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: 4

Document Type: Manual (MAN)

Released on: 2024-12-11

Document Classification: Public

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

Release

This document corresponds to rad[1] v5.5.0.

Authors

Name	Affiliation
Andolfato, Luigi	ESO/DOE/CSE

Change Record from Previous Version

Affected Section(s)	Changes / Reason / Remarks
	See CRE ET-1517
All	All sections updated
2.1,2.4,3.2.8-3.2.11	New sections added

[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 *Create 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 Guards

Guards represent logical expressions associated to transitions which are evaluated by the State Machine interpreter before taking a transition. If the expression is True, the transition is taken, otherwise it is not. Similarly to Actions, Guards are implemented using methods of a C++ class which return a boolean value. The expression evaluation should take ideally zero time.

2.5 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.

2.6 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.

2.7 Error Handling

Exceptions and errors occurring within Actions, Guards, or Activities, can be handled in three 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 the Local Error Handling). E.g. an exception occurs in a Do-Activity (secondary thread) and the Do-Activity post an error event into the State Machine (main thread) to, for example, send an error reply.
- Global Error Handling, the exception is caught by the main() function within the global try-catch.

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

2.8 Application Development

In order to develop a RAD based application, the developer has 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, runtime data, and Online DB interface classes.

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 (CII)
1	Development Env.	Linux, GNU C++, waf, etc.
0	Hardware or VM	Servers

At 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 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 ELT[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).

Note: Since there was the need to start developing applications before the introduction of CII, a Prototype SW platform (made of ZeroMQ, Google Protocol Buffers, C++ exceptions, EasyLogging, Redis in-memory DB, YAML configuration files) is also supported but should not 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	CII Config-NG service
Online-DB	CII OLDB in-memory key/value DB

The application framework, level 3, can be used to develop State Machine based applications that

[3] http://eso.org/~eltmgr/ESO-288431_3_1%20Guide%20to%20Developing%20Software%20for%20the%20EELT.pdf

use the services described in the table above.

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
- cii
- services
- sm
- app
- appif
- utest
- itest

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().
DoubleMap	This class allows to share a map of attributes and associated values between producer threads and one consumer thread. It is used, for example, by OldbAsyncWriter to write asynchronously to the CII OLDB.

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

The following free functions in the Helper class are going to be replaced by the Time Library once available (see ESO-331947[5]).

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.

3.2.2 core

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

Warning: logger.hpp file still provides logging macros for the Prototype SW platform which have been declared obsolete.

Class	Description
ErrorCategory	Class representing RAD errors.
Exception	RAD exception. This class is similar to the CII Exception but it does not depend on MAL.
LogInitializer	Class to initialize and shutdown log4cplus.
Logger	Class used for logging as alternative to log4cplus. It is required for example by M1LCS but should not be used by CII based applications.

The file logger.hpp provides the following free functions:

Function	Description
Assert	Assert a condition. If the condition is false, it logs a fatal error.
LogInitialize	Initializes log services.
GetDefaultLogProperties	To get the default log4cplus configuration.
GetLogger	Returns the RAD logger (name = “rad”).
GetSmLogger	Returns the RAD State Machine logger (name = “rad.sm”).
GetAppLogger	Returns the a generic application logger (name = “app”).
GetAppLogger	Returns the log4cplus root logger.

[5] https://pdm.eso.org/kronodoc/HQ/ESO-331947

3.2.3 events

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

Class	Description
EventT	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 MAL libraries.

Class	Description
Publisher	Class that can be used to publish a topic using CII/ZPB.
Subscriber	Class that can be used to subscribe to a 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 cii

Library providing CII services. It depends on *core* library and requires CII MAL, Config, OLDB, and open-trace libraries.

Class	Description
OldbAdapter	Class that can be used to set/get OLDB attributes in CII OLDB.

3.2.6 services

Library providing messaging and in memory DB services for applications based on the Prototype Software Platform.

Class	Description
DbAdapter	Interface to read/write to a key-value in-memory DB.
DbAdapterRedis	Realization of DbAdapter interface for Redis DB.
MsgHandler	Base class for a ZMQ message handler.
MsgReplier	Class to deal with incoming ZMQ commandss
MsgRequestor	Class to send typed ZMQ commands and receive type ZMQ replies.
MsgRequestorRaw	Class to send raw ZMQ commands and receive raw ZMQ replies.
TopicHandler	Base class for a ZMQ pub/sub topic handler.
TopicPub	Class to publish ZMQ topics.
TopicSub	Class to subscribe and receive ZMQ topics.

Note: This library is to be considered obsoleted and replaced by the *mal* and *cii* libraries. It is still part of RAD to provide support to old applications like M1LCS which are still based on the Prototype Software Platform.

3.2.7 sm

Library providing State Machine services. It depends on *mal*, *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 dealing with UNIX signals events.
Timer	Class for dealing with time-out events.
TrsHealth	Class for dealing with Time Reference Signal health notifications.
SMEvent	Class to wrap RAD events into SCXML events.
SMAdapter	Facade to the SCXML State Machine engine.

Note: PthreadActivity allows to set some thread properties like priority, core assignment, scheduling algorithm via the constructor, ThreadActivity does not. CoroActivity is still experimental and it should allow to implement long lasting I/O operations without blocking using a method invocation instead of a thread.

3.2.8 appif

Library containing the MAL ICD for the RAD Interface used by the _app_ library. It does not depend on other RAD libraries. It contains the specification of the following commands:

Command	Description
SetConfig	Allows to reconfigure one or more configuration attributes by passing a string with the fully qualified identifier and its value or a complete or partial configuration in YAML format.
GetConfig	Return the value of a configuration attribute specified in the fully identifier of the parameter. If an empty string is given as identifier, the complete configuration is returned in YAML format.
LoadConfig	Load the configuration file specified in the argument.
SaveConfig	Save the configuration to file.
LoadStateMachine	Load the SCXML state machine model specified in the argument. If the given model is invalid the previous one is maintained.
LoadStateMachineExtension	Append to the current SCXML state machine model an extension loaded from file.
SaveStateMachine	Save to file the currently loaded SCXML model (including any added extensions).
GetStateMachine	Return the currently loaded SCXML state machine model in text format.
GetTrsHealth	Return the health status of the Time Reference Signal.

The MAL ICD XML file appif.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="appif">

  <exception name="Exception">
   <member name="desc" type="string"/>
   <member name="code" type="int32_t"/>
  </exception>

  <interface name="AppCmds">
   <method name="SetConfig" returnType="string" throws="Exception">
    <argument name="keyval" type="string"/>
   </method>
```

(continues on next page)

(continued from previous page)

```
    <method name="GetConfig" returnType="string" throws="Exception">
      <argument name="key" type="string"/>
    </method>

    ....

    <method name="GetStateMachine" returnType="string" throws="Exception">
    </method>
   </interface>
  </package>
</types>
```

3.2.9 app

Library to develop ELT applications implementing the ELT standard interface and the ELT standard state machine. It depends on several RAD libraries: appif, core, sm, cii, events, utils. It provides the following classes and files:

File/Class	Description
StdCmdsImpl	Class receiving the MAL RPC calls for the ELT Standard Interface and creating the corresponding State Machine events (listed in eventsStd.rad.ev file).
AppCmdsImpl	Class receiving the MAL RPC calls for the RAD Interface and creating the corresponding State Machine events (listed in eventsApp.rad.ev file).
eventsStd.rad.ev	Events injected into the State Machine engine and associated to the ELT Standard Interface.
eventsApp.rad.ev	Events injected into the State Machine engine and associated to the RAD Interface.
ActionsStd	Class implementing callback actions to deal with the events listed in eventsStd.rad.ev.
ActionsApp	Class implementing callback actions to deal with the events listed in eventsApp.rad.ev.
ConfigurableActionGroup	Class used to notify actions when configuration parameters are changed. It contains the Initialize and Configure virtual methods that can be specialized by the developer in subclasses to reinitialize and reconfigure action callbacks.
ConfigurableActivity	Class used to notify activities when configuration parameters are changed. It contains the Initialize and Configure virtual methods that can be specialized by the developer in subclasses to reinitialize and reconfigure activites.
ConfigurableActionMgr	Class used to notify actions and activities when configuration parameters are changed. It provides the Initialize and Configure virtual methods that can be used to trigger the re-initialization and reconfiguration of the actions and activities.
Application	Class implementing the application initialization and interfacing to the State Machine engine. If needed, it can be customized via inheritance by the developer.
Config	Base class to parse common command line options and retrieve configuration parameters. It should specialized via inheritance by a corresponding application Config class.
DataContext	Interface containing mandatory methods required by the Application class. This interface should be realized by the developer when implementing the application specific DataContext containing the application configuration and run-time data.
OldbInterface	Class to write to the OLDB synchronously via the OldbAdapter.
OldbAsyncWriter	Class to write to the OLDB asynchronously via the ActivityUpdateOldb.
ActivityUpdateOldb	Class implementing an activity to write periodically to the OLDB.

3.2.9.1 rad::StdCmdsImpl

The rad::StdCmdsImpl class realizes the interface class stdif::AsyncStdCmds generated by MAL from the MAL ICD XML file ELT Standard Commands[6].

For each RPC call specified in the generated interface class, a method is implemented as shown below for the Exit and SetLogLevel commands:

```
class StdCmdsImpl : public stdif::AsyncStdCmds {
public:
  explicit StdCmdsImpl(rad::SMAdapter& sm);
  virtual ~StdCmdsImpl();

  virtual elt::mal::future<std::string> Exit() override;
  virtual elt::mal::future<std::string> Init() override;
  virtual elt::mal::future<std::string> Stop() override;
  virtual elt::mal::future<std::string> Reset() override;
  virtual elt::mal::future<std::string> GetState() override;
  virtual elt::mal::future<std::string> GetStatus() override;
  virtual elt::mal::future<std::string> GetVersion() override;
  virtual elt::mal::future<std::string> Enable() override;
  virtual elt::mal::future<std::string> Disable() override;
  virtual elt::mal::future<std::string> SetLogLevel(const std::shared_ptr<stdif::LogInfo>& info)␣
 ↪override;

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

```
elt::mal::future<std::string> StdCmdsImpl::Exit() {
  RAD_TRACE(GetLogger());
  auto ev = std::make_shared<EventsStd::Exit>();
  m_sm.PostEvent(ev);
  return ev->GetPayload().GetReplyFuture();
}
...
elt::mal::future<std::string> StdCmdsImpl::SetLogLevel(const std::shared_ptr<stdif::LogInfo>&␣
 ↪info) {
  RAD_TRACE(GetLogger());
  auto ev = std::make_shared<EventsStd::SetLogLevel>(info->clone());
  m_sm.PostEvent(ev);
  return ev->GetPayload().GetReplyFuture();
}
```

[6] https://gitlab.eso.org/ecos/ecs-interfaces/-/blob/master/std/if/src/stdif.xml/

The StdCmdsImpl constructor takes as parameter a reference to the State Machine adapter which is needed to inject the event into the State Machine interpreter.

In the StdCmdsImpl::Exit() method the event EventsStd::Exit (which is specified in *eventsStd.rad.ev* file) is created and injected into the State Machine interpreter via the PostEvent() method. Finally the future provided by the event is returned to CII MAL.

StdCmdsImpl::SetLogLevel() method is similar to the the Exit case with the difference that the SetLogLevel command takes a parameter of type stdif::LogInfo. In this case the corresponding event EventsStd::SetLogLevel is instantiated passing a copy of the parameter as event's payload.

Note: Data structure parameters (e.g. *info* in the SetLogLevel command) are passed as std::shared_ptr by MAL but they cannot be shared (see ECII-195[7]) and therefore they must be copied via the clone() method.

3.2.9.2 rad::AppCmdsImpl

The class rad::AppCmdsImpl class realizes the interface class appif::AsyncAppCmds generated by MAL from the MAL ICD XML file described in *appif* .

This class is similar to *rad::StdCmdsImpl* but, in addition to the State Machine adapter, it takes in the constructor also a reference to the ActionMgr which is needed by the LoadStateMachine() and LoadStateMachineExtension() methods to load a State Machine model or extension.

```
class AppCmdsImpl : public appif::AsyncAppCmds {
public:
  explicit AppCmdsImpl(rad::SMAdapter& sm, rad::ActionMgr& action_mgr);
  virtual ~AppCmdsImpl();

  virtual elt::mal::future<std::string> GetConfig(const std::string& p) override;
  virtual elt::mal::future<std::string> SetConfig(const std::string& p) override;
  virtual elt::mal::future<std::string> LoadConfig(const std::string& p) override;
  virtual elt::mal::future<std::string> SaveConfig(const std::string& p) override;
  virtual elt::mal::future<std::string> GetTrsHealth() override;
  virtual elt::mal::future<std::string> LoadStateMachine(const std::string& p) override;
  virtual elt::mal::future<std::string> LoadStateMachineExtension(const std::string& p) override;
  virtual elt::mal::future<std::string> SaveStateMachine(const std::string& p) override;
  virtual elt::mal::future<std::string> GetStateMachine() override;

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

[7] https://jira.eso.org/browse/ECII-195

```
elt::mal::future<std::string> AppCmdsImpl::GetConfig(const std::string& p) {
  RAD_TRACE(GetLogger());
  auto ev = std::make_shared<EventsApp::GetConfig>(p);
  m_sm.PostEvent(ev);
  return ev->GetPayload().GetReplyFuture();
}

elt::mal::future<std::string> AppCmdsImpl::SetConfig(const std::string& p) {
  RAD_TRACE(GetLogger());
  auto ev = std::make_shared<EventsApp::SetConfig>(p);
  m_sm.PostEvent(ev);
  return ev->GetPayload().GetReplyFuture();
}
...
```

3.2.9.3 eventsStd.rad.ev

The eventsStd.rad.ev lists all the events that are associated to the commands in *rad::StdCmdsImpl*:

```
# Event definitions for ELT Standard Interface
version: "1.0"

namespace: EventsStd

includes:
  - boost/exception_ptr.hpp
  - rad/mal/request.hpp
  - Stdif.hpp

events:
    Disable:
        payload: rad::cii::Request<std::string>
    ...
    SetLogLevel:
        payload: rad::cii::Request<std::string, std::shared_ptr<stdif::LogInfo>>
```

From this file the tool *radgen* generates in the build/ directory the files eventsStd.rad.hpp and eventsStd.rad.cpp containing the C++ classes representing the events to be injected into the State Machine interpreter.

3.2.9.4 eventsApp.rad.ev

Similarly to *eventsStd.rad.ev*, this file lists all the events which are associated to the commands in *rad::AppCmdsImpl*, the events associated to the Linux signals (CtrlC, SigUsr1), and the Error internal event that can be triggered by the application in case of errors:

