgumentException`.

This method doesn't take the multiplicity into account. It only investigates the multiplicity configuration class as specified in the related parameter definition. So it still may return `EEcucConfigurationClass.POST_BUILD` even if the multiplicity is 1:1 for example. The post-build loadable use case differs here from post-build selectable (see `supportsVariantMultiplicity()`) because the changeability in the post-build phase is being inherited from parent objects. So, if you want to find out if a parameter actually permits changes in the post-build phase, you should use `IParameterStatePublished`.

This method is for post-build loadable only!

**Remark about AUTOSAR versions:** Prior to AUTOSAR 4.2.1 the parameter definitions contain the `implementationConfigClass` table to define post-build loadable support. This method internally investigates the `implementationConfigClass` in this case but the `multiplicityConfigClass` table for AUROSAR 4.2.1 and newer versions.

#### `getValueConfigurationClass()`

The `getValueConfigurationClass(EEcucConfigurationVariant)` method returns the value configuration class for the specified module implementation variant. The returned value defines in which configuration phase the value of parameter instances latest may change if the module implements the specified variant.

Supported values for the variant are

- `EEcucConfigurationVariant.VARIANT_PRE_COMPILE`
- `EEcucConfigurationVariant.VARIANT_LINK_TIME`
- `EEcucConfigurationVariant.VARIANT_POST_BUILD_LOADABLE`

Other values lead to an `IllegalArgumentException`.

This method never returns `EEcucConfigurationClass.LINK`.

This method is for post-build loadable only!

**Remark about AUTOSAR versions:** Prior to AUTOSAR 4.2.1 the parameter definitions contain the `implementationConfigClass` table to define post-build loadable support. This method internally investigates the `implementationConfigClass` in this case but the `valueConfigClass` table for AUROSAR 4.2.1 and newer versions.

### 5.5.1.2 Post-build selectable

**EcucModuleDefinition** `IEcucModuleDefinition` is the interface of the module definition wrapper. It provides the following method(s):

**supportsPostBuildVariance()**

The `supportsPostBuildVariance()` method returns `true` if this module configuration supports post-build selectable.

**Remark about AUTOSAR versions:** Prior to AUTOSAR 4.2.1 the module definitions `supportedSupportedConfigurationVariants` defined both, post-build loadable and selectable support. With AUTOSAR 4.2.1 the `supportedSupportedConfigurationVariants` specifies post-build loadable only and this method returns the value of the new `postBuildVariantSupport` flag.

**EcucCommonAttributes** `IEcucContainerDefinition` is the interface of the container definition wrapper. It provides the following method(s):

**supportsVariantMultiplicity()**

The `supportsVariantMultiplicity()` method returns `true` if this container type supports variant multiplicity. If `true` is returned this means that different variants may contain different number of instances of this container type.

This method takes the multiplicity into account. So, if the container definition specifies the multiplicity with lower == upper, it always returns `false`. Concerning post-build selectable it never makes sense to permit variance if lower and upper multiplicity are equal.

This method is for post-build selectable only!

**Remark about AUTOSAR versions:** Prior to AUTOSAR 4.2.1 the container definitions contained the `postBuildChangeable` flag to define post-build loadable support. This method internally investigates the `postBuildChangeable` flag in this case but the `postBuildVariantMultiplicity` flag for AUROSAR 4.2.1 and newer versions.

**supportsVariantShortname()**

The `supportsVariantShortname()` method returns `true` if one of the following conditions apply.

- `supportsVariantMultiplicity()` returns `true`
- The ADMIN-DATA flag `postBuildSelectableChangeable` is `true`

The use case for this specification are 1:1 containers. When this method returns `true`, 1:1 containers may have different shortnames in different variants. This is a Vector specific semantic which is not provided by AUTOSAR.

**EcucCommonAttributes** `IEcucCommonAttributes` is the base interface of all parameter and reference definition wrappers. It provides the following method(s):

#### **supportsVariantMultiplicity()**

The `supportsVariantMultiplicity()` method returns `true` if this parameter type supports variant multiplicity. If `true` is returned this means that different variants may contain different number of instances of this parameter type.

This method takes the multiplicity into account. So, if the parameter definition specifies the multiplicity with lower == upper, it always returns `false`. Concerning post-build selectable it never makes sense to permit variance if lower and upper multiplicity are equal.

This method is for post-build selectable only!

**Remark about AUTOSAR versions:** Prior to AUTOSAR 4.2.1 the parameter definitions contain the `implementationConfigClass` table to define post-build selectable support. This method internally investigates the `implementationConfigClass` in this case but the `postBuildVariantMultiplicity` flag for AUROSAR 4.2.1 and newer versions.

#### **supportsVariantValue()**

The `supportsVariantValue()` method returns `true` if this parameter type supports a variant value. If `true` is returned this means that different variants may contain different values in instances of this parameter type.

This method is for post-build selectable only!

**Remark about AUTOSAR versions:** Prior to AUTOSAR 4.2.1 the parameter definitions contain the `implementationConfigClass` table to define post-build selectable support. This method internally investigates the `implementationConfigClass` in this case but the `postBuildVariantValue` flag for AUROSAR 4.2.1 and newer versions.

### **5.5.2 EcuConfigurationAccess**

The `IEcuConfigurationAccess` provides convenient and typesafe access to configuration objects (modules, containers, parameters and references). The contained `cfg()` methods take MDF (ECU configuration) objects and return wrappers which can be used to retrieve specific characteristics of the configuration content.

Example:

```

IEcuConfigurationAccess eca;
MINumericalValue intParam;

// Get the parameter wrapper
IEcucNumericalParameter numCfg = eca.cfg(intParam);

// Check if this is an integer parameter
if (numCfg instanceof IEcucIntegerParameter) {
    IEcucIntegerParameter intCfg = (IEcucIntegerParameter) numCfg;

    // Get the parameter value
    boolean hasValue = intCfg.hasValue();
    BigInteger value = intCfg.getValue();

    // Get the related definition wrapper
    IEcucIntegerDefinition def = intCfg.getEcucDefinition();
}

```

Listing 5.26: Integer parameter configuration access examples

### 5.5.2.1 Post-build loadable

**EcucModuleConfiguration** `IEcucModuleConfiguration` is the base interface of all module configuration wrappers. It provides the following method(s):

#### `getConfigurationVariant()`

The `getConfigurationVariant()` method returns the modules configuration variant.

This method never returns `null`. If the module has no value specified, this method returns a default value as follows:

- `EEcucConfigurationVariant.VARIANT_PRE_COMPILE`, if it is contained in the supported config variants of the related module definition
- otherwise `EEcucConfigurationVariant.VARIANT_LINK_TIME`, if it is contained in the supported config variants of the related module definition
- otherwise `EEcucConfigurationVariant.VARIANT_POST_BUILD_LOADABLE`, if it is contained in the supported config variants of the related module definition
- otherwise `EEcucConfigurationVariant.VARIANT_PRE_COMPILE`, even if not contained in the supported config variants of the related module definition or if the definition is not available

**Remark about AUTOSAR versions:** Prior to AUTOSAR 4.2.1 the module configurations implementation config variant defined if this module implements post-build loadable and/or selectable. With AUTOSAR 4.2.1 the implementation config variant defines only if the module implements post-build loadable. The post-build selectable aspect has been separated from this definition. This method handles the loadable semantic, independent of the AUTOSAR version.

This is for post-build loadable only!

#### `setConfigurationVariant()`

The `setConfigurationVariant(EEcucConfigurationVariant)` method sets the specified implementation configuration variant.

This is for post-build loadable only!

Supported values are

- `EEcucConfigurationVariant.VARIANT_PRE_COMPILE`
- `EEcucConfigurationVariant.VARIANT_LINK_TIME`
- `EEcucConfigurationVariant.VARIANT_POST_BUILD_LOADABLE`

#### Remarks concerning AUTOSAR versions:

- If the modules definition has schema version 4.2.1 or higher, the specified value is being written directly to the model
- If the modules definition has a schema version lower than 4.2.1, the modules implementation configuration variant in the MDF model encodes both, post-build loadable and post-build selectable. The following behavior is being implemented in this case:

Current model value	Parameter	Result in the model
PRE_COMPILE	PRE_COMPILE	PRE_COMPILE
	LINK_TIME	LINK_TIME
	POST_BUILD_LOADABLE	POST_BUILD_LOADABLE
LINK_TIME	PRE_COMPILE	PRE_COMPILE
	LINK_TIME	LINK_TIME
	POST_BUILD_LOADABLE	POST_BUILD_LOADABLE
POST_BUILD_LOADABLE	PRE_COMPILE	PRE_COMPILE
	LINK_TIME	LINK_TIME
	POST_BUILD_LOADABLE	POST_BUILD_LOADABLE
POST_BUILD_SELECTABLE	PRE_COMPILE	POST_BUILD_SELECTABLE
	LINK_TIME	POST_BUILD_SELECTABLE
	POST_BUILD_LOADABLE	POST_BUILD
POST_BUILD	PRE_COMPILE	POST_BUILD_SELECTABLE
	LINK_TIME	POST_BUILD_SELECTABLE
	POST_BUILD_LOADABLE	POST_BUILD

**EcucContainer** `IEcucContainer` is the base interface of all container wrappers. It provides the following method(s):

##### `getEffectiveMultiplicityConfigurationClass()`

The `getEffectiveMultiplicityConfigurationClass()` method walks up the model tree to find the related module configuration. Then it uses the module implementation configuration variant to return the selected configuration class as specified in the container definition.

This method never returns `null`. In case the detection of the configuration class fails (e.g. if the related module configuration cannot be detected), this method returns `EEcucConfigurationClass.PRE_COMPILE` by default. It also never returns `EEcucConfigurationClass.LINK`.

This method is for post-build loadable only!

##### `getEffectiveMultiplicityConfigurationClassDefRef()`

The `getEffectiveMultiplicityConfigurationClass(DefRef)` method walks up the model tree to find the related module configuration. Then it uses the module implementation configuration variant to return the selected configuration class of the specified parameter definition.

This method never returns `null`. In case the detection of the configuration class fails (e.g. if the related module configuration cannot be detected), this method returns `EEcucConfigurationClass.PRE_COMPILE` by default. It also never returns `EEcucConfigurationClass.LINK`.

This method is for post-build loadable only!

##### `getEffectiveValueConfigurationClass()`

The `getEffectiveValueConfigurationClass(DefRef)` method walks up the model tree to

find the related module configuration. Then it uses the module implementation configuration variant to return the selected configuration class of the specified parameter definition.

This method never returns `null`. In case the detection of the configuration class fails (e.g. if the related module configuration cannot be detected), this method returns `EEcucConfigurationClass.PRE_COMPILE` by default. It also never returns `EEcucConfigurationClass.LINK`.

This method is for post-build loadable only!

**EcucParameter** `IEcucParameter` is the base interface of all parameter and reference wrappers. It provides the following method(s):

#### `getEffectiveMultiplicityConfigurationClass()`

The `getEffectiveMultiplicityConfigurationClass()` method walks up the model tree to find the related module configuration. Then it uses the module implementation configuration variant to return the selected configuration class as specified in the parameter definition.

This method never returns `null`. In case the detection of the configuration class fails (e.g. if the related module configuration cannot be detected), this method returns `EEcucConfigurationClass.PRE_COMPILE` by default.

This is for post-build loadable only!

#### `getEffectiveValueConfigurationClass()`

The `getEffectiveValueConfigurationClass()` method walks up the model tree to find the related module configuration. Then it uses the module implementation configuration variant to return the selected configuration class as specified in the parameter definition.

This method never returns `null`. In case the detection of the configuration class fails (e.g. if the related module configuration cannot be detected), this method returns `EEcucConfigurationClass.PRE_COMPILE` by default.

This is for post-build loadable only!

### 5.5.2.2 Post-build selectable

**EcucModuleConfiguration** `IEcucModuleConfiguration` is the base interface of all module configuration wrappers. It provides the following method(s):

#### `supportsPostBuildVariance()`

The `supportsPostBuildVariance()` method returns `true` if this module configuration supports post-build selectable.

This is for post-build selectable only!

What this method actually does:

- It checks if the related definition specifies post-build selectable as supported
- It checks if the module configuration implements post-build variance. That's `true` in the following cases
  - If the modules definition has schema version 4.2.1 or higher: Check if the modules ADMIN-DATA flag "postBuildVariantSupport" is `true` (false is default if this flag is missing)

- If the modules definition has a schema version lower than 4.2.1: Check if the modules implementation configuration variant contains one of the following values VARIANT\_POST\_BUILD\_SELECTABLE or VARIANT\_POST\_BUILD

It returns **true** if both conditions are **true**.

#### **setPostBuildVarianceSupport()**

The **setPostBuildVarianceSupport(boolean)** method sets the post-build support flag in the module configuration.

This is for post-build selectable only!

#### Remarks concerning AUTOSAR versions:

- If the modules definition has schema version 4.2.1 or higher, this method sets the modules ADMIN-DATA flag "postBuildVariantSupport" to the specified value.
- If the modules definition has a schema version lower than 4.2.1, the modules implementation configuration variant in the MDF model encodes both, post-build loadable and post-build selectable. The following behavior is being implemented in this case:

Current model value	Parameter	Result in the model
PRE_COMPILE	true	POST_BUILD_SELECTABLE
	false	PRE_COMPILE
LINK_TIME	true	POST_BUILD_SELECTABLE
	false	LINK_TIME
POST_BUILD_LOADABLE	true	POST_BUILD
	false	POST_BUILD_LOADABLE
POST_BUILD_SELECTABLE	true	POST_BUILD_SELECTABLE
	false	PRE_COMPILE
POST_BUILD	true	POST_BUILD
	false	POST_BUILD_LOADABLE

**EcucContainer IEcucContainer** is the base interface of all container wrappers. It provides the following method(s):

#### **supportsVariantMultiplicity()**

The **supportsVariantMultiplicity()** method returns **true** if the related module configuration supports variance and this containers definition support variant multiplicity. If **true** is returned this means that different variants may contain different number of instances of this container.

If the container has no definition, this method returns **false**.

This method is for post-build selectable only!

**EcucParameter IEcucParameter** is the base interface of all parameter and reference wrappers. It provides the following method(s):

#### **supportsVariantMultiplicity()**

The **supportsVariantMultiplicity()** method returns **true** if the related module configuration supports variance and this parameters definition support variant multiplicity. If **true** is returned this means that different variants may contain different number of instances of this parameter.

If the parameter has no definition, this method returns **false**.

