SCADE® Python API Guide

SCADE 2024 Products

CONTACTS

Legal Contact

Ansys France
15 place Georges Pompidou
78180 Montigny-le-Bretonneux FRANCE
Phone: +33 1 30 60 15 00
Fax: +33 1 30 64 19 42

Technical Support

Ansys France
Parc Avenue, 9 rue Michel Labrousse
31100 Toulouse FRANCE
Phone: +33 5 34 60 90 50
Fax: +33 5 34 60 90 41

Submit questions to SCADE Products Technical Support at scade-support@ansys.com.

Contact one of our Sales representatives at scade-sales@ansys.com.

Direct general questions about SCADE products to scade-info@ansys.com.

Discover latest news on our products at www.ansys.com/products/embedded-software.

LEGAL INFORMATION

TERMS OF USE

DOCUMENTATION DISCLAIMER

THIRD-PARTY LEGAL INFO

The Legal Notice (PDF) about third-party software can be found under the help folder of the SCADE installation.

Shipping date: 29/04/24 **Revision**: SCF-APG-24 - DOC/rev/80409-04

SCADE Python API Guide Overview

This manual provides advanced users with technical background focusing on the Python API and scripting capabilities for SCADE products. It is divided into parts containing the following chapters:

RELATED DOCUMENTS

- *SCADE Architect User Manual*
- *SCADE Suite User Manual*
- *SCADE Display User Manual*
- *SCADE UA Page Creator User Manual*
- *SCADE Test User Manual*
- *SCADE Metamodels Technical Reference Card*

TYPOGRAPHICAL CONVENTIONS

Bold	GUI buttons, options, tabs, radio buttons, drop-down lists, window titles, panels.
Italics	Path for menu selection from main menu bar or in contextual menus, new technical concepts or words, references to product manuals.
`Courier New`	Code extracts, constant or variable names, file names, directory path, or typed text.

Table of Contents

Table of Contents

Table of Contents

Table of Contents

1 / Introduction

This manual provides you with technical information about SCADE Python API[1] available with SCADE products. It is intended for advance usage with SCADE projects and SCADE Model-Based Design Environments.

- "SCADE Python API Basics"
- "Python API Environment"
- "API Capabilities by SCADE Products"

SCADE Python API Basics

- "Access to SCADE Models Data"
- "Access to Generated Data"

Access to SCADE Models Data

The Python API provides various access methods to the following SCADE data:

- Content of SCADE projects (files, configurations, ...)
- Design element descriptions in SCADE models

---

1. For details about SCADE Python APIs, you may also read this article.

PROJECT CONTENT DATA

All project data can be accessed by Python API: the list of files, the configuration, and their properties.

Note

For details about accessible data, refer to "SCADE Projects Metamodel" in *SCADE Metamodels Technical Reference Card*.

SCADE DESIGN DATA

All data of a model can be accessed by Python API: the design elements, their attributes and relations, the graphical information, specific tools, as well as user information.

Access to Generated Data

The API provides various access methods to the following data generated by SCADE tools:

- Mapping files generated by SCADE Display KCG and SCADE Suite KCG
- Test results and model coverage results

MAPPING FILES

The Python API provides access to the generated C and Ada code and the relationships with the SCADE Suite or SCADE Display models, allowing for the generation of some integration code for instance.

TEST RESULTS AND MODEL COVERAGE RESULTS

The Python API provides access to test model, test execution results and coverage results for example to generate specific reports, metrics or checks.

Python API Environment

This section explains how to setup the environment for the Python API modules and introduces the commands and tools available for executing scripts in Python or SCADE environments.

- "Python Path and General Setup"
- "Executing Scripts in Python Environment"
- "Executing Scripts in SCADE Environment"
- "Script Wizard in SCADE Environments"

Python Path and General Setup

SCADE integrates Python APIs as a set of Python files located under `\SCADE\APIs\Python\lib` in SCADE installation directory.

Some SCADE APIs have no Python file but are loaded dynamically as for SCADE Architect, FACE, AADL.

One can write scripts in Python by referencing and loading the appropriate SCADE modules as illustrated below:

```python
import scade
import scade.model.project
import scade.model.suite
import scade.model.architect
```

You can either use directly the usual Python environment installed in the product, or use SCADE commands to execute a script in SCADE environment and get access to more tools and services. Notice that some of SCADE APIs are C-wrapped APIs and can only be used in Python environment under specific conditions.

Executing Scripts in Python Environment

The usual tools for editing or debugging in Python are available. The Python script can invoke several APIs importing the needed modules and calling explicitly the dedicated load function to get information from a model, code, or result.

To execute a script in Python environment

1 Use the general command: `python.exe <script.py>`:

- Set `\SCADE\APIs\Python\lib` path from SCADE installation directory in the `PYTHONPATH` environment variable.
- You can use the Python installation provided under `\contrib\Python310` and add this repository to your `PATH` environment variable (if needed)

In case of C-Wrapped API, it is necessary to fulfill the following conditions:

- Use only Python 3.10 as the one distributed in `\contrib`.
- Set both `\SCADE\APIs\Python\lib` and `\SCADE\bin` paths from SCADE installation directory in the `PYTHONPATH` environment variable.
- Import the `scade_env` module at the beginning of the script.
- Load models using the command: `scade_env.load_project(<projet_path>)`

Executing Scripts in SCADE Environment

It is possible to execute Python scripts in batch or in SCADE environment to benefit from services in the user interface (UI).

To execute a script in batch

1 Use the general command:

```
scade.exe -script [-project] <project>+ [-script] <script> [[-command] <command>]
```

- `<project>`+ is the path to one or more SCADE projects (.etp file) to be loaded before executing the script(s)
- `<script>` is the path to the script file to execute
- `<command>` is an optional command to be executed in the context of the script

All models are automatically loaded and one can access to the models using different SCADE APIs. Users have access to UI-specific commands (for SCADE Suite, SCADE Test, and SCADE Architect). For details, see "GUI Display-Related Python Commands" on page 97. If one wants to use Python modules that are not available by default, it is possible to enrich the PYTHONPATH variable by adding the content of the SCADEPYTHONPATH environment variable if it exists. The SCADEPYTHONPATH must contain only additional modules paths.

To execute a script from UI

1 Load the script in SCADE Suite, SCADE Test, or SCADE Architect.

2 Click **Execute Script** [!] (Ctrl+F5) in **Script** toolbar.

It is also possible to execute scripts for code generation using GUI mode or code generation services. One can execute code generation scripts using `scade -code` provided services are defined as detailed in "Generating Integration Code from SCADE Suite" on page 42.

Script Wizard in SCADE Environments

With SCADE Script Wizard, it is also possible to generate and execute Python scripts directly in SCADE environments. Script Wizard can extract data from SCADE Architect, SCADE Suite, or SCADE Test projects. It facilitates the creation of scripts by browsing and navigating in project data structure from the following entry points:

project: list of loaded project (ETP project files)

errors: information and errors produced by last tool execution (ERR files)

display: mapping information about graphical displays inserted in loaded project

scade: data extracted from XSCADE files

annot: data extracted from ANN files

timingstack: data extracted from TSO/TSV analysis results

verifier: data extracted from I4 files (if Design Verifier enabled)

testenv: data extracted from test procedure contained by SCADE Test project

system: data extracted from SCADE Architect project (.uml, .di, .notation files)

Each entry point belongs to a specific metamodel possibly linked to other metamodels. For instance, the **scade** metamodel contains links to the **project** metamodel. Navigating in metamodel allows to check which classes or associations define the metamodel. For details about using Script wizard, see

API Capabilities by SCADE Products

Product	API Building Blocks	API Type	Access Mode
All SCADE Products	Project	Native (Full Python)	python.exe scade -script
SCADE Architect	Architect	Dynamic	
	Annotations		
SCADE Suite	Scade	C-Wrapped	
	Annotations		
	SCADE Display coupling		
	KCG mapping file	Native	
SCADE Display	Display	Native (Full Python)	
	ARINC		
	KCG mapping file		
SCADE Test	Test and results	C-Wrapped	
	Model coverage results		
SCADE LifeCycle ALM	ALM Gateway	Native (Full Python)	
SCADE Packages	Avionics	Dynamic	
	AADL		
	FACE		
	AUTOSAR		
	Asterios		

2 / Data Access

This chapter introduces the different APIs available for each SCADE product for accessing SCADE model data. It also explains how to use Script Wizard to extract model data and execute Python scripts interactively from SCADE environments.

- "Accessing Project Content (ETP)"
- "Accessing SCADE Architect Models"
- "Accessing SCADE Suite Models"
- "Accessing SCADE Display Models"
- "Accessing SCADE ARINC 661 Models"
- "Accessing SCADE Test Models"
- "Accessing Models from SCADE Packages"
- "Accessing Traceability Information"
- "Using Script Wizard for Python Scripting"

Accessing Project Content (ETP)

All SCADE projects can be accessed using the same API.

Project API Presentation

The Project API allows to get all project information, the list of files, as well as configurations and their properties. This is a read/write Python API. Details about accessible project data are available from the project metamodel.

Note

For using Python API to get information available on a project, refer to "SCADE Projects Metamodel" in *SCADE Metamodels Technical Reference Card* and to "API-Related Python Commands" on page 94.

The Project API sources are accessible in the **scade.model.project** Python package.

To access all API root elements, it is necessary to call **scade.model.project.stdproject.get_roots()**.

Example

The following is an example for getting all file paths of a project.

```python
import scade, scade.model.project
def outputln(text):
    scade.output(text + '\n')
def for_the_pathname(pathname):
    outputln(str(pathname))
def for_each_element(element):
    vpathname = element.pathname
    for_the_pathname(vpathname)
def for_each_root(root):
    for v in root.elements:
        if isinstance(v, scade.model.project.FileRef): for_each_element(v)
        if isinstance(v, scade.model.project.Folder): for_each_root(v)
for item in scade.model.project.stdproject.get_roots():
    for v in item.roots: for_each_root(v)
```

Accessing SCADE Architect Models

The data of SCADE Architect models can be accessed using the SCADE Architect Model API as detailed below.

Model API Presentation

The SCADE Architect Model API allows to load a model and retrieve all Architect design elements and their attributes, the graphical information, and annotations. This is a read/write dynamic Python API that cannot be used by an external user interface. Details about accessible model data are available from SCADE Architect metamodels.

Note

For accessing SCADE Architect model content using Python API, refer to "SCADE Architect Metamodels" in *SCADE Metamodels Technical Reference Card* and to "API-Related Python Commands" on page 94.

The SCADE Architect Model API sources are contained in the `scade.model.architect` Python package.

To access all API root elements, it is necessary to call `scade.model.architect.get_roots()`.

The following API functions and their parameters are available:

```
scade.architect.create_project(project_path[, configuration[, annotSchemaList]])
```

Creating new project

- `project_path` is the path of the project to create
- `configuration` is the name of the configuration to apply to the created project (optional)
- `annotSchemaList` is the list of ATY files to apply to the created project (optional)

```
scade.architect.begin_write_transaction(element)
```

Starting write transaction necessary to modify SCADE Architect models loaded in SCADE

- element is the element for which the write transaction is started (its containing model more precisely)

Warning: Starting a write transaction must always be closed by an end write transaction, otherwise SCADE Architect may not react properly. A good practice would be to write as follows:

```
try:
    scade.architect.begin_write_transaction
    ... <modification of element> ...
finally:
    scade.architect.end_write_transaction
```

```
scade.architect.end_write_transaction(element)
```

Ending write transaction necessary to modify SCADE Architect models loaded in SCADE

- element is the element for which the write transaction is ended (its containing model more precisely)
- end_write_transaction is equivalent to a commit of the transaction (since the start)

```
scade.architect.save
```

Saving the SCADE Architect project to which a specified object belongs

- Parameter: any SCADE Architect object

Example

The following is an example for obtaining the list of all blocks.

```
import scade, scade.model.architect
def outputln(text):
    scade.output(text + '\n')
def for_the_qualified_name(qualified_name):
    outputln(str(qualified_name))
def for_each_block(block):
    vqualified_name = block.qualified_name
    for_the_qualified_name(vqualified_name)
def for_each_package(package):
    for v in package.blocks:
        for_each_block(v)
    for v2 in package.packages:
        for_each_package(v2)
for item in scade.model.architect.get_roots():
    for v in item.packages:
        for_each_package(v)
```

Accessing SCADE Suite Models

The data of SCADE Suite models can be accessed using the SCADE Suite Model API as detailed below.

Model API Presentation

The SCADE Suite Model API allows to navigate in the SCADE Suite design model, accessing design elements, their attributes, the graphical information, as well as annotations and pragmas. This is a read/write Python API. Details about accessible model data are available from SCADE Suite metamodels.

Note

For accessing SCADE Suite model content using Python API, refer to "SCADE Suite Metamodels for Python API" in *SCADE Metamodels Technical Reference Card* and to "API-Related Python Commands" on page 94.

The SCADE Suite Model API sources are contained in the `scade.model.suite` Python package.

To access all API root elements, it is necessary to call `scade.model.suite.get_roots()`.

Example

The following is an example for obtaining the Scade full path of all operators.

```python
import scade, scade.model.suite
def outputln(text):
    scade.output(text + '\n')
def for_each_all_operator(all_operator):
    outputln(str(all_operator.get_full_path()))
def for_the_model(model):
    if model == None: return
    for v in model.all_operators:
        for_each_all_operator(v)
for item in scade.model.suite.get_roots():
    vmodel = item.model
    for_the_model(vmodel)
```

Accessing SCADE Display Models

The data of SCADE Display models can be accessed using the SCADE Display Model API as detailed below.

Model API Presentation

The SCADE Display Model API enables you to manipulate SCADE Display model files. This is a read/write Python API. This API uses the `pyparsing` package when expressions are used. If using the API in another python environment than the one installed with SCADE, one may need to install this package. Details about accessible model data are available from SCADE Display metamodels.

Note

For accessing SCADE Display model content using Python API, refer to "SCADE Display Metamodels" in *SCADE Metamodels Technical Reference Card* and to "API-Related Python Commands" on page 94.

The SCADE Display Model API sources are contained in the `scade.model.display` Python package. It provides access to read/write functions for handling the following formats:

Format	Functions
SGFX	`load_sgfx and save_sgfx`
OGFX	`load_ogfx and save_ogfx`
PGFX	`load_pgfx and save_pgfx`
DGFX	`load_dgfx and save_dgfx`
SCT	`load_sct and save_sct`
SWT	`load_swt and save_swt`
SST	`load_sst and save_sst`
STT	`load_stt and save_stt`
SGT	`load_sgt and save_sgt`

The Python package provides utilities for manipulating types and expressions:

Utilities	Functions

Decoding strings and building corresponding type or expression	`parse_type`
	`parse_global_type`
	`parse_expr`
	`parse_global_constant`

Computing string representation of type or expression	`print_type`
	`print_global_type`
	`print_expr`
	`print_global_constant`

The Python package provides class constructors (`init_methods`) for initializing class attributes with optional arguments. It is possible to specify values only for the arguments that are relevant to user needs. Default values are automatically generated for all other arguments that are not explicitly specified by users. For instance, to build a circle, one can write different variants as illustrated below.

`Circle()`	All parameters are initialized by default values
`Circle(center = (10.0, 20.0), radius = 100.0, outline_color = 41, fill_color = 39)`	Circle is positioned on specified coordinates with specified radius, outlining color, and filling color. All other parameters are given default values.
`Circle(name = 'my circle', oid = '123456789', center = (10.0, 20.0), radius = 100.0)`	Circle is positioned on specified coordinates with specified radius, name, and OID. All other parameters are given default values.

Example

The following is an example for parsing objects recursively in containers and displaying their name.

```python
import scade.model.display as SDY
def print_graphic_object(go, indent = ' '):
    # print the object's name
    print(indent, go.name)
    # iterate over the children of containers (recursively)
    if isinstance(go, SDY.AContainer):
        for child in go.children:
            print_graphic_object(child, indent + '  ')
if __name__ == '__main__':
    spec = SDY.load_sgfx('C:\\examples\\GlassCockpitPFD.sgfx')
    # iterate over the layers of the specification
    for layer in spec.layers:
        print(layer.name)
        # iterate over the graphic objects of the layer
        for go in layer.children:
            print_graphic_object(go)
        print()
```

Other API usage examples are in SCADE installation at
SCADE\APIs\Python\examples\display.

Accessing SCADE ARINC 661 Models

The data of SCADE ARINC 661 models can be accessed using the SCADE ARINC 661 Model API as detailed below.

Model API Presentation

The SCADE ARINC 661 Model API enables you to manipulate SCADE A661 model files. This is a read/write Python API. Details about accessible model data are available from SCADE ARINC 661 metamodels.

Note

For accessing SCADE ARINC 661 model content using Python API, refer to <u>"SCADE UA Page Creator Metamodels"</u> in *SCADE Metamodels Technical Reference Card* and to <u>"API-Related Python Commands"</u> on page 94.

The ARINC 661 Model API sources are contained in the `scade.model.a661` Python package. It provides access to read/write functions for handling the following formats:

Format	Functions
XML*	`load_standard and save_standard`
SGFX (DF)	`load_sgfx and save_sgfx`
OGFX (Widget Sets)	`load_ogfx and save_ogfx`
PGFX	`load_pgfx and save_pgfx`
DGFX	`load_dgfx and save_dgfx`
SDT	`load_sdt and save_sdt`
PDT	`load_pdt and save_pdt`

* For files compliant with ARINC 661 Standard description format

The Python package provides class constructors (`init_methods`) for initializing class attributes with optional arguments. It is possible to specify values only for the arguments that are relevant to user needs. Default values

are automatically generated for all other arguments that are not explicitly specified by users. For instance, to build a circle, one can write different variants as illustrated below.

```
WidgetInstance(name = 'My New Widget
Instance', type =
helpers.get_widget(a661_std.widgets,
'Label'))
```

A Label widget is instantiated with specified name and type. All other parameters are given default values.

Example

The following is an example for creating and inserting a widget in model tree.

```
import scade.model.a661 as A661
if __name__ == '__main__':
    a661_std = A661.load_standard('C:\\examples\\a661_description\\a661.xml')
    df = A661.load_sgfx('C:\\examples\\FMS_demo.sgfx', a661_std)
    # We add an A661 label under the FCT_INIT widget
    layer = df.layers[0]
    print('Open Layer', layer.name)
    # retrieve Pages TabbedPanel Group
    pages_widget = layer.children[0]
    assert isinstance(pages_widget, A661.WidgetInstance)
    print('Open widget', pages_widget.name)
    # get FCT_INIT tabbed panel
    fct_widget = pages_widget.children[0]
    print('Open widget', fct_widget.name)
    # We create a Label Widget Instance
    label_widget = A661.WidgetInstance()
    # Get the corresponding Widget standard description
    widget_desc = A661.helpers.get_widget(a661_std.widgets, 'Label')
    label_widget.type = widget_desc
    # set a widget Id not already defined in the layer
    label_widget.id = 230
    label_widget.name = 'My New Widget Instance'
    print('Definition of new widget:', label_widget.name)
    assert isinstance(fct_widget, A661.WidgetInstance)
    # then insert the widget at the first position under the FCT_INIT tabbed panel
    fct_widget.children.insert(0, label_widget)
    # Save the model at its new location
    A661.save_sgfx(df, 'C:\\examples\\FMS_demo2.sgfx')
```

Other API usage examples are in SCADE installation at SCADE\APIs\Python\examples\a661.

Accessing SCADE Test Models

The data of SCADE Test models can be accessed using the SCADE Test Model API as detailed below.

Model API Presentation

The SCADE Test Model API enables you to access all elements of a test project: procedures definitions and the results of execution. This is a read/write Python API. Details about accessible model data are available from SCADE Test metamodels.

Note

For accessing SCADE Test model content using Python API, refer to "SCADE Test Metamodels" in *SCADE Metamodels Technical Reference Card* and to "API-Related Python Commands" on page 94.

The SCADE Test Model API sources are contained in the `scade.model.testenv` Python packages.

To access all API root elements, it is necessary to call `scade.model.testenv.get_roots()`.

Example

The following is an example for listing the scenarios of a procedure.

```python
import scade, scade.model.testenv

def outputln(text):
    scade.output(text + '\n')

def display_scenario_path(scenario):
    pathname = scenario.pathname
    outputln(str(pathname))

for app in scade.model.testenv.get_roots():
    for procedure in app.procedures:
        vname = procedure.name
        outputln('procedure name: '+ vname)
        vdescription = procedure.description
        outputln('procedure description: '+ vdescription)
        operator = procedure.operator
        outputln('tested operator: '+ operator)
        for record in procedure.records:
            vname = record.name
            outputln('scenarios of record: '+ vname)
            for init in record.inits:
                display_scenario_path(init)
            for preamble in record.preambles:
                display_scenario_path(preamble)
            for scenario in record.scenarios:
                display_scenario_path(scenario)
```

Accessing Models from SCADE Packages

The Python APIs allow to access to Avionics, AADL, FACE, AUTOSAR, or ASTERIOS design elements. To access any of these design elements, use the functions described in "API-Related Python Commands" on page 94.

- For details about SCADE Avionics design elements, refer to Appendix B about "Avionics Configurations" in *SCADE Avionics Package Guidelines*.

- For details about SCADE FACE design elements, refer to Appendix A about "AADL Configuration" in *SCADE Avionics Package Guidelines*.

- For details about SCADE AUTOSAR design elements, refer to Appendix A about "AUTOSAR Configuration and Timing Extension Concepts" in *SCADE Automotive Package Guidelines*.

- For details about Open System Description (OSD) design elements for ASTERIOS, refer to Appendix B about "OpenSystemDescription Configuration" in *SCADE Multi-Rate Application Guidelines*.

Accessing Traceability Information

The traceability data of SCADE projects can be accessed using the Traceability API as detailed below.

Traceability API Presentation

The Traceability API allows to access all traceability data of SCADE projects: requirement documents, document sections, requirements, status of requirements, traceable elements, and traceability links. This is a read/write Python API. Details about accessible traceability data are available from SCADE metamodels.

Note

For accessing traceability data using Python API, refer to "SCADE Traceability Metamodels for Python API" in *SCADE Metamodels Technical Reference Card*.

Traceability is made of information found in the ALMGR (requirements and exported traceability) and ALMGT (non-exported traceability) files.

The SCADE Traceability API sources are contained in the `scade.model.traceability` Python package.

To access to the root element of traceability information, it is necessary to load the traceability project: `scade.model.traceability.Project`.

The following API functions and their parameters are available:

```
scade.model.traceability.load(<etp_path>)
```

Loading traceability information from SCADE project and returning traceability project

- `etp_path` is the absolute path to the SCADE project (.etp)

For instance:

```
import scade.model.traceability as trace
project = trace.load(<etp_path>)
```

```
scade.model.traceability.get_non_exported_links(project)
```

Returning the list of links that are not exported in the traceability project

- `project` is the loaded traceability project

For example:

```
import scade.model.traceability as trace
links = trace.get_non_exported_links(project)
```

```
scade.model.traceability.get_pending_links(project)
```

Returning the list of pending links in the specified traceability project

- `project` is the loaded traceability project

For example:

```
import scade.model.traceability as trace
links = trace.get_pending_links(project)
```

```
scade.model.traceability.save(project, <etp_path>)
```

Saving traceability changes in the current traceability project

- `project` is the modified traceability project
- `etp_path` is the absolute path to the SCADE project (.etp) where traceability changes are saved

For example:

```
import scade.model.traceability as trace
trace.save(project, <etp_path>)
```

```
scade.model.traceability.Project.add_link (oid, req)
```

Adding a traceability link in traceability project once project is loaded

- `oid` is the ID of the SCADE element to trace
- `req` is the requirement or the ID of the requirement to trace

```
scade.model.traceability.Project.remove_link (oid, req)
```

Removing a traceability link in traceability project once project is loaded

- same as above for add_link

Example

The following is an example for loading a project, modifying traceability in project, and saving project with traceability changes.

```python
import sys
import scade_env
import scade.model.suite as suite
import scade.model.traceability as traceability

DEFAULT_IDENT = " " * 4

def print_element(elt: traceability.Element, ident: str = ""):
    elt_to_print = elt
    if isinstance(elt, traceability.RequirementRef):
        elt_to_print = elt.requirement
    print(f"{ident}Identifier: {elt_to_print.identifier}")
    print(f"{ident}Name: {elt_to_print.name}")
    print(f"{ident}Description: {elt_to_print.description}")

def print_subelements(element: traceability.HierarchicalElement, ident: str = ""):
    for sub_elt in element.owned_elements:
        print_element(sub_elt, ident)
        print_subelements(sub_elt, ident + DEFAULT_IDENT)

def print_document_tree(doc: traceability.ALMDocument):
    for node in doc.owned_elements:
        print_element(node)
        print_subelements(node, DEFAULT_IDENT)

def print_requirement_documents(trace_project: traceability.Project):
    if trace_project is None or len(trace_project.alm_documents) == 0:
        raise Exception("No requirement documents in the project.")
    print("#############################")
    print("Requirement documents")
    print("#############################")
    for doc in trace_project.alm_documents:
        print(f"Identifier: {doc.identifier}")
        print(f"Name: {doc.name}")
        print(f"Location: {doc.location}")
        print("#############################")
        print("List of requirements:")
        print("#############################")
        for req in doc.requirements:
            print_element(req)
        print("#############################")
        print("Document tree:")
        print("#############################")
        print_document_tree(doc)
```

```python
def print_traced_elements_from_active_scade_project(trace_project: traceability.Project):
    if trace_project.active_scade_document is None:
        print("No traced SCADE elements in the project.")
        return
    if len(trace_project.active_scade_document.scade_elements) == 0:
        print("No traced SCADE elements in the project.")
        return
    print("##############################")
    print("Traced elements in active SCADE project")
    print("##############################")
    for scade_elt in trace_project.active_scade_document.scade_elements:
        print_element(scade_elt)

def print_links(trace_project: traceability.Project):
    if len(trace_project.traceability_links) == 0:
        print("No links in the project.")
        return
    print("##############################")
    print("List of links")
    print("##############################")
    print("(OID, Requirement ID, Status)")
    for link in trace_project.traceability_links:
        print(f"({link.source.identifier},{link.target.identifier}, {link.status.name})")

if __name__ == '__main__':
    if len(sys.argv) != 2:
        sys.exit("Wrong arguments: 'exemple_traceability.py <etp_path>'")
    etp_path = sys.argv[1]
    scade_env.load_project(etp_path)
    model = suite.get_roots()[0].model
    trace_project = traceability.load(etp_path)
    print_requirement_documents(trace_project)
    print_traced_elements_from_active_scade_project(trace_project)
    print_links(trace_project)
    print("##############################")
    print("Add links")
    print("##############################")
    if len(model.all_operators) == 0:
        raise Exception("No operators to trace in project")
    oid1 = model.all_operators[1].get_oid()
    print(f"Get first operator from project: OID={oid1}")
    if len(trace_project.alm_documents[0].requirements) == 0:
        raise Exception("No requirements in the requirement document.")
    req1 = trace_project.alm_documents[0].requirements[0]
    print(f"Get first requirement from requirement document: {req1.identifier}")
    print(f"Add link: ({oid1}, {req1.identifier})")
    trace_project.add_link(oid1, req1)
```

```python
if len(trace_project.traceability_links) != 0:
    print("#############################")
    print("Remove links")
    print("#############################")
    link_to_remove = trace_project.traceability_links[0]
    source_link = link_to_remove.source
    target_link = link_to_remove.target
    print(f"Remove link: ({source_link.identifier}, {target_link.name})")
    trace_project.remove_link(source_link.identifier, target_link.identifier)
print_links(trace_project)
print("#############################")
print("Save traceability: Update ALMGT file if exists, otherwise generate the file")
print("#############################")
traceability.save(trace_project, etp_path)
print("#############################")
print("Remove Added/removed links")
print("#############################")
print(f"Remove link: ({oid1}, {req1.identifier})")
trace_project.remove_link(oid1, req1)
if len(trace_project.traceability_links) != 0:
    print(f"Add link: ({source_link.identifier}, {target_link.name})")
    trace_project.add_link(source_link.identifier, target_link.identifier)
traceability.save(trace_project, etp_path)
print("#############################")
print("Save traceability: No new links, ALMGT is removed if it did not exist before")
print("#############################")
```

Using Script Wizard for Python Scripting

You can use Script Wizard to navigate in project and model data and execute scripts able to extract data from SCADE Architect, SCADE Suite, or SCADE Test projects.

- "Launching Script Wizard"
- "Navigating in Project Data"
- "Executing Python Scripts"
- "Examples of Python Script Execution"

Launching Script Wizard

You launch the Script Wizard from SCADE Architect, SCADE Suite, or SCADE Test environments.

To launch Script Wizard

- Select *View > Script Wizard* or click **Script Wizard** in **Script** toolbar (available from *View > Toolbars > Script*) to open the **Script Wizard**.

Navigating in Project Data

You navigate in project data via links on the Script Wizard pages. Each page describes a class by: Name, Attributes, Associations, Operations, and Heirs (or derived classes). Tables present the class that defines each feature (*i.e.,* current class or one of its ancestors). Navigation keeps track of all followed links and allows to generate Python code that reproduces the different navigation steps through the metamodel.

To navigate in project data

1 Select the entry point to browse through on the Script Wizard start page. By default Script Wizard starts in Python scripting mode.

2 Continue navigation clicking on the following hyperlinks:

- In **Attributes** tables: to conclude navigation on a selected attribute.
- In **Associations** tables: to navigate to an associated class.
- In **Heirs** tables: to navigate to the derived class.

3 Use ⬅ or ➡ in **Navigate** toolbar to move back and forth between pages.

Executing Python Scripts

After navigating and selecting a specific attribute or association in the metamodels, you can evaluate and create Python scripts. as explained hereafter.

Warning

Script execution instantiates items only for the duration of a working session.

The results of script execution (if displayable) are displayed in the **Browse** output tab or in the **Script** output tab for textual information. If the script is syntactically incorrect, an error message is displayed in the **Script** tab. The result of a script executed on **Attributes** is always displayed in the **Script** tab. For adding new script files in the project, see "Creating New Python Files" on page 98 in *SCADE Suite User Manual*.

- "Evaluating Scripts from Project Data"
- "Creating Scripts from Project Data"

Evaluating Scripts from Project Data

You can execute directly the underlying script whenever you stop navigating in the project data of the models.

To evaluate scripts from Script Wizard

1 Navigate in the metamodel and select an attribute or an association.

2 Click **Eval script code** on top of page.

3 Examine the result of script execution in the Output Docking Window tabs.

Creating Scripts from Project Data

You can visualize, edit, execute and/or save the output script resulting from navigation.

To create scripts with Script Wizard

1 Navigate in the metamodel and select an attribute or an association.

2 Click **Create Python script** on top of page to open an editing view of the script.

3 Click **Execute Script** in **Script** toolbar to execute the script immediately.

4 Select *File > Save* or *File > Save As* to save the script. Created scripts are not evaluated by Script Wizard and may block the application if they contain loops.

Examples of Python Script Execution

- <u>"Fetching List of Constants"</u>
- <u>"Fetching Text"</u>
- <u>"Displaying Enumerations"</u>

FETCHING LIST OF CONSTANTS

Run the following example to see how the Script Wizard can be used to fetch the list of constants from the project data of a model and create the corresponding script.

To run a script fetching the list of constants

1   Follow the *scade > model (in Associations table) > allConstant (in Associations table)* links starting from the first page of the **Script Wizard**.

2   Click **Eval script code** to see the result of the script execution.
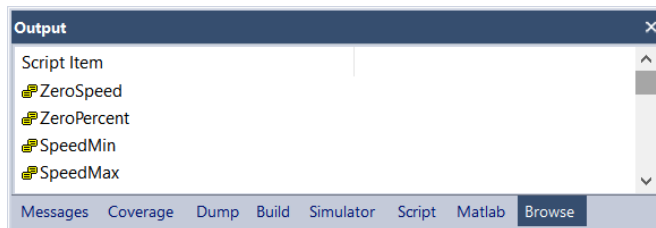
Figure 2.1: Result of script evaluation displaying list of model constants

3   Click **Create Python script** to see the content of the script.

```
import scade, scade.model.suite

def outputln(text):
    scade.output(text + '\n')

def for_each_all_constant(all_constant):
    scade.report(all_constant)

def for_the_model(model):
    if model == None: return
    for v in model.all_constants:
        for_each_all_constant(v)

# Initialization of the columns of the report
scade.create_report('Browse', ('Script Item', 300, 0))

for item in scade.model.suite.get_roots():
    vmodel = item.model
    for_the_model(vmodel)
```

Figure 2.2: Python script corresponding to evaluation fetching list of constants

FETCHING TEXT

Run the following example to see how to use Script Wizard to fetch textual information about constants in the project data of a model and create the corresponding script.

To run a script fetching text

1   Follow the *scade- model - allConstant - name* links starting from the first page of the **Script Wizard**.

2   Click **Eval script code** to see the result of the script execution.
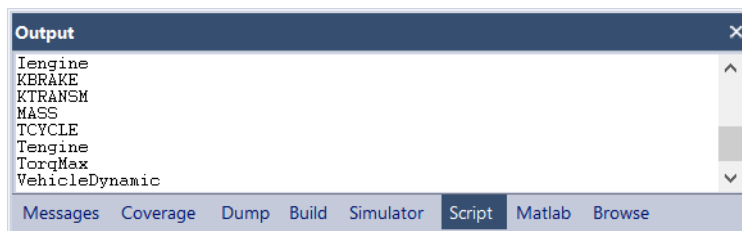
Figure 2.3: Result of evaluation displaying textual data about model constants

3   Click **Create Python script** to see the content of the script.

```python
import scade, scade.model.suite

def outputln(text):
    scade.output(text + '\n')

def for_the_name(name):
    outputln(str(name))

def for_each_all_constant(all_constant):
    vname = all_constant.name
    for_the_name(vname)

def for_the_model(model):
    if model == None: return
    for v in model.all_constants:
        for_each_all_constant(v)

for item in scade.model.suite.get_roots():
    vmodel = item.model
    for_the_model(vmodel)
```

Figure 2.4: Python script corresponding to evaluation fetching text

DISPLAYING ENUMERATIONS

Run the following example to see how to use Script Wizard to fetch the description of enumerations in the project data of a model and create the corresponding script.

To run a script displaying all enumerations in a model

1 Follow the *scade- model - allNamedType - definition - Enumeration* links starting from the first page of the **Script Wizard**.

2 Click **Eval script code** to see the result of the script execution.
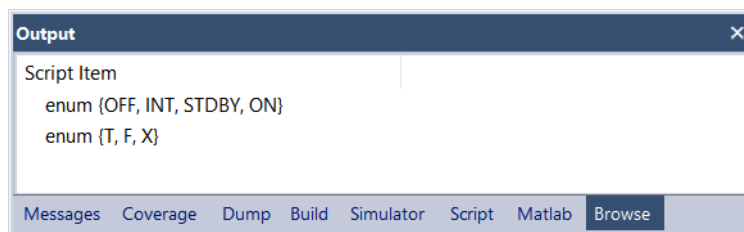
Figure 2.5: Result of script evaluation displaying model enumerations

3 Click **Create Python script** to see the content of the script.

```python
import scade, scade.model.suite

def outputln(text):
    scade.output(text + '\n')

def for_the_definition(definition):
    if definition == None: return
    scade.report(definition)

def for_each_all_named_type(all_named_type):
    vdefinition = all_named_type.definition
    if isinstance(vdefinition, Enumeration):
        for_the_definition(vdefinition)


def for_the_model(model):
    if model == None: return
    for v in model.all_named_types:
        for_each_all_named_type(v)

# Initialization of the columns of the report
scade.create_report('Browse', ('Script Item', 300, 0))

for item in scade.model.suite.get_roots():
    vmodel = item.model
    for_the_model(vmodel)
```

Figure 2.6: Python script corresponding to evaluation fetching model enumerations

3 / Code Integration Toolbox

This chapter provides some general information to get started with the Mapping File APIs for SCADE Suite KCG and SCADE Display KCG. It also explains how one can customize extensions able to interact with SCADE Suite Code Generators.

- "Mapping File Access for SCADE Display KCG 6.7"
- "Mapping File Access for SCADE Suite KCG/MCG/ACG"
- "Generating Integration Code from SCADE Suite"
- "Multicore Code Integration API"

Mapping File Access for SCADE Display KCG 6.7

This section introduces what is the Mapping File Python API for SCADE Display KCG 6.7, gives guidelines to get started, and illustrates the API with metamodel diagrams.

- [“Fundamentals about Mapping File API”](#)
- [“Getting Started with Mapping File API”](#)
- [“Mapping File Metamodels”](#)

Fundamentals about Mapping File API

The mapping file generated by SCADE Display KCG provides traceability information between the SCADE Display model and the generated code (C). Starting from SCADE Display KCG 6.7, this mapping file has a Python API allowing to access information.

The structure of the mapping file sections is the following:

- A description of the configuration used to generate code: the options with attributes and values for all command-line options of the SCADE Display KCG execution.
- A description of the structure of the model: the content of the layer or container of the model used to generate the code.
- A description of the content of the generated interface C files.
- A description of the mapping between model elements and their counterparts in the generated code.

Getting Started with Mapping File API

The sources of the Mapping File Python API for SCADE Display KCG are contained in the `scade.code.display.mapping` Python package. It provides a `load_xmap` function to read files in XMAP format.

Mapping File Metamodels

All class diagrams are available in ["SCADE Display Mapping Files Metamodels"](#) of *SCADE Metamodels Technical Reference Card*.

Mapping File Access for SCADE Suite KCG/ MCG/ACG

This section introduces what is the Mapping File Python API for SCADE Suite KCG Multicore Code Generator (MCG), or AUTOSAR Code Generator (ACG) gives direct access to its dedicated documentation, and illustrates the API with an example of API usage.

- <u>"Fundamentals about Mapping File API"</u>
- <u>"Getting Started with Mapping File API"</u>
- <u>"Examples"</u>

Fundamentals about Mapping File API

The mapping file generated by SCADE Suite KCG, Multicore Code Generator, or AUTOSAR Code Generator provides traceability information between the SCADE Suite model and the generated C or Ada code.

The structure of the mapping file is the following:

- A section listing the code generation options used to produce the mapping file.
- A section describing the structure of the model that was compiled. This section corresponds to a compiled version of the model interface, taking into account expansion and resolution of generic operator calls.
- A section describing the typed interface of the generated code. This section is specific to each back-end. If the mapping file corresponds to generated C code, the section contains the `.h` files and all declarations in them. If the mapping file corresponds to generated Ada code, the section contains the `.ads` files and all declarations in them.

- A section giving the mapping between model elements and their counterparts in the generated code. Each mapping can be annotated with a role to disambiguate mapping when necessary.

- A MCG-specific section describing the architecture of the generated Kahn Process Network (KPN). This section represents the generated tasks and channels as well as their mappings with the input model and the generated code.

- A potentially empty section giving the mapping between generated elements. Each mapping can be annotated with a role to disambiguate mapping when necessary.

The entry point depends on the target language specified when generating code:

- *C back-end* from `scade.code.suite.mapping.c` module (KCG, MCG, and ACG)

- *Ada back-end* from `scade.code.suite.mapping.ada` module (only KCG)

The API for the model part is accessed through the back-end specific API only, responsible for loading and interpreting the mapping file. The API provides the necessary functions to walk-through the generated code or the corresponding model. Check the API documentation for more details.

Getting Started with Mapping File API

This section describes how and where to find the API detailed information. A small example can be used as a starting point.

ACCESSING API MODULES

Start a mapping file script in Python by referencing the `scade.code.suite.mapping` module and loading the appropriate modules as in:

```
from scade.code.suite.mapping import c
```

or

```
from scade.code.suite.mapping import ada
```

To access all API root elements, it is necessary to call **`scade.code.suite.mapping.open_mapping(file_path)`**.

ACCESSING API DOCUMENTATION

All API documentation is available in HTML format from the default browser. Specific documentation about tasks, channels, and methods for MCG code integration activities is available from the **Architecture Module** section.

Launch API documentation

Examples

EXAMPLE WITH SCADE SUITE KCG

The following code snippet loads a KCG-generated mapping file and lists all root nodes for generated C code:

```python
from scade.code.suite.mapping import c
mapping = c.open_mapping('/path/to/mapping.xml')
roots = [ op for op in mapping.get_all_operators() if op.is_root() ]
```

The following snippet assumes that the `root` variable contains a root operator, and gets the name of the associated context memory and the fields of the context. See the API documentation for more usage details.

```python
ctx = root.get_generated().get_context()
ctx_name = ctx.get_name()
ctx_fields = ctx.get_fields()
```

EXAMPLE WITH MULTICORE CODE GENERATOR

The following code snippet loads a MCG-generated mapping file and lists all generated tasks with their methods.

```python
from scade.code.suite.mapping import c
mapping = c.open_mapping('KCG/mapping.xml')
for t in mapping.get_all_tasks():
    methods = [m.get_name() for m in t.get_methods()]
    print("task:{}, methods:{}".format(t.get_name(), ', '.join(methods)))
```

Generating Integration Code from SCADE Suite

SCADE Suite allows to define a Code Generator extension using information from SCADE Suite, SCADE Display, and SCADE Architect models.

- "Declaring Code Generator Extension"
- "Description of Code Generator API"
- "Description of WrapGen"
- "Predefined Generation Services"

Declaring Code Generator Extension

Users can generate their own code in the same way as SCADE code generation (using **Generate** command in GUI or `scade -code` in batch mode). With the help of Python APIs, code generation can take as input a set of information from SCADE Suite project, like linked models (Display or Architect), files, settings, and the mapping files.

Defining such extensions consists in:

- Registering code extension for displaying the extension name in **Code Integration** tab
- Providing a custom settings page for the extension (optional)
- Implementing Generate and Build functions called by the **Generate**, **Build**, and **Rebuild All** commands

The interface of a custom extension is described below.

- "Code Generation Extension Registration"
- "Custom Settings Page"
- "Generation Module"
- "Generation Service"

Code Generation Extension Registration

A code generation extension must be registered SCADE Suite user interface through a specific mechanism explained below. This is necessary to display any extension in the **Code Integration** tab of the Code Generator Settings and be able to select such extension for code generation using a given configuration.

A Code Generator extension has the following properties:

`Identifier`		Any valid identifier like MyExtension.
`DisplayName`	optional	A string displayed in user interface to identify the extension as "My Extension".
`Kind`		Defines the extension kind as: Target, Adaptor, or Other. The last value is used for "Other extensions".
`Server<n>`		A value to name a settings page where `n` goes from 1 to `n` for each settings page (if any). Use `<no_settings>` as value if the extension has no settings page.
`AllowRun, true|false`	optional default=false	Specifies whether the result of a **Run** command can be executed or not (in case Kind is Target only).
`CPU`	optional default= user-selected one	List of values separated by semi-colon (`win32; win64`) for defining the kind of CPU to be targeted by the compiler
`Languages`	optional default=c	List of values separated by semi-colon (`c; ada`) for defining the language of generated code supported by the extension
`CodeGenerators`	optional default=KCG66	List of values separated by semi-colon (`KCG66; MCG; ACG`) for defining the code generator supported by the extension

InternalIdent	optional	The generation module `ident` to invoke. By default, the `internalIdent` is the `Identifier`. `InternalIdent` is also the name of the project property saved when the extension is selected (see "Generation Module" on page 48).

All the above properties must be defined and provided with the extension using an `.srg` file whose content is illustrated in the example below.

- To register a Code Generator extension for a specific SCADE version, save the file at this system file location: `%APPDATA%/Scade/Customize/<Product Version Number>/`.
- To register a Code Generator extension for any SCADE version, save the file in this system file location: `%APPDATA%/Scade/Customize/`.

Example: MyExtension.srg

```
[Scade/Simulator/Extensions/MyExtension]
"Server1"="MyExtensionSettings"
"Kind"="Target"
"Languages"="c"
"AllowRun"="false"
"DisplayName"="My Extension"
"CPU"="win64"
"InternalIdent"="MyExtension"
```

Custom Settings Page

In case the Code Generator extension requires specific settings, it is possible to implement a dedicated settings page in Tcl or Python. For details about customizing settings pages, see"Adding New Settings Page" on page 156 in *SCADE Java/Tcl API Guide* or "Customization-Related Python Commands" on page 108 in *SCADE Python API Guide*.

Example: MyExtensionSettings.tcl

```tcl
#' Helpers ################################################################
source "ToolBox.tcl"
#' Packages ###############################################################
package require reporter
package require dialogs
#' page MyExtensionSettings ##############################################
declare ReporterPage -variable "MyExtensionSettings" -name "My Extension" -OnBuildPage
OnBuildPage -OnDisplayPage OnDisplayPage -OnValidatePage OnValidatePage -OnClosePage
OnClosePage
#' build functions
proc OnBuildPage { adrDialog } {
    global wndParam1
Label -variable "wndParam1Label" -parent "$adrDialog" -name "Param &1:" -x 7 -y 7 -w
110 -h 21
    Edit -variable "wndParam1" -parent "$adrDialog" -x 124 -y 4 -w 240 -h 21
    return 1
}
#' window functions
proc OnDisplayPage { project configuration } {
    global wndParam1

    set value [ ::Toolbox::GetScalarToolProp $project $configuration "MYEXTENSION"
"PARAM1" "" ]
    $wndParam1 -name $value
    return 1
}
proc OnValidatePage { project configuration } {
    global wndParam1

    set value [ $wndParam1 -name ]
    ::Toolbox::SetScalarToolProp $project $configuration "MYEXTENSION" "PARAM1" ""
    return 1
}
proc OnClosePage {} {
    return 1
}
#' main entry procedure ##################################################
proc Init {} {
    AddReporterPage "MyExtensionSettings"
}

#' start #################################################################
Init
```

If this script is copied in the `SCADE\scripts` directory of SCADE products installation it can be referenced relatively to this directory in the page registration otherwise it can be referenced to an absolute directory.

The settings page must be registered to SCADE Suite Code Generation user interface by adding entries to the `.srg` file as illustrated in the example below.

Example: MyExtension.srg (continued)

```
[Custom/Studio/Extensions/MyExtensionSettings]
"PathName"="MyExtensionSettings.tcl"
[Studio/Work Interfaces/MyExtensionSettings]
"PathName"="ETCUST.DLL"
```

USER INTERFACE RESULTING FROM EXTENSION DECLARATION

The Code Generator settings are enriched with the extension declared above. The different fields are filled or enabled/disabled depending on the different properties you defined.

The user interface resulting from the above extension declaration with code integration settings and custom settings page is illustrated below.

• The extension is a `"Target"` with name `"My Extension"`:

```
"Kind"="Target"
"DisplayName"="My Extension"
```
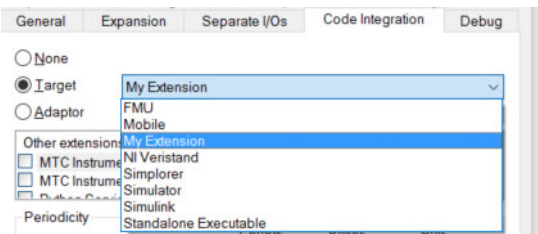

• The extension supports only C language:

```
"Languages"="c"
```

• The extension supports only 64 bits compilation:

```
"CPU"="win64"
```

- The binary produced by the extension is not runnable:

```
"AllowRun"="false"
```

All of the above attributes in extension declaration are applied to the corresponding settings and commands displayed in the user interface.

- The extension references a custom settings page:

```
"Server1"="MyExtensionSettings"
```

When the extension is selected, a new **My Extension** tab is displayed allowing to set a dummy parameter called "Param 1":

The project file content looks as follows:

```
<Prop id="510" name="@GENERATOR:TARGET_ADAPTOR">
    <value>MyExtension</value>
    <configuration>16</configuration>
</Prop>
<Prop id="511" name="@MYEXTENSION:PARAM1">
    <value>dummyval</value>
    <configuration>16</configuration>
</Prop>
```

Generation Module

A generation module allows to package a set of generation services. Such module must be added to the `.srg` file previously defined for Code Generator extension registration.

A generation module has an identifier and defines the pathname of the corresponding Python script. The pathname is either relative to the `SCADE\scripts` directory of the SCADE products installation or it is an absolute pathname.

Example: MyModule.srg

```
[Scade/Generator/MyExtension]
"Pathname"="MyExtension.py"
```

A generation module declares a set of generation services to SCADE Suite Code Generator. A generation service has a name and three interface methods: initialization, generation, and build. The Python script must provide a `<classname>` class where `<classname>` is the base filename of the script implementing a `get_services()` class method returning the list of defined generation services with their name as well as their interface methods.

The syntax of the returned list is:

```
[("<name>", ("-OnInit", <init_method>), ("-OnGenerate", <generate_method>), ("-
OnBuild", <build_method>))*]
```

The methods signatures are defined as detailed in <u>"Generation Service"</u> on page 49. When SCADE Suite Code Generator starts, it searches the declared generation modules for the selected extension(s) and initializes all contained generation services.

Example: MyExtension.py

```
# base filename of the script
class MyExtension:

    # list of services provided by the module
    @classmethod
    def get_services(cls):
        service1 = ("MyService1", ("-OnInit", MyExtension.service1_init), ('-
OnGenerate', MyExtension.service1_generate), ('-OnBuild',
MyExtension.service1_build))
        # add more services if required
        return [service1]
```

Generation Service

As introduced previously, a generation service has a name and three interface methods: initialization, generation, and build. Each method is detailed hereafter.

INITIALIZATION

Signature:

```
def <init_method> (cls, target_dir, project, configuration)
```

`<init_method>` must be used for internal initializations, of course, but one of its main usage is both to involve other generation services and add ordering constraints with respect to other services. It is expected to return the list of required generation services, specifying for each of them if it must be executed before or after another service.

Example: MyExtension.py (continued)

```
class MyExtension:
[…]
    # list of external service dependencies
    @classmethod
    def service1_init(cls, target_dir, project, configuration):
        tu = ("Type Utils", ("-Order", "Before"))
        return [tu]
```

This means that "MyService1" needs the service "Type Utils" which must be executed before "MyService1".

By default, the service "Code Generator" (KCG) is called by scade -code before all other generation services. It is not necessary to add it in the dependencies list of a custom service. But if a service needs to be executed before KCG, the <init_method> must provide its dependency to "Code Generator" service with "After" order:

```
class MyExtension:
[…]
    # list of external service dependencies
    @classmethod
    def service1_init(cls, target_dir, project, configuration):
        cg = ("Code Generator", ("-Order", "After"))
        return [cg]
```

This means that "MyService1" needs the "Code Generator" service to be executed after itself: that is "MyService1" needs to be executed before service "Code Generator".

A given service may not need KCG generation. For instance, it is possible to generate configuration files, only by retrieving information from models (SCADE Suite and/or SCADE Architect). To deactivate KCG code generation, <init_method> must set the "GENERATOR:DISABLE_CG" project property to true:

```
class MyExtension:
[…]
    @classmethod
    def service1_init(cls, target_dir, project, configuration):
        project.set_tool_prop("GENERATOR", "DISABLE_CG", "true", configuration)
        return []
```

GENERATION/BUILD

Both Generation and Build methods have the following signature:

```
def <generate_method>|<build_method> (cls, target_dir, project, configuration)
```

- <generate_method> is called when "Generate" command is invoked.

- `<build_method>` is called when the "`Build`" or "`Rebuild All`" commands are invoked (only available when the extension kind is "`Target`").

To generate or build one's own code, one may need some information from the project settings, the model or linked models, or generated code (KCG), one can:

- retrieve extension settings in input `<project>` and `<configuration>` (for accessing project data, see "Accessing Project Content (ETP)" on page 10):

```
param1 = project.get_scalar_tool_prop_def("MYEXTENSION", "PARAM1", "defaultval",
configuration)
```

- retrieve information from input models (for accessing model data from SCADE models, see Chapter 2 about "Data Access"):

```
# load SCADE Suite model
suite_model = scade.model.suite.get_roots()[0].model
# load SCADE Architect model
architect_model = scade.model.architect.get_roots()[0]
[…]
```

- retrieve information from mapping file generated by KCG, MCG, or ACG:

```
mapping_file_path = pathlib.Path(target_dir) / "mapping.xml"
mapping = scade.code.suite.mapping.open_mapping(str(mapping_file_path))
```

- generate all required files in `<target_dir>` or in any other place:

```
# generate dummy .c file
c_dummy_name = "dummy.c"
c_dummy_pathname = pathlib.Path(target_dir) / c_dummy_name
f = open(str(c_dummy_pathname), 'w')
f.write("\nvoid dummy(){}\n")
f.close()
```

During the code generation of one's own extension, it is possible to communicate with the code generation general service (`scade -code`) through the Code Generator API (see "Description of Code Generator API" on page 53):

- Add Make directives:

```
# tells scade -code to compile the generated dummy.c
scade.code.suite.sctoc.add_c_files([c_dummy_name], False, "MyService1")
# tells scade -code to build a DLL including KCG generated code + mydllmain.c +
files generated by service "Type Utils" and its dependencies
scade.code.suite.sctoc.add_dynamic_library_rule("MyDll", [c_dllmain_name], [], [],
"Type Utils", True)
```

- Fill "Generated Files" section in sc2c.log file. When scade -code is used from GUI, the section is displayed in **Output** docking window, **Info Log** tab:

```
# generated files
scade.code.suite.sctoc.add_generated_files("MyService1", [c_dummy_name,
c_dllmain_name])
```

- Display messages in sc2c.log file. When scade -code is used from GUI, the section is displayed in **Output** docking window, **Info Log** tab:

```
# display messages
scade.code.suite.sctoc.add_warning("MyService1", "W001", [("warning msg","")])
```

Example 1: Resulting Makefile win64

```
win64/dummy.o: dummy.c
gcc [options] "dummy.c" -o "win64\dummy.o"
win64/mydllmain.o: mydllmain.c
gcc [options] "mydllmain.c" -o "win64\mydllmain.o"
MyDll.dll: [dependencies object files] win64/mydllmain.o
gcc [options]  [dependencies object files] "win64\mydllmain.o" [options]
-o "MyDll.dll"
```

Example 2: Resulting **Info Log** tab when launched from SCADE user interface
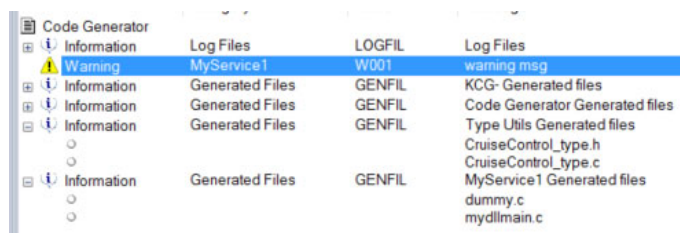
Description of Code Generator API

The Code Generator API is a set of functions in a `scade.code.suite.sctoc` module allowing any custom extension to interact with `scade -code`.

Several categories of functions can be used for:

- "Getting Information from Input Project and Model"
- "Adding Make Directives"
- "Sending Feedback to SCADE Suite User Interface"
- "Helpers"

Getting Information from Input Project and Model

The following get functions are available:

```
get_operator_sample_time()
```

Returns a tuple (period, offset, periodic) containing the settings defined in **Periodicity** frame of **Code Integration** tab.

Return type: (float, float, boolean)

```
get_list_of_project_files (file_ext, …)
```

Returns a list of file pathnames referenced by the current project and its libraries, filtered according to provided file extensions.

Parameters:

- `file_ext (str)` – Filter applied on the returned pathnames list

Return type: list[str]

```
get_list_of_project_files (file_ext=None, project=None, configuration=None)
```

Returns a list of file pathnames referenced by provided project and library projects, filtered according to provided file extensions. Returns also graphical panels model files, only if the provided configuration implies graphical panels.

Parameters:

- `file_ext (list[str])` – Optional; allows to filter returned pathnames list
- `project (Project)` – Optional; if not provided, the current project is used
- `configuration (Configuration)` – Optional; if not provided, the current configuration is used

Return type: list[str]

```
get_list_of_external_files (kind, …)
```

Returns a list of external file pathnames referenced by the current project and its libraries, filtered according to provided file kind.

Parameters:

- `kind (str)` – Optional, possible values are: "CS": Source File for C ; AdaS: "Source file for Ada" ; "Obj": Object File ; "Macro": Macro File ; "Type", Type definition file

Return type: list[str]

Adding Make Directives

The following compilation and link directives are available:

`add_c_files` (c_files, relative_to_project, service)

Requests `scade -code` to add sources files to the Makefile for C build.

Parameters:

- `c_files` (`list[str]`) – List of source file pathnames
- `relative_to_project` (`bool`) – If set to true, the provided `c_files` are considered as relative to the project
- `service` (`str`) – Name of the service; it allows another service to add a dependency to the file generated by the current service

`add_ada_files` (ada_files, relative_to_project, service)

Requests `scade -code` to add sources files to the Makefile for Ada build.

Parameters:

- `ada_files` (`list[str]`) – List of source files pathnames
- `relative_to_project` (`bool`) – If set to true, the provided `ada_files` are considered as relative to the project
- `service` (`str`) – Name of the service; it allows another service to add a dependency to the file generated by the current service

`add_obj_files` (obj_files, relative_to_project)

Requests `scade -code` to add object files to the Makefile.

Parameters:

- `obj_files` (`list[str]`) – List of object file pathnames
- `relative_to_project` (`bool`) – If set to true, the provided `obj_files` are considered as relative to the project

```
add_include_files (include_files, relative_to_project)
```

Requests `scade -code` to add include directories directives to the Makefile.

Parameters:

- `include_files (list[str])` – List of include file pathnames
- `relative_to_project (bool)` – If set to true, the provided `include_files` are considered as relative to the project

```
add_dynamic_library_rule (basename, c_files, o_files, def_files, dependencies,
main, forced_cpu_type, language)
```

Requests `scade -code` to add a dynamic library (.dll) build rule to the Makefile.

Parameters:

- `basename (str)` – Basename of the DLL
- `c_files (list[str])` – List of source file pathnames part of the DLL
- `o_files (list[str])` – List of object file pathnames part of the DLL
- `def_files (list[str])` – List of DLL definition files
- `dependencies (list[str])` – List of services dependencies. Files generated by these services are included in the DLL
- `main (bool)` – If set to true, DLL link rule is added to the "all" rule of the Makefile
- `forced_cpu_type (str)` – If set, the **CPU Type** selected in **Compiler** tab is ignored and the DLL is built with the provided one
- `language (str)` – Target language (c/ada)

```
add_static_library_rule (basename, c_files, o_files, main, forced_cpu_type,
language)
```

Requests `scade -code` to add a static library (.lib) build rule to the Makefile.

Parameters:

- `basename (str)` – Basename of the library
- `c_files (list[str])` – List of source file pathnames part of the library
- `o_files (list[str])` – List of object file pathnames part of the library
- `main (bool)` – When set to true, library link rule is added to the "all" rule of the Makefile
- `forced_cpu_type (str)` – If set, the **CPU Type** selected in **Compiler** tab is ignored and the library is built with the provided one
- `language (str)` – Target language (c/ada)

```
add_executable_rule (basename, c_files, o_files, dependencies, main,
forced_cpu_type, language)
```

Requests `scade -code` to add an executable (.exe) build rule to the Makefile.

Parameters:

- `basename (str)` – Basename of the executable
- `c_files (list[str])` – List of source file pathnames part of the executable
- `o_files (list[str])` – List of object file pathnames part of the executable
- `dependencies (list[str])` – List of services dependencies. Files generated by these services are included in the executable
- `main (bool)` – When set to true, executable rule is added to the "all" rule of the Makefile
- `forced_cpu_type (str)` – If set, the **CPU Type** selected in **Compiler** tab is ignored and the executable is built with the provided one
- `language (str)` – Target language (c/ada)

```
add_custom_rule (basename, dependencies, commands, main, forced_cpu_type,
language)
```

Requests `scade -code` to add a custom build rule to the Makefile.

Parameters:

- `basename (str)` – Name of the rule
- `dependencies (list[str])` – List of services dependencies. Files generated by these services are compiled before this rule
- `commands (list[str]` – List of custom commands added to the rule
- `main (bool)` – When set to true, the custom rule is added to the "all" rule of the Makefile
- `forced_cpu_type (str)` – If set, the **CPU Type** selected in **Compiler** tab is ignored and the custom rule is built with the provided one
- `language (str)` – Target language (c/ada)

```
add_variable (name, value)
```

Requests `scade -code` to add line `<variable>=<value>` to the Makefile.

Parameters:

- `name (str)` – Name of the variable
- `value (str)` – Value of the variable

```
add_path_variable (name, value, relative_to_project)
```

Requests `scade -code` to add line `<variable>=<path>` to the Makefile.

Parameters:

- `name (str)` – Name of the variable
- `value (str)` – Value of the variable
- `relative_to_project (bool)` – If set to true, the provided value is considered as relative to the project

```
set_compiler_kind (kind)
```

Specifies `scade -code` the kind of compiler expected to be used.

Parameters:

- `kind (str)` – Compiler kind; possible values are: "`Cross`", "`Native`"

```
add_preprocessor_definitions (definition,…)
```

Requests `scade -code` to add preprocessor definitions to the Makefile.

Parameters:

- `definition (str)` – Preprocessor definition(s)

```
get_compiler_object_directory ()
```

Returns the object directory for the selected compiler and CPU Type.

Return type: str

Sending Feedback to SCADE Suite User Interface

`scade -code` generates a log file named `sc2c.log` containing generated files but also messages for the user. When `scade -code` is launched from SCADE Suite user interface, `sc2c.log` is parsed to fill the **Info Log** tab on the **Output** Docking Window.

The following functions are available for adding generated files and messages (information, warning error) to `sc2c.log` and to the user interface:

```
add_error (category, code, messages)
```

Displays error messages.

Parameters:

- category (str) – Error category
- code (str) – Error code
- messages (List[(str, str)]): Error messages list; each message object being a tuple (message, localization)

```
add_warning (category, code, messages)
```

Displays warning messages.

Parameters:

- `category (str)` – Warning category
- `code (str)` – Warning code
- `messages (List[(str, str)])` – Warning messages list; each message object being a tuple (message, localization)

```
add_information (category, code, messages)
```

Displays information messages.

Parameters:

- `category (str)` – Information category
- `code (str)` – Information code
- `messages (List[(str, str)])` – Information messages; each message object being a tuple (message, localization)

```
add_generated_files (name, files)
```

Display the list of files generated by the extension.

Parameters:

- `name (str)` – Identifier of the extension
- `list (List[str])` – List of file pathnames

Helpers

Additionally, the following helpers are available:

`raw_tcl` (script)

Executes a TCL command and returns the resulting string.

Parameters:

- `script (str)` – TCL command

Return type: str

`get_relative_path` (absolute_path, reference_path)

For a given `absolute_path`, returns a path relative to a given `reference_path`.

Parameters:

- `absolute_path (str)` – Input absolute path
- `reference_path (str)` – Input reference path

Return type: str

Description of WrapGen

The WrapGen API is a set of classes and functions in a `scade.code.suite.wrapgen` module helping the generation of glue code for a given operator. It supports model-specific cases and any combination of code generation options. WrapGen is a layer over SCADE Suite KCG/MCG/ACG Mapping File API (`scade.code.suite.mapping`) supporting KCG C, KCG Ada, MCG, and ACG.

At model-level, the WrapGen API:

- Returns a list of paths for Scade variables (sensors, inputs, outputs, probes)

At code-level, the WrapGen API:

- Returns C/Ada generated variable path and type from Scade variables' path
- Returns `#include/` with statements
- Returns the declarations of KCG context variables and sensors
- Returns a call to each init and/or reset functions and to the cycle function
- Returns calls to scenario recorder API functions

The following file descriptions and examples provide an overview of the API. Detailed documentation is available in Python files (`.py`) themselves.

`__init__.py File`

This file contains global factory functions used to create objects whose classes are described below with the description of Python files.

```
def create_mapping_file(path_mapping)
def create_mapping_helpers(mapping_file)
def create_interface_printer(mapping_helpers, root, sep_ctx=False)
```

`model.py` File

The `scade.code.suite.wrapgen.model.MappingHelpers` class contains methods returning Scade path for sensors, operator inputs/outputs, and probes.

```python
class MappingHelpers:
    # Model-level methods returning the Scade paths of variables:
    def get_sensors(self)
    def get_inputs(self, root='')
    def get_outputs(self, root='')
    def get_probes(self, root='')
    # Retrieve mapping.model.Element from Scade variable path:
    def get_model_element (self, var_path)
```

`c.py` File

The `scade.code.suite.wrapgen.c.InterfacePrinter` class provides methods returning formatted C code that includes the appropriate headers, declares KCG context variables, and calls init and cycle functions.

```python
class InterfacePrinter:
    # Positioning the printer on a given operator:
    def get_operator(self)
    def set_operator(self, path)

    # Getting C path of generated variable from Scade variable path:
    def get_generated_path(self, var_path, subst=None)
    # Getting C type name of generated variable from Scade variable path:
    def get_generated_type_name(self, var_path)
    # Getting mapping.c.Type object from Scade variable path:
    def get_generated_type (self, var_path)

    # Returning string containing C code for wrapping an operator:
    def print_includes(self)
    def print_context_def(self)
    def print_context_decl(self, mode=None)
    def print_sensors_decl(self)
    def print_init_call(self, init_and_reset, init_outs=False)
    def print_cycle_call(self)

    # Returning string containing C code for calling "Scenario Recorder API":
    def init_recorder(self, project_name, record_probes, global_call_data)
    def print_recorder_includes(self)
    def print_start_recorder(self, call_arguments)
    def print_end_recorder(self, call_arguments)
    def print_start_cycle(self, call_arguments)
    def print_end_cycle(self, call_arguments)
    def print_record_inputs(self, call_arguments)
    def print_record_outputs(self, call_arguments)
```

ada.py File

The `scade.suite.wraputils.ada.InterfacePrinter` class provides methods returning formatted Ada code that includes the appropriate packages, declares KCG context variables, and calls init and cycle functions.

```
class InterfacePrinter:
    # Positioning the printer on a given operator:
    def get_operator(self)
    def set_operator(self, path)

    # Getting Ada path of generated variable from Scade variable path:
    def get_generated_path(self, var_path, subst=None)
    # Getting Ada type name of generated variable from Scade variable path:
    def get_generated_type_name(self, var_path)
    # Getting mapping.ada.Type object from Scade variable path:
    def get_generated_type (self, var_path)

    # Returning string containing Ada code for wrapping an operator:
    def print_context_def(self)
    def print_context_decl(self)
    def print_init_call(self, init_and_reset)
    def print_cycle_call(self)

    # Returning string containing Ada code for calling "Scenario Recorder API":
    def init_recorder(self, project_name, record_probes, global_call_data)
    def print_recorder_withs(self)
    def print_start_recorder(self, call_arguments)
    def print_end_recorder(self, call_arguments)
    def print_start_cycle(self, call_arguments)
    def print_end_cycle(self, call_arguments)
    def print_record_inputs(self, call_arguments)
    def print_record_outputs(self, call_arguments)
```

EXAMPLE (C)

An example that generates glue code.

```
# generate glue code for first root operator:
def generate_glue(mapping_file, \
                  is_scnrec_active, \
                  scnrec_probes, \
                  scnrec_global_call_data, \
                  scnrec_project_basename, \
                  glue_pathname):
    mh = scade.code.suite.wrapgen.create_mapping_helpers(mapping_file)
    root_path = mapping_file.get_options()['root'][0]
    prt = scade.code.suite.wrapgen.create_interface_printer(mh, root_path)
    if is_scnrec_active:
        prt.init_recorder(scnrec_project_basename, scnrec_probes, scnrec_global_call_data)
    from scade.code.suite.cghelper import IndentedStream as istream
    with open(glue_pathname, 'w+t') as f:
        ios = istream(f, '  ')
        ios << '/* includes */\n'
        ios    << prt.print_includes()
        if is_scnrec_active:
            ios << prt.print_recorder_includes() << '\n'
        ios << '\n/* sensors */\n'
        ios << prt.print_sensors_decl().strip(' \t\n')
        ios << '\n/* context */\n'
        ios << prt.print_context_def() << '\n\n'
        ios << prt.print_context_decl() << '\n'
        ios << '\nint main(void) {\n'
        ios << istream.INDENT
        if is_scnrec_active:
            ios << 'FILE *f = %s\n' % prt.print_start_recorder('"test.sss"')
        ios << '\n/* sensors */\n'
        for s in mh.get_sensors():
            ios << ('%s; /* %s */\n' % (prt.get_generated_path(s),
prt.get_generated_type_name(s)))
        ios << '\n/* inputs */\n'
        for i in mh.get_inputs():
            ios << ('%s; /* %s */\n' % (prt.get_generated_path(i),
...
```

```
...
prt.get_generated_type_name(i)))
        ios << '\n/* init */\n'
        ios << prt.print_init_call(True, True) << '\n'
        ios << '\n/* cycle */\n'
        ios << prt.print_cycle_call() << '\n'
        ios << '\n/* outputs */\n'
        for o in mh.get_outputs():
            ios << ('%s; /* %s */\n' % (prt.get_generated_path(o),
prt.get_generated_type_name(o)))
        ios << '\n/* probes */\n'
        for p in mh.get_probes():
            ios << ('%s; /* %s */\n' % (prt.get_generated_path(p),
prt.get_generated_type_name(p)))
        if is_scnrec_active:
            ios << '\n/* record scenario */\n'
            ios << prt.print_start_cycle('f') << '\n'
            ios << prt.print_record_inputs('f') << '\n'
            ios << prt.print_record_outputs('f') << '\n'
            ios << prt.print_end_cycle('f') << '\n'
            ios << prt.print_end_recorder('f') << '\n'
        ios << '\nreturn 0;\n'
        ios << istream.OUTDENT
        ios << '}\n'
```

The generated glue code is as follows:

```
/* includes */
#include "Root.h"
#include "kcg_sensors.h"

/* sensors */
kcg_float64 Sensor1;
kcg_float64 Sensor2_P1;
kcg_float64 Sensor3_P2;
/* context */
typedef struct {
  outC_Root outC;
  inC_Root inC;
} WU_Root;
#define WU_CTX_TYPE_Root WU_Root
WU_CTX_TYPE_Root Wu_Ctx_Root;
...
```

```
...
int main(void) {
    FILE *f = start_recorder("test.sss");

    /* sensors */
    Sensor1; /* kcg_float64 */
    Sensor2_P1; /* kcg_float64 */
    Sensor3_P2; /* kcg_float64 */

    /* inputs */
    Wu_Ctx_Root.inC.I_int8; /* kcg_int8 */
    Wu_Ctx_Root.inC.I_uint8; /* kcg_uint8 */

    /* init */
    #ifndef KCG_USER_DEFINED_INIT
      Root_init(&Wu_Ctx_Root.outC);
    #endif /* KCG_USER_DEFINED_INIT */
    #ifndef KCG_NO_EXTERN_CALL_TO_RESET
      Root_reset(&Wu_Ctx_Root.outC);
    #endif /* KCG_NO_EXTERN_CALL_TO_RESET */

    /* cycle */
    Root(&Wu_Ctx_Root.inC, &Wu_Ctx_Root.outC);

    /* outputs */
    Wu_Ctx_Root.outC.O_int8; /* kcg_int8 */
    Wu_Ctx_Root.outC.O_uint8; /* kcg_uint8 */

    /* probes */
    Wu_Ctx_Root.outC.Context_AddFloat64_1.Probe1; /* kcg_float64 */
    Wu_Ctx_Root.outC.Context_AddFloat64_3[0].Probe1; /* kcg_float64 */
    Wu_Ctx_Root.outC.Context_AddFloat64_3[1].Probe1; /* kcg_float64 */
    Wu_Ctx_Root.outC.Context_AddFloat64_3[2].Probe1; /* kcg_float64 */

    /* record scenario */
    start_cycle(f);
    record_inputs(&Wu_Ctx_Operator1.inC, f);
    record_outputs(&Wu_Ctx_Operator1.outC, f);
    end_cycle(f);
    end_recorder(f);

    return 0;
}
```

Predefined Generation Services

Code Generator

This service is responsible for launching SCADE Suite KCG with the options defined in the input project and configuration.

Type Utils

This service is responsible for generating C/Ada functions for operations on generated types (conversion, comparison...). Generated files are:

KCG Version	Generated Files
C	`<ProjectName>_type.h` and `<ProjectName>_type.c`
Ada	`Smuw_type.ads` and `Smuw_type.adb`

C GENERATION

For each generated type, a set of functions are generated, especially:

- Conversion from a variable pointed by `pValue` of `<type>` type to a string pointed by `pStrObj`, using a provided string append function pfnStrAppend:

```
int <type>_to_string(const void *pValue, PFN_STR_APPEND pfnStrAppend, void *pStrObj);
```

- Conversion from a string pointed by `str` to a variable pointed by `pValue`, with update of `endptr` with the last parsed character +1:

```
int string_to_<type>(const char *str, void *pValue, char **endptr);
```

ADA GENERATION

For each generated type, a set of functions are generated in package `Smuw_Type. Smuw_<type>`, especially:

- Conversion from a variable pointed by `Value_Addr` of `<type>` type to a string pointed by `Str_Addr`, using a provided string append function `Str_Append_Func`:

```
function To_String(Value_Addr: System.Address; Str_Append_Func:
Smu_Types.PFN_STR_APPEND; Str_Addr: System.Address) return Interfaces.C.int;
```

- Conversion from a string pointed by `C_Str` to a variable pointed by `Value_Addr`, with update of `End_Ptr` with the last parsed character +1:

```
function String_To(C_Str: Interfaces.C.Strings.chars_ptr; Value_Addr:
System.Address; End_Ptr: access System.Address) return Interfaces.C.int;
```

Multicore Code Integration API

The SCADE Multicore Code Integration API provides various API functions that give access to multicore integration data generated by Multicore Code Generator in a mapping file. This API provides services to:

- represent the dependency graph of methods used for scheduling
- represent an allocation of tasks to workers with an associated file format

It contains several predefined allocation strategies and a schedule checker.

ACCESSING API MODULES

Start a script in Python by referencing the `scade.code.suite.tasks` module and loading the appropriate module as in:

```
from scade.code.suite.tasks import allocation
```

ACCESSING API DOCUMENTATION

All API documentation is available in HTML format from the default browser.

Launch API documentation

EXAMPLE

The following code snippet loads an allocation from a JSON file, verifies this allocation, and provides the list of methods for each worker:

```python
mapping = c.open_mapping('/path/to/mapping.xml')
allocation = allocation.JsonAllocation(mapping, '/path/to/allocation.json')
# check allocation
try:
    allocation.check_allocation()
except AllocationException as e:
    print('Error: ' + e.msg)
# print methods in worker
for w in allocation.get_task_allocation():
    print('- Worker: %s\n' % w.get_index())
    for mt in w.get_methods():
        print('  - %s\n' % mt.get_name())
```

4 / APIs for Dedicated SCADE Activities

This chapter introduces the APIs available for accessing data and results produced by different SCADE activities. Activities include test execution, model coverage measurements, timing and stack measurements, model synchronization between SCADE Architect and SCADE Suite, connections between behavior designed in SCADE Suite and graphical specifications designed in SCADE Display or SCADE Rapid Prototyper, or connections between ARINC 661 UA behavior designed in SCADE Suite and UA DF models designed in SCADE UA Page Creator.

- "Test Results API"
- "Model Coverage API"
- "Timing and Stack API"
- "Synchronization API"
- "Graphical Panel Coupling API"

Test Results API

The data produced by SCADE Test Environment for Host can be accessed using the Test Results API as detailed below.

API Presentation

The Test Results API enables you to get all results produced by SCADE Test Environment for Host: status of records as well as details on each check information. This is a read/write Python API.

Details about accessible test results data are available from the project metamodel.

Note

For accessing results from test environment using Python API, refer to <u>"Test Environment for Host Metamodels (Python)"</u> in *SCADE Metamodels Technical Reference Card*.

The Test Results API sources are contained in the **scade.model.testenv** Python package.

To access all API root elements, it is necessary to call **scade.model.testenv.get_roots()**.

Example

The following is an example for obtaining the results of each record in a procedure.

```python
import scade, scade.model.testenv
def outputln(text):
    scade.output(text + '\n')
for app in scade.model.testenv.get_roots():
    for result_file in app.result_files:
        procedure = result_file.procedure
        if procedure != None:
            name = procedure.name
            outputln(str(name))
        raw_file = result_file.raw_file
        if raw_file != None:
            test_procedure = raw_file.test_procedure
            if test_procedure != None:
                name = test_procedure.name
                operator = test_procedure.operator
                outputln('Procedure: ' + str(name))
                outputln('Tested Operator: ' + str(operator))
            for record in raw_file.result_records:
                outputln((str(record.name)) + " successful:" + str(record.get_ok()) + "/" +
str(record.get_total()))
```

Model Coverage API

The data produced by SCADE Test Model Coverage for SCADE Suite models can be accessed using the Model Coverage API as detailed below.

API Presentation

The Model Coverage API for SCADE Suite models enables to access the model coverage results produced by Model Coverage for SCADE Suite. This is a read Python API.

The Model Coverage API sources are contained in the **scade.test.suite.mcoverage** Python package.

To access all API root elements, it is necessary to call

scade.test.suite.mcoverage.report.Report(json_file)

where json_file is the output result report produced by Model Coverage execution.

All API documentation is available in HTML format from the default browser.

[Launch API documentation](#)

Example

The following is an example for reading the coverage report and displaying the list of coverage points as well as the percentage of covered or justified points.

```
report = scade.test.suite.mcoverage.report.Report(json_filename)

# print coverage points
for p in report.get_coverage_points():
    print('%s %s %s\n' % (p.get_pretty_status().upper(), p.get_pretty_kind().upper(),
p.get_path()))

# Count the number of coverage points
nb_cov_points = report.get_coverage_points().count()
# Covered and not justified
nb_covered = cov_points.filter_status([S.OBSERVED]).count()
# Covered and justified
nb_justif_cov = cov_points.filter_status(S.OBSERVED_JUSTIFIED).count()
# Justified and not covered points
nb_justif = cov_points.filter_status(S.JUSTIFIED).count()

# Print the percentage of covered or justified coverage points
total_percent = (nb_covered+nb_justif_cov+nb_justif) / nb_cov_points
print("Covered or justified / Total : %.2f" % total_percent)
```

Timing and Stack API

The data produced by SCADE Suite for Timing and Stack analysis can be accessed using the Timing and Stack API as detailed below.

API Presentation

The Timing and Stack API enables you to get all information and results produced by SCADE Suite Timing and Stack Analysis tools: sessions attached to SCADE Suite project (all `.scts` files), operators analyzed by session, routines (init, reset, cycle) of each operator, and all timing or stack measures by routines. This is a read-only Python API.

Details about accessible analysis results data are available from the project metamodel.

Note

For accessing results from timing and stack analysis tools using Python API, refer to "Timing and Stack Analysis Metamodel (Python API)" in *SCADE Metamodels Technical Reference Card*.

The Timing and Stack API sources are contained in the **scade.tool.suite.timingstack** Python package.

To access all API root elements (a list of objects from **Application** class), it is necessary to call **scade.tool.suite.timingstack.get_roots()**.

Helper Functions

Some helper functions are available with this API from **`Application`** class.

`get_all_time_measures` `(operator_path, routine_type)`

Returns all time results for a given operator and routine type sorted by result date and time.

Parameters:

- `operator_path` `(str)` – SCADE operator path
- `routine_type` `(str` or `list[str])` – Optional, default value is `RK_CYCLE`. Other possible values: `RK_INIT` or `RK_RESET`

Return type: list[Time]

`get_all_space_measures` `(operator_path, routine_type)`

Returns all space results for a given operator and routine type sorted by result date and time.

Parameters:

- `operator_path` `(str)` – SCADE operator path
- `routine_type` `(str` or `list[str])` – Optional, default value is `RK_CYCLE`. Other possible values: `RK_INIT` or `RK_RESET`

Return type: list[Space]

Example

The following example lists the results of Timing and Stack analysis sessions.

```
import scade
import scade.tool.suite.timingstack as tas
tasapp = tas.get_roots()[0]
results = tasapp.results.copy()
results.sort(key = lambda k: k.date)

# 1/  Print all results found in input project:
# - all sessions (timing or stack),
# - all instantiated operators,
# - all routine kind (init, reset or cycle)
# - all measure kind (WCET or Stack)
# For each session...

for result in results:
    scade.output(result.date + ': ' + result.path + '\n')
    # parse Configuration class (exists only if inpu t.scts contains an option with
name="Configuration"):
    if result.configuration:
        scade.output('Configuration: ' + result.configuration.name + '\n')
        for p in result.configuration.props:
            scade.output('  + prop ' + p.name + ': ' + str(p.values) + '\n')
    # For each instantiated operator...
    for operator in result.operators:
        scade.output(' ' + operator.name + '\n')
        # For each kind of routine...
        for routine in operator.routines:
            scade.output('  ' + routine.name + ' / '
                    + ('Time' if isinstance(routine.measure, tas.Time) else 'Space') + '\n')
            # ... print WCET information
            if isinstance(routine.measure, tas.Time):
                t = routine.measure
                scade.output('    WCET(sum): \t' + str(t.wcet_sum) + ', \tWCET(max): \t' +
str(t.wcet_max)
                             + ', \tWCET(avg): \t' + str(t.wcet_avg) + '\n')
                scade.output('    CWCET(sum): \t' + str(t.c_wcet_sum) + ', \tCWCET(max): \t'
+ str(t.c_wcet_max)
                             + ', \tCWCET(avg): \t' + str(t.c_wcet_avg) + '\n')
            # ... print Stack information
            elif isinstance(routine.measure, tas.Space):
                s = routine.measure
            scade.output('   Stack: ' + str(s.stack) + ', \tCStack: ' + str(s.c_stack) + '\n')
    scade.output('\n')
```

```
# 2/ Print all WCET measures on cycle routine of the ABC_N::Button operator
time = tasapp.get_all_time_measures('ABC_N::Button', tas.RK_CYCLE)
scade.output('tas.get_all_time_measures(\'ABC_N::Button\', RK_CYCLE):\n')
for t in time:
    scade.output(' ' + t.routine.operator.result.date + '\n')
    scade.output('  ' + t.routine.operator.name + '/' + t.routine.name + '\n')
    scade.output('   WCET(sum): \t' + str(t.wcet_sum) + ', \tWCET(max): \t' + str(t.wcet_max)
                 + ', \tWCET(avg): \t' + str(t.wcet_avg) + '\n')
    scade.output('   CWCET(sum): \t' + str(t.c_wcet_sum) + ', \tCWCET(max): \t' +
str(t.c_wcet_max)
                 + ', \tCWCET(avg): \t' + str(t.c_wcet_avg) + '\n')
    scade.output('\n')

# 3/ Print all Stack measures on reset routine of the ABC_N::Button operator
space = tasapp.get_all_space_measures('ABC_N::Button', tas.RK_RESET)
scade.output('tas.get_all_space_measures(\'ABC_N::Button\', RK_RESET):\n')
for s in space:
    scade.output(' ' + s.routine.operator.result.date + '\n')
    scade.output('  ' + s.routine.operator.name + '/' + s.routine.name + '\n')
    scade.output('   Stack: ' + str(s.stack) + ', \tCStack: ' + str(s.c_stack) + '\n')
    scade.output('\n')
```

Synchronization API

The data produced by SCADE Architect for synchronizing SCADE Architect and SCADE Suite models can be accessed using the Synchronization API as detailed below.

- *"API Presentation"*
- *"API Objects"*
- *"Examples"*

API Presentation

The Synchronization API enables you to get all information produced by SCADE Architect Synchronization tool which is saved in `.synchro` file. This is a read-only Python API.

Details about accessible synchronization data are available from the project metamodel.

Note

For accessing results from synchronization tools using Python API, refer to *"Synchronization Metamodel"* in *SCADE Metamodels Technical Reference Card*.

The function to access the root of the Synchronization API is:

```
scade.architect.get_synchronization_api(projectpath)
```

- `projectpath` is the path to the SCADE Architect or Suite project

The function returns the root Synchronization API for the project at the given path if it exists.

API Objects

It is possible to retrieve values or navigate through the Synchronization API from the root Synchronization API for a project following the relations and functions defined in the API.

All operation functions must be called as follows:

```
scade_api.call(<ApiObject>, <FunctionName>, <Param1>, <Param2>, ...)
```

SyncApi Object

The following are accessible for this API object:

References:

`root : Root`

A reference to the corresponding Root for which SyncApi is the wrapper

Operations:

`getAllSettingsApi() : List<SettingsApi>`

Returns a list of SettingsApi wrapping each of the Settings of the Root wrapped by the current SyncApi

`getSettingsApiById(id : EString) : SettingsApi`

Returns the SettingsApi wrapping the Settings with the id given as parameter

`getSettingsApiByName(name : EString) : SettingsApi`

Returns the SettingsApi wrapping the Settings with the name given as parameter

SettingsApi Object

The following are accessible for this API object:

Attributes:

References:

```
settings : Settings
```
A reference to the corresponding Settings for which SettingsApi is the wrapper

Operations:

```
getReferencedProjectAbsolutePath() : EString
```
Returns the absolute path of the referenced project or the wrapped synchronization Settings

```
getSynchronizedElementsOid(oid: EString) : List<EString>
```
Returns the list of internal ids of the synchronized elements for the element with the internal id given as parameter

```
getMainSynchronizedElementOid(oid : EString) : EString
```
Returns the internal id of the main synchronized element for the element with the internal id given as parameter

```
getSynchronizedElements(obj : EObject) : List<EObject>
```
Returns the list of synchronized elements for the element given as parameter

```
getMainSynchronizedElement(obj : EObject) : EObject
```
Returns the main synchronized element for the element given as parameter

```
getSynchronizedElementsByOid(oid: EString) : List<EObject>
```
Returns the list of synchronized elements for the element with the internal id given as parameter

```
getMainSynchronizedElementByOid(oid : EString) : EObject
```
Returns the main synchronized element for the element with the internal id given as parameter

Root Object

The following are accessible for this API object:

Attributes:

`lastSettingID : EInt`
The id of the last Settings used in the current synchronization file

`activeSetting : EInt`
The id of the active synchronization Settings for the current synchronization file

References:

`settings : List<Settings>`
The list of synchronization Settings defined in the current synchronization file

Operations:

`getSettings(settingsId : EInt) : Settings`
Returns the Settings with the id given as parameter

`getSettings(settingsName : EString) : Settings`
Returns the Settings with the name given as parameter

Settings Object

The following are accessible for this API object:

Attributes:

`id : EInt`
The id of the current Settings

`hidden : EBoolean`
A Boolean value specifying if the current Settings is hidden

`obsolete : EBoolean`
A Boolean value specifying if the current Settings is obsolete

References:

`properties : List<Property>`
The list of synchronization properties of the current Settings

`trace : Trace`
The QVTo trace root of the last execution of the current Settings

Operations:

```
getProperty(propertyName : EString) : Property
```
Returns the Property with the name given as parameter

```
getPropertyValue(propertyName : EString) : EString
```
Returns the value of the Property with the name given as parameter

Property Object

The following are accessible for this API object:

Attributes:

```
name : EString
```
The name of the current Property

```
value : EString
```
The value of the current Property

Examples

The following script examples illustrate how API functions can be used to retrieve and access information from SCADE Architect-SCADE Suite synchronization activities.

Example 1: Retrieves the root Synchronization API of the opened SCADE
Architect project, the Settings API of active synchronization
settings, and SCADE Suite synchronized elements for all packages in
the main model, blocks in packages, and ports in blocks.

```
import scade, scade.model.architect
def get_project_by_name(modelname) :
        for item in scade.model.project.stdproject.get_roots():
            vpathname = item.pathname
            if vpathname.endswith(modelname + '.etp') :
                return item
        return
def get_project_by_path(path) :
        for item in scade.model.project.stdproject.get_roots():
            vpathname = item.pathname
            if vpathname == path :
                return item
        return
def search_synched(e, suitesyncproject, syncsettapi):
        oid = e.oid
        scade.output('Search synched element for (' + oid + '): ' + e.qualified_name + '\n')
        suitesynchedoid = _scade_api.call(syncsettapi, "getMainSynchronizedElementOid", oid)
        if not (suitesynchedoid is None) :
                scade.output('found sync elt oid: ' + suitesynchedoid + '\n')
                # TODO find element with oid from suite project
for model in scade.model.architect.get_roots():
        scade.output('model: ' + model.name + '\n')
        project = get_project_by_name(model.name)
        scade.output('project: ' + project.pathname + '\n')
        syncapi = scade.architect.get_synchronization_api(project.pathname)
        scade.output('sync api: ' + str(syncapi) + '\n')
        activesetting = syncapi.root.active_setting
      #syncsettapi = _scade_api.call(syncapi, "getSettingsApiByName", 'Synchronization with
test2')
        syncsettapi = _scade_api.call(syncapi, "getSettingsApiById", str(activesetting))
        scade.output('sync sett api: ' + str(syncsettapi) + '\n')
        suitesyncprojectpath = _scade_api.call(syncsettapi,
"getReferencedProjectAbsolutePath")
        scade.output('sync suite project path: ' + str(suitesyncprojectpath) + '\n')
        suitesyncproject = get_project_by_path(suitesyncprojectpath)
        scade.output('project: ' + str(suitesyncproject) + '\n')
        scade.output('    ### Search for synchronized elements ###\n')
        for p in model.packages:
                search_synched(p, suitesyncproject, syncsettapi)
                for b in p.blocks:
                        search_synched(b, suitesyncproject, syncsettapi)
                        for po in b.ports:
                                search_synched(po, suitesyncproject, syncsettapi)
```

Example 2: Retrieves the root Synchronization API of the opened SCADE Suite project, the Settings API of active synchronization settings, and SCADE Architect synchronized elements for all packages in the main model, operators in packages, and variables in operators.

```python
import scade, scade.model.architect
def get_project_by_name(modelname) :
        for item in scade.model.project.stdproject.get_roots():
            vpathname = item.pathname
            if vpathname.endswith(modelname + '.etp') :
                return item
        return
def get_project_by_path(path) :
        for item in scade.model.project.stdproject.get_roots():
            vpathname = item.pathname
            if vpathname == path :
                return item
        return
def search_synched(e, suitesyncproject, syncsettapi):
        oid = e.get_oid()
        scade.output('Search synched element for (' + str(oid) + '): ' + e.name + '\n')
        archsynchedoid = _scade_api.call(syncsettapi, "getMainSynchronizedElementOid", oid)
        if not (archsynchedoid is None) :
                scade.output('found sync elt oid: ' + archsynchedoid + '\n')
        archsynched = _scade_api.call(syncsettapi, "getMainSynchronizedElementByOid", oid)
        if not (archsynched is None) :
                scade.output('found sync uml elt: ' + archsynched.name + '\n')
for root in scade.model.suite.get_roots():
        model = root.model
        scade.output('model: ' + model.name + '\n')
        project = get_project_by_name(model.name)
        scade.output('project: ' + project.pathname + '\n')
        syncapi = scade.architect.get_synchronization_api(project.pathname)
        scade.output('sync api: ' + str(syncapi) + '\n')
        activesetting = syncapi.root.active_setting
        syncsettapi = _scade_api.call(syncapi, "getSettingsApiById", str(activesetting))
        scade.output('sync sett api: ' + str(syncsettapi) + '\n')
        architectsyncprojectpath = _scade_api.call(syncsettapi,
"getReferencedProjectAbsolutePath")
        scade.output('sync suite project path: ' + str(architectsyncprojectpath) + '\n')
        architectsyncproject = get_project_by_path(architectsyncprojectpath)
        scade.output('project: ' + str(architectsyncproject) + '\n')
        scade.output('    ### Search for synchronized elements ###\n')
        for p in model.packages:
                search_synched(p, architectsyncproject, syncsettapi)
                for o in p.operators:
                        search_synched(o, architectsyncproject, syncsettapi)
                        for v in o.variables:
                                search_synched(v, architectsyncproject, syncsettapi)
```

Graphical Panel Coupling API

The data used to connect SCADE Suite operators to SCADE Display or SCADE Rapid Prototyper graphical panels can be accessed using the Graphical Panel Coupling API as detailed below. You can also use this API to access the data used to connect UA behavior models designed in SCADE Suite and UA DF models designed in SCADE UA Page Creator. Regardless of whether graphical panels are created with SCADE Display, SCADE UA Page Creator, or SCADE Rapid Prototyper, the API is the same.

- "API Presentation"
- "API Functions"
- "Example"

API Presentation

The Graphical Panel Coupling API enables to access connection data produced when connecting graphical panels between SCADE Suite and SCADE Display, SCADE UA Page Creator, or SCADE Rapid Prototyper. The connection data is saved in an `.sdy` file. This is a read/write Python API. Details about accessible data are available from the project metamodel.

Note

For accessing connection data using Python API, refer to "Graphical Panel Coupling Metamodel (Python API)" in *SCADE Metamodels Technical Reference Card*.

The Graphical Panel Coupling API sources are contained in the **scade.model.suite.displaycoupling** Python package.

To access all API root elements (a list of `SdyApplication` objects), it is necessary to call **scade.model.suite.displaycoupling.get_roots()**. The following sections will demonstrate how to use it.

API Functions

Hereafter, we refer to "*logics-side*" for SCADE Suite connection points and to "*graphics-side*" for SCADE Display/SCADE Rapid Prototyper/SCADE UA Page Creator connection points.

The Graphical Panel Coupling API functions allowing to manipulate connections are described below:

- "List Graphics-Side Entities"
- "List Logics-Side Entities"
- "List Existing Connections"
- "Create or Delete Connections"
- "Save Changes"

List Graphics-Side Entities

You can use Graphical Panel Coupling API to list graphics-side entities, regardless whether they are connected or not using the following functions:

```
class SdyApplication:
    [...]
    def ensure_mapping_plugs(self) -> None:
```

Ensures all possible MappingPlugs, that wrap SCADE Display/ARINC 661 plugs, are created in the API. It is necessary to call this function before manipulating graphics-side entities.

Use the following algorithm to list all graphics-side entities:

```
import scade, scade_env
import scade.model.suite.displaycoupling

scade_env.load_project('path/to/my/project.etp')

app = scade.model.suite.displaycoupling.get_roots()[0] # Only one project loaded
app.ensure_mapping_plugs()
for plug in app.mapping_plugs:
    scade.output(f'{plug.full_name}\n')
```

List Logics-Side Entities

You can use Graphical Panel Coupling API to list logics-side entities, regardless of whether they are connected or not using the following functions:

```
class SdyApplication:
    [...]
    def ensure_mapping_variables(self, scade_operator :
'scade.model.suite.Operator') -> 'MappingOperator':
```

Ensures all possible MappingVariables, that wrap SCADE Suite variables, are created in the API for a given SCADE Suite operator. It is necessary to call this function before manipulating logics-side entities.

Returns the MappingOperator of the API mapping the SCADE Suite operator.

Use the following algorithm to list all logics-side entities:

```
import scade, scade_env
import scade.model.suite.displaycoupling
etp_path, logic_operator_path = 'path/to/my/project.etp', 'PACKAGE::OPERATOR'
scade_env.load_project(etp_path)

app = scade.model.suite.displaycoupling.get_roots()[0] # Only one project loaded
logic_op = app.mapping_file.model.get_object_from_path(logic_operator_path)
app.ensure_mapping_variables(logic_op)
for var in app.mapping_variables:
    scade.output(f'{var.full_name}\n')
```

List Existing Connections

You can make an API call to list existing connections by name.

Use the following algorithm to list all connections:

```
import scade, scade_env
import scade.model.suite.displaycoupling
from scade.model.suite.displaycoupling import MappingOperator
etp_path, logic_operator_path = 'path/to/my/project.etp', 'PACKAGE::OPERATOR'
scade_env.load_project(etp_path)

app = scade.model.suite.displaycoupling.get_roots()[0] # Only one project loaded
logic_op = app.mapping_file.model.get_object_from_path(logic_operator_path)
mapping_op : MappingOperator = app.ensure_mapping_variables(logic_op)
for conn in mapping_op.mapping_items:
    for receiver in conn.receivers:
        scade.output(f'{conn.sender.full_name} -> {receiver.full_name}\n')
```

Create or Delete Connections

You can make an API call to Create or Delete a connection using the following functions:

```
class MappingOperator:
    [...]
    def connect(self, logical_io : MappingVariable, graphical_io :
SdyEntity, allow_type_mismatch : bool) -> bool:
```

Parameters:

• `logical_io : MappingVariable` – SCADE Suite variable to be considered
• `graphical_io : SdyEntity` – SCADE Display / Rapid Prototyper / A661 entity to be considered
• `allow_type_mismatch` – if `true`, creates the connection even if the types mismatch

Return value: `true` if the connection was created

```
class MappingOperator:
    [...]
    def disconnect(self, logical_io : MappingVariable, graphical_io :
SdyEntity) -> bool:
```

Parameters:

• `logical_io : MappingVariable` – SCADE Suite variable to be considered
• `graphical_io : SdyEntity` – SCADE Display / Rapid Prototyper / A661 entity to be considered

Return value: `true` if the connection was deleted

```
ConnectPlugPred = Callable[['MappingVariable', 'MappingPlug'], bool]
class MappingOperator:
    [...]
    def connect_all_plugs(self, predicate: 'ConnectPlugPred',
allow_type_mismatch : bool) -> None:
```

Allows to connect iteratively all variables of the mapping operator with all entities of the graphical panel. **Function to be used in the context of SCADE Display and SCADE Rapid Prototyper. For ARINC 661 messages, see the `connect_all_messages` function below.**

Parameters:

• `predicate : 'ConnectPlugPred'` – callback for each pair of variable and plug specifying if they should be connected. The callback is called for compatible pairs (plug output with variable input, etc...). This is typically the place where regular expressions can be used with `MappingVariable.full_name` and `MappingPlug.full_name`.
• `allow_type_mismatch` – If `false`, the types of the variable and the plug are checked and the callback is called only if compatible. If `true`, the callback is also called when types do not match.

```
ConnectMessagePred = Callable[['MappingVariable', 'Message'], bool]
class MappingOperator:
    [...]
    def connect_all_messages(self, predicate: 'ConnectMessagePred',
allow_type_mismatch : bool) -> None:
```

Allows to connect iteratively all variables of the mapping operator with all entities of the graphical panel. **Function to be used in the context of ARINC 661 messages**. **For SCADE Display and SCADE Rapid Prototyper, see the `connect_all_plugs` function above.**

Parameters:

- `predicate : 'ConnectMessagePred'` – callback for each pair of variable and message specifying if they should be connected. The callback is called for compatible pairs (message output with variable input, etc..). This is typically the place where regular expressions can be used with `MappingVariable.full_name` and `Message.full_name`.
- `allow_type_mismatch` – If `false`, the types of the variable and the plug are checked and the callback is called only if compatible. If `true`, the callback is also called when types do not match.

You can make an API call to check that one logics-side entry and one graphics-side entry can be connected using the following function:

```
class Connectability(IntEnum):
    OK = 0
    INCOMPATIBLE = 1 (output with output, etc…)
    TYPE_MISMATCH = 2 (compatible, but types are not matching)


class MappingOperator:
    [...]
    def get_connectability(self, logical_io : MappingVariable, graphical_io
: SdyEntity) -> 'Connectability'
```

Parameters:

- `logical_io : MappingVariable` – SCADE Suite variable to be considered
- `graphical_io : SdyEntity` – SCADE Display / A661 entity to be considered

Return value: `Connectability` status

Save Changes

When using the Graphical Panel Coupling API, connection data may be modified. Such connection data is saved in a dedicated file named `<SuiteProjectName>.sdy`. To update the file with the modified list of connections, use the following algorithm:

```
for app in scade.model.suite.displaycoupling.get_roots():
    file = app.mapping_file
    file.save(file.pathname)
```

This algorithm should be used before exiting the script.

Example

A script example illustrates how API functions can be used to retrieve and access connection information between SCADE Suite models and SCADE Display or SCADE UA Page Creator graphical panels. The API example is in SCADE installation at SCADE\APIs\Python\examples\displaycoupling.

5 / Specific Commands for Python Scripting

This chapter describes specific Python commands available for writing Python scripts for SCADE Suite, SCADE Test, or SCADE Architect user environments.

- "API-Related Python Commands"
- "General Feedback-Related Python Commands"
- "GUI Display-Related Python Commands"
- "Customization-Related Python Commands"
- "Access to Predefined Operators in Python"

API-Related Python Commands

SCADE Suite, SCADE Test, and SCADE Architect provide regular commands to access model elements in read or write mode.

This section describes the commands available for each class, attributes, or association roles. The classes, attributes, and roles are presented in the corresponding meta model UML diagrams.

For each UML class, corresponds a Python class, and for each UML attributes or association roles corresponds a Python attribute (via getter/setter properties). The class operations are available changing the words uppercases by lower case separated by underscore.

Available commands are listed with a brief description of their function.

Table 5.1: List of API-related Python commands

Commands	Definition
"get_roots"	As the entry point of the API, gives access to the loaded model
"get"	Returns from a source object the value of the specified attribute or role
"set"	Sets to an object the value of the specified attribute or role
"call"	Calls the method of a source object and returns the result

GET_ROOTS

Command syntax and description:

```
scade.model.project.stdproject.get_roots()
```

Returns the list of all API entry points for the loaded project. If two projects are loaded, the command returns a list with two objects corresponding to the loaded projects.

GET

Command syntax and description:

```
_scade_api.get(<sourceObject>, <name>)
```

Returns the value of the attribute or role whose `<name>` is given, from `<sourceObject>`. In case the name is the one of an association whose multiplicity is greater than 1, the command returns the list of objects associated with `<sourceObject>` through this association.

SET

Command syntax and description:

```
_scade_api.set(<object>, <name>, <value>)
```

Sets the value of the attribute or role whose `<name>` is given, to the object. In case the name is the one of an association whose multiplicity is greater than 1, the value is a list of objects associated with the object through this association.

CALL

Command syntax and description:

```
_scade_api.call(<sourceObject>, <methodName> [,<arg>]*)<object>
```

Calls the `<methodName>` of `<sourceObject>` with `arg` and returns the result.

General Feedback-Related Python Commands

SCADE Suite, SCADE Test, or SCADE Architect provide the following general commands for producing feedback information:

Table 5.2: List of general Python commands

Commands	Definition
"errput"	Displays the specified error message whether in GUI or batch mode
"output"	Displays the specified text whether in GUI or batch mode

ERRPUT

Command syntax and description:

```
scade.errput (<text>)
```

Displays an error message specified by `<text>` into the **Script** tab of the Output Window. In batch mode, the message is written on the standard error output.

OUTPUT

Command syntax and description:

```
scade.output (<text>)
```

Displays the message specified by `<text>` into the **Script** tab of the Output Window. In batch mode, the message is written on the standard error output.

GUI Display-Related Python Commands

These commands are dedicated to handle and display API information in the user interfaces of SCADE Suite, SCADE Architect, or SCADE Test environments. Available commands are listed with a brief description of their function.

Table 5.3: List of GUI-related Python commands

Commands	Definition
"activate"	Simulates a double click on specified element
"activate_browser"	Displays the browser display with the specified tab
"activate_project"	Sets a project as the active one
"activate_tab"	Activates the specified tab as main output tab
"browser_report"	Adds a new `child_object` in the **View** tree
"clear_tab"	Clears the content of the specified tab
"create_browser"	Creates a **View** tab
"create_report"	Creates a new report tab in output window
"get_active_configuration"	Returns the active configuration
"get_active_project"	Returns the active project
"locate"	Generic command for locating an object described by string
"locate"	Locates all specified objects (SCADE Suite only)
"open_document_view"	Opens the specified document
"open_html_in_browser"	Opens a file in Windows default web browser
"open_html_view"	Opens a web browser to display a file
"open_source_code_view"	Opens specified file and displays cursor on set line/column
"output_log"	Logs output displayed on some output tab in specified file
"print and printer_setup"	Prints specified object in EMF, text, PostScript, or PNG file
"print_ssl"	Prints the custom symbol representation (SCADE Suite only)

Table 5.3: List of GUI-related Python commands (Continued)

Commands	Definition
"register_decoration"	Associates a named decoration with two images
"report"	Adds new lines and content in the report tab
"selection"	Variable with list of selected objects
"set_decoration"	Associates a SCADE object with a registered decoration
"set_output_tab"	Activates the specified tab to display outputs in
"set_style"	Sets the style of a presentation element object (SCADE Suite only)
"tabput"	Outputs text in specified tab and sets the tab as output tab
"unset_decoration"	Removes a decoration from a SCADE object
"version"	Returns SCADE version information

ACTIVATE

Command syntax and description:

```
scade.activate(<object>)
```

Simulates a double click on the `<object>`.

ACTIVATE_BROWSER

Command syntax and description:

```
scade.activate_browser(<tab_name>)
```

Activates a browser display with the tab called `<tab_name>`.

ACTIVATE_PROJECT

Command syntax and description:

```
scade.activate_project(<project_name>)
```

Sets the project named `<project_name>` as the active project.

ACTIVATE_TAB

Command syntax and description:

```
scade.activate_tab([tab])
```

Activates the `<tab>` tab as the main output tab.

BROWSER_REPORT

Command syntax and description:

```
scade.browser_report(child_object: Union['Object', str], parent_object:
Union['Object', str, None] = None, expanded: bool = False, user_data: Any = None,
name: str = '', icon_file: str = '') -> None
```

Adds a new `child_object` in the trees of design views.

- `parent_object`: is the parent of the node to be created
- `expanded`: makes the node expanded, otherwise it is collapsed
- `user_data`: associates user-data with the node, user-data value is provided to the callback called on activate events
- `name`: overrides the default name of the reported object
- `icon_file`: overrides the default image of the reported object

CLEAR_TAB

Command syntax and description:

```
scade.clear_tab([tab])
```

Clears the content of the `<tab>` tab.
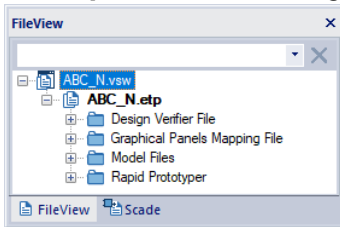
CREATE_BROWSER

Command syntax and description:

```
scade.create_browser(tab_name: str, icon_file: str = None, keep: bool = False,
callback: Callable[['Object', Any], None] = None) -> None
```

Activates a tree tab inside the browser view (tab is created if needed).

- `tab_name`: name of the tab to be created.
- `icon_file`: used as tab image.
- `keep`: keeps the content of the tab (if any), otherwise content is reset.
- `callback`: provides Python `callable` to be executed when a tree object is activated
  (by double-click or enter); will be called with the object and `user_data` (see
  "browser report") as parameters.

Example: Considering the following project to insert new view


```
import scade, scade.model.project
scade.create_browser('Example', 'D:\Tests\Images\EmbedSoft.ico')
def for_each_file_ref(file_ref, parent):
    scade.browser_report(file_ref, parent, True)
for p in scade.model.project.get_roots():
    scade.browser_report(p, None, True)
    for fr in p.file_refs:
        for_each_file_ref(fr, p)
```

Executing the above script adds the following new view:
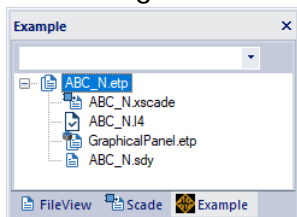
CREATE_REPORT

Command syntax and description:

```
scade.create_report(tab_name: str, header: Tuple[str, int, int]+, check: bool =
False) -> None
```

Creates a new report tab in the output window.

- `tab_name`: the name of the tab to be created
- `check`: option to display a check box before each line
- `header`: tuple(s) (`column_label`, `column_width`, `column_alignment`) – for each tuple, a new column is created with given label, width, alignment (0 for left align, 1 for right align)

For sorting, right-aligned columns are handled as number column, left-aligned as text. This improves navigation: `F4`-key allows to navigate to the following item of the tab and locate it in the model together.

Example: Consider the following model
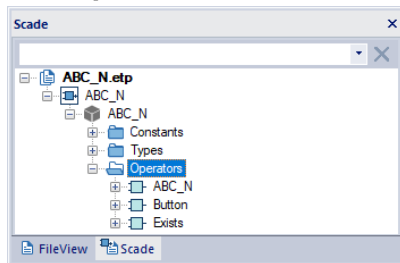

```
import scade, scade.model.suite
scade.create_report('Example', ('Operator', 300, 0), ('Type', 300, 0))
def for_each_all_operator(operator):
    scade.report(operator, 'Node' if operator.state else 'Function')
def for_the_model(model):
    for o in model.all_operators:
        for_each_all_operator(o)
for s in scade.model.suite.get_roots():
    for_the_model(s.model)
```

Executing the above script returns the following display:
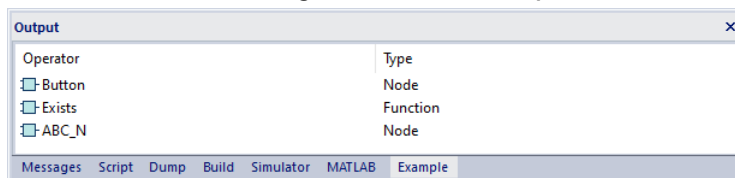
GET_ACTIVE_CONFIGURATION

Command syntax and description:

```
scade.get_active_configuration (project: scade.model.project.Project, tool_name:
string) -> Optional[Configuration]
```

Returns the activate configuration for the specified project and a tool.

GET_ACTIVE_PROJECT

Command syntax and description:

```
scade.get_active_project()
```

Returns the active project.

LOCATE

Command syntax and description:

```
scade.locate(locate_string: str) -> None
```

Locates the object described by `locate_string`.

LOCATE

Command syntax and description:

```
scade.model.suite.locate([(<object1> [color1, tooltip1, pin]), {(<object2>
[color2, tooltip2, pin2]), …])
```

Locates all specified objects. Only available for SCADE Suite.

The syntax of the command is a list of tuples, each tuple indicating a Scade object and optionally a color, a tooltip, and a pin. The color is represented by a string containing 3 pairs of hexadecimals, each pair representing the R, G, and B component of a Windows color. If not specified, a default color is used. The tooltip type is a string. The pin type is an integer indicating a pin number. See example below.

Example:

```
import scade, scade.model.suite
def outputln(text):
    scade.output(text + '\n')
mylist = []
number = 0
for item in scade.selection:
    if isinstance(item, scade.model.suite.Operator):
        number += 1
        mytuple = (item, "00FF00", "info op " + str(number))
        mylist.append(mytuple)
        for eq in item.equations:
            number += 1
            mytuple = (eq, "0000FF")
            mylist.append(mytuple)
scade.model.suite.locate(mylist)
```

OPEN_DOCUMENT_VIEW

Command syntax and description:

```
scade.open_document_view(<file_name>)
```

Opens the `<file_name>` document in an embedded view. If the file format is not supported, the document opens in ASCII mode.

OPEN_HTML_VIEW

Command syntax and description:

```
scade.open_html_view(file: Union[str, List[str]]], use: Union [str, None] = None,
delete: bool= False) -> None
```

Opens an embedded web browser to display specified file.

- `file`: either a local or remote file or list of files
- `use=<ident>`: allows to reuse the same browser from different calls. If the option is not set, each call to `report` opens a new browser.
- `delete`: if set, the files are deleted once no more used.

Example:

```
import scade
with open('c:/temp/example.html', 'w+') as f:
    f.write('<html><head><title>Example</title></head><body>Hello world !!!</body></html>\n')
scade.open_html_view('c:/temp/example.html', delete=True)
```

Executing the above script returns the following display:
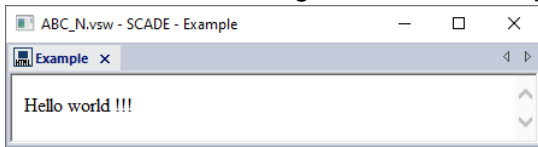
OPEN_HTML_IN_BROWSER

Command syntax and description:

```
scade.open_html_in_browser(file_name: str) -> None
```

Opens the specified `file_name>` in Windows default web browser.

OPEN_SOURCE_CODE_VIEW

Command syntax and description:

```
scade.open_source_code_view(file_name[, line[, column]])
```

Opens the `file_name` in an embedded view and sets the cursor at `line`, `column`.

OUTPUT_LOG

Command syntax and description:

```
scade.output_log(tab_name: str, command: str, path_name: str, separator: str) -> None
scade.output_log(tab_name: str, command: str) -> None
```

Logs the output produced on an output tab identified as `<tab_identifier>` in the file defined by `<path_name>`.

Example: Log all output from Simulator tab into the c:/temp/dummy.txt file

```python
import scade
scade.output_log('SCADE SIMULATOR', 'on', 'c:/temp/dummy.txt', ' ')
scade.tabput('SCADE SIMULATOR', 'hello world\n')
scade.output_log('SCADE SIMULATOR', 'off')
```

PRINT AND PRINTER_SETUP

Command syntax and description:

```
scade.print(source_object: 'Object', path_name: str, format: str ='[emf]|text|png|ps', rotation: int = 0)
-> None
scade.printer_setup(printer_name: str) -> None
```

Prints the `source_object` at `path_name` in emf, text, png, or ps (PostScript) file format. The `printer_setup` command enables the printer specified by `printer_name`. Note that `source_object` must support the specified format. Only available for SCADE Suite and SCADE Architect.

PRINT_SSL

Command syntax and description:

```
scade.model.suite.print_ssl(scade_operator, path_name, format='emf|png|ps', rotation=0)
```

Prints the custom symbol representation, if any, of the `scade_operator` in the pathname with emf, png ,or PostScript file format. Note that if `scade_operator` does not have any custom representation (.ssl file), the command does nothing. Only available for SCADE Suite.

REGISTER_DECORATION

Command syntax and description:

```
scade.model.suite.register_decoration(decoration_name: str, small_image_for_tree: str,
image_for_diagrams: str)
```

Associates the named decoration with two images:

- A first image used as overlay on icons
- A second image used on graphical diagrams

Additional calls with same `decoration_name` are ignored.

Due to technical limitations, the maximum number of different overlay icons in a session is limited. If the limit is reached, the additional overlay images are ignored.

Both images must be 16x16. Supported image format is 32-bit transparent PNG. Image paths must be specified as absolute paths.

REPORT

Command syntax and description:

```
scade.report(object[,string]*)
```

Adds new lines in the report tab. Sets the `object` in the first column (icon and text) and adds label strings in the remaining column(s). See example in <u>"create_report"</u> on page 101.

SELECTION

Command syntax and description:

```
scade.selection
```

Variable containing GUI selected object. Not available in batch mode.

SET_DECORATION

Command syntax and description:

```
scade.model.suite.set_decoration(scadeObject, decoration_name: str)
```

Associates a Scade object with a decoration, displaying the related icon on its representation in SCADE views and diagrams. If another decoration was already associated to this object, it is replaced.

The Scade object must be of the following types: Package, NamedType, Constant, Sensor, Operator, Input, Output, Local Variable, Signal, Equation, StateMachine, State, Transition, IfBlock, WhenBlock, Action, Graphical Diagrams, or Textual Diagrams.

See: <u>"register_decoration"</u> and <u>"unset_decoration"</u>

SET_OUTPUT_TAB

Command syntax and description:

```
scade.set_output_tab([<tab>])
```

Activates the `<tab>` so that outputs are displayed in this tab.

SET_STYLE

Command syntax and description:

```
scade.model.suite.set_style(presentation_element, style_name)
```

Sets the style whose `style_name` is given as parameter to the `presentation_element` object. `style_name` is case-sensitive. The function only applies to presentation elements (*i.e.,* objects) that are displayed in graphical diagrams. Only available for SCADE Suite.

TABPUT

Command syntax and description:

```
scade.tabput(tab, text)
```

Outputs the text in the tab and sets the tab as main output tab. The command works regardless of the type of tab (textual or report).

UNSET_DECORATION

Command syntax and description:

```
scade.model.suite.unset_decoration(scadeObject)
```

Removes decoration from a Scade object.

The Scade object must be of the following types: Package, NamedType, Constant, Sensor, Operator, Input, Output, Local Variable, Signal, Equation, StateMachine, State, Transition, IfBlock, WhenBlock, Action, Graphical Diagrams, or Textual Diagrams.

See: "register_decoration" and "set_decoration"

VERSION

Command syntax and description:

```
scade.version (['number'|'folderName'|'buildNumber'|'endYear'|'versionName'|'copyright']: str) -> str
```

Returns information about the current SCADE version with respect to used parameter.

Customization-Related Python Commands

These commands enable users to customize SCADE Suite, SCADE Architect, or SCADE Test user interface with additional commands, dialogs, properties/settings pages, wizards, or controls.

All these commands can only be used in scripts registered from SCADE Suite, SCADE Architect, or SCADE Test user interfaces. For registration see "Registering Customization Scripts" on page 77 in *SCADE Suite Technical Manual*.

The following describes the classes and functions available from the `scade.tool.suite.gui` Python package.

- "Custom Commands"
- "Custom Dialogs"
- "Custom Properties and Settings Pages"
- "Custom Wizards"
- "Custom Controls"
- "Callback Registration"

Custom Commands

Table 5.4: List of Python commands for command customization

Commands	Definition
"Command"	Registering a command
"Menu"	Adding a menu
"ContextMenu"	Adding a contextual menu
"Toolbar"	Adding a toolbar

An example of custom script using these commands is provided in SCADE installation. For details, see "Custom Script Example" on page 79 in *SCADE Suite Technical Manual*.

COMMAND

Class description:

`scade.tool.suite.gui.commands.Command`

Use this class to register a command.

Constructor:	

Command(name, on_activate = None, status_message = '', tooltip_message = '', accelerator = '', image_file = '', on_enable = None, on_check = None)	
`name`	Text to display when the command is added in a menu
`on_activate`	Callable to invoke upon command activation (via menu or toolbar)
`on_enable`	Callable to invoke when menu item or toolbar button is about to be displayed. Returns `True` or `False` to make the command enabled or disabled. If not provided, the command is always enabled.
`on_check`	Callable to invoke when menu item or toolbar button is about to be displayed. Returns `True` or `False` to make the command checked or unchecked. If not provided, the command is always unchecked.
`status_message`	Text to display in the status bar as command description
`tooltip_message`	Text to display in a tooltip
`accelerator`	Associates an accelerator with the command. The accelerator is a string containing the key name separated by '+' when more than one keys are used - for example "Ctrl+Shift+A".
`image_file`	Associates an image with the command. The image of the command is displayed in the toolbar or in the menu item. The file path must be absolute (see example below). This image file must be a Bitmap image (.bmp).

Callables (command handlers) can be provided or the class may be sub-classed and each specific (`on_activate`, `on_enable`, `on_check`) method overridden.

Class members:	
`id`	Unique ID of the command
`SEPARATOR`	A static member to be used as separator in menus and toolbars

```
from scade.tool.suite.gui.commands import Command
from scade import output
def on_activate(command):
    output('on_activate(' + str(command.id) + ')\n')
cmd = Command('A Command', on_activate)
```

Example 2: building absolute path in Python when image file path relative to the script

```
import os
from scade.tool.suite.gui.commands import Command
from scade import output
def on_activate(command):
    output('Command activated ! \n')
script_folder = os.path.dirname(__file__)
cmd = Command('A Command', on_activate, image_file = os.path.join(script_folder,
'MyImage.bmp')
```

MENU

Class description:

`scade.tool.suite.gui.commands.Menu`

Use this class to add a command menu.

Constructor:

Menu(commands, menu_path = '', position = 'last')

commands	List of commands to be added in a menu
menu_path	'/' separated list describing the hierarchical path to the new menu. If the whole path does not exist yet, missing popup menus are created. A first level menu has only a menu text as path. The character '&' can be used before the letter to underline in the menu (displayable by Alt-key usage).
position	Enables insertion at specific menu position: • menu-text: new item is inserted after the menu whose text is "menu-text" • first: new item is inserted as the first child of its parent • last: new item is inserted as the last child of its parent By default, a menu without position data is created as the last child of its parent.

```
from scade.tool.suite.gui.commands import Command, Menu
from scade import output
def on_activate(command):
    output('on_activate(' + str(command.id) + ')\n')
cmd = Command('A Command', on_activate, accelerator = 'Alt+C')
Menu(cmd, '&Tools/Test')
```

CONTEXTMENU

Class description:

`scade.tool.suite.gui.commands.ContextMenu`

Use this class to register a context menu.

Constructor:

`ContextMenu``(commands, on_enable = None, menu_path = '')`

`commands`	List of commands to be added in a menu
`on_enable`	Callable to invoke when a context menu is about to be displayed. Expected signature is `on_enable(context)`, where `context` parameter is a string giving the `ident` of the area like 'LOG', 'Scade', 'FILEVIEW', or 'SCADE NETVIEW', from which the contextual menu is invoked. Always enabled by default. As there may be several contexts, including tabs created by the user, all possible values for context cannot be listed here. A simple way is to print-out the context parameter value using the output API function.
`menu_path`	List separated by `/` that describes the hierarchical path to the new menu. Popup menus are created for the whole path each time.

Callables (menu handler) can be provided or the class may be sub-classed and the specific (`on_enable`) method overridden.

Example:

```
from scade.tool.suite.gui.commands import Command, ContextMenu
from scade import output
def on_activate(command):
    output('on_activate(' + str(command.id) + ')\n')
cmd = Command('A Command', on_activate)
ContextMenu(cmd, lambda context: context == 'LOG') # with lambda as callback
```

TOOLBAR

Class description:

`scade.tool.suite.gui.commands.Toolbar`

Use this class to create a toolbar.

Constructor:

`Toolbar``(name, commands)`

name	Name of the toolbar
commands	List of commands to be added

Example:

```
from scade.tool.suite.gui.commands import Command, Toolbar
from scade import output
cmds = [
    Command('Cmd 1', image_file = '2.bmp', on_activate = lambda cmd: output('on_1(' +
str(cmd.id) + ')\n')),
    Command.SEPARATOR,
    Command('Cmd 2', image_file = '2.bmp', on_activate = lambda cmd: output('on_2(' +
str(cmd.id) + ')\n'))]
Toolbar('A Toolbar', cmds)
```

Custom Dialogs

Table 5.5: List of Python commands for dialog customization

Commands	Definition
"Dialog"	Creating a modal or non-modal dialog box
"Message Box"	Creating a message box
"File Open"	Opening a file open dialog box
"File Save"	Opening a file save dialog box
"Browse Directory"	Opening a directory selection dialog box

DIALOG

Class description:

`scade.tool.suite.gui.dialogs.Dialog`

Use this class to create modal or non-modal dialog boxes.

Constructor:

`Dialog``(name, width, height, on_build = None, on_display = None, on_close = None)`

`name`	Title of the dialog box
`width/height`	Initial width and height of the dialog box
`on_build`	Callable invoked to build the dialog box and the controls of the window. The dialog box is built each time it is displayed.
`on_display`	Callable invoked when the dialog box is about to be displayed. It is used to initialize the data of the dialog box.
`on_close`	Callable invoked when the dialog window is about to be closed.

Callables (dialog handlers) can be provided or the class may be sub-classed and each specific (`on_build`, `on_display`, `on_close`) method overridden.

Class members:

`do_modal()`	Displays a modal dialog box
`create()`	Displays a modeless dialog box
`close()`	Closes the dialog box

Example:

```
from scade.tool.suite.gui.commands import Command, Menu
from scade.tool.suite.gui.dialogs import *
from scade import output
class MyDialog(Dialog):
    def __init__(self, name):
        super().__init__(name, 320, 240)
    def on_build(self):
        output('my_dialog.on_build(' + str(self.id) + ')\n')
    def on_display(self):
        output('my_dialog.on_display(' + str(self.id) + ')\n')
    def on_close(self):
        output('my_dialog.on_close(' + str(self.id) + ')\n')
Menu(menu_path = 'Dialogs', commands = [Command('Modal Dialog', lambda cmd:
        MyDialog('Test Dialog').do_modal() ) ])
```

MESSAGE BOX

Function description:

`scade.tool.suite.gui.dialogs.message_box`

Use this function to open a dialog box containing a message.

Constructor:

`message_box``(name, message, style = 'ok', icon = 'information')`

`name`	Caption of the message box
`message`	Text of the message box
`style`	Sets the style of the box: ok | okcancel | retrycancel | yesno | yesnocancel
`icon`	Sets the icon displayed in the box: stop | question | warning | information

The function ends when the user closes the dialog box. The return value depends of the user answer with the following values: OK: 1 - Cancel: 2 - Retry: 4 - Yes: 6 - No: 7

Example:

```
from scade.tool.suite.gui.commands import Command, Menu
from scade.tool.suite.gui.dialogs import *
Menu(menu_path = 'Dialogs', commands = [
    Command('Message Box', lambda cmd:
        message_box('Question?', 'The answer is 42 of course.', icon = 'question') ) ])
```

FILE OPEN

Function description:

`scade.tool.suite.gui.dialogs.file_open`

Use this function to open a standard Open dialog box.

Constructor:

`file_open``(filter = '', directory = '')`

`filter`	A string describing the format of the visible files in the dialog box
`directory`	Initial directory of the open file dialog. The default value is the current directory.

The return value is the string of the chosen file path, or `' '` if the user cancels.

Example:

```
from scade.tool.suite.gui.commands import Command, Menu
from scade.tool.suite.gui.dialogs import *
Menu(menu_path = 'Dialogs', commands = [
    Command('File Open', lambda cmd:
        file_open('Custom modules (*.py)|*.py|All Files (*.*)|*.*||') ) ])
```

FILE SAVE

Function description:

`scade.tool.suite.gui.dialogs.file_save`

Use this function to open a standard Save dialog box.

Constructor:

`file_save``(file_name, extension = '', directory = '', filter = '')`

`file_name`	Name of the file to save
`extension`	Default extension of the file to save
`directory`	Initial directory of the save file dialog. The default value is the current directory.
`filter`	A string describing the format of the visible files in the dialog box

The return value is the string of the chosen file path, or `' '` if the user cancels.

Example:

```
from scade.tool.suite.gui.commands import Command, Menu
from scade.tool.suite.gui.dialogs import *
Menu(menu_path = 'Dialogs', commands = [
    Command('File Save', lambda cmd:
        file_save('test.py', filter = 'Custom modules (*.py)|*.py|All Files (*.*)|*.*||') )
])
```

BROWSE DIRECTORY

Function description:

`scade.tool.suite.gui.dialogs.browse_directory`

Use this function to open a standard directory selection dialog.

Constructor:

`browse_directory``(initial_directory = '')`

`initial_directory`	Initial directory of the open file dialog. The default value is the current directory.

The return value is the string of the chosen path, or `''` if the user cancels.

Example:

```
from scade.tool.suite.gui.commands import Command, Menu
from scade.tool.suite.gui.dialogs import *
Menu(menu_path = 'Dialogs', commands = [
    Command('Browse Directory', lambda cmd:
        browse_directory() ) ])
```

Custom Properties and Settings Pages

Table 5.6: List of Python commands for page customization

Commands	Definition
"Properties Page"	Creating a property page
"Settings Page"	Creating a settings page

PROPERTIES PAGE

Class description:

`scade.tool.suite.gui.properties.Page`

Use this class to create a property page.

Constructor:

`Page`(name, width_min = 0, height_min = 0, is_available = None, on_context = None, on_build = None, on_layout = None, on_display = None, on_validate = None, on_close = None, help_id = 0)	
`name`	Title of the page
`width_min/` `height_min`	Initial width and height of the dialog box
`is_available`	Callable invoked to know if the property page must be displayed or not. The procedure must return `True` to make the page available or `False` otherwise. The callable is expected to take a list of models, that can be used for availability evaluation.
`on_context`	Callable invoked with the list of models for which the property page is evaluated as available.
`on_build`	Callable invoked when the property page is opened for the first time before the actual creation of the page.
`on_layout`	Callable invoked when the property page is opened for the first time after the actual creation of the page. This is the typical place to specify how controls are moved/resized when the property page size is changed using the `Widget.set_constraint(...)` function.
`on_display`	Callable invoked when the dialog box is about to be displayed. It is used to initialize the data of the dialog box.

on_validate	Callable invoked each time a new Property Page is no more selected in the Property Sheet, and before closing the Property Sheet. The callable must analyze the fields filled by the user and update the corresponding data.
on_close	Callable invoked when the dialog window is about to be closed.
help_id	Identifier used to map the contextual help for this property page.

Callables (page handlers) can be provided or the class may be sub-classed and each specific (is_available, on_context, on_build, on_layout, on_display, on_validate, on_close) method overridden.

Example:

```python
from scade.tool.suite.gui.properties import Page
from scade.tool.suite.gui.widgets import Widget, GroupBox, EditBox
from scade import output
class Test(Page):
    def __init__(self, name):
        super().__init__(name)
    def is_available(self, models):
        output('test_properties.is_available(' + str(models) + ')\n')
        return True
    def on_context(self, models):
        output('test_properties.on_context(' + str(models) + ')\n')
    def on_build(self):
        self.group = GroupBox(self, 'Output')
        self.edit = EditBox(self, style = ['multiline', 'vscroll'])
        output('test_properties.on_build(' + str(self.id) + ')\n')
    def on_layout(self):
        output('test_properties.on_layout(' + str(self.id) + ')\n')
        self.group.set_constraint(Widget.LEFT, self, Widget.LEFT, 10),
        self.group.set_constraint(Widget.TOP, self, Widget.TOP, 10),
        self.group.set_constraint(Widget.RIGHT, self, Widget.RIGHT, -10),
        self.group.set_constraint(Widget.BOTTOM, self, Widget.BOTTOM, -10)
        self.edit.set_constraint(Widget.LEFT, self.group, Widget.LEFT, 10)
        self.edit.set_constraint(Widget.TOP, self.group, Widget.TOP, 20)
        self.edit.set_constraint(Widget.RIGHT, self.group, Widget.RIGHT, -10)
        self.edit.set_constraint(Widget.BOTTOM, self.group, Widget.BOTTOM, -10)
    def on_display(self):
        output('test_properties.on_display(' + str(self.id) + ')\n')
    def on_validate(self):
        output('test_properties.on_validate(' + str(self.id) + ')\n')
    def on_close(self):
        output('test_properties.on_close(' + str(self.id) + ')\n')
Test('Properties')
```

SETTINGS PAGE

Class description:

`scade.tool.suite.gui.settings.Page`

Use this class to create a settings page.

Constructor:

`Page`(name, on_build = None, on_display = None, on_validate = None, on_close = None, help_id = 0)	
`name`	Title of the page
`on_build`	Callable invoked when the settings page is opened for the first time before the actual creation of the page.
`on_display`	Callable invoked when the page is about to be displayed. It is used to initialize the data of the page.
`on_validate`	Callable invoked each time a new page is no more selected in the Settings dialog, and before closing the Settings dialog. The callable must analyze the fields filled by the user and update the corresponding data.
`on_close`	Callable invoked when the dialog window is about to be closed.
`help_id`	Identifier used to map the contextual help for this page.

Callables (page handlers) can be provided or the class may be sub-classed and each specific (`on_build`, `on_display`, `on_validate`, `on_close`) method overridden.

```
from scade.tool.suite.gui.settings import Page
from scade.tool.suite.gui.widgets import Label
import scade.model.project
from scade import output
class Test(Page):
    def __init__(self, name):
        super().__init__(name)
    def on_build(self):
        self.label = Label(self, 'Test Settings Page', 5, 5, 400, 40)
        output('on_build(' + str(self.id) + ')\n')
    def on_display(self, project, configuration):
      output('on_display(' + str(self.id) + ', ' + str(project) + ', ' + str(configuration)
+ ')\n')
    def on_validate(self, project, configuration):
      output('on_validate(' + str(self.id) + ', ' + str(project) + ', ' + str(configuration)
+ ')\n')
    def on_close(self):
        output('on_close(' + str(self.id) + ')\n')
Test('Test Settings Page')
```

Custom Wizards

The project and file creation wizards available in SCADE Suite can be customized with additional wizards for project or file creation.

Table 5.7: List of Python commands for wizard customization

Commands	Definition
"File or Project Wizards"	Creating a custom wizard (for project or file creation)
"Wizards Page"	Creating a wizard page

FILE OR PROJECT WIZARDS

Class description:

`scade.tool.suite.gui.wizards`

Use this class to create a custom wizard. Two types of wizards are available **FileWizard** and **ProjectWizard**, which use common interface inherited from `Wizard` class.

Constructor:

`FileWizard` (name, extension, icon = '', help_text = '', pages = [], on_start = None, on_finish = None)
`ProjectWizard` (name, project_kind, icon = '', help_text = '', pages = [], on_start = None, on_finish = None)

name	Title of the wizard dialog
extension	Extension of the file to be produced by the wizard
project_kind	Kind of the project to be produced by the wizard. This is the value set in the @STUDIO:PRODUCT project prop
icon	Icon displayed in the available wizard list in the tabs of the wizards dialog. If not provided, a default icon is associated with the wizard.
help_text	Help text of the wizard
pages	Ordered list of the wizard pages composing the wizard
on_start	Callable invoked after the user has acknowledged the wizard, before the display of the wizard
on_finish	Callable invoked after the user has finished the wizard, typically when clicking on "**Finish**" button. The callable is expected to take a Boolean parameter indicating whether the user finished or canceled the wizard.

Callables (wizard handlers) can be provided or the class may be sub-classed and each specific (`on_start`, `on_finish`) method overridden.

WIZARDS PAGE

Class description:

`scade.tool.suite.gui.wizards.Page`

Use this class to create custom wizard pages.

Constructor:

`Page`(name, image = '', on_build = None, on_display = None, on_cancel = None)	
`name`	Title of the page
`image`	Image associated with the page. If not provided, a default image is associated with the page.
`on_build`	Callable invoked when the page is opened for the first time, before the actual creation of the page.
`on_display`	Callable invoked when the page is about to be displayed. It is used to initialize the data of the page.
`on_cancel`	Callable invoked when the user clicks Cancel button and before the cancel action has taken place. Returns `False` to prevent the cancel operation or `True` to allow it.

Callables (page handlers) can be provided or the class may be sub-classed and each specific (`on_build`, `on_display`, `on_cancel`) method overridden.

Example:

```
from scade.tool.suite.gui.wizards import *
from scade.tool.suite.gui.widgets import GroupBox
from scade.tool.suite.gui.dialogs import message_box
from scade import output

class TestPage1(Page):
    def __init__(self):
        super().__init__('Test Page 1')
    def on_build(self):
        output('page_1.on_build(' + str(self.id) + ')\n')
        GroupBox(self, 'Test Page 1', 140, 5, 330, 268)
    def on_display(self):
        output('page_1.on_display(' + str(self.id) + ')\n')
    def on_cancel(self):
        output('page_1.on_cancel(' + str(self.id) + ')\n')
        return message_box('Wizard','Do you want to cancel?','yesno','question') == 6 # IDOK

class TestPage2(Page):
    def __init__(self):
        super().__init__('Test Page 2')
    def on_build(self):
        output('page_2.on_build(' + str(self.id) + ')\n')
        GroupBox(self, 'Test Page 2', 140, 5, 330, 268)
    def on_display(self):
        output('page_2.on_display(' + str(self.id) + ')\n')
    def on_cancel(self):
        output('page_2.on_cancel(' + str(self.id) + ')\n')
        return message_box('Wizard','Do you want to cancel?','yesno','question') == 6 # IDOK

class TestProjectWizard(ProjectWizard):
    def __init__(self):
        pages = [TestPage1(), TestPage2()]
        super().__init__('Test Project Wizard', 'py.ico', 'Test Project Wizard Help Text',
pages)
    def on_start(self, model):
        output('project_wizard.on_start(' + str(model) + ')\n')
    def on_finish(self, finish, model):
        output('project_wizard.on_finish(' + str(finish) + + str(model) ')\n')
TestProjectWizard()
```

Custom Controls

Module and description:

`scade.tool.suite.gui.widgets`

Use this module to create various controls to be used in dialogs and property pages. Base class for all controls is **`Widget`**.

Class members:	
`set_name(name)/get_name()`	Set/get name or text for various widgets
`set_box(x0, y0, x1, y1)/` `get_box()`	Set/get the bounding box of the widget. Object of class Box, which is returned by `get_box()`, is a simple wrapper of widget's coordinates and provides width/height properties in addition.
`set_visible(visible)/` `is_visible()`	Access widget's visibility
`set_enable(enable)/` `is_enabled()`	Access enabled/disabled state of the widget
`set_constraint(constraint , to, constraint_to, add = 0, mul = 1)`	Specify how to adjust the current widget's size/coordinate with respect to another widget's size/coordinate. This allows to control left, right, top, bottom, width, height, or centering position, specifically when the parent window size is changed. • `constraint`: size/coordinate of current widget to adjust. Possible values are: `Widget.LEFT`, `Widget.RIGHT`, `Widget.TOP`, `Widget.BOTTOM`, `Widget.WIDTH`, `Widget.HEIGHT`, `Widget.HCENTER`, `Widget.VCENTER`. • `to`: widget variable or property page variable as reference for moving/ resizing • `constraint_to`: size/coordinate of the `to` widget variable used as reference. Possible values are: `Widget.LEFT`, `Widget.RIGHT`, `Widget.TOP`, `Widget.BOTTOM`, `Widget.WIDTH`, `Widget.HEIGHT`, `Widget.HCENTER`, `Widget.VCENTER`. • `add`: specifies a value to be added to `constraint_to` • `mul`: specifies a value to be multiplied to `constraint_to` All usages of the above parameters are illustrated in examples below.

Examples

```
#set left position of widget A equal to left position of widget B
A.set_constraint(Widget.LEFT, B, Widget.LEFT)

#set left position of widget A equal to right position of widget B
A.set_constraint(Widget.LEFT, B, Widget.RIGHT)

#Set left position of A equal to left position of B plus 10
A.set_constraint(Widget.LEFT, B, Widget.LEFT, add = 10)

#Set top position of A equal to bottom position of B minus 20
A.set_constraint(Widget.TOP, B, Widget.BOTTOM, add = -20)

#Set width of A equal to 0.5 times B width
A.set_constraint(Widget.WIDTH, B, Widget.WIDTH, mul = 0.5)

#Set height of A equal to 1.2 times B width
A.set_constraint(Widget.HEIGHT, B, Widget.WIDTH, 1.2)

#A(width) = (0.5 * B(width)) + 10
A.set_constraint(Widget.WIDTH, B, Widget.WIDTH, add = 10, mul = 0.5)

#A(left) = (0.1 * B(width)) + (-20)
A.set_constraint(Widget.LEFT, B, Widget.WIDTH, -20, 0.1)
```

Static members:

`show(widgets, show)`	Set visibility state of a list of widgets
`enable(widgets, enable)`	Set enabled/disabled state of a list of widgets
`group(buttons)`	Group a list of radio buttons

Table 5.8: List of Python commands for controls customization

Commands	Definition
"Label"	Creating static text control
"Image"	Creating picture control
"GroupBox"	Creating named frame to group inside other controls
"Button"	Creating button control
"CheckBox"	Creating check box control
"RadioButton"	Creating radio button with text
"EditBox"	Creating edit control
"ComboBox"	Creating combo box control
"ObjectComboBox"	Creating combo box control containing list of objects
"ListBox"	Creating list box control
"ObjectListBox"	Creating list box control containing list of objects
"TreeListBox"	Creating tree list control
"ObjectTreeListBox"	Creating tree list control containing list of objects
"Example of Base Widgets" and *"Example of Object Widgets"*	

LABEL

Class description:

Use **Label** class to create a static text control without any user action.

Constructor:

```
Label(owner, name, x = 0, y = 0, w = 0, h = 0, style = [])
```

owner	Specifies the owner (dialog, property page) of the control
name	Specifies the text of the control
x, y, w, h	Specifies the control position and size
style	Specifies visibility and state: 'visible|hidden', 'enable|disable'

IMAGE

Class description:

Use **Image** class to create a picture control.

Constructor:	
Image(owner, file, x = 0, y = 0, w = 0, h = 0, style = [])	
owner	Specifies the owner (dialog, property page) of the control
file	Specifies the path of the picture to be displayed. The path can be either absolute or relative to the script file path. The file is in *.bmp format.
x, y, w, h	Specifies the control position and size
style	Specifies visibility and state: 'visible|hidden', 'enable|disable'

GROUPBOX

Class description:

Use **GroupBox** class to create a frame with a name that is used to group inside other controls.

Constructor:	
GroupBox(owner, name, x = 0, y = 0, w = 0, h = 0, style = [])	
owner	Specifies the owner (dialog, property page) of the control
name	Specifies the text of the control
x, y, w, h	Specifies the control position and size
style	Specifies visibility and state: 'visible|hidden', 'enable|disable'

BUTTON

Class description:

Use **Button** class to create a button control with text.

Constructor:	
Button(owner, name, x = 0, y = 0, w = 0, h = 0, on_click = None, check = False, style = [], help_id = 0)	
owner	Specifies the owner (dialog, property page) of the control
name	Specifies the text of the control
x, y, w, h	Specifies the control position and size
on_click	Callable to invoke when the check box is clicked
check	Initial 'check' state of the check box
style	Specifies visibility and state: 'visible|hidden', 'enable|disable'
help_id	Help ID of the control

Callable (button handler) can be provided or the class may be sub-classed and the specific (on_click) method overridden.

Class members:	
set_check(check)/get_check()	Access the 'check' state of the button

CHECKBOX

Class description:

Use **CheckBox** class to create a check-box control with text. Inherits from class **Button**.

Constructor:	
CheckBox(owner, name, x = 10, y = 10, w = 50, h = 10, on_click = None, check = False, style = [], help_id = 0)	
owner	Specifies the owner (dialog, property page) of the control
name	Specifies the text of the control
x, y, w, h	Specifies the control position and size

`on_click`	Callable to invoke when the check box is clicked
`check`	Initial 'check' state of the check box
`style`	Specifies visibility and state: 'visible|hidden', 'enable|disable'
`help_id`	Help ID of the control

Callable (checkBox handler) can be provided, or the class may be sub-classed and the specific (`on_click`) method overridden.

Class members:

`set_check(check )/get_check()`	Access the 'check' state of the check box

RADIOBUTTON

Class description:

Use **RadioButton** class to create a radio-button control with text. Inherits from class **Button**.

Constructor:

RadioButton(owner, name, x = 10, y = 10, w = 50, h = 10, on_click = None, check = False, style = [], help_id = 0)

`owner`	Specifies the owner (dialog, property page) of the control
`name`	Specifies the text of the control
`x, y, w, h`	Specifies the control position and size
`on_click`	Callable to invoke when the check box is clicked
`check`	Initial 'check' state of the check box
`style`	Specifies visibility and enable state: 'visible|hidden', 'enable|disable'
`help_id`	Help ID of the control

Callable (radioButton handler) can be provided, or the class may be sub-classed and the specific (`on_click`) method overridden.

Class members:

`set_check(check )/get_check()`	Access the 'check' state of the radio button

EDITBOX

Class description:

Use **EditBox** class to create an edit box control.

Constructor:	
EditBox(owner, x = 10, y = 10, w = 50, h = 14, name = '', style = [], help_id = 0)	
`owner`	Specifies the owner (dialog, property page) of the control
`x, y, w, h`	Specifies the control position and size
`name`	Specifies the text of the control
`style`	Specifies visibility, enable state, and edit-box styles: 'visible|hidden', 'enable|disable', 'multiline', 'hscroll', 'vscroll' 'hscroll' and 'vscroll' valid only with 'multiline'
`help_id`	Help ID of the control
Class members:	
`set_multiline(multiline)/ get_multiline()`	Access 'multiline' style of the edit box
`set_hscroll(hscroll)/ get_hscroll()`	Access 'hscroll' style of the edit box
`set_vscroll(vscroll)/ get_vscroll()`	Access 'vscroll' style of the edit box

COMBOBOX

Class description:

Use **ComboBox** class to create a combo box control.

Constructor:

```
ComboBox(owner, items, x = 0, y = 0, w = 0, h = 0, on_change_selection = None, on_change_edit
= None, selection = '', style = [], help_id = 0)
```

`owner`	Specifies the owner (dialog, property page) of the control
`items`	Specifies the items list of the combo box (list of strings)
`x, y, w, h`	Specifies the control position and size
`on_change_selection(index)`	Callable invoked when the selection is changed. The index parameter specifies the index of the selected item in the list.
`on_change_edit(text)`	Callable invoked when the text in the edit-box is changed. The text parameter specifies the text in the edit box.
`selection`	Specifies the initially selected item of the combo box (string)
`style`	Sets visibility, enable state, and combo box styles: 'visible|hidden', 'enable|disable', 'simple|dropdown|dropdownlist', 'sort' • `simple`: displays the list box at all times. The current selection in the list box is displayed in the edit control. • `dropdown`: similar to `simple`, except that list box is not displayed unless user selects an icon next to the edit control. • `dropdownlist`: similar to `dropdown`, except that edit control is replaced by a static text item displaying current selection in the list box.
`help_id`	Help ID of the control

Callables (combobox handlers) can be provided or the class may be sub-classed and each specific (`on_change_selection`, `on_change_edit`) method overridden.

Class members:

`set_items(items)/get_items()`	Access combo box's items
`set_selection(selection)/` `get_selection()`	Access combo box's selection
`set_edit(edit)/get_edit()`	Change/access the 'dropdown|dropdownlist' style
`set_sort(sort)/get_sort()`	Change/access the `'sort'` style

OBJECTCOMBOBOX

Class description:

Use **ObjectComboBox** class to create a combo-box control containing a list of objects. Inherits from class **ComboBox** and has no additional properties.

LISTBOX

Class description:

Use **ListBox** class to create a list box control.

Constructor:	
ListBox(owner, items, x = 0, y = 0, w = 0, h = 0, on_change_selection = None, selection = [], style = [], help_id = 0)	
owner	Specifies the owner (dialog, property page) of the control
items	Specifies the items list of the list box (list of strings)
x, y, w, h	Specifies the control position and size
on_change_selection(index)	Callable invoked when the selection is changed. The index parameter specifies the list of selected items indexes in the list.
selection	Specifies the initially selected item of the list box (string)
style	Sets visibility, enable state, and list box styles: 'visible|hidden', 'enable|disable', 'sort'
help_id	Help ID of the control

Callable (listbox handler) can be provided or the class may be sub-classed and the specific (on_change_selection) method overridden.

Class members:	
set_items(items)/ get_items()	Access list box's items
set_selection(selection)/ get_selection()	Access list box's selection
set_sort(sort)/get_sort()	Change/access the 'sort' style of the list box

OBJECTLISTBOX

Class description:

Use **ObjectListBox** class to create a list-box control containing a list of objects. Inherits from class **ListBox** and has no additional properties.

TREELISTBOX

Class description:

Use **TreeListBox** class to create a tree-list control.

Constructor:	
`TreeListBox(owner, items, x = 0, y = 0, w = 0, h = 0, on_change_selection = None, headers = [], column_style = [], selection = [], images = [], style = [], help_id = 0)`	
`owner`	Specifies the owner (dialog, property page) of the control
`items`	Specifies the item list of the tree-list box. Each item is a dictionary object with the following keys: • `text` – item's list of strings (for each column, see headers parameter below) • `image` – zero-based index of image in image list (see images parameter below) • `parent` – if not empty, the new item is inserted at the root level, else it is inserted under the parent item. The parameter item contains the first column parent item text. An item may be created using helper static method: `TreeListBox.Item(text, image =-1, parent = '')`
`x, y, w, h`	Specifies the control position and size
`on_change_selection (index)`	Callable invoked when the selection is changed. The index parameter specifies the list of selected items indices in the list.
`headers`	Specifies the header control of the tree-list control. Each header is a dictionary object with the following keys: • `name` – column's name • `width` – column's width, if -1, the column spans the width of the control • `align` – alignment of column's content – 0 (left), 1 (right), 2 (center) A header may be created using helper static method: `TreeListBox.Header(name, width = -1, align = 0)`

`column_style`	Sets the style of the column of the tree-list. It can be a combination of : • `'linesatroot'`: Uses lines to link items at the root of the tree view control • `'linesbetweenitems'` : Paints vertical lines between columns • `'linesbetweencolumns'`: Paints horizontal lines between columns
`selection`	Specifies the initially selected items of the tree list box (string)
`images`	Specifies the list of file paths to *.bmp containing each image of the tree image list. It can be either absolute or relative to the script file. The zero-based index of the list can be used for tree list box items.
`style`	Sets visibility, enable state, and tree-list box styles: 'visible|hidden', 'enable|disable', 'sort'
`help_id`	Help ID of the control

Callable (treelistbox handler) can be provided or the class may be sub-classed and the specific (`on_change_selection`) method overridden.

Class members:	
`insert_item(item)`	Inserts in the control an item as a dictionary object with the following keys: • `text` – item's text • `image` – item's image index (see images) • `parent` – item's parent An item may be created using helper static method: `TreeListBox.Item(text, image = -1, parent = '')`
`set_items(items)/` `get_items()`	Access tree list box items (see insert_item for item description)
`set_headers(headers)/` `get_headers()`	Access tree list box headers
`set_column_style(style)/` `get_column_style()`	Access tree list box column style
`set_selection(selection)` `/get_selection()`	Access tree list box selection
`set_images(sort)/` `get_images()`	Access tree list box image list
`set_sort(sort)/` `get_sort()`	Change/access the 'sort' style of the list box

OBJECTTREELISTBOX

Class syntax and description:

Use **ObjectTreeListBox** class to create a tree-list control containing a list of objects. Inherits from class **TreeListBox** and has no additional properties.

EXAMPLE OF BASE WIDGETS

```python
from scade import output
from scade.tool.suite.gui.commands import Command, Menu
from scade.tool.suite.gui.dialogs import Dialog
from scade.tool.suite.gui.widgets import *
from ctypes import windll
class WidgetsDialog(Dialog):
    def __init__(self, name):
        super().__init__(name, 640, 480)
    def on_build(self):
        # panel
        GroupBox(self, 'Panel', 5, 5, 615, 40)
        Widget.group([
            RadioButton(self, 'ComboBox', 15, 20, 75, 20, self.on_combo_box_click, True),
            RadioButton(self, 'ListBox', 95, 20, 75, 20, self.on_list_box_click),
            RadioButton(self, 'TreeListBox', 175, 20, 75, 20, self.on_tree_box_click)
        ])
        Image(self, 'Custom.Tests\Images\CmdE.bmp', 255, 20, 20, 20)
        CheckBox(self, 'Enable', 290, 20, 50, 20, self.on_enable_click, True)
        Button(self, 'Close', 570, 15, 45, 25, self.on_close_click)
        #output
        Label(self, 'Output', 5, 185, 60, 20, style = 'disable')
        self.output = EditBox(self, 5, 200, 615, 243, style = ['multiline', 'vscroll'])
        self.output_lines = 0
        # widget
        GroupBox(self, 'Widget', 5, 50, 615, 130)
        # combo
        self.combo = [
            ComboBox(self, ['one', 'two', 'three'], 15, 65, 128, 20,
self.on_combo1_change_selection, selection = 'one', style = ['sort', 'dropdownlist']),
            Button(self, '<>', 148, 65, 30, 20, self.on_combo_swap_click),
          ComboBox(self, ['1', '2', '3'], 183, 65, 128, 20, self.on_combo2_change_selection,
self.on_combo2_change_edit, selection = '1')
        ]
        # list
        self.list = [
            ListBox(self, ['one', 'two', 'three'], 15, 65, 128, 108,
self.on_list1_change_selection, style = ['sort', 'hidden']),
            Button(self, '<>', 148, 65, 30, 20, self.on_list_swap_click),
          ListBox(self, ['1', '2', '3'], 183, 65, 128, 108, self.on_list2_change_selection,
style = 'hidden'),
        ]
```

```
        # tree
        _headers = [ TreeListBox.Header('0', 60), TreeListBox.Header('1', 68) ]
        _items = [
            TreeListBox.Item(['1', '2']), TreeListBox.Item('3', parent = '1'),
            TreeListBox.Item(['4', '5']), TreeListBox.Item('6', parent = '4')
        ]
        self.tree = [
            TreeListBox(self, _items, 15, 65, 128, 108, self.on_tree1_change_selection,
_headers, column_style = ['linesatroot'], style = 'hidden'),
            Button(self, '<>', 148, 65, 30, 20, self.on_tree_swap_click),
            TreeListBox(self, [TreeListBox.Item('B'), TreeListBox.Item('A')], 183, 65, 128,
108, self.on_tree2_change_selection, TreeListBox.Header('a', 68), style = ['sort', 'hidden'])
        ]
    def log(self, message):
        self.output.set_name(self.output.get_name() + '\n' + message)
        self.output_lines = self.output_lines + message.count('\n') + 1
        windll.user32.SendMessageA(self.output.id, 182, 0, self.output_lines) #
EM_LINESCROLL(182)
    # panel
    def on_combo_box_click(self, combo):
        self.log('on_combo_box_click()')
        Widget.show(self.list, False)
        Widget.show(self.tree, False)
        Widget.show(self.combo, True)
    def on_list_box_click(self, list):
        self.log('on_list_box_click()')
        Widget.show(self.combo, False)
        Widget.show(self.tree, False)
        Widget.show(self.list, True)
    def on_tree_box_click(self, tree):
        self.log('on_tree_box_click()')
        Widget.show(self.combo, False)
        Widget.show(self.list, False)
        Widget.show(self.tree, True)
    def on_enable_click(self, button):
        enable = button.get_check()
        self.log('on_enable_click(' + str(enable) + ')')
        Widget.enable(self.list, enable)
        Widget.enable(self.tree, enable)
        Widget.enable(self.combo, enable)
    def on_close_click(self, button):
        self.log('on_close_click()')
        self.close()
```

```
    # combo
    def on_combo1_change_selection(self, combo, index):
        self.log('on_combo1_change_selection(' + str(index) + ') => ' + combo.get_selection())
    def on_combo_swap_click(self, button):
        self.log('on_combo_swap_click()')
        items = [self.combo[0].get_items(), self.combo[2].get_items()]
        selection = [self.combo[0].get_selection(), self.combo[2].get_selection()]
        self.combo[0].set_items(items[1])
        self.combo[2].set_items(items[0])
        self.combo[0].set_selection(selection[1])
        self.combo[2].set_selection(selection[0])
    def on_combo2_change_selection(self, combo, index):
        self.log('on_combo2_change_selection(' + str(index) + ') => ' + combo.get_selection())
    def on_combo2_change_edit(self, combo, text):
        self.log('on_combo2_change_edit(' + text + ')')
    #list
    def on_list1_change_selection(self, list, index):
        self.log('on_list1_change_selection(' + str(index) + ') => ' +
str(list.get_selection())))
    def on_list_swap_click(self, button):
        self.log('on_list_swap_click()')
        items = [self.list[0].get_items(), self.list[2].get_items()]
        selection = [self.list[0].get_selection(), self.list[2].get_selection()]
        self.list[0].set_items(items[1])
        self.list[2].set_items(items[0])
        self.list[0].set_selection(selection[1])
        self.list[2].set_selection(selection[0])
    def on_list2_change_selection(self, list, index):
        self.log('on_list2_change_selection(' + str(index) + ') => ' +
str(list.get_selection())))
    # tree
    def on_tree1_change_selection(self, tree):
        self.log('on_tree1_change_selection(' + str(tree.get_selection()) + ')')
    def on_tree_swap_click(self, button):
        self.log('on_tree_swap_click()')
        headers = [self.tree[0].get_headers(), self.tree[2].get_headers()]
        items = [self.tree[0].get_items(), self.tree[2].get_items()]
        self.tree[0].set_items([])
        self.tree[2].set_items([])
        self.tree[0].set_headers(headers[1])
        self.tree[2].set_headers(headers[0])
        self.tree[0].set_items(items[1])
        self.tree[2].set_items(items[0])
    def on_tree2_change_selection(self, tree):
        self.log('on_tree2_change_selection(' + str(tree.get_selection()) + ')')
Menu(
    Command('Widgets', lambda id: WidgetsDialog('Widgets Dialog').do_modal()),
    'Custom.Tests'
)
```

Executing the above script adds the following new dialog:
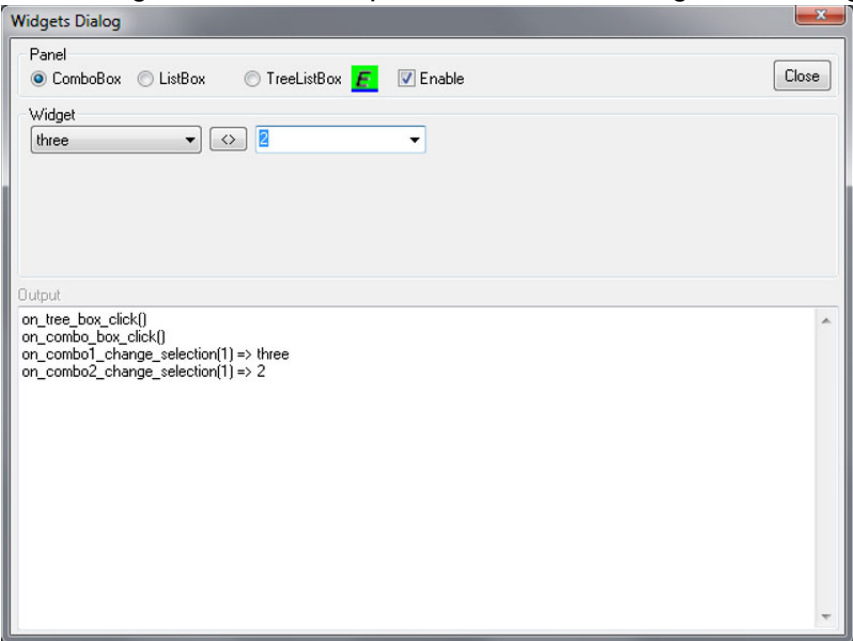
EXAMPLE OF OBJECT WIDGETS

```
from scade.tool.suite.gui.properties import Page
from scade.tool.suite.gui.widgets import Widget, GroupBox, ObjectComboBox, ObjectListBox,
TreeListBox, ObjectTreeListBox
import scade.model.suite as suite_scade
from scade import output
class Test(Page):
    def __init__(self, name):
        super().__init__(name)
    def is_available(self, models):
        available = len(models) == 1 and isinstance(models[0], suite_scade.Package)
        output('test_object_widgets.is_available(' + str(models) + ') => ' + str(available)
+ '\n')
        return available
    def on_context(self, models):
        output('test_object_widgets.on_context(' + str(models) + ')\n')
        self.models = models
    def on_build(self):
        self.group = GroupBox(self, 'Object Widgets')
        self.combo = ObjectComboBox(self, [], 20, 40, 100, 20, self.on_combo_change_selection,
style = 'dropdownlist')
        self.list = ObjectListBox(self, [], 20, 40, 200, 20, self.on_list_change_selection)
        self.tree = ObjectTreeListBox(self, [], 20, 40, 200, 20, self.on_tree_change_selection,
column_style = ['linesatroot'], headers = ObjectTreeListBox.Header('Interface'))
    def on_layout(self):
        output('test_object_widgets.on_layout(' + str(self.id) + ')\n')
        self.group.set_constraint(Widget.LEFT, self, Widget.LEFT, 10)
        self.group.set_constraint(Widget.TOP, self, Widget.TOP, 10)
        self.group.set_constraint(Widget.RIGHT, self, Widget.RIGHT, -10)
        self.group.set_constraint(Widget.BOTTOM, self, Widget.BOTTOM, -10)
        self.combo.set_constraint(Widget.LEFT, self.group, Widget.LEFT, 10)
        self.combo.set_constraint(Widget.TOP, self.group, Widget.TOP, 20)
        self.combo.set_constraint(Widget.WIDTH, self.group, Widget.WIDTH, -10, 0.2)
        self.list.set_constraint(Widget.LEFT, self.combo, Widget.RIGHT, 5)
        self.list.set_constraint(Widget.TOP, self.group, Widget.TOP, 20)
        self.list.set_constraint(Widget.WIDTH, self.group, Widget.WIDTH, -10, 0.4)
        self.list.set_constraint(Widget.BOTTOM, self.group, Widget.BOTTOM, -10)
        self.tree.set_constraint(Widget.LEFT, self.list, Widget.RIGHT, 5)
        self.tree.set_constraint(Widget.TOP, self.group, Widget.TOP, 20)
        self.tree.set_constraint(Widget.WIDTH, self.group, Widget.WIDTH, -10, 0.4)
        self.tree.set_constraint(Widget.BOTTOM, self.group, Widget.BOTTOM, -10)
```

```
    def on_display(self):
        # assert(len(self.models) == 1 and isinstance(self.models[0], suite_scade.Package))
        output('test_object_widgets.on_display(' + str(self.id) + ')\n')
        if self.models != None and len(self.models) > 0:
            # combo
            packages = []
            output('test_object_widgets.on_display Package1(' + str(packages) + ')\n')
            packages.extend(self.models[0].packages)
            output('test_object_widgets.on_display Package2(' + str(packages) + ')\n')
            if len(packages) == 0:
                packages.append(self.models[0])
            output('test_object_widgets.on_display Package2(' + str(packages) + ')\n')
            self.combo.set_items(packages)
            self.combo.set_selection(packages[0] if len(packages) > 0 else None)
            self.combo.set_enable(len(packages) > 1)
            # list / tree
            self.on_combo_change_selection(self.combo, 0)
    def on_validate(self):
        output('test_object_widgets.on_validate(' + str(self.id) + ')\n')
    def on_close(self):
        output('test_object_widgets.on_close(' + str(self.id) + ')\n')
    def on_combo_change_selection(self, combo, index):
        output('test_object_widgets.on_combo_change_selection(' + str(index) + ') => ' +
str(combo.get_selection()))
        operators = []
        package = combo.get_selection()
        if package:
            for o in package.sub_operators:
                operators.append(o)
        self.list.set_items(operators)
        self.list.set_selection(operators[0] if len(operators) > 0 else [])
        self.on_list_change_selection(self.list, [0])
    def on_list_change_selection(self, list, index):
        output('test_object_widgets.on_list_change_selection(' + str(index) + ')\n')
        self.tree.set_items([])
        for o in list.get_selection():
            self.tree.insert_item(ObjectTreeListBox.Item(o))
            for io in o.inputs:
                self.tree.insert_item(ObjectTreeListBox.Item(io, o))
            for io in o.hiddens:
                self.tree.insert_item(ObjectTreeListBox.Item(io, o))
            for io in o.outputs:
                self.tree.insert_item(ObjectTreeListBox.Item(io, o))
    def on_tree_change_selection(self, tree):
       output('test_object_widgets.on_tree_change_selection(' + str(tree.get_selection()) +
')\n')
Test('Widgets.Objects')
```

Executing the above script adds the following new properties:
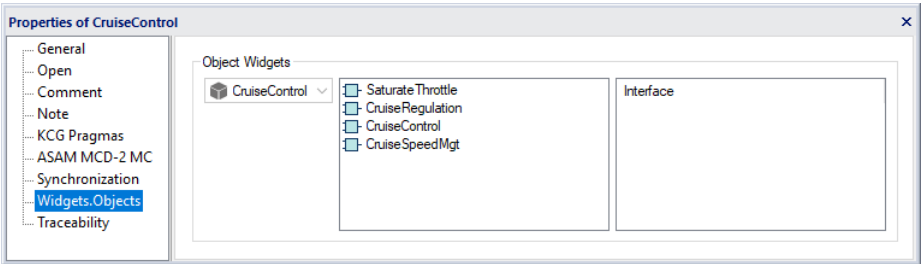
Callback Registration

The following functions can be used to register callbacks:

`scade.tool.suite.gui.register_terminate_callable`

Registering callback to be invoked when a custom module is about to be closed.

```
register_terminate_callable(on_terminate)
```

on_terminate	Callable to be invoked.

Example:

```
from scade.tool.suite.gui import register_terminate_callable
from scade import output

register_terminate_callable(lambda:
    output('script.py: on_terminate()\n')
)
```

`scade.tool.suite.gui.register_load_model_callable`

Registering callback to be invoked after a model is loaded.

```
register_load_model_callable (on_load_model)
```

on_load_model	Callable to be invoked taking a single parameter, the project pathname.

Example:

```
from scade.tool.suite.gui import register_load_model_callable
from scade import output

register_load_model_callable(lambda prj_path:
    output('script.py: on_load_model for project:' + prj_path + '\n')
)
```

`scade.tool.suite.gui.register_unload_model_callable`

Registering callback to be invoked after a model is unloaded.

```
register_unload_model_callable (on_unload_model)
```

on_unload_model	Callable to be invoked taking a single parameter, the project pathname.

Access to Predefined Operators in Python

A set of codes and enumerated values give access to all SCADE Suite predefined operators and higher-order operators needed when writing scripts using the SCADE Suite metamodel. To access such operators in Python scripts, you can use directly the enumerated value (see table below) or set a variable with its enumerated value. Unlike predefined operators and higher-order operators which are part of the Scade language, SCADE Suite library operators are identified by name and library where they are defined.

All available codes and enumerated values are listed in the following table:

Table 5.9: List of enumerated values and operator codes

Enumerated value	Code	Predefined Operators and Patterns
1	SC_ECK_NONE	No predefined operator
2	SC_ECK_AND	AND
3	SC_ECK_OR	OR
4	SC_ECK_XOR	XOR
5	SC_ECK_NOT	NOT
6	SC_ECK_SHARP	#
7	SC_ECK_PLUS	+
8	SC_ECK_SUB	binary –
9	SC_ECK_NEG	unary –
10	SC_ECK_MUL	*
14	SC_ECK_SLASH	/
16	SC_ECK_MOD	mod
18	SC_ECK_PRJ	projection
19	SC_ECK_CHANGE_ITH	assign of a structured element
20	SC_ECK_LESS	<

Table 5.9: List of enumerated values and operator codes (Continued)

21	SC_ECK_LEQUAL	<=
22	SC_ECK_GREAT	>
23	SC_ECK_GEQUAL	>=
24	SC_ECK_EQUAL	=
25	SC_ECK_NEQUAL	<>
26	SC_ECK_PRE	pre
28	SC_ECK_WHEN	When
29	SC_ECK_FOLLOW	->
30	SC_ECK_FBY	Fby
31	SC_ECK_IF	If
32	SC_ECK_CASE	Case
33	SC_ECK_SEQ_EXPR	sequence building
34	SC_ECK_BLD_STRUCT	Data structure
35	SC_ECK_MAP	Map
36	SC_ECK_FOLD	Fold
37	SC_ECK_MAPFOLD	Mapfold
38	SC_ECK_MAPI	Mapi
39	SC_ECK_FOLDI	Foldi
40	SC_ECK_SCALAR_TO_VECTOR	Scalar to vector
41	SC_ECK_BLD_VECTOR	Data array (vectors)
42	SC_ECK_PRJ_DYN	Projection with dynamic indexation
43	SC_ECK_MAKE	Make
44	SC_ECK_FLATTEN	Flatten
45	SC_ECK_MERGE	Merge
46	SC_ECK_REVERSE	Reverse

Table 5.9: List of enumerated values and operator codes (Continued)

47	SC_ECK_TRANSPOSE	Transpose
49	SC_ECK_TIMES	Times
50	SC_ECK_MATCH	Match
51	SC_ECK_SLICE	Slice
52	SC_ECK_CONCAT	Concatenation
53	SC_ECK_ACTIVATE	Activate
54	SC_ECK_RESTART	Restart
55	SC_ECK_FOLDW	Foldw
56	SC_ECK_FOLDWI	Foldwi
57	SC_ECK_ACTIVATE_NOINIT	Activate with default values
58	SC_ECK_CLOCKED_ACTIVATE	Clocked activate
59	SC_ECK_CLOCKED_NOT	NOT in clock expression
60	SC_ECK_POS	unary +
61	SC_ECK_MAPW	Mapw
62	SC_ECK_MAPWI	Mapwi
63	SC_ECK_NUMERIC_CAST	numeric cast
64	SC_ECK_MAPFOLDI	mapfoldi
65	SC_ECK_MAPFOLDW	mapfoldw
66	SC_ECK_MAPFOLDWI	mapfoldwi
67	SC_ECK_LAND	land
68	SC_ECK_LOR	lor
69	SC_ECK_LXOR	lxor
70	SC_ECK_LNOT	lnot
71	SC_ECK_LSL	lsl
72	SC_ECK_LSR	lsr

6 / Managing SCADE Extensions

This chapter explains how to manage Python packages stored on PyPI or on local machines to enrich your SCADE environment. SCADE Extension Manager enables installing, uninstalling, and updating such packages. SCADE user interface customization add-ons developed by SCADE experts in Python scripts can be managed in the same way using dedicated Python packages.

- "Installing/Updating Python Packages"
- "Defining Own Python Packages"

Installing/Updating Python Packages

From SCADE Extension Manager, you can install, update, or uninstall ANSYS extension packages stored at "*https://pypi.org*" URL. It is also possible to use local packages located in "*C:\Users\<user_name>\Downloads*" folder.

For advanced usage, it is possible to specify other package locations as explained later on.

To install, update, and uninstall Python packages

1 Select **Tools > Manage Extensions** to open the **Extension Manager** window.
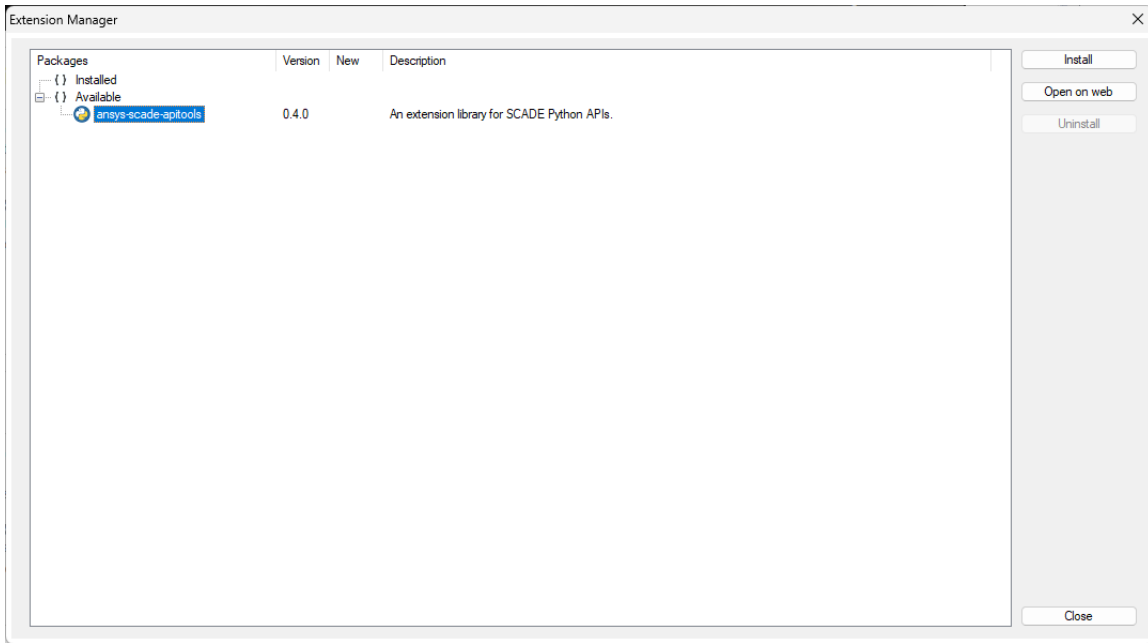
Figure 6.1: Extension Manager user interface

2 Manage extensions as follows:

- In the list of **Available** packages, select one or several packages to be installed and press **Install**.

- In the list of **Installed** packages, in case a package is installed and a new version is available, the new version is displayed in the **New** column. Select one or several packages to be updated and press **Install**. When the **New** column is empty, the latest version is installed.

- In the list of **Installed** packages, select one or several packages to be removed and press **Uninstall**.

Once a module is installed, you can get more information about it in your default web browser: select any installed module and press **Open on web**.

To specify Python package locations

1 Open file located at `%appdata%/Scade/Extension Manager/ settings.json`. This file contains settings for specifying package location. If the file does not exist, copy the `settings.json` file from SCADE installation at `SCADE/scripts/Extension Manager`. The default file content is:

```
{
    "repositories": [
        {
            "url": "https://pypi.org",
            "type": "pypi"
        },
        {
            "url": "%USERPROFILE%/Downloads",
            "type": "folder"
        }
    ],
    "check_at_startup": true,
    "timeout": [ 2, 5 ]
}
```

2 In the `repositories` section, you can specify the different locations you want to use. The `type` parameter can take the following values:

`pypi`	for using a standard pypi index. Only Ansys SCADE packages are considered in that context.
`gitlab_group`	for using the GitLab GraphQL API with a group URL.
`gitlab_project`	for using the GitLab GraphQL API with a project URL.
`gitlab_rest`	for using the GitLab Rest API with a project or a group URL.
`folder`	for looking for packages in a local folder. Expanded path values are supported as illustrated in the default file content above.

Defining Own Python Packages

Scripts can be defined and included in Python packages to extend SCADE User Interface as detailed below.

- "Python Packages for SCADE UI Customization"
- "Python Packages for Metrics and Rules"

PYTHON PACKAGES FOR SCADE UI CUSTOMIZATION

You can package Python customization scripts developed to extend SCADE User Interface by setting a Python package to be installed in SCADE Python environment.

The Python package must:

- Include a customization script as a Python module and an `.srg` file containing the following to register this script in SCADE environment:

```
[Custom/Studio/Extensions/<Extension Name>]
"Package"="<Package Name>.<Module Name>"
[Studio/Work Interfaces/<Extension Name>]
"Pathname"="ETCUST.DLL"
```

where `<Extension Name>` can be any name, `<Package Name>` is the name of the Python package being created, and `<Module Name>` is the name of the module containing the customization script.

The commands supported in customization scripts are available from "Customization-Related Python Commands" on page 108.

- In the metadata used to configure the Python package, add an entry point dedicated to SCADE usage. This entry point group is `ansys.scade.registry`. The entry point name is `srg` and it must contain the path to a Python function defined into the package. This function must have no arguments and must return the full path to the `.srg` file.

You can find below an example of the steps necessary for the creation of a Python package dedicated to SCADE UI customization. Consider the `my_package` package to be used as customization extension:

1 Create a `my_module.py` file corresponding to the customization script.

2 Add `my_module.srg` file to the package and configure the package to
   copy that file when the package is installed:

```
[Custom/Studio/Extensions/MyPackage]
"Package"="my_package.my_module"
[Studio/Work Interfaces/MyPackage]
"Pathname"="ETCUST.DLL"
```

3 Specify the SCADE entry point in `pyproject.toml`:

```
[project.entry-points."ansys.scade.registry"]
srg = "my_package:my_srg"
```

4 Implement the entry point function, for instance in `__init__.py` file:

```
def my_srg() -> str:
    # In this example, the srg file is located in the same directory
    return str(Path(__file__).parent / 'my_module.srg')
```

Once the package is built and pushed on PyPI or stored on a local directory, it
is available for installation using the Extension Manager as explained in
"Installing/Updating Python Packages" on page 149.

PYTHON PACKAGES FOR METRICS AND RULES

You can package metrics and rules in a Python package to be installed in
SCADE Python environment.

The Python package must:

• Include definitions of metrics and/or rules as a Python module.

You can specify metrics and rules as explained in "Creating User-Defined Metrics and Rules"
on page 508.

• In the metadata used to configure the Python package, add an entry point dedicated to
  SCADE usage. This entry point group is `ansys.scade.mrc`. The entry point name is
  `mrc_modules` and it must contain the path to a Python function defined into the
  package. This function must have no arguments and must return a space-separated list
  of Python module names which contain the definitions of rules and/or metrics.

You can find below an example of the steps necessary for the creation of a Python package dedicated to metrics and rules. Consider the `my_metrics_rules` package to be used for metrics and rules checking:

1 Create one module `my_metrics.py` defining the metrics and another module `my_rules.py` defining the rules.

2 Specify the SCADE entry point in `pyproject.toml`:

```
[project.entry-points."ansys.scade.mrc"]
mrc_modules = "my_metrics_rules:my_mrc_modules"
```

3 Implement the entry point function, for instance in `__init__.py` file:

```
def my_mrc_modules() -> str:
    return "my_metrics_rules.my_metrics my_metrics_rules.my_rules"
```

Once the package is built and pushed on PyPI or stored on a local directory, it is available for installation using the Extension Manager as explained in "Installing/Updating Python Packages" on page 149.

The metrics and rules included in Python packages are available for use in SCADE Suite Metrics and Rules Checker. For instructions, refer to Chapter 7 about "Computing Metrics and Checking Rules on Model" in *SCADE Suite User Manual*.

Index

Index