System Manual

Dario Lucia

Version 1.0.0, 2024-05-04

Table of Contents

Applicable Version

This manual described the design, behaviour and configuration of ReatMetric version 1.0.x.

System Overview

ReatMetric is a Java-based software infrastructure for the implementation of Monitoring & Control (M&C) systems, with a strong focus on the space domain. ReatMetric components provide a simple but efficient implementation of the typical functions used in an M&C system.

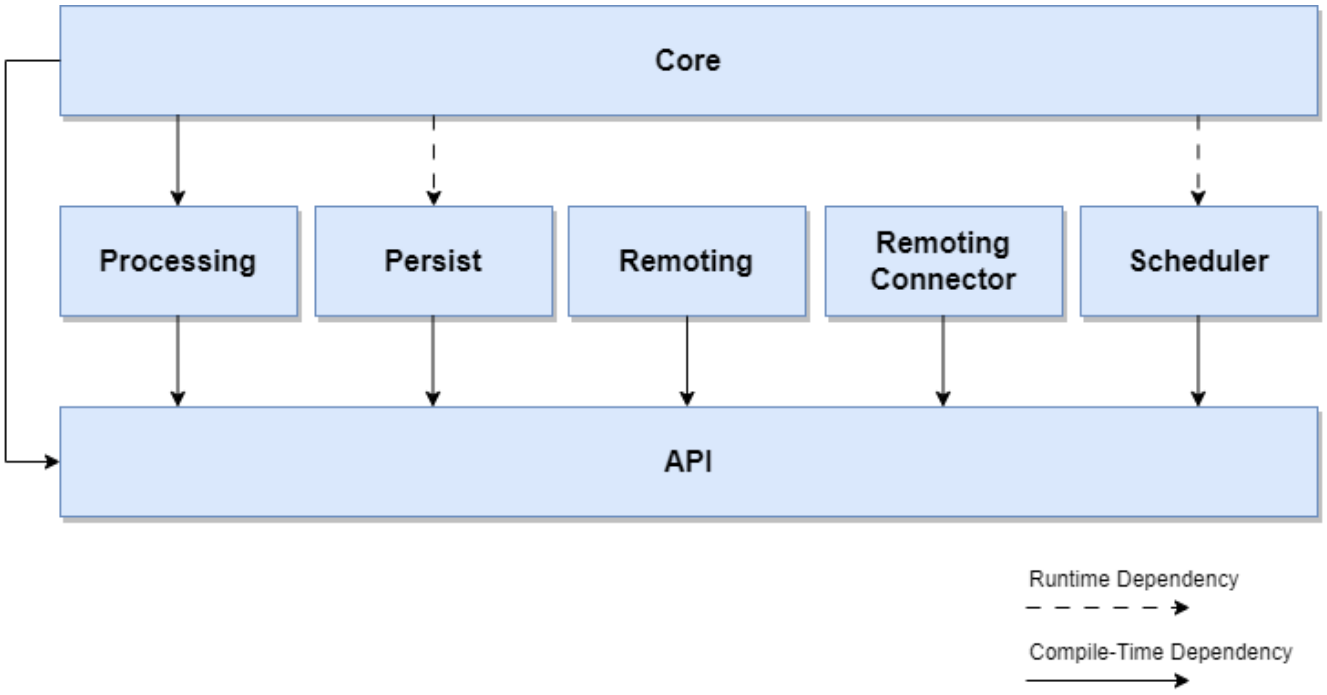
Introduction

A Monitoring and Control (abbreviated as M&C) system is an application used to:

- retrieve and process information of the status of hardware and software devices in a given context,
- allow control and commanding of such devices,
- present such information to human users, operators and other software systems,
- detect non-nominal situation and raise alarms accordingly,
- schedule operations and activities to be executed on the devices,
- record the most important data for subsequent analysis.

ReatMetric is designed as a modular M&C framework, whereas each module follows the Java Platform Module System specification (https://en.wikipedia.org/wiki/Java_Platform_Module_System). Each module provide a specific set of functionalities, data structures and interfaces that can be used by other modules. While some modules provide data structures and interfaces that are at the foundation of the framework, other modules can be replaced in isolation.

The modules that form the foundation of the ReatMetric system are shown in the following figure.



Monitoring and Control Concepts

In the ReatMetric world, the environment to be monitored and controlled is seen as a hierarchical decomposition, starting from a top level object, further decomposed into groups, systems, sub-systems, devices and so on. For instance, the monitoring of a domotic system for a home automation system could be hierarchically arranged as follows:

- My house
  - Blinders
    - Bedroom 1
      - Window Roller 1
        - Engine (device)
        - Control Panel (device)
      - Window Roller 2
        - ...
    - Bedroom 2
      - Window Roller
        - ...
    - Kitchen
      - Window Roller
        - ...
    - Bathroom
      - Window Roller
        - ...
  - Appliances
    - Kitchen
      - Dishwasher (device)
      - Oven (device)
      - Microwave (device)
      - Fridge (device)
    - Bathroom
      - Washing Machine (device)
      - Dryer (device)
    - ...
  - Heating System
    - ...
  - Light System

        - ...
    - Network
        - ...

The arrangement of the hierarchy is sometimes obvious to design, while in other cases can be more a matter of taste. In the example above, for instance, the hierarchy is a functional hierarchy, based on the functional elements present in the domotic system. However, a hierarchical breakdown based on the rooms of the house could have been an alternative. What it is not a matter of taste are the physical objects (labelled as 'device'), which are ultimately monitored and controlled. Such objects are characterised by:

- A readable state - i.e. a set of *parameters* - with related values, which you can actively monitor and, in some cases, also change. For instance, a television has the following state parameters: if it is on or off, selected input source, selected program, current volume, if is muted or not, version of the installed firmware... Some parameters can be also set (e.g. changing the program or muting/unmuting the device), while some others are read-only (e.g. the version of the installed firmware).
- A commanding interface - i.e. a set of defined *activities* - that specify what you can do with the device. For instance, an oven can have a way to request the start of the self-cleaning. Depending on the protocol exposed by the device, the lifecycle of activity executions can be monitored (e.g. the device informs that the operation has been accepted and, after a while, that the operation has been completed - with success or with error).
- A way to signal when something happened - i.e. an *event* - which is relevant for the functioning purpose of the device. For instance, a television might signal when a new version of the firmware is detected.

ReatMetric concepts are derived from this general device characterisation, as described above. As it is a M&C framework oriented to the space domain, such concepts are actually a re-elaboration of the M&C concepts reported in the ECSS standard ECSS-E-ST-70C (https://esastar-publication.sso.esa.int/api/filemanagement/download?url=emits.sso.esa.int/emits-doc/ESOC/1-6223/ECSS-E-ST-70-31C(31July2008).pdf). ReatMetric manages a hierarchical tree composed of so-called *System Entities*. A *System Entity* (at any level of the hierarchy) is characterised by *parameters*, *activities* and *events*. Some system entities are mapped to actual devices, while others are introduced only as 'containers', to partition large systems in subsystems.

Parameters

A *parameter* is a property containing a value. A parameter defines a so-called raw type and an engineering type:

- The raw type is the type of the value that will be reported for that parameter, for further processing. For instance, the status of the mute/unmute parameter of a television might be reported by the television interface as 0 if the TV is muted, and 1 if it is unmuted. In such case, the raw type of the 'muted status' parameter is an unsigned integer. The value as reported by the TV takes the name of *source value*.
- The engineering type is the type of the value that will be reported after the processing. For

instance, it might be desirable to have a mapping between the value 0 to the string 'MUTED' and the value 1 to the string 'UNMUTED'. The function mapping the source value to the engineering value takes the name of *calibration function*.

The value of a parameter might be considered valid depending on certain conditions. For instance, the current value of the TV volume level should be irrelevant, if the TV is muted. The parameter mapping the TV volume level will have therefore a *validity condition*, linked to the parameter value of the mute/unmute parameter. Parameters that are not valid are not calibrated.

Each parameter may define a set of *monitoring checks*, i.e. conditions that are evaluated again the engineering value of the parameter every time a new engineering value of that parameter is available. If the value is not satisfying the defined check, the parameter *alarm state* will change to a non-nominal state. Each check may define an applicability condition, which is evaluated to understand if the related check must be verified or not. Parameters that are not valid are never checked.

There is a type of parameters, for which the *source value* is not retrieved by devices, but it is computed internally by ReatMetric, based on an arithmetic/algorithmic expression. Such parameters are called *synthetic parameters*: the source value of synthetic parameters is recomputed every time one of the dependant parameters is updated.

Activities

An *activity* is the definition of an operation that can be performed by a given system entity. It is an abstract representation of an action.

An activity is characterized by a set of named arguments, each having a raw type and an engineering type. As per parameters, a de-calibration function may be present for each argument, to convert the argument value provided as engineering value into a source value, which is then provided to the underlying layer that implements the execution of such activity. Properties can also be defined.

Once invoked, an *activity occurrence* is created and dispatched to the specific ReatMetric driver for implementation. Upon its creation, an activity occurrence starts its lifecycle. Such lifecycle in ReatMetric is defined by the following *states*:

- CREATION: this state is assigned upon creation of the activity occurrence.
- RELEASE: this state is reached when the activity occurrence is prepared for release.
- TRANSMISSION: this state is reached as soon as the activity occurrence leaves the ReatMetric system.
- SCHEDULING: this state is reached when the activity occurrence will be executed by the final destination at a given point in time.
- EXECUTION: this state is reached when the activity occurrence is under execution by the final destination
- VERIFICATION: this state is reached when the effects of the activity occurrence can be verified at parameter level in the ReatMetric processing model.

- COMPLETED: this state is reached when no further state updates are expected. The activity occurrence lifecycle is considered complete.

In order to transition from one state to the other, *verification stages* about the progress of the activity occurrence are announced to the ReatMetric processing model, internally by ReatMetric itself and by the specific driver that manages the implementation of the activity occurrence. A verification stage can be considered as a concrete step in the progress of an activity occurrence. Each verification stage is announced and further updated by means of *reports*. Typically, such reports depend on the way the transmission protocol is defined and in the specific lifecycle of the activity implementation. Each report specifies: the name of the verification stage, the activity state it belongs to; the estimated/exact execution time of the activity occurrence; the activity state the activity occurrence shall transition to, upon processing the report (if the report is successful); an optional result.

The report of a verification stage can have one of the following states:

- UNKNOWN: The stage state is unknown and no better prediction can be done
- EXPECTED: The stage should be concluded, but no confirmation of the stage was received yet
- PENDING: The stage is reported as currently open and the system is waiting for its confirmation
- TIMEOUT: The timeout linked to the stage expired
- OK: The stage is reported as successfully executed
- FAIL: The stage is reported as failed, but this failure is not fatal for the execution of the activity occurrence
- FATAL: The stage is reported as failed, the activity occurrence shall be considered completed
- ERROR: This specific state is reported in relation to a verification expression, linked to the activity occurrence state VERIFICATION when the expression cannot be evaluated due to expression's errors

An important state of an activity occurrence lifecycle is represented by the VERIFICATION state: in such state, ReatMetric will evaluate a specific expression. Such expression, if defined, is based on parameters and activity occurrence arguments, and it is used to verify that the effects of the activity occurrence execution are actually visible in the monitored system, regardless of any execution report possibly generated by the end system.

Events

An *event* is used to indicate the occurrence of a situation. Differently from activity occurrences and parameters, events do not have a state as such, because their lifetime is, theoretically speaking, istantaneous. For instance, an event could be raised by a television to indicate the presence of a new firmware version. It could be that the same information is delivered by a parameter as well (e.g. a parameter named "Latest available firmware version"): the change in value of this specific parameter triggers the event. In more correct terms, the object generated by ReatMetric should be called *event report* rather than just *event*: the terminology has been used to align it to the ECSS standard.

An event has a qualifier and a source, and it is defined by a severity and a type. Events in

ReatMetric can include a generic report object, which contains additional information about the raised event.

In ReatMetric events can be reported by the drivers (so-called *reported events*), or can be detected autonomously by the ReatMetric processing model, by evaluating conditions upon change of parameters (so-called *TM-based events*).