This is for post-build selectable only!

### **supportsVariantValue()**

The `supportsVariantValue()` method returns `true` if the related module configuration supports variance and this parameters definition support variant values. If `true` is returned this means that different variants may contain different values in instances of this parameter.

If the parameter has no definition, this method returns `false`.

This is for post-build selectable only!

# 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 7.9 on page 376
    - \* **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 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 next page.

### 6.3.2 Javadoc HTML Pages

You find this documentation as described in chapter 6.2 on the preceding 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 previous 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 preceding page . And it contains other libraries which are described in **libs** in 6.2 on the previous page.

## 6.5 Beta API Usage

The beta annotation is exempt from any compatibility guarantees made by its containing library. Note that the presence of this annotation implies nothing about the quality or performance of the API in question, only the fact that it is not "API-frozen".

Note that the client which uses this API must upgrade to each product release, to guarantee, that the used API is still available.

See below how beta annotation looks like.

```
@PublishedBeta  
@PublishedApi(ChangePolicy.ADDITIONS_ALLOWED)  
@PublishedApiDomain(DomainType.AUTOMATION_IF)  
@ThreadSafe  
public interface IConfigureVariantsApiEntryPoint {
```

Figure 6.1: Beta API Annotation

Use Beta Annotated API in Script Projects.



The screenshot shows a code editor with a file tree on the left and a code editor on the right. The file tree shows a project structure with a `src` folder containing a `main` folder which has a `groovy` folder containing a file named `MyScript.groovy`. Below the `src` folder is a `build.gradle` file, which is currently selected.

```
1  /*
2   * AutomationInterface bootstrap
3   */
4  apply from:'dvCfgAutomationBootstrap.gradle'
5
6
7  dvCfgAutomation {
8      classes project.ext.automationClasses
9  }
10
11 dvCfgAutomation {
12     allowBetaApiUsage = true
13 }
14
```

Figure 6.2: Add beta API usage - build.gradle

# 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 7.9 on page 376.

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 16.

## 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 7.9.2 on page 376 for details
    - \* `gradlew.bat`
    - \* `build.gradle`
    - \* `settings.gradle`
    - \* `projectConfig.gradle`
    - \* `dvCfgAutomationBootstrap.gradle`

- IntelliJ Project files (optional) - **DO NOT** commit it into a version control system
  - \* `ProjectName.iws`
  - \* `ProjectName.iml`
  - \* `ProjectName.ipr`

The IntelliJ Project files (`*.iws`, `*.iml`, `*.ipr`) can be recreated with the command in the windows command shell (`cmd.exe`): `gradlew idea`

## 7.4 Deployment of the Jar File

To deploy your automation script project you only need to deploy the built jar file located in `<ProjectDir>/build/libs/<ProjectName>-<Version>.jar`. All other files in your automation script project are **not required** for the script **execution**.

So if you want to use your script project in an DaVinci Configurator project, copy the jar file into the DaVinci Configurator project and add the folder containing the jar file in the Script Locations view with the Project scope.

## 7.5 IntelliJ IDEA Usage

### 7.5.1 Supported versions

The supported IntelliJ IDEA versions are:

- 2016.3
- 2017.2
- 2017.3
- 2018.1
- 2018.2
- 2018.3
- 2019.1
- 2019.3.1

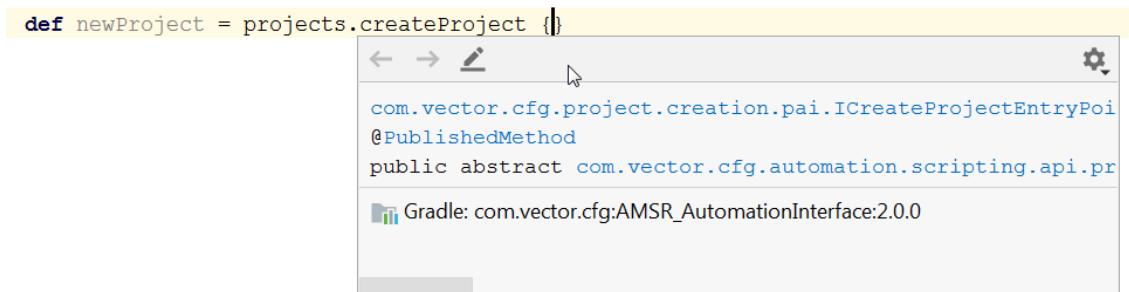
Please use one of the versions above. With other versions, there could be problems with the editing, code completion and so on. The support for the IntelliJ IDEA versions prior to 2016.3 was removed. Please update your version.

The free **Community edition** is **fully sufficient**, but you could also use the *Ultimate edition*.

### 7.5.2 Show API Specifications (JavaDoc)

In newer IntelliJ versions the automatically download of the source files and their respective javadocs has been disabled. In order to benefit from the API-Specifications during coding it is necessary to download the source files. This has to be done manually.

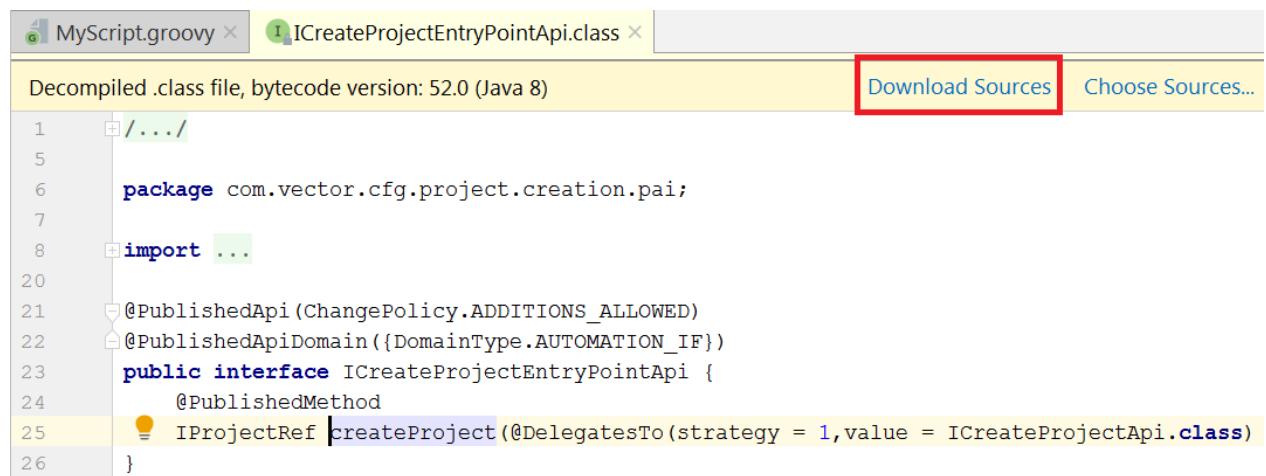
If source files are not yet downloaded, it looks like 7.1.



```
def newProject = projects.createProject {}  
com.vector.cfg.project.creation.pai.ICreateProjectEntryPoint  
@PublishedMethod  
public abstract com.vector.cfg.automation.scripting.api.pr  
Gradle: com.vector.cfg:AMSR_AutomationInterface:2.0.0
```

Figure 7.1: No JavaDoc

To download source files enter with F3 the API and click on "Download Sources" 7.2.



Decompiled .class file, bytecode version: 52.0 (Java 8)

Download Sources Choose Sources...

```
1 package com.vector.cfg.project.creation.pai;  
2 import ...  
3 @PublishedApi(ChangePolicy.ADDITIONS_ALLOWED)  
4 @PublishedApiDomain({DomainType.AUTOMATION_IF})  
5 public interface ICreateProjectEntryPointApi {  
6     @PublishedMethod  
7     IProjectRef createProject(@DelegatesTo(strategy = 1,value = ICreateProjectApi.class)  
8 }
```

Figure 7.2: No JavaDoc

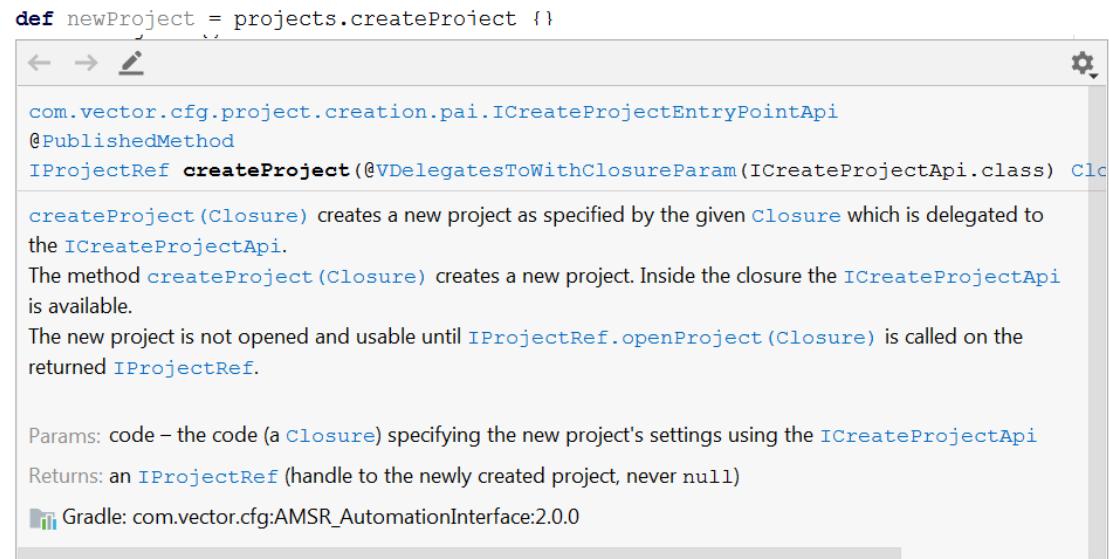


Figure 7.3: JavaDoc

### 7.5.3 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.

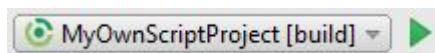


Figure 7.4: Project Build

**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.

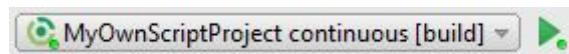


Figure 7.5: Project Continuous Build

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



Figure 7.6: 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.

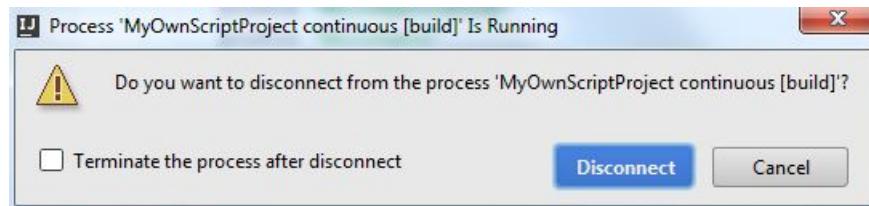


Figure 7.7: Disconnect from Continuous Build Process

#### 7.5.4 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 7.8 on page 375.

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.

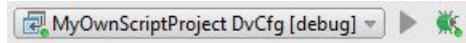


Figure 7.8: Project Debug

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



Figure 7.9: 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.

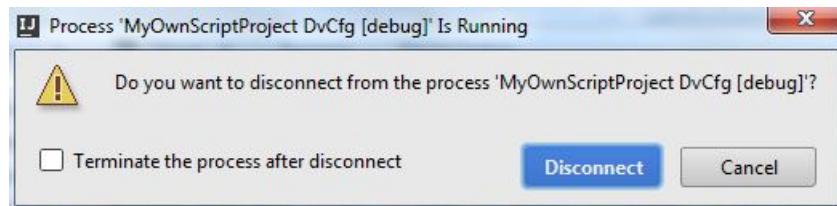


Figure 7.10: Disconnect from Debug Process

### 7.5.5 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 19.

**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 19.

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 7.5 on page 369.

**Groovy SDK not configured** If you get the message 'Groovy SDK is not configured for ...' in IntelliJ IDEA you probably have to migrate your project as described in chapter 7.7 on page 375.

**No JavaDoc Shown** If you don't see a javadoc description for the APIs. See chapter 7.5.2 on page 369.

**Debugging - DaVinci Configurator does not start** If the DaVinciCfg.exe does not start when the enableDebugger option is passed, please check if the default port (8000) is free, or choose another free port by appending the port number to the enableDebugger option.

**Compile errors - Could not find com.vector.cfg:DVCfgAutomationInterface** If you get compile errors inside of the IntelliJ IDEA, after updating the DaVinci Configurator or moving projects.

Please execute the **Project Migration to newer DaVinci Configurator Version** step, see 7.7 on the next page.

**Download of Gradle Distribution Error** If you get an error when you start the `gradlew` like:

Downloading

```
http://vistrcfgci1.vi.vector.int/buildcomponents/Gradle/distributions/gradle-4.0.1-bin.zip
```

```
Exception in thread "main" java.io.FileNotFoundException:
```

```
http://vistrcfgci1.vi.vector.int/buildcomponents/Gradle/distributions/gradle-4.0.1-bin.zip
at sun.net.www.protocol.http.HttpURLConnection.getInputStream0(HttpURLConnection.java:1836)
```

The problem is you can't connect to the server, where the Gradle installation is located<sup>1</sup>. To change the location, you have to open the file `<YourProject>/gradle/wrapper/gradle-wrapper.properties` and change the line `distributionUrl=`.

You have multiple options for the content of the `distributionUrl`:

- Change the URL to the Gradle default (needs internet access):
  - `https://services.gradle.org/distributions/gradle-4.0.1-bin.zip`
- Change the URL to a Server location of your choice. E.g inside your company.
- Download Gradle manually and change the URL to a local file system location like:
  - `file:/D:/YourFolder/gradle-4.0.1-bin.zip`

**Caution:** You have to escape a `:` with `\:` so an HTTP address would start with `http\://` and the local filesystem would start with `file\:/`.

So the default line in the file ‘gradle-wrapper.properties’ for the default Gradle server would be:  
`distributionUrl=https\://services.gradle.org/distributions/gradle-4.0.1-bin.zip`

## 7.6 Project Usage in different DaVinci Configurator Versions

You can execute the script tasks of a script project in different versions of the DaVinci Configurator as long as the following conditions are met:

- You compile your script project against the oldest DaVinci Configurator version to use
  - E.g. You want to use Cfg5.15 and Cfg5.16, you have to compile your script project with Cfg5.15.
- The DaVinci Configurator version span must not contain a breaking change. These changes are documented in the chapter 8 on page 382. Normally the versions have **NO** breaking changes! The default text is something like:

---

<sup>1</sup> The vector internal server `http://vistrcfgci1.vi.vector.int` is not accessible from outside of the vector network and shall only be used by internal projects. If you have a project with the internal server and your are not inside the network, please change it to another location.