```
# RAD Applications Common Event definitions
version: "1.0"

namespace: EventsApp

includes:
  - boost/exception_ptr.hpp
  - rad/mal/request.hpp
  - Appif.hpp

events:
    CtrlC:
        doc: Event representing the SIGINT and SIGTERM Linux signals to quit the application.
    SigUsr1:
        doc: Event representing the SIGUSR1 Linux signal used by Nomad to notify a change in
↪the deployment configuration.
    Error:
        doc: Event triggered by the ActivityEstimate when an error occurs.
    SetConfig:
        payload: rad::cii::Request<std::string, std::string>
    GetConfig:
        payload: rad::cii::Request<std::string, std::string>
    LoadConfig:
        payload: rad::cii::Request<std::string, std::string>
    SaveConfig:
        payload: rad::cii::Request<std::string, std::string>
    GetTrsHealth:
        payload: rad::cii::Request<std::string>
    GetStateMachine:
        payload: rad::cii::Request<std::string>
    SaveStateMachine:
        payload: rad::cii::Request<std::string, std::string>
```

From this file the tool *radgen* generates in the build/ directory the files eventsApp.rad.hpp and eventsApp.rad.cpp containing the C++ classes representing the events to be injected into the State Machine interpreter.

3.2.9.5 rad::ActionsStd

This class is used to group the methods implementing the actions (i.e. callbacks) invoked according to the triggered events (listed in *eventsStd.rad.ev*) and the current state (as defined in the *State Machine Model*).

For example it include the method ActionsStd::GetState() which is invoked when the EventsStd::GetState event object is injected in the State Machine interpreter. The EventsStd::GetState object is created by the StdCmdsImpl::GetState() method which is invoked by CII/MAL when stdif::GetState RPC is invoked by a client application.

The following table shows the list of available actions:

Method	Description
Exit	Reply OK to the originator of the command and terminates the Boost ASIO event loop forcing the application to quit.
GetState	Reply to the originator of the command with the current active state(s).
GetStatus	To be specialized by the application. Reply to the originator of the command with the current active state(s).
GetVersion	Reply to the originator of the command with the version of the application. The version is taken by WAF from the project wscript and injected to the application via the compiler preprocessor.
Init	Initializes all actions and activities by invoking the Initialize() method of rad::ConfigurableActionMgr and reply OK to the originator of the command.
Enable	To be specialized by the application. Reply OK to the originator of the command.
Disable	To be specialized by the application. Reply OK to the originator of the command.
Reset	To be specialized by the application. Reply OK to the originator of the command.
Stop	To be specialized by the application. Reply OK to the originator of the command.
SetLogLevel	Set the log level for a given logger. Replies OK to the originator of the command if the log level and the logger exist.

3.2.9.6 rad::ActionsApp

This class is used to group the methods implementing the actions (i.e. callbacks) invoked according to the triggered events (listed in *eventsApp.rad.ev*) and the current state (as defined in the *State Machine Model*). In addition it includes some actions to deal with the rad::TrsHealth events: TrsHealthGoodEvent and TrsHealthBadEvent.

The following table shows the list of available actions:

Method	Description
GetConfig	Queries the *Config* object and return the full application configuration or the configuration of the given parameters.
SetConfig	Allows to set one configuration parameter of a subset of parameters.
LoadConfig	Load the given configuration file.
SaveConfig	Save to the given file the complete application configuration.
TrsHealth	Logs a warning if the TrsHealthBadEvent event was triggered or an info if the TrsHealthGoodEvent was triggered.
GetTrsHealth	Return the health of the Time Reference Signal and the reason.
GetStateMachine	Return the State Machine Model in text format.
SaveStateMachine	Save to the given file the State Machine Model in text format.

3.2.9.7 rad::ConfigurableActionGroup

The ConfigurableActionGroup specializes the rad::ActionGroup by adding the Initialize() and Configure() methods which can be invoked when the application is initialized (e.g. when Init command is received) or when the configuration changes (e.g. when the SetConfig or LoadConfig commands are received).

An ActionGroup is a class that contains a group of methods which correspond to State Machine actions. These methods are invoked by the State Machine interpreter when entering/exiting a state or when a transition is taken.

3.2.9.8 rad::ConfigurableActivity

The ConfigurableActivity specializes the rad::ThreadActivity class by adding the Initialize() and Configure() interfaces which can be used when the application is initialized (e.g. when Init command is received) or when the configuration changes (e.g. when the SetConfig or LoadConfig commands are received).

An Activity is a class that allows to run long lasting tasks in dedicated threads (using std::thread). The thread is started by the State machine interpreter when a state is entered and stopped when the state is exited.

3.2.9.9 rad::ConfigurablePthreadActivity

The ConfigurablePthreadActivity specializes the rad::PthreadActivity class by adding the Initialize() and Configure() interfaces which can be used when the application is initialized (e.g. when Init command is received) or when the configuration changes (e.g. when the SetConfig or LoadConfig commands are received).

The difference w.r.t ConfigurableActivity is that this class uses pthread instead of std::thread. pthread API allows to set at creation time the priority and the CPU node where to run the thread.

3.2.9.10 rad::ConfigurableActionMgr

The ConfigurableActionMgr specializes the rad::ActionMgr class by adding the Initialize() and Configure() methods which can be used to invoke the Initialize() and Configure() methods of all registered ConfigurableActionGroup, ConfigurableActivity, and ConfigurablePthreadActivity objects.

The ActionMgr is a factory class that can be used to instantiate and register ActionGroup and Activity objects.

3.2.9.11 rad::Config

The Config class provides methods to store and access the application configuration. It uses CII config-ng to store internally the configuration parameters which in turn uses yaml-cpp. Configuration parameters can be set using constant values, values from environment variables, values from YAML configuration files, or values from command line options.

Each configuration parameter has a string identifier which is defined in the Config.hpp header file. The app library provides the following identifiers:

Identifier	Type	Description
cfg/version	String	Read only parameter storing the version of the application retrieved from WAF project wscript.
cfg/modname	String	Parameter storing the module name.
cfg/procname	String	Read only parameter storing the process name. It can be changes via the -n command line option.
cfg/filename	String	Parameter to store the file path of the loaded YAML configuration file.
cfg/log_level	String	Configure the log level of the main application logger.
cfg/log_properties	String	File path of the log4cplus property file containing the logging configuration of the application's loggers.
cfg/sm_scxml	String	File path of the SCXML State Machine model.
cfg/sm_scxml_append	String	File path of the SCXML State Machine model extension.
cfg/req_endpoint	String	URI used to receive the commands using MAL ZPB request/reply protocol.
cfg/oldb_uri_prefix	String	URI used to connect to the CII OLDB. It can contain an initial data point prefix.
cfg/oldb_conn_timeout	int	Timeout in sec to connect to the CII OLDB.
cfg/trs_health_enabled	bool	Flag enabling or disabling the periodic TRS health check.

The Config class allows to:

- Add (AddParam()), read (GetParam()), write (SetParam()), and check (HasParam()) configuration parameters.
- Load configuration file (LoadConfig())
- Load and apply log4cplus logging properties file (ConfigureLogging())
- Merge configurations (MergeConfig())
- Parse command line options (ParseOptions())

The Config constructor allows to initialize the logging level for the main application logger. The main application logger name is also used to create the filename of the logging file.

3.2.9.12 rad::DataContext

The DataContext interface allows to:

- access the application specific configuration parameters via the GetConfig() method.
- publish attributes to the OLDB via the GetOldbInterface() and UpdateDb() methods.
- reload the configuration file via the ReloadConfig() method.

This interface should be implemented by an application specific DataContext class which should include, via composition, all the data (runtime and configuration) used and produced by the application's

actions and activities.

3.2.9.13 rad::OldbInterface

This base class can be used to write synchronously to the OLDB via the OldbAdapter. It can be specialized by an application specific OldbInterface containing the methods to publish information to the OLDB.

It contains methods to:

- Read attributes from the OLDB (GetValue())
- Write attributes to the OLDB (SetValue())
- Write the configuration parameters specified in *rad::Config* to the OLDB (SetConfig()).
- Write the application state to the OLDB (SetControlState()).
- Write the TRS health status to the OLDB (SetTrsHealth()).

3.2.9.14 rad::OldbAsyncWriter

This class can be used to write to the OLDB asynchronously via the *rad::ActivityUpdateOldb* activity.

It provides the methods to:

- Start and stop the thread that writes to the OLDB (StartWriter() and StopWriter()).
- Write single or vector of attributes and related values (Set()).

The attributes/values are stored in a std::map shared with the *rad::ActivityUpdateOldb* activity which is responsible to remove them and write them to the OLDB.

Note: If the Set() method is invoked faster than the period at which the *rad::ActivityUpdateOldb* activity is configured to run (and to pop the attributes/values), the old values will be overwritten by the new ones. This is the intended behavior since it is assumed that the most recent value is the most important.

3.2.9.15 rad::ActivityUpdateOldb

This class implements an activity to write periodically to the OLDB the attributes and values stored in a std::map data structured shared with the *rad::OldbAsyncWriter* class.

It can be started/stopped by the *rad::OldbAsyncWriter* class using the StartWriter() and StopWriter() or the activity can be added in a state of the SCXML State Machine model (the thread will be started by the State Machine interpreter when entering the state and stopped when leaving the state).

3.2.9.16 rad::Application

This class groups the Boost ASIO event loop and the State Machine interpreter and provides methods to initialize the application.

It uses configuration information retrieved via the *rad::DataContext* interface. It is used by the *rad::ConfigurableActionMgr* since actions and activities may need to be able to inject events into the State Machine interpreter. Therefore is has to be created after the application specific DataContext and before the application specific ActionMgr.

3.2.10 utest

Library containing some helper classes to facilitate the creation of unit tests for applications based on the app library. It contains, within the rad::utest namespace, the following classes:

Class	Description
ActionMgr	Basic implementation of the rad::ConfigurableActionMgr interface that creates the actions for the standard and app commands.
Activity	Basic implementation of the rad::ConfigurableActivity interface.
Application	Specialization of the rad::Application class that allows to initialize the application using a string state machine model (instead of loading the model from file).
Config	Specialization of the rad::Config class that initializes the common configuration parameters.
DataContext	Basic implementation of the rad::DataContext interface.
OldbAdapter	Specialization of the rad::cii::OldbAdapter class that allows to run the CII OLDB in local memory of the application.
OldbInterface	Specialization of the rad::OldbInterface class.

3.2.11 itest

Library to facilitate the creation of integration tests with Robot Framework.

It provides Robot keywords as:

- Robot resource files containing custom Robot keywords.
- Python classes implementing custom Robot keywords.

Resource File	Keywords
itest.resource	**VerifySendCmdNoParams** to send a command with msgsend without parameters. **VerifySendCmd** to send a command with msgsend with parameters. **FileShouldContain** to check whether a string is contained in a file. **CheckValgrindResults** to check for errors in Valgrind memory leak report.
itestStdIf.resource	**VerifyGetState** to verify the stdif::StdCmds::GetState command. **VerifyGetStatus** to verify the stdif::StdCmds::GetStatus command. **VerifyGetVersion** to verify the stdif::StdCmds::GetVersion command. **VerifyStop** to verify the stdif::StdCmds::Stop command. **VerifyExit** to verify the stdif::StdCmds::Exit command. **VerifyInit** to verify the stdif::StdCmds::Init command. **VerifyEnable** to verify the stdif::StdCmds::Enable command. **VerifyDisable** to verify the stdif::StdCmds::Disable command. **VerifyReset** to verify the stdif::StdCmds::Reset command. **VerifySetLogLevel** to verify the stdif::StdCmds::SetLogLevel command. **VerifyInterface** all the commands above.
itestAppIf.resource	**VerifyGetConfig** to verify the appif::AppCmds::GetConfig command. **VerifySetConfig** to verify the appif::AppCmds::SetConfig command. **VerifyLoadConfig** to verify the appif::AppCmds::LoadConfig command. **VerifySaveConfig** to verify the appif::AppCmds::SaveConfig command. **VerifyGetTrsHealth** to verify the appif::AppCmds::GetTrsHealth command. **VerifyLoadStateMachine** to verify the appif::AppCmds::LoadStateMachine command. **VerifyLoadStateMachineExtension** to verify the appif::AppCmds::LoadStateMachineExtension command. **VerifySaveStateMachine** to verify the appif::AppCmds::SaveStateMachine command. **VerifyGetStateMachine** to verify the appif::AppCmds::GetStateMachine command. **VerifyCfgInterface** to verify all configuration commands. **VerifyTrsInterface** to verify all TRS commands. **VerifySmInterface** to verify all State Machine commands.
itest-Startup.resource	**LaunchSubscriberLocal** starts a topic subscriber on local machine. **TerminateSubscriberLocal** terminates a topic subscriber on local machine and logs the stderr and stdout. **RunSubscriberLocal** run a topic subscriber on local machine until a number of messages are received. **LaunchAppLocal** starts a RAD application on local machine. **LaunchAppWithNameLocal** starts a RAD application with different name on local machine. **TerminateAppLocal** terminates a RAD application running on local machine. **SignalAppLocal** sends a signal to a RAD application running on local machine.

Python Classes	Keywords
oldb.py	**acquire_oldb** Connect to the OLDB. **read_from_oldb** Read an attributed from the OLDB. **should_match_oldb** Verifies that an attribute has the given value. **should_not_match_oldb** Verifies that an attribute does not match the given value. **should_match_bool_oldb** Verifies that a boolean attribute has the given value. **write_to_oldb** Write a values to an attribute of the OLDB. Note that these keywords assume that the OLDB service is running.

3.2.12 scxml4cpp

scxml4cpp is an ESO product able to parse and execute an SCXML model. It is made of two libraries: the parser and the engine. The parser is based on xerces-c++[8] 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[9]. Only a subset of the SCXML features are supported. In particular the SCXML standard actions and the possibility to use an interpreted action language is not implemented. Instead actions are mapped to methods of C++ classes.

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[10]) that is used to generate a RAD based applications from templates. The templates are stored in rad/rad/cpp/template/resource/template directory.

Note:

- If RAD is cloned from Git, templates can be installed in the $INTROOT/resource/template directory via the waf install command.

[8] https://xerces.apache.org/xerces-c/
[9] https://www.w3.org/TR/scxml/
[10] https://cookiecutter.readthedocs.io

- If RAD is installed via RPM, templates can be found in $RAD_ROOT/resource/template directory.

3.3.2 radgen

radgen 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*

4 RAD Installation

4.1 Environment Configuration

To configure environment variables LMOD tool (see LMOD User Guide[11]) 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 $HOME/modulefiles/private.lua file. For example:

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

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

load("introot")

local cfgpath = pathJoin(home, "ELT/ELT-INTROOT/resource")
setenv("CFGPATH", cfgpath)

setenv("CII_LOGS", home)
```

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.
- CII_LOGS is used by CII Logging to store the log files.

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

```
>module load private
```

[11] https://lmod.readthedocs.io/en/latest/010_user.html

4.2 Installation with RPM

RAD is distributed as RPMs archived in the eso-elt-projects repo:

- elt-rad.x86_64
- elt-rad-devel.x86_64
- elt-rad-doc.noarch

Note that elt-rad-devel contains only the libraries and binaries. It is recommended to install elt-rad-devel which includes also the templates.

RAD can be installed using the dnf install command as root:

```
root>dnf install elt-rad-devel.x86_64
```

If RAD is installed via RPM, the binaries, libraries, interfaces, sources, and templates are located in $RAD_ROOT directory (e.g. /elt/rad directory).

4.3 Installation from GIT

4.3.1 Retrieving RAD from GIT

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

It can be retrieved via HTTP with the following command:

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

Note: Username and password have to be provided.

As alternative it can be retrieved via SSH:

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

Note: SSH key must be configured.

4.3.2 Building and Installing RAD

RAD can be compiled on a (virtual) machine installed with the ELT DevEnv 3.5.0-7 (or more recent version) with the following commands:

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

RAD can be installed into the $PREFIX directory by:

```
>waf install
```

RAD documentation, doxygen and User Manual, can be generated using:

```
>waf --with-docs
```

The doxygen documentation is in in rad/build/docs directory. The User Manual is in in rad/build/doc/manual/html directory.

4.3.3 Directory Structure

RAD project is organized in the following directories:

Directory	Description
doc	RAD User Manual.
rad	RAD Libraries and tools.
rad/rad/codegen	Code generator tool to create C++ event classes.
rad/rad/cpp	Libraries for C++ Applications
rad/rad/py	Libraries for Python Applications (not supported)
rad/rad/itest	Library to facilitate development of integration test.
scxml4cpp	SCXML State Machine engine for C++.
scxml4py	SCXML State Machine engine for Python.
test	RAD Integration Tests.

The Libraries for C++ Applications are organized in the following directories:

Directory	Description
utils	Library providing common utility functions (e.g. FindFile)
core	Library providing error handling and logging services.
events	Library providing events related services.
mal	Library providing CII messaging services.
cii	Library providing other CII messaging services like OLDB.
services	Library providing ZMQ messaging, DB, and other services.
sm	Library providing State Machine service.
app	Library to facilitate the development of ELT CII Applications.
appif	MAL library containing RAD interface for ELT CII Applications.
gtlogcap	Library that allows to capture Unit Test log messages.
template	Templates to create RAD based projects and applications.
_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/templates* 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 from Git.

6 Tutorial 1: Creating an Application with RAD + CII

This tutorial shows how to develop an application for the ELT CII software platform implementing:

- the standard ELT State Machine and command interface. The standard command interface is part of the DevEnv.
- the RAD app command interface which is part of RAD and it is specified in the rad/cpp/appif SW module: *appif*

In order to develop the application, the following steps are performed:

1. Generate WAF Project
2. Generate Interface Module
3. (optional) Generate Topic Subscriber 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: Step 1 can be skipped if you are adding your application to an existing WAF project.

Step 2 can be skipped if you are adding your application to an existing WAF project which has already the interface module.

6.1 Generate CII WAF Project

The build system for the ELT software is based on WAF and requires the creaton of a WAF project. A WAF project is made of a directory (e.g. “hello”) that contains a “wscript” file, declaring the root of a WAF project, and the SW modules organized in sub-directories. See WAF User Manual[12] for more information on WAF projects.

An “hello” WAF project can be generated from a template by executing the following commands and entering the requested information:

```
> cookiecutter rad/rad/cpp/template/resource/template/rad-waftpl-ciiprj

project_name [hello]: hello
modules_name [hellociiif hellocii hellociisub]:
```

[12] https://www.eso.org/~eeltmgr/documents/latest/wtools-docs/archive/html/index.html

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.
- If RAD has been installed via RPM, the template is located in $RAD_ROOT/resource/template/rad-waftpl-ciiprj directory.

From the template Cookiecutter generates the directory *hello* and inside the file *wscript*. This file 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 MAL XML ICD language.

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

```
> cd hello
> cookiecutter ../rad/rad/cpp/template/resource/template/rad-cpptpl-ciiapplif

module_name [hellociiif]: hellociiif
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 *hellociiif* containing the following files:

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

The file hellociiif.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="hellociiif">

  <struct name="TelPosition">
   <member name="ra" type="float" />
   <member name="dec" type="float" />
  </struct>

 </package>
</types>
```

It specifies an example of data structure, the *TelPosition*, that can be used to specify the RA/DEC coordinates in commands and pub/sub topics.