Connectors, activity handlers and routes

In order to receive data from monitored devices and to send commands to such devices, ReatMetric needs to establish data connections to them. Typically, a device allows a connection using a specific network/transport protocol, which is used to exchange data using an application protocol. For instance:

- you can connect using a TCP/IP connection, which it is used to exchange binary or ASCII-based Protocol Data Units (PDUs);
- you can connect using a TCP/IP connection, which it is used to exchange HTTP(s) requests/responses;
- you can query the device using SNMP GET requests and send commands via SNMP SET requests;
- you can connect via a serial port, which it is used to exchange binary or ASCII-based PDUs.

The way of connecting and exchanging messages to the device is device-dependent. Even the number of connections is device-dependent. Some devices might deliver monitoring data using a TCP/IP port and receive control requests and commands using a separate TCP/IP port.

In ReatMetric, a *connector* is used to represent and manage a logical connection between the ReatMetric system and the target device. For every device, you may have one or more connectors. Via the connector interface, you can start, stop, abort, initialise and monitor the logical connection to the target device. A logical connection can be mapped to a single physical connection to the device, or to multiple physical connections, even to different devices. This depends on the way the support to the device is implemented (see section Drivers).

The state of a logical connection determines the availability of a so-called *route*. A route is a uniquely identified path from the ReatMetric system to the target device. Every parameter or event received from a given route reports also the incoming route as part of its state. In the same way, when an activity occurrence is dispatched to an *activity handler*, the outgoing route must be indicated. If the route, whose state is reported by the corresponding activity handler, is not available, then the dispatch of the activity occurrence fails.

Even if there is typically a logical link between a *connector* and a specific *route*, this link is not enforced by ReatMetric directly. In fact:

- A connector can manage one or more logical connections;
- A logical connection is mapped to zero or more physical connections;
- A route is declared as managed by one activity handler;
- An activity handler manages one or more routes;

- A route can be logically mapped to one or more logical connections by the activity handler that manages the route.

For the ReatMetric architecture, all this management happens inside the so-called *drivers* (see section Drivers).

Raw Data

In ReatMetric, a *raw data* is an semi-opaque data object, with the meaning that ReatMetric does not know the internal structure or data of the object, but only some meta-data attributes, such as the originating source, the route, the generation and reception time, and so on. Any module in the ReatMetric architecture can store and retrieve *raw data*.

For instance, *raw data* can be used to distribute and optioanlly store low level PDUs that are exchanged at connection level between ReatMetric and the target devices. In the same way, it can be used to store changes of state or internal states of modules and drivers, as required.

ReatMetric Main Interface

To be written

Drivers

A *driver* is the object that connect ReatMetric with an external device or system. A driver is responsible for:

- Managing and providing a set of activity handlers, which are the entry point for activity dispatching by the ReatMetric system;
- Managing and providing a set of connectors, which are used to manage the connections between ReatMetric and the external device or system;
- Managing the specific protocols and formats to retrieve parameters and events from the external device or system, and ingest such data into the ReatMetric processing model;
- Managing and providing the raw data interpreters, i.e. renderers that can be used to display raw data internals, as made visible by the driver;
- Providing internal state information, for debug and monitoring purposes.

The lifecycle of a driver is pretty simple:

- A driver is instantiated and initialised by the ReatMetric Core module, by invoking the initialise(...) method. This method contains several arguments, among others the name and configuration of the drivers, and the *context* of the ReatMetric Core instance.
- The driver is enquired to know the available activity handlers, raw data renderers and transport connectors. Each of this elements is then used accordingly by the relevant ReatMetric modules, i.e. the activity handlers are used by the processing model when dispatching activity occurrences; the renderers are used by the MMI; the transport connectors are used by the ReatMetric Core and by the MMI to activate and manage the external connections.

- At ReatMetric Core instance shutdown, the dispose() method is invoked, to perform clean-up activities and to release the resources.

Once a driver is disposed, it is never re-used. Therefore, the initialise(…) and dispose() methods are called only once during a single ReatMetric Core instance lifecycle.

In the initialise(…) method, a reference to a context object is provided. Such object allows the driver to reach the different functions made available by the ReatMetric system:

- The archive, if available;
- The scheduler, if available;
- The processing model;
- The raw data broker;
- The operational message broker;
- The ReatMetric system main interface.

Foundation Modules

API

Overview

The eu.dariolucia.reatmetric.api module provides the definition of all interfaces and data structures that are used by ReatMetric and exposed to the 'outside world'. Unless you are coding a driver, importing this module ensures that all ReatMetric interfaces and data structure are available for you to use. For instance, the MMI implementation provided by the eu.dariolucia.reatmetric.ui module depends only on this module and not on a specific ReatMetric implementation.

The facade interface to a ReatMetric system is specified by the IReatMetricSystem Java interface: such interface provides all necessary methods to initialise, access and dispose ReatMetric functions. An implementation of such interface is provided by the eu.dariolucia.reatmetric.core module.

Each class in this module is provided with Javadocs, to help developers in the understanding of the ReatMetric API. Therefore, for guidelines and help about its usage it is recommended to check the Javadocs.

This module has no internal or external dependencies.

Processing

Overview

The eu.dariolucia.reatmetric.processing module provides an implementation of the IProcessingModel interface specified in the API module. Such model defines the M&C structure of the system(s) monitored by ReatMetric, in terms of system elements, parameters, events, activities.

Upon loading the definitions of the model, this module builds a graph of dependencies across the various entities, in order to understand which entity state needs to be recomputed, when another entity state changes. For instance, synthetic parameters defined by expressions are recomputed only upon change of their input parameters. With the derivation of the topological ordering of the entities, it is ensured that the number of entity state recomputations is only one per set of updates.

In addition, the graph is used to determine when two requests for parameter/event updates can be processed in parallel, because there is no processing overlap between the two requests. For instance, if two set of parameter updates are provided at the same time (e.g. since they come from the monitoring of two separate devices), then the processing will be performed in parallel by two different processing threads. Of course, this happens only if there is no 'shared entity' affected by the two updates. For instance, if a synthetic parameter is defined, having parameters coming from two different devices as input parameters, then the updates will be serialised due to the synthetic parameter presence.

Since the topological ordering is an expensive operation, the module might create a cache the first

time this ordering is computed, generating a file named ".ordering.cache", and reuse this cache every time the model is loaded again.

In terms of activity dispatching, activity handlers need to be registered to the processing model implementation via the methods defined in the IProcessingModel interface. At registration time, the processing model enquiries the activity handler about the supported routes and keeps a correspondence between the route and the activity handler. An activity handler also reports which type of activity it supports. The type is a simple string and it must be assigned to each activity in the processing definition.

All state updates generated by the processing model are sent to a single sink: this approach is not specified by the IProcessingModel interface, but it is a characteristic of this implementation.

This module has one internal dependency:

- On eu.dariolucia.reatmetric.api

This module has two external dependencies:

- On JAXB library, since the processing definitions are defined in XML files;
- On Groovy library, since expression-based checks, expression-based calibrations, parameter-based verifications for activities and synthetic parameters/events are defined using Groovy as scripting language.

Behaviour

Once instantiated, the IProcessingModel implementation must be used according to the contract specified by the IProcessingModel interface (check the related Javadoc). The processing of the related request is performed asynchronously, even if some parts (e.g. the derivation of the unique ID mapped to an activity invocation request) are done synchronously, in order to return a meaningful value to the caller.

Once a request is received by the processing model, it is routed to a specific dispatching queue:

- Requests linked to *reporting operations* (parameter updates, requests to raise events, activity progress reports) are added to a *reporting queue*
- Requests linked to *commanding operations* (activity invocations, activity control operations, system element enable/disable) are added to a *commanding queue*

The *commanding queue* is of limited capacity (20000) and can therefore block the caller, if it becomes full. Half of the queue is reserved for internal operations of the processing model that result in internal commanding requests, to avoid deadlocks and delays in the processing of reporting operations. This means that, if more than half of the *commanding queue* is full, externally requested commanding operations will block until at least half of the *commanding queue* is empty.

The *reporting queue* is of limited capacity (1000) and can therefore block the caller, if it becomes full. Backpressure is hence generated in case the processing model is saturated.

Each queue has an associated dispatch thread, which is responsible to:

- Remove one item from the queue, containing the list of operations to be performed
- Finalize the list of operations by extending it with the necessary re-evaluations
- Identify the involved processors and the affected system elements
- Re-order the final list of operations according to the topological sort
- Record the affected system elements as part of the *working set*
- Request the implementation of the operations to the *processing thread pool*

The *working set* identifies what system elements are currently under update by the *processing thread pool*. At the end of the update, such system elements are removed from the working set. If there is an overlap between the working set and the system elements affected by one list of operations (i.e. there is a processing conflict), the dispatch thread waits until the conflict is resolved.

The *processing thread pool* is a fixed thread pool, having a number of thread equals to the number of available processors in the machine (Runtime.getRuntime().availableProcessors()). A thread in this pool is responsible to apply and process an operation on the related system elements. Only the system elements having state ENABLED and IGNORED are processed.

Each operation on a system element generates a visible state (ParameterData, EventData, ActivityOccurrenceData, AlarmParameterData), which is then delivered to the output sink provided in the constructor of the model implementation (IProcessingModelOutput).

The processing model does not provide any subscription or archiving mechanisms: such functions are provided by the eu.dariolucia.reatmetric.core module.

Parameter Processing

Parameter samples are injected into the model using the method: void injectParameters(List<ParameterSample>).

The processing is done according to the following logic:

1. If the generation time of the sample is older than the generation time of the previous sample, the injected sample is ignored and the processing is over
2. The validity of the parameter is computed
3. If the parameter is valid, then the engineering value is computed
4. If the parameter is valid, then the alarm state is computed
5. A ParameterData object is generated
6. The alarm state is evaluated, to decide whether an AlarmParameterData object must be generated
7. The defined triggers are evaluated

Event Processing

Events are injected into the model using the method: void raiseEvent(EventOccurrence).

The processing is done according to the following logic:

1. If the event occurs within the defined inhibition period (based in generation time), then the injected event is ignored and the processing is over
2. A log message is generated, if the event does not lie in the log repetition inhibition period
3. An EventData object is generated

Note that the generation time is not used to ignore events based on previous occurrences, as it is done for parameters.

Activity Processing

Execution of activities is requested using the method: IUniqueId startActivity(ActivityRequest).

The processing is done according to the following logic:

1. The request is checked for its consistency: all arguments are present with the right type, the route is defined and it is existing
2. The activity occurrence is created
3. The activity occurrence is forwarded to the handler linked to type and route, for implementation
4. An ActivityOccurrenceData is generated

The IUniqueId returned by the method uniquely identifies the created activity occurrence in the system.

Progresses about the activity execution are injected into the model using the method: void reportActivityProgress(ActivityProgress).

The processing is done according to the following logic: . The activity state is updated according to the ActivityProcess information . If an activity state transition is detected, the timer for the timeout is started (if defined) . An ActivityOccurrenceData is generated

The processing model defines specific verification stages: some of them **must** be used by callers as part of the ActivityProgress objects, so that the correct transitions can take place:

1. The constant ActivityOccurrenceReport.CREATION_REPORT_NAME ("Creation") is internally used by the processing model to indicate the creation stage of the activity occurrence and it is the only stage of the activity state ActivityOccurrenceState.CREATION. Its use is reserved and shall not be used by activity progresses.
2. The constant ActivityOccurrenceReport.FORWARDING_REPORT_NAME ("Forwarding to Activity Handler") is announced by the processing model as part of the activity state ActivityOccurrenceState.RELEASE. The stage indicates the forwarding phase of the activity occurrence to the associated activity handler via the method IActivityHandler.executeActivity(). Any exception raised by the handler is caught by the processing model and a FATAL result is reported associated to this stage. The activity handler must report an OK linked to this stage, if the activity occurrence is accepted for release and remain in the ActivityOccurrenceState.RELEASE activity state.
3. The constant ActivityOccurrenceReport.RELEASE_REPORT_NAME ("Release") must be

announced by the activity handler when the encoded activity is about to leave the system and reported with a OK or FATAL depending on the result. Upon report, the next state of the activity shall be moved to ActivityOccurrenceState.TRANSMISSION.