"The Cfg5.16 automation interface is compatible to the Cfg5.15. So a script written with Cfg5.15 will also run in the Cfg5.16 version."

- If you use the BswmdModel, you have to use a compatible SIP
  - E.g. the used BSW module definitions (bswmd files) must have compatible names and multiplicities.

## 7.7 Project Migration to newer DaVinci Configurator Version / SIP

If you update your DaVinci Configurator version in your SIP or copy the project into another SIP, you should execute the IntelliJ IDEA task of Gradle to update the compile dependencies.

Steps to execute:

1. Close IntelliJ IDEA.
2. Update the DaVinci Configurator in your SIP / Change the path to the DaVinci Configurator in the file `projectConfig.gradle`
3. Update the IntelliJ IDEA project, see 7.9.3.2 on page 378

This will update the compile time dependencies of your Automation Script Project according to the new DaVinci Configurator version.

After this, please read the Changes (see chapter 8 on page 382) in the new release and update your script, if something of interest has changed.

## 7.8 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). If the DaVinci Configurator does not start with the option, please see 7.5.5 on page 373.

### Different Debug Port

```
DVCfgCmd -s MyApplScriptTask --enableDebugger <YOUR-PORT> --waitForDebugger  
Example:
```

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

You could attach a debugger at port 12345 (select any free port) 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.

## 7.9 Build System

The build system uses Gradle<sup>2</sup> 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 20.

### 7.9.1 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`

### 7.9.2 Gradle File Structure

The default automation project contains the following Gradle build files:

- `gradlew.bat`
  - Gradle batch file to start Gradle (Gradle Wrapper<sup>3</sup>)
- `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<sup>4</sup>
- `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.

#### 7.9.2.1 `projectConfig.gradle` File settings

The file contains three essential parts of the build:

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

<sup>2</sup><https://gradle.org/> [2018-11-27]

<sup>3</sup>[https://docs.gradle.org/current/userguide/gradle\\_wrapper.html](https://docs.gradle.org/current/userguide/gradle_wrapper.html) [2018-11-27]

<sup>4</sup><https://docs.gradle.org/current/dsl/org.gradle.api.initialization.Settings.html> [2018-11-27]

**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

**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.

```
// Please specify the path to your DaVinci Configurator installation
project.ext.dvCfgInstallation = new File("<PATH-TO-DaVinciConfiguratorFolder>")
```

Listing 7.2: The dvCfgInstallation in projectConfig.gradle

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.

```
// Use a System environment variable as path to the DaVinci Configurator
project.ext.dvCfgInstallation = new File(System.getenv('YOUR_ENV_VARIABLE'))
```

Listing 7.3: The dvCfgInstallation with an System env in projectConfig.gradle

**version** The `project.version` defines the version of your Automation project, e.g. defines the version suffix of the jar file.

### 7.9.3 Advanced Build Topics

#### 7.9.3.1 Usage of external Libraries (Jars) in the AutomationProject

You could reference external libraries (Jar files) in your `AutomationProject`. But you have to configure the libraries in the Gradle build files. **DO NOT** add a dependency in IntelliJ, this will not work.

The easiest and preferred way is the use a library from any Maven repository like MavenCentral or JCenter. This will also handle versions, and transitive dependencies automatically.

Otherwise you could download the jar file and place it in your project<sup>5</sup>, but this is **NOT** recommended.

The referenced libraries will be automatically bundled into your Automation project, see chapter 7.9.3.5 on page 380 `includeDependenciesIntoJar` for details.

**How to add a Library?** We assume we have a jar from a Maven repository like Apache Commons IO (the identifier would be '`commons-io:commons-io:2.5`', See MavenCentral).

- Open your `build.gradle`
- Add the code for the dependency

```
dependencies {
    // Change the identifier to your library to use
    compile 'commons-io:commons-io:2.5'
    // You could add multiple libraries with additional compile lines
}
```

- Optional: if you are behind a proxy or firewall:
  - You must either set proxy options for gradle<sup>6</sup>
  - **Preferred way:** use a Maven repository inside your network: To set a repository, add before the dependencies block:

```
repositories {
    // URL to your repository
    // The URL below is the Vector internal network server
    // Please change the URL to your server
    maven { url 'https://vistrpesart1.vi.vector.int/artifactory/pes-davinciall-
        maven' }
    // Or reference MavenCentral server
    mavenCentral()
}
```

- Update the IntelliJ IDEA project, see 7.9.3.2

Now your project has access to the specified library.

### 7.9.3.2 Update IntelliJ IDEA project

If you have changed dependencies or versions of any library, you have to update your IDEA project. The best way to do this is to select the Gradle auto-import feature in the IntelliJ IDEA, see figure 7.11 on the next page.

---

<sup>5</sup>See Gradle online documentation, how to add local jar files to the build dependencies

<sup>6</sup>Gradle and Java online documentation for details how to set proxy settings

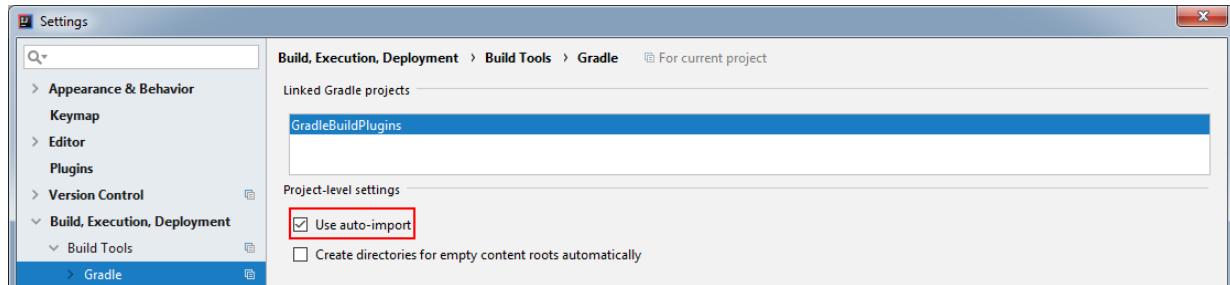


Figure 7.11: Activate the auto-import

Or you can manually execute the following steps:

1. Close IntelliJ IDEA.
2. Open a command shell (`cmd.exe`) at your project folder
  - Folder containing the `gradlew.bat`
3. Type `gradlew idea` and press enter
4. Wait until the task has finished
5. Open IntelliJ IDEA

### 7.9.3.3 Static Compilation of Groovy Code

The AutomationInterface contains a Groovy compiler extension. In some use cases performance has priority and therefore it is possible to use Groovy with static compilation.

Groovy is a dynamic JVM language using dynamic dispatch for its method calls. Dynamic dispatch in Groovy is approximately three times slower compared to a normal Java method call. Groovy has added the static compilation feature via `@CompileStatic` annotation, which allows to compile most of Groovy method calls into direct JVM bytecode method calls, thus avoiding all the dynamic dispatch overhead.<sup>7)</sup>

Mark your classes or methods with:

```
@CompileStatic(extensions = 'com.vector.cfg.groovy.extensions.
    AutomationTypeChecking')
def myMethod(){

}

@CompileStatic(extensions = 'com.vector.cfg.groovy.extensions.
    AutomationTypeChecking')
class MyClass{

}
```

Listing 7.4: `@CompileStatic` with Automation API

The same applies, if you want to use the `@TypeChecked` annotation:

<sup>7</sup><http://java-performance.info/static-code-compilation-groovy-2-0/> [2018-11-29]

```
@TypeChecked(extensions = 'com.vector.cfg.groovy.extensions.
    AutomationTypeChecking')
def myMethod(){}
```

Listing 7.5: @TypeChecked with Automation API

#### 7.9.3.4 Gradle Maven publishing of an AutomationProject

The Gradle dvCfgAutomation automatically adds a Gradle MavenPublication instance for all signed Jar files. The publication is named signedJar, so the publish and publishToMavenLocal will publish the signed Jar files.

#### 7.9.3.5 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 7.6: 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.
- **useJarSignerDaemon** (Type `boolean`) - Enables or disables the usage of the Jar Signer Daemon process.
- **setJarSignerWaitTimeoutMin** (Type `int`) - Sets the timeout for the jar signer daemon (0 means infinite wait time).
- **includeDependenciesIntoJar** (Type `boolean`) - Enables or disables the inclusion of dependencies during build

**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 7.7: DaVinci Configurator build Gradle DSL API - useBswmdModel

**useJarSignerDaemon** The `useJarSignerDaemon` enables or disables the usage of the Jar Signer Daemon process. The process is spawned when a jar file shall be signed. This will speedup the build process especially when the project is built often. The daemon is closed automatically, when not used in a certain time span.

The default of `useJarSignerDaemon` is `true`.

The Gradle task `stopJarSignerDaemon` will stop any running Signer daemon.

```
dvCfgAutomation {  
    useJarSignerDaemon true  
}
```

Listing 7.8: DaVinci Configurator build Gradle DSL API - `useJarSignerDaemon`

**setJarSignerWaitTimeoutMin** The `setJarSignerWaitTimeoutMin` sets the timeout for the jar signer daemon (0 means infinite wait time). The default is set to 30 minutes, if the property is not configured. The `setJarSignerWaitTimeoutMin` only gets used if `useJarSignerDaemon` is `true`.

```
dvCfgAutomation {  
    setJarSignerWaitTimeoutMin 10  
}
```

Listing 7.9: DaVinci Configurator build Gradle DSL API - `setJarSignerWaitTimeoutMin`

**includeDependenciesIntoJar** The `includeDependenciesIntoJar` enables or disables bundling of gradle runtime dependencies (e.g. referenced jar files) into the resulting project jar. If `includeDependenciesIntoJar` is enabled the project jar file will contain all jar dependencies under the folder `jars` inside of the jar file.

The default of `includeDependenciesIntoJar` is `true`.

```
dvCfgAutomation {  
    includeDependenciesIntoJar false  
}
```

Listing 7.10: DaVinci Configurator build Gradle DSL API - `includeDependenciesIntoJar`

**generatorTests** The `generatorTests` allows to configure generator tests, see chapter 4.17.1.2 on page 318 for details.

# 8 AutomationInterface Changes between Versions

This chapter describes the supported functionality of different versions and all API changes between different MICROSAR releases.

## 8.1 Currently Supported Features

The table below contains a list of functionalities of the DaVinci Configurator Automation Interface.

**Legend:** A functionality is available if the **Since** column contains the DaVinci Configurator version (see Since). Otherwise the functionality is not yet available.

Component	Functionality	Since
Scripts	Loading, Execution, Script-Projects	5.13
	User defined Script Task Arguments in UI and Cmd	5.14 SP1
	Stateful Script Tasks	5.14
Project	Open, modify, save and close project	5.13
	Accessing the active UI project	5.13
	Create a new project	5.13
	Accessing/Modifying the target project settings	5.18
	Accessing/Modifying the useCase project settings	5.19
	Open ARXML files as Project without DPA	5.14
	Switch configuration phase (Post-build loadable)	
	Access to project search (FindView)	5.20
Model	Access to the whole AUTOSAR model (EcuC and System-Desc)	5.13
	Transaction support (Undo, Redo)	5.13
	Type save access to ECU model using definitions provided by the generated BswmdModel	5.13
	Post-build selectable support	5.13
	Access of variants, Access/Modification of variant data	5.13
	Post-build loadable support	5.13
	CE-States: UserDefined, Changeable, Deletable	5.13
	Consideration of pre-configuration status	5.13
	Access and modification of User Annotations at the configuration element	5.15
	Information and deletion of derived containers	5.16
	Unresolved references API	5.19
	Relative search for model element based on a root element	5.20
Generation	Generate code for specific modules	5.13
	Generate code for predefined code generation sequence	5.13
	Execute external generation steps	5.13
	External generation step creation and changes	
	Execute Custom workflow	
	Custom workflow creation and changes	

	Modify code generation sequence to enable/disable specific modules or generation steps	5.13
	SWC Templates and Contract Headers Generation	5.15
	Add a ScriptTask as external generation step	5.13
	Add a ScriptTask as custom workflow step	5.14
	Add generation report settings	5.20
Validation	Access to project validation result	5.13
	Access to validation results of specific model elements	5.13
	Solve validation results (by group, by id, by solving action type (preferred solving action))	5.13
Update Workflow	Updating a project	5.13
	Create Variant Configuration (EVS)	5.19
	Input file access and modification (non-variant)	5.13
	Input file access and modification (variant)	5.19
	Configuration of system description merge	
	Access to update report	
	Script execution after successful update workflow	5.18
	Access to the FilePreProcessing result file within the update workflow	5.18
	Access to every Autosar input file except of ProjectStandardConfiguration within the FilePreProcessing of the update workflow	5.18
	Update Workflow Settings	5.19
	Added Update Mode for Legacy Diagnostic only update	5.19
Reporting	Create predefined project reports	
	Create Custom Report	5.20
	Create Ecu Configuration report	5.18
Diff and Merge	Diff and Merge a Project (ActiveEcuC)	
	Diff and Merge a Project (SystemDescription)	
	Access to diff report	
	Auto-Merge	5.20
Persistency	Export of configuration artefacts	5.14
	Export of ActiveEcuc	5.14
	Export of Post-build selectable Variants per Variant	5.14
	Export of AUTOSAR model trees	5.15
	Export of Module Configurations	
	Import of configuration artefacts (Please inform Vector, if you need an import)	
	Import of Module Configurations	5.17
Testing	Unit test execution with Junit	5.13
	Unit test execution with Spock	5.13
	Generator Black-box tests	5.13
	Generator White-box tests	5.18
Domains		
Base	Nothing planned	
Communication	Access and modification of Can controller configuration	5.13
	Access and modification of Can filter masks	5.13
	Access and modification of CanBusTimings registers	
	Access and modification of FullCan flag of PDUs	5.20
	PDU and Channel abstraction	

Diagnostic	Access and modification of Diagnostic Data Identifier	
	Access and modification of Diagnostic Event Data	5.14
	Access and modification of Diagnostic Events	5.14
	Setup of event memory blocks	
	Access and modification of Production error handling	
I/O	Nothing planned	
Memory	Memory Domain Model Partitions, Memory blocks	
	FeeOptimization	
Mode Management	BSW Management	5.15
	API to provide the auto configuration (e.g. ECU state, module initialization, communication control, ...)	5.15
	API to configure logical expressions	
	API for custom configuration	
	Watchdogs: Access to the watchdogs settings and supervised entities	
	Initialization: Auto initialization and reset, Access to driver init lists	
Runtime System	Component Port Connection	5.14
	Data Mapping	5.14
	Task Mapping	5.16
	Component prototype creation	5.16
	Delegation port prototype creation	5.17
Communication Control	Nothing planned	

## 8.2 Changes in MICROSAR AR4-R24 - Cfg5.21

### 8.2.1 General

The Cfg5.21 (AR4-R24) automation interface is compatible to the Cfg5.20. So a script written with Cfg5.20 will also run in the Cfg5.21 version.

