European Organisation for Astronomical Research in the Southern Hemisphere

Programme: ELT

Project/WP: Instrumentation Framework

ELT ICS Framework - Application Framework - User Manual

Document Number: ESO-363137

Document Version: 1

Document Type: Manual (MAN)

Released on: 2021-05-31

Document Classification: Public

Owner:	Andolfato, Luigi
Validated by PM:	Kornweibel, Nick
Validated by SE:	González Herrera, Juan Carlos
Validated by PE:	Biancat Marchet, Fabio
Appoved by PGM:	Tamai, Roberto
	Name

Release

This document corresponds to `rad`[1] v3.0.0.

Authors

Name	Affiliation
Andolfato, Luigi	ESO/DOE/CSE

Change Record from previous Version

Affected Section(s)	Changes / Reason / Remarks
	See CRE ET-1084
All	All sections updated

[1] https://gitlab.eso.org/ifw/rad

Table of Contents

1 Introduction

This User Manual describes how to build C++ applications for the ELT using the Rapid Application Development (RAD) toolkit.

RAD is an application framework that enables the development of event-driven distributed applications based on state machines.

The rest of the document describes:

- How an application based on RAD looks like.
- RAD libraries and tools.
- How to configure the user development environment, retrieve, build, and install RAD.
- How to create an application based on RAD from templates.
- Examples of applications based on RAD.
- COMODO tool for UML/SysML model transformations

2 RAD Based Applications

An application based on RAD toolkit reacts to internal or external events by invoking actions and/or starting activities as specified in a State Machine model.

2.1 Events

RAD based applications reacts to events. Events can be:

- Requests
- Replies
- Topics
- Timeouts
- Unix signals (CTRL-C, etc.)
- Internal events (events generated by the application itself)

Events are implemented by C++ classes containing an event identifier and a payload. To facilitate the application development, it is possible to define in a text file with extension .rad.ev the list of events (the identifier and the payload data structure). This file is then processed at compile time by the RAD tool codegen to generate the C++ classes (see for example *events.rad.ev*).

2.2 Event Loop

The event loop is responsible for continuously listening to requests, replies, topics, timeouts, UNIX signals, etc. and for invoking the associated callback. The callback creates an event which is inject it into the State Machine Engine. The State Machine engine depending on the current state and the injected event, selects which actions to invoke, which activities to start and to which state to move in.

In RAD the event loop is implemented using BOOST ASIO[1].

2.3 Actions

Actions represent short lasting tasks (ideally lasting 0 time) implemented using methods of a C++ class. They are similar to callback functions invoked when an event occurs and the application is in a given state.

[1] https://www.boost.org/doc/libs/1_72_0/doc/html/boost_asio.html

2.4 Activities

Activities represent long lasting tasks. They are started when entering a given state and are stopped when exiting the state. They can be implemented by:

- classes with a run() method which is executed on a separate thread.
- classes implementing co-routines.

2.5 State Machine Model

When to invoke an action or to start/stop an activity is defined in the State Machine model. The model describes for each state and event which action to invoke and which activity to start/stop. The State Machine model is specified using a domain specific language: StateChartXML (SCXML). SCXML[2] is a W3C recommendation that allows to specify a State Machine using XML (for an example see *sm.xml*). The SCXML State Machine model can be executed at run-time using an SCXML interpreter. RAD provides scxml4cpp library as SCXML interpreter. Note that events, actions, and activities C++ implementation have an identifier that should match the names in the SCXML model.

2.6 Error Handling

Exceptions and errors occurring within Actions, Guards, or Do-Activities, can be handled as follows in 3 ways:

- Local Error Handling: catching the exception and, in case of request, sending an error reply to the originator of the request, or, in case of other events, logging the error.
- State Machine Error Handling: catching the exception and triggering a related error event. In this case the error event should be handled by another action locally (via Local Error Handling). E.g. an exception occurs in a Do-Activity and the Do-Activity (secondary thread) post an error event into the State Machine (main thread) to send an error reply.
- Global Error Handling, the exception is caught by the main() function within the global try-catch.

2.7 Application Development

In order to develop a RAD based application, the developer as to provide:

- A text file with extension .rad.ev containing the list of internal and external events processed by the application.
- A text file in SCXML format containing the State Machine model.
- C++ implementation of the actions and activities classes.
- C++ implementation of the application configuration and runtime data classes.

[2] https://www.w3.org/TR/scxml/

RAD provides a fast way to create an application using Cookiecutter templates. By running the template(s) a fully working application with a basic State Machine model, events, and actions are generated.

See the tutorial *Tutorial 1: Creating an Application with RAD + CII* for detailed information on how to develop an application using RAD.

3 RAD Libraries and Tools

RAD libraries provide transparent access and integration with the Software Platform services. They also group functionalities common to all applications and not provided by the Software Platform.

3.1 Application Stack

ELT applications based on RAD are built on top of the following application stack:

Level	Application Stack	Description
4	Application	Your application(s)
3	Application Framework	RAD Libraries and Tools
2	Software Platform	Core Integration Infrastructure
1	Development Env.	Linux CentOS, GNU C++, waf, etc.
0	Hardware or VM	Servers

The ground level of the application stack are the ESO standard servers and Virtual Machines (VM). They are installed with the ELT Development Environment.

The ELT Development Environment, level 1, is based on Linux Cent OS and includes the GNU C++ compiler, waf building tool, and many other libraries such the Google Unit Tests, Robot framework for the integration tests, etc. (see: Guide to Developing Software for the EELT[3]).

The Software Platform, level 2, is a set of libraries, running on top of the Development Environment, that provides common services such as: Error Handling, Logging, Messaging, Configuration, In-memory DB (Online-DB), Alarms, etc. The official ELT Software Platform is the Core Integration Infrastructure (CII). Since there was the need to start developing applications before the introduction of CII and since not all CII services have been released yet, a Prototype SW platform (made of ZeroMQ, Google Protocol Buffers, C++ exceptions, EasyLogging, Redis in-memory DB, YAML configuration files) can also be used.

The currently official services to be used are listed in the following table.

Service	Description
Error Handling	C++ Exceptions
Logging	CII logging API based on log4cplus
Messaging	CII/MAL ZPB Req/Rep and Pub/Sub
Configuration	Based on files using YAML
Online-DB	Redis in-memory key/value DB

The application framework, level 3, can be used to develop State Machine based applications that use the services described in the table above.

[3] https://pdm.eso.org/kronodoc/HQ/ESO-288431

Warning: RAD will use more CII services as soon as they become available, therefore applications developed with the current version of RAD may have to be ported.

3.2 Libraries

RAD is made of the following libraries:

- utils
- core
- events
- mal
- services
- sm

All RAD classes and functions are declared within the *rad* namespace. Classes and functions using CII specific features have an additional namespace: *rad::cii*. For example: *rad::Helper*, *rad::cii::Publisher*.

For detailed information on the libraries classes and methods see the online RAD Doxygen documentation[4].

3.2.1 utils

Library providing common utility classes and functions. It does not depend on other RAD libraries.

Class	Description
Helper	Helper class providing static methods such as: GetHostname(), FindFile(), FileExists(), GetEnvVar(), CreateIdentity(), SplitAddrPort(), GetVersion().

The following free functions are going to be replaced by the Time Library once available (see ESO-331947).

Function	Description
GetTime	Get time of the day as double.
ConvertToIsoTime	Covert time of the day to ISO time string.
GetTimestamp	Get current time in ISO format.

[4] https://www.eso.org/~eltmgr/ICS/documents/RAD/doxygen_doc/html/index.html

3.2.2 core

Library providing error handling and logging services. It depends on *utils* library.

Class	Description
ErrorCategory	Class representing RAD errors.
Exception	RAD exception.

Function	Description
Assert	Assert a condition. If the condition is false, it logs a fatal error.
LogInitialize	Initializes log services.
LogConfigure	Load and configure log properties.
logGetLogger	Returns the default RAD logger (name = “rad”).
logGetSmLogger	Returns the RAD State Machine logger (name = “rad.sm”).

3.2.3 events

Library providing events related services. It does not depend on other RAD libraries.

Class	Description
Event	Class representing a specific event with template type for the payload.
AnyEvent	Class used to represent any event.

Function	Description
getPayload	Return a reference to the event payload.

3.2.4 mal

Library providing CII messaging services. It depends on *core* library and requires CII.

Class	Description
Publisher	Class that can be used to publish a topic using CII/ZPB.
Subscriber	Class that can be used to subscribe to topic using CII/ZPB.
Replier	Class that can be used to receive commands and send replies using CII/ZPB.
Requestor	Class that can be used to send commands and receive replies using CII/ZPB.
Request	Class representing a command and the associated [error] reply.

3.2.5 services

Library providing DB, and other services. It depends on *core* library.

Class	Description
DbAdapter	Interface to read/write to a key-value in-memory DB.
DbAdapterRedis	Realization of DbAdapter interface for Redis DB.

Note: This library contains other classes that allows to use the messaging services from the Prototype Software Platform. These classes are now replaced by the *mal* library.

3.2.6 sm

Library providing State Machine services. It depends on *services*, *events*, and *scxml4cpp* libraries.

Class	Description
ActionCallback	Class mapping a void class method to an scxml4cpp::Action object.
GuardCallback	Class mapping a boolean class method to an scxml4cpp::Action object.
ActionGroup	Base class for classes grouping action methods.
ThreadActivity	Base class for do-activities implemented as standard C++ threads.
PthreadActivity	Base class for do-activities implemented as Posix threads.
CoroActivity	Base class for do-activities implemented as Co-routines.
ActionMgr	Base class for instantiating actions and do-activities.
Signal	Class for deealing with UNIX signals events.
Timer	Class for dealing with time-out events.
SMEvent	Class to wrap RAD events into SCXML events.
SMAdapter	Facade to the SCXML State Machine engine.

Note: This library contains other classes that allows to use the Prototype Software Platform instead of CII.

3.2.7 scxml4cpp

scxml4cpp is an ESO library able to parse and execute an SCXML model. It is made two components: the parser and the engine. The parser is based on xerces-c++[5] and it is used to parse the XML file containing the SCXML State Machine model. The engine is used to interpret at run-time the SCXML State Machine model following the W3C algorithm[6].

[5] https://xerces.apache.org/xerces-c/
[6] https://www.w3.org/TR/scxml/

3.3 Tools

The following tools are part of RAD toolkit:

- cookiecutters to create C++ skeleton application.
- codegen to create events C++ classes.
- COMODO to translate State Machine models from SysML/UML to SCXML.

3.3.1 Cookiecutters

Cookiecutters is an open-source tool (see Cookiecutter[7]) that is used to generate a RAD based applications from templates. The templates are stored in rad/rad/cpp/templates/config directory.

3.3.2 codegen

codegen is an ESO tool that takes as input a YAML text file and generates C++ classes with events implementation. It is invoked by waf at compile time. Generated files are in the build/ directory.

3.3.3 COMODO

COMODO is an ESO tool that takes as input a SysML/UML model of an application following the COMODO profile and generates the XML file containing the SCXML State Machine model. For more information see *Tool*

[7] https://cookiecutter.readthedocs.io

4 RAD Installation

4.1 Environment Configuration

To configure environment variables LMOD tool (https://europeansouthernobservatory.sharepoint.com/sites/EELT-ICS-SWFW/Wiki/EELT%20LMOD.aspx) is used. It replaces the VLT PECS tool.

LMOD is based on LUA language. The configuration of the env. variables should be stored in the "/modulefiles/private.lua file. For example:

```
local home = os.getenv("HOME")

local introot = pathJoin(home, "EELT/EELT-INTROOT")
setenv("INTROOT", introot)
setenv("PREFIX", introot)

load("introot")

local cfgpath = pathJoin(home, "EELT/EELT-INTROOT/resource/config")
setenv("CFGPATH", cfgpath)
```

Note:

- PREFIX is needed by waf to know where to install binaries and libraries.
- INTROOT is usually the same as PREFIX.
- CFGPATH can be used to define the paths where applications configuration files are located. It has therefore to include the INTROOT/PREFIX directory.

To (re-)load your private.lua module from the terminal:

```
>module load private
```

4.2 Retrieving RAD from GIT

RAD is archived in GIT repository: https://gitlab.eso.org/ifw/rad

It can be retrieve with the following command:

```
>git clone https://gitlab.eso.org/ifw/rad
```

Note: Username and password have to be provided.

4.3 Building and Installing RAD

RAD can be compiled with the following commands:

```
>waf configure
>waf build
```

RAD can be installed into the $PREFIX directory by:

```
>waf install
```

4.4 Directory Structure

RAD project is organized in the following directories:

Directory	Description
docs	RAD User Manual.
rad	RAD Libraries.
scxml4cpp	SCXML State Machine engine for C++.
scxml4py	SCXML State Machine engine for Python.
test	RAD Integration Tests.

RAD Libraries are organized in the following directories:

Directory	Description
rad/codegen	Code generator to create the event classes.
rad/cpp/utils	Library providing common utility functions (e.g. FindFile)
rad/cpp/core	Library providing error handling and logging services.
rad/cpp/events	Library providing events related services.
rad/cpp/mal	Library providing CII messaging services.
rad/cpp/services	Library providing ZMQ messaging, DB, and other services.
rad/cpp/sm	Library providing State Machine service.
rad/cpp/templates	Templates to create RAD projects and applications.
rad/cpp/_examples	Examples of applications based on RAD.

5 RAD Integration Tests

RAD has two sets of integration tests located in rad/test directory:

- The first set is in *rad/test/rad*. They use the application described in *Examples* to test RAD libraries.
- The second set is in *rad/test/template* and are used to test the templates. These integration tests use the Cookiecutter templates illustrated in the tutorials to generate RAD applications and associated interfaces. Then it compiles the generated modules and executes the generated tests to verify RAD libraries.

These tests are executed daily by the ELT Continuous Integration infrastructure using the latest RAD sources. The results can be accessed from Jenkins[8].

To execute manually the tests:

```
> cd rad/test/rad/src
> robot *.robot

> cd rad/test/templates/src
> robot *.robot
```

[8] https://eltjenkins.hq.eso.org/job/ICS/job/RAD/job/ifw-rad-daily/

6 Tutorial 1: Creating an Application with RAD + CII

In order to develop an application based on RAD, the following steps are performed:

1. Generate WAF Project
2. Generate Interface Module
3. Generate msgSend Module
4. Generate Application Module
5. Generate Integration Test Module
6. Build and Install Generated Modules
7. Run Integration Tests
8. Customize Application, Test, and Interface modules

Note: Steps 1, 2, 3 can be skipped if you are adding your application to an existing project.

6.1 Generate CII WAF Project

In order to be able to build, an ELT application needs to be part of a WAF project. This means that there must be a directory (e.g. "hello") that contains a "wscript" file declaring the root of a WAF project. See WAF User Manual[9] for information on WAF projects.

An "hello" WAF project can be created by executing the following commands and entering the requested information:

```
> cookiecutter rad/rad/cpp/templates/config/rad-waftpl-malprj

project_name [hello]: hello
modules_name [hellomalif hellomalifsend hellomal]:
```

The input values to the template are:

- *project_name* the name of the WAF project which is used to create the directory containing the project SW modules.
- *modules_name* the name of the SW modules part of this project.

Note: By pressing enter, the default values (in square brackets) are selected.

From the template Cookiecutter generates the directory *hello* and inside the file *wscript*. This file

[9] http://eeltdev8.hq.eso.org:8080/job/DevEnv-WTools-doc/lastSuccessfulBuild/artifact/html/index.html

contains the WAF project declaration, the features and libraries required to compile this project, and the name of SW modules to compile.

6.2 Generate CII Interface Module

All commands, replies, and topics used to communicate between ELT applications, must be specified in dedicated interface modules. For the CII Software Platform, interfaces are specified using XML.

A CII interface module containing the “standard” commands can be created by executing the following commands and entering the requested information:

```
> cd hello
> cookiecutter ../rad/rad/cpp/templates/config/rad-cpptpl-malapplif

module_name [hellomalif]: hellomalif
parent_package_name [hello]: hello
```

The input values to the template are:

- *module_name* the name of the SW module to be generated (which contains the interface specification).
- *parent_package_name* the name of the directory that contains the module. In this case it is the project directory.

From the template Cookiecutter generates the directory *hellomalif* containing the following files:

File	Description
hellomalif/wscript	WAF file to compile the SW module.
hellomalif/src/hellomalif.xml	CII MAL XML file with the interface definition.

The file hellomalif.xml looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="schemas/icd_type_definition.xsd">

 <package name="hellomalif">
  <exception name="ExceptionErr">
   <member name="desc" type="string"/>
  </exception>

  <struct name="LogInfo">
     <member name="level" type="string"/>
     <member name="logger" type="string"/>
  </struct>

  <interface name="StdCmds">
```

(continues on next page)

(continued from previous page)

```
    <method name="Init" returnType="string" throws="ExceptionErr"/>
    <method name="Reset" returnType="string" throws="ExceptionErr"/>
    <method name="Enable" returnType="string" throws="ExceptionErr"/>
    <method name="Disable" returnType="string" throws="ExceptionErr"/>
    <method name="GetState" returnType="string" throws="ExceptionErr"/>
    <method name="GetStatus" returnType="string" throws="ExceptionErr"/>
    <method name="GetVersion" returnType="string" throws="ExceptionErr"/>
    <method name="Stop" returnType="string" throws="ExceptionErr"/>
    <method name="Exit" returnType="string" throws="ExceptionErr"/>
    <method name="SetLogLevel" returnType="string" throws="ExceptionErr">
      <argument name="info" type="nonBasic" nonBasicTypeName="LogInfo"/>
    </method>
  </interface>

 </package>
</types>
```

It specifies:

- the exception *ExceptionErr*, that can be used for error replies.
- the data structure *LogInfo*, used as parameter in the *SetLogLevel* command.
- nine commands that return a string as normal reply.
- the *SetLogLevel* can raise an exception of type ExceptionErr in case of errors. The *SetLogLevel* command is the only one taking a parameter of type *LogInfo*.

For more information on the CII/MAL XML interface definition language, refer to MAL ICD User Manual[10].

The hellomalif.xml is transformed into C++ code at compile time by the CII/MAL code generator and by Google ProtoBuf compiler. Generated code is located in hello/build directory.

6.3 Generate CII msgSend Module

msgSend is a command-line tool that can be used to send the commands defined in the Interface Module(s) to ELT applications and to collect the replies.

The msgSend tool used in this tutorial is not meant to be used for production. It is just a very basic implementation needed to run the examples.

Note: Since interfaces are specified at compile time in the Interface Modules, msgSend must also depend, at compile time, on the Interface Modules. In the VLT, msgSend could be used with the '-n' to send commands not defined in the CDT. This is not possible in the ELT.

---

[10] https://pdm.eso.org/kronodoc/HQ/ESO-348602

A SW module implementing the msgSend tool to send the “standard” commands to CII applications can be created by executing the following commands and entering the requested information:

```
> cd hello
> cookiecutter ../rad/rad/cpp/templates/config/rad-cpptpl-malsend

interface_name [hellomalif]: hellomalif
interface_module [hellomalif]: hellomalif
module_name [hellomalifsend]: hellomalifsend
application_name [hellomalifSend]: hellomalifSend
parent_package_name [hello]: hello
```

The input values to the template are:

- *interface_name* the name of the CII Interface module (which specifies the commands sent by the msgSend tool and is defined in *Generate CII Interface Module*).
- *interface_module* fully qualified name of CII Interface library.
- *module_name* the name of the SW module to be generated (which contains the msgSend tool).
- *application_name* the name of the binary to be produced when compiling the generated module.
- *parent_package_name* the name of the directory that contains the tool. In this case it is the project directory.

From the template Cookiecutter generates the directory *hellomalifsend* containing the following files:

File	Description
hellomalifsend/wscript	WAF file to compile the SW module.
hellomalifsend/src/main.cpp	Implementation of msgSend.

The tool can be invoked by:

```
hellomalifSend [-v] <serviceURI> <request> <parameters>

-v            verbose with debug info.
<serviceURI>  destination of the command (e.g. zpb.rr://127.0.0.1:12081/StdCmds)
<request>     command to be sent (e.g. GetState)
<parameters>  parameters of the command
```

6.4 Generate CII Application Module

RAD provides the templates to create a simple server application implementing the standard ELT State Machine model using the CII Software Platform services.

An application that uses the services of the CII Software Platform can be created by executing the following commands and entering the required information:

```
> cd hello
> cookiecutter ../rad/rad/cpp/templates/config/rad-cpptpl-malappl

module_name [hellomal]: hellomal
application_name [hellomal]: hellomal
parent_package_name [hello]: hello
interface_name [hellomalif]: hellomalif
interface_module [hellomalif]: hellomalif
```

The input values to the template are:

- *module_name* the name of the SW module to generate.
- *application_name* the name of the binary to be produced when compiling the generated module.
- *parent_package_name* the name of the directory that contains the SW module. In this case it is the project directory.
- *interface_name* the name of SW module containing the interface specification.
- *interface_module* the fully qualified name name of the interface library.

From the template Cookiecutter generates the directory *hellomal* containing the following files:

File	Description
wscript	WAF file to build the application.
resource/config/config.yaml	YAML application configuration file.
resource/config/sm.xml	SCXML file with the State Machine model.
resource/config/log.properties	Logging configuration file.
src/events.rad.ev	List of events processed by the application.
src/actionMgr.[hpp|cpp]	Class responsible for instantiating actions and activities/
src/actionsStd.[hpp|cpp]	Class implementing standard action methods.
src/config.[hpp|cpp]	Class loading YAML configuration file.
src/dataContext.[hpp|cpp]	Class used to store application run-time data shared between action classes.
src/dbInterface.[hpp|cpp]	Class interfacing to the in-memory DB.
src/logger.[hpp|cpp]	Default logger definition.
src/stdCmdsImpl.hpp	Class implementing CII/MAL asynchronous server interface.
src/main.cpp	Application entry function.

6.4.1 wscript

This file is used by WAF to build the application binary.

```
from wtools.module import declare_cprogram

declare_cprogram(target='hellomal',
                 features='radgen',
                 use='log4cplus yaml-cpp hiredis protobuf xerces-c
                      rad.cpp.core rad.cpp.services rad.cpp.events
                      rad.cpp.sm rad.cpp.utils rad.cpp.mal hellomal-cxx')
```

It specifies the target binary name *hellomal*, which tools to use for building (e.g. *radgen* to transform *events.rad.ev* into C++ classes), and which libraries to link:

- *log4cplus* for logging.
- *yaml-cpp* to load YAML configuration files.
- *hiredis* to access Redis DB.
- *protobuf* to serialize/deserialize Google Protocol Buffers messages.
- *xerces-c* required to parse SCXML State Machine Model.
- *rad.cpp.core*, . . . RAD libraries.
- *hellomal-cxx* CII/MAL generated library.

6.4.2 config.yaml

This file contains the application configuration in YAML format.

```
cfg.req.endpoint     : "zpb.rr://127.0.0.1:12081/"
cfg.db.endpoint      : "127.0.0.1:6379"
cfg.db.timeout_sec   : 2
cfg.sm.scxml         : "hellomal/sm.xml"
cfg.log.properties   : "hellomal/log.properties"
```

The fields are mapped to the configuration attributes defined in the *Config* class (see *config.hpp*|*cpp*) which is responsible for loading the config.yaml file.

- *cfg.req.endpoint* CII/MAL endpoint to be used to receive commands.
- *cfg.db.endpoint* IP address and port to connect to Redis DB.
- *cfg.db.timeout_sec* Timeout when accessing Redis DB.
- *cfg.sm.scxml* SCXML State Machine model file path (see *sm.xml*).
- *cfg.log.properties* Logging configuration file path (see *log.properties*).

6.4.3 log.properties

Logging APIs are provided by log4cplus library[11]. Logging service can be configured via the following configuration file.

```
log4cplus.rootLogger=INFO, console

log4cplus.logger.rad=INFO, console
log4cplus.additivity.rad=false

log4cplus.logger.rad.sm=INFO, console
log4cplus.additivity.rad.sm=false

log4cplus.logger.scxml4cpp=INFO, console
log4cplus.additivity.scxml4cpp=false

log4cplus.logger.hellomal=INFO, console
log4cplus.additivity.hellomal=false

log4cplus.appender.console=log4cplus::ConsoleAppender
log4cplus.appender.console.layout=log4cplus::PatternLayout
log4cplus.appender.console.layout.ConversionPattern={{'[%D{%H:%M:%S:%q}][%-5p][
↪%c] %m%n'}}
```

In the file it is possible to specify the log level for each logger (*rootLogger*, *rad*, *scxml4cpp*, *hellomal*), where the logs should be published (appenders, e.g. *console*) and the format for each appender.

Default application logger is specified in *logger.hpp|cpp* files.

6.4.4 sm.xml

This file contains the SCXML representation of the standard ELT State Machine model.

```
<?xml version="1.0" encoding="us-ascii"?>
<scxml xmlns="http://www.w3.org/2005/07/scxml" xmlns:customActionDomain="http://
↪my.custom-actions.domain/CUSTOM"
 version="1.0" initial="State">

  <state id="On">
   <initial>
     <transition target="NotOperational"/>
   </initial>

   <state id="NotOperational">
     <initial>
       <transition target="NotReady"/>
     </initial>
```

(continues on next page)

[11] https://github.com/log4cplus/log4cplus

(continued from previous page)

```
    <state id="NotReady">
      <transition event="Events.Init" target="Ready">
        <customActionDomain:ActionsStd.Init name="ActionsStd.Init"/>
      </transition>
    </state>

    <state id="Ready">
       <transition event="Events.Enable" target="Operational">
         <customActionDomain:ActionsStd.Enable name="ActionsStd.Enable"/>
       </transition>
    </state>
  </state>

  <state id="Operational">
    <initial>
      <transition target="Idle"/>
    </initial>

    <state id="Idle">
      <transition event="Events.Stop" target="Ready">
         <customActionDomain:ActionsStd.Stop name="ActionsStd.Stop"/>
      </transition>
    </state>

    <transition event="Events.Disable" target="Ready">
      <customActionDomain:ActionsStd.Disable name="ActionsStd.Disable"/>
    </transition>
  </state>

  <transition event="Events.Init">
    <customActionDomain:ActionsStd.Init name="ActionsStd.Init"/>
  </transition>
...
   <transition event="Events.Exit" target="Off">
    <customActionDomain:ActionsStd.Exit name="ActionsStd.Exit"/>
  </transition>

  <transition event="Events.CtrlC" target="Off">
    <customActionDomain:ActionsStd.ExitNoReply name="ActionsStd.ExitNoReply"/>
  </transition>
 </state>

 <final id="Off">
 </final>
</scxml>
```

The State Machine consists of the following states:

- A composite outer state *On* indicating that the application has started.

- A composite state *On/NotOperational* indicating that the application and controlled devices cannot be used yet for operation.
- A leaf state *On/NotOperational/NotReady* indicating that the application and devices have not been fully initialized yet.
- A leaf state *On/NotOperational/Ready* indicating that the application has been initialized (but the devices may not be ready).
- A composite state *On/Operational* indicating that the application and controlled devices provide all operational functionalities.
- A leaf state *On/Operational/Idle* indicating that the application and the controlled devices are available to be used for operation.
- A final pseudo-state *Off* to indicate that the application has terminated.

The State Machine presents the following transitions:

- It is possible to move from *On/NotOperational/NotReady* to *On/NotOperational/Ready* via the *Events.Init* event.
- It is possible to move from *On/NotOperational/Ready* to *On/Operational/Idle* via the *Events.Enable* event.
- It is possible to move from *On/Operational/Idle* to *On/NotOperational/Ready* via the *Events.Disable* event.
- It is possible to move from any state back to *On/NotOperational/NotReady* via the *Events.Reset* event.
- It is possible to terminate the application from any state via the *Events.Exit* or the *Events.CtrlC* events.

Note: The SCXML State Machine model is Software Platform independent. The same model is used for the CII Software Platform and for the Prototype Software Platform. The SCXML engine is also Software Platform independent: scxml4cpp is also used in WSF2 for the VLT Software Platform.

6.4.5 events.rad.ev

This file contains the definition of the events triggering the transitions in the State Machine model.

```
# Event definitions for hellomal application
version: "1.0"

namespace: Events

includes:
    - boost/exception_ptr.hpp
```

(continues on next page)

(continued from previous page)

```
    - rad/mal/request.hpp
    - Hellomalif.hpp

events:
     Exit:
             doc: Event for the Exit request message.
             payload: rad::cii::Request<std::string>
     GetState:
             payload: rad::cii::Request<std::string>
     GetStatus:
             payload: rad::cii::Request<std::string>
     GetVersion:
             payload: rad::cii::Request<std::string>
         Stop:
             payload: rad::cii::Request<std::string>
     Reset:
             payload: rad::cii::Request<std::string>
     Init:
             payload: rad::cii::Request<std::string>
     Enable:
             payload: rad::cii::Request<std::string>
     Disable:
             payload: rad::cii::Request<std::string>
     SetLogLevel:
             payload: rad::cii::Request<std::string, std::shared_ptr
↪<Hellomalif::LogInfo>>
     CtrlC:
             doc: Event representing the CTRL-C (SIGINT) linux signal.
```

For each event it is possible to specify an optional event description and event payload type.

For CII commands, the event payload is associated to the command and reply payloads. RAD provides a wrapper class, *rad::cii::Request*, that allows to access the command payload and to set the replay payload. The wrapper class takes therefore two templates parameters:

- the data type of the reply parameter
- the data type of the command parameter (optional)

For example the SetLogLevel event is associated to the CII/MAL SetLogLevel command and reply. The reply payload data type is *std::string* while the command takes as parameter a shared pointer of the LogInfo data structure defined in CII Interface module.

Other events like *Exit*, . . . , *Config* are associated to commands without parameters and with *std::string* as reply data type.

Finally *CtrlC* event has no payload and it is not associated to any command.

From this file, if the feature “radgen” is specified in the *wscript*, the *events.rad.ev.hpp* files are generated in the *build/* directory and compiled as part of the WAF build process.

The events.rad.hpp file, generated from events.rad.ev, looks like:

```
#ifndef EVENTS_EVENTS_RAD_HPP
#define EVENTS_EVENTS_RAD_HPP

#include <rad/AnyEvent.hpp>
#include <boost/exception_ptr.hpp>
#include <rad/mal/request.hpp>
#include <Hellomalif.hpp>

namespace Events {

class Init final : public rad::AnyEvent {
public:
 static constexpr char const* id = "Events.Init";
 static constexpr rad::EventInfo::Context ctx = rad::EventInfo::Context::any;
 using payload_t = rad::cii::Request<std::string>;
 ...
private:
 rad::cii::Request<std::string> m_payload;
};
...
```

6.4.6 actionsStd.hpp|cpp

The *ActionsStd* class implements the actions defined in the State Machine model (see *sm.xml*).

For example the action *ActionStd.Init* executed when the event *Events.Init* occurs:

```
<transition event="Events.Init" target="Ready">
   <customActionDomain:ActionsStd.Init name="ActionsStd.Init"/>
</transition>
```

is mapped to the C++ method defined in *actionsStd.hpp* which takes the *last_event* as parameter:

```
/**
 * Implementation of the Init action. This action:
 * - replies back for the originator of the ReqInit request.
 *
 * @param[in] lastEvent Last event received which should be a Init event
 * (the Init event is triggered by a ReqInit request).
 */
void Init(const rad::AnyEvent& last_event);
```

The method's implementation in *actionsStd.cpp* extracts from the *last_event* the event's payload (defined in *events.rad.ev*) which is of type *std::shared_ptr<rad::cii::Request<std::string>>>*, sets the OK reply message and returns.

```
void ActionsStd::Init(scxml4cpp::Context* c) {
    RAD_TRACE(GetLogger());

    auto req = rad::GetLastEventPayloadNothrow< Events::Init > (c);
    if (req == nullptr) {
        LOG4CPLUS_ERROR(GetLogger(), "Init event has no associated request!");
        return;
    }
    req->SetReplyValue("OK");
}
```

The one above is just the simple case where the actions does nothing but replying to the originator of the Init command. More elaborated actions implementation can be found in the *Examples* section.

6.4.7 actionMgr.hpp|cpp

The *ActionMgr* class is responsible, via the *CreateActions* method, for instantiating the classes implementing the actions methods (e.g. *ActionsStd*). This method sets up also the callbacks to be invoked by the State Machine engine when an action has to be executed. The callback mechanism is based on the a function call stored in the *ActionCallback* object.

```
void ActionMgr::CreateActions(boost::asio::io_service& ios,
                              rad::SMAdapter& sm,
                              DataContext& the_data) {
    RAD_TRACE(GetLogger());

    /* Create action group classes */
    ActionsStd* actions_std = new ActionsStd(ios, sm, the_data);
    if (actions_std == nullptr) {
        LOG4CPLUS_ERROR(GetLogger(), "Cannot create actions_std object.");
        return;
    }
    AddActionGroup(actions_std);

    /* Create SM call-backs */
    scxml4cpp::Action* the_action = nullptr;
    using std::placeholders::_1;

    the_action = new rad::ActionCallback("ActionsStd.Init",
                 std::bind(&ActionsStd::Init, actions_std, _1));
    AddAction(the_action);

    the_action = new rad::ActionCallback("ActionsStd.GetState",
                 std::bind(&ActionsStd::GetState, actions_std, _1));
    AddAction(the_action);

    ...
}
```

Moreover the ActionMgr class is responsible for instantiating the classes implementing the do-activities as threads or co-routines.

```
void ActionMgr::CreateActivities(rad::SMAdapter& sm,
                                 DataContext& the_data) {
    RAD_TRACE(GetLogger());
    /*
     * Instantiate here rad::ThreadActivity,
     * rad::PthreadActivity, or rad::CoroActivity.
     */
    /*
     * For example:
     * rad::ThreadActivity activity = nullptr;
     * activity = new ActivityMoving("ActivityMoving", sm, the_data);
     * AddActivity(activity);
     */
}
```

The CreateActions and CreateActivities methods take as parameters:

- a reference to *rad::SMAdapter* which is used to post internal events to the State Machine engine.
- a reference to the application *DataContext* to be able to share run-time data among actions and do-activities.

In addition, the CreateActions method takes a reference to the *boost::asio::io_service* to allow the *ActionsStd::Exit* and *ActionsStd::ExitNoReply* methods to stop the BOOST ASIO event loop.

The ActionMgr object is instantiated in *main.cpp*.

6.4.8 config.hpp|cpp

The *Config* class is responsible for providing (read-only) access to the application configuration. The configuration can come from:

- Default values defined in *config.hpp*
- Environment Variables
- Command line parameters
- Application configuration file *config.yaml*

The *Config* class constructor initializes the configuration attributes with the default values and the environment variables (if available). The class provides the methods *ParseOptions()* to read the command line parameters and *LoadConfig()* to load the configuration file.

Configuration parameters can be obtained via the Get methods.

Note: Configuration files are loaded from the current directory or from any directory listed in the

CFGPATH environment variable.

The Config object is instantiated in *main.cpp*.

6.4.9 dbInterface.hpp|cpp

The *DbInterface* class is responsible for getting/setting application information in the in-memory DB via the Get/Set methods.

The header file provides the key definitions to be used when writing key-value pairs in the DB or when reading the values associated to given keys.

The generated class provides some Get and Set methods to read/write the application state and configuration:

- GetControlState(), GetControlSubstate()
- SetControlState(), SetControlSubstate()
- SetConfig(Config& cfg)

The developer can add all the Get/Set methods required by the application.

The DbInterface constructor takes as parameters:

- A string representing the prefix to be added to all the “keys” before writing in DB
- A reference to an object that allows to talk to the Online DB. This object should implement the rad::DbAdapter interface. RAD provides a class which is specialized to talk to Redis DB: rad::DbAdapterRedis.

The DbInterface object instantiated in *main.cpp* and used by the *DataContext* class.

6.4.10 dataContext.hpp|cpp

The *DataContext* class provides (read/write) access to actions and activities to the application run-time information. This class allows to also to write the run-time information to the in-memory DB via the *DbInterface* class.

Since run-time information may need to be initialized with application configuration data, a reference to the *Config* class is passed via the constructor. The class provides also a method to reload the application configuration (*ReloadConfig()*).

The DataContext is instantiated in *main.cpp*.

6.4.11 logger.hpp|cpp

log4cplus library provides the possibility to associated logs to different loggers. This features allows to set the log level (and therefore enable/disable logging) for given loggers. For example it is possible to enable logging for an application secondary thread and disable the logging for the main thread by using different loggers: one associated to the main thread and one associated to the secondary thread.

RAD suggests to use for the main application thread a common global logger which takes the name from the SW module name and it is defined in the logger.hpp file.

```
const std::string LOGGER_NAME = "hellomal";
```

The logger can be obtained from the free function implemented in logger.cpp:

```
log4cplus::Logger& GetLogger() {
    static log4cplus::Logger logger = log4cplus::Logger::getInstance(LOGGER_
↪NAME);
    return logger;
}
```

For secondary threads (e.g. Activity classes) or in case of loggers dedicated to given classes, it is suggested to declare the logger as class attribute and use it in the logging macros. The name of specialized logger should use the SW module name as prefix (e.g. “hellomal.ActivityName”) to allow an easy enabling/disabling of all application logs.

Note:

- There is an overhead in using the *log4cplus::Logger::getInstance(loggerName)* method and therefore it is preferable to avoid calling that method every time we need to log.
- The *RAD_ASSERT* macros use the rootLogger.

6.4.12 stdCmdsImpl.hpp

The *StdCmdsImpl* class implements the CII/MAL/ZPB interface specified in CII Interface module. In the constructor it takes a reference to the *rad::SMAdapter* used to inject the event associated to the command into the State Machine engine.

```
class StdCmdsImpl : public hellomalif::AsyncStdCmds {
public:
 explicit StdCmdsImpl(rad::SMAdapter& sm) : m_sm(sm) {
     RAD_TRACE(GetLogger());
 }

 virtual elt::mal::future<std::string> Init() override {
```

(continues on next page)

(continued from previous page)

```
    RAD_TRACE(GetLogger());
    auto ev = std::make_shared<Events::Init>();
    m_sm.PostEvent(ev);
    return ev->GetPayload().GetReplyFuture();
 }

 ...

 virtual elt::mal::future<std::string> SetLogLevel(const std::shared_ptr
↪<hellomalif::LogInfo>& info) override {
    RAD_TRACE(GetLogger());
    auto ev = std::make_shared<Events::SetLogLevel>(info->clone());
    m_sm.PostEvent(ev);
    return ev->GetPayload().GetReplyFuture();
 }

private:
 rad::SMAdapter& m_sm;
};
```

The *Init()* method is invoke when the hellomalif.Init command is sent to the hellomal application (for example using the msgSend application). This method creates the event *Events::Init* associated to the command and post it to the State Machine engine via the rad::SMAdapter::PostEvent() method. Note that the Init command has no parameters and the reply has std::string as payload. Finally the *Init()* method return a future of type std::string to the CII/MAL server.

The *SetLogLevel()* method is similar to the *Init()* but, in addition, it takes a parameter of type *hellomalif::LogInfo* as argument. The parameter becomes the payload of the *Events::SetLogLevel* event.

Note: The methods of the StdCmdsImpl class are invoked from the CII/MAL server thread and not from the main application thread.

The StdCmdsImpl object is instantiated in the *main.cpp* as part of the CII/MAL server initialization.

6.4.13 main.cpp

The *main()* function of an application based on RAD is responsible for creating all the required objects, initializing the services and start the event loop.

It starts by initializing the logging library and loading the ZeroMQ+ProtoBuf CII/MAL middleware. Follows the loading of the application configuration via the *Config* class (see *config.hpp|cpp*) and the creation of the *DataContext* used to exchange run-time information between actions and activities.

```
int main(int argc, char *argv[]) {
 rad::LogInitializer log_initializer;
```

(continues on next page)

(continued from previous page)

```
LOG4CPLUS_INFO(hellomal::GetLogger(), "Application hellomal started.");

try {
    /*
     *  Load CII/MAL middleware here because it resets
     *  the log4cplus configuration!
     */
    rad::cii::LoadMiddlewares({"zpb"});

    /* Read only configuration */
    hellomal::Config config;
    if (config.ParseOptions(argc, argv) == false) {
        // request for help
        return EXIT_SUCCESS;
    }
    config.LoadConfig();
    log_initializer.Configure(rad::Helper::FindFile(config.GetLogProperties()));

    /*
     * LAN 2020-07-09 EICSSW-717
     * Create CII/MAL replier as soon as possible to avoid problems when
     * an exceptions is thrown from an Action/Guard.
     */
    rad::cii::Replier mal_replier(elt::mal::Uri(config.
↪GetMsgReplierEndpoint()));

    /* Runtime DB */
    rad::DbAdapterRedis redis_db;

    /* Runtime data context */
    hellomal::DataContext data_ctx(config, redis_db);
...
```

At this point the event loop based on BOOST ASIO[12] (*io_service*) and the State Machine engine related objects (*external_events*, *state_machine_ctx*, and *state_machine*) are created.

The *state_machine_ctx* is passed by the State Machine engine to any invoked actions.

The *external_events* is the event queue processed by the State Machine engine.

The *state_machine* is the RAD facade to the State Machine engine and it is used to load the State Machine model, register event/state listeners, and inject events.

Before loading the State Machine model, the actions and activities objects are created via the *ActionMgr* (see *actionMgr.hpp|cpp*).

```
...
    /* Create event loop */
```

(continues on next page)

[12] https://www.boost.org/doc/libs/1_72_0/doc/html/boost_asio.html

(continued from previous page)

```
    boost::asio::io_service io_service;

    /* State Machine related objects */
    scxml4cpp::EventQueue external_events;
    scxml4cpp::Context state_machine_ctx;
    rad::SMAdapter state_machine(io_service,
                                 &state_machine_ctx,
                                 external_events);

    // Actions and activities
    hellomal::ActionMgr action_mgr;
    action_mgr.CreateActions(io_service, state_machine, data_ctx);
    action_mgr.CreateActivities(state_machine, data_ctx);

    // Load SM model
    state_machine.Load(config.GetSmScxmlFilename(), &action_mgr.GetActions(),
            &action_mgr.GetActivities());

    // Register handlers to reject events
    state_machine.RegisterDefaultRequestRejectHandler<Events::Init>();
    state_machine.RegisterDefaultRequestRejectHandler<Events::Enable>();
    state_machine.RegisterDefaultRequestRejectHandler<Events::Disable>();

    // Register publisher to write state information to Redis
    using std::placeholders::_1;
    state_machine.SetStatusPublisher(std::bind(
            &{{cookiecutter.module_name}}::DbInterface::SetControlState,
            &data_ctx.GetDbInterface(), _1));

...
```

The last part of the *main()* function is dedicated to start:

- the CII/MAL server
- the State Machine engine (*state_machine.Start()*)
- the BOOST ASIO event loop (*io_service.run()*)

Note: The event loop io_service.run() methods returns only when there are no more callbacks registered or when it is stopped via the io_service.stop() method (see ActionsStd::Exit() method).

```
...
    /* Register CII/MAL replier */
    malReplier.RegisterService<{{cookiecutter.interface_name}}::AsyncStdCmds>(
↪"StdCmds",
            std::make_shared<hellomal::StdCmdsImpl>(state_machine));
```

(continues on next page)

(continued from previous page)

```
    /* Start event loop */
    state_machine.Start();
    io_service.run();
    state_machine.Stop();
 } catch (rad::Exception& e) {
    LOG4CPLUS_ERROR(hellomal::GetLogger(), e.what());
    return EXIT_FAILURE;
 } catch (...) {
    LOG4CPLUS_ERROR(hellomal::GetLogger(), boost::current_exception_diagnostic_
↪information());
    return EXIT_FAILURE;
 }

 // to avoid valgrind warnings on potential memory loss
 google::protobuf::ShutdownProtobufLibrary();

 LOG4CPLUS_INFO(hellomal::GetLogger(), "Application hellomal terminated.");
 return EXIT_SUCCESS;
}
```

6.5 Generate CII Integration Test Module

RAD provides templates to generate some basic integration tests based on Robot Framework[13]. They verify the “standard” commands and memory leaks.

A module containing some basic integration tests to verify applications using CII Software Platform can be created by executing the following commands and entering the requested information:

```
> cd hello
> cookiecutter ../rad/rad/cpp/templates/config/rad-robtpl-maltest/

module_name [hellomaltest]: hellomaltest
module_to_test [hellomal]: hellomal
application_to_test [hellomal]: hellomal
interface_prefix [hellomalif]: hellomalif
application_to_send [hellomalifSend]: hellomalifSend
```

The input values to the template are:

- *module_name* the name of the SW module to be generated (which contains the tests).
- *module_to_test* the name of the SW module to test.
- *application_to_test* the name of application to test.
- *interface_prefix* the name of the interface module.
- *application_to_send* the name of the msgSend application to use in the tests.

[13] https://robotframework.org/

From the template Cookiecutter generates the directory *hellomaltest* containing the following files:

File	Description
hellomaltest/etr.yaml	Configuration file to be able to run the tests with ETR tool.
hellomaltest/src/genStdcmds.robot	Tests verifying the “standard” commands.
hellomaltest/src/genMemleaks.robot	Similar to genStdcmds.robot tests but executed with Valgrind tool to check for memory leaks.
hellomaltest/src/genUtilities.txt	Utility functions and configuration parameters used by the tests.

6.6 Build and Install CII Generated Modules

Generated code can be compiled and installed by executing the following commands:

```
> cd hello
> waf configure
> waf install
```

Note: Make sure that the PREFIX environment variable is set to the installation directory (which usually coincides with the INTROOT).

6.7 CII Applications Execution

In order to execute the generated application, the DB must be started first:

```
> redis-server
```

Note: It is possible to monitor the content of Redis DB via a textual client like *redis-cli* or a graphical one *dbbrowser*.

After the DB has started, the generated CII application can be executed and its current state can be queries by:

```
> hellomal -c hellomal/config.yaml -l DEBUG&

> hellomalifSend zpb.rr://127.0.0.1:12081/StdCmds GetState ""
```

The default application command line options are as follow:

```
-h [ --help ] Print help messages
-n [ --proc-name ] arg Process name
-l [ --log-level ] arg Log level: ERROR, WARNING, STATE, EVENT, ACTION, INFO,␣
↪DEBUG, TRACE
-c [ --config ] arg Configuration filename
-d [ --db-host ] arg In-memory DB host (ipaddr:port)
```

Note:

- Make sure that the *CFGPATH* environment variable contains the path(s) where the configuration files are located.
- Redis IP address and port number must be either the default one (127.0.0.1:6379), or specified as command line parameter with the option -d, or defined in the DB_HOST environment variable, or defined in the application configuration file.

To terminate the application it is enough to send an Exit command or press Ctrl-C:

```
> hellomalifSend zpb.rr://127.0.0.1:12081/StdCmds Exit ""
```

6.8 Execute CII Integration Tests

Integration tests can be executed via Extensible Test Runner (ETR) tool (see ETR User Manual[14]) or directly using Robot Framework.

In the first case:

```
> cd hellomaltest
> etr
```

Note: ETR may not be part of the ELT DevEnv and therefore it has be installed separately.

Using Robot directly:

```
> cd hellomaltest/src
> robot *.robot
```

[14] https://www.eso.org/~eltmgr/ICS/documents/ETR/sphinx_doc/html/index.html

7 Tutorial 2: Customizing an Application with RAD + CII

This tutorial explains how to customize an application created in *Tutorial 1: Creating an Application with RAD + CII*. It shows how to add a custom command with associated actions, an activity, and run-time data to be shared between actions and activities.

The resulting application is similar to the *exmalserver* example that can be found in rad/rad/cpp/_examples directory.

7.1 Add a Command

As example, we introduce a new Preset command that should emulate the pointing of a telescope.

In order to add a new command to the application the following files have to be updated/created:

- **update** hellomalif/src/hellomalif.xml (CII Interface Module)
- **update** hellomalifsend/src/main.cpp (CII msgSend Module)
- **update** hellomal/src/events.rad.ev (CII Application Module)
- **update** hellomal/src/stdCmdsImpl.hpp (CII Application Module)
- **update** hellomal/resource/config/sm.xml (CII Application Module)
- **create** hellomal/src/actionsPreset.hpp|cpp (CII Application Module)
- **update** hellomal/src/actionMgr.cpp (CII Application Module)

7.1.1 Update CII Interface Module

If a command has to be added (modified, or removed), the file *hellomalif/src/hellomalif.xml* has to be edited.

For example, in order to introduce a Preset command that takes 2 parameters (e.g. ra and dec), a new data structure (*TelPosition*) and the new interface method can be added to the CII interface specification:

```
...
<struct name="TelPosition">
  <member name="ra" type="float" />
  <member name="dec" type="float" />
</struct>
...
<interface name="StdCmds">
  ...
  <method name="Preset" returnType="string">
    <argument name="pos" type="nonBasic" nonBasicTypeName="TelPosition" />
  </method>
```

(continues on next page)

(continued from previous page)

```
  ...
</interface>
...
```

7.1.2 Update CII msgSend Module

The msgSend tool has to be updated to introduce the possibility of sending the new *Preset* command. In the file *hellomalifsend/src/main.cpp* the following code can be added:

```
...
#include <boost/tokenizer.hpp>
...
} else if (command == "Preset") {
    // default values
    double ra = 0.0;
    double dec = 0.0;
    boost::char_separator<char> separator(" ");
    boost::tokenizer<boost::char_separator<char>> tokens(params, separator);
    auto param_it = tokens.begin();
    if (param_it != tokens.end()) {
        ra = std::stof(std::string(*param_it));
        param_it++;
    }
    if (param_it != tokens.end()) {
        dec = std::stof(std::string(*param_it));
    }
    auto mal = client->getMal();
    auto p = mal->createDataEntity<::hellomalif::TelPosition>();
    p->setRa(ra);
    p->setDec(dec);
    auto reply = client->Preset(p);
    std::cout << reply << std::endl;
} ...
```

7.1.3 Update CII Application Module

7.1.3.1 Update events.rad.ev

In the application we need to define a new Preset event associated to the new command in *hellomal/src/events.rad.ev* file:

```
events:
     ...
     Preset:
             doc: event triggered when the Preset command is received.
```

(continues on next page)

(continued from previous page)

```
            payload: rad::cii::Request<std::string, std::shared_ptr
↪<hellomalif::TelPosition>>
    ...
```

7.1.3.2 Update stdCmdsImpl.hpp

The MAL interface implementation must be updated with the introduction of a new method *Preset()* which creates the corresponding event and inject it into the State Machine engine.

```
virtual elt::mal::future<std::string> Preset(const std::shared_ptr
↪<hellomalif::TelPosition>& pos) override {
    RAD_TRACE(GetLogger());
    auto ev = std::make_shared<Events::Preset>(pos->clone());
    m_sm.PostEvent(ev);
    return ev->GetPayload().GetReplyFuture();
}
```

7.1.3.3 Update sm.xml

The State Machine model can be updated by adding a new Presetting state. This state indicates that a preset command is being executed. Since Preset involves moving the telescope axes, the Presetting state is added as substate of *Operational* and is can be reached once the system has been initialized (e.g. from *On/Operation/idle*). The resulting State Machine model looks like:

```
...
<state id="Operational">
  <initial>
    <transition target="Idle"/>
  </initial>

  <state id="Idle">
    <transition event="Events.Preset" target="Presetting"/>
  </state>

  <state id="Presetting">
    <onentry>
      <customActionDomain:ActionsPreset.Start name="ActionsPreset.Start"/>
    </onentry>
  </state>
</state>
...
```

Note that when entering the state *Presetting*, the new action *ActionsPreset.Start* will be executed. This action should be responsible for initiating the preset of the telescope.

7.1.3.4 Create actionsPreset.hpp|cpp

Instead of implementing the Preset action by adding a new method to the existing *ActionsStd* class, we introduce a new class: *ActionsPreset*. To create the class, simply copy actionsStd.hpp and actionsStd.cpp files, rename them, and remove the superfluous code. The header *hellomal/src/actionsPreset.hpp* should look like:

```
#include <rad/actionGroup.hpp>
#include <rad/smAdapter.hpp>
#include <events.rad.hpp>

namespace hellomal {

class DataContext;

class ActionsPreset : public rad::ActionGroup {
 public:
    ActionsPreset(rad::SMAdapter& sm,
                  DataContext& data);

    void Start(scxml4cpp::Context* c);

    ActionsPreset(const ActionsPreset&) = delete;          //! Disable copy␣
↪constructor
    ActionsPreset& operator=(const ActionsPreset&) = delete;  //! Disable␣
↪assignment operator

 private:
    rad::SMAdapter&           m_sm;
    DataContext&              m_data;
};
}  // namespace hellomal
```

The source file *hellomal/src/actionsPreset.cpp* implementing the action should look like:

```
#include "actionsPreset.hpp"
#include "dataContext.hpp"
#include "logger.hpp"
#include <rad/mal/request.hpp>
#include <rad/getPayload.hpp>

namespace hellomal {

ActionsPreset::ActionsPreset(rad::SMAdapter& sm,
                             DataContext& data)
                : rad::ActionGroup("ActionsPreset"),
                  m_sm(sm),
                  m_data(data) {
}
```

(continues on next page)

(continued from previous page)

```
void ActionsPreset::Start(scxml4cpp::Context* c) {
    auto req = rad::GetLastEventPayloadNothrow< Events::Preset > (c);
    if (req == nullptr) {
        LOG4CPLUS_ERROR(GetLogger(), "Preset event has no associated request!");
        return;
    }
    auto reqParams = req->GetRequestPayload();
    float ra = reqParams->getRa();
    float dec = reqParams->getDec();
    LOG4CPLUS_DEBUG(GetLogger(), "Received Preset to RA " << ra << " DEC " <<␣
↪dec);
    req->SetReplyValue("Preset Started");
}

}  // namespace hellomal
```

The action implementation prints the RA/DEC and replies back to the originator of the Preset command.

7.1.3.5 Update actionMgr.cpp

Once the action has been implemented it can be added to the *ActionMgr* class so that it is created at start-up. The method *CreateActions()* in *hellomal/src/actionMgr.cpp* can be updated as follows:

```
#include "actionsPreset.hpp"
...
void ActionMgr::CreateActions(boost::asio::io_service& ios,
                              rad::SMAdapter& sm,
                              DataContext& the_data) {
...
    ActionsPreset* actions_preset = new ActionsPreset(sm, the_data);
    if (actions_preset == nullptr) {
        LOG4CPLUS_ERROR(GetLogger(), "Cannot create ActionsPreset object.");
        return;
    }
    AddActionGroup(actions_preset);
...
    the_action = new rad::ActionCallback(
                    "ActionsPreset.Start",
                    std::bind(&ActionsPreset::Start, actions_preset, _1));
    AddAction(the_action);
...
}
```

7.2 Add an Activity

After having added the Preset command (see *Add a Command*), we introduce the an activity that is started after entering the *Presetting* state and simulate the moving telescope axis. The activity takes some time (long lasting task) and when it is completed, it triggers an internal event, *MoveDone*, to go back to the Idle state. This behavior can be achieved by updating/creating the following files:

- **update** hellomal/src/events.rad.ev (CII Application Module)
- **update** hellomal/resource/config/sm.xml (CII Application Module)
- **create** hellomal/src/activityMoving.hpp|cpp (CII Application Module)
- **update** hellomal/src/actionMgr.cpp (CII Application Module)

7.2.1 Update CII Application Module

7.2.1.1 Update events.rad.ev

TheMoveDone event (without payload) is added in *hellomal/src/events.rad.ev* file to indicate that the activity has terminated:

```
events:
     ...
     MoveDone:
             doc: event triggered when the ActivityMoving has terminated.
     ...
```

7.2.1.2 Update sm.xml

The State Machine model is updated with the invocation of the activity *ActivityMoving* and the new transition from *Presetting* to *Idle* on event *MoveDone*:

```
<state id="Operational">
  <initial>
    <transition target="Idle"/>
  </initial>

  <state id="Idle">
    <transition event="Events.Preset" target="Presetting"/>
  </state>

  <state id="Presetting">
    <onentry>
      <customActionDomain:ActionsPreset.Start name="ActionsPreset.Start"/>
    </onentry>
```

(continues on next page)

(continued from previous page)

```
    <invoke id="ActivityMoving"/>
  </state>
  <transition event="Events.MoveDone" target="Idle"/>
</state>
...
```

7.2.1.3 Create activityMoving.hpp|cpp

The activity that simulates the telescope axes movement can be implemented using a dedicated thread. The thread is implemented by the *Run()* method of the *ActivityMoving* class.

```
#include "logger.hpp"
#include <rad/activity.hpp>
#include <rad/smAdapter.hpp>
#include <string>

namespace hellomal {

class DataContext;

class ActivityMoving : public rad::ThreadActivity {
 public:
    ActivityMoving(const std::string& id,
                   rad::SMAdapter& sm,
                   DataContext& data);
    virtual ~ActivityMoving();

    void Run() override;

    ActivityMoving(const ActivityMoving&) = delete;          //! Disable copy␣
↪constructor
    ActivityMoving& operator=(const ActivityMoving&) = delete;  //! Disable␣
↪assignment operator

 private:
    log4cplus::Logger m_logger = log4cplus::Logger::getInstance(LOGGER_NAME + ".
↪ActivityMoving");
    rad::SMAdapter& m_sm;
    DataContext& m_data;
};
}  // namespace hellomal
```

The *Run()* method waits 10s and then trigger the *MoveDone* event.

```
#include "activityMoving.hpp"
#include "dataContext.hpp"
#include <events.rad.hpp>
```

(continues on next page)

(continued from previous page)

```
namespace hellomal {

ActivityMoving::ActivityMoving(const std::string& id,
                               rad::SMAdapter& sm,
                               DataContext& data)
                : rad::ThreadActivity(id),
                  m_sm(sm),
                  m_data(data) {
}

ActivityMoving::~ActivityMoving() {
}

void ActivityMoving::Run() {
    int i = 0;
    const int maxIterations = 10;
    while (IsStopRequested() == false) {
        LOG4CPLUS_DEBUG(m_logger, "Moving ALT/AZ ...");
        using namespace std::chrono;
        std::this_thread::sleep_for(1s);
        if (i == maxIterations) {
            LOG4CPLUS_INFO(m_logger, "Target position reached.");
            m_sm.PostEvent(rad::UniqueEvent(new Events::MoveDone()));
            break;
        }
        i++;
    }
}
}  // namespace hellomal
```

7.2.1.4 Update actionMgr.cpp

In order to have the activity created at application start-up, the *ActionMgr::CreateActivities()* method has to be updated as follows:

```
#include "activityMoving.hpp"
...
void ActionMgr::CreateActivities(rad::SMAdapter& sm, DataContext& the_data) {
    rad::ThreadActivity* the_activity = nullptr;
    the_activity = new ActivityMoving("ActivityMoving", sm, the_data);
    AddActivity(the_activity);
}
```

7.3 Add Data Attributes

The telescope axes movement can be made more realistic by logging the intermediate telescope positions. We need therefore a way to inform the *ActivityMoving* about the target RA/DEC. This can be achieved by sharing the RA/DEC via the *DataContext* class adding the *SetRaDec()* and *GetRaDec()* methods. The *DataContext::SetRaDec()* method can be used by the *ActionsPreset::Start()* to store the target position while *ActivityMoving::Run()* uses the *DataContext::GetRaDec()* to compute the current position.

To make debugging easier, the target RA/DEC are also written in the DB and therefore the class *DbInterface* needs also to be updated.

This behavior can be achieved by updating the following files:

- **update** hellomal/src/dbInterface.hpp|cpp (CII Application Module)
- **update** hellomal/src/dataContext.hpp|cpp (CII Application Module)
- **update** hellomal/src/actionsPreset.cpp (CII Application Module)
- **update** hellomal/src/activityMoving.cpp (CII Application Module)

7.3.1 Update CII Application Module

7.3.1.1 Update dbInterface.hpp|cpp

The *DbInterface* class is updated with a method to write in the DB the RA and DEC attributes and the associated keys.

```
const std::string KEY_CONTROL_RA = "ctr.ra";
const std::string KEY_CONTROL_DEC = "ctr.dec";

class DbInterface {
 public:
 ...
 void SetRaDec(const float ra, const float dec) {
    std::vector < std::string > kvs;
    kvs.push_back(m_prefix + KEY_CONTROL_RA);
    kvs.push_back(std::to_string(ra));
    kvs.push_back(m_prefix + KEY_CONTROL_DEC);
    kvs.push_back(std::to_string(dec));
    m_runtime_db.MultiSet(kvs);
 }
 ...
}
```

7.3.1.2 Update dataContext.hpp|cpp

Two member attributes, *mRa* and *mDec*, and the associated getter and setter methods are added to the *DataContext* class. The setter method is also for writing the new target position to the DB.

```
class DataContext {
 public:
    ...
    void GetTargetRaDec(float& ra, float& dec) {
        ra = m_ra;
        dec = m_dec;
    }

    void SetTargetRaDec(const float ra, const float dec) {
        m_ra = ra;
        m_dec = dec;
        m_db_interface.SetRaDec(ra, dec);
    }

 private:
    ...
    float m_ra = 0.0;
    float m_dec = 0.0;
};
```

7.3.1.3 Update actionsPreset.cpp

The *ActionsPreset::Start()* method is updated with the writing into the *DataContext* of the pointing target coordinates.

```
void ActionsPreset::Start(const rad::AnyEvent& last_event) {
    auto req = rad::getPayloadNothrow< Events::Preset > (last_event);
    if (req == nullptr) {
        LOG4CPLUS_ERROR(GetLogger(), "Preset event has no associated request!");
        return;
    }
    auto reqParams = req->GetRequestPayload();
    float ra = reqParams->getRa();
    float dec = reqParams->getDec();
    LOG4CPLUS_DEBUG(GetLogger(), "Received Preset to RA " << ra << " DEC " <<␣
↪dec);

    m_data.SetTargetRaDec(ra, dec);

    req->SetReplyValue("Preset Started");
}
```

7.3.1.4 Update activityMoving.cpp

Finally the *AcitivityMoving::Run()* method can be re-factored to take into account the real target coordinates.

```
void ActivityMoving::Run() {
    float target_ra = 0.0;
    float target_dec = 0.0;
    m_data.GetTargetRaDec(target_ra, target_dec);

    int i = 0;
    const int max_iterations = 10;

    float cur_ra = 0.0;
    float cur_dec = 0.0;
    float step_ra = target_ra / max_iterations;
    float step_dec = target_dec / max_iterations;

    while (IsStopRequested() == false) {
        cur_ra = i * step_ra;
        cur_dec = i * step_dec;
        LOG4CPLUS_DEBUG(m_logger, "Moving ALT/AZ: RA = " << cur_ra << " DEC = " <
↪< cur_dec);
        using namespace std::chrono;
        std::this_thread::sleep_for(1s);
        if (i == max_iterations) {
            LOG4CPLUS_INFO(m_logger, "Reached target position RA = " << target_ra
                           << " DEC = " << target_dec);
            m_sm.PostEvent(rad::UniqueEvent(new Events::MoveDone()));
            break;
        }
        i++;
    }
}
```

8 Tutorial 3: Creating an Application with RAD + Prototype (obsolete)

The steps to build an application with RAD and the Prototype Software Platform are identical to the ones defined in *Tutorial 1: Creating an Application with RAD + CII*. The differences lie:

- on the name of the templates (they do not have *mal* postfix),
- the way the application interface is specified (see *Generate Prototype Interface Module*),
- the RAD classes used by the application, in particular the ones related to the middleware services.

Warning: RAD still provides the templates to create application using the Prototype Software Platform however these templates will be declared obsolete as soon as the complete CII Software Platform is delivered and integrated in RAD.

8.1 Generate Prototype WAF Project

Similar to *Generate CII WAF Project* but using rad/rad/cpp/templates/config/rad-waftpl-prj template:

```
> cookiecutter rad/rad/cpp/templates/config/rad-waftpl-prj

project_name [hello]: hello
modules_name [helloif helloifsend hello]:
```

The input values to the template are:

- *project_name* the name of the WAF project which is used to create the directory containing the project SW modules.
- *modules_name* the name of the SW modules part of this project.

8.2 Generate Prototype Interface Module

All commands, replies, and topics used to communicate between ELT applications, must be specified in dedicated interface modules. The Prototype Software Platform uses Google Protocol Buffers[15] to specify the data structures exchanged by the application via request/reply (parameters) and pub/sub (topics).

An interface module containing the “standard” commands can be created by executing the following commands and entering the requested information:

```
> cd hello
> cookiecutter ../rad/rad/cpp/templates/config/rad-cpptpl-applif
```

(continues on next page)

[15] https://developers.google.com/protocol-buffers

(continued from previous page)

```
module_name [helloif]: helloif
library_name [helloif]: helloif
package_name [examples]: hello
```

The input values to the template are:

- *module_name* the name of the SW module to be generated (which contains the interface specification).
- *library_name* the name of the binary to be produced when compiling the generated module.
- *package_name* the name of the directory that contains the module. In this case it is the project directory.

From the template Cookiecutter generates the directory *helloif* containing the following files:

File	Description
helloif/wscript	WAF file to compile the SW module.
helloif/interface/helloif/requests.proto	Google ProtoBuf data structures.

The requests.proto file contains the definition data structures used to send requests and replies, for example the Init command (without parameters) and the related reply (with a string parameter):

```
syntax = "proto3"

package helloif;

message ReqInit {
}

message RepInit {
    string reply = 1;
}
```

The .proto files are complied by the protoc compiler which generates, in the build directory the following C++ files:

- *hello/build/.../helloif.pb.cpp*
- *hello/build/.../helloif.ph.h*

These files are used by the application to send/receive commands/replies. Generated files contain the C++ classes representing the data structures. These classes provide the methods to deserialize (parse) message payloads and to serialize.

The protoc compiler is invoked by waf every time you compile (and the .proto files have been modified).

8.3 Generate Prototype msgSend Module

A SW module implementing the msgSend tool to send the "standard" commands to applications based on Prototype Software Platform can be created by executing the following commands and entering the requested information:

```
> cd hello
> cookiecutter ../rad/rad/cpp/templates/config/rad-cpptpl-send/

interface_name [helloif]: helloif
interface_module [helloif]: helloif
module_name [helloifsend]: helloifsendhelloifsend
application_name [helloifSend]: helloifSend
parent_package_name [hello]: hello
```

The input values to the template are:

- *interface_name* the name of the Prototype Interface module (which specifies the commands sent by the msgSend tool and it was defined in *Generate Prototype Interface Module*).
- *interface_module* fully qualified name of Prototype Interface library.
- *module_name* the name of the SW module to be generated (which contains the msgSend tool).
- *application_name* the name of the binary to be produced when compiling the generated module.
- *parent_package_name* the name of the directory that contains the tool. In this case it is the project directory.

From the template Cookiecutter generates the directory *helloifsend* containing the following files:

File	Description
helloifsend/wscript	WAF file to compile the SW module.
helloifsend/src/main.cpp	Implementation of msgSend.

The tool can be invoked by:

```
helloifSend <timeout> <IP> <port> <command> <parameters>

<timeout>    reply timeout in msec
<IP>         IP address
<port>       port
<command>    command to be sent (e.g. helloif.ReqStatus)
<parameters> parameters of the command
```

8.4 Generate Prototype Application Module

RAD provides the templates to create a simple server application implementing the standard ELT State Machine model using the Prototype Software Platform services.

A SW module implementing implementing a server application able to process the “standard” commands using ZeroMQ and ProtoBuf services can be created by executing the following commands and entering the requested information:

```
> cd hello
> cookiecutter ../rad/rad/cpp/templates/config/rad-cpptpl-appl

module_name [hello]:
application_name [hello]:
package_name [examples]: hello
interface_name [helloif]:
libs [cpp._examples.helloif]: helloif
```

From the template Cookiecutter generates the directory *hello* containing the following files:

File	Description
wscript	WAF file to build the application.
resource/config/config.yaml	YAML application configuration file.
resource/config/sm.xml	SCXML file with the State Machine model.
resource/config/log.properties	Logging configuration file.
src/events.rad.ev	List of events processed by the application.
src/actionMgr.[hpp|cpp]	Class responsible for instantiating actions and activities/
src/actionsStd.[hpp|cpp]	Class implementing standard action methods.
src/config.[hpp|cpp]	Class loading YAML configuration file.
src/dataContext.[hpp|cpp]	Class used to store application run-time data shared between action classes.
src/dbInterface.[hpp|cpp]	Class interfacing to the in-memory DB.
src/logger.[hpp|cpp]	Default logger definition.
src/msgParsers.[hpp|cpp]	Classes parsing the ZeroMQ commands/topics.
src/main.cpp	Application entry function.

The generated application is very similar to the application generated for CII Software Platform (see *Generate CII Application Module*). Instead of getting the commands via the realization of the CII/MAL interface (*stdCmdsImpl.hpp*) the “naked” ZMQ messages are parsed by the *MsgParsers* and *TopicParsers* classes defined in *msgParsers.[hpp|cpp]* and injected into the State Machine Engine in form of events..

8.5 Generate Prototype Integration Test Module

A module containing some basic integration tests to verify applications using Prototype Software Platform can be created by executing the following commands and entering the requested information:

```
> cd hello
> cookiecutter rad/rad/cpp/templates/config/rad-robtpl-test/

module_name [hellotest]: hellotest
module_to_test [hello]: hello
application_to_test [hello]: hello
interface_prefix [helloif]: helloif
application_to_send [helloifSend]: helloifSend
```

The input values to the template are:

- *module_name* the name of the SW module to be generated (which contains the tests).
- *module_to_test* the name of the SW module to test.
- *application_to_test* the name of application to test.
- *interface_prefix* the name of the interface module.
- *application_to_send* the name of the msgSend application to use in the tests.

From the template Cookiecutter generates the directory *hellotest* containing the following files:

File	Description
hellotest/etr.yaml	Configuration file to be able to run the tests with ETR tool.
hellotest/src/genStdcmds.robot	Tests verifying the “standard” commands.
hellotest/src/genMemleaks.robot	Similar to genStdcmds.robot tests but executed with Valgrind tool to check for memory leaks.
hellotest/src/genUtilities.txt	Utility functions and configuration parameters used by the tests.

8.6 Build and Install Generated Prototype Modules

Generated code can be compiled and installed by executing the following commands:

```
> cd hello
> waf configure
> waf install
```

Note: Make sure that the PREFIX environment variable is set to the installation directory (which usually coincides with the INTROOT).

8.7 Prototype Applications Execution

In order to execute the generated application, the DB must be started first:

```
> redis-server
```

Note: It is possible to monitor the content of Redis DB via a textual client like *redis-cli* or a graphical one *dbbrowser*.

After the DB has started, the generated CII application can be executed and its current state can be queried by:

```
> hello -c hello/config.yaml -l DEBUG&

> helloifSend 5000 127.0.0.1 5588 helloif.ReqStatus ""
```

The default application command line options are as follow:

```
-h [ --help ] Print help messages
-n [ --proc-name ] arg Process name
-l [ --log-level ] arg Log level: ERROR, WARNING, STATE, EVENT, ACTION, INFO,␣
↪DEBUG, TRACE
-c [ --config ] arg Configuration filename
-d [ --db-host ] arg In-memory DB host (ipaddr:port)
```

Note:

- Make sure that the *CFGPATH* environment variable contains the path(s) where the configuration files are located.
- Redis IP address and port number must be either the default one (127.0.0.1:6379), or specified as command line parameter with the option -d, or defined in the DB_HOST environment variable, or defined in the application configuration file.

To terminate the application it is enough to send an Exit command or press Ctrl-C:

```
> helloifSend 5000 127.0.0.1 5588 helloif.ReqExit ""
```

8.8 Execute Prototype Integration Tests

Integration tests can be executed via Extensible Test Runner (ETR) tool (see ETR User Manual[16]) or directly using Robot Framework.

In the first case:

```
> cd hellotest
> etr
```

Note: ETR may not be part of the ELT DevEnv and therefore it has be installed separately.

Using Robot directly:

```
> cd hellotest/src
> robot *.robot
```

8.9 Adding New Command

In order to add a new command to the application the following steps have to be performed:

- Add the request and related reply in the interface module (e.g. helloif/interface/helloif/Requests.proto file)
- Add the event corresponding to the request in the event definition file (e.g. hello/src/Events.rad.ev file)
- Update the SCXML model with the transition dealing with the new request (e.g. hello/config/hello/sm.yaml)
- Add a new method in the ActionsStd class or add a new actions class
- Update the ActionsMgr class with the registration of the new action

[16] https://www.eso.org/~eltmgr/ICS/documents/ETR/sphinx_doc/html/index.html

9 Examples

This section contains some example applications created using RAD.

Examples are located in *rad/rad/cpp/_examples/* directory.

9.1 Example Using Prototype Software Platform

9.1.1 exif

This is an example of interface module with the definition of the commands, replies, topics used by the example applications.

It contains requests.proto for the definition of the requests/replies and topics.proto for the definition of the pub/sub topics.

9.1.2 exsend

This is an example of how to build a utility application (similar to the VLT msgSend) able to send requests and receive replies defined in exif interface module. This application is using some RAD libraries but it is not generated from the RAD templates.

The exsend module implements the exSend application that can be used as follow:

```
exSend <timeout> <IP> <port> <command> <parameters>

where:

   <timeout> reply timeout in msec
   <IP> IP address
   <port> port <command> command to be sent to the server (e.g. exif.ReqInit)
   <parameters> parameters of the command
```

for example to query the status (with 5 sec timeout) of an application running on the same local host on port 5577:

```
exSend 5000 127.0.0.1 5577 exif.ReqStatus ""
```

9.1.3 server

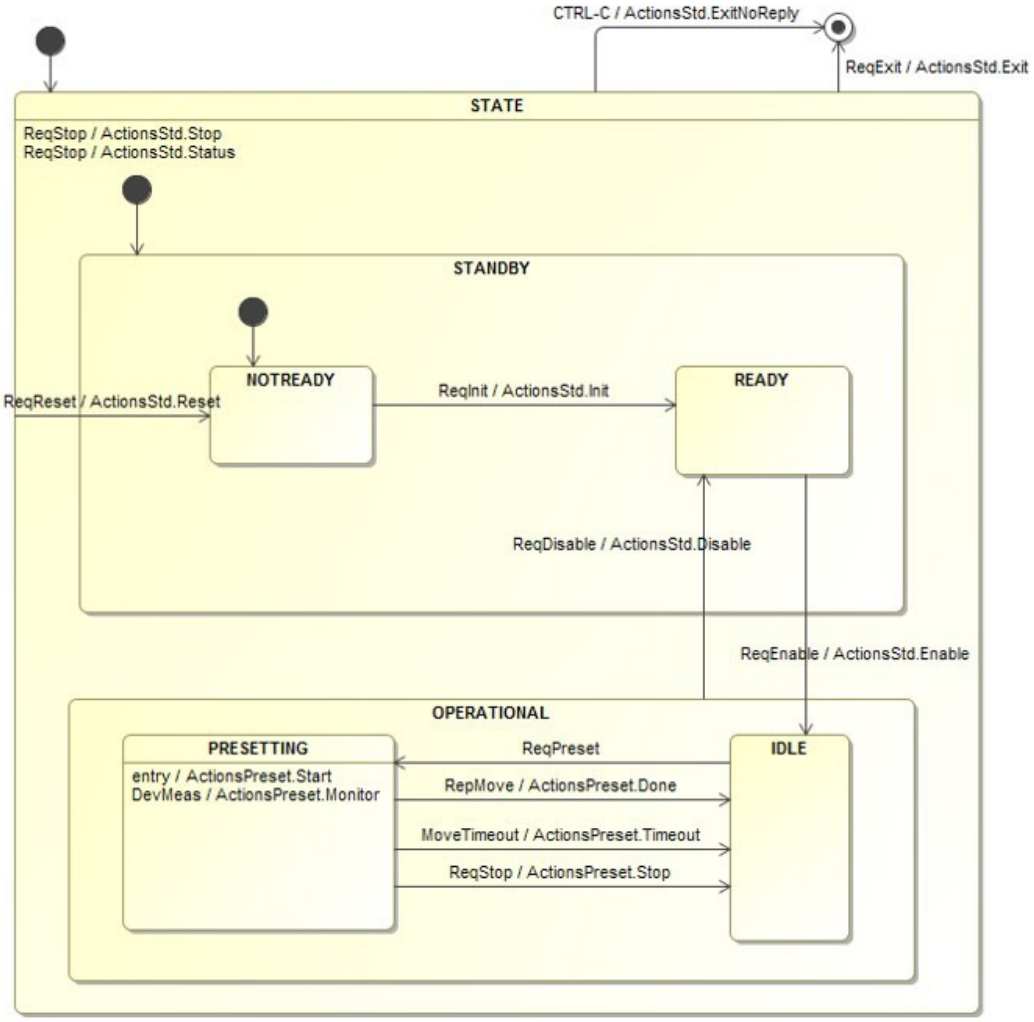
This is an example that shows how to create RAD based applications that uses request/reply, pub/sub, timers, Linux signals. It uses the interface defined in exif interface module and can be controlled by sending the commands via the exSend application (see exsend module).

The server module has two possible configuration files and state machines:

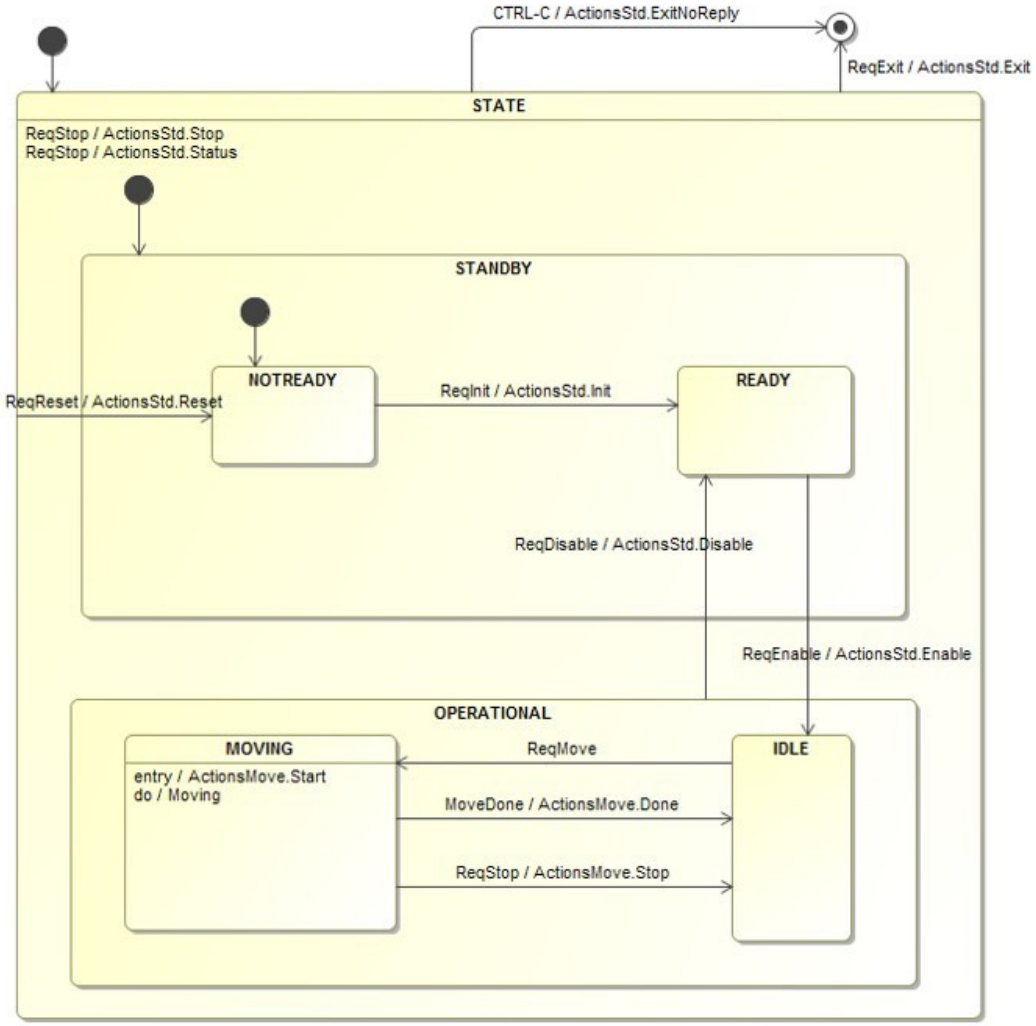
server/config/radServer/config.yaml server/config/radServer/sm.xml

server/config/radServer/config1.yaml server/config/radServer/sm1.xml

The first configuration (config.yaml and sm.xnl) is used to instantiate a prsControl application that is able to process exif.ReqPreset commands. When a exif.ReqPreset command is received, the application executes the ActionPreset::Start action which sends a exif.ReqMove to a second application (altazControl) that simulate the movement of the axes of a telescope. While waiting for the completion of the preset, it monitors the axes position by subscribing to topic XYMeas topic published on port 5560. The topic is processed by the XYMeas topic is processed by the ActionPreset::Monitor action. See the picture below for a more complete overview of the behaviour of prsControl application.

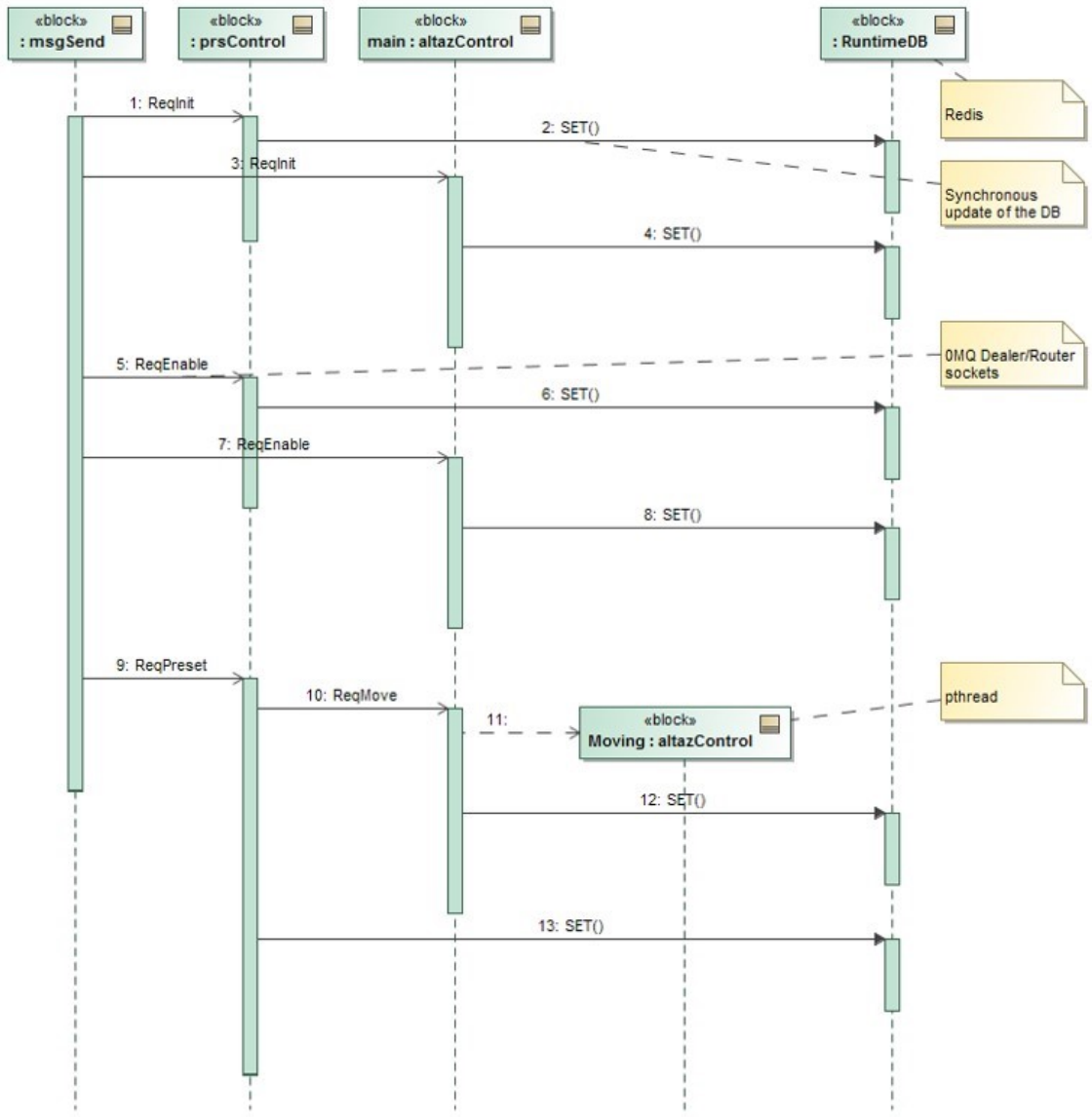


The second application (altazControl) is configured using config1.yaml and sm1.xml files. It receives the exif.ReqMove command and executes the ActionsMove::Start action and starts a do-activity: the ActivityMoving thread. The thread simulates the movement of the axes and publishes the intermediate positions via the XYMeas topic. When the target position is reached, the do-activity terminates and a reply (exif.RepMove) to the originator of the exif.ReqMove command is sent by the ActionsMove::Done action. See the picture below for a more complete overview of the behaviour of prsControl application. See the picture below for a more complete overview of the behaviour of altazControl application.

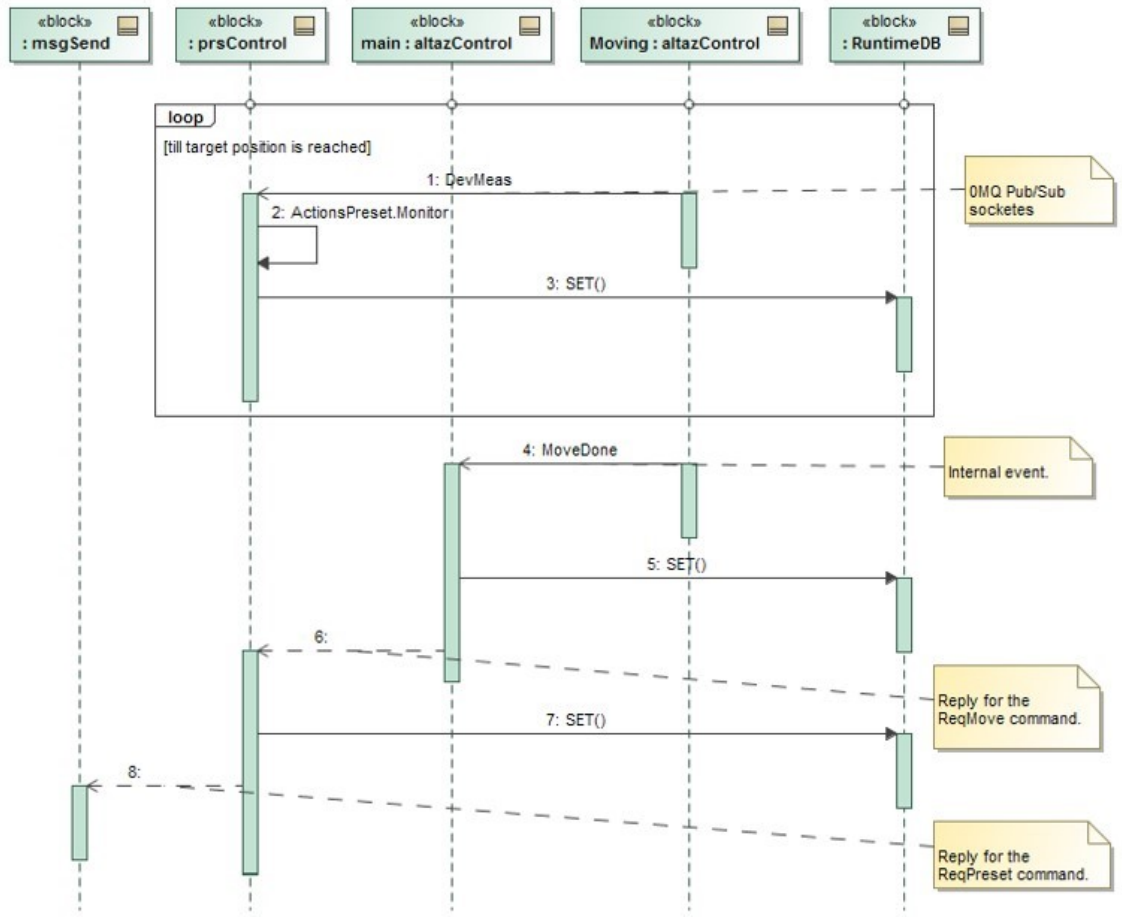


Note that both applications store the configuration, status, and telescope position information in the Redis runtime DB.

The sequence of messages to initialize, enable, and start the preset is shown below.



The sequence of messages to preset the axes is shown below.



In order to run the server example refer to the RAD integration test section.

9.1.4 hellorad + server

This is an example that shows how a Python client can talk to a C++ server. The client sends a ReqTest request containing “Ping pong” text, the server receives the requests and replies with a RepTest reply. To run the example first start the server:

```
radServer -c radServer/config.yaml -l DEBUG &
```

and the start the client:

```
hellorad client --req-endpoint='tcp://localhost:5577'
```

9.2 Example Using CII Software Platform

9.2.1 exmalif

This is the porting of exif interface definition using CII MAL XML language.

9.2.2 exmalsend

This tool is similar to exsend and allows to send the commands specified in exmalif to a CII server (see exmalsever below) implementing the exmalif interface.

It shows how to send synchronous CII/MAL/ZPB requests and get the related replies.

9.2.3 exmalserver

This is the porting of the server example application to CII.

It shows how to implement a server that is able to:

- reply asynchronously to commands including error replies and partial replies.
- send commands synchronously and get replies asynchronously.
- publish topics and subscribe to topics.

10 COMODO

10.1 Tool

COMODO is a model-to-text transformation toolkit based on Xpand/Xtend that takes as input a UML/SysML model and transforms it into different artifacts depending on the selected target platform.

The toolkit is made of:

- A Java application to transform models: comodo.jar
- A UML profile called comodoProfile containing stereotypes used to identify what has to be transformed.

With COMODO it is possible for example to generate the SCXML document from a UML/SysML State Machine model created with MagicDraw tool.

10.1.1 Syntax

COMODO can be executed as follow:

```
java -jar comodo.jar {options}

-c,--config <arg>     Configuration parameters for the platform

-d,--debug            Debug information

-e,--modules <e>      Specify the module(s) to generate

-g,--mode <arg>       Generation mode [all|normal|update]

-h,--help             Print help for this application

-m,--model <arg>      Model file path

-o,--output <arg>     Output folder path

-p,--profile <arg>    Path to comodoProfile

-t,--platform <arg>   Specify the target software platform [SCXML|
                      JAVA|VLT|ACS|JPFSC|RMQ|ELT|PLC]
```

The input model (-m, –model) must be in the EMF UML XMI format (.uml) and it should comply with the COMODO Profile (comododProfile).

Currently, the supported target platforms are:

- SCXML: transform the input model into SCXML document.
- VLTSW: transform the input model into C++ application for the Very Large Telescope SW Platform.

- ACS: transform the input model into Java application for the ALMA Common SW platform.
- RMQ: transform the input model into Java application using RabbitMQ middleware.
- JPF: transform the input model (limited to State Machines) into Java application that can be verified by Java Pathfinder model checker.

10.1.2 Example

The following example generates an SCXML document from a UML State Machine diagram. The input parameters are:

- 'mymodel.uml' the input model.
- 'comodoProfile.profile.uml' the COMODO Profile.
- 'outputDirectory' directory where to store the generated artifacts.
- 'mymodule' the UML package (marked with <<cmdoModule>> stereotype) on which the transformation has to be applied.
- 'SCXML' the target platform.
- 'all' mode to generate all artefacts.

```
java -jar comodo.jar -m mymodel.uml -o ./outputDirectory -p
comodoProfile.profile.uml -e mymodule -t SCXML -g all
```

10.1.3 Repository

COMODO can be retrieved from: http://svnhq9.hq.eso.org/p9/trunk/EELT/DevEnv/comodo/comodo.jar/

10.2 Profile

A model can be transformed by COMODO only if it is a valid instance of the COMODO metamodel. COMODO metamodel is defined in the COMODO profile (comodoProfile) and includes the stereotypes listed in the table below.

Table 10.1: COMODO Stereotypes

Stereotype	UML Element	Description
cmdoModule	Package	Package containing components or interfaces. It represents the basic unit of transformation and maps to a SW module.
cmdoComponent	Class	Abstract representation of a SW component. Its behavior can be described using a State Machine.
cmdoCommand	Signal	Indicates an event triggered by the arrival of a request.
cmdoInternal	Signal	Indicates an event triggered by the SW component itself.
cmdoTimer	Signal	Indicates an event triggered by a time-out.
cmdoFileio	Signal	Indicates an event triggered by FILE I/O.
cmdoIOSignal	Signal	Indicates an event triggered by Linux signal.
cmdoTopic	Signal	Indicates an event triggered by the arrival of a pub/sub topic.

For more information please refer to the documentation in the comodoProfile.mdzip project.

10.2.1 Repository

comodoProfile is located in MagicDraw Teamwork server under “Common Profiles and Libraries” section.

10.3 MagicDraw

10.3.1 Profile Configuration

COMODO profile (comodoProfile.mdzip) must be either copied in the Profile directory of your MagicDraw installation before launching MagicDraw or it can be added to the project from ESO MagicDraw Teamcloud server. Note that the second option seems to work only for projects created on Teamcloud server.

10.3.2 Start-up MagicDraw

When starting MagicDraw, two dialogs are displayed: one for the license information and one for selecting the edition and the plug-ins. In the second dialog it is enough to select the “Standard Edition” (Figure 1). No plug-ins are mandatory for COMODO since it works with both pure UML profile. If transformations have to be applied to SysML models, the SysML plug-in must be selected and loaded.

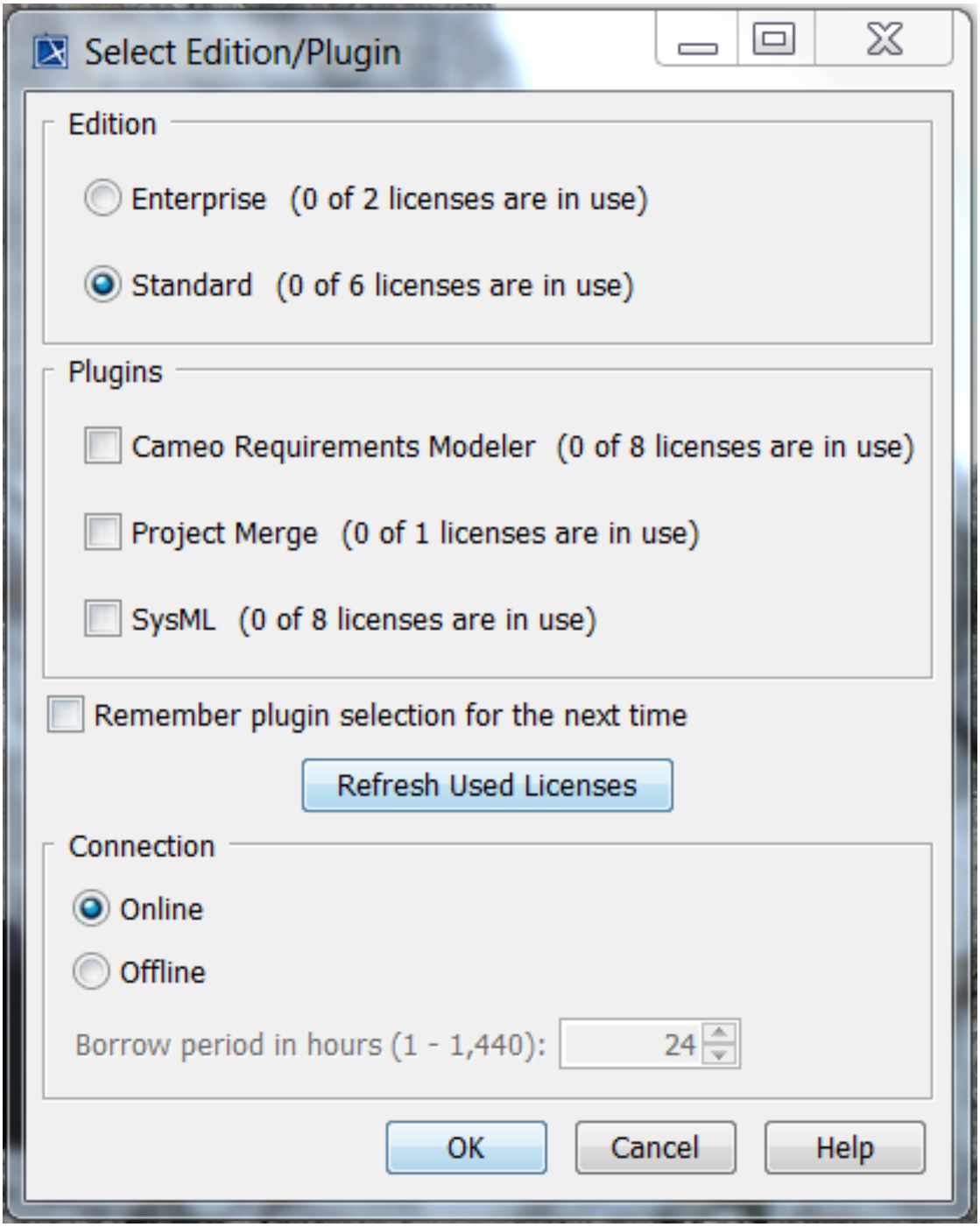


Figure 1 – Select edition and plug-ins dialog.

10.3.3 Switch to Fully Featured Perspective

In order to see all available MagicDraw/UML/SysML options, go to “Options” menu, “Perspectives” item, “Perspectives” sub-item, select the “Full Featured” option, and click on “Apply” button.

10.3.4 Creating UML Model compliant with COMODO Profile

It is possible to create a UML model compliant with COMODO Profile by executing the following steps:

- Create a MagicDraw Project
- Add comodoProfile to the Project
- Create a <<cmdoModule>> Package
- Create the Packages “Signals”, “Actions”, and “Activities” with all the events, actions, do-activities
- Create a <<cmdoComponent>> Class with associated State Machine as behavior
- Create States and Transitions for the State Machine

10.3.4.1 Creating MagicDraw Project

Use the “New” option of the “File” menu to:

- Select the type of project “UML Project”
- enter the SW module name (e.g. test) as project “Name”
- select the “Project location” (e.g. test/config/model/)

as illustrated in Figure 2.

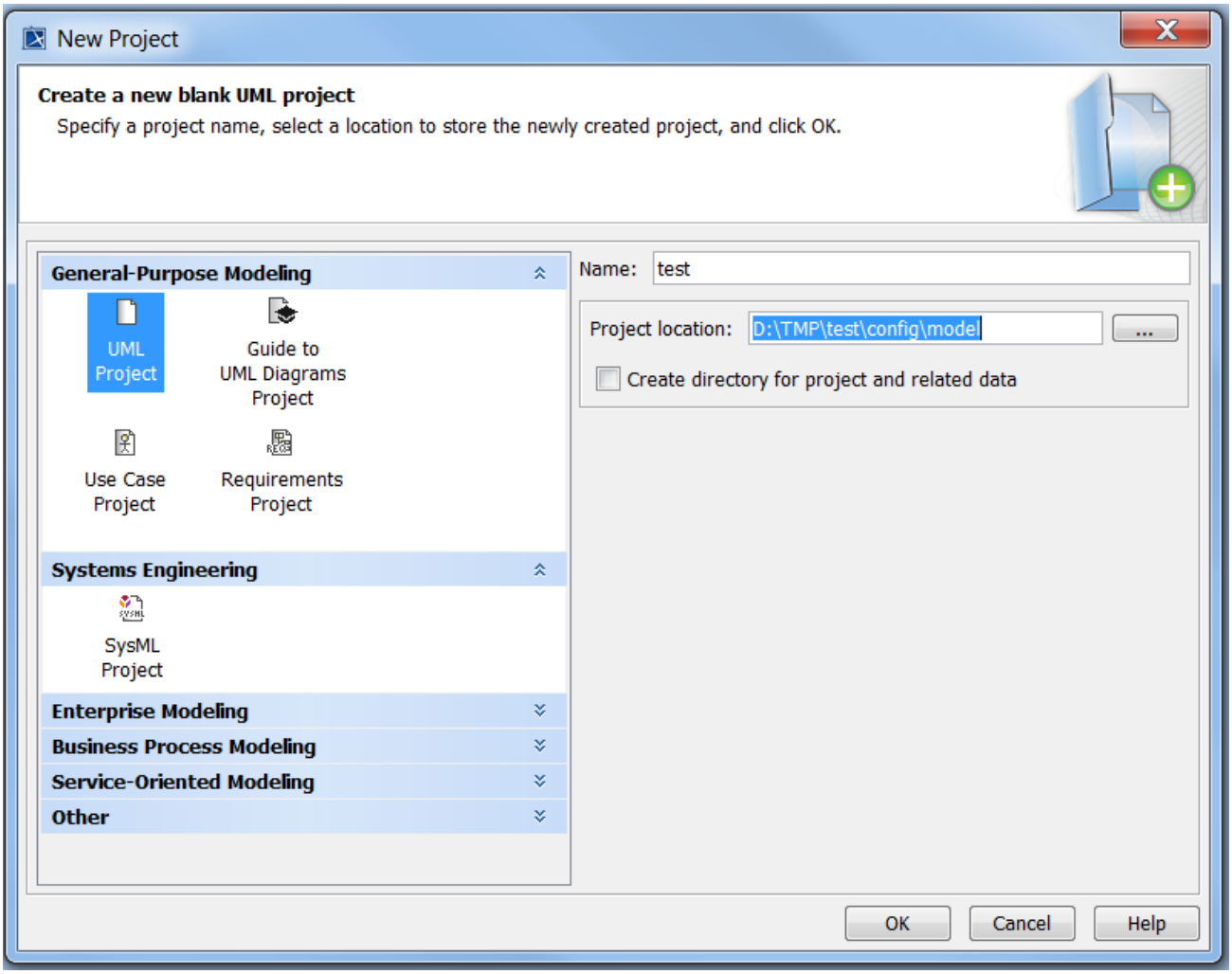
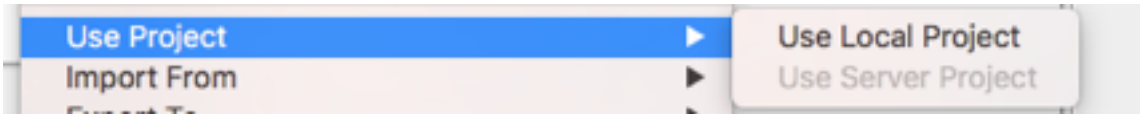


Figure 2 - Creating a new project.

10.3.4.2 Adding comodoProfile to the Project

Once the project is created, use the “Use Project . . . ” option from the “File” menu, select “Use Project” item, “Use Local Project” or “Use Server Project” depending whether COMODO profile is loaded from local file or from the server.



select comodoProfile, and click on “Finish” button to load COMODO stereotypes (Figure 3).

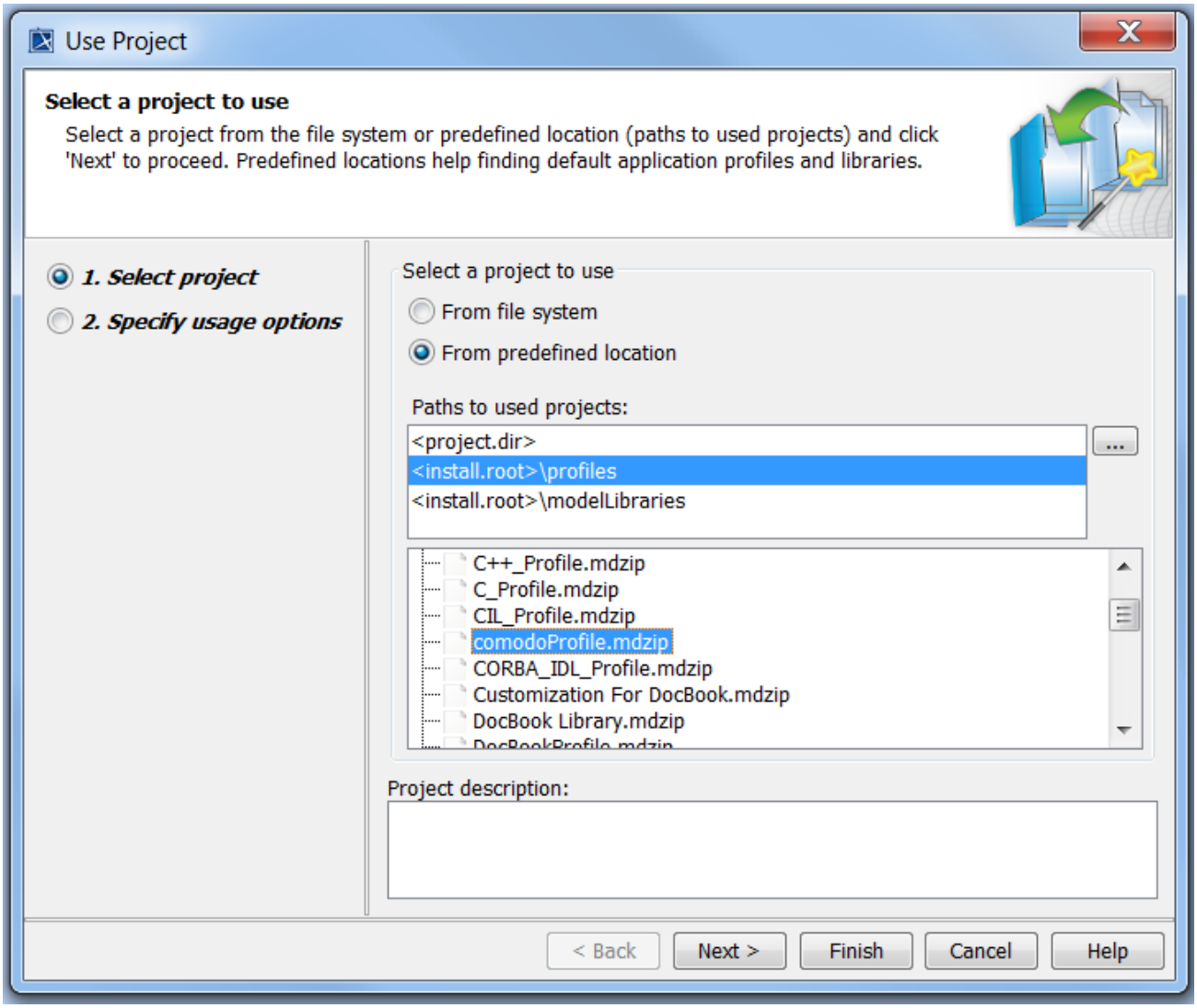
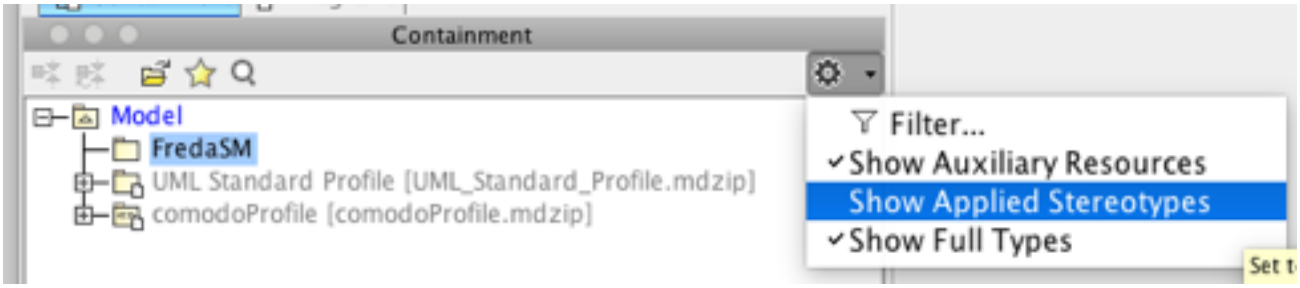


Figure 3 - Adding comodoProfile to a project.

Note that in order to see the (COMODO) stereotypes applied to your modeling elements, click on the top-right corner of the “Containement” panel and select “Show Applied Stereotypes”.



10.3.4.3 Create a <<cmdoModule>> Package

Select the “Data” package in the “Containment” tab on the left side. With the mouse-right-click navigate through “Create Element” menu and select “Package” option as illustrated in Figure 4.

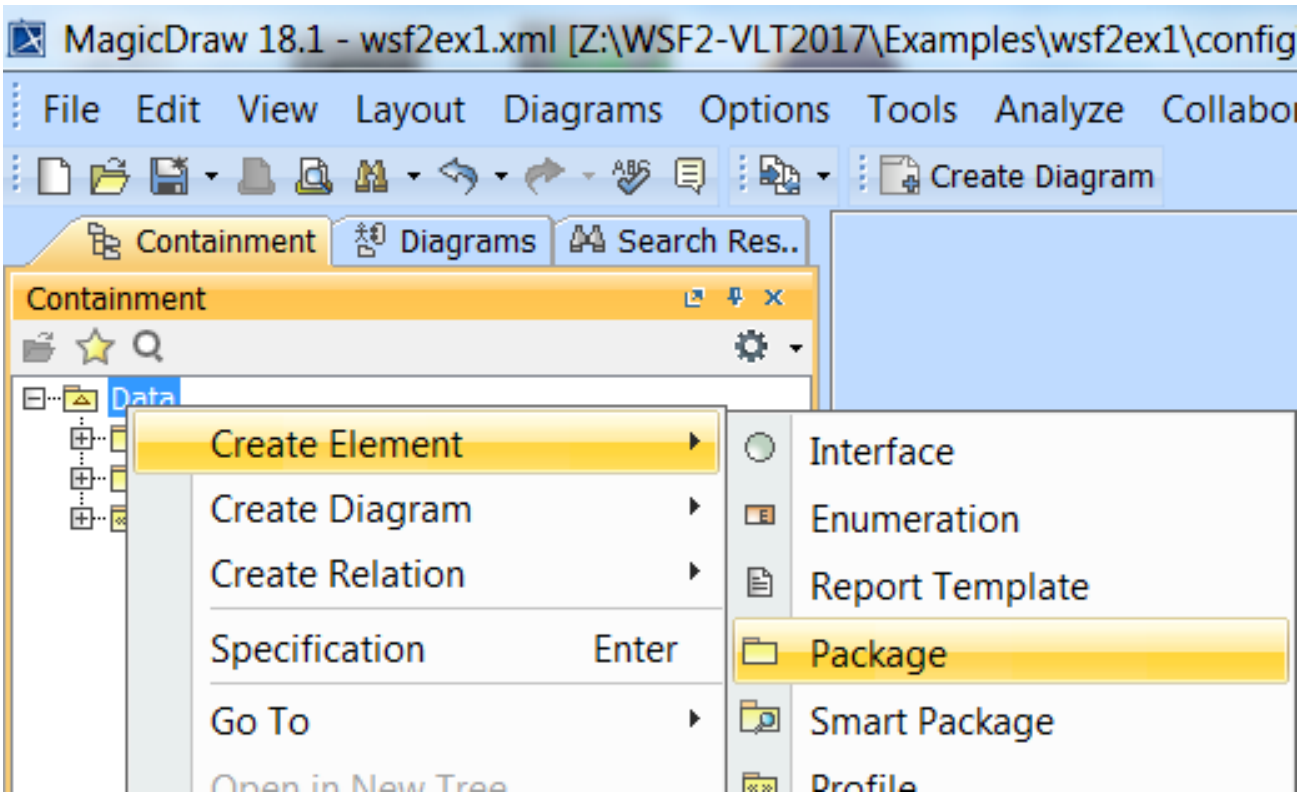


Figure 4 – Creating a Package.

Enter as Package name the name of the SW module (e.g. “test”).

Mouse-right-click on the newly created Package and select the “Stereotype” option. Select the “cmdoModule” stereotype and click on “Apply”. At the end it should look like in Figure 5.

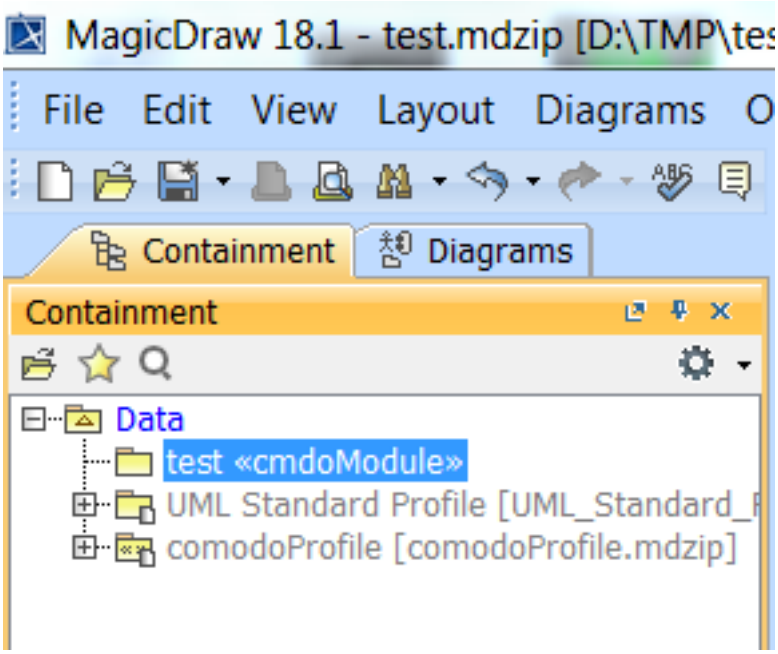


Figure 5 – Applying <<cmdoModule>> stereotype.

The package can be populated with signals, actions, activities, and the <<cmdoComponent>> class representing the SW component with associated classifier behavior.

10.3.4.4 Creating Signals

The events (such as commands, DB notifications, timers, file I/O, UNIX signals, and internal events) handled by the SW component are modeled via UML signals stereotyped by one of the COMODO stereotypes.

Signals can be grouped in a package called “Signals”. To create the package, right-click on the <<cmdoModule>> package and click on “Create Element” “Package” entering the package name “Signals”.

In order to create a signal, right-mouse click on the “Signals” package and select the option “Signal” from the menu “Create Element” as shown in Figure 6.

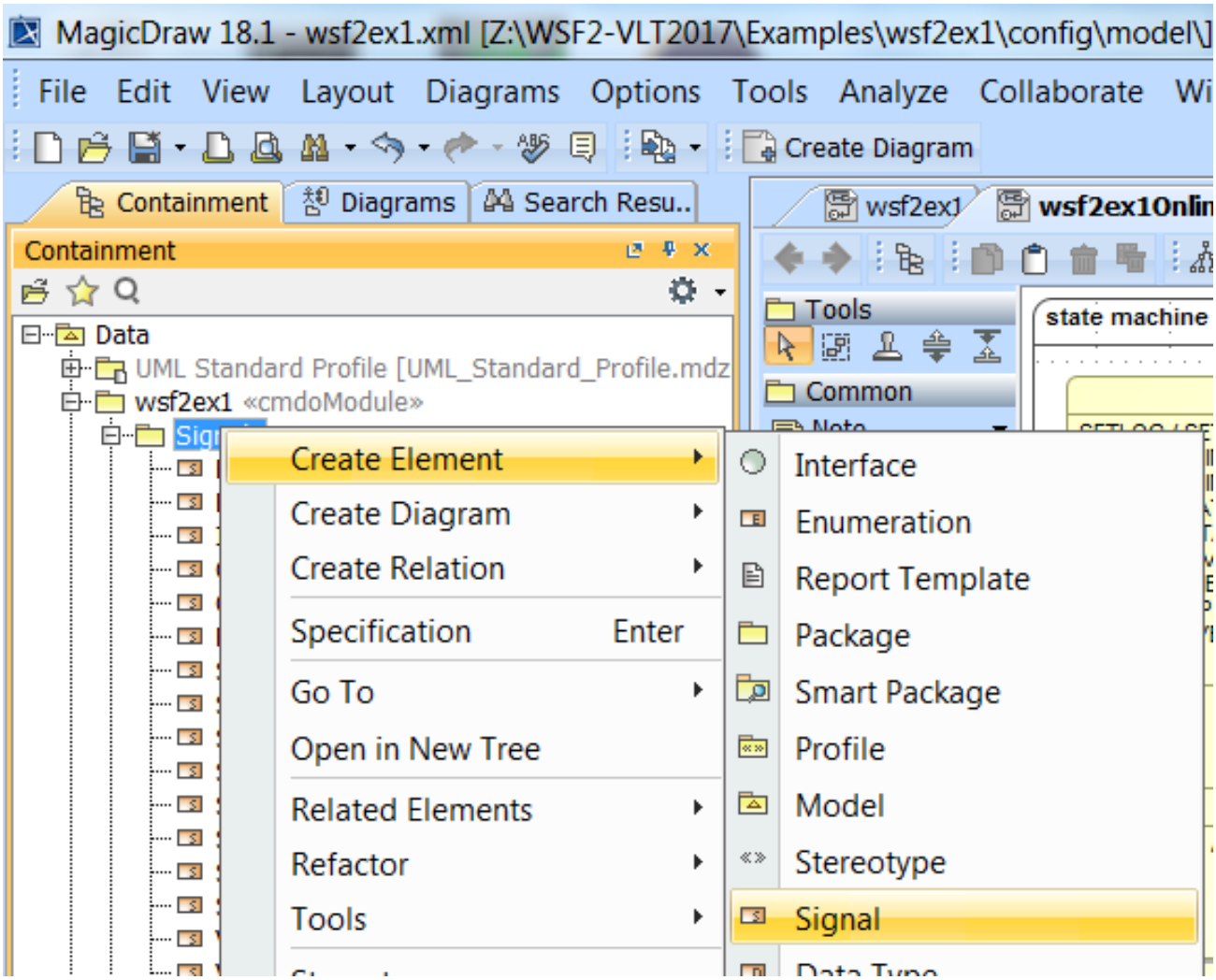


Figure 6 - Creating a new signal.

Once the signal has been created it should be named and the proper stereotype applied. The stereotype is applied by mouse-right-click on the newly created signal and by selecting the “Stereotype” option. comodoProfile offers the following stereotypes for Signals:

- <<cmdoCommand>> for events representing commands received by the application (i.e. commands defined in the CDT).
- <<cmdoInternal>> for events created by the application itself to trigger a transition.
- <<cmdoNotificaiton>> for events representing changes of DB attributes.
- <<cmdoTimer>> for events representing time-outs.
- <<cmdoSignal>> for events representing UNIX signals.
- <<cmdoFileio>> for events representing UNIX file I/O events.

Select the stereotype and click on the “Apply” button. After successful creation, the new signal should

appear in the Signals package with the correct stereotype.

It is good practice to group the signals into a dedicated package.

10.3.4.5 Creating Actions

Statecharts actions are piece of code executed when entering/exiting a state (entry/exit actions) or when a transition is taken. Statecharts actions are modeled in UML with UML Activities. To create a UML Activity follow the instructions for creating a Signal (*Creating Signals*) and select the option "Activity" instead of "Signal".

A Statecharts action is translated by COMODO into an invocation of a method of a class.

The name of the UML Activity should follow the convention: "GroupName.MethodName" where "GroupName" is the name of the class containing the method "MethodName". The method GroupName::MethodName() is invoked by the State Machine engine when executing the model.

It is good practice to group all the actions into a dedicated package named "Actions".

10.3.4.6 Creating Do-Activities

Statecharts Do-Activities are long lasting actions which are mapped to threads. In UML they are modeled with UML Activities. To create a UML Activity follow the instructions for creating a Signal (*Creating Signals*) and select the option "Activity" instead of "Signal". The name of the UML Activity is translated by COMODO to the name of the class implementing the thread.

It is recommended to group all the do-activities in a UML Package named "Activities".

10.3.4.7 Creating SW Components

A SW Component represents an application to be developed. In UML it is modeled by a UML Class with stereotype <<cmdoComponent>>. To create a Class follow the instructions for creating a Signal (*Creating Signals*) and select the option "Class" instead of "Signal".

Once the Class has been created it should be named and the <<cmdoComponent>> stereotype applied. The stereotype is applied by mouse-right-click on the newly created Class and by selecting <<cmdoComponent>> from the "Stereotype" option.

10.3.4.8 Creating State Machine

In order to specify the behavior of a SW Component using a State Machine, mouse-right-click on the SW Component Class element, select “Create Diagram” option and click on “State Machine Diagram”. A State Machine with associated diagram will be created and assigned as classifier behavior to the SW Component.

10.3.4.9 Creating State Machine Diagrams

In order to create a new State Machine diagram, mouse-right-click on the State Machine element in the Containment tree, select the menu “Create Diagram” and the “State Machine Diagram” option. Rename the newly created diagram using F2 (or opening the Specification Dialog). Drag&drop from the Containment tree in to the diagram the states which are needed and should be specialized (Figure 7).

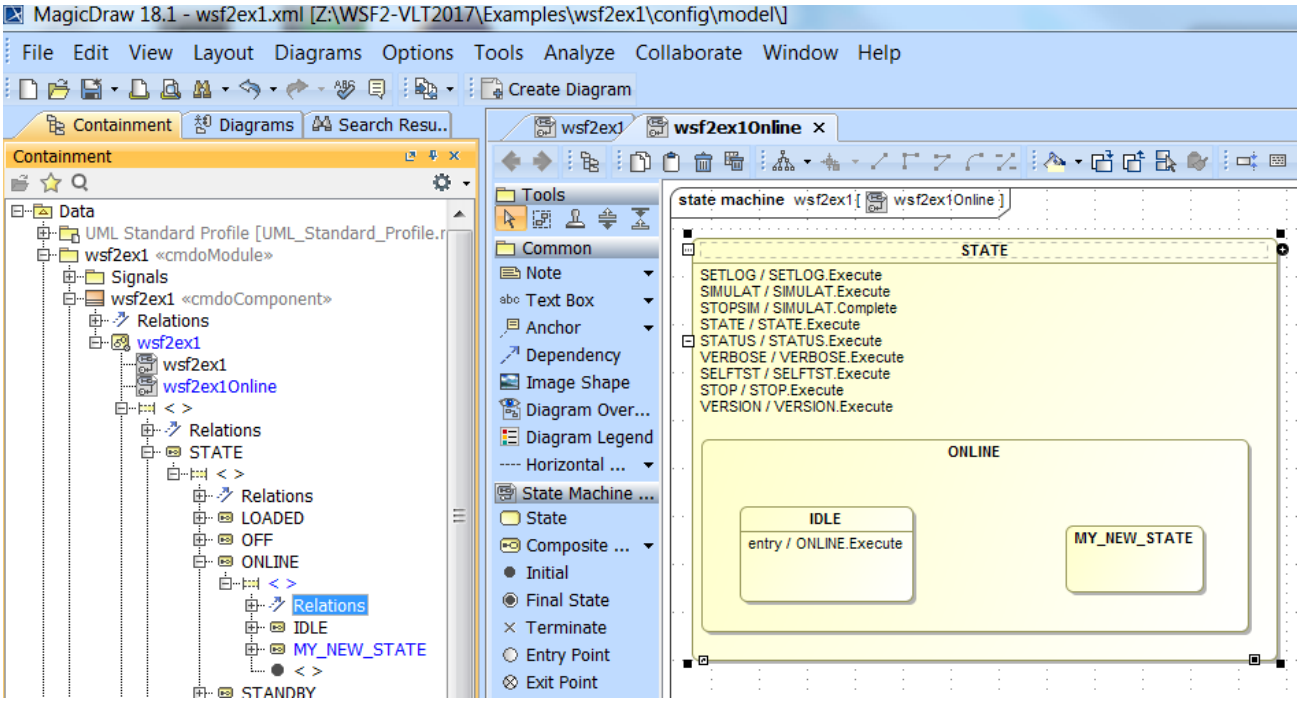


Figure 7 - New State Machine diagram and new sub-state.

10.3.4.10 Creating States

To create a new state, open the State Machine diagram and select, from the Tools, the type of state to create. Click in the diagram on the position where the state should be located. It is suggested to create Composite states (instead of leaf states) since they can be specialized.

The state must be named either by clicking on the state and typing the name or by opening the Specification Dialog and filling in the “Name” property.

Important: verify in the Containment tree whether the new state belong to the correct super-state (parent composite state). For example, in the Containment tree of Figure 7, the new state MY_NEW_STATE is a sub-state of ONLINE which in turn is a sub-state of STATE.

10.3.4.10.1 Initial Pseudo-state

Each composite state containing sub-states, must indicate the default initial active sub-state. This is done by drag&drop the “Initial” pseudo-state from Tools into the composite state and creating a transition from the “Initial” pseudo-state to the default initial sub-state.

Important: a composite state that contains sub-state must define a default initial state using the “Initial” pseudo-state.

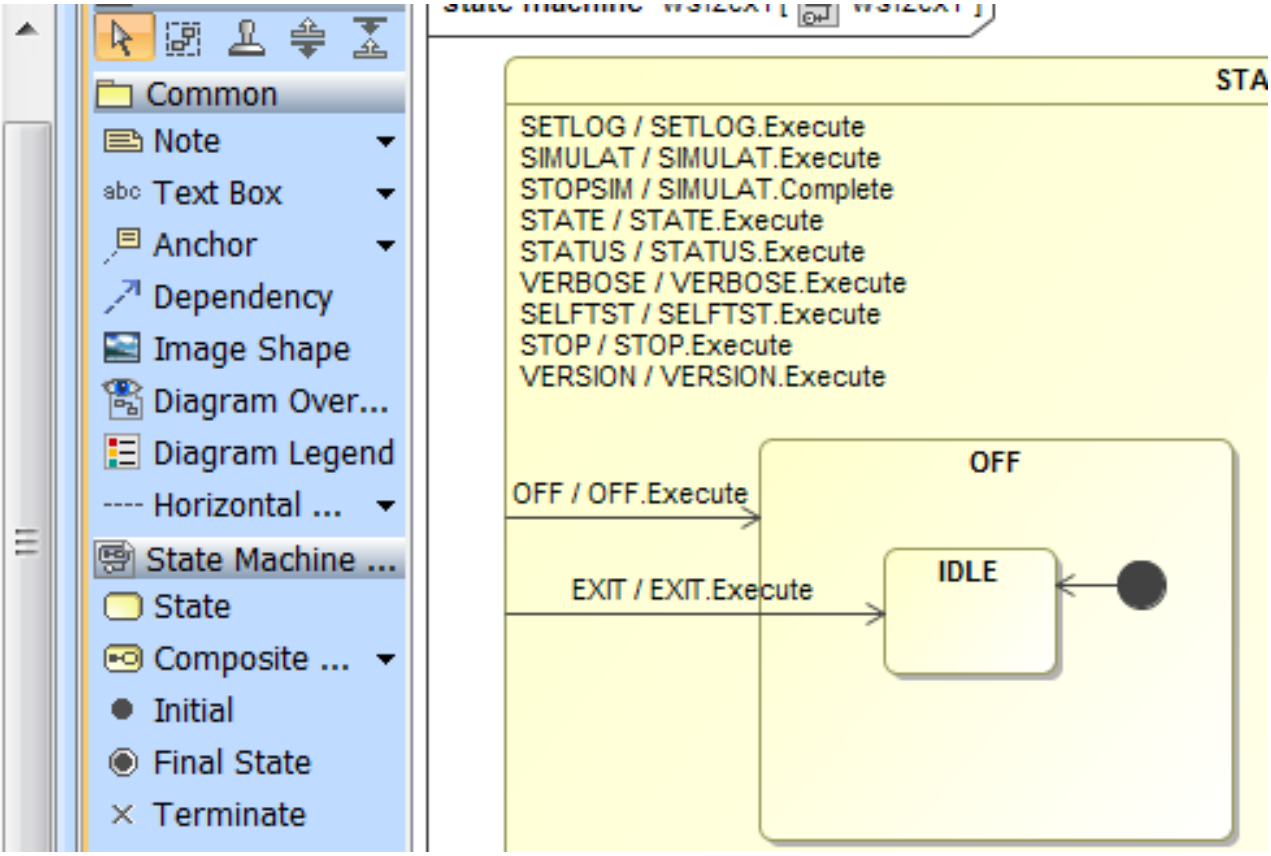


Figure 8 - Initial pseudo-state.

Figure 8 shows that the default initial state of the OFF composite state is IDLE.

10.3.4.10.2 Entry/Exit Actions

To specify an entry or exit action to be executed when a state is entered or exited simply drag the UML Activity (see section *Creating Actions*) and drop it on the state. A pop-up menu with the following three options will appear: Entry, Exit, Do activity. Select the “Entry” or the “Exit” option.

10.3.4.10.3 Do-Activities

To specify a Do-Activity to be executed while the application is in a given state, simply drag the UML Activity (see section *Creating Do-Activities*) and drop it on the state. A pop-up menu with the following three options will appear: Entry, Exit, Do activity. Select the “Do activity” option.

10.3.4.11 Creating Transitions

10.3.4.11.1 Normal Transition

Transition between two states can be create by clicking on the source state, selecting the “Transition” tool from the palette (Figure 9), and dragging the line to the destination state.

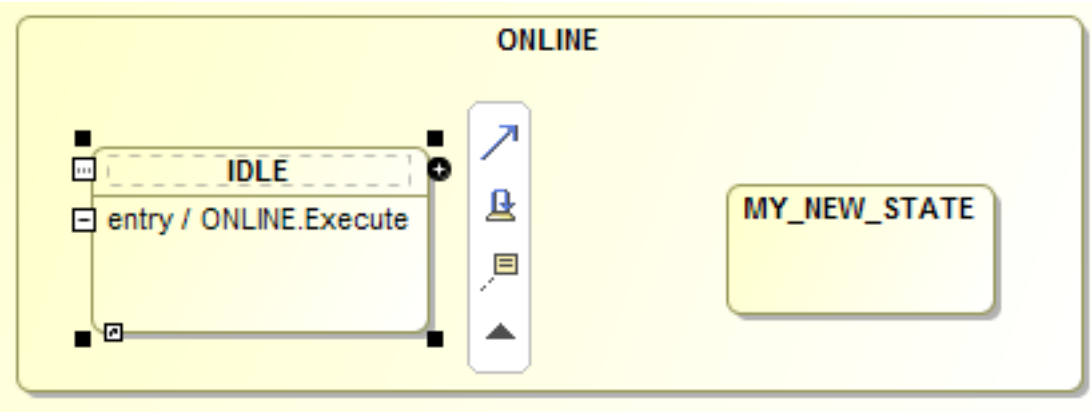


Figure 9 - Creating a transition between IDLE and MY_NEW_STATE state.

10.3.4.11.2 Self-Transitions

A self-transition is a transition where the source and destination state is the same. It can be created like a normal transition but selecting the “Self-transition” tool which is just below the “Transition” tool. Note that when taking a self-transition, the state is exited and reentered and therefore the exit/entry actions of the state, if defined, are executed.

10.3.4.11.3 Internal Transitions

An internal-transition is a transition where, like a self-transition, the state does not change. However, since in this case the state is never exited nor entered, the entry/exit actions are not executed.

To create an internal transition, open the Specification Dialog of the state and select on the left side the element “Internal Transitions” as shown in Figure 10. Click on “Create” button and enter the following properties:

- Section “Transition” property “Name”: the name of the event/signal triggering the transition.
- Section “Transition” property “Guard”: the name of the guard to be verified before executing the transition. Guards have the same syntax of actions: “ClassName.MethodName” (see section *Creating Actions*).
- Section “Trigger” property “Event Type”: select the value “SignalEvent”
- Section “Trigger” property “Signal”: select the name of the signal that should have been previously created and added in the Package “Signals” (see section *Creating Signals*).
- Section “Effect” property “Behavior Type”: select the value “Activity”
- Section “Effect” property “Name”: enter the name of the action using the syntax “ClassName.MethodName”.

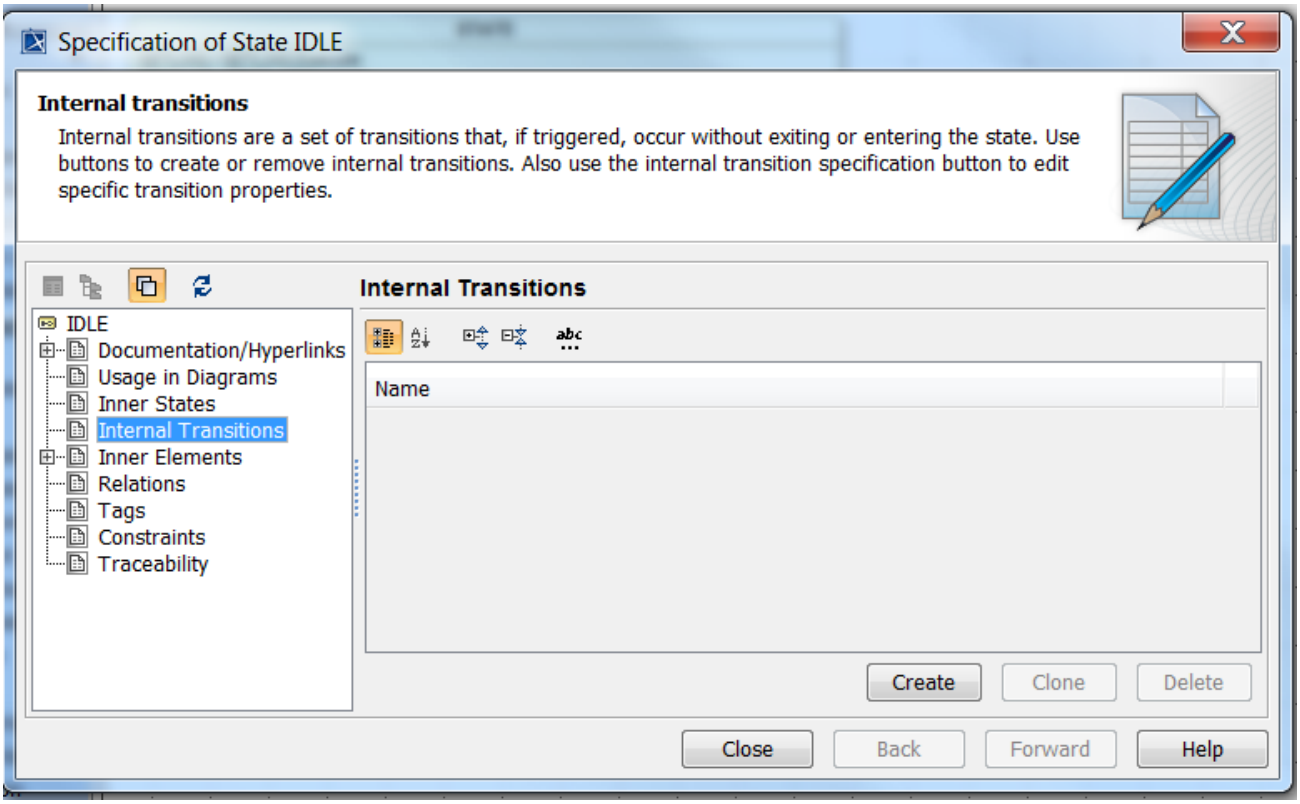


Figure 10 - Creation of internal transition for ONLINE/IDLE state.

10.3.4.11.4 Triggers

To specify the trigger (or event) of a transition simply drag from the Signal package the signal and drop it on the transition. Internal transitions are a special case, see section *Internal Transitions*.

10.3.4.11.5 Actions

To specify the action to be executed when a transition is taken simply drag from the Activity and drop it on the transition. Internal transitions are a special case, see section *Internal Transitions*.

10.3.4.11.6 Guards

To specify the guard to be evaluated before taking a transition, open the Specification dialog for the Transition and, in Section “Transition” property “Guard” enter the name of the guard. Guards name have the same syntax of actions: “ClassName.MethodName”.

10.3.4.12 Creating Orthogonal Regions

Statecharts orthogonal regions correspond to UML regions. By default, each UML state contains one UML region. To add an orthogonal region, mouse-right-click on the state and select the “Add New Region” option.

Figure 11 shows the result of adding a region to the ONLINE state of wsf2ex1 application. In the Containment tree within the ONLINE state the two unnamed regions are indicated with the symbol “< >”. Note that, in this example, the one that can be expanded is the region containing IDLE and MY_NEW_STATE sub-states, while the other one is the newly added region.

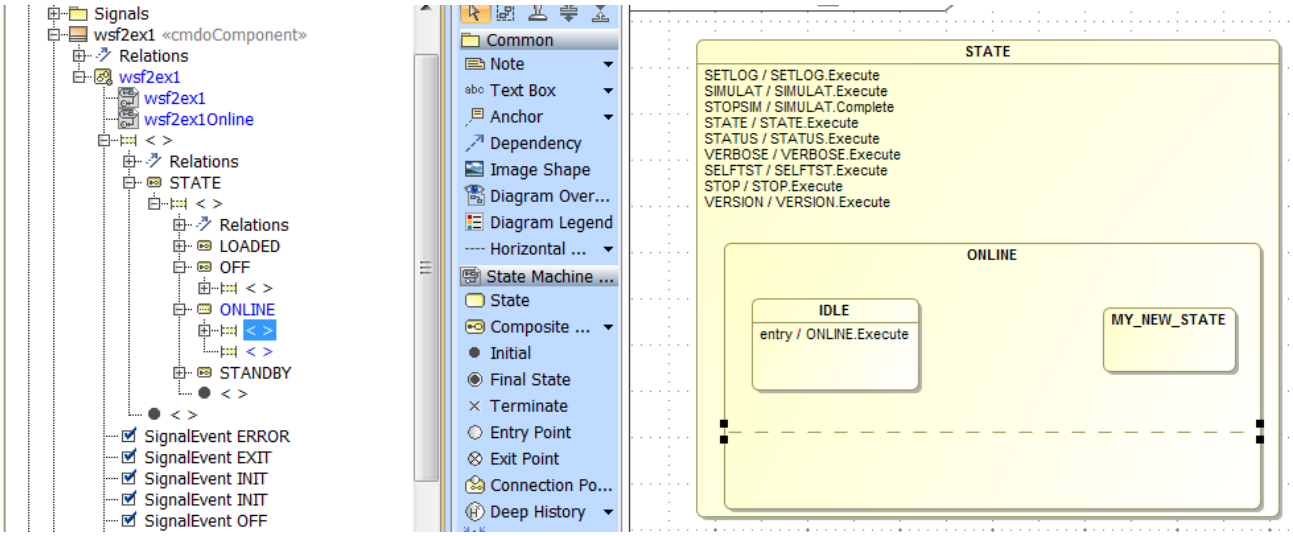


Figure 11 - Adding an orthogonal region to the ONLINE state.

Regions can be named by right-click them in the Containment tree and selecting the “Rename” option. The name of the regions can be displayed in the diagram by right-click on the state in the diagram and selecting the “Symbols Properties” option. Within the “Symbols Properties” dialog, check the box “Show Region Name” as shown in Figure 12.

Important: when two or more orthogonal regions are defined, they must always have a name.

Figure 12 shows on the left side the Containment tree with the two ONLINE regions named “Region1” and “Region2”. On the right side is the “Symbol Properties” dialog with the “Show Region Name” flag set to true. In the center is the diagram displaying the name of the regions.

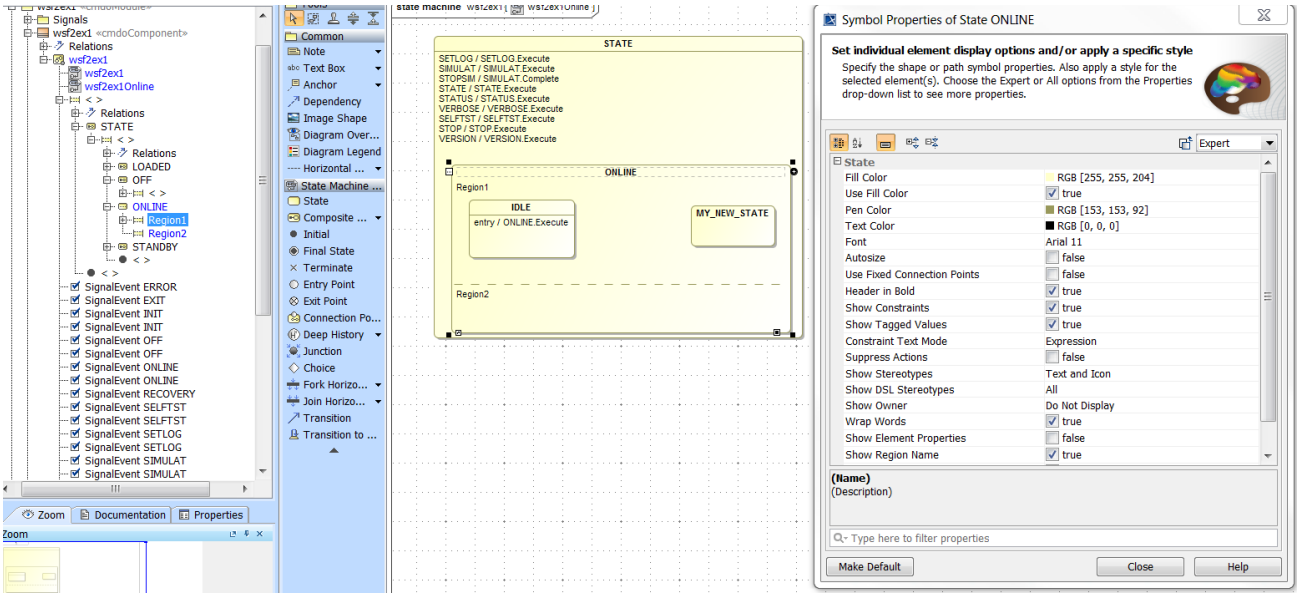


Figure 12 - Displaying the regions name of state ONLINE.

Important: events are broadcasted to all regions but they are processed sequentially one region after the other following the alphabetic order of the region name. In our example, events will be first processed in Region1 and then in Region2.

10.3.5 Loading, Saving and Exporting Models

10.3.5.1 Loading Models from File

A model can be loaded via the “Open Project . . . ” option of the “File” menu and selecting the UML model to open. When loading a model that uses COMODO profile, MagicDraw will try to open also the profile. If comodoProfile.mdzip is not located in the Profile directory of MagicDraw installation, the tool will ask the user to locate the it.

10.3.5.2 Loading Models from Teamwork Server

A model archived in Teamwork Server can be loaded by:

- Selecting the “Login” option from the “Collaborate” menu and entering the login information.
- Selecting the “Open Server Project. . . ” option from the “Collaborate” menu and clicking on the project to open.

10.3.5.3 Saving and Exporting Models

Modification to the model can be saved in the MagicDraw proprietary format (.mdzip format) using the “Save Project” option of the “File” menu. In order to use COMODO tool to generate artifacts, the model must be exported to the Eclipse UML2 XMI v.2 or above format (.uml) which is tool independent and a de-facto standard since supported by the majority of the modeling tools (e.g. all Eclipse Modeling Framework tools but also commercial tools). The option can be found under the “File” menu as illustrated in Figure 13.

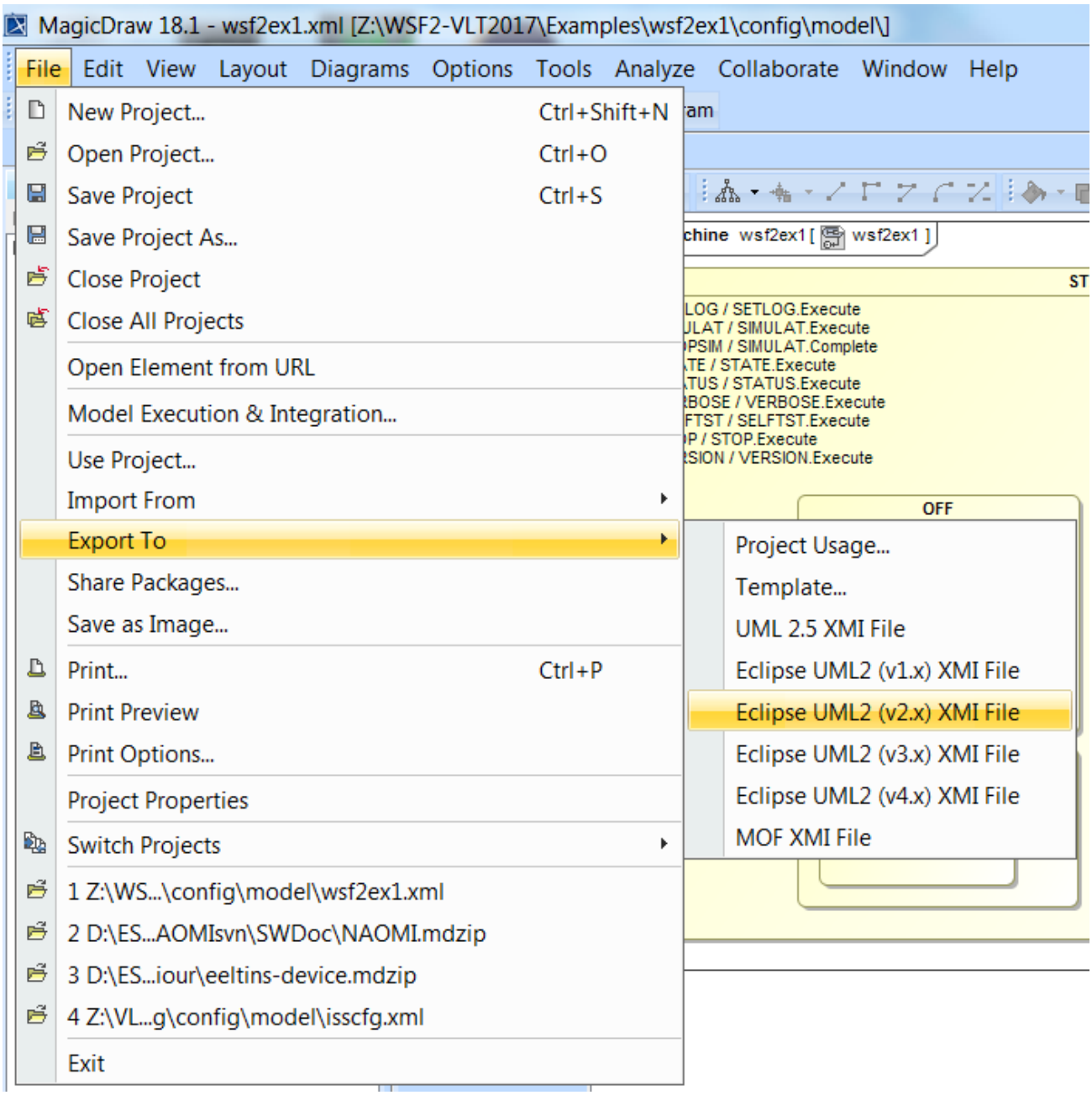


Figure 13 - Exporting to EMF XMI format.

After confirming the location of the exported model (e.g. wsf2ex1/config/model directory) and clicking on “Yes” button of the alert dialog asking permission to overwrite the existing (.uml) files, the model is converted and COMODO can be used to generate artifacts.

Important: MagicDraw format (mdzip) contains the model information and the graphical information (diagrams). Eclipse UML2 XMI (.uml) contains only model information.

Important: COMODO can be used only on SysML/UML models archived in Eclipse UML2 XMI format

(.uml).

The save and export operation can be simplified by setting the configuration option that can be found in “Options” menu, “Environment”, “Eclipse UML2 (v2.x) XMI”, “Export Model to Eclipse UML2 XMI . . . ” and selecting “Always export” value as illustrated in Figure 14. In this way, every time a model is saved, it is also exported to Eclipse UML2 XMI format.

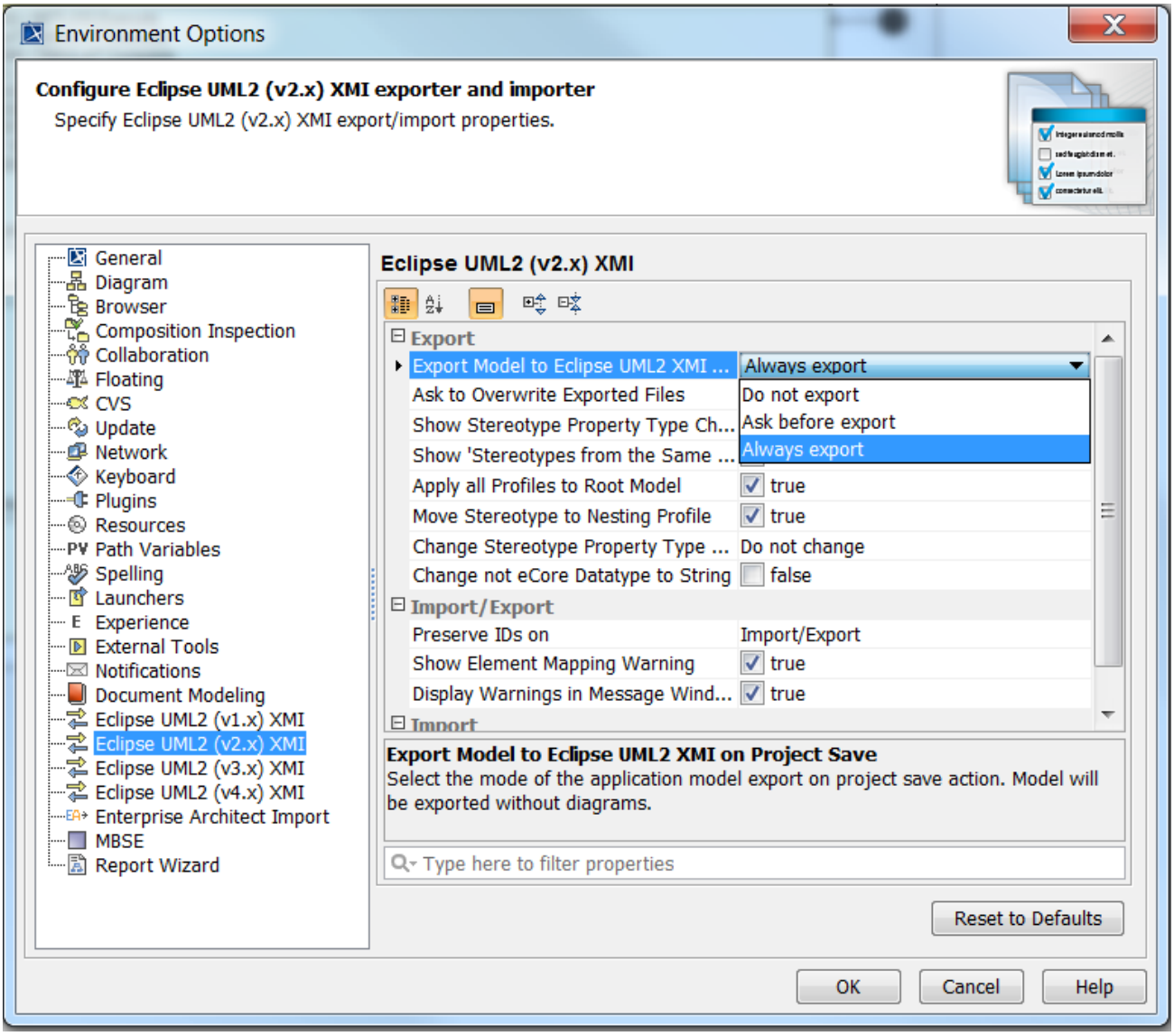


Figure 14 – Making Export to EMF XMI automatic.

10.3.6 Model-View

In MagicDraw, the screen is split into 4 main sections: left section, center section, right section and bottom section. The left section hosts the Containment tree (to navigate through all the model elements and relations), the center section hosts the Tools, the right section hosts the diagrams, while the bottom section (located below the first three sections) is used for logging purposes (info, errors, warnings, and validations information).

Important: The *model* consists of the elements and relations contained in the Containment tree. A diagram is only one possible *view* of the model.

Note that diagrams may display only some of the model elements/relations of the model.

Important: removing a model element from a diagram using the “Delete” key, does not delete the model element from the Containment tree. Instead using Ctrl-D will delete a selected element in the diagram also from the Containment tree. Another possibility is to select the element in the Containment tree and with the mouse-right-click select the “Delete” option: the element will be deleted from the model and from all the diagrams.

Figure 15 shows the Containment tree (left section), the Tools (center section), and the State Machine diagram (right section) after loading wsf2ex1/config/model/wsf2ex1.xml MagicDraw model. The Containment tree includes three Packages:

- “UML Standard Profile” containing all UML stereotypes
- “wsf2ex1” containing the model of wsf2ex1 application
- “comodoProfile” containing COMODO stereotypes

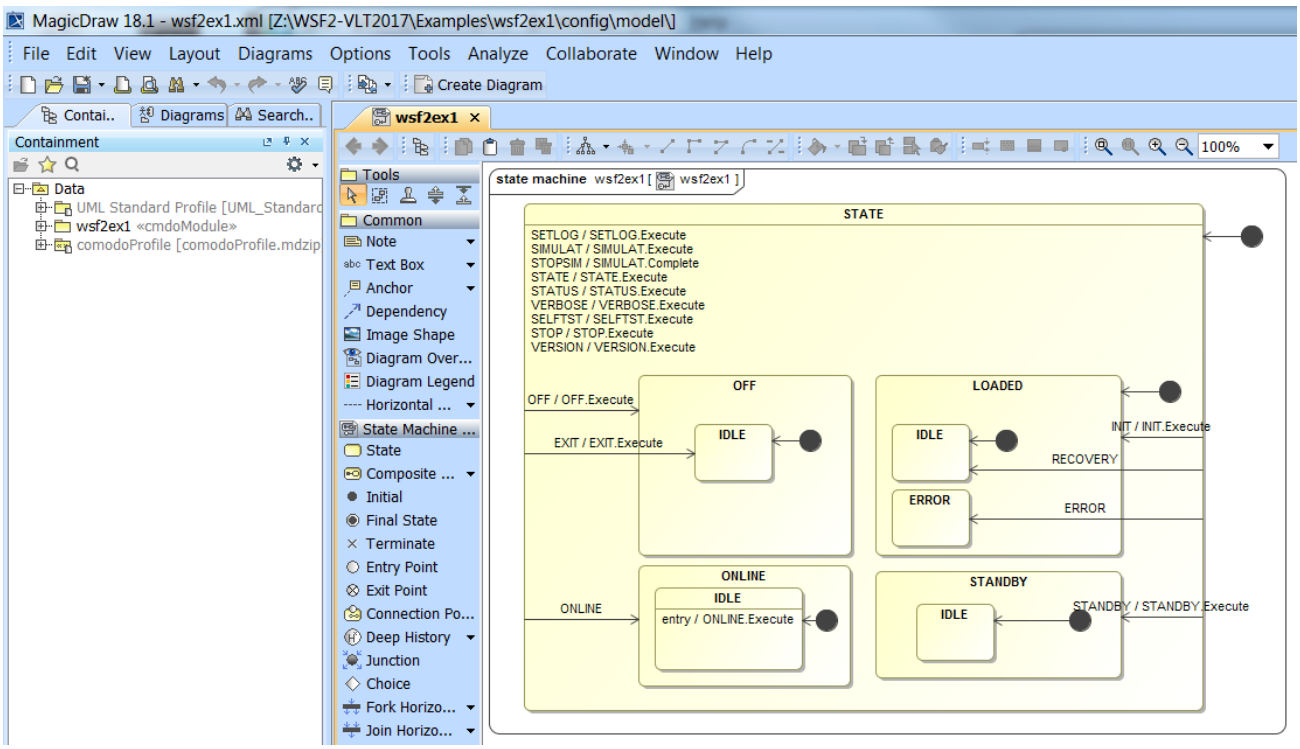


Figure 15 - Containment tree (left section), Tools (center section) and State Machine diagram (right section).

Figure 16 shows the content of the Package “wsf2ex1” in the Containment tree. Only the first two elements are used in the context of WSF2:

- The Package “Signals” used to group all signals (i.e. events) used in the State Machine to trigger a transition.
- The Class “wsf2ex1” with stereotype <<cmdoComponent>> which represents the application to be modeled. This class contains the State Machine which describes its behavior.

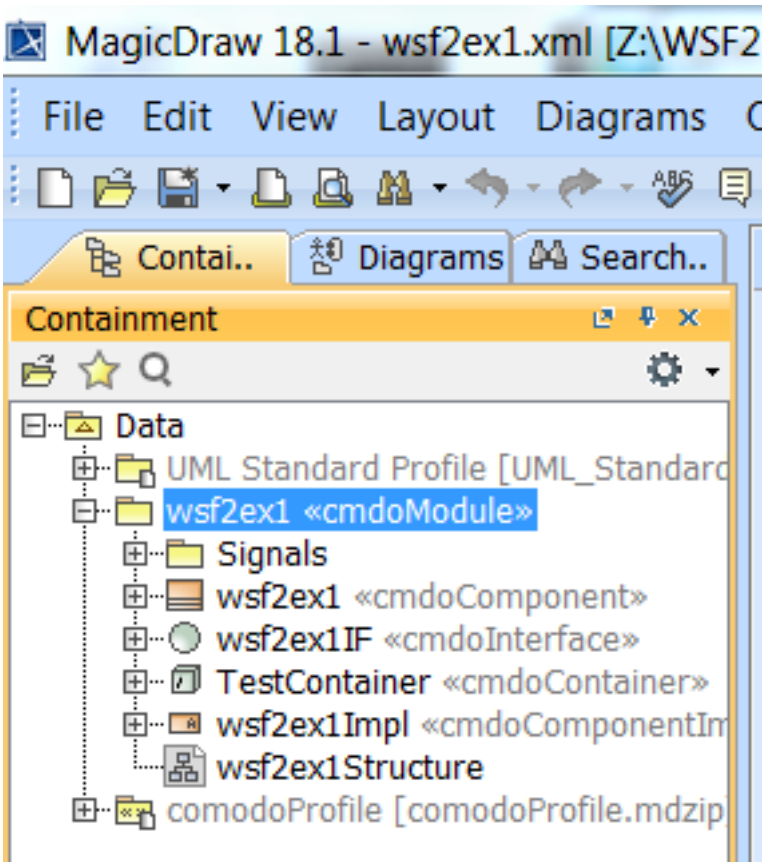


Figure 16 - <<cmdoModule>> Package’s content.

Figure 17 shows the signals (events) with their stereotypes included in the Package Signal (1) and used by the State Machine. It shows also the modeling elements representing: the State Machine (2), the State Machine diagram (3), the Transitions (4) and the States (5). Each composite state can be expanded; it contains its transitions and its sub-states. For example, STATE contains OFF, LOADED, ONLINE, STANDBY sub-states and the transitions from STATE to LOADED, etc.

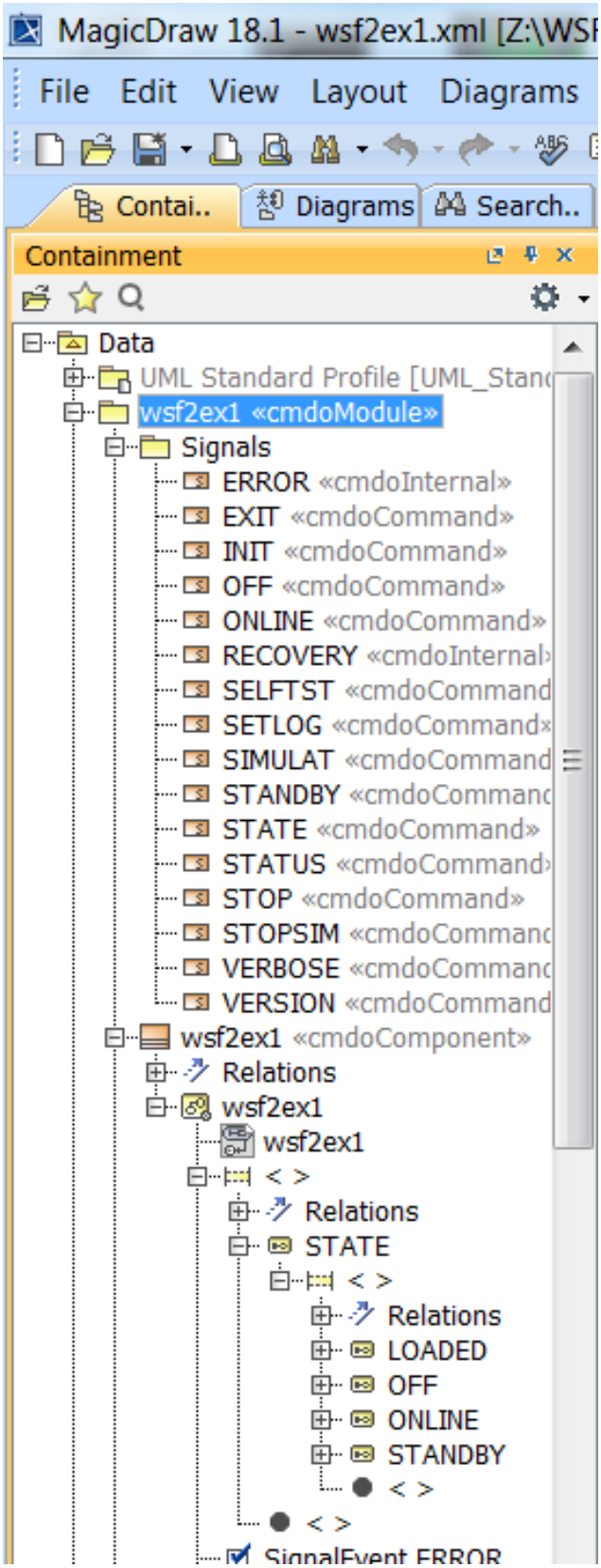

MagicDraw 18.1 - wsf2ex1.xml [Z:\WSF
File Edit View Layout Diagrams
Contai.. Diagrams Search..
Containment
Data
UML Standard Profile [UML_Stanc
wsf2ex1 «cmdoModule»
Signals
ERROR «cmdoInternal»
EXIT «cmdoCommand»
INIT «cmdoCommand»
OFF «cmdoCommand»
ONLINE «cmdoCommand»
RECOVERY «cmdoInternal»
SELFTST «cmdoCommand
SETLOG «cmdoCommand»
SIMULAT «cmdoCommand
STANDBY «cmdoCommanc
STATE «cmdoCommand»
STATUS «cmdoCommand»
STOP «cmdoCommand»
STOPSIM «cmdoCommanc
VERBOSE «cmdoCommanc
VERSION «cmdoCommand
wsf2ex1 «cmdoComponent»
Relations
wsf2ex1
wsf2ex1
< >
Relations
STATE
< >
Relations
LOADED
OFF
ONLINE
STANDBY
< >
< >
SignalEvent ERROR

Figure 17 – Signals and State Machine model elements for wsf2ex1 applications.

10.3.7 Opening Diagrams and Specification Dialogs

To open a diagram, double click on the diagram in the Containment tree.

To open the Specification Dialog of a model element, double click on the element in the Containment tree (or select the element and mouse-right-click to select the “Specification” option).

Important: in the top-right corner of the Specification Dialog, make sure that “Properties: All” is selected to be able to see all UML properties (Figure 18).

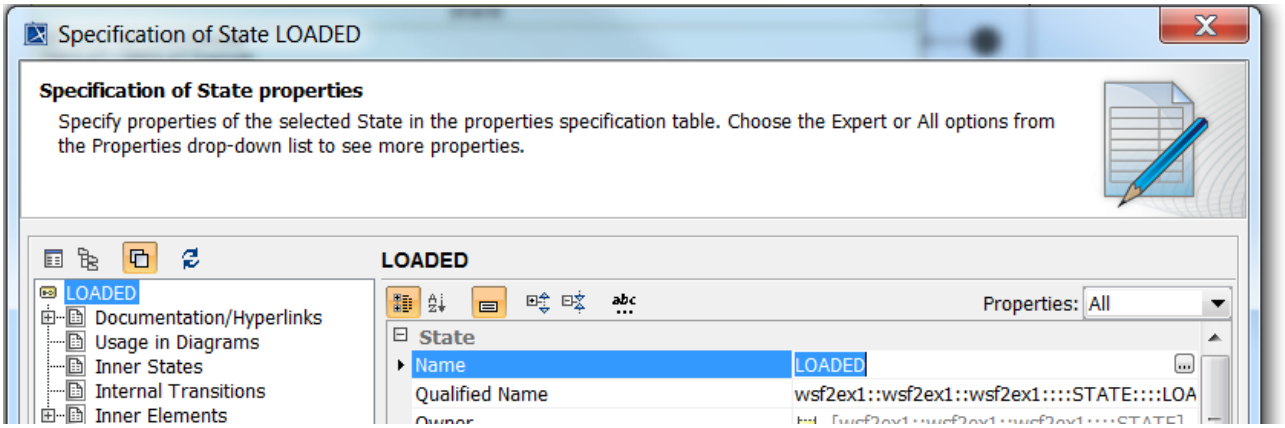


Figure 18 - Specification Dialog, all properties.