4. The constant ActivityOccurrenceReport.VERIFICATION_REPORT_NAME ("Verification") is internally used by the processing model when the activity state moved to ActivityOccurrenceState.VERIFICATION, as announced by the activity handler/driver that monitors the progress of the activity occurrence.

An activity handler can report, for the TRANSMISSION and EXECUTION state, an unlimited number of verification stages.

The sequence of activity occurrence states is the following:

CREATION → RELEASE → (TRANSMISSION) → (SCHEDULING) → EXECUTION → VERIFICATION

The TRANSMISSION and SCHEDULING states are optional: in fact, activity occurrences could be implemented by internal ReatMetric functions/drivers, so the RELEASE state could be immediately followed by the EXECUTION state. The SCHEDULING state is used when the activity occurrence is scheduled for execution at a later point in time on the remote device/equipment.

Upon entering the TRANSMISSION, EXECUTION and VERIFICATION states, the related timers defined in the definition (if present) are started: the stages belonging to the state, that are PENDING at the expiration of the timer linked to the state, are reported as TIMEOUT by the processing model.

Mirrored Elements

A processing model can be kept synchronised with the state of another processing model via the mirror(List<AbstractDataItem>) method. The processing of such requests simply overwrite the state of the system element with the one provided as input. Such requests are only considered for system entities marked as 'mirrored' in the definition.

Configuration

The configuration of the processing module via the processing definition XML files is complex, but it can be broken down to configuration elements that are simpler to understand. The configuration structure is defined in the package eu.dariolucia.reatmetric.processing.definition. It is one or more XML files using namespace definition *http://dariolucia.eu/reatmetric/processing/definition*.

A commented example of such file (broken down in parts) is presented below.

Root element

The XML definition file has a root node named *processing*.

```
<ns1:processing xmlns:ns1="http://dariolucia.eu/reatmetric/processing/definition">
    <parameters>
    ...
    </parameters>
    <events>
```

```
    ...
    </events>
    <activities>
    ...
    </activities>
</ns1:processing>
```

The *processing* node has the following optional attributes:

synthetic_parameter_processing_enabled: default is 'true'. When set to 'false', the processing of synthetic parameters will be disabled. However, if more than one file is used to build up the configuration and one of these files has this attribute set to false, the processing of the synthetic parameters will be disabled for all definitions in all files.

mirrored: default is 'false'. This flag is used to indicate that the elements described in the file are managed by a separate processing model. When set to 'true', the processing model will only process updates coming from the mirror() method, and will not recomputed/recalculate any value.

path_prefix: default is empty. When set to a string, e.g. 'STATION.DEVICE3.', the path will be prefixed to all configured locations of the elements defined in the file. This approach is particularly useful in case some definitions of a separate processing model must be imported in a different processing model for mirroring and must appear in a specific sub-branch of the hierarchical decomposition.

Parameters

The *parameters* element contains zero or more *parameter* elements. An example is provided in the XML fragment below.

```
<parameter id="#170"
    location="SUB1.STATUS"
    description="Subsystem Status"
    raw_type="ENUMERATED"
    eng_type="CHARACTER_STRING"
    eng_unit=""
    log_repetition_period="2000"
    user_parameter="false">
    <validity>
        <matcher parameter="#312"
            operator="EQUAL"
            value="32"
            value_type="ENUMERATED"
            use_raw_value="true" />
    </validity>
    <calib_enum default="UNKNOWN">
        <point x="0" y="OFF" />
        <point x="1" y="ON" />
    </calib_enum>
    <checks>
        <expected type="CHARACTER_STRING">
            <value>ON</value>
```

```
        </expected>
    </checks>
    <setter activity="#104" set_argument="new_value">
        <!-- Convert? -->
        <decalib_ienum>
            <point y="0" x="OFF" />
            <point y="1" x="ON" />
        </decalib_ienum>
        <fixed_argument name="device_subsystem" value="1" raw_value="true" />
        <fixed_argument name="parameter" value="0" raw_value="true" />
    </setter>
    <triggers>
        <trigger condition="ON_VALUE_CHANGE" event="#123" />
    </triggers>
</parameter>
```

The *parameter* node has the following attributes:

id: mandatory special string. The string is a positive integer number, prefixed with the hash # symbol. The number must be unique among all system element ids defined in the ReatMetric model under monitoring, including those defined in separate processing models.

location: mandatory string. This string represents the location of the parameter in the hierarchical decomposition. Each element of the hierarchy is separated by a dot .

description: mandatory string. The description of the parameter.

raw_type: mandatory enumerated value. The type of the source value: when a parameter sample is injected into the model, the source value of the parameter sample must have this type.

eng_type: mandatory enumerated value. The output type of the parameter, after the processing.

eng_unit: default is empty string. The unit of the parameter engineering value.

log_repetition_period: default is 0 (disabled). The minimum log generation period in milliseconds. If an alarm generates a log within the minimum repetition period window, the log message is skipped and a counter increased. This is a way to limit log flooding for parameters in case of high sampling rates.

user_parameter: default is 'false'. This attribute indicates whether a parameter must be considered a user-parameter, i.e. not linked to external device parameters but rather settable directly from users, drivers or other ReatMetric elements and not via a setter element, which is mapped to an activity for dispatching and remote execution. User parameters can be considered 'internal' parameters, which can be useful to define globally available values and properties, with all the processing capabilities of validity, calibrations, checks and triggers available for standard parameters.

The *parameter* node has the following sub-elements:

validity: optional, if not provided the parameter is always considered valid. The *validity* element

can contain either a *match* element or a *condition* sub-element. A *match* element is used to determine the validity based on a comparison between the value (source or engineering) of a second parameter, and the value specified in the definition, with the related operator (equal, higher, lower, not equal…). A *condition* element specifies a Groovy expression that shall return a boolean value upon evaluation.

synthetic: optional, to be provided only if the parameter shall be a synthetic parameter. A synthetic parameter requires an expression to compute its source value: therefore, injection of such parameters via the injectParameter() method will be rejected by the processing model.

default_value: optional, null if not provided. It allows to specify a default value (either raw or engineering) that the processing model will initialise as default value upon instantiation. Uninitialised parameters will have a Java null value as source and engineering value.

calib_x: optional (zero or more), no calibration if none is provided. When no calibration is assigned, the source/raw value is simply assigned as engineering value. When provided, the calibration to be applied is selected by checking the list of specified calibration in the order they appear in the definition, and the first calibration matching the applicability criterium is selected and applied. If no applicability criterium is specified, then the calibration is always considered applicable. The following calibrations are available:

- calib_xy: a list of x,y pairs is defined. The raw value must be defined as a number (integer, real, enumerated). The engineering value must be defined as a real. The processing model computes the output linked to the provided input number by linearly interpolating the y-value between the pair that encloses the x-input. Extrapolation outside the minimum and maximum x can be activated: it is de-activated by default. In such case, the processing model will reject the attempt to calibrate a input outside the series-defined boundaries.
- calib_poly: the definition specifies 6 coefficients (from a0 to a5), which compose the polynomial function: a0 + a1*x + a2*x^2 + a3*x^3 + a4*x^4 + a5*x^5. The raw value must be defined as a number (integer, real, enumerated). The engineering value must be defined as real.
- calib_log: the definition specifies 6 coefficients (from a0 to a5), which compose the logarithmic function: 1/(a0 + a1*log(x) + a2*log(x)^2 + a3*log(x)^3 + a4*log(x)^4 + a5*log(x)^5). The raw value must be defined as a number (integer, real, enumerated). The engineering value must be defined as real.
- calib_enum: a list of key,value pairs is defined, plus a default string value. The raw value must be defined as an integer number or enumerated. The engineering value must be defined as string. The provided input number is mapped to the corresponding textual value, or to the default value if no correspondence is found.
- calib_range_enum: a list of range-to-value pairs is defined. The raw value must be defined as a number (integer, real, enumerated). The engineering value must be defined as string. Given the provided input number, the range containing it is detected, and the corresponding textual value is returned as engineering value. If no range contains the provided input, the default value is returned.
- calib_expression: input and output values can be of any type. The defined Groovy expression is used to compute the output value.
- calib_external: input and output values can be of any type. The processing model uses the

registered ICalibrationExtension implementation, as specified in the definition, to request the calibration of the provided input value.

checks: optional (zero or more) checks can be specified inside the element *checks*. Checks are applied to the engineering value of the parameter of a parameter by default (unless differently specified), if the check applicability criteria is satisfied. If a check does not have an applicability criteria, the check is always considered to be applied. By default, a violated check immediately raises an alarm: it is possible to modify the severity of the check, as well as the number of consecutive checks that the check must fail before raising the failure. The sub-elements of the *checks* element are:

- limit: the value must be inside the low/high limit specified in the definition. If it is outside the limit, the check is declared failed.
- expected: the value must match one of the values specified in the definition, otherwise the check is declared failed.
- delta: the difference between the new value and the previous value is computed. If such difference is not within the low/high limit specified in the definition, the check is declared failed. The absolute (positive) difference can also be used.
- expression: a Groovy expression, returning a boolean value, is defined. If the evaluation of the definition returns false, then the check is declared failed.
- external: The processing model uses the registered ICheckExtension implementation, as specified in the definition, to request the check of the provided parameter.

setter: optional (zero or one) reference to an existing activity, which shall be used to set the value of the parameter to a new value. The definition must specify which argument of the activity shall be set to the new value specified by the user. The activity invocation is complemented by the list of fixed argument values, each mapping to an activity argument. If such setter is specified, the SetParameterRequest will translate to an activity invocation. It is important to note that the setter definition must specify a *complete* activity invocation, i.e. all activity arguments must be specified. An optional de-calibration can be specified (including inverted enumerations) to convert the value specified as engineering value into raw value.

triggers: optional (zero or more) triggers can be specified inside the element *triggers* (sub-node *trigger*). Triggers can be linked to events, which are raised when there is a specific change in the parameter state. The specific change is specified with the attribute *condition* (which takes four possible values: ON_NEW_SAMPLE, ON_VALUE_CHANGED, ON_ALARM_RAISED, ON_BACK_TO_NOMINAL), while the event to raise is specified in the attribute *event*.

Events

The *events* element contains zero or more *event* elements. An example is provided in the XML fragment below.

```
<event id="#190"
    location="SUB1.STATE_CHANGE"
    description="Subsystem State Changed event"
    severity="INFO"
```

```
    type="Event Type 1"
    inhibition_period="1000"
    log_repetition_period="2000">
</event>
```

The *event* node has the following attributes:

id: mandatory special string. The string is a positive integer number, prefixed with the hash # symbol. The number must be unique among all system element ids defined in the ReatMetric model under monitoring, including those defined in separate processing models.

location: mandatory string. This string represents the location of the event in the hierarchical decomposition. Each element of the hierarchy is separated by a dot .

description: mandatory string. The description of the event.

severity: optional, default is INFO. The severity of the event.

type: optional, default is empty string. The type of the event.

inhibition_period: optional default is 0 (disabled). The number of milliseconds between two reporting of the event. This is a way to limit event flooding.

log_repetition_period: optional, default is 0 (disabled). The minimum log generation period in milliseconds. If an event generates a log within the minimum repetition period window, the log message is skipped and a counter increased. This is a way to limit log flooding for events.

log_enabled: optional, default is true (enabled). When set to false, the event does not generate log messages.

The *event* node has the following sub-elements:

condition: optional, to be provided only if the event occurrence shall be computed by the processing model. The provided Groovy expression shall return a boolean value: when a transition false → true in the condition evaluation is detected, the processing model raises the event.