This file can be updated with more data structures, exceptions to report errors, and command interfaces (see Tutorial 2 for how to add commands).

For more information on the CII/MAL XML interface definition language, refer to MAL ICD User Manual[13].

The hellociiif.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 Topic Subscriber Module

The subscriber module implements a simple tool that can be used to verify that the server application is publishing the telescope position.

```
> cd hello
> cookiecutter ../rad/rad/cpp/template/resource/template/rad-cpptpl-ciisub

module_name [hellociisub]: hellociisub
application_name [hellociisub]: helloCiiSub
parent_package_name [hello]: hello
interface_name [hellociiif]: hellociiif
interface_module [hellociiif]: hellociiif
topic_name [TelPosition]: TelPosition
```

The input values to the template are:

- *module_name:* the name of the SW module to generate.

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

- *application_name:* the name of the binary to be produced when compiling the generated module. *Note that ELT convention for binaries is lowerCamelCase.*
- *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 of the interface library.
- *topic_name:* The name of the topic to subscribe to (and published by the server application).

From the template Cookiecutter generates the directory *hellociisub* containing one file main.cpp implementing a tool to subscribe to the given topic (e.g. TelPosition) which has been specified in the interface module (e.g. hellociiif).

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/template/resource/template/rad-cpptpl-ciiappl

module_name [hellocii]: hellocii
application_name [hellocii]: helloCii
parent_package_name [hello]: hello
interface_name [hellociiif]: hellociiif
interface_module [hellociiif]: hellociiif
```

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. *Note that ELT convention for binaries is lowerCamelCase.*
- *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 *hellocii* containing the following files:

File	Description
wscript	WAF file to build the application.
resource/config/hellocii/config.yaml	YAML application configuration file.
resource/config/hellocii/sm.xml	SCXML file with the State Machine model.
resource/config/hellocii/log.properties	Logging configuration file.
src/actionMgr.[hpp|cpp]	Class responsible for instantiating actions and activities.
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/olddbInterface.[hpp|cpp]	Class interfacing with the Online DB.
src/logger.[hpp|cpp]	Default logger definition.
src/main.cpp	Application entry function.
test/testActionMgr.cpp	Example of Unit Test.

6.4.1 wscript

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

```
from wtools.module import declare_cprogram

declare_cprogram(target='helloCii',
                 features='radgen',
                 use=('BOOST yaml-cpp log4cplus cpp-netlib-uri xerces-c config-ng.cpp.config-ng '
                      'rad.cpp.utils rad.cpp.core rad.cpp.mal rad.cpp.cii rad.cpp.app rad.cpp.sm '
                      'rad.cpp.events rad.cpp.appif-cxx hellociiif-cxx '
                      'trs-ptpmon-client.ptpmonLib.cpp gsl'))
```

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

- *BOOST* for event loop etc.
- *yaml-cpp* to load YAML configuration files.
- *log4cplus* for logging.
- *xerces-c* required to parse SCXML State Machine Model.
- *config-ng.cpp.config-ng* CII Config Service.
- *rad.cpp.utils, rad.cpp.core, . . .*, RAD libraries.
- *hellocii-cxx* CII/MAL generated interface 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/"
  sm_scxml        : "config/hellocii/sm.xml"
  log_properties  : "config/hellocii/log.properties"
  oldb_uri_prefix : "cii.oldb:/elt/"
  oldb_conn_timeout : 1
  trs_health_enabled : 0
```

These configuration parameters correspond to the attributes defined in the *rad::Config* class. This set of parameters can be extended by adding application specific parameters.

6.4.3 log.properties

Logging APIs are provided by log4cplus library[14]. Logging service can be configured via the following configuration file.

```
log4cplus.logger.malZpbClientAsyncImpl=ERROR
log4cplus.logger.malZpbServer=ERROR
log4cplus.logger.rad=INFO
log4cplus.logger.rad.sm=INFO
log4cplus.logger.scxml4cpp=INFO
log4cplus.logger.hellocii=INFO
```

In the file it is possible to specify the log level for each logger (e.g. *rad*, *scxml4cpp*, *hellocii*). The log appenders, used to print the log messages to console or save to file, are specified via API for the root logger and inherited by all the loggers.

The 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"?>
<!-- hellocii StateMachine -->
<scxml xmlns="http://www.w3.org/2005/07/scxml" xmlns:customActionDomain="http://my.
↪custom-actions.domain/CUSTOM"
 version="1.0" initial="On">
```

(continues on next page)

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

(continued from previous page)

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

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

  <state id="On::NotOperational::NotReady">
   <transition event="EventsStd.Init" target="On::NotOperational::Ready">
    <customActionDomain:ActionsStd.Init name="ActionsStd.Init"/>
   </transition>
  </state>

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

  <transition event="EventsApp.LoadConfig">
   <customActionDomain:ActionsApp.LoadConfig name="ActionsApp.LoadConfig"/>
  </transition>
 </state>

 <state id="On::Operational">
  <transition event="EventsStd.Disable" target="On::NotOperational::Ready">
   <customActionDomain:ActionsStd.Disable name="ActionsStd.Disable"/>
  </transition>
 </state>

 <transition event="EventsStd.Reset" target="On::NotOperational::NotReady">
  <customActionDomain:ActionsStd.Reset name="ActionsStd.Reset"/>
 </transition>

...

 <transition event="EventsApp.SetConfig">
  <customActionDomain:ActionsApp.SetConfig name="ActionsApp.SetConfig"/>
 </transition>
</state>
```

(continues on next page)

(continued from previous page)

```
  <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 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 *EventsStd.Init* event.
- It is possible to move from *On/NotOperational/Ready* to *On/Operational* via the *EventsStd.Enable* event.
- It is possible to move from *On/Operational* to *On/NotOperational/Ready* via the *EventsStd.Disable* or the *EventsStd.Stop* event.
- It is possible to move from any state back to *On/NotOperational/NotReady* via the *EventsStd.Reset* event.
- It is possible to terminate the application from any state via the *EventsStd.Exit* or the *EventsApp.CtrlC* events.
- It is possible to move from any state to *On/NotOperational/NotReady* via the *EventsStd.Reset* event.
- It is possible to move from any state to *On/NotOperational/Ready* via the *EventsStd.Init* event.
- All the remaining transitions are so called “internal transition”: they do not trigger a change of state (e.g. EventsStd.GetState).

The initial state after the application has started is: *On/NotOperational/NotReady*.

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.

The *app* library provides the events EventsStd.* (e.g. EventsStd.Init) and EventsApp.* (e.g. EventsApp.SetConfig) with the *eventsStd.rad.ev* and *eventsApp.rad.ev* files together with a basic implementation of the actions ActionsStd.* (e.g. ActionsStd.Reset) and ActionsApp.* (e.g. ActionsApp.SetConfig) via the *rad::ActionsStd* and *rad::ActionsApp* classes.

6.4.5 actionMgr.hpp|cpp

The *ActionMgr* class is responsible, via the *CreateActions* and *CreateActivies* methods, for instantiating the objects implementing the actions (callbacks) and the activities (e.g. threads).

In this part of the tutorial only the actions provided by *app* library are used and therefore the ActionMgr::CreateActions() method simply invokes the base class helper methods to create the ActionsStd and ActionsApp objects and register the associated callbacks:

```
void ActionMgr::CreateActions(rad::Application& appl) {
    RAD_TRACE(GetLogger());

    CreateActionsForStdEvents(appl, m_data);
    CreateActionsForAppEvents(appl, m_data);
}
```

m_data represents the application runtime and configuration data object which is instantiated in the *main.cpp* and it is provided to the *ActionMgr* via constructor. The appl reference allows to interact with the event loop and the State Machine interpreter.

The second part of the tutorial (*Tutorial 2: Customizing an Application with RAD + CII*) illustrates how to add new commands, actions, and activities.

6.4.6 config.hpp|cpp

The *Config* class (which inherits from *rad::Config* class provided by the *app* library) is responsible for providing access to the application configuration and for initializing the configuration parameters with default values in the constructor.

Durign the initialization the following order should be followed:

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

The *Config* class constructor initializes the configuration attributes with the default values and, if applicable, the environment variables values.

The *Config* class is usually a member of the DataContext (*dataContext.hpp|cpp*) class which is instantiated in *main.cpp*.

The command line options are parsed and the YAML configuration file loaded as part of the application start-up sequence coded in the *rad::Application* class Init() methods.

Note: Due to CII limitations, configuration files are loaded *only* from directories specified in the CFGPATH environment variable.

6.4.7 oldbInterface.hpp|cpp

The *OldbInterface* class inherits from the *rad::OldbInterface* class and should be used to read/write application specific configuration and runtime data to the OLDB synchronously.

The *OldbInterface* constructor takes as parameter a string representing the prefix to be added to all the attributes identifiers before accessing the OLDB.

The *OldbInterface* class is usually a member of the *DataContext* (*dataContext.hpp|cpp*) class which is instantiated in *main.cpp*.

6.4.8 dataContext.hpp|cpp

The *DataContext* class allows to share run-time and configuration data among actions and activities. This class allows also to write publish the data to the OLDB via the *OldbInterface* class.

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

6.4.9 logger.hpp|cpp

log4cplus library provides the possibility to associate 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.

It suggested 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 = "hellocii";
```

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

```
log4cplus::Logger& GetLogger() {
    static log4cplus::Logger logger = elt::log::CiiLogManager::GetLogger(LOGGER_NAME);
```

(continues on next page)

(continued from previous page)

```
    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. "hellocii.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 use the GetLogger() function or to create the logger once and store it in a member attribute.
- The *RAD_ASSERT* macros use the rootLogger.

6.4.10 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 creating the application runtime and configuration *data*: the *DataContext* object. Note that the *DataContext* class is composed of the *Config* (see *config.hpp|cpp*) and *OldbInterface* (see *oldbInterface.hpp|cpp*) classes to allow to retrieve configuration information and to publish data to the OLDB.

It then creates the application object *app* (see *rad::Application*) and the *action_mgr* (see *actionMgr.hpp|cpp*) objects and initializes the application via the appl.Init() method. The Init() method performs the following tasks:

- parse the command line options
- load the YAML configuration file
- apply the logging properties
- publish the configuration to the OLDB
- create the object to deal with the incoming requests
- create the State Machine actions and activities objects
- load the State Machine model (and the extension if provided)
- register the application state publisher to the OLDB

```
int main(int argc, char *argv[]) {
  /*
```

(continues on next page)

(continued from previous page)

```
 * Initialize logging via constructor.
 */
rad::LogInitializer log_initializer;

try {
    /*
     * Create data context which includes the configuration information.
     */
    hellocii::DataContext data;

    /*
     * Create state machine based application object.
     */
    rad::Application appl(hellocii::CONFIG_DEFAULT_MODNAME, data);

    /*
     * Create actions and activities invoked by the state machine.
     */
    hellocii::ActionMgr action_mgr(data);
    if (appl.Init(argc, argv, action_mgr) == false) {
        return EXIT_SUCCESS;  // request for help
    }
    ...
```

At this point the application specific default reject handlers and the supported MAL interfaces are registered.

The reject handlers allow to reply with a reject message every time the associated command is received in a state that has no valid transition to deal with the command. For example, from the State Machine model (see *sm.xml*) of this example application it is possible to verify that the LoadConfig is accepted only in the On/NotOperational/NotReady state. Without the reject handler, if the LoadConfig is received in any other state than On/NotOperational/NotReady, it would be ignored (with a log message but without any notification to the originator of the command).

For this example application only two MAL inetrfaces are registered: the one to deal with ELT Standard Interface (*rad::StdCmdsImpl*) and the one for the RAD Application Interface (*rad::AppCmdsImpl*).

```
/*
 * Application specific customizations:
 * - reject event handlers
 * - MAL RPC interfaces
 */
appl.RegisterDefaultRequestRejectHandler<EventsStd::Init>();
appl.RegisterDefaultRequestRejectHandler<EventsStd::Enable>();
appl.RegisterDefaultRequestRejectHandler<EventsStd::Disable>();
```

(continues on next page)

(continued from previous page)

```
appl.RegisterDefaultRequestRejectHandler<EventsApp::LoadConfig>();

appl.RegisterService<stdif::AsyncStdCmds>("StdCmds",
        std::make_shared<rad::StdCmdsImpl>(appl));
appl.RegisterService<appif::AsyncAppCmds>("AppCmds",
        std::make_shared<rad::AppCmdsImpl>(appl, action_mgr));
```

The last part is dedicated to start the asynchronous OLDB writer and run the application. The rad::Application::Run() methods starts the State Machine interpreter and the BOOST ASIO event loop.

```
    /*
     * Start OLDB async writer thread.
     */
    data.GetOldbAsyncWriter().StartWriter();

    /*
     * Start State Machine interpreter and the
     * event loop.
     */
    appl.Run();

    /*
     * Stop OLDB async writer thread.
     */
    data.GetOldbAsyncWriter().StopWriter();
 } catch (std::exception& e) {
    LOG4CPLUS_ERROR(hellocii::GetLogger(), e.what());
    return EXIT_FAILURE;
 } catch (...) {
    LOG4CPLUS_ERROR(hellocii::GetLogger(), boost::current_exception_diagnostic_
↪information());
    return EXIT_FAILURE;
 }
 return EXIT_SUCCESS;
}
```

Note: The Run() method returns only when there are no more callbacks registered in BOOST ASIO or when it is stopped (e.g. with CTRL-C signal or via the Exit command).

6.5 Build and Install CII Generated Modules

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

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

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

6.6 CII Applications Execution

In order to execute the generated application, the CII services must be started first. Since starting and stopping the CII services require root permission, the eltdev user should be added in the /etc/sudoers file with the entries to execute the *cii-services start all* and *cii-services stop all* commands.

To start the CII services as eltdev:

```
eltdev> sudo cii-services start all
```

Note: It is possible to monitor the status of the CII services via the commands *cii-services info* and *cii-services status*. To verify that the CII OLDB is working one can start the *oldbGui* panel.

After the CII services have been started, the generated CII application can be executed in a dedicated terminal:

```
> helloCii -c config/hellocii/config.yaml -l DEBUG
```

Note: The first time application is started, CII OLDB ERROR/WARN messages may be logged to the terminal. This problem has been reported with ticket ECII-497[15].

The application state can be queried by running on a different terminal the following command:

```
> msgsend -u zpb.rr://127.0.0.1:12081/StdCmds ::stdif::StdCmds::GetState
```

The default application command line options are as follow:

[15] https://jira.eso.org/browse/ECII-497

```
-h [ --help ] Print help messages
-n [ --proc-name ] arg Process name
-l [ --log-level ] arg Log level: ERROR, WARN, INFO, DEBUG, TRACE
-c [ --config ] arg Configuration filename
-o [ --oldb-prefix ] arg OLDB URI prefix
```

Note:

- Make sure that the *CFGPATH* environment variable contains the path(s) where the configuration files are located and that the directory and files exist.
- Make sure that the *CII_LOGS* environment variable is defined with the path where the log file will be located and that the directory exists.

To terminate the application it is enough to send an Exit command (or press Ctrl-C in the application's terminal):

```
> msgsend -u zpb.rr://127.0.0.1:12081/StdCmds ::stdif::StdCmds::Exit
```

6.7 CII Applications Debugging with Eclipse

For each waf project it is possible to create an Eclipse C/C++ project via the following command:

```
> cd hello
> waf eclipse
```

From a terminal Eclipse can be started and the project imported via:

- From the "File" menu select the "Import" option
- Select "Existing Projects into Workspace"
- Click on "Next" button
- Select the "hello" root directory using the "Browse" button
- Click on "Finish" button

Create a Debugging Configuration for the hellocii application:

- From the "Run" menu select the "Debug Configurations. . . " option
- Right click on "C/C++ Application" and select "New Configuration"
- Enter Name = hellocii
- In the "Main" tab enter: Project = hello
- In the "Main" tab enter: C/C++ Application = /home/landolfa/EELT/TUTORIAL/hello/build/hellocii/hellocii

- In the “Arguments” tab enter: Program arguments = -l DEBUG -c config/hellocii/config.yaml
- Click on “Debug” button to start debugging

6.8 Unit Tests Execution

An example of unit test for the class ActionMgr is generated by the template in the hellocii/test directory. In order to execute the Unit Tests:

```
> cd hello
> waf test
```

To force the re-execution of all unit tests:

```
> waf test --alltests
```

To run the unit tests with valgrind to detect memory leaks:

```
> waf test --alltests --valgrind
```

6.9 Generate CII Integration Test Module

RAD provides templates to generate some basic integration tests based on Robot Framework[16]. The tests verify the “standard” commands and check for 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/template/resource/template/rad-robtpl-ciitest/

module_name [hellociitest]: hellociitest
module_to_test [hellocii]: hellocii
application_to_test [hellocii]: helloCii
interface_prefix [hellociiif]: hellociiif
application_to_send [msgsend]: msgsend
```

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.