#### 8.2.1.1 Groovy

The used Groovy version was updated from 2.5.0 to 2.5.8, please see Groovy website for details. The new Groovy version does not have the groovy-all dependency anymore. If you use an extension API from Groovy you have to add the dependency manually in your `build.gradle` file.

Example:

```
compileOnly group: 'org.codehaus.groovy', name: 'groovy-xml', version: '2.5.8',
            classifier: 'indy'
```

Listing 8.1: Additional Groovy XML dependency

### 8.2.2 Project Creation TA Tool Suite workspace settings

New API added to customize TA Tool Suite workspace and executable settings for project creation. See chapter 4.5.5.8 on page 75 for details.

## 8.3 Changes in MICROSAR AR4-R23 - Cfg5.20

### 8.3.1 General

The Cfg5.20 (AR4-R23) automation interface is compatible to the Cfg5.19. So a script written with Cfg5.19 will also run in the Cfg5.20 version.

### 8.3.2 Automation Script Project

You have to migrate your project to the new compile bindings. Please execute the instructions in chapter 7.7 on page 375.

#### 8.3.2.1 Supported IntelliJ IDEA Version

The IntelliJ IDEA version 2019.1 was added to the supported versions. See 7.5.1 on page 369 for details.

### 8.3.3 Unit testing API

The dependency to the spock testing framework was incremented to version `org.spockframework:spock-core:1.3-groovy-2.5`. If you use spock, please update the version in your `build.gradle` file accordingly.

See chapter 4.16.2.2 on page 315 for more details.

### 8.3.4 Communication Domain

#### 8.3.4.1 FullCAN Flag of PDUs

Setting the CAN handle type (FullCAN/BasicCAN) is now possible using the communication domain api. See 4.10.1.3 on page 166 for more details.

### 8.3.5 Mode Management Domain

Introduced overloaded methods in BswM auto configuration API for multi partition use case (multiple BswMConfig containers). See chapter 4.10.3.1 on page 172 for details.

### 8.3.6 Runtime System Domain

#### 8.3.6.1 Changed Structure of Runtime System Domain Documentation

Since more and more different actions can be performed on the selections the structure of the documentation was changed. In older versions it was separated only in the different use cases. Now each selection is introduced in an own chapter, so that the complexity of the use case chapters could be reduced. This outlines also the fact that the selections can be used also only for filter elements and do some reporting without necessarily creating new elements.

### 8.3.6.2 Origin Context

- Introduced a completed flag for origin context ports and according completed() and notCompleted() predicates for the selection of origin component ports. See 4.10.4.8 on page 202.
- The origin context API offers new methods for creating new delegation ports (inside the closure of createFlatExtractDelegationPorts(Closure)). A default direction can be used now (useDefaultDirection()) which is the direction of the selected origin port and there is a possibility to derive the name (of the delegation port to be created) from the selected origin context (nameFromOriginPort(Closure)).

### 8.3.6.3 New Methods for Objects of Runtime System Domain

- IIIdentifiable.getAsrPath()
- IComponentPort.isCompleted().
- IComponentPort.getAsrPath().
- IComponentPort.getConnectedComponents().
- ICommunicationElement.getAsrPath().
- IPortInterface.getElementNames().
- IPortInterface.getElementsAsMdfObjects().
- IEVENT.getEventType()
- IEVENT.getTriggerCondition() - textual representation of the trigger condition, e.g. "10ms" for timing events with 10ms period, "MyPort.MyModeDeclrGroup.ModeA" for mode exit events or "MyPort.MyDataElement" for data reception triggers. See java-doc of IEVENT.getTriggerCondition() for more examples.
- IPositionInTaskEntry.getMappedTask()
- IPositionInTaskEntry.areTaskMappingsMapped() - Hint: When used in the order closure of the mapToTask call, the 'old' mapping state is returned, since task mapping operation is not finished until the whole mapToTask closure is done.
- IComponentPort.getComElementsFullExpanded() - this method can be used to get all communication elements for the data mapping with all parent elements to map to system signal groups / system signals and all children to map to group signals.
- IComponentPort.getComElementsFullExpandedExceptPrimitiveArrays() - see getComElementsFullExpanded(), the difference is that this method does not expand primitive arrays (e.g. array of uint8).
- IDataCommunicationElement.getLeafsFullExpanded() - corresponding method for IComponentPort.getComElementsFullExpanded() at the communication element.
- IDataCommunicationElement.getLeafsFullExpandedExceptPrimitiveArrays() - corresponding method for IComponentPort.getComElementsFullExpandedExceptPrimitiveArrays() at the communication element.
- IDataCommunicationElement.isComplex(), IDataCommunicationElement.isRecord() and IDataCommunicationElement.isArray().

- ISenderReceiverDataMapping.getLeafDataMappings() - this is the method you call if you need to know the mapped communication elements for all signals of a complex mapping. The difference to getChildDataMappings() is that getLeafDataMappings() analyzes the whole hierarchy (not only one level) of a data mapping and ignores all data mappings which are only necessary to define the structure (composite mappings).
- IAbstractSignalInstance.matches(ICommunicationElement, boolean) allows to compute a compatibility between a signal instance and a communication element (in the context of a data mapping, you probably know the compatibility already from the data mapping assistant in the GUI). This method is slower in terms of performance compared to using the communication element and signal instance selection API's compatibility evaluation methods.

#### 8.3.6.4 TaskMapping

- Task Mapping successor definition API allows now to sort the resulting task mapping without violating the defined successor relationships between the task mappings (see 4.10.4.16 on page 264).
- The successor definition API provides an additional slightly more complex example 4.10.4.16 on page 262.
- Documented priority of methods affecting the order of task mappings at the java-doc of ITaskMapper.defineSuccessors(Closure), ITaskMapper.order(Closure) and ITaskMapper.queue(). queue() may override the sequence defined in order(Closure) and order(Closure) may override defineSuccessors(Closure). In other words defineSuccessors(Closure) is executed before order(Closure) and order(Closure) is executed before queue().

#### 8.3.6.5 Unmapping Component Ports

Introduced a new API for unmapping component ports which removes connectors using the component port selection API. More details can be found at 4.10.4.10 on page 215.

#### 8.3.6.6 Removing Data Mappings

Removing data mappings is now possible using the communication element and the signal instance selection APIs. More details can be found at 4.10.4.13 on page 243.

#### 8.3.6.7 Select Elements by a Collection of Names

- Some of the selection APIs offer now new predicates which can handle not only one, but multiple names. The purpose is to support storing names in external files and make selecting elements by a collection of names easier. Below a complete list with new predicates of that type:
  - ICommunicationElementSelector: names, fullyQualifiedNames, ports, components
  - IComponentPortSelector: names, components, componentPortNames, originComponentPortNames
  - IComponentTypeSelector: names

- IEventSelector: names, components, moduleConfigurations
- IExecutableEntitySelector: names, components, moduleConfigurations
- IOrganicComponentPortSelector: names, components
- IPortInterfaceSelector: names, components, componentPorts
- ISignalInstanceSelector: names, asrPaths

### 8.3.6.8 Create Delegation Ports

- The use case of creating new delegation ports is now also supported at the selection of component ports (IComponentPortSelection). See example 4.10.4.15 on page 252. Direction and name of selected component port can be used for the new delegation port.

### 8.3.6.9 Create Selection Based On Existing Elements

- Most selection APIs offer a new put(List) method that allows to use elements that were created or selected in previous steps within the known selection API. See 4.10.4.21 on page 287 for an example.

### 8.3.6.10 Connect Ports

- Added new simple API for connecting multiple component ports at once. See 4.10.4.9 on page 212 for more details and examples.

### 8.3.6.11 Map Communication Elements to System Signals

- The simple API offers now a comfort function. If mapping only the root elements for a complex data mapping, the auto-mapper will try to match the children. See example at 4.10.4.12 on page 240.
- Added a new simple API for mapping multiple communication elements to system signals at once. See 4.10.4.12 on page 236 for more details and examples.
- Added a check to the simple API that makes sure the owner ports of the communication elements to be mapped are not terminated.
- Introduced the already from the GUI known compatibility for data mappings. That means the compatibility between a communication element and a signal instance. It is available by calling matches(...) method at the signal instance or using evaluateMatchesWithCompatibility(Closure) (see 4.10.4.12 on page 225 for examples) and confirmByCompatibility (see 4.10.4.12 on page 225 for examples) at the communication element and signal instance selection APIs. For more information about the compatibility see Javadoc of ECompatibility or 4.10.4.12 on page 232.

### 8.3.6.12 Select Communication Elements

- It is possible to use the communication element selection for selecting children of complex data communication elements (e.g. records, arrays) now. Calling ICommunicationElementSelector.selectFullyExpanded() or

`ICommunicationElementSelector.selectFullyExpandedButPrimitiveArraysAsLeafs()` will set the according behavior of the selection. See also 4.10.4.3 on page 188. This can be useful for example to run reports for record elements that are not mapped to system signals yet or do any other evaluations for unmapped communication element leafs.

### 8.3.6.13 Deleting Elements

- A new chapter was added to the documentation on how to delete objects. See 4.10.4.18 on page 280 for more details.

### 8.3.6.14 Variant Handling

- Added a new chapter about PostBuild variance for the runtime system domain. See 4.10.4.19 on page 285 for more details.

### 8.3.6.15 Performance

- Added some comments on how the performance may be optimized. For more details please read 4.10.4.21 on page 288.

## 8.3.7 Create Custom Report

- Create Custom Report API added see chapter 4.12.1 on page 294 for details.

## 8.3.8 Automatic Merge

The automatic merge API was added to the automation interface.

## 8.3.9 Post-build Selectable

### 8.3.9.1 Model Views

- Added a new getter `getAllPostBuildVariantViewsOrInvariant()` at the `IModelViewManager`. This getter can be used to write script code which is executed for projects with variance as well as for projects without variance. Use it to iterate over the model views. For projects with variants the method will return model views of all PostBuild predefined variants defined in the evaluated variant set. For projects with no variants it will return the `IInvariantView`.

## 8.3.10 mdfModel Api

Added a new method to the `mdfModel` API to retrieve `mdf` model elements with a relative path, see chapter 4.6.4.2 on page 103 for details.

### 8.3.11 Generation

#### 8.3.11.1 Report settings

The report settings for generation was added. See chapter 4.7.1.1 on page 128 for details.

### 8.3.12 Create Search Access

- The search provides the possibility to look for parameters, containers or module configurations by several criteria. 4.5.3 on page 63 for details.

## 8.4 Changes in MICROSAR AR4-R22 - Cfg5.19

### 8.4.1 General

The Cfg5.19 (AR4-R22) automation interface is compatible to the Cfg5.18. So a script written with Cfg5.18 will also run in the Cfg5.19 version.

### 8.4.2 Automation Script Project

You have to migrate your project to the new compile bindings. Please execute the instructions in chapter 7.7 on page 375.

#### 8.4.2.1 Bootstrap file

You have to update your `dvCfgAutomationBootstrap.gradle`. Please copy the file `dvCfgAutomationBootstrap.gradle` from

`<SIP>/DaVinciConfigurator/Core/AutomationInterface/_doc/templates/Gradle-Project`  
or  
`<SIP>/CLI/Core/AutomationInterface/_doc/templates/Gradle-Project`

into your Automation Script Project root folder.

#### 8.4.2.2 Supported IntelliJ IDEA Version

The IntelliJ IDEA versions 2018.3 and 2019.1 were added to the supported versions. See 7.5.1 on page 369 for details.

#### 8.4.2.3 Gradle

Support for Gradle version 5.2 added. You can now use Gradle 5.2 or earlier. The lowest required Gradle version is 4.0.1.

### 8.4.3 Unresolved Reference API

The `IUnresolvedReferenceApi` now provides interfaces for unresolved references.

Entry point is the `getUnresolvedReferences` method of the `IUnresolvedReferenceApiEntryPoint`.

### 8.4.4 Update Workflow Settings

Added update workflow settings.

Added Update Mode for Legacy Diagnostic only update.

- Select the Mode `LEGACY_DIAG_ONLY`

New workflow API added to update an existing project:

- Use method `updateProject()` to update a project

- Old method update() is deprecated

New API for "Selective Update"

- Use method selectiveUpdate(e.g. true) to enable the selective update

#### 8.4.5 Project Settings UseCase Api

Project Settings UseCase API added, see chapter 4.5.4 on page 64 for details.

#### 8.4.6 ECUC Unresolved Reference API

The `IEcucUnresolvedReferenceApi` now provides functionality to find and edit unresolved references of the active ECUC.

Entry point is the `getActiveEcuc` method of the `IEcucUnresolvedReferenceApiEntryPoint`.

The `select` and `selectChangeable` methods of the `IEcucUnresolvedReferenceApi` provides possibilities to filter the unresolved references by the predicates of the `IEcucUnresolvedReferenceSelector`.

The returned `IEcucUnresolvedReferenceSelection` and `IChangeableEcucUnresolvedReferenceSelection` provide getters and methods for edition.

## 8.5 Changes in MICROSAR AR4-R21 - Cfg5.18

### 8.5.1 General

The Cfg5.18 (AR4-R21) automation interface is compatible to the Cfg5.17. So a script written with Cfg5.17 will also run in the Cfg5.18 version.

### 8.5.2 Automation Script Project

You have to migrate your project to the new compile bindings. Please execute the instructions in chapter 7.7 on page 375.

#### 8.5.2.1 Groovy

The used Groovy version was updated from 2.4.12 to 2.5.0, please see Groovy website for details. The new Groovy version does not have the groovy-all dependency anymore. If you use an extension API from Groovy you have to add the dependency manually in your `build.gradle` file.

Example:

```
compileOnly group: 'org.codehaus.groovy', name: 'groovy-xml', version: '2.5.0',
classifier: 'indy'
```

Listing 8.2: Additional Groovy XML dependency

If you use the Spock framework in your automation project, you should change the dependency in your `build.gradle` as described in chapter 4.16.2.2 on page 315.