Activities

The *activities* element contains zero or more *activity* elements. An example is provided in the XML fragment below.

```
<activity id="#79" location="SUB1.REBOOT" description="Reboot subsystem" type="DEV3-
BIN-CMD"
    verification_timeout="3000">
    <argument name="delay" raw_type="UNSIGNED_INTEGER" eng_type="UNSIGNED_INTEGER"
eng_unit="" />
    <argument name="running" raw_type="UNSIGNED_INTEGER" eng_type="UNSIGNED_INTEGER"
eng_unit="" />
    <property key="test1" value="V1" />
    <property key="test2" value="V2" />
```

```
    <verification>
        <expression>STATUS == "ON"</expression>
        <symbol name="STATUS" reference="#170" binding="ENG_VALUE" />
    </verification>
</activity>
```

The *activity* node has the following attributes:

id: mandatory special string. The string is a positive integer number, prefixed with the hash # symbol. The number must be unique among all system element ids defined in the ReatMetric model under monitoring, including those defined in separate processing models.

location: mandatory string. This string represents the location of the activity in the hierarchical decomposition. Each element of the hierarchy is separated by a dot .

description: mandatory string. The description of the activity.

type: mandatory string. The type of the activity.

default_route: optional, null if not set. The route to be used in case no route is specified as part of the activity invocation request.

transmission_timeout: optional, default is 0 (disabled). The number of seconds after entering the transmission state, before the state of the specific activity occurrence goes in timeout.

execution_timeout: optional, default is 0 (disabled). The number of seconds after entering the execution state, before the state of the specific activity occurrence goes in timeout.

verification_timeout: optional, default is 0 (disabled). The number of seconds after entering the verification state, before the state of the specific activity occurrence goes in timeout.

expected_duration: optional, default is 1000. The expected duration of an activity occurrence execution, in milliseconds.

The *activity* node has the following sub-elements:

argument: optional (zero or more), it can be interleaved with *array* elements. It defines a simple argument with the given name, type and decalibration function, which is applied if the argument is provided in engineering value. *array*: optional (zero or more), it can be interleaved with *argument* elements. It defines an array of values. *property*: optional (zero or more). It is used to pre-define properties and values to be used as part of an activity invocation. *verification*: optional (zero or one), default is null. It is used to specify a Groovy expression, which is executed when the activity occurrence enters the *VERIFICATION* state and at every change of the contributing inputs, until the timeout elapses.

Expressions

In the definition of the elements of the processing model, it is possible to use a Groovy expression in several places (validity conditions, monitoring checks, calibrations, events, activity verification). An expression is composed by two sub-elements:

expression: it contains the expression that it is used to compute the desired value. The type of such value depends on the use of such expression: for validity conditions, monitoring checks, events and activity verification, the expression must return a boolean value. Expressions used for synthetic parameters and calibrations can return any value, as long as the returned value is in line with the declared parameters types.

symbol: optional (zero or more). Each *symbol* specifies the mapping between a variable present in the expression and a system element or related property in the processing model.

A symbol can be linked to one of the following system element properties using the attribute *binding*:

- PATH: The location of the system entity (as string)
- GEN_TIME: The generation time (as Java java.time.Instant)
- RCT_TIME: The reception time (as Java java.time.Instant)
- ROUTE: The route (as string)
- SOURCE_VALUE: The raw/source value (only for parameters)
- ENG_VALUE: The engineering value (only for parameters)
- ALARM_STATE: The alarm state (as eu.dariolucia.reatmetric.api.model.AlarmState, only for parameters)
- VALIDITY: The validity (as eu.dariolucia.reatmetric.api.parameters.Validity, only for parameters)
- QUALIFIER: The qualifier (as string, only for events)
- SOURCE: The source (as string)
- OBJECT: The bound object (i.e. eu.dariolucia.reatmetric.api.processing.scripting.IParameterBinding for parameters, eu.dariolucia.reatmetric.api.processing.scripting.IEventBinding for events

An example of a synthetic parameter is provided below. The synthetic parameter computes the maximum value between two parameters, depending on the value and alarm state of a third parameter.

```
<parameter id="#1430"
    location="SUB1.MAX_LEVEL"
    description="Subsystem Maximum Voltage Level"
    raw_type="REAL"
    eng_type="REAL"
    eng_unit="mV">
    <synthetic>
        <expression>
            if(COMBINER_VALUE == "COMBINING" &amp;&amp; !COMBINER_ALARM.isAlarm()) {
                return Math.max(LEVEL1, LEVEL2);
            } else {
                return 0;
            }
```

```
        </expression>
        <symbol name="LEVEL1" reference="#1421" binding="ENG_VALUE" />
        <symbol name="LEVEL2" reference="#1422" binding="ENG_VALUE" />
        <symbol name="COMBINER_VALUE" reference="#1424" binding="ENG_VALUE" />
        <symbol name="COMBINER_ALARM" reference="#1424" binding="ALARM_STATE" />
    </synthetic>
</parameter>
```

Persist

Overview

The eu.dariolucia.reatmetric.persist module is the implementation of a storage system for ReatMetric data such as parameters, events, activity occurrences, raw data and operational messages. The implementation is based on the Apache Derby RDBMS, and it can be used in file-based and client-server deployments, depending on the way the argument "archiveLocation" is provided:

- If the path to a folder is provided, then the file-based archive is used;
- If a '//<server>[:<port>]/<databaseName>[;user=<value>;password=<value>[;...]]' string is provided, then a client connection to the server is established. The server must have been already started in advance.

In case the path to a folder is provided:

- If the folder exists, it is expected to contain an Apache Derby database. If not, an error is returned;
- If the folder does not exist, an Apache Derby database is created and configured according to the database schema defined by this module.

Including this module in a ReatMetric system is not mandatory: if not present, the system will work as usual, but it will not store any data. In the same way, it will not be possible to retrieve any historical data, but only monitor the live state of the external devices/systems.

This module can be deployed outside a ReatMetric system, if a means to access the stored data outside the ReatMetric system is desirable, keeping in mind that:

- File-based archives can be accessed only by a single system at once;
- Archives in a client-server deployment can be accessed in parallel by more than a single system.

This module has one internal dependency:

- On eu.dariolucia.reatmetric.api

This module has one external dependency:

- On Apache Derby.

Configuration

None.

Scheduler

Overview

The eu.dariolucia.reatmetric.scheduler module provides the implementation of a scheduler with three main functionalities:

- Schedule activity executions with time-related triggers (absolute time, relative time to other scheduled activities with optional delay);
- Schedule activity executions with event-related triggers, i.e. activities that are dispatched when a specific event is raised by the processing model;
- Schedule activity executions based on state machine transitions (called 'bots').

For all these types of scheduling strategies, the ReatMetric scheduler handles resources and conflicts:

- When a schedule request is sent to the scheduler via the load(...), schedule(...) or update(...) method, the request must have a set of resources linked to it. From a scheduling point of view, a resource is simply a string, identifying the resource. The set of resources can be empty: this means that the scheduled activity does not declare any resource as needed for its execution.
- A conflict can materialize:
  - At insertion time: each insertion/update method for scheduling request foresees the provision of a CreationConflictStrategy specification (an enumeration): such strategy tells the scheduler what to do in case of conflict.
  - At execution time: each scheduling request must provide an indication of what to do, when the scheduled activity should be triggered, but one of the declared resources is declared by a different scheduled activity that is currently running.

The CreationConflictStrategy can be applied only for the scheduling of activities whose scheduling trigger is time-related:

- ABORT: Abort the complete operation if there is a resource conflict: the schedule is unmodified
- SKIP_NEW: Do not add the new activity if a resource conflict exists
- REMOVE_PREVIOUS: Remove the resource-conflicting scheduled items before adding the new activity
- ADD_ANYWAY: Add the activity anyway, with the risk of having problems later

The ConflictStrategy, defined per scheduling request or activity invocation (in case of bots), is applied to all activities that are triggered and should start the execution:

- WAIT: Wait until the latest invocation time (if present) or indefinitely (if not present). Start as soon as the resources are freed.

- DO_NOT_START_AND_FORGET: Do not invoke the activity and forget about its execution. The activity is basically skipped.
- ABORT_OTHER_AND_START: Abort ALL activities that are holding up the required resources, and then start the activity.

If the scheduled activity is triggered to start and the declared resources are available, the resources are acquired and the activity is requested to the processing model to be dispatched.

Even if there is no compile-time dependency, the Scheduler implementation of this module requires an implementation of several ReatMetric interfaces, in order to be instantiated and started.

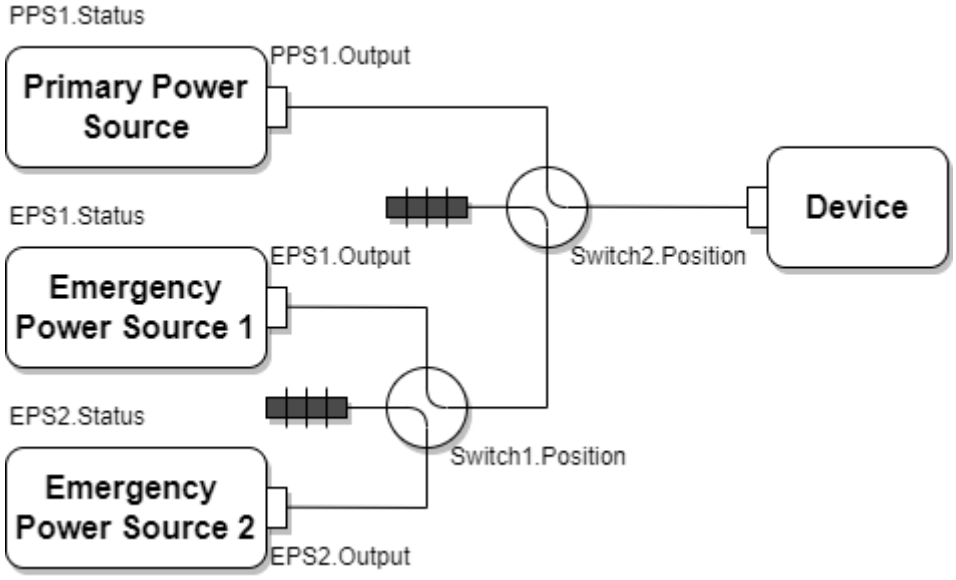
The introduction to the so-called 'bots' require an ad-hoc paragraph. Bots are used to request automated activity invocations depending on changes of state of the monitored systems. Each bot contains a formalized description of the different states of interest of a system, in the form of a list of states. Each state contains:

- A list of criterium targeting a parameter, each of them evaluating to a boolean: true or false; different parameters can be used.
- A list of activities to be invoked.

When the bot is initialised, it will go through the defined states one by one, according to the definition order, and it will evaluate each criterium linked to the state. If the criteria defined for a specific state are all evaluated to true, the bot state is assigned to that state, and it stops evaluating the remaining states. The actions assigned to the new state are executed, if the bot is configured to do so (BotProcessingDefinition executeOnInit set to true).

When any of the monitored parameters in the list of criterium for all states change, the bot will evaluate again the criteria defined for the current state. If all criteria evaluate again to true, there will be no change in the state. Otherwise, the bot will start again evaluating the states one by one, according to the definition order, until it will find a new matching state and perform the transition. If there is no matching state, no transition will be performed.

The function can be better understood with an example. Let’s assume that a device could be powered up by three independent power supplies (a primary power source and two emergency power sources, selected by a switch). There is a second switch that selects which powerline must be used to power up the device. Each power source output can be enabled or disabled, and they should be disabled if not used. The device has an internal capacitor to allow short power breaks in the order of few seconds.



A bot can be configured to automatically handle the necessary commands to enable/disable outputs and change switch positions, depending on the status of the power sources. In order to do that, we define the following bot states:

State name	Criteria	Activities
Nominal	PPS1.Status == "ACTIVE"	Switch2.Set_Position("UP") PPS1.Set_Output("ON") EPS1.Set_Output("OFF") EPS2.Set_Output("OFF")
Degraded 1	EPS1.Status == "ACTIVE"	Switch2.Set_Position("DOWN") Switch1.Set_Position("UP") EPS1.Set_Output("ON") PPS1.Set_Output("OFF") EPS2.Set_Output("OFF")
Degraded 2	EPS2.Status == "ACTIVE"	Switch2.Set_Position("DOWN") Switch1.Set_Position("DOWN") EPS2.Set_Output("ON") PPS1.Set_Output("OFF") EPS1.Set_Output("OFF")

At system start-up, the bot will initialise itself to a state (let’s assume "Nominal") and, if so

configured, it will execute the configured activities. If there is any variation of any parameter configured in the Criteria column, then the bot will re-evaluate its state. What it is important to see, is that the re-evaluation of the states will always start from the current state first, and only if the criteria of the current state are not satisfied anymore, the bot will evaluate the list of states from the beginning. This strategy avoids oscillating state transitions, i.e. moving from one state to the other very quickly.