[16] https://robotframework.org/

- *application_to_send:* the name of the application to use in the tests to send commands. By default it is the python msgsend one installed in the ELT DevEnv.

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

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

6.10 Execute CII Integration Tests

Integration tests can be executed via Extensible Test Runner (ETR) tool (see ETR User Manual[17]) or directly using Robot Framework.

In the first case:

```
> cd hellociitest
> etr
```

Note: ETR should be part of the ELT DevEnv or it should be installed from RPM. It is possible to run the integration tests without ETR using directly the Robot Framework tool (e.g.: cd hellociitest/src/; robot genStdcmds.robot)

Using Robot directly:

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

6.11 Doxygen Documentation Generation

In order to generate the doxygen documentation:

```
> cd hello
> waf --with-docs
```

The generated html files are in hello/build/docs directory.

[17] 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** hellociiif/src/hellociiif.xml (CII Interface Module)
- **create** hellocii/src/events.rad.ev (CII Application Module)
- **create** hellocii/src/cmdsImpl.hpp (CII Application Module)
- **update** hellocii/resource/config/hellocii/sm.xml (CII Application Module)
- **create** hellocii/src/actionsPreset.hpp|cpp (CII Application Module)
- **update** hellocii/src/actionMgr.cpp (CII Application Module)
- **update** hellocii/src/main.cpp (CII Application Module)

7.1.1 Update CII Interface Module

If a command is added (modified, or removed), the MAL interface (*hellociiif/src/hellociiif.xml*) has to be edited.

For example, in order to introduce a new PresetCmds interface with a Preset command that takes 2 parameters (e.g. ra and dec) and that can return an exception, the following XML should be added:

```
...
<exception name="ExceptionErr">
  <member name="desc" type="string"/>
  <member name="code" type="int32_t"/>
</exception>

<interface name="PresetCmds">
  <method name="Preset" returnType="string" throws="ExceptionErr">
    <argument name="pos" type="nonBasic" nonBasicTypeName="TelPosition" />
  </method>
</interface>
...
```

7.1.2 Update CII Application Module

7.1.3 Create events.rad.ev

In order to be able to process the Preset command with the State Machine, the event associated to command should be added to a .rad.ev file.

If the .rad.ev file is not available, it has to be first to be created:

```
> cd hello/hellocii/src
> touch events.rad.ev
```

and then the following content copied:

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

namespace: Events

includes:
   - boost/exception_ptr.hpp
   - rad/mal/request.hpp
   - Hellociiif.hpp

events:
    Preset:
          doc: event triggered when the Preset command is received.
          payload: rad::cii::Request<std::string, std::shared_ptr<hellociiif::TelPosition>>
```

For each event it is possible to specify:

- an optional description and
- the 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 provide access to the command payload and to set the replay payload. The wrapper class takes two parameters:

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

In this case the Preset event has std::string reply and std::shared_ptr<hellociiif::TelPosition> command parameter.

Similarly to the *eventsStd.rad.ev* and *eventsApp.rad.ev*, this file will be transformed during compilation by *radgen* tool into events.rad.cpp and events.rad.hpp files containing the C++ classes associated to the events. The generated C++ files are located in *hello/build/hellocii/src* directory and look like:

```
namespace Events {
...
/**
 * event triggered when the Preset command is received.
 */
class Preset final : public rad::AnyEvent {
 public:
  static constexpr char const* ID = "Events.Preset";
...
```

7.1.4 Create cmdsImpl.hpp

The *CmdsImpl* 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.

If the class is not available it has first to be created:

```
> cd hello/hellocii/src
> touch cmdsImpl.hpp
```

and then the following content copied:

```
#ifndef HELLOCIISERVER_CMDS_IMPL_HPP
#define HELLOCIISERVER_CMDS_IMPL_HPP

#include "events.rad.hpp"
#include "logger.hpp"
#include <rad/exceptions.hpp>
#include <rad/smAdapter.hpp>

namespace hellocii {

class CmdsImpl : public hellociiif::AsyncPresetCmds {
public:
  explicit CmdsImpl(rad::SMAdapter& sm) : m_sm(sm) {
     RAD_TRACE(GetLogger());
  }

  virtual ~CmdsImpl() {
     RAD_TRACE(GetLogger());
  }

  virtual elt::mal::future<std::string> Preset(const std::shared_ptr<hellociiif::TelPosition>& pos)␣
```

(continues on next page)

(continued from previous page)

```
↪override {
    RAD_TRACE(GetLogger());
    auto ev = std::make_shared<Events::Preset>(pos->clone());
    m_sm.PostEvent(ev);
    return ev->GetPayload().GetReplyFuture();
  }

private:
  rad::SMAdapter& m_sm;
};

}  // namespace hellociiserver
#endif  // HELLOCIISERVER_CMDS_IMPL_HPP
```

Similarly to the *rad::StdCmdsImpl* and *rad::AppCmdsImpl*, the *CmdsImpl* class realizes the corresponding hellociiif::AsyncPresetCmds MAL interface by implementing the CmdsImpl::Preset() method. The method creates the corresponding Events::Preset event and injects it into the State Machine. The event takes as payload a copy of the command's argument (hellociiif::TelPosition>& pos).

7.1.4.1 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 it can be reached once the system has been initialized (e.g. from *On/Operation/Idle*). The resulting State Machine model in *hellocii/resource/config/hellocii/sm.xml* should look like:

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

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

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

(continues on next page)

(continued from previous page)

```
  ...
</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.

7.1.4.2 Create actionsPreset.hpp|cpp

The new action ActionsPreset.Start added to the State Machine model can be implemented by by adding a new method to an existing *rad::ConfigurableActionGroup* class or by creating a dedicated a new one. To create a new one:

```
> cd hello/hellocii/src
> touch actionsPreset.hpp
> touch actionsPreset.cpp
```

The header should look like:

```
#ifndef HELLOCII_ACTION_PRESET_HPP
#define HELLOCII_ACTION_PRESET_HPP

#include <rad/configurableActionGroup.hpp>
#include <rad/application.hpp>

namespace hellocii {

class DataContext;

/**
 * This class contains the implementation of the actions dealing with
 * the Preset use case.
 */
class ActionsPreset : public rad::ConfigurableActionGroup {
 public:
  /**
   * Constructor.
   *
   * @param[in] sm Reference to the SM Adapter used to inject internal events.
   * @param[in] data Data shared within the application among actions and activities.
   */
  ActionsPreset(rad::Application& sm, DataContext& data);
```

(continues on next page)

(continued from previous page)

```
  /**
   * Method invoked when Init command is received to re-initialize
   * the actions class attributes.
   */
  void Initialize() override;

  /**
   * Method invoked when SetConfig or LoadConfig command is received
   * to re-configure the actions class attributes.
   *
   * @param keys Modified parameters. Empty vector means all params have changed.
   */
  void Configure(const std::vector<std::string>& keys) override;

  void Start(scxml4cpp::Context* c);

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

 private:
  rad::Application&        m_appl;
  DataContext&             m_data;
};

}  // namespace  hellocii

#endif  // HELLOCII_PRESET_STD_HPP
```

The source should look like:

```
#include "actionsPreset.hpp"
#include "dataContext.hpp"
#include "logger.hpp"
#include <events.rad.hpp>

#include <rad/mal/request.hpp>
#include <rad/smEvent.hpp>


namespace hellocii {

ActionsPreset::ActionsPreset(rad::Application& appl, DataContext& data)
```

(continues on next page)

(continued from previous page)

```
  : rad::ConfigurableActionGroup("ActionsPreset"),
    m_appl(appl),
    m_data(data) {
  RAD_TRACE(GetLogger());
}

void ActionsPreset::Initialize() {
  RAD_TRACE(GetLogger());
}

void ActionsPreset::Configure(const std::vector<std::string>& keys) {
  RAD_TRACE(GetLogger());
}

void ActionsPreset::Start(scxml4cpp::Context* c) {
  RAD_TRACE(GetLogger());

  const Events::Preset::payload_t* req = rad::GetLastEventPayloadNothrow<Events::Preset>(c);
  if (req == nullptr) {
      LOG4CPLUS_ERROR(GetLogger(), "Preset event has no associated request!");
      return;
  }

  std::shared_ptr<hellociiif::TelPosition> req_params = req->GetRequestPayload();
  float ra = req_params->getRa();
  float dec = req_params->getDec();
  LOG4CPLUS_DEBUG(GetLogger(), "Received Preset to RA " << ra << " DEC " << dec);

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

} // namespace hellocii
```

The Start action implementation logs the RA/DEC and replies back to the originator of the Preset command the message “Preset Started”.

7.1.4.3 Update actionMgr.cpp

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

```
#include "actionsPreset.hpp"
...
void ActionMgr::CreateActions(rad::Application& appl) {
...
    ActionsPreset* actions_preset = new ActionsPreset(appl, m_data);
    if (actions_preset == nullptr) {
        LOG4CPLUS_ERROR(GetLogger(), "Cannot create ActionsPreset object.");
        return;
    }
    AddActionGroup(actions_preset);
...
    RegisterAction<ActionsPreset>("ActionsPreset.Start",
        &ActionsPreset::Start, actions_preset);
...
}
```

The code first create the ActionsPreset object that groups all the actions related to the Preset use case and then register the ActionsPreset::Start method with the name “ActionsPreset.Start”. In this way the State Machine interpreter can invoke the correct C++ method when interpreting the “ActionsPreset.Start” statement in the SCXML State Machine model.

Note: If the name of an action in the SCXML State Machine model does not match the ones registered in the ActionMgr, the SCXML parser will log a warning.

7.1.4.4 Update main.cpp

Since we added a new MAL interface (i.e. PresetCmds), it has to be registered by updating the *hellocii/src/main.cpp*:

```
#include "cmdsImpl.hpp"
...
int main(int argc, char *argv[]) {
  try {
    ...
    appl.RegisterDefaultRequestRejectHandler<Events::Preset>();
    ...
```

(continues on next page)

(continued from previous page)

```
    appl.RegisterService<hellociiif::AsyncPresetCmds>("PresetCmds",
        std::make_shared<hellocii::CmdsImpl>(appl));
```

Also the default reject handler for the Preset command has been added.

7.2 Add an Activity

After having added the Preset command (see *Add a Command*), we introduce an activity that is started after entering the *Presetting* state. This activity simulates the moving of the 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** hellocii/resource/config/hellocii/log.properties (CII Application Module)
- **update** hellocii/src/events.rad.ev (CII Application Module)
- **update** hellocii/resource/config/hellocii/sm.xml (CII Application Module)
- **create** hellocii/src/activityMoving.hpp|cpp (CII Application Module)
- **update** hellocii/src/actionMgr.cpp (CII Application Module)
- **update** hellocii/test/testActionMgr.cpp (CII Application Module)

7.2.1 Update CII Application Module

7.2.1.1 Update log.properties

A dedicate logger for the activity ActivityMoving is added to *hellocii/resource/config/hellocii/log.properties* file and configured as follows:

```
...
log4cplus.logger.hellocii.ActivityMoving=DEBUG
...
```

7.2.1.2 Update events.rad.ev

The MoveDone and MoveError events (without payload) are added in *hellocii/src/events.rad.ev* file to indicate that the activity has terminated or an error has occurred while moving:

```
events:
    ...
    MoveDone:
```

(continues on next page)

(continued from previous page)

```
        doc: event triggered when the ActivityMoving has terminated.
    MoveError:
        doc: event triggered when an error occurs while moving.
    ...
```

7.2.1.3 Update sm.xml

The State Machine model is updated with the invocation of the activity *ActivityMoving* and the new transitions from *Presetting* to *Idle* on event *MoveDone* and from *Presetting* to *NotOperation/NotReady* in case of errors:

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

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

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

    <invoke id="ActivityMoving"/>
  </state>
  <transition event="Events.MoveDone" target="On::Operational::Idle"/>
  <transition event="Events.MoveError" target="On::NotOperational::NotReady"/>
  ...
</state>
...
```

7.2.1.4 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.

```
> cd hello/hellocii/src
> touch activityMoving.hpp
> touch activityMoving.cpp
```

The class can be implemented with the following code:

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

namespace hellocii {

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 hellocii
```

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

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

#include <rad/mal/publisher.hpp>

namespace hellocii {

ActivityMoving::ActivityMoving(const std::string& id,
                               rad::SMAdapter& sm,
```

(continues on next page)

(continued from previous page)

```
                        DataContext& data)
            : rad::ThreadActivity(id),
              m_sm(sm),
              m_data(data) {
}

ActivityMoving::~ActivityMoving() {
}

void ActivityMoving::Run() {
    RAD_TRACE(m_logger);

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

7.2.1.5 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::Application& appl) {
    rad::ThreadActivity* the_activity = nullptr;
    the_activity = new ActivityMoving("ActivityMoving", appl, m_data);
    AddActivity(the_activity);
}
```

7.2.1.6 Update testActionMgr.cpp

Since an new action and a new activity has been added to the ActionMgr, the unit tests for the CreateActions and CreateActivities methods should be updated accordingly.

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 OLDB and therefore the class *OldbInterface* also needs to be updated.

This behavior can be achieved by updating the following files:

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

7.3.1 Update CII Application Module

7.3.1.1 Update oldbInterface.hpp|cpp

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

```
...
const std::string KEY_MON_TARGET_RA = "mon/target/ra";
const std::string KEY_MON_TARGET_DEC = "mon/target/dec";
const std::string KEY_MON_ACTUAL_RA = "mon/actual/ra";
const std::string KEY_MON_ACTUAL_DEC = "mon/actual/dec";
...

class OldbInterface {
 public:
 ...
 void SetTargetRaDec(const float ra, const float dec);
 void SetActualRaDec(const float ra, const float dec);
```

(continues on next page)

(continued from previous page)

```
 ...
}
```

```
...
void OldbInterface::SetTargetRaDec(const float ra, const float dec) {
    RAD_TRACE(GetLogger());
    SetValue<float>(KEY_MON_TARGET_RA, ra);
    SetValue<float>(KEY_MON_TARGET_DEC, dec);
}

void OldbInterface::SetActualRaDec(const float ra, const float dec) {
    RAD_TRACE(GetLogger());
    SetValue<float>(KEY_MON_ACTUAL_RA, ra);
    SetValue<float>(KEY_MON_ACTUAL_DEC, dec);
}
...
```

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);
    void SetTargetRaDec(const float ra, const float dec);

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

```
DataContext::DataContext()
: m_oldb_async_writer(std::chrono::seconds(1), std::chrono::milliseconds(100)),
  m_ra(0.0),
  m_dec(0.0) {
    RAD_TRACE(GetLogger());

    m_oldb_async_writer.SetOldbPrefix(
        m_config.GetParam<std::string>(rad::KEY_CONFIG_OLDB_URI_PREFIX) +
```

(continues on next page)

(continued from previous page)

```
        m_config.GetParam<std::string>(rad::KEY_CONFIG_MODNAME) + "/");
}

void DataContext::GetTargetRaDec(float& ra, float& dec) {
    RAD_TRACE(GetLogger());
    ra = m_ra;
    dec = m_dec;
}

void DataContext::SetTargetRaDec(const float ra, const float dec) {
    RAD_TRACE(GetLogger());
    m_ra = ra;
    m_dec = dec;
    m_oldb_interface.SetTargetRaDec(ra, dec);
}
```

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(scxml4cpp::Context* c) {
  RAD_TRACE(GetLogger());

  const Events::Preset::payload_t* req = rad::GetLastEventPayloadNothrow<Events::Preset>(c);
  if (req == nullptr) {
      LOG4CPLUS_ERROR(GetLogger(), "Preset event has no associated request!");
      return;
  }

  std::shared_ptr<hellociiif::TelPosition> req_params = req->GetRequestPayload();
  float ra = req_params->getRa();
  float dec = req_params->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

The *AcitivityMoving::Run()* method can be re-factored to take into account the real target coordinates and to publish to the OLDB the actual coordinates.

```
void ActivityMoving::Run() {
    RAD_TRACE(m_logger);

    /*
     * Retrieve target coordinates.
     */
    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) {
        /*
         * Compute actual position.
         */
        cur_ra = i * step_ra;
        cur_dec = i * step_dec;
        LOG4CPLUS_DEBUG(m_logger, "Moving ALT/AZ: RA = " << cur_ra << " DEC = " <
↪< cur_dec);

        /*
         * Published actual position.
         */
        try {
            // Update OLDB asynchronously
            m_data.GetOldbAsyncWriter().Set({{KEY_MON_ACTUAL_RA, cur_ra}, {KEY_
↪MON_ACTUAL_DEC, cur_dec}});
        } catch (const std::exception& e) {
            LOG4CPLUS_DEBUG(m_logger, e.what());
        }

        /*
```

(continues on next page)

(continued from previous page)

```
        * Check for preset completion.
        */
       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++;
    }
}
```

7.3.1.5 Adding ZPB publisher to activityMoving.cpp

The *AcitivityMoving::Run()* method can be extended to publish the actual coordiates via ZPB.