#### 8.5.2.2 Supported IntelliJ IDEA Version

The IntelliJ IDEA version 2018.2 was added to the supported versions. See 7.5.1 on page 369 for details.

The support for the IntelliJ IDEA versions prior to 2016.3 was removed. Please update your version.

#### 8.5.2.3 BuildSystem

**Gradle Build** DAVID00138752: Gradle Maven Publishing added, see chapter 7.9.3.4 on page 380 for details.

### 8.5.3 Unit testing API

**Spock-core** Spock-core version was updated to version 1.1. Please change the dependency in your `build.gradle` file, if you use it. See chapter 4.16.2.2 on page 315 for details.

### 8.5.4 Model TestInfrastructure

New class `ITransactionUndoAllRule` added which provides a JUnit rule to undo all transaction executed in a JUnit or Spock test case. See chapter 4.16.2.4 on page 316 for details.

### 8.5.5 Automation Project TestInfrastructure

New class `IProjectLoadRule` added which provides a JUnit rule to load Projects before test execution in a JUnit or Spock test case. See chapter 4.16.2.5 on page 317 for details.

### 8.5.6 Generator Testing

New API added to test generators with white-box JUnit or Spock tests. See chapter 4.17 on page 318 for details.

### 8.5.7 Model changes

These changes could break your existing client code, if you have used these interfaces or methods.

- Some methods have been changed or removed:
  - Interface `com.vector.cfg.model.mdf.ar4x.swcomponenttemplate.datatype.datatypes.MIDataTypeMap`
    - \* `MIIImplementationDataTypeARRef getImplementationDataType()` changed to  
`MIAbstractImplementationDataTypeARRef getImplementationDataType()`
    - \* `void setImplementationDataType(MIIImplementationDataTypeARRef)` changed to  
`void setImplementationDataType(MIAbstractImplementationDataTypeARRef)`
  - Interface `com.vector.cfg.model.mdf.ar4x.commonstructure.modedeclaration.MIModeRequestTypeMap`
    - \* `MIIImplementationDataTypeARRef getImplementationDataType()` changed to  
`MIAbstractImplementationDataTypeARRef getImplementationDataType()`
    - \* `void setImplementationDataType(MIIImplementationDataTypeARRef)` changed to  
`void setImplementationDataType(MIAbstractImplementationDataTypeARRef)`
  - Interface `com.vector.cfg.model.mdf.ar4x.commonstructure.datadefproperties.MISwDataDefPropsContent`
    - \* `MIIImplementationDataTypeARRef getImplementationDataType()` changed to  
`MIAbstractImplementationDataTypeARRef getImplementationDataType()`
    - \* `void setImplementationDataType(MIIImplementationDataTypeARRef)` changed to  
`void setImplementationDataType(MIAbstractImplementationDataTypeARRef)`

### 8.5.8 Workflow

New workflow hook: On successful update workflow

- full access to the project after successful update workflow

New workflow hook: On FilePreProcessing result

- access to the result file created in the update workflow by the FilePreProcessing

New workflow hook: On FilePreProcessing input file

- access to every Autosar input file except of ProjectStandardConfiguration files within the FilePreProcessing of the update workflow

### 8.5.9 Create Ecu Configuration Report

- Create Ecu Configuration Report API added see chapter 4.5.7 on page 77 for details.

### 8.5.10 Project Settings Target Api

Project Settings Target API added, see chapter 4.5.4 on page 64 for details.

### 8.5.11 Runtime System Domain

#### 8.5.11.1 Component Port Connection

- It is possible to use the origin context of a port as additional mapping criteria. Origin contexts are the ends of all incomplete connections in the structured extract for a corresponding port. See 4.10.4.9 on page 210.

#### 8.5.11.2 Task Mapping

- The event and the executable entity selection offer now a getTaskMapping method, which can be used for example to check the current task mapping for a task, without opening a transaction and without modifying any mappings. To make sure that the task mappings are up to date, the system description should be synchronized (see chapter about Model Synchronization). See 4.10.4.16 on page 274.
- A new way to specify the order by defining successor relationships between the task mappings was added. See 4.10.4.16 on page 260. The examples can be found at 4.10.4.16 on page 261.
- To make it easier to define successors a lot of new task mapping predicates were added. It is now possible to filter task mappings for attributes of their referenced events or the executable entities which they trigger. You might have a look at 4.10.4.16 on page 270.

#### 8.5.11.3 Origin Component Port

- Introduced a new model element called IOriginComponentPort which is an abstraction of a component port from the structured extract. The purpose of this abstraction is to

provide some data from the structured extract as additional information for the existing selection APIs. See 4.10.4.8 on page 201.

#### 8.5.11.4 Origin Context Selection

- A selection for ports of the structured extract (origin component ports) is introduced to the automation interface. The origin component ports can be used as additional information for the other APIs. For example the port interface of an origin context port can be used to create a new delegation port in flat extract. See 4.10.4.8 on page 201.
- A getter for origin component ports at the IComponentPort is now available (IComponentPort.getOriginComponentPorts).
- The selection API also offers a shortcut to create new delegation ports in the flat extract for the selected origin component ports. See 4.10.4.15 on page 253.

#### 8.5.11.5 Component Port Selection

- Added additional predicates to use the origin context component ports as additional selection criteria. See 4.10.4.1 on page 179.

## 8.6 Changes in MICROSAR AR4-R20 - Cfg5.17

### 8.6.1 General

The Cfg5.17 (AR4-R20) automation interface is compatible to the Cfg5.16. So a script written with Cfg5.16 will also run in the Cfg5.17 version.

### 8.6.2 Automation Script Project

You have to migrate your project to the new compile bindings. Please execute the instructions in chapter 7.7 on page 375.

#### 8.6.2.1 BuildSystem

**Gradle Build** DAVID00136153: Fixed issue where `includeDependenciesIntoJar = true` was not working. Now the jar dependencies are again included in the built script project.

#### 8.6.2.2 Supported IntelliJ IDEA Version

The IntelliJ IDEA version 2017.3 was added to the supported versions. See 7.5.1 on page 369 for details.

### 8.6.3 Persistency

#### 8.6.3.1 Model Module Import

The methods added to import module configurations from a given file into the current project:

- `IPersistencyModelImportApi.importModuleConfigurations(Path, Closure<?>)`
- `IPersistencyModelImportApi.importModuleConfigurations(List<Path>, Closure<?>)`
- `IPersistencyModelImportApi.importModuleConfigurations(Path)`
- `IPersistencyModelImportApi.importModuleConfigurations(List<Path>)` The methods added to specify an import mode and filter for the module import:
- `IImportModuleConfigurationApi.replaceInModel(IModuleApi, Closure<?>)`
- `IImportModuleConfigurationApi.addToModel(IModuleApi, Closure<?>)`
- `IModuleApi.module(DefRef)`
- `IModuleApi.module(List<?>)`
- `IModuleApi.module(AsrPath)`
- `IModuleApi.module(String)`

#### 8.6.3.2 Model Export

Support for `-exporterArgs` added, same as in the new command line feature.

## 8.6.4 BswmdModel

### 8.6.4.1 SIP DefRefs

The `IBswmdModelDefRefsApi` now provides new methods to check, if a module or any other definition exists in the loaded SIP:

- `IBswmdModelDefRefsApi.hasDefRef(String)` - returns `true`, if the passed definition exists.

## 8.6.5 Runtime System Domain

### 8.6.5.1 Component Port Selection

- Added predicates to the component port selector to filter component ports which are completed. See 4.10.4.1 on page 177.
- Added predicates to the component port selector to filter component ports which are terminated. See 4.10.4.1 on page 177.
- The component port offers now a getter method for the port interface of the port.

### 8.6.5.2 Communication Element Selection

- Added predicates to the communication element selector to filter for communication elements whose owner component ports are terminated. See 4.10.4.3 on page 187.

### 8.6.5.3 Create Delegation Ports

Delegation port prototypes can be created via AutomationAPI on the ECU Composition of the flat extract. Therefore a port interface selection with some predicates was added. See 4.10.4.15 on page 250.

### 8.6.5.4 Task Mapping

- A new 'queue' option was added which allows to keep the existing task mappings on the task at their current position. See 4.10.4.16 on page 275.
- Added new predicates to the event selection to filter for mode switch events by their mode activation kind (e.g. on mode entry or on mode exit) and published according method at the `IEvent` interface. See 4.10.4.5 on page 194.
- Setting the activation offset at the task mapping container is now possible using the event or the executable entity selection. See 4.10.4.16 on page 276 and 4.10.4.16 on page 276.

### 8.6.5.5 Simple API for connection between ports

Added a simple API to connect ports by using only the component and port names and setting an optional port interface mapping reference. See 4.10.4.9 on page 211.

### 8.6.5.6 Simple API for data mapping

Added another simple API to map communication elements to signals (and signal groups). The API requires the fully qualified names of the communication elements and the AUTOSAR paths to the signals and signal groups. See 4.10.4.12 on page 235.

### 8.6.5.7 Port Terminators

Added port terminators to the automation interface. Port terminators prevent open (unconnected and/or without data mapping) ports to produce validation messages. Ports can be terminated via the selection of component ports (also simple API) as well as the selection of communication elements (no simple API). See 4.10.4.11 on page 216 and 4.10.4.11 on page 219.

### 8.6.5.8 Data Mapping

Published an interface for TriggerToSignalMappings, which can be used for example to filter the new created mappings for TriggerToSignalMappings.

### 8.6.5.9 Service Proxy Components

Added predicates to filter component ports and component types for ServiceProxyComponent-Types. See 4.10.4.1 on page 178 and 4.10.4.4 on page 191.

### 8.6.5.10 Retrieve Short Name Paths and Fully Qualified Names

Added a chapter with examples on how to retrieve short name paths of port interface mappings, signals and signal groups and the fully qualified names of communication elements. See 4.10.4.20 on page 286.

### 8.6.5.11 More Examples

Added a few more examples to the documentation.

- How to select component ports which are missing a data mapping or connection.
- How to use the advanced filter for events when performing task mappings.
- Another example of ordering task mappings.
- Create a delegation port based on another port with using its name and port interface.

## 8.7 Changes in MICROSAR AR4-R19 - Cfg5.16

### 8.7.1 General

The Cfg5.16 (AR4-R19) automation interface is compatible to the Cfg5.15. So a script written with Cfg5.15 will also run in the Cfg5.16 version.

### 8.7.2 Automation Script Project

You have to migrate your project to the new compile bindings. Please execute the instructions in chapter 7.7 on page 375.

#### 8.7.2.1 Groovy

The used Groovy version was updated from 2.4.7 to 2.4.12, please see Groovy website for details.

#### 8.7.2.2 BuildSystem

**Gradle Build** The Gradle will now mark the Script inside of the DaVinci Configurator with an error, if there were any problems with the last Gradle execution. This shall help the script developers to find quickly the cause of the issue.

**Gradle version** The used Gradle version was updated from 3.0 to 4.0.1, please see Gradle website for details. Your automation script project must now use the Gradle version 4.0.1. Please install the new Gradle version, if you manually installed the old version.

The dependency configuration `compileDvCfg` was removed, please use the configuration `compileOnly` instead.

#### 8.7.2.3 Supported IntelliJ IDEA Version

The IntelliJ IDEA version 2017.2 was added to the supported versions. See 7.5.1 on page 369 for details.

### 8.7.3 ScriptAccess

New API added to retrieve the loaded scripts and call other script task inside of a script task. See 4.4.11 on page 60 for details.

### 8.7.4 UserInteraction - Progress Indication

The UserInteraction has now a new API to indicate progress to the User, by updating the text and the progress bar. Some long running operations like:

- Project Load
- Project Creation

- Transactions
- Update Workflow

now report the progress. See section 4.4.5.2 on page 47 for details.

### 8.7.5 Project Handling

Advanced API to open a Project added, which is not automatically closed, see section 4.5.6.2 on page 77 for details.

### 8.7.6 Model Automation API

#### 8.7.6.1 Derived Containers

New API added to retrieve information about derived containers and delete derived containers, see chapter 4.6.4.12 on page 113 for details.

#### 8.7.6.2 Variance API

Old Variance API deprecated and replaced by new names with PostBuild:

- `getAllVariantViews() => getAllPostBuildVariantViews()`
- `getInvariantValuesView() => getPostBuildInvariantValuesView()`
- `getInvariantEcucDefView() => getPostBuildInvariantEcucDefView()`
- `getAllVariantViewsOrInvariant() => getAllPostBuildVariantViewsOrInvariant()`
- `isValueInvariant() => isPostBuildValueInvariant()`
- etc.

#### 8.7.6.3 CE State

The methods `ICeStatePublished.isChangeable()` and `ICeStatePublished.isDeletable()` are now part of the published API.

#### 8.7.6.4 MDF Modification API

New method added `IMdfFeatureListHasDefinitionExtensions.byDefOrCreate(TypedDefRef<?, R, ?>)` which allows to update or create 0:1 or 1:1 parameter and container in a convenient manner.

### 8.7.7 Persistency

#### 8.7.7.1 Model Export

The methods added to export models into a specified file instead of a folder:

- `IPersistencyModelExportApi.exportModelTreeToFile(Object, MIObject, MIObject...)`

- `IPersistencyModelExportApi.exportActiveEcucToFile(Object)`
- `IPersistencyModelExportApi.IModelExporter.exportToFile(Object, Object...)`

**PreBuild Export API** The methods added to export PreBuild variant info into a folder:

- `IPersistencyModelExportApi.exportPreBuildVariants(Object)`
- `IModelExporter.exportAsPreBuildVariants(Object folder, Object... args)`

## 8.7.8 Generation

### 8.7.8.1 Generation Steps

Now the `IGenerationStep` class has a getter for the TargetType (`EEnvironmentTargetType`). See chapter 4.7.1.2 on page 131 for details.

## 8.7.9 Runtime System Domain

### 8.7.9.1 Component Port Selection

Automation API supports now trigger interfaces and trigger to signal mappings. See 4.10.4.1 on page 177 and 4.10.4.3 on page 187.

Added predicates to the component port selector to filter component ports for attributes of their component types. See 4.10.4.1 on page 177.

### 8.7.9.2 Signal Instance Selection

It is now possible to filter SystemSignals with new predicates for PhysicalChannels, CommunicationClusters, Pdus and Frames. See 4.10.4.2 on page 183.