With reference to the example, as soon as the bot enters the "Degraded 1" state, it will stay in that state as long as the emergency power source 1 stays active, no matter what happens to the primary power source status.

Configuration

The configuration structure of the eu.dariolucia.reatmetric.scheduler module is defined in the package eu.dariolucia.reatmetric.scheduler.definition. It is an XML file using namespace definition *http://dariolucia.eu/reatmetric/scheduler*.

A commented example of such file is presented below.

```
<ns1:scheduler xmlns:ns1="http://dariolucia.eu/reatmetric/scheduler" scheduler-
enabled="true"
    run-past-scheduled-activities="false">
    <!-- scheduler-enabled: status of the scheduler when the system starts, default is
true -->
    <!-- run-past-scheduled-activities: if true, activities in status SCHEDULED in the
past are restored and
    immediately executed; if false, scheduled activities in the past are marked as
aborted. Future scheduled
    activities are restored and configured ready for execution at the scheduled time
-->
    <!-- Define a new bot with name 'Bot Test 1', which is enabled by default and will
execute the configured
    actions for the state upon initialisation -->
    <bot-definition name="Bot Test 1" execute-on-init="false" enabled="true">
        <!-- Define a bot state -->
        <bot-state name="Nominal Power Line">
            <!-- First condition to be checked: STATION.POWERSUPPLY.PS_TENSION > 200
            Attribute 'value-type' is used to interpret the content of the 'value'
attribute.
            Check the literals defined for enumeration
            eu.dariolucia.reatmetric.api.value.ValueTypeEnum -->
            <condition parameter="STATION.POWERSUPPLY.PS_TENSION" operator="GT" value-
type="REAL" value="200"/>
            <!-- Second condition to be checked (ANDed with the first condition):
            STATION.POWERSUPPLY.PS_OUTPUT == ON -->
            <condition parameter="STATION.POWERSUPPLY.PS_OUTPUT" operator="EQUAL"
value-type="CHARACTER_STRING" value="ON"/>
            <!-- List of actions to be added to the schedule (in the specified order,
with
            trigger 'Now') upon entering this state.
```

```
            Attribute 'max-invocation-time' is in seconds, it defines the
latestExecutionTime
            for activity dispatch, in case the activity cannot be dispatched due to
resource
            conflicts and it has to wait.
            Attribute 'conflict-strategy' defines the strategy to be used in case of
conflict
            upon activity dispatch. Default value is 'WAIT'. -->
            <action activity="STATION.MATRIX.SET_INPUT_STATUS" route="STATION ROUTE"
max-invocation-time="10" conflict-strategy="WAIT">
                <!-- Attribute 'raw-value' indicates if the provided value is raw or
engineered
                (i.e. decalibration will be applied by the processing model, if
defined -->
                <fixed-argument name="ARG1" value="INPUT1" raw-value="false"/>
                <fixed-argument name="ARG2" value="ON" raw-value="false"/>
                <!-- Properties: way to provide custom/driver specific properties,
required for
                activity implementation. -->
                <property key="p1" value="v1" />
                <property key="p2" value="v2" />
                <!-- List of resources -->
                <resources>
                    <resource>station</resource>
                </resources>
            </action>
            <action activity="STATION.MATRIX.WIRING" route="STATION ROUTE" max-
invocation-time="10" conflict-strategy="WAIT">
                <fixed-argument name="ARG1" value="INPUT1" raw-value="false"/>
                <resources>
                    <resource>station</resource>
                </resources>
            </action>
        </bot-state>
        <!-- Define a bot state -->
        <bot-state name="Degraded Power Line 1">
            <condition parameter="STATION.DIESEL_GEN1.DG1_TENSION" operator="GT"
value-type="REAL" value="200"/>
            <condition parameter="STATION.DIESEL_GEN1.DG1_OUTPUT" operator="EQUAL"
value-type="CHARACTER_STRING" value="ON"/>
            <action activity="STATION.MATRIX.SET_INPUT_STATUS" route="STATION ROUTE"
max-invocation-time="10">
                <fixed-argument name="ARG1" value="INPUT2" raw-value="false"/>
                <fixed-argument name="ARG2" value="ON" raw-value="false"/>
                <resources>
                    <resource>station</resource>
                </resources>
            </action>
            <action activity="STATION.MATRIX.WIRING" route="STATION ROUTE" max-
invocation-time="10">
                <fixed-argument name="ARG1" value="INPUT2" raw-value="false"/>
```

```
                <resources>
                    <resource>station</resource>
                </resources>
            </action>
            <action activity="STATION.SWITCH.SW_CMD_POSITION" route="STATION ROUTE"
max-invocation-time="10">
                <fixed-argument name="ARG1" value="POSITION_1" raw-value="false"/>
                <resources>
                    <resource>station</resource>
                </resources>
            </action>
        </bot-state>
        <!-- Define a bot state -->
        <bot-state name="Degraded Power Line 2">
            <condition parameter="STATION.DIESEL_GEN2.DG2_TENSION" operator="GT"
value-type="REAL" value="200"/>
            <condition parameter="STATION.DIESEL_GEN2.DG2_OUTPUT" operator="EQUAL"
value-type="CHARACTER_STRING" value="ON"/>
            <action activity="STATION.MATRIX.SET_INPUT_STATUS" route="STATION ROUTE"
max-invocation-time="10">
                <fixed-argument name="ARG1" value="INPUT2" raw-value="false"/>
                <fixed-argument name="ARG2" value="ON" raw-value="false"/>
                <resources>
                    <resource>station</resource>
                </resources>
            </action>
            <action activity="STATION.MATRIX.WIRING" route="STATION ROUTE" max-
invocation-time="10">
                <fixed-argument name="ARG1" value="INPUT2" raw-value="false"/>
                <resources>
                    <resource>station</resource>
                </resources>
            </action>
            <action activity="STATION.SWITCH.SW_CMD_POSITION" route="STATION ROUTE"
max-invocation-time="10">
                <fixed-argument name="ARG1" value="POSITION_2" raw-value="false"/>
                <resources>
                    <resource>station</resource>
                </resources>
            </action>
        </bot-state>
    </bot-definition>
</ns1:scheduler>
```

Core

Overview

The eu.dariolucia.reatmetric.core module provides an implementation of the IReatmetricSystem

and IReatmetricRegister interfaces specified in the API module. A IReatmetricRegister provides a list of systems, which are available to be used: the implementation provided by this module returns always a single IReatmetricSystem implementation. The IReatmetricSystem is the main access interface to the functionalities provided by a ReatMetric system with respect to users and external world.

This module defines (and implements in some parts) an internal API, which is needed to develop plug-ins and extensions to the system:

- It provides the specification of the IDriver interface (and related interfaces), which must be implemented in order to extend the monitoring and control support to specific devices and systems;
- It provides the specification and implementation of the IOperationalMessageBroker and IRawDataBroker, which provide means, for internal drivers and functions, to generate, distribute and store operational messages (including managing their acknowledgements) and raw data.
- It provides the specification of the IRawDataRenderer, which is a support object to be provided by drivers. An implementation provides the generic extraction of information from raw data objects handled by the driver.
- It provides the specification and an implementation of the IServiceCoreContext interface, which is the context object provided to the drivers upon system initialisation. From such object, a driver can access all system core functions.
- It provides an implementation of all the processing model service factories, to access live and historical data generated by the processing model.

With respect to operational messages, the ReatMetric Core initialises an internal log handler, which registers itself to the java.util.logging framework, and translate all log messages raised with a level equal to or greater than INFO to corresponding operational messages.

When the IReatmetricSystem implementation is instantiated by the IReatmetricRegister implementation, the system is initialised in the following steps:

1. The 'core' configuration file is loaded from the location specified in the VM property *reatmetric.core.config*;
2. The path to the logging configuration file is retrieved from the core configuration and applied;
3. If the use of the archive is specified in the core configuration, then the archive is initialised via the standard Java ServiceLoader. An implementation of the IArchive interface must be present as registered Java service in the Java VM;
4. The brokers (for operational/acknowledged messages and raw data) are initialised.
5. The processing model is initialised according to the configuration path specified in the core configuration. The path shall point to a folder containing all the definition files that must be loaded;
6. If the use of the scheduler is specified in the core configuration, then the scheduler implementation is instantiated via the standard Java ServiceLoader, via interface ISchedulerFactory. An implementation of the ISchedulerFactory interface must be present as

registered Java service in the Java VM;

7. The specified list of drivers is loaded, one by one, in the order declared in the core configuration file. For each driver, the connectors, activity handlers and raw data renderers are retrieved and registered;
8. If the scheduler is instantiated, then it is initialised.
9. The connectors to be started automatically upon system start-up are started.
10. The final system status is derived: if a driver could not be initialised or it is in a non-nominal condition, the system status can be set to WARNING or ALARM (the highest severity value is used). If everything is OK, the system status is set to NOMINAL. It must be clarified that the system status refers to the internal status of the initialisation only, and it has nothing to do with e.g. failed connections to external devices. A system status set to WARNING or ALARM means that one or more drivers are not properly working as expected and their configuration shall be checked.

This module has two internal dependency:

- On eu.dariolucia.reatmetric.api
- On eu.dariolucia.reatmetric.processing

This module has one external dependencies:

- On JAXB library, since the configuration is defined in XML files.

Configuration

The configuration structure of the eu.dariolucia.reatmetric.core module is defined in the package eu.dariolucia.reatmetric.core.configuration. It is an XML file using namespace definition *http://dariolucia.eu/reatmetric/core/configuration*.

The elements that can contain a path support the special value $HOME, which is replaced at runtime with the contents of the *user.home* system variable.

A commented example of such file is presented below.

```
<ns1:core xmlns:ns1="http://dariolucia.eu/reatmetric/core/configuration">
    <!-- Name of the system (a string).
    This element is optional.
    -->
    <name>Test System</name>
    <!-- Path to the java.util.logging configuration file.
    This element is optional. See below for further explanation.
    -->
    <log-property-file>$HOME\reatmetric\log.properties</log-property-file>
    <!-- Archive configuration location: the string depends on the IArchive
implementation
    and it might not be necessary a path to a folder. It could be a JDBC connection
string,
    for instance, or any other string that is needed to configure the specific
```

```
IArchive
    implementation.
    This element is optional. If not present, no archive is used.
    -->
    <archive-location>$HOME\reatmetric\archive</archive-location>
    <!-- Processing model definition location: path to the folder containing the XML
files
    with the definitions for the processing model.
    This element is mandatory.
    -->
    <definitions-location>$HOME\reatmetric\processing</definitions-location>
    <!-- Processing model init strategy:
    -   If no element is provided, the processing model is not initialised from any
        archived data
    -   If the init-resume element is provided, each parameter state in the processing
model
        is initialised with the latest parameter state present in the archive, up to
        'look-back-time' seconds in the past.
    -   If the init-from-time element is provided, each parameter state in the
processing
        model is initialised with the parameter state at the time specified by 'time',
up
        to 'look-back-time' seconds in the past. If the 'archive-location' attribute
is
        present, then a new IArchive implementation, configured with the string in the
        'archive-location' attribute is instantiated, and it is used as source of the
        archived data.
    -->
    <!-- <init-from-time look-back-time="3600" time="2023-02-09T12:32:32Z"
            archive-location="$HOME\reatmetric\another_archive" /> -->
    <init-resume look-back-time="3600" />
    <!-- Scheduler configuration location: the string depends on the IScheduler
implementation
    and it might not be necessary a path to a folder. It could be any other string
that is
    needed to configure the specific IScheduler implementation.
    This element is optional. If not present, no scheduler is used.
    -->
    <scheduler-configuration>$HOME\reatmetric\scheduler</scheduler-configuration>
    <!-- List of <driver> elements. Each driver must be configured with a name (any
string),
    a type (the qualified class name of the driver) and a configuration string, which
can be
    a path to a folder or any other string, depending on the driver implementation.
    -->
    <driver name="Driver 1"
            type="my.driver.for.device1.MyDriver1"
            configuration="$HOME\reatmetric\driver1" />
    <driver name="Driver 2"
            type="my.driver.for.device1.MyDriver2"
            configuration="$HOME\reatmetric\driver2" />
```

```
    <driver name="Driver 1 Test"
            type="my.driver.for.device1.MyDriver1"
            configuration="$HOME\reatmetric\driver1_test" />
    <!-- The autostart-connectors element indicates whether the connectors made
available
    by the different drivers shall be started automatically after the initialisation
of
    the system, and if automated reconnection shall be configured by default.
    It is possible to exclude specific connectors from this behaviour, by listing them
    using element 'startup-exclusion' and 'reconnect-exclusion'.
    This element is optional. If not present, connectors are not automatically started
    and automated reconnection behavior is not set.
    -->
    <autostart-connectors startup="true" reconnect="true">
        <startup-exclusion>Connector 1 Name</startup-exclusion>
        <startup-exclusion>Connector 2 Name</startup-exclusion>
        <reconnect-exclusion>Connector 2 Name</reconnect-exclusion>
    </autostart-connectors>
</ns1:core>
```