```
void ActivityMoving::Run() {
    RAD_TRACE(m_logger);

    /*
     * Create ZPB publisher
     */
    elt::mal::Mal::Properties mal_properties;
    mal_properties["zpb.ps.slowJoinerDelayMs"] =
        "100";  // small initial delay to allow pub/sub synchronization
    rad::cii::Publisher<hellociiif::TelPosition> publisher(
        elt::mal::Uri("zpb.ps://127.0.0.1:12345/TelPosition"), mal_properties);
    auto sample = publisher.CreateTopic();
    if (sample == nullptr) {
        LOG4CPLUS_ERROR(GetLogger(), "Instance publisher cannot create data entity");
        m_sm.PostEvent(rad::UniqueEvent(new Events::MoveError()));
        return;
    }

    /*
     * Retrieve target coordinates.
     */
    float target_ra = 0.0;
    float target_dec = 0.0;
    m_data.GetTargetRaDec(target_ra, target_dec);
```

(continues on next page)

(continued from previous page)

```
  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) {
    /*
     * Compute actual position.
     */
    cur_ra = i * step_ra;
    cur_dec = i * step_dec;
    LOG4CPLUS_DEBUG(m_logger, "Moving ALT/AZ: RA = " << cur_ra << " DEC = " <
↪< cur_dec);

    /*
     * Publish the actual position.
     */
    try {
      // Update OLDB asynchronously
      m_data.GetOldbAsyncWriter().Set({{KEY_MON_ACTUAL_RA, cur_ra}, {KEY_
↪MON_ACTUAL_DEC, cur_dec}});

      // Publish with ZPB
      sample->setRa(cur_ra);
      sample->setDec(cur_dec);
      publisher.Publish(*sample);
    } catch (const std::exception& e) {
      LOG4CPLUS_DEBUG(m_logger, e.what());
    }

    /*
     * Check for preset completion.
     */
    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()));
```

(continues on next page)

(continued from previous page)

```
            break;
        }
        i++;
    }
}
```

7.4 Building and Executing a Preset

The application binary can be obtained from the modified source with the following command:

```
> cd hello
> waf uninstall build install
```

Note: The uninstall is required to delete from the INTROOT the old hellociiif library and force the linking of the newly built interface library located in the hello/build/ directory.

In order to be able to process the Preset command, the application has to be in On/Operational/Idle. This can be achieved with the following commands:

```
> helloCii -c config/hellocii/config.yaml -l DEBUG&
> msgsend -u zpb.rr://127.0.0.1:12081/StdCmds ::stdif::StdCmds::Init
> msgsend -u zpb.rr://127.0.0.1:12081/StdCmds ::stdif::StdCmds::Enable
> msgsend -u zpb.rr://127.0.0.1:12081/StdCmds ::stdif::StdCmds::GetState
```

The Preset command with the RA/DEC parameters can be sent as follows:

```
> msgsend -u zpb.rr://127.0.0.1:12081/PresetCmds ::hellociiif::PresetCmds::Preset '{ "ra":"10", "dec
↪":"20" }'
> msgsend -u zpb.rr://127.0.0.1:12081/StdCmds ::stdif::StdCmds::GetState
```

In the stdout it should be visible from the log messages that the state has changed to On/Operational/Preset and the ActivityMoving has been started and it is simulating the telescope axes movement. The movement can be stopped by sending the Disable command:

```
> msgsend -u zpb.rr://127.0.0.1:12081/StdCmds ::stdif::StdCmds::Disable
```

The movement can be observed via the CII oldbGui panel or using a ZPB subscriber started on a dedicated terminal:

```
> helloCiiSub -u zpb.ps://127.0.0.1:12345 -v
```

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/template/resource/template/rad-waftpl-prj template:

```
> cookiecutter rad/rad/cpp/template/resource/template/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[18] 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:

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

```
> cd hello
> cookiecutter ../rad/rad/cpp/template/resource/template/rad-cpptpl-applif

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/template/resource/template/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/template/resource/template/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/hello/config.yaml	YAML application configuration file.
resource/config/hello/sm.xml	SCXML file with the State Machine model.
resource/config/hello/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 (`**cmdsImpl.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/template/resource/template/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 config/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[19]) 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.xml)
- 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

[19] 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

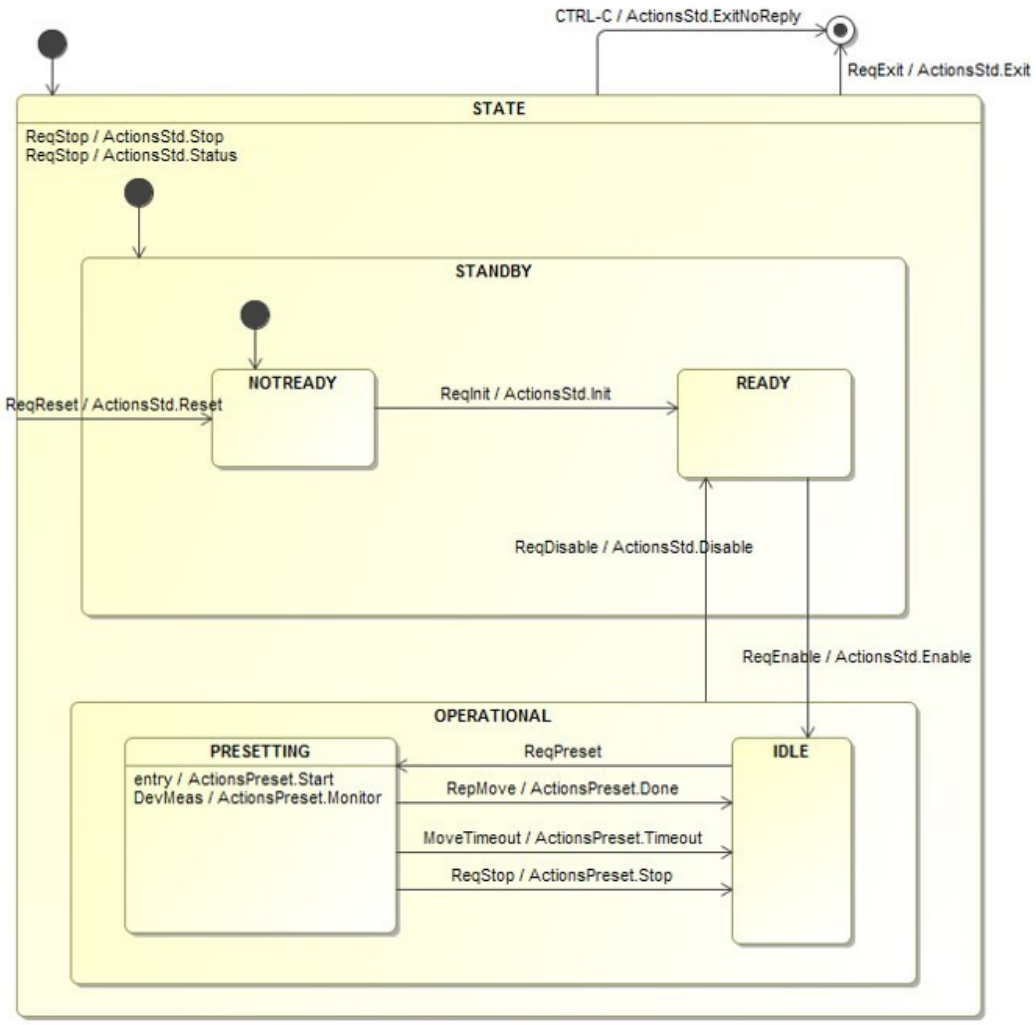
This is an example that shows how to create RAD based applications that uses request/reply, pub/sub, timers, Linux signals. It uses the interface defined in exif interface module and can be controlled by sending the commands via the exSend application (see exsend module).

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

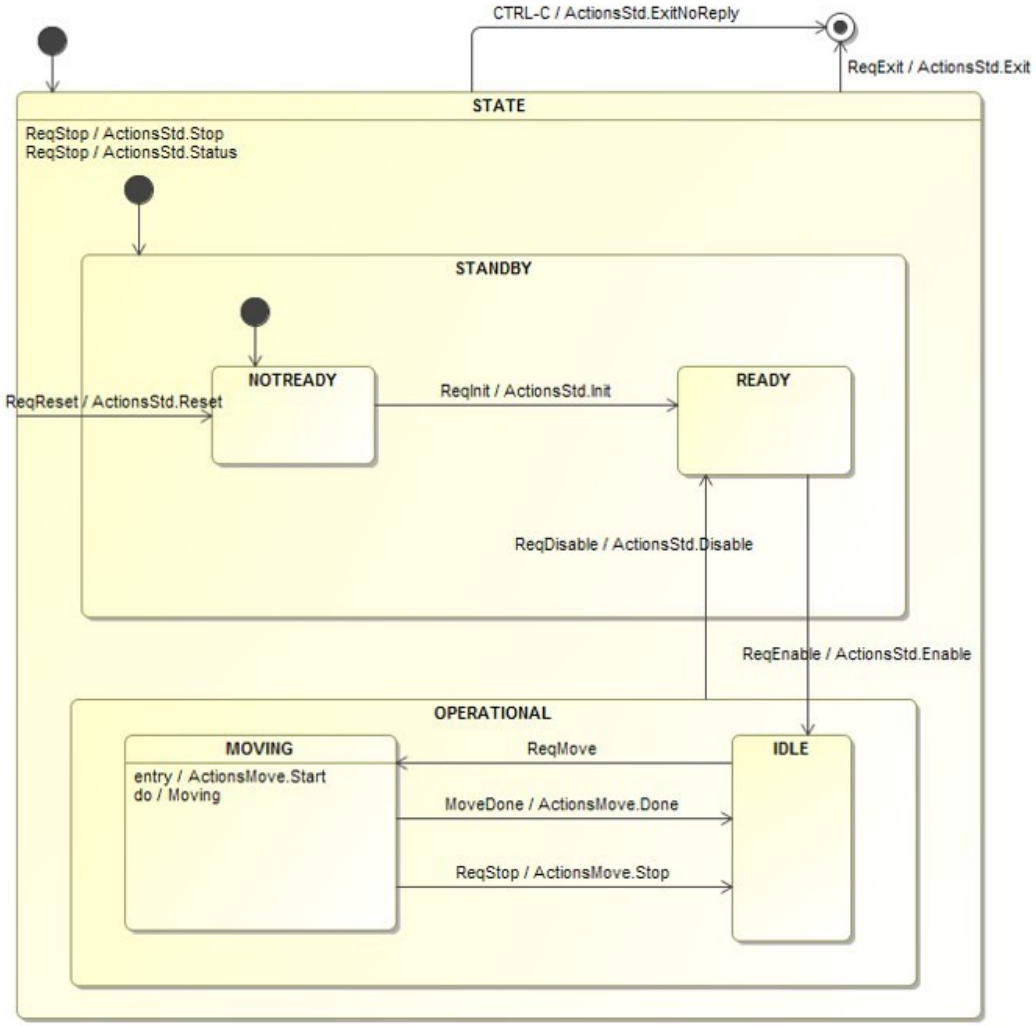
server/config/radServer/config.yaml server/config/radServer/sm.xml

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.

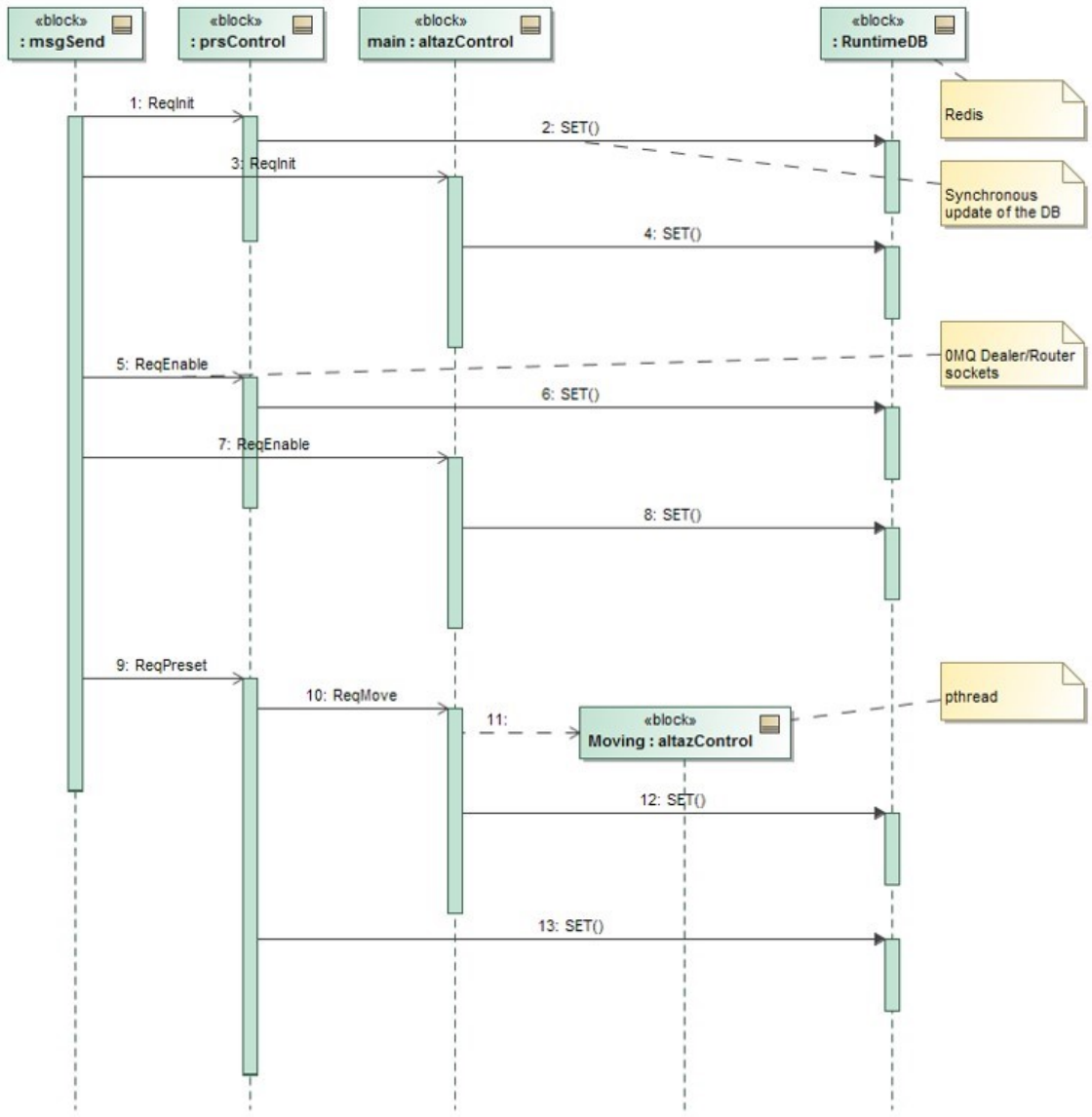


The second application (altazControl) is configured using config1.yaml and sm1.xml files. It receives the exif.ReqMove command and executes the ActionsMove::Start action and starts a do-activity: the ActivityMoving thread. The thread simulates the movement of the axes and publishes the intermediate positions 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.

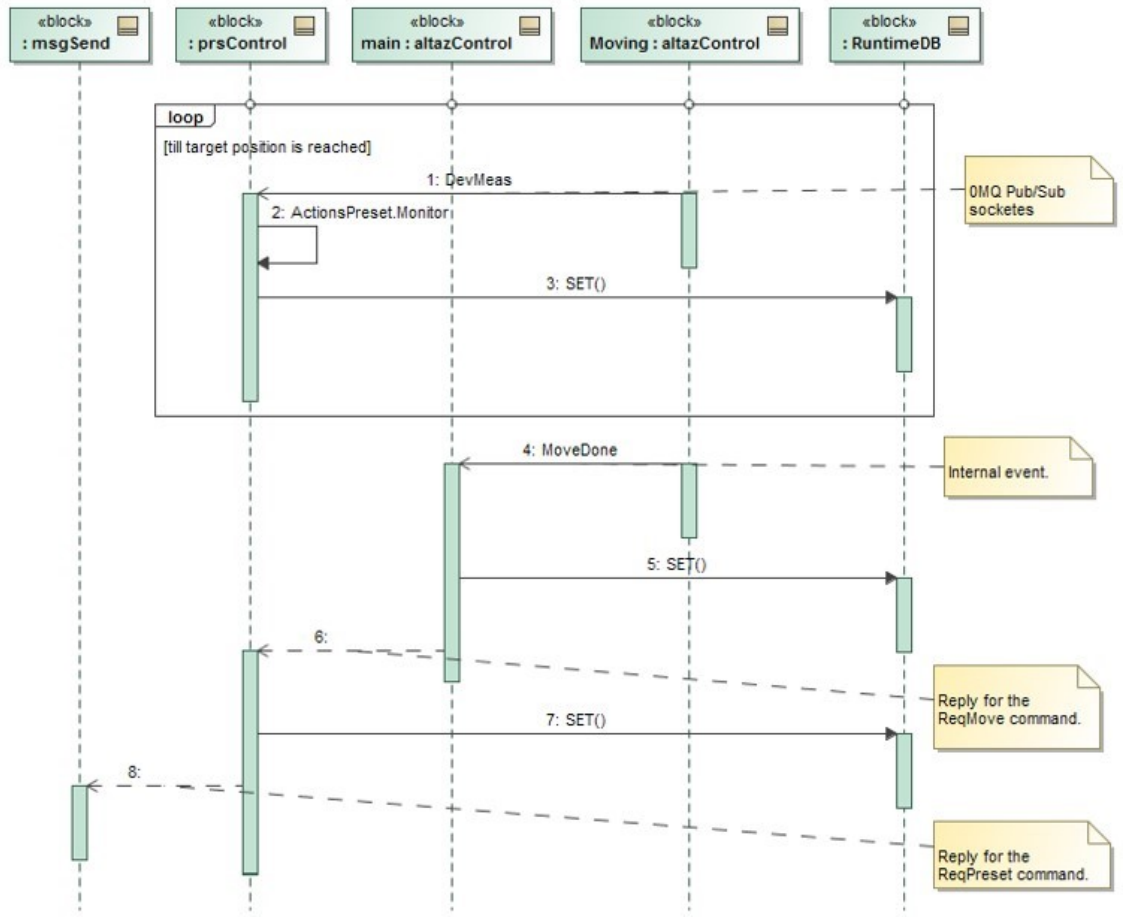


Note that both applications store the configuration, status, and telescope position information in the Redis runtime DB.

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 (hellorad) can talk to a C++ server (radServer). 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 redis-server:

```
redis-server --port 6383 &
```

then start the hellorad server application:

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

and then 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 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. If transformations have to be applied to SysML models, the SysML plug-in must be selected and loaded.

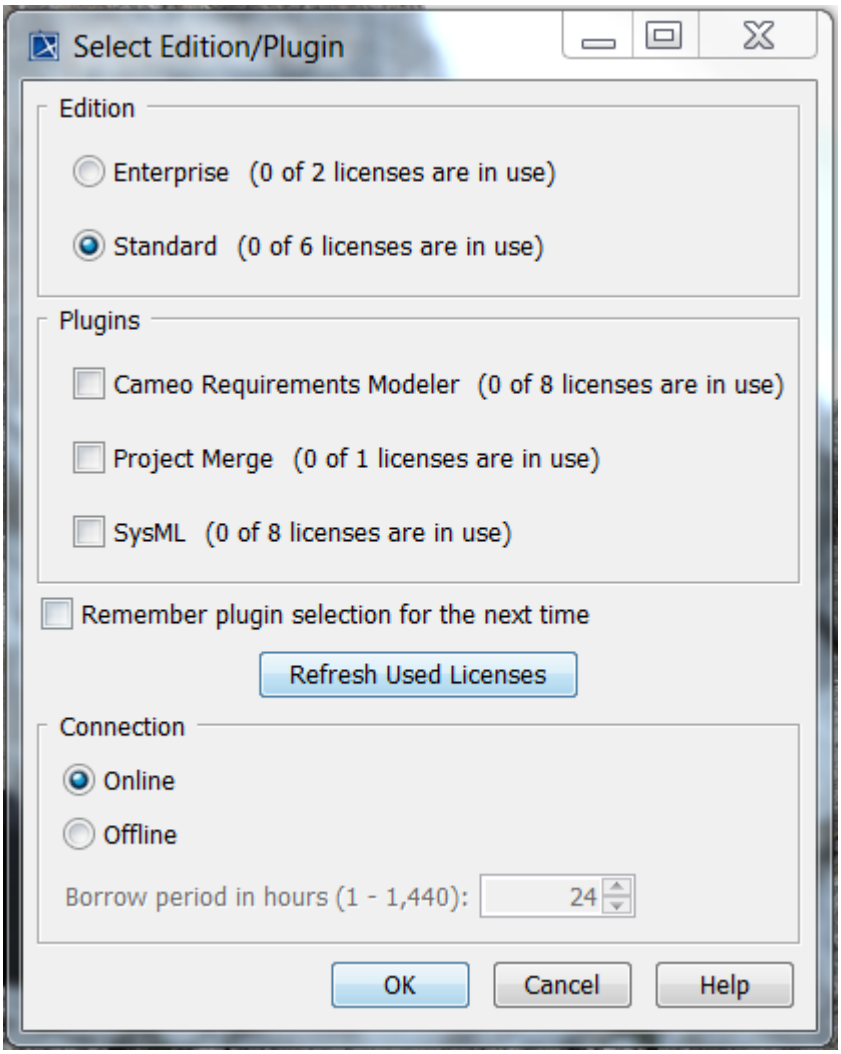


Figure 1 – Select edition and plug-ins dialog.

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.

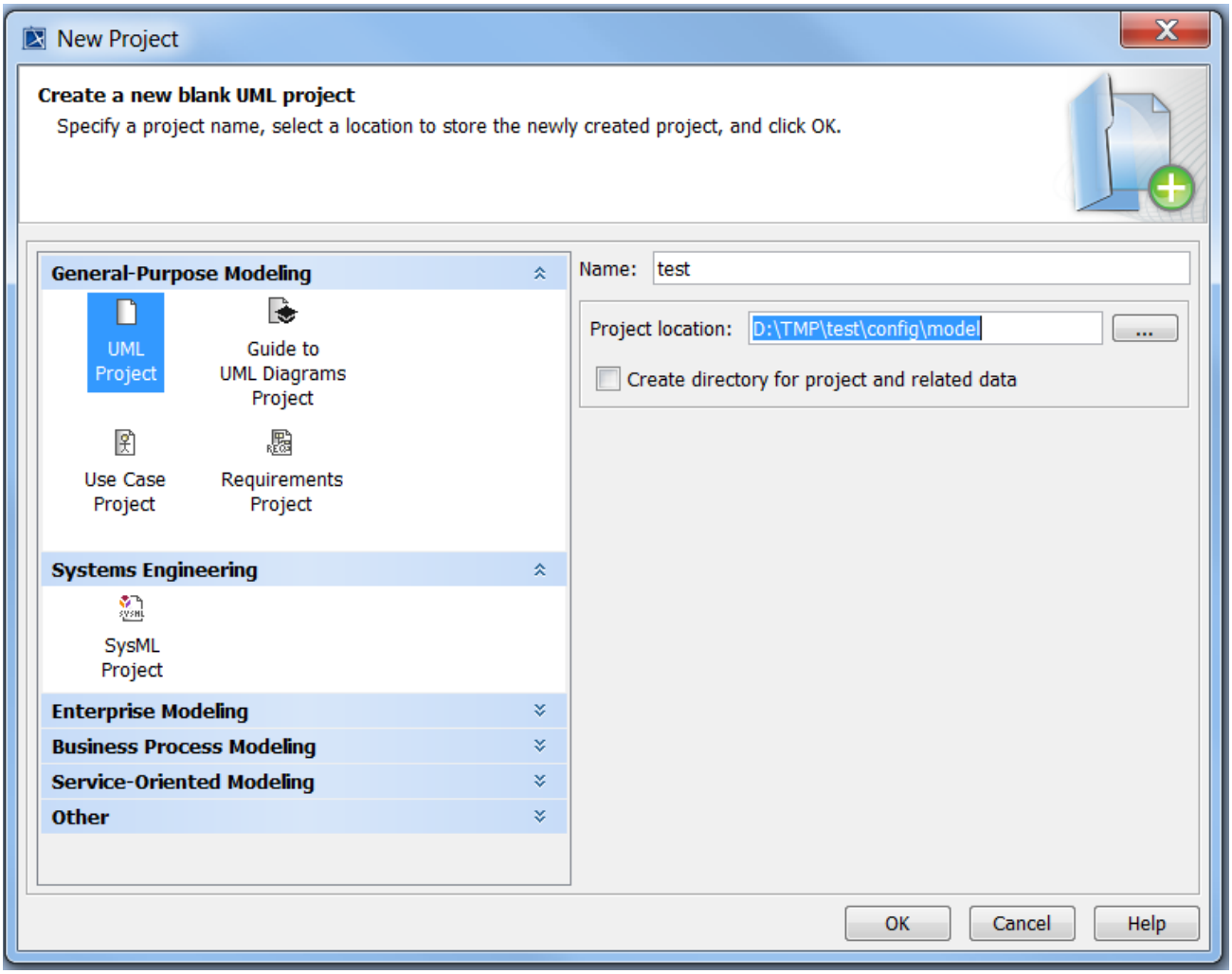
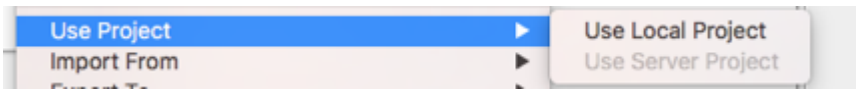


Figure 2 - Creating a new project.

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).

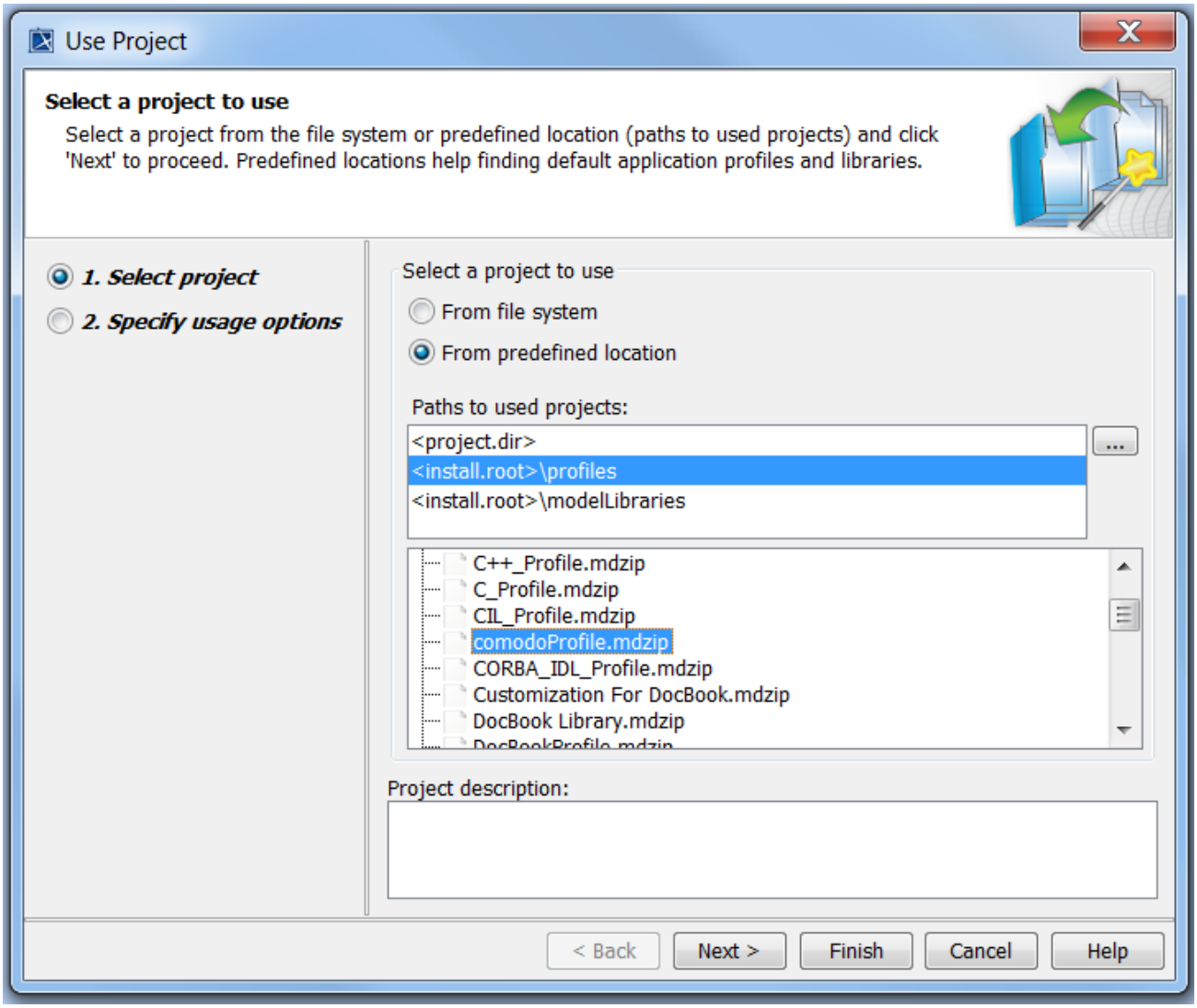


Figure 3 - Adding comodoProfile to a project.

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.

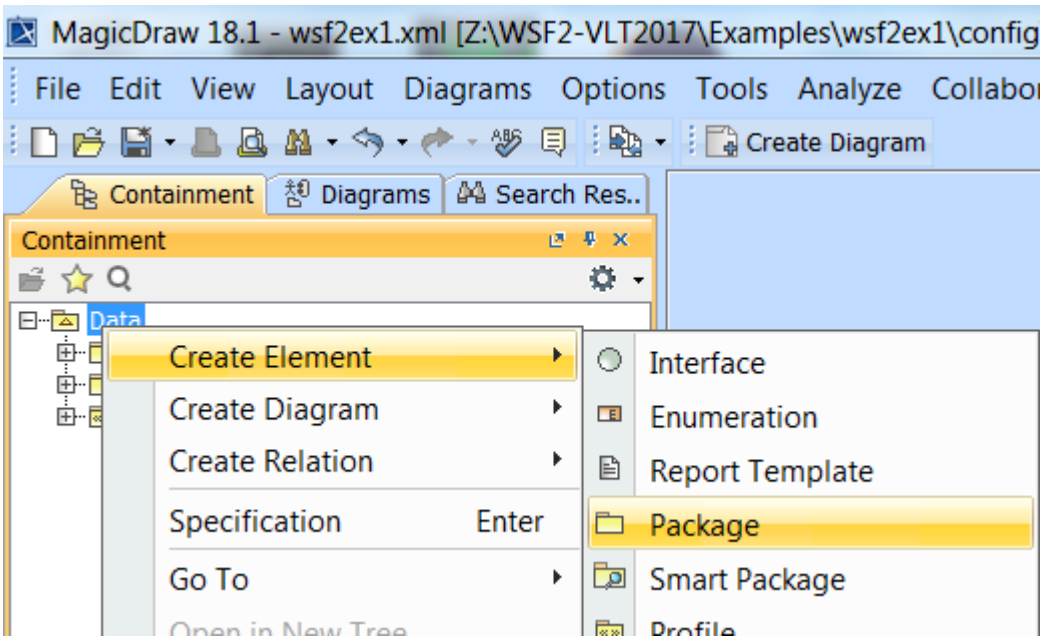


Figure 4 – Creating a Package.

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.

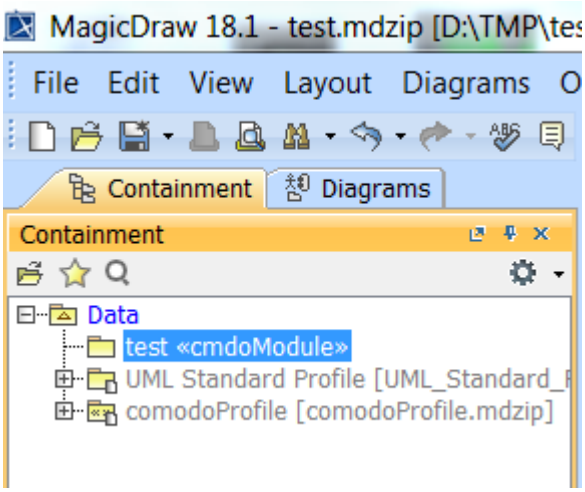


Figure 5 – Applying <<cmdoModule>> stereotype.

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.

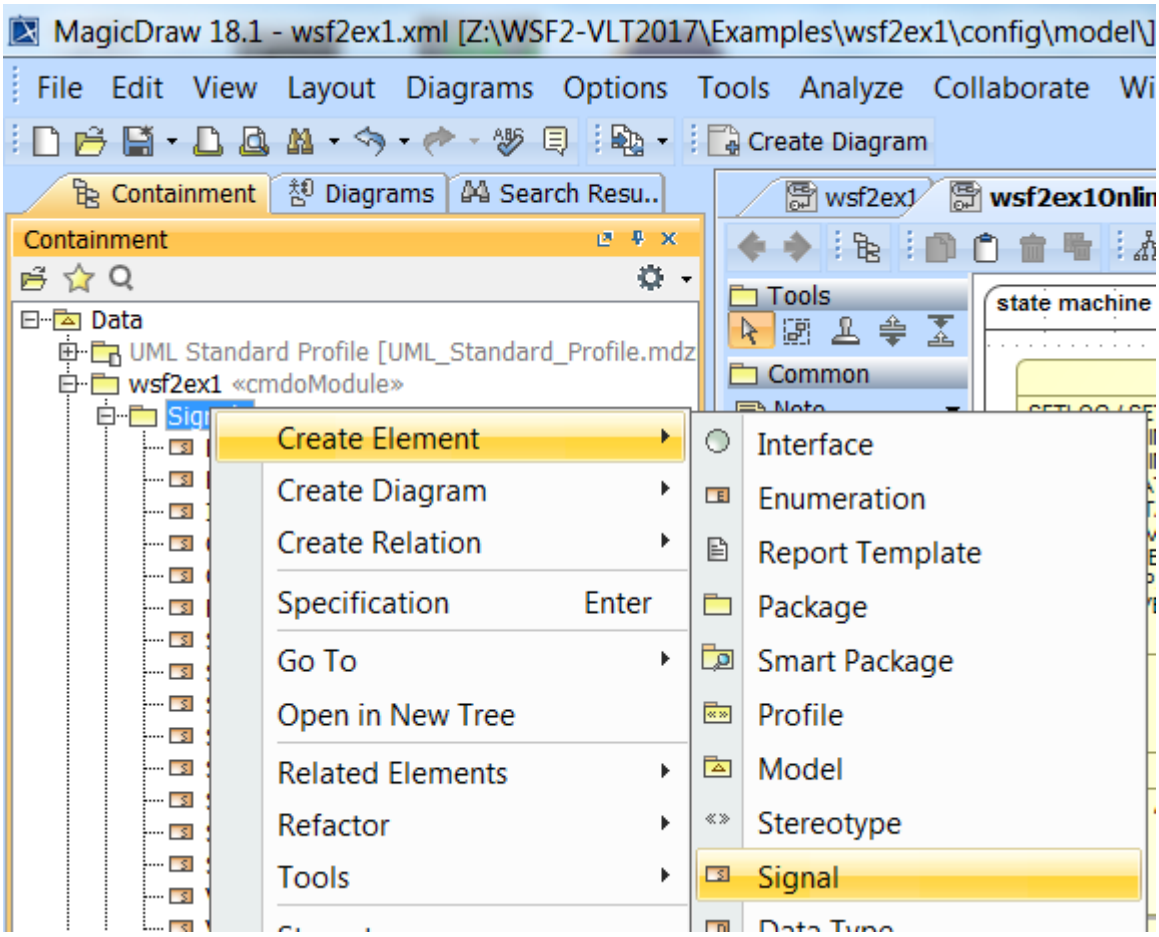


Figure 6 - Creating a new signal.

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).

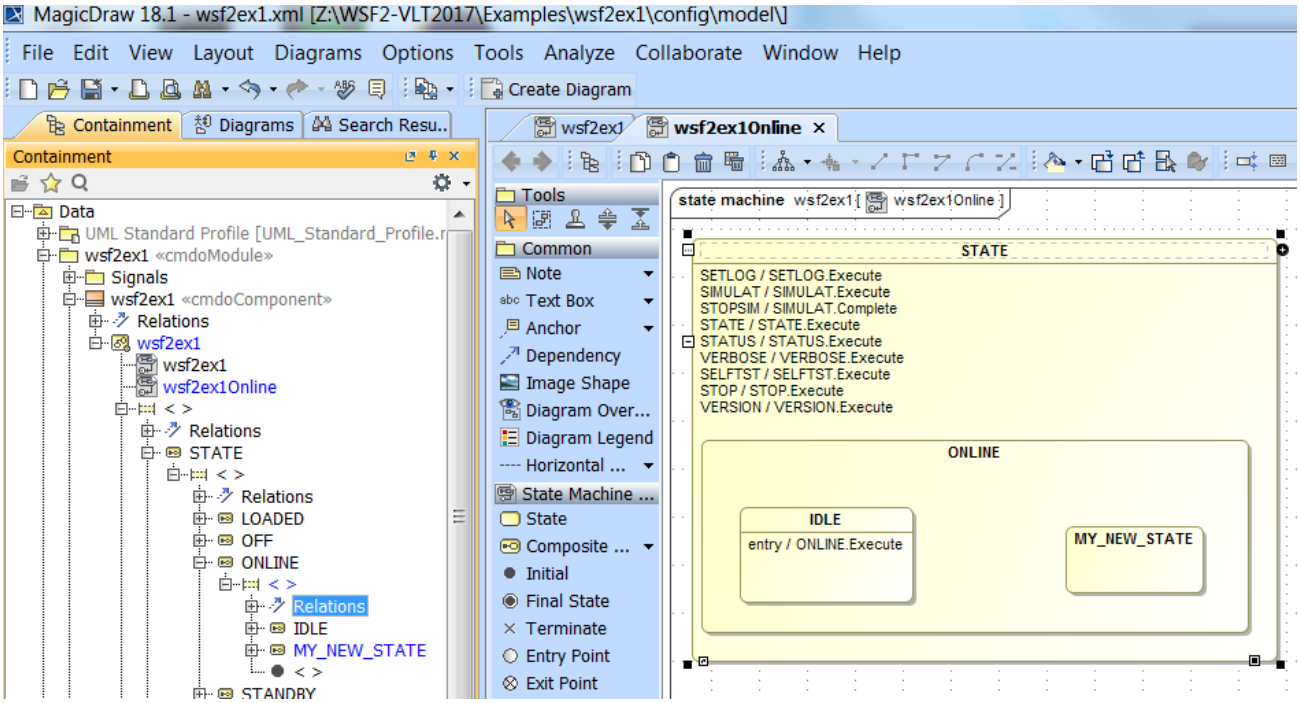


Figure 7 - New State Machine diagram and new sub-state.

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.

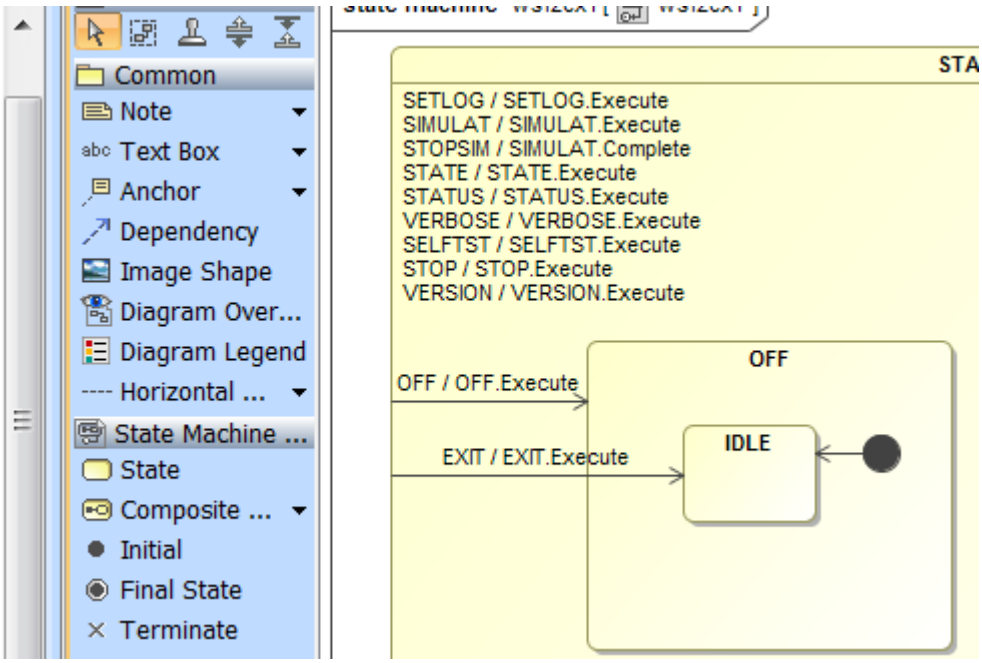


Figure 8 - Initial pseudo-state.

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.

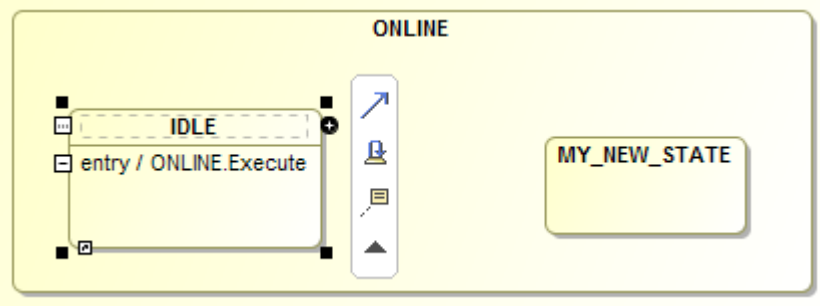


Figure 9 - Creating a transition between IDLE and MY_NEW_STATE state.

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”.

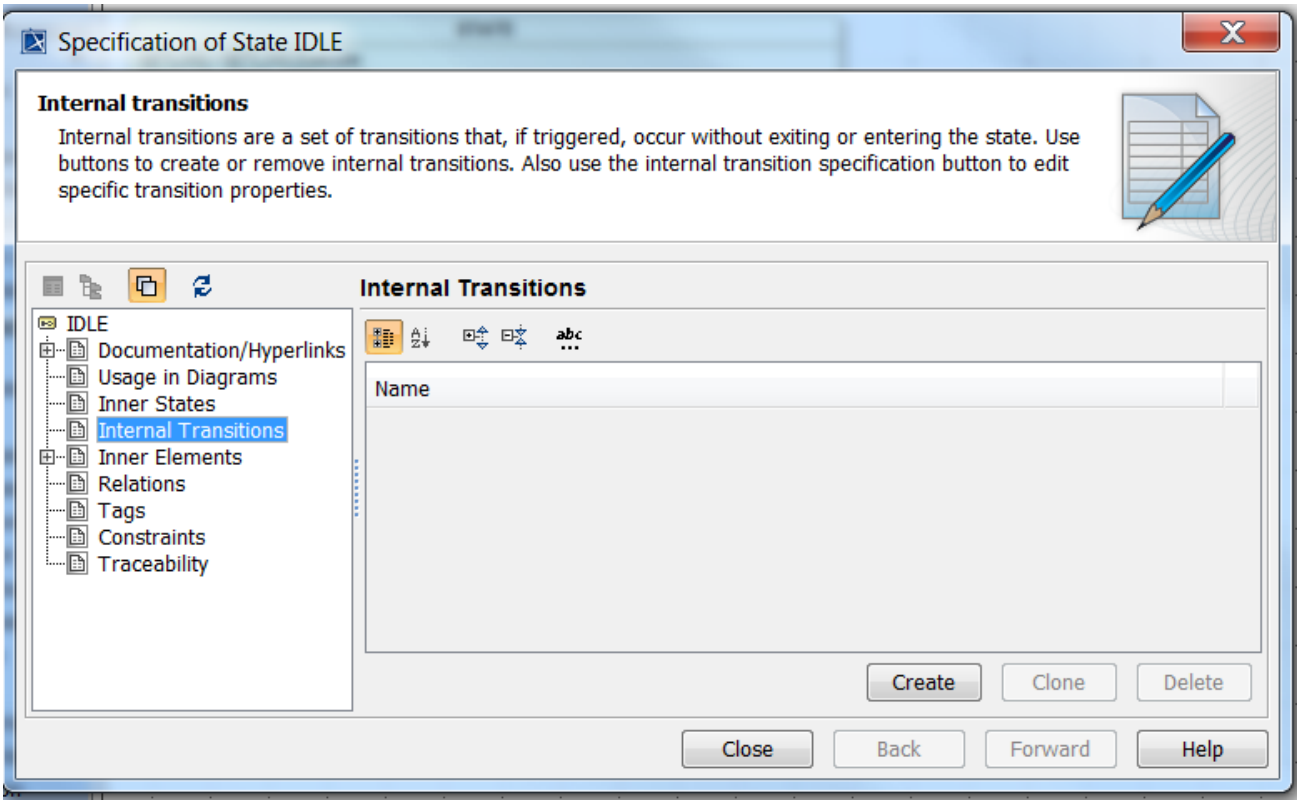


Figure 10 - Creation of internal transition for ONLINE/IDLE state.

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.

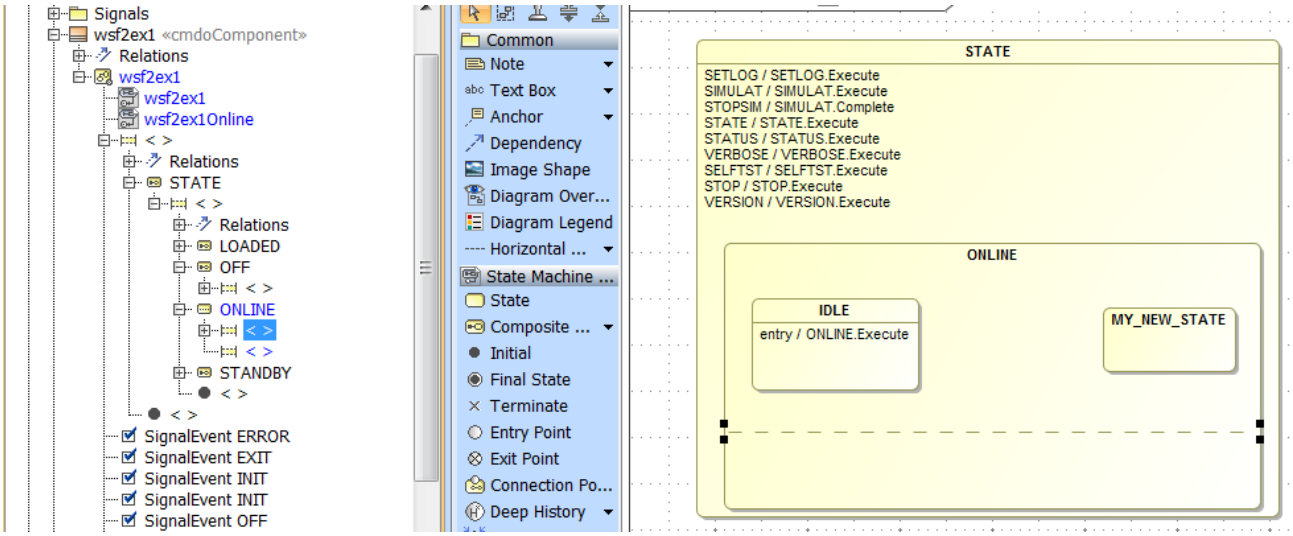


Figure 11 - Adding an orthogonal region to the ONLINE state.

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.

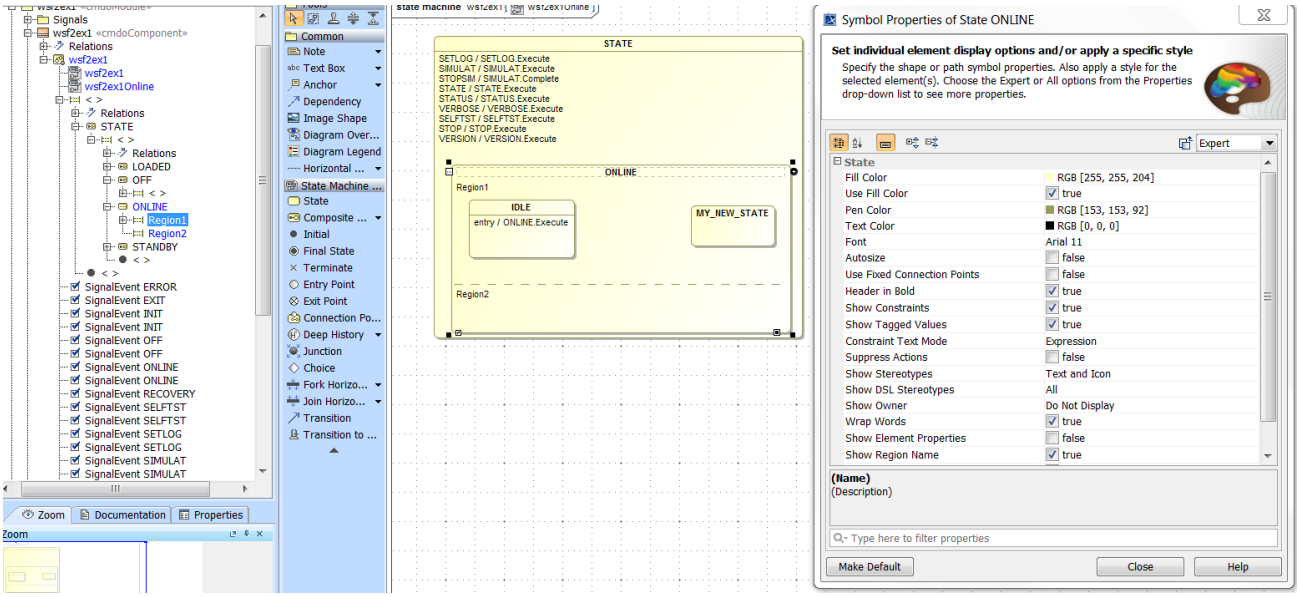


Figure 12 - Displaying the regions name of state ONLINE.

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.

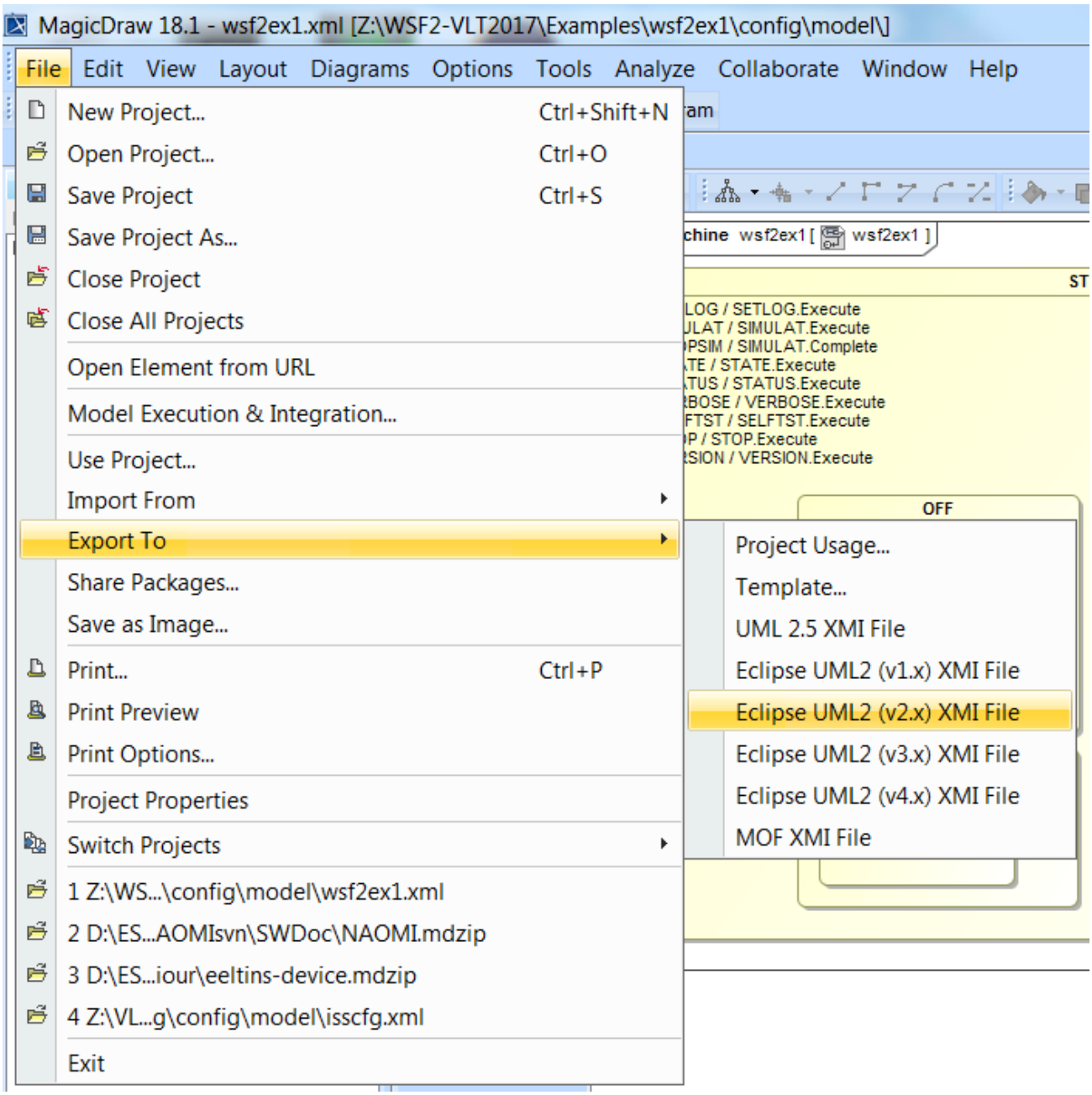


Figure 13 - Exporting to EMF XMI format.

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.

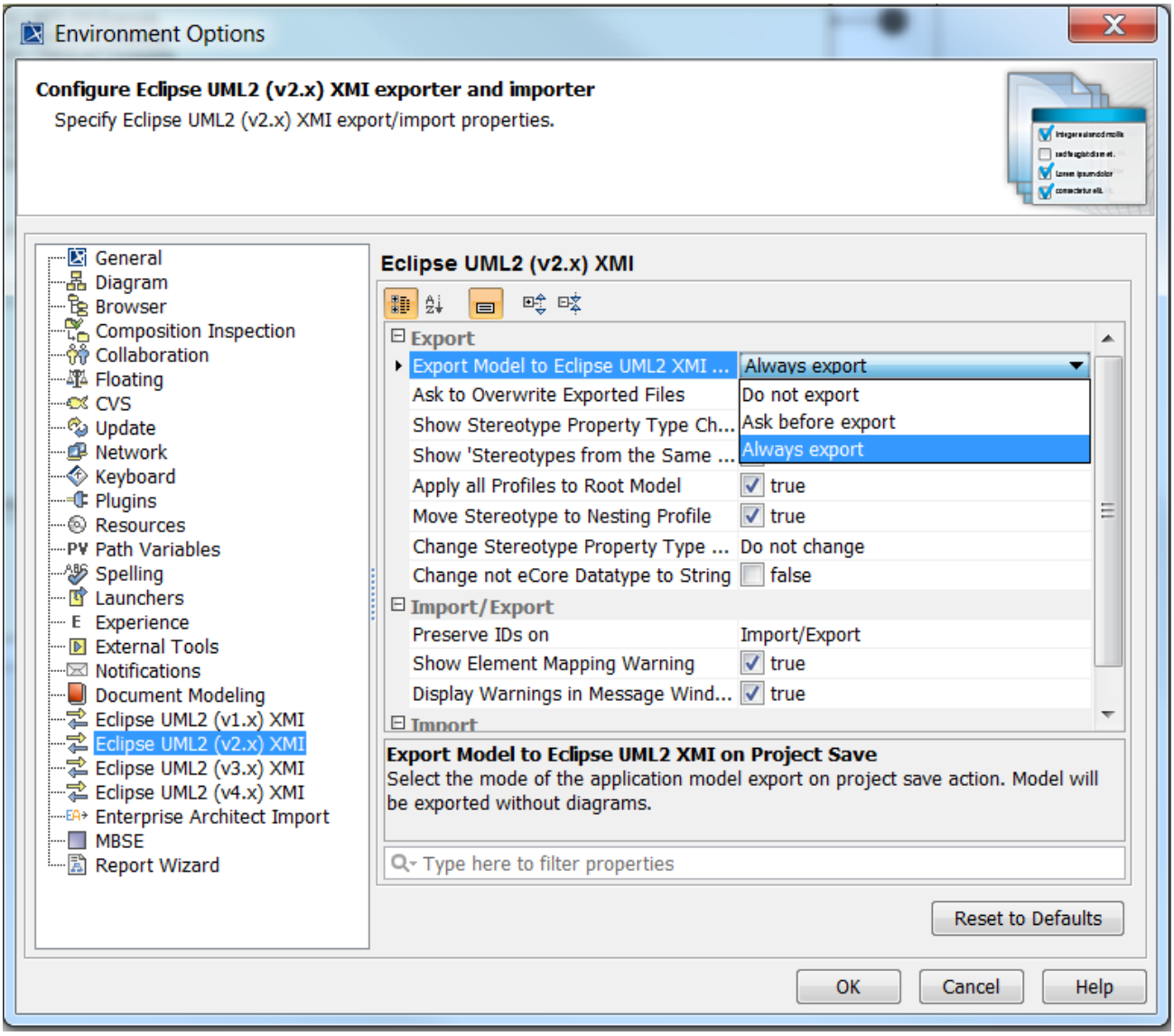


Figure 14 – Making Export to EMF XMI automatic.

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

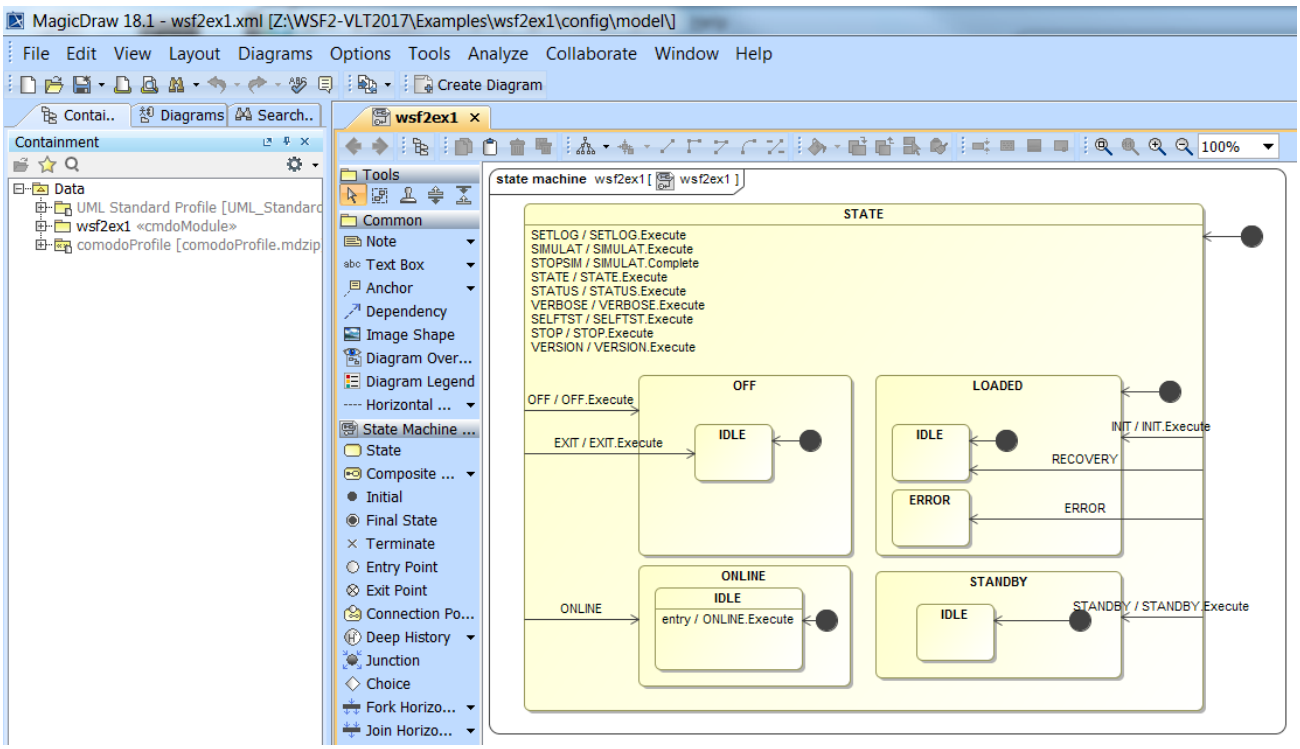


Figure 15 - Containment tree (left section), Tools (center section) and State Machine diagram (right section).

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.

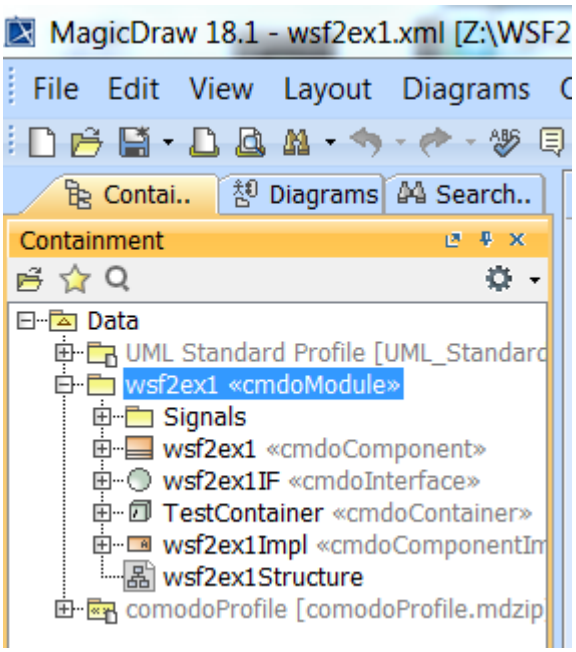


Figure 16 - <<cmdoModule>> Package’s content.

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.

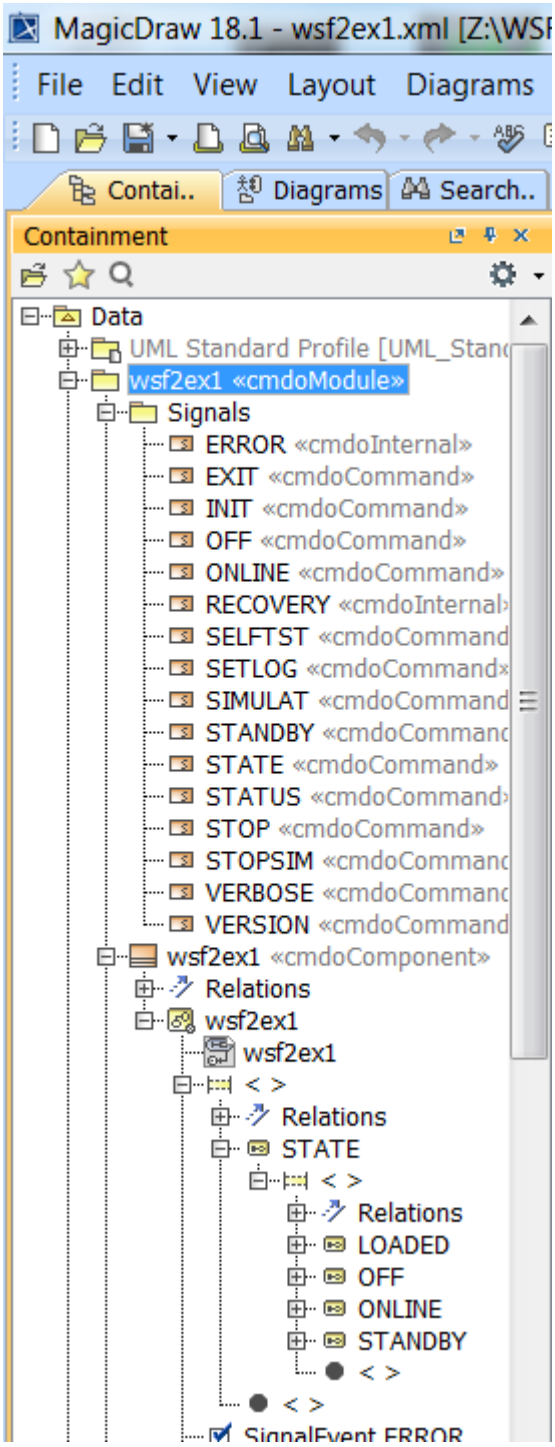


Figure 17 – Signals and State Machine model elements for wsf2ex1 applications.

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).

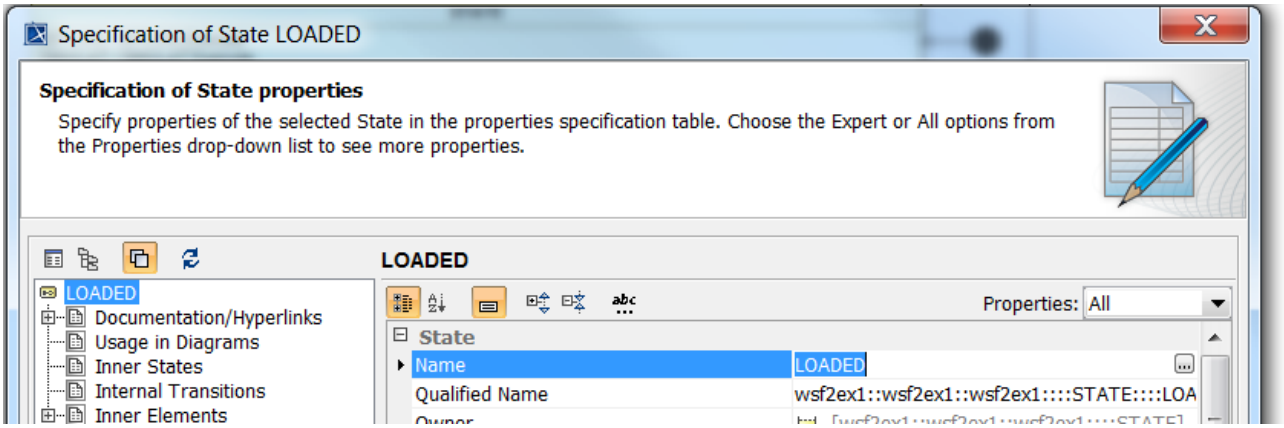


Figure 18 - Specification Dialog, all properties.