### 8.7.9.3 Bridge between mdf and model abstractions

Introduced a bridge to navigate between mdf model objects and model abstraction objects of the runtime system domain. See 4.10.4.17 on page 279.

### 8.7.9.4 Create Component Prototypes

Component prototypes can be created via AutomationAPI. Therefore a component type selection with some predicates was added. See 4.10.4.14 on page 248.

### 8.7.9.5 Task Mapping

The task mapping can be done using the AutomationAPI now. Two entry points were added. An event selection and an executable entity selection. After that it is possible to select a task and to customize the task mapping by e.g. map all events of a runnable to the same position or applying an execution order constraint. See 4.10.4.16 on page 255.

## 8.8 Changes in MICROSAR AR4-R18 - Cfg5.15

### 8.8.1 General

The Cfg5.15 (AR4-R18) automation interface is mostly compatible to the Cfg5.14. So a script written with Cfg5.14 will also run in the Cfg5.15 version.

### 8.8.2 Automation Script Project

You have to migrate your project to the new compile bindings. Please execute the instructions in chapter 7.7 on page 375.

#### 8.8.2.1 Supported IntelliJ IDEA Version

The IntelliJ IDEA version 2016.3 was added to the supported versions. See 7.5.1 on page 369 for details.

#### 8.8.2.2 BuildSystem

**Groovy - Static compilation** The AutomationInterface Groovy compiler extension added. This allows you to use Automation API in static compiled Groovy code. See 7.9.3.3 on page 379.

**includeDependenciesIntoJar** The `includeDependenciesIntoJar` Gradle build setting added. See 7.9.3.5 on page 380 for details. The Gradle build will now automatically include jar dependencies into your project jar.

### 8.8.3 Script Execution

#### 8.8.3.1 User defined arguments

The ScriptTask user defined arguments now support validators to validate the input before executing the task, like checking if the file exists. This provides fast user feedback. See 4.4.9.1 on page 55 for details.

### 8.8.4 Project Handling

New API added to create empty raw AUTOSAR model projects, see chapter 4.5.10.1 on page 83 for details.

### 8.8.5 Project Creation vVIRTUALtarget settings

New API added to customize vVIRTUALtarget project and executable settings for project creation. See chapter 4.5.5.7 on page 73 for details.

### 8.8.6 Model changes

These changes could break your existing client code, if you have used these interfaces or methods.

- Some interfaces have been renamed or moved:
  - Interface `MIMcFunctionDataRefSet` moved
    - \* from package `com.vector.cfg.model.mdf.ar4x.commonstructure.measurementcalibrationsupport`  
to package `com.vector.cfg.model.mdf.ar4x.commonstructure.measurementcalibrationsupport.rptsupport`
  - Interface `MIMcFunctionDataRefSetConditional` moved
    - \* from package `com.vector.cfg.model.mdf.ar4x.commonstructure.measurementcalibrationsupport`  
to package `com.vector.cfg.model.mdf.ar4x.commonstructure.measurementcalibrationsupport.rptsupport`
  - Interface `MIMcFunctionDataRefSetContent` moved
    - \* from package `com.vector.cfg.model.mdf.ar4x.commonstructure.measurementcalibrationsupport`  
to package `com.vector.cfg.model.mdf.ar4x.commonstructure.measurementcalibrationsupport.rptsupport`
  - Interface `MIFT` moved
    - \* from package `com.vector.cfg.model.mdf.model.autosar.commonpatterns.textmodel.languagedatamodel.specializedloverviewparagraph`  
to package `com.vector.cfg.model.mdf.model.autosar.commonpatterns.textmodel.singlelanguagedata.specializedsloverviewparagraph`
  - Interface `MIFT` moved
    - \* from package `com.vector.cfg.model.mdf.model.autosar.commonpatterns.textmodel.languagedatamodel.specializedlparagraph`  
to package `com.vector.cfg.model.mdf.model.autosar.commonpatterns.textmodel.singlelanguagedata.specializeds1paragraph`
- Some methods have been changed or removed:
  - Interface `com.vector.cfg.model.mdf.ar4x.diagnosticextract.dcm.diagnosticservice.databyidentifier.MIDiagnosticDataByIdentifier`
    - \* `MIDiagnosticDataIdentifierARRef getDataIdentifier()`  
changed to  
`MIDiagnosticAbstractDataIdentifierARRef getDataIdentifier()`
    - \* `void setDataIdentifier(MIDiagnosticDataIdentifierARRef)`  
changed to  
`void setDataIdentifier(MIDiagnosticAbstractDataIdentifierARRef)`
  - Some `...Owner()` methods were removed. The usage of these methods is not recommended. Instead use the `MIObject.miImmediateComposite()` method.

## 8.8.7 Model Automation API

### 8.8.7.1 IVarianceApi

New method `IVarianceApi.getAllVariantViewsOrInvariant()` added.

### 8.8.7.2 Access methods

New access methods for the `EcuConfigurationAccess` and `EcucDefinitionAccess` added. See chapter 4.6.4.10 on page 111 for details.

New MDF access method added `mdfModel(String)`. This method tries to resolve a model element by testing multiple ways. See chapter 4.6.4.2 on page 102 details.

### 8.8.7.3 Reverse Reference Resolution - ReferencesPointingToMe

New methods to query references starting from reference targets added. See chapter 4.6.4.11 on page 112 for details.

### 8.8.7.4 Operations

New method `setConfigurationVariantOfAllModuleConfigurations()` added to `IOperations` class. See chapter 4.6.6.2 on page 120 for details.

New method `createUniqueMappedAutosarPackage()` added to `IOperations` class. See chapter 4.6.6.2 on page 120 for details.

### 8.8.7.5 User Annotations

New API to access and modify User Annotations was added. See chapter 4.6.9.1 on page 125 for details.

### 8.8.7.6 Variance

New method `variance.variantView(String name)` added to retrieve a variant view by name.

### 8.8.7.7 Model Synchronization

New API added to execution the new Model Synchronization operation. See chapter 4.6.7 on page 121 for details.

## 8.8.8 Persistency

New Persistency model exporter added `exportModelTree()`. See chapter 4.13 on page 296 for details.

### 8.8.9 Workflow

New workflow API added to configure settings with `updateSettings{}`:

- Select the update mode (`ECUC_ONLY`, `ECUC_AND_DEVELOPER_WORKSPACE`)
- Parameter `uuidUsageInStandardConfigurationEnabled`
- Parameter `uuidUsageInSystemDescriptionEnabled`

### 8.8.10 Validation

#### 8.8.10.1 Validation-Result Access Methods

New two new methods added to retrieve validation by model object in a recursive manner like the editors.

- `MIObject.getValidationResultsRecursive()`
- `IViewedModelObject.getValidationResultsRecursive()`

### 8.8.11 Generation

#### 8.8.11.1 SWC Templates and Contract Headers Generation

The SWC Templates and Contract Headers Generation (Swct) automation API was added, see chapter 4.7.3 on page 135 for details.

### 8.8.12 BswmdModel

#### 8.8.12.1 BswmdModel Groovy

Two new methods added to access the BswmdModel by MDF model objects in a generic way, without knowing a `DefRef`. This is handy, if you want traverse an unknown Ecu configuration structure.

- `GIContainer bswmdModel(MIContainer)`
- `GIModuleConfiguration bswmdModel(MIModuleConfiguration)` Both methods return the base bswmd model types for the corresponding MDF model objects.

New methods added to access BswmdModel elements by path and or by Type:

- `List bswmdModel(Class)`
- `List bswmdModel(Class, Closure)`
- `List bswmdModel(Class, String)`
- `List bswmdModel(Class, String, Closure)`

### 8.8.12.2 DerivativeMapping

Until R17 modules with DerivativeMapping were ignored from the DaVinciConfigurator and no BswmdModel classes were generated for these modules. Just the corresponding AsrXxx (e.g. AsrOs) model classes were included in the BswmdModel. Now the BswmdModel classes for these modules are generated for one certain derivative.

By default, the first derivative is selected, sorted by UUID. The AsrXxx usages have to be replaced by the actual module in the scripts. See 5.3.2.1 on page 349 for more details.

### 8.8.13 Mode Management Domain

Introduced BswM auto configuration API for automatically creating dedicated parts of the BswM configuration. See chapter 4.10.3.1 on page 172 for details.

### 8.8.14 Runtime System Domain

#### 8.8.14.1 Data Mapping

'autoMapTo' allows control now about the handling of nested arrays of primitive. See 4.10.4.12 on page 222 and 4.10.4.12 on page 228.

## 8.9 Changes in MICROSAR AR4-R17 - Cfg5.14

### 8.9.1 General

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

### 8.9.2 Script Execution

#### 8.9.2.1 Stateful Script Tasks

A new API was added to support cache and retrieve data over multiple script task executions. See 4.4.10 on page 58 for more details.

### 8.9.3 Automation Script Project

You have to migrate your project to the new compile bindings. Please execute the instructions in chapter 7.7 on page 375.

#### 8.9.3.1 Groovy

The used Groovy version was updated from 2.4.5 to 2.4.7, please see Groovy website for details.

#### 8.9.3.2 Supported IntelliJ IDEA Version

The IntelliJ IDEA version 2016.2 was added to the supported versions. See 7.5.1 on page 369 for details.

#### 8.9.3.3 BuildSystem

**Gradle** The used default Gradle version was updated from 2.13 to 3.0, please see Gradle website for details.

**useJarSignDaemon** The `useJarSignDaemon` Gradle build setting added. See 7.9.3.5 on page 380 for details.

### 8.9.4 Converter Refactoring

The converters previously provided by `com.vector.cfg.automation.api.Converters` have been moved to the new `com.vector.cfg.automation.scripting.api.ScriptConverters` and `com.vector.cfg.model.groovy.api.ModelConverters`.

### 8.9.5 UserInteraction

UserInteraction API added to show messages to the user, see 4.4.5.1 on page 46.

## 8.9.6 Project Load

### 8.9.6.1 AUTOSAR Arxml Files

New API added to open AUTOSAR `arxml` files as a temporary project. See chapter 4.5.10 on page 82 for details.

## 8.9.7 Model

**Script Tasks Types** The existing script task type `DV_MODULE_ACTIVATION` renamed to the new name `DV_ON_MODULE_ACTIVATION`.

A new `DV_ON_MODULE_DEACTIVATION` task type added, which is execution when a module configuration is deleted.

### 8.9.7.1 Transactions

A new `ITransactionsApi` added which provide access to the `transactionHistory` and API to retrieve information of running transactions. A new method `transactions.isTransactionRunning()` added.

The `ITransactionHistoryApi` was moved to the new `ITransactionsApi`. The access to the history is now `transactions.transactionHistory{}`.

**Operations** The new operations added:

- `deactivateModuleConfiguration()` to delete a module configuration
- `activateModuleConfiguration(DefRef, String shortName)` to activate a module configuration with the specified short name
- `createModelObject(Class<T>)` to create arbitrary MDF model objects
- `parameter.setUserDefined(boolean)` method added to set and reset the user defined flag

### 8.9.7.2 MDF Model Read and Write

The whole MDF model API was changed from the old `mdfRead()` and `mdfWrite()` to one method `mdfModel()` with explicit write/create methods. You have to change all your `mdfRead()` and `mdfWrite()` calls to `mdfModel()`. And every `mdfWrite()` closure the implicit creation to explicit create calls.

This was necessary due to the fact that the old implicit API leads to surprising results, when methods are called, which use the read API, but called in a write context. So the method would yield different results, when called in different contexts.

The new MDF model API will never create any elements implicitly. Now there are explicit create methods, like in the `BswmdModel`:

- For 0:1 elements: `get<Element>OrCreate()` method
- For 0:/\* elements: `list.createAndAdd()` and `byNameOrCreate()` methods

The write context is not needed anymore, but you have to open a `transaction()` before calling any write API.

See the chapter 4.6.4.1 on page 99 for the read API and 4.6.4.3 on page 104 for the write API.

### 8.9.7.3 SystemDescription Access

The SystemDescription Access API added to retrieve paths to elements like flat map, flat extract and the corresponding model elements. See chapter 4.6.5 on page 117 for details.

### 8.9.7.4 ActiveEcuc

The class `IActiveEcuc` was renamed to `IActiveEcucApi` to reflect that it is not the active ecuc element, but the API of the active ecuc.

## 8.9.8 Persistency

New Persistency API added to import and export model data. See chapter 4.13 on page 296 for details.

## 8.9.9 Generation

The generation script tasks `DV_GENERATION_ON_START` and `DV_GENERATION_ON_END` renamed to `DV_ON_GENERATION_START` and `DV_ON_GENERATION_END`.

The new script task type `DV_CUSTOM_WORKFLOW_STEP` added to execute tasks in the custom workflow. See 4.3.1.4 on page 36 for details.

The return type of validation and generation methods has changed to `IGenerationResultModel`. This type provides more detailed information about the executed steps.

## 8.9.10 BswmdModel

### 8.9.10.1 Writing with BswmdModel

The BswmdModel supports now a write access for ecuc configuration elements. This means new elements can be created and existing elements can be modified and deleted by the BswmdModel. See 5.3.1.9 on page 345 for more details.

## 8.9.11 BswmdModel Groovy

**bswmdModelRead** The BswmdModel access was changed from the old `bswmdModelRead()` to the new `bswmdModel()` method. This was done to support the new write access.

**Domain Object Navigation** The BswmdModel API now support the navigation from domain model to the BswmdModel. See 4.6.3.7 on page 98.

### 8.9.12 Diagnostics Domain

Introduced new API which allows creation and querying of diagnostic events. Also OBD and J1939 state of the configuration can be queried.

### 8.9.13 Communication Domain

Communication Domain API moved from  
`com.vector.cfg.dom.com.model.groovy` into  
`com.vector.cfg.dom.com.groovy.api`.

Can Controller classes moved from  
`com.vector.cfg.dom.com.model.groovy.can` into  
`com.vector.cfg.dom.com.groovy.can`.

### 8.9.14 Runtime System Domain

Runtime System API `IRuntimeSystemApi` now provides functionality to map ports and system signals.

Entry points are the `selectComponentPorts`, `selectSignalInstances` and `selectCommunicationElements` methods.

## 8.10 Changes in MICROSAR AR4-R16 - Cfg5.13

### 8.10.1 General

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

### 8.10.2 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.

### 8.10.3 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

2.1	Script Samples location . . . . .	15
2.2	Script Locations View . . . . .	15
2.3	Script Tasks View . . . . .	15
2.4	Create New Script Project... Button . . . . .	16
2.5	Project Settings . . . . .	17
2.6	Project Build . . . . .	18
2.7	Project SDK Setting . . . . .	19
2.8	Gradle JVM Setting . . . . .	20
3.1	DaVinci Configurator components and interaction with scripts . . . . .	21
3.2	Structure of scripts and script tasks . . . . .	23
4.1	The API overview and containment structure . . . . .	28
4.2	IScriptTaskType interfaces . . . . .	33
4.3	ScriptTaskType input parameters in code closure . . . . .	34
4.4	Access Editor Selection . . . . .	35
4.5	Script Task Execution Sequence . . . . .	41
4.6	ScriptingException and sub types . . . . .	49
4.7	Search for active project in getActiveProject() . . . . .	62
4.8	SearchApi Exception Message . . . . .	64
4.9	Report Output Path . . . . .	128
4.10	example situation with the GUI . . . . .	138
4.11	The structure of a merge result . . . . .	307
5.1	ECUC container type inheritance . . . . .	324
5.2	MIOBJECT class hierarchy and base interfaces . . . . .	325
5.3	Autosar package containment . . . . .	325
5.4	The ECUC container definition reference . . . . .	326
5.5	Invariant views hierarchy . . . . .	332
5.6	Example of a model structure and the visibility of the IIInvariantValuesView . . . . .	333
5.7	Variant specific change of a parameter value . . . . .	336
5.8	Variant common change of a parameter value . . . . .	337
5.9	The relationship between the MDF model and the BswmdModel . . . . .	338
5.10	SubContainer DefRef navigation methods . . . . .	341
5.11	Untyped reference interfaces in the BswmdModel . . . . .	342
5.12	Creating a BswmdModel in the Post-build selectable use case . . . . .	343
5.13	Class and Interface Structure of the BswmdModel . . . . .	345
5.14	DefRef class structure . . . . .	351
5.15	IParameterStatePublished class structure . . . . .	354
5.16	IContainerStatePublished class structure . . . . .	355
6.1	Beta API Annotation . . . . .	366
6.2	Add beta API usage - build.gradle . . . . .	367
7.1	No JavaDoc . . . . .	370
7.2	No JavaDoc . . . . .	370
7.3	JavaDoc . . . . .	371
7.4	Project Build . . . . .	371
7.5	Project Continuous Build . . . . .	371

---

## Figures

7.6 Stop Continuous Build . . . . .	372
7.7 Disconnect from Continuous Build Process . . . . .	372
7.8 Project Debug . . . . .	372
7.9 Stop Debug Session . . . . .	373
7.10 Disconnect from Debug Process . . . . .	373
7.11 Activate the auto-import . . . . .	379

# Tables

5.1 Different Class types in different models . . . . .	339
---	-----

# Listings

3.1	Static field memory leak . . . . .	26
3.2	Memory leak with closure variable . . . . .	27
4.1	Task creation with default type . . . . .	29
4.2	Task creation with TaskType Application . . . . .	30
4.3	Task creation with TaskType Project . . . . .	30
4.4	Define two tasks in one script . . . . .	30
4.5	Script creation with IDE support . . . . .	30
4.6	Task with isExecutableIf . . . . .	31
4.7	Script with description . . . . .	31
4.8	Task with description . . . . .	32
4.9	Task with description and help text . . . . .	32
4.10	Usage of ScriptTaskType: DV_EDITOR_MULTI_SELECTION . . . . .	36
4.11	Access automation API in Groovy clients by the IScriptExecutionContext . . . . .	39
4.12	Access to automation API in Java clients by the IScriptExecutionContext . . . . .	40
4.13	Script task code block arguments . . . . .	40
4.14	Resolves a path with the resolvePath() method . . . . .	42
4.15	Resolves a path with the resolvePath() method . . . . .	43
4.16	Resolves a path with the resolveScriptPath() method . . . . .	43
4.17	Resolves a path with the resolveProjectPath() method . . . . .	44
4.18	Resolves a path with the resolveSipPath() method . . . . .	44
4.19	Resolves a path with the resolveTempPath() method . . . . .	44
4.20	Get the project output folder path . . . . .	45
4.21	Get the SIP folder path . . . . .	45
4.22	Usage of the script logger . . . . .	46
4.23	Usage of the script logger with message formatting . . . . .	46
4.24	Usage of the script logger with Groovy GString message formatting . . . . .	46
4.25	UserInteraction from a script . . . . .	47
4.26	Display progress to the user . . . . .	47
4.27	Display progress to the user nested . . . . .	48
4.28	Display progress to the user with progress bar work . . . . .	48
4.29	Stop script task execution by throwing an ScriptClientExecutionException . . . . .	49
4.30	Changing the return code of the console application by throwing an ScriptClientExecutionException . . . . .	50
4.31	Using your own defined method . . . . .	51
4.32	Using your own defined class . . . . .	51
4.33	Using your own defined method with a daVinci block . . . . .	51
4.34	ScriptApi.scriptCode{} usage in own method . . . . .	52
4.35	ScriptApi.scriptCode() usage in own method . . . . .	52
4.36	ScriptApi.activeProject{} usage in own method . . . . .	53
4.37	ScriptApi.activeProject() usage in own method . . . . .	53
4.38	Script task UserDefined argument with no value . . . . .	54
4.39	Define and use script task user defined arguments from commandline . . . . .	54
4.40	Script task UserDefined argument with default value . . . . .	54
4.41	Script task UserDefined argument with multiple values . . . . .	55
4.42	Script task UserDefined argument with predefined validator . . . . .	55
4.43	Script task UserDefined argument with own validator . . . . .	56

4.44	executionData - Cache and retrieve data during one script task execution . . . . .	58
4.45	sessionData - Cache and retrieve data over multiple script task executions . . . . .	58
4.46	sessionData and executionData syntax samples . . . . .	59
4.47	Call another script task from a script task . . . . .	60
4.48	Call another script task with arguments . . . . .	60
4.49	Accessing IProjectHandlingApi as a property . . . . .	61
4.50	Accessing IProjectHandlingApi in a scope-like way . . . . .	61
4.51	Switch the active project . . . . .	62
4.52	Accessing the active IProject . . . . .	63
4.53	Using the search API . . . . .	63
4.54	Access and modify Project Settings - Variant 1 . . . . .	65
4.55	Access and modify Use Project Settings UseCases . . . . .	66
4.56	Creating a new project (mandatory parameters only) . . . . .	67
4.57	Creating a new project (with some optional parameters) . . . . .	68
4.58	Creating a new project with custom VTT settings . . . . .	74
4.59	Creating a new TA Tool Suite workspace . . . . .	75
4.60	Opening a project from .dpa file . . . . .	76
4.61	Parameterizing the project open procedure . . . . .	76
4.62	Create Ecu Configuration Report with default settings . . . . .	78
4.63	Create Ecu Configuration Report . . . . .	78
4.64	Minimal Example of Create Support Request Package . . . . .	79
4.65	Create Support Request Package with all project informations . . . . .	80
4.66	Create Support Request Package with individual project informations . . . . .	80
4.67	Opening, modifying and saving a project . . . . .	81
4.68	Opening Arxml files as project . . . . .	82
4.69	Create an empty AUTOSAR model . . . . .	83
4.70	Read with BswmdModel objects starting with a module DefRef (no type declaration) . . . . .	85
4.71	Read with BswmdModel objects starting with a module class (strong typing) . .	85
4.72	Read with BswmdModel objects with closure argument . . . . .	86
4.73	Read with BswmdModel object for an MDF model object . . . . .	86
4.74	Read with BswmdModel objects with the untyped model (DefRefAPI) . . . . .	87
4.75	Write with BswmdModel required/optional objects . . . . .	88
4.76	Write with BswmdModel multiple objects . . . . .	89
4.77	Write with BswmdModel - Duplicate a container . . . . .	89
4.78	Write with BswmdModel - Delete elements . . . . .	90
4.79	Read system description starting with an AUTOSAR path in closure . . . . .	91
4.80	Read system description starting with an AUTOSAR path in property style . .	92
4.81	Changing a simple property of an MIVariableDataPrototype . . . . .	92
4.82	Creating non-existing member by navigating into its content with OrCreate() .	92
4.83	Creating new members of child lists with createAndAdd() by type . . . . .	93
4.84	Updating existing members of child lists with byNameOrCreate() by type . .	93
4.85	BswmdModel usage with import . . . . .	94
4.86	Read with BswmdModel the EcuC module configuration . . . . .	95
4.87	Read with BswmdModel the EcuC module configuration with DefRef . . . . .	95
4.88	Write with BswmdModel the EcucGeneral container . . . . .	95
4.89	Usage of the sipDefRef API to retrieve DefRefs in script files . . . . .	96
4.90	Check if a definition exists in the SIP . . . . .	96
4.91	Usage of generated DefRefs form the bswmd model . . . . .	97
4.92	Usage of the untyped BswmdModel with SipDefRefs . . . . .	98
4.93	Switch from a domain model object to the corresponding BswmdModel object .	98

4.94 Navigate into an MDF object starting with an AUTOSAR path . . . . .	100
4.95 Find an MDF object and retrieve some content data . . . . .	100
4.96 Navigating deeply into an MDF object with nested closures . . . . .	101
4.97 Ignoring non-existing member closures . . . . .	101
4.98 Get a MIReferrable child object by name . . . . .	101
4.99 Retrieve child from list with byName() . . . . .	102
4.100Get elements with mdfModel(String) . . . . .	103
4.101Read definitions elements with a relative path using the mdfModel . . . . .	103
4.102Read activeEcuc elements with a relative path using the mdfModel . . . . .	104
4.103Changing a simple property of an MIVariableDataPrototype . . . . .	105
4.104Creating non-existing member by navigating into its content with OrCreate() . .	105
4.105Creating child member by navigating into its content with OrCreate() with type	105
4.106Creating new members of child lists with createAndAdd() by type . . . . .	106
4.107Updating existing members of child lists with byNameOrCreate() by type . . .	108
4.108Delete a parameter instance . . . . .	109
4.109Check is a model instance is deleted . . . . .	109
4.110Duplicates a container under the same parent . . . . .	110
4.111Get the AsrPath of an MIReferrable instance . . . . .	110
4.112Get the AsrObjectLink of an AUTOSAR model instance . . . . .	110
4.113Get the DefRef of an Ecuc model instance . . . . .	110
4.114Set the DefRef of an Ecuc model instance . . . . .	111
4.115Get the CeState of an Ecuc parameter instance . . . . .	111
4.116Retrieve the user-defined flag of an Ecuc parameter in Groovy . . . . .	111
4.117Set an Ecuc parameter instance to user defined . . . . .	111
4.118Get the IEcucDefinition of an Ecuc model instance . . . . .	112
4.119Get the IEcucHasDefinition of an Ecuc model instance . . . . .	112
4.120referencesPointingToMe sample . . . . .	112
4.121systemDescriptionObjectsPointingToMe sample . . . . .	112
4.122Derived Container API access . . . . .	113
4.123Delete a derived container unconditionally . . . . .	113
4.124Get the AUTOSAR root object . . . . .	113
4.125Get the active Ecuc and all module configurations . . . . .	114
4.126Iterate over all module configurations . . . . .	114
4.127Get module configurations by definition . . . . .	114
4.128Get subContainers and parameters by definition . . . . .	114
4.129Check parameter values . . . . .	115
4.130Get integer parameter value . . . . .	116
4.131Get reference parameter value . . . . .	117
4.132Get the FlatExtract and FlatMap paths by the SystemDescription API . . . .	117
4.133Get FlatExtract instance by the SystemDescription API . . . . .	117
4.134Execute a transaction . . . . .	118
4.135Execute a transaction with a name . . . . .	118
4.136Check if a transaction is running . . . . .	119
4.137Undo a transaction with the transactionHistory . . . . .	119
4.138Redo a transaction with the transactionHistory . . . . .	120
4.139Activation of the ModuleConfiguration Dio . . . . .	120
4.140Model synchronization inside an open project . . . . .	122
4.141Retrieve and use a variant view by name . . . . .	122
4.142The default view is the IPostBuildInvariantValuesView . . . . .	123
4.143Execute code in a model view . . . . .	124
4.144Get a UserAnnotation of a container . . . . .	125

4.145Create a new UserAnnotation . . . . .	126
4.146Create or get the existing UserAnnotation by label name . . . . .	126
4.147Basic structure . . . . .	127
4.148Validate with default project settings . . . . .	128
4.149Generate with standard project settings . . . . .	128
4.150Generation of components with a result report . . . . .	129
4.151Generate one module . . . . .	129
4.152Generate one module . . . . .	130
4.153Generate two modules . . . . .	130
4.154Generate one module with two configurations . . . . .	131
4.155Execute an external generation step . . . . .	131
4.156Retrieval of the TargetType of a Generation Step . . . . .	132
4.157Evaluate the generation result . . . . .	132
4.158Use a script task as generation step during generation . . . . .	133
4.159Use a script task as custom workflow step . . . . .	134
4.160Hook into the GenerationProcess at the start with script task . . . . .	134
4.161Hook into the GenerationProcess at the end with script task . . . . .	134
4.162Basic Swct structure . . . . .	135
4.163SWC Templates and Contract Headers generation with standard project settings	135
4.164SWC Templates and Contract Headers generation of all components . . . . .	136
4.165SWC Templates and Contract Headers generation of one selected component . .	136
4.166Swct generation get component and select component . . . . .	136
4.167Swct generation of multiple components . . . . .	137
4.168Access all validation-results and filter them by ID . . . . .	139
4.169Solve a single validation-result with a particular solving-action . . . . .	140
4.170Fast solve multiple results within one transaction . . . . .	141
4.171Solve all validation-results with its preferred solving-action (if available) . . .	141
4.172Access all validation-results of a particular object . . . . .	142
4.173Access all validation-results of a particular DefRef . . . . .	143
4.174Filter validation-results using an ID constant . . . . .	143
4.175Fast solve multiple validation-results within one transaction using a solving-action-group-ID . . . . .	144
4.176IValidationResultUI overview . . . . .	145
4.177IValidationResultUI in a variant (post build selectable) project . . . . .	145
4.178CE is affected by (matches) an IValidationResultUI . . . . .	146
4.179Advanced use case - Retrieve Erroneous CEs with descriptors of an IValidationResultUI . . . . .	147
4.180Examine an ISolvingActionSummaryResult . . . . .	148
4.181Create a ValidationResult . . . . .	149
4.182Report a ValidationResult when MD license option is available . . . . .	149
4.183Turn off auto solving action execution . . . . .	150
4.184>Selecting the according file set" . . . . .	152
4.185Access and modify the Workflow Update settings . . . . .	154
4.186Access and modify the update report settings . . . . .	155
4.187Enable the selective update . . . . .	156
4.188>"Update existing project" . . . . .	157
4.189Exchange all input files and start update . . . . .	157
4.190Set diag data as input file . . . . .	158
4.191Change list of communication extracts and update . . . . .	159
4.192Executes the file preprocessing without updating the configuration . . . . .	160
4.193Accessing the API for creating variants . . . . .	161

4.194Accessing IDomainApi as a property . . . . .	163
4.195Accessing IDomainApi in a scope-like way . . . . .	163
4.196Accessing ICommunicationApi as a property . . . . .	163
4.197Accessing ICommunicationApi in a scope-like way . . . . .	164
4.198Optimizing Can Acceptance Filters . . . . .	165
4.199Enable FullCAN feature for PDU . . . . .	168
4.200Disable FullCAN feature for all PDUs of CanController . . . . .	168
4.201Accessing IDiagnosticsApi as a property . . . . .	169
4.202Accessing IDiagnosticsApi in a scope-like manner . . . . .	169
4.203Create a new UDS DTC with event . . . . .	170
4.204Enable OBD II and create a new OBD related DTC with event . . . . .	170
4.205Enable WWH-OBD and create a new OBD related DTC with event . . . . .	171
4.206Open a project, enable J1939 and create a new J1939 DTC with event . . . . .	171
4.207Accessing IModeManagementApi as a property . . . . .	172
4.208Accessing IModeManagementApi in a scope-like way . . . . .	172
4.209ECU State Handling Auto Configuration . . . . .	173
4.210Inspecting Auto Configuration Elements . . . . .	175
4.211Accessing IRuntimeSystemApi as a property . . . . .	176
4.212Accessing IRuntimeSystemApi in a scope-like way . . . . .	176
4.213Selects all component ports . . . . .	180
4.214Selects all unconnected component ports . . . . .	180
4.215Select all unconnected sender/receiver or connected mode-switch component ports	181
4.216Selects not completed component ports . . . . .	181
4.217Use origin context predicates for selecting component ports . . . . .	182
4.218Select all unmapped signal instances . . . . .	185
4.219Select all unmapped rx or transformed signal instances . . . . .	185
4.220Select signal instances using an advanced filter . . . . .	186
4.221Select all unmapped delegation port communication elements . . . . .	189
4.222Select all communication elements with their leafs . . . . .	189
4.223Select communication elements using an advanced filter . . . . .	190
4.224Example of the IComponentType model abstraction . . . . .	190
4.225Select component type by name . . . . .	192
4.226Select not instantiated component types . . . . .	192
4.227Select events example . . . . .	195
4.228Select executable entities example . . . . .	198
4.229Select PortInterface by name and type . . . . .	200
4.230Select PortInterfaces by component ports . . . . .	200
4.231Select ends of incomplete connections . . . . .	203
4.232Select the ends of incomplete connections for a specific flat extract component port	203
4.233Tries to auto-map all ports . . . . .	204
4.234Tries to auto-map all unconnected component ports . . . . .	204
4.235Tries to auto-map all unconnected sender/receiver and client/server ports . . . . .	205
4.236Tries to auto-map port determined by advanced filter . . . . .	205
4.237Tries to auto map all unconnected ports to the ports of one component prototype	206
4.238Tries to auto-map all unconnected ports and evaluate matches . . . . .	207
4.239Another example for using evaluate matches . . . . .	208
4.240Auto-map a component port and realize 1:n connection by using evaluate matches	209
4.241Create mapping between two ports which names do not match. . . . .	210
4.242Use the origin context for the port name matching . . . . .	211
4.243Example how to create a simple assembly connection . . . . .	212
4.244Example how to create a simple delegation connection . . . . .	213

4.245 Create connector with port interface mapping . . . . .	213
4.246 Connect ports using simple API . . . . .	214
4.247 Remove Connectors between Component Ports . . . . .	215
4.248 Remove Connectors between Component Ports Filtering Targets . . . . .	216
4.249 Terminate port using the component port selection API . . . . .	218
4.250 Remove port terminator using the component port selection API . . . . .	218
4.251 Create a port terminator using the simple API . . . . .	219
4.252 Remove a port terminator using the simple API . . . . .	219
4.253 Terminate port using the communication element selection API . . . . .	220
4.254 Remove port terminator using the communication element selection API . . . . .	220
4.255 Auto data map all unmapped signal instances . . . . .	221
4.256 Auto data map all unmapped signal instances to unmapped communication elements and evaluate . . . . .	222
4.257 Auto data map all signal instances and do not expand nested array elements . . . . .	223
4.258 Auto data map all signal instances and expand specific nested array element . . . . .	224
4.259 Evaluate matched communication elements using compatibility . . . . .	225
4.260 Decide which mappings should be created by compatibility . . . . .	226
4.261 Auto data map all unmapped sender/receiver delegation port communication elements . . . . .	227
4.262 Auto data map all unmapped communication elements to unmapped rx signal instances and evaluate . . . . .	228
4.263 Autodatamap and do not expand nested array elements . . . . .	229
4.264 Autodatamap and do expand a specific nested array element . . . . .	230
4.265 Evaluate matched signal instances using compatibility . . . . .	231
4.266 Decide which mappings should be created by compatibility . . . . .	232
4.267 Create sender receiver to signal mapping . . . . .	237
4.268 Create data mapping for delegation port . . . . .	238
4.269 Create client server to signal mapping . . . . .	238
4.270 Map record to signal group . . . . .	239
4.271 Map array to signal group . . . . .	240
4.272 Map complex data element to system signal group and let the auto-mapper complete the mapping if possible . . . . .	241
4.273 Map communication elements to system signals using simple API Part1 . . . . .	242
4.274 Map communication elements to system signals using simple API Part2 . . . . .	243
4.275 Remove Data Mapping of Communication Element . . . . .	245
4.276 Remove Data Mapping of Communication Element Considering Signals . . . . .	246
4.277 Remove Data Mapping of Signal . . . . .	247
4.278 Remove Data Mapping of Signals Considering Communication Elements . . . . .	248
4.279 Create component prototypes for not instantiated types . . . . .	249
4.280 Specify name of created component . . . . .	249
4.281 Create more than 1 component prototype . . . . .	250
4.282 Create delegation port simple . . . . .	251
4.283 Create delegation port customized . . . . .	252
4.284 Create delegation port based on existing component port . . . . .	253
4.285 Create a delegation port using the port interface of an origin context port . . . . .	254
4.286 Create a delegation port using the origin context port to specify name and direction . . . . .	255
4.287 Perform task mapping example . . . . .	257
4.288 Advanced Filter for Events . . . . .	258
4.289 Do not combine runnable and bsw module entity via symbol . . . . .	259
4.290 Map all events of a runnable together . . . . .	260
4.291 Order the task mappings by defining successor relationships . . . . .	261

4.292Order groups by using successor calls . . . . .	262
4.293Insert new task mappings always below existing - Part 1 . . . . .	263
4.294Insert new task mappings always below existing - Part 2 . . . . .	264
4.295Define successors and sort elements for better overview . . . . .	266
4.296Simple example of ordering the task mappings . . . . .	267
4.297Manually order the task mappings . . . . .	268
4.298Order task mappings on OsTask . . . . .	269
4.299Filter task mappings . . . . .	272
4.300Use execution order constraints for the task mapping . . . . .	273
4.301Check which events are currently mapped to OsTask . . . . .	275
4.302Use the queue option example . . . . .	276
4.303Set the activation offset using the event selection . . . . .	277
4.304Set the activation offset using the executable entity selection . . . . .	278
4.305Set the activation offset while mapping to a task . . . . .	279
4.306Switch between MDF and model abstraction example . . . . .	280
4.307Delete connectors to delegation ports . . . . .	282
4.308Delete data mappings of sender receiver delegation ports - Part 1 . . . . .	283
4.309Delete data mappings of sender receiver delegation ports - Part 2 . . . . .	284
4.310Create variant data mappings . . . . .	285
4.311Create delegation ports for selected origin ports and connect them . . . . .	288
4.312Accessing IUnresolvedReferenceApi as a property . . . . .	291
4.313Accessing IUnresolvedReferenceApi in a scope-like way . . . . .	291
4.314Accessing IEcucUnresolvedReferenceApi as a property. . . . .	291
4.315Accessing IEcucUnresolvedReferenceApi in a scope-like way. . . . .	292
4.316Get a filtered set of all unresolved references at the ECUC configuration. . . . .	292
4.317Get a filtered set of all unresolved references at the ECUC configuration with a specific pattern at the owner path. . . . .	292
4.318Get a filtered set of the changeable unresolved references at the ECUC configuration. . . . .	292
4.319Set changeable unresolved references. . . . .	293
4.320Replace 'MICROSAR' with 'OTHER' in all changeable unresolved references. . . . .	293
4.321Replace the prefix '/MICROSAR' with '/OTHER' in all changeable unresolved references. . . . .	294
4.322Create Custom Report . . . . .	294
4.323Accessing the model export persistency API . . . . .	296
4.324Export the ActiveEcuc to a file . . . . .	296
4.325Export the ActiveEcuc into a folder . . . . .	296
4.326Export a PostBuild project into files per predefined variant . . . . .	297
4.327Export a PreBuild project into files per predefined variant . . . . .	297
4.328Exports a module configuration . . . . .	298
4.329Export the project with an exporter into a folder . . . . .	298
4.330Export the project with an exporter and checks . . . . .	299
4.331Export an AUTOSAR package into a file . . . . .	299
4.332Export an AUTOSAR package into a folder . . . . .	299
4.333Exports two elements and all references elements . . . . .	300
4.334Use exporter arguments like in the commandline . . . . .	300
4.335Accessing the model import persistency API . . . . .	300
4.336Accessing the import module configuration persistency API . . . . .	301
4.337Specify the module configuration import mode and filter . . . . .	302
4.338Accessing the automatic merge API . . . . .	303
4.339Importing a platform function definition . . . . .	305

4.340Filter for a platform function . . . . .	306
4.341Integrating a platform function . . . . .	306
4.342Evaluating the automatic merge result (cancel if something is left) . . . . .	307
4.343Evaluating the automatic merge result (abort at a certain identifier) . . . . .	308
4.344Java code usage of the IScriptFactory to contribute script tasks . . . . .	312
4.345Accessing WorkflowAPI in Java code . . . . .	313
4.346Java Closure creation sample . . . . .	313
4.347Additional JUnit4 dependency in Gradle . . . . .	314
4.348Run all JUnit tests from one class . . . . .	314
4.349Run all JUnit tests using a Suite . . . . .	315
4.350Run unit test with the Spock framework . . . . .	315
4.351Additional Spock dependency in Gradle . . . . .	316
4.352Add a UnitTest task with name MyUnitTest . . . . .	316
4.353The projectConfig.gradle file content for unit tests . . . . .	316
4.354Usage of the ITransactionUndoAllRule in a JUnit test . . . . .	316
4.355Usage of the IProjectLoadRule in a JUnit test . . . . .	317
4.356DaVinci Configurator build Gradle DSL API - generatorTests . . . . .	319
4.357DaVinci Configurator build Gradle DSL API - generatorTests - genDevKit . . . . .	319
4.358Spock test using a GeneratorPackage . . . . .	320
4.359JUnit test using a GeneratorPackage . . . . .	320
4.360Spock test case which generates an OutputFile and verifies the content . . . . .	320
4.361JUnit test case which generates an OutputFile and verifies the content . . . . .	321
4.362Spock test case which verifies a ValidationResult . . . . .	321
4.363JUnit test case which verifies a ValidationResult . . . . .	322
5.1 Check object visibility . . . . .	329
5.2 Get all available variants . . . . .	329
5.3 Get all available variants or if project does not contain variants the invariant view	329
5.4 Execute code with variant visibility . . . . .	329
5.5 Get all variants, a specific object is visible in . . . . .	331
5.6 Retrieving an InvariantValues model view . . . . .	333
5.7 Retrieving an InvariantEcucDefView model view . . . . .	334
5.8 Execute code with variant specific changes . . . . .	336
5.9 Sample code to access element in an Untyped model with DefRefs . . . . .	340
5.10 Resolves a Reference target of an Reference Parameter . . . . .	340
5.11 The value of a GIPparameter . . . . .	340
5.12 Java: Execute code with creation IMModelView of BswmdModel object . . . . .	343
5.13 Java: Execute code with creation IMModelView of BswmdModel object via runnable	344
5.14 Java: Execute code with creation IMModelView of BswmdModel object . . . . .	344
5.15 Additional write API methods for EcucGeneral . . . . .	346
5.16 EcucCoreDefinition as GICList<EcucCoreDefinition> . . . . .	347
5.17 Deleting model objects . . . . .	347
5.18 Duplication of containers . . . . .	348
5.19 Set parameter values with the BswmdModel Write API . . . . .	348
5.20 Set reference targets with the BswmdModel Write API . . . . .	348
5.21 Settings.xml sample for DerivativeMapping . . . . .	349
5.22 DefRef isDefinitionOf methods . . . . .	351
5.23 Creation of DefRef with wildcard from EDefRefWildcard . . . . .	353
5.24 Getting CeState objects using the BSWMD model . . . . .	353
5.25 Integer parameter definition access examples . . . . .	355
5.26 Integer parameter configuration access examples . . . . .	360

7.1	The automationClasses list in projectConfig.gradle . . . . .	377
7.2	The dvCfgInstallation in projectConfig.gradle . . . . .	377
7.3	The dvCfgInstallation with an System env in projectConfig.gradle . . . . .	377
7.4	@CompileStatic with Automation API . . . . .	379
7.5	@TypeChecked with Automation API . . . . .	380
7.6	DaVinci Configurator build Gradle DSL API . . . . .	380
7.7	DaVinci Configurator build Gradle DSL API - useBswmdModel . . . . .	380
7.8	DaVinci Configurator build Gradle DSL API - useJarSignerDaemon . . . . .	381
7.9	DaVinci Configurator build Gradle DSL API - setJarSignerWaitTimeoutMin . . . . .	381
7.10	DaVinci Configurator build Gradle DSL API - includeDependenciesIntoJar . . . . .	381
8.1	Additional Groovy XML dependency . . . . .	385
8.2	Additional Groovy XML dependency . . . . .	394

## **Todo list**