When instantiated, the implementation looks for a system variable, containing the absolute path of the XML file with the configuration. It is therefore mandatory to include such variable when starting up the application containing this module, or to set up such variable programmatically, before loading the corresponding service via the ServiceLoader.

Example:

```
java -Dreatmetric.core.config=/home/reatmetric/rm.core.config.xml ...
```

ReatMetric uses the java.util.logging implementation to log messages and traces. The ReatMetric Core module is responsible for the initialisation of the logging, based on the contents of the provided configuration file, as specified by the <log-property-file> element. An example of the log configuration file can be seen below.

```
handlers = java.util.logging.ConsoleHandler, java.util.logging.FileHandler

.level = OFF
eu.dariolucia.level = INFO
eu.dariolucia.ccsds.tmtc.cop1.fop.level = ALL
eu.dariolucia.reatmetric.driver.spacecraft.tmtc.level = ALL
eu.dariolucia.reatmetric.driver.automation.internal.level = ALL

java.util.logging.ConsoleHandler.level = ALL

java.util.logging.FileHandler.level = ALL
java.util.logging.FileHandler.pattern=/home/user/reatmetric.log
java.util.logging.FileHandler.limit=5000000
java.util.logging.FileHandler.count=1
```

```
java.util.logging.FileHandler.formatter=java.util.logging.SimpleFormatter
```

The above file configures the logging to log on the console and on a file. The logging levels can be assigned per package and per handler: in the example above, logging is disabled for all packages (.level = OFF), and it is then selectively enabled at INFO level for the eu.dariolucia package (recursively). Three specific packages have the log level specified without any filtering (level = ALL).

For the file handler it is possible to specify the path to the log file (pattern), the maximum file size (limit) and the number of log files (count) before start rotating them. Further information about the java.util.logging configurability and usage can be found on the Oracle tutorial https://docs.oracle.com/javase/8/docs/technotes/guides/logging/overview.html

It is not recommended to reduce the log levels under INFO for the eu.dariolucia.reatmetric package, otherwise many operational messages will not be generated and shown to the user by the ReatMetric Core logging handler.

Remoting

Overview

The eu.dariolucia.reatmetric.remoting module provides an application to publish a IReatmetricSystem implementation on a RMI registry, in order to make it accessible via Java RMI to the ReatMetric UI application and to other external applications via Java RMI. In this way, a ReatMetric system can run as a back-end application, regardless of the presence of a connected user interface.

The application requires one mandatory parameter and one optional parameter:

- The first, mandatory, parameter indicates the TCP/IP port, the application will use to register the RMI Naming Service;
- The second, optional, parameter specifies the name of the ReatMetric system that must be registered to the RMI Naming Service. If no name is provided, then the application registers all ReatMetric systems that are found.

This module looks up an implementation of the IReatmetricRegistry using the Java ServiceLoader. This means that, if this application needs to be used in conjunction with the eu.dariolucia.reatmetric.core implementation, all the prerequisites in terms of system variable and configurations of such module must be respected. The module then registers the IReatmetricSystem instance using the system name of the instance.

This module has one internal dependency:

- On eu.dariolucia.reatmetric.api

This module has no external dependencies.

Configuration

None.

Remoting Connector

Overview

The eu.dariolucia.reatmetric.remoting.connector module provides an implementation of the IReatmetricSystem and IReatmetricRegister interfaces specified in the API module, which are in reality stubs, connecting to remotely activated IReatMetric systems via the Remoting module, using Java RMI. The IReatmetricRegister of this implementation provides a list of systems, as defined in the related configuration file.

When the IReatmetricRegister of this module is instantiated, the system is initialised in the following steps:

1. The 'connector' configuration file is loaded from the location specified in the VM property *reatmetric.remoting.connector.config*;
2. For each system entry in the configuration file, a corresponding stub implementation of a IReatmetricSystem is instantiated and made available for retrieval with the *availableSystems()* method.

This module can be therefore used by external applications and libraries, which have the need to connect to the services provided by a ReatMetric system in an efficient way, using Java RMI.

This module has one internal dependency:

- On eu.dariolucia.reatmetric.api

This module has one external dependencies:

- On JAXB library, since the configuration is defined in XML files.

Configuration

The configuration structure of the eu.dariolucia.reatmetric.remoting.connector module is defined in the package eu.dariolucia.reatmetric.remoting.connector.configuration. It is an XML file using namespace definition *http://dariolucia.eu/reatmetric/remoting/connector/configuration*.

A commented example of such file is presented below.

```
<ns1:connectors xmlns:ns1=
"http://dariolucia.eu/reatmetric/remoting/connector/configuration">
    <!-- Declare a ReatMetric system, remotely exposed with the given remote name
(Test System),
    on the specified host and port, as a local ReatMetric system, with the provided
    local name (Prime System). -->
    <connector local-name="Test System" remote-name="Test System" host="10.0.8.1"
```

```
port="19000" />
    <!-- Declare a ReatMetric system, remotely exposed with the given remote name
(Test System),
    on the specified host and port, as a local ReatMetric system, with the provided
    local name (Backup System). -->
    <connector local-name="Backup System" remote-name="Test System" host="10.0.8.2"
port="19000" />
</ns1:connectors>
```

When instantiated, the implementation looks for a system variable, containing the absolute path of the XML file with the configuration. It is therefore mandatory to include such variable when starting up the application containing this module, or to set up such variable programmatically, before loading the corresponding service via the ServiceLoader.

Example:

```
java
-Dreatmetric.remoting.connector.config=/home/reatmetric/rm.remoting.connector.config.x
ml ...
```

Driver Modules

Automation

Overview

The eu.dariolucia.reatmetric.driver.automation.base module provides a common codebase and utility classes to implement language-specialized automation functions. Three different automation modules are provided by ReatMetric:

- eu.dariolucia.reatmetric.driver.automation.groovy: automation implementation that uses Groovy as scripting language. This is the module to use for reliable and optimised script execution.
  - Activity type: GROOVY_SCRIPT
  - Activity route: Groovy Automation Engine
  - Script file extension: groovy
- eu.dariolucia.reatmetric.driver.automation.js: automation implementation that uses Javascript as scripting language via GraalVM.
  - Activity type: JS_SCRIPT
  - Activity route: JS Automation Engine
  - Script file extension: js
- eu.dariolucia.reatmetric.driver.automation.python: experimental automation implementation that uses Python as scripting language via Jython. Jython is an old Python 2 implementation in Java, therefore its use is discouraged: plans exist to migrate this module to use JEP, with full Python 3 support.
  - Activity type: PYTHON_SCRIPT
  - Activity route: Python Automation Engine
  - Script file extension: py

Being a driver, an *automation* module must be registered as such in the system Core's configuration. The module provides an IActivityHandler implementation, capable to map an activity occurrence to the corresponding script file to be executed.

This module has the following internal dependencies:

- On eu.dariolucia.reatmetric.api
- On eu.dariolucia.reatmetric.core

This module has one external dependencies:

- On JAXB library, since the configuration is defined in XML files.

Automation Language API

Script structure

A script must be a sequence of Javascript/Groovy/Python statements, which will be executed when the script is evaluated by the script engine. The statements shall not be placed into a function wrapper, even though Javascript/Groovy/Python functions can be specified at the beginning of the script file.

API function names as well as the variable names 'FILENAME' and '_scriptManager' are reserved and shall not be used as declared variable names.

Declaration of arguments

Script arguments do not need declaration: the automation engine binds all the activity arguments to equivalent language symbols. The only argument which is not mapped is the argument named "FILENAME", which is the reserved argument name to map to the specific script file inside the script directory.

API functions

The automation engine provides to scripts the following API functions.

Log messages

```
info(String message):void
```

```
warning(String message):void
```

```
alarm(String message):void
```

Raise a message with the specified severity as operational message.

State fetching

```
parameter(String paramPath):ParameterData
```

Return the current state (as ParameterData) of the specified parameter. Can return null if no such parameter can be located in the processing model.

```
event(String eventPath):EventData
```

Return the current state (as EventData) of the specified event. Can return null if no such event can be located in the processing model.

```
wait_for_event(String eventPath, int timeoutSeconds):EventData
```

Wait timeoutSeconds for the next occurrence of the specified event. Return the event (as EventData) of the specified event, if received during the specified time. Can return null if no such event is received during the specified time.

```
wait_for_parameter(String parameterPath, int timeoutSeconds):ParameterData
```

Wait timeoutSeconds for the next occurrence of the specified parameter. Return the parameter (as

ParameterData) of the specified parameter, if received during the specified time. Can return null if no such parameter is received during the specified time.

Parameter injection

`inject_parameter(String paramPath, Object value):boolean`

Inject the source value of the specified parameter. Return true if the function call to the processing model went OK, false in case of injection exception.

Event injection

`raise_event(String eventPath):boolean`

Raise the specified event. Return true if the function call to the processing model went OK, false in case of injection exception.

`raise_event(String eventPath, String qualifier):boolean`

Raise the specified event with the attached qualifier. Return true if the function call to the processing model went OK, false in case of injection exception.

`raise_event(String eventPath, String qualifier, Object report):boolean`

Raise the specified event with the attached qualifier and report. Return true if the function call to the processing model went OK, false in case of injection exception.

Activity execution and monitoring

`prepare_activity(String activityPath):ActivityInvocationBuilder`

Prepare the invocation of an activity. Return an object that can be used to customise the activity execution and invoke it subsequently. Can return null if no such activity can be located in the processing model..

`ActivityInvocationBuilder::with_route(String route):ActivityInvocationBuilder`

Set the activity invocation route. Return the builder object.

`ActivityInvocationBuilder::with_property(String k, String v):ActivityInvocationBuilder`

Set a property. Return the builder object.

`ActivityInvocationBuilder::with_argument(String name, Object value, boolean engineering):ActivityInvocationBuilder`

Set a plain argument. Return the builder object.

`ActivityInvocationBuilder::with_argument(AbstractActivityArgument arg):ActivityInvocationBuilder`

Set an argument (plain or array). Return the builder object.

`ActivityInvocationBuilder::execute():ActivityInvocationResult`

Execute the activity. Return an ActivityInvocationResult object that can be used to asynchronously

monitor the status of the activity.

```
ActivityInvocationBuilder::execute_and_wait():boolean;
```

Execute the activity and wait for its completion. Return true if the activity execution completed with success, otherwise false.

```
ActivityInvocationBuilder::execute_and_wait(int timeoutSeconds):boolean
```

Execute the activity and wait for its completion in the specified time in seconds. Return true if the activity execution completed with success, otherwise (also in case of timeout) false.

```
ActivityInvocationBuilder::prepare_schedule(String source, String externalId, Integer
expectedDurationSeconds):SchedulingActivityInvocationBuilder
```

Starting preparing a scheduling request for the activity. If expectedDurationSeconds is null, then the expected duration as per activity definition is used.

```
SchedulingActivityInvocationBuilder::with_resource(String
resource):SchedulingActivityInvocationBuilder
```

Add a resource to the resource set of the scheduling request.

```
SchedulingActivityInvocationBuilder::with_resources(Collection<String>
resources):SchedulingActivityInvocationBuilder
```

Add resources to the resource set of the scheduling request.

```
SchedulingActivityInvocationBuilder::with_resources(String…
resources):SchedulingActivityInvocationBuilder
```

Add resources to the resource set of the scheduling request.

```
SchedulingActivityInvocationBuilder::with_latest_invocation_time(Instant
time):SchedulingActivityInvocationBuilder
```

Set the latest invocation time to the scheduling request.

```
SchedulingActivityInvocationBuilder::with_conflict_strategy(ConflictStrategy
conflictStrategy):SchedulingActivityInvocationBuilder
```

Set the conflict strategy for the activity execution.

```
SchedulingActivityInvocationBuilder::with_creation_conflict_strategy(CreationConflictStrategy
strategy):SchedulingActivityInvocationBuilder
```

Set the conflict strategy for the scheduling request.

```
SchedulingActivityInvocationBuilder::schedule_absolute(Instant scheduledTime):boolean
```

Schedule the activity at the specified time.

```
SchedulingActivityInvocationBuilder::schedule_relative(int delaySeconds, String…
predecessors):boolean
```

Schedule the activity to start after the completion of all listed predecessors (external IDs), and with the addition of the specified delay in seconds.

`SchedulingActivityInvocationBuilder::schedule_event(String eventPath, int millisecondsProtectionTime):boolean`

Schedule the activity to start when the specified event is detected, and with the specified protection time.

`ActivityInvocationResult::wait_for_completion():boolean`

Wait for the activity to complete. Return true if the activity execution completed with success, otherwise false.

`ActivityInvocationResult::wait_for_completion(int timeoutSeconds):boolean`

Wait for the activity to complete in the specified time in seconds. Return true if the activity execution completed with success, otherwise (also in case of timeout) false.

`ActivityInvocationResult::is_invocation_failed():boolean`

Return true if the activity invocation failed, otherwise false.

`ActivityInvocationResult::is_completed():boolean`

Return true if the activity execution completed, otherwise false.

`ActivityInvocationResult::current_status():ActivityReportState`

Return the current status (as ActivityReportState) of the activity execution.

`delete_scheduled_activity(String externalId):boolean`

Delete the specified entry in the scheduler. Return true if the entry was deleted, otherwise false.

How to return a value

For Javascript and Groovy it is enough to evaluate the variable as last instruction at the end of the script. For instance, if the script uses the variable `theResult` and the value must be reported, it is enough to declare, at the end of the script:

```
theResult;
```

For Python, the special variable name `_result` shall be used to report the return value, since Python does not return a value using the last evaluated expression.

Configuration

Being a driver, the automation implementation driver must be registered as such in the system configuration file.

```
<ns1:core xmlns:ns1="http://dariolucia.eu/reatmetric/core/configuration">
    <name>Test System</name>
    <log-property-file>$HOME\Reatmetric\reatmetric_test\log.properties</log-property-file>
    <definitions-location>$HOME\Reatmetric\reatmetric_test\processing</definitions-location>
```

```
    <driver name="Automation Driver" type=
"eu.dariolucia.reatmetric.driver.automation.groovy.GroovyAutomationDriver"
    configuration="$HOME\Reatmetric\reatmetric_test\automation"/>
</ns1:core>
```

The folder specified in the *configuration* attribute of the *driver* element must contain a file named *configuration.xml*, which defines the configuration properties of the specific automation driver (in the example, the Groovy implementation).

The configuration structure of the eu.dariolucia.reatmetric.driver.automation.base module is defined in the package eu.dariolucia.reatmetric.driver.automation.base.definition. It is an XML file named *configuration.xml* using namespace definition *http://dariolucia.eu/reatmetric/driver/automation*.

Such configuration is used by all automation module implementations (Groovy, Python, Javascript), i.e. such implementation modules do not have language-specific configuration.

The elements that can contain a path support the special value $HOME, which is replaced at runtime with the contents of the *user.home* system variable.

A commented example of such file is presented below.

```
<ns1:automation xmlns:ns1="http://dariolucia.eu/reatmetric/driver/automation">
    <!-- Optional element: number of scripts that can be executed in parallel by the
automation driver. Default is 1. -->
    <max-parallel-scripts>2</max-parallel-scripts>
    <!-- Mandatory element: the folder containing the script files -->
    <script-folder>$HOME\Reatmetric\Automation\Scripts</script-folder>
</ns1:automation>
```

In order to map an activity to a script execution, the processing model definition of the activity must contain at least one argument named *FILENAME* and type *GROOVY_SCRIPT* for Groovy scripts, *JS_SCRIPT* for Javascript scripts, *PYTHON_SCRIPT* for Python scripts. An example follows below.

```
<activity id="#79" location="ROOT.ELEMENT.SUBELEMENT.OPERATION1" description=
"Operation 1 of sub-element"
type="GROOVY_SCRIPT">
    <argument name="FILENAME" raw_type="CHARACTER_STRING" eng_type="CHARACTER_STRING"
eng_unit="" fixed="true">
        <default_fixed type="RAW" value="script1.groovy"/>
    </argument>
    <argument name="arg1" raw_type="UNSIGNED_INTEGER" eng_type="UNSIGNED_INTEGER" />
    <argument name="arg2" raw_type="CHARACTER_STRING" eng_type="CHARACTER_STRING" />
    <argument name="arg3" raw_type="REAL" eng_type="REAL" />
</activity>
```

In the example, the *FILENAME* argument is fixed and it is used to block the value of such argument

to 'script1.groovy'. Such file must be present inside the *script-folder* as specified in the configuration file. The other arguments must be specified at invocation time, as for normal activities, and the specified values are mapped to the variables with the same argument name, defined in the script. It is of course also possible to have a non-fixed FILENAME argument: in that case, the name of the script to be executed can be provided as argument at activity invocation time.

An example of groovy script is provided below.

```
// arg1 and arg2 are from the activity arguments, arg3 is not used
info("Test message from script1.groovy with values " + arg1 + ", " + arg2)
123
```

The script above raises a log message with severity *INFO*, and then returns, as activity result, the value *123*.

HTTP Server

Overview

The eu.dariolucia.reatmetric.driver.httpserver module provides a driver that allows accessing ReatMetric functions and interfaces using a JSON/REST API.

Being a driver, a *httpserver* module must be registered as such in the system Core's configuration.

The module also provides a Javascript library as part of its resources. Examples to use this library are provided in the HTML files delivered with this module.

This module has the following internal dependencies:

- On eu.dariolucia.reatmetric.api
- On eu.dariolucia.reatmetric.core

This module has two external dependencies:

- On JAXB library, since the configuration is defined in XML files.
- On jsonpath library, used to handle JSON objects.

Protocol

When the driver is in use, the following API will be available to the specified IP and port.

Parameter descriptor list

```
GET     http://<host>:<port>/<system name>/parameters/list
```

The request returns the list of parameters defined in the system, as an array of objects so defined:

```
{
"type" : "PARAMETER",
"path" : <string>,
"externalId" : <integer>,
"description" : <string> or null,
"rawDataType" : <string>, // See ValueTypeEnum enum names in
eu.dariolucia.reatmetric.api.value
"engDataType" : <string>, // See ValueTypeEnum enum names in
eu.dariolucia.reatmetric.api.value
"unit" : <string> or null,
"synthetic" : <boolean>,
"settable" : <boolean>
}
```

Javascript library function: async listParameters() : return array of parameter descriptors.

Event descriptor list

```
GET     http://<host>:<port>/<system name>/events/list
```

The request returns the list of events defined in the system, as an array of objects so defined:

```
{
"type" : "EVENT",
"path" : <string>,
"externalId" : <integer>,
"description" : <string> or null,
"severity" : <string>, // See Severity enum names in
eu.dariolucia.reatmetric.api.messages
"eventType" : <string> or null
}
```

Javascript library function: async listEvents() : return array of event descriptors.

Activity descriptor list

```
GET     http://<host>:<port>/<system name>/activities/list
```

The request returns the list of activities defined in the system, as an array of objects so defined:

```
{
"type" : "ACTIVITY",
"path" : <string>,
"externalId" : <integer>,
"description" : <string> or null,
```

```
"activityType" : <string>,
"defaultRoute" : <string>,
"arguments" : <array of argument descriptors>,
"properties" : JSON object // associative array
}
```

An argument descriptor can be plain or array. A plain argument descriptor is defined as:

```
{
"name" : <string>,
"description" : <string> or null,
"type" : "plain",
"rawDataType" : <string>, // See ValueTypeEnum enum names in
eu.dariolucia.reatmetric.api.value
"engDataType" : <string>, // See ValueTypeEnum enum names in
eu.dariolucia.reatmetric.api.value
"unit" : <string> or null,
"fixed" : <boolean>,
"decalibrationPresent" : <boolean>
}
```

An array argument descriptor is defined as:

```
{
"name" : <string>,
"description" : <string> or null,
"type" : "array",
"expansionArgument" : <string>,
"elements" : <array of argument descriptors>
}
```

Javascript library function: async listActivities() : return array of event descriptors.

Parameter state subscription

This type of interaction allows clients to request the creation of a specific parameter subscription, based on the provided filter. On the server, parameters matching the subscription are kept in a map, where the key is the parameter path. Upon each GET, the map values are returned to the caller and the map is cleared. Hence the interaction, by design, returns only the latest parameter state available at the time of each GET, if there were changes with respect to its previously returned state. This type of interaction can be used to retrieve information to be used in ANDs and mimics.

Subscription

```
POST    http://<host>:<port>/<system name>/parameters/current/register
```

The request must provide in its body a ParameterDataFilter in JSON format, according to the

following structure:

```
{
"parentPath" : <string> or null,
"parameterPathList" : <array of string> or null,
"routeList" : <array of string> or null,
"validityList" : <array of string> or null, // See Validity enum names in
eu.dariolucia.reatmetric.api.parameters
"alarmStateList" : <array of string> or null, // See AlarmState enum names in
eu.dariolucia.reatmetric.api.model
"externalIdList" : <array of integer> or null
}
```

The response returns in its body a key value (UUID) in the following structure:

```
{
"key" : <string> // UUID string
}
```

The effect of this invocation is the creation, on the server side, of a subscription that will deliver the related parameters via a GET request (see below).

Javascript library function: async registerToStateParameters(filter) : return the key value as string.

Javascript library function: parameterFilter(parentPath, parameterPathList, routeList, validityList, alarmStateList, externalIdList) : return a parameter filter object.

Removal of subscription

```
DELETE  http://<host>:<port>/<system name>/parameters/current/deregister/<key>
```

The effect of this invocation is the removal, on the server side, of the subscription linked to the specified key.

Javascript library function: async deregisterFromStateParameters(key) : **void**

Parameter fetch

```
GET     http://<host>:<port>/<system name>/parameters/current/get/<key>
```

The request returns the list of ParameterData object related to the subscription. Returned objects are removed from the server side map. The array of objects so defined:

```
{
"internalId" : <integer>,
```

```
"gentime" : <time string>, // Format as YYYY-MM-DD'T'hh:mm:ss.SSSZ
"externalId" : <integer>,
"path" : <string>,
"eng" : <string or time string or integer or float or boolean> or null,
"raw" : <string or time string or integer or float or boolean> or null,
"rcptime" : <time string>, // Format as YYYY-MM-DD'T'hh:mm:ss.SSSZ
"route" : <string> or null,
"validity" : <string>, // See Validity enum names in
eu.dariolucia.reatmetric.api.parameters
"alarm" : <string> // See AlarmState enum names in eu.dariolucia.reatmetric.api.model
}
```

Javascript library function: async getStateParameters(key) : return array of parameter data objects.

Parameter stream subscription

This type of interaction allows clients to request the creation of a specific parameter subscription, based on the provided filter. On the server, each parameter update matching the subscription is kept in a queue. Upon each GET, the values in the queue are returned to the caller and the queue is cleared. Hence the interaction, by design, returns all the parameter samples generated by the system from a previous GET. This type of interaction can be used to retrieve information to be used in plots and charts.

The driver has a maximum queue size per subscription.

Subscription

```
POST    http://<host>:<port>/<system name>/parameters/stream/register
```

The request must provide in its body a ParameterDataFilter in JSON format, according to the structure defined above.

The response returns in its body a key value according to the structure defined above.

The effect of this invocation is the creation, on the server side, of a subscription that will deliver the related parameters via a GET request (see below).

Javascript library function: async registerToStreamParameters(filter) : return the key value as string.

Removal of subscription

```
DELETE  http://<host>:<port>/<system name>/parameters/stream/deregister/<key>
```

The effect of this invocation is the removal, on the server side, of the subscription linked to the specified key.

Javascript library function: async deregisterFromStreamParameters(key) : **void**

Parameter fetch

```
GET    http://<host>:<port>/<system name>/parameters/stream/get/<key>
```

The request returns the list of ParameterData objects related to the subscription. Returned objects are removed from the server side queue. The array of objects is defined according to the structure defined above.

Javascript library function: async getStreamParameters(key) : return array of parameter data objects.

Parameter current state

This operation allows to request the current state of a parameter, as currently stored in the processing model.

```
GET    http://<host>:<port>/<system name>/parameters/state?path=<parameter path>
```

```
GET    http://<host>:<port>/<system name>/parameters/state?id=<parameter external ID>
```

The response returns in its body a single ParameterData object in JSON format (see above "Parameter state subscription").

Javascript library function: async getParameterByPath(path) : return a parameter data object.

Javascript library function: async getParameterByID(id) : return a parameter data object.

Event stream subscription

This type of interaction allows clients to request the creation of a specific event subscription, based on the provided filter. On the server, each event update matching the subscription is kept in a queue. Upon each GET, the values in the queue are returned to the caller and the queue is cleared. Hence the interaction, by design, returns all the events generated by the system from a previous GET. This type of interaction can be used to retrieve information to be used in scrollable displays.

The driver has a maximum queue size per subscription.

Subscription

```
POST    http://<host>:<port>/<system name>/events/register
```

The request must provide in its body a EventDataFilter in JSON format, according to the following structure.

```
{
"parentPath" : <string> or null,
"eventPathList" : <array of string> or null,
"routeList" : <array of string> or null,
"typeList" : <array of string> or null,
"sourceList" : <array of string> or null,
"severityList" : <array of string> or null, // See Severity enum names in
eu.dariolucia.reatmetric.api.messages
"externalIdList" : <array of integer> or null
}
```

The response returns in its body a key value according to the structure defined above.

The effect of this invocation is the creation, on the server side, of a subscription that will deliver the related events via a GET request (see below).

Javascript library function: async registerToEvents(filter) : return the key value as string.

Javascript library function: eventFilter(parentPath, eventPathList, sourceList, routeList, typeList, severityList, externalIdList) : return an event filter object.

Removal of subscription

```
DELETE  http://<host>:<port>/<system name>/events/deregister/<key>
```

The effect of this invocation is the removal, on the server side, of the subscription linked to the specified key.

Javascript library function: async deregisterFromEvents(key) : **void**

Event fetch

```
GET     http://<host>:<port>/<system name>/events/get/<key>
```

The request returns the list of EventData objects related to the subscription. Returned objects are removed from the server side queue. The array of objects is so defined.

```
{
"internalId" : <integer>,
"gentime" : <time string>, // Format as YYYY-MM-DD'T'hh:mm:ss.SSSZ
"externalId" : <integer>,
"path" : <string>,
"qualifier" : <string> or null,
"rcptime" : <time string>, // Format as YYYY-MM-DD'T'hh:mm:ss.SSSZ
"type" : <string> or null,
"route" : <string> or null,
"source" : <string> or null,
```

```
"severity" : <string> // See Severity enum names in
eu.dariolucia.reatmetric.api.messages
}
```

Javascript library function: async getEvents(key) : return array of event data objects.

Raw Data stream subscription

This type of interaction allows clients to request the creation of a specific raw data subscription, based on the provided filter. On the server, each raw data update matching the subscription is kept in a queue. Upon each GET, the values in the queue are returned to the caller and the queue is cleared. Hence the interaction, by design, returns all the raw data generated by the system from a previous GET. This type of interaction can be used to retrieve information to be used in scrollable displays.

The driver has a maximum queue size per subscription.

Subscription

```
POST    http://<host>:<port>/<system name>/rawdata/register
```

The request must provide in its body a RawDataFilter in JSON format, according to the following structure.

```
{
"nameContains" : <string>,
"contentSet" : <boolean>,
"routeList" : <array of string> or null,
"typeList" : <array of string> or null,
"sourceList" : <array of string> or null,
"qualityList" : <array of string> or null // See Quality enum names in
eu.dariolucia.reatmetric.api.rawdata
}
```

The response returns in its body a key value according to the structure defined above.

The effect of this invocation is the creation, on the server side, of a subscription that will deliver the related raw data via a GET request (see below).

Javascript library function: async registerToRawData(filter) : return the key value as string.

Javascript library function: rawDataFilter(contentSet, nameContains, sourceList, routeList, typeList, qualityList) : return a raw data filter object.

Removal of subscription

```
DELETE  http://<host>:<port>/<system name>/rawdata/deregister/<key>
```

The effect of this invocation is the removal, on the server side, of the subscription linked to the specified key.

Javascript library function: async deregisterFromRawData(key) : **void**

Event fetch

```
GET    http://<host>:<port>/<system name>/rawdata/get/<key>
```

The request returns the list of RawData objects related to the subscription. Returned objects are removed from the server side queue. The array of objects is so defined.

```
{
"internalId" : <integer>,
"gentime" : <time string>, // Format as YYYY-MM-DD'T'hh:mm:ss.SSSZ
"name" : <string>,
"rcptime" : <time string>, // Format as YYYY-MM-DD'T'hh:mm:ss.SSSZ
"type" : <string> or null,
"route" : <string> or null,
"source" : <string> or null,
"quality" : <string>, // See Quality enum names in
eu.dariolucia.reatmetric.api.rawdata
"data" : <Base64 encoded string> or null
}
```

Javascript library function: async getRawData(key) : return array of raw data objects.

Activity stream subscription

This type of interaction allows clients to request the creation of a specific activity subscription, based on the provided filter. On the server, each activity update matching the subscription is kept in a queue. Upon each GET, the values in the queue are returned to the caller and the queue is cleared. Hence the interaction, by design, returns all the activity occurrence updates generated by the system from a previous GET. This type of interaction can be used to retrieve information to be used in scrollable displays.

The driver has a maximum queue size per subscription.

Subscription

```
POST    http://<host>:<port>/<system name>/activities/register
```

The request must provide in its body a ActivityOccurrenceDataFilter in JSON format, according to the following structure.

```
{
"parentPath" : <string> or null,
```

```
"activityPathList" : <array of string> or null,
"routeList" : <array of string> or null,
"typeList" : <array of string> or null,
"sourceList" : <array of string> or null,
"stateList" : <array of string> or null, // See ActivityOccurrenceState enum names in
eu.dariolucia.reatmetric.api.activity
"externalIdList" : <array of integer> or null
}
```

The response returns in its body a key value according to the structure defined above.

The effect of this invocation is the creation, on the server side, of a subscription that will deliver the related activity occurrence updates via a GET request (see below).

Javascript library function: async registerToActivities(filter) : return the key value as string.

Javascript library function: activityFilter(parentPath, activityPathList, sourceList, routeList, typeList, stateList, externalIdList) : return an activity occurrence filter object.

Removal of subscription

```
DELETE  http://<host>:<port>/<system name>/activities/deregister/<key>
```

The effect of this invocation is the removal, on the server side, of the subscription linked to the specified key.

Javascript library function: async deregisterFromActivities(key) : **void**

Activity Occurrence fetch

```
GET     http://<host>:<port>/<system name>/activities/get/<key>
```

The request returns the list of ActivityOccurrenceData objects related to the subscription. Returned objects are removed from the server side queue. The array of objects is so defined.

```
{
"internalId" : <integer>,
"gentime" : <time string>, // Format as YYYY-MM-DD'T'hh:mm:ss.SSSZ
"externalId" : <integer>,
"path" : <string>,
"name" : <string>,
"exectime" : <time string>, // Format as YYYY-MM-DD'T'hh:mm:ss.SSSZ
"type" : <string> or null,
"route" : <string> or null,
"source" : <string> or null,
"currentState" : <string> // See ActivityOccurrenceState enum names in
eu.dariolucia.reatmetric.api.activity
"result" : <string or time string or integer or float or boolean> or null,
```

```
"arguments" : <JSON object with key-value (any value) pairs>,
"properties" : <JSON object with key-value (string) pairs>,
"reports" : <array of reports>:
{
"internalId" : <integer>,
"gentime" : <time string>, // Format as YYYY-MM-DD'T'hh:mm:ss.SSSZ
"name" : <string>,
"exectime" : <time string>, // Format as YYYY-MM-DD'T'hh:mm:ss.SSSZ
"state" : <string> // See ActivityOccurrenceState enum names in
eu.dariolucia.reatmetric.api.activity
"transition" : <string> // See ActivityOccurrenceState enum names in
eu.dariolucia.reatmetric.api.activity
"status" : <string> // See ActivityReportState enum names in
eu.dariolucia.reatmetric.api.activity
"result" : <string or time string or integer or float or boolean> or null
}
}
```

Javascript library function: async getActivities(key) : return array of activity occurrence data objects.

Activity invocation

This type of interaction allows clients to request the execution of a specific activity, based on the provided request.

Invocation

```
POST    http://<host>:<port>/<system name>/activities/invoke
```

The request must provide in its body an ActivityRequest in JSON format, according to the following structure.

```
{
"id" : <integer>,
"path" : <string>,
"route" : <string>,
"arguments" : <array of activity arguments>,
"source" : <string>,
"properties" : <JSON object with key-value (string) pairs>
}
```

The array of activity arguments contains either plain or array activity arguments.

```
Plain:
{
"name" : <string>,
```

```
"type" : "plain",
"value" : <string or time string or integer or float or boolean> or null,
"engineering" : <boolean>
}

    Array:
    {
         "name" : <string>,
         "type" : "array",
         "records" : <array of records>:
         {
            "elements" : <array of activity arguments>
         }
    }
```

The response returns in its body an id value according to the structure defined below.

```
{
"id" : <integer> // Activity occurrence ID
}
```

Javascript library function: async invoke(activityRequest) : return the id as integer.

Javascript library function: activityRequest(activityId, activityPath, activityArguments, activityRoute, activitySource, activityProperties) : return an activity request object.

Javascript library function: plainArgument(name, value, engineering) : return an activity request plain argument.

Javascript library function: arrayArgument(name, records) : return an activity request array argument.

Javascript library function: arrayArgumentRecord(elements) : return an activity request array argument record.

Operational message stream subscription

This type of interaction allows clients to request the creation of a specific operational message subscription, based on the provided filter. On the server, each operational message matching the subscription is kept in a queue. Upon each GET, the values in the queue are returned to the caller and the queue is cleared. Hence the interaction, by design, returns all the messages generated by the system from a previous GET. This type of interaction can be used to retrieve information to be used in scrollable displays.

The driver has a maximum queue size per subscription.

Subscription

```
POST    http://<host>:<port>/<system name>/messages/register
```

The request must provide in its body a OperationalMessageDataFilter in JSON format, according to the following structure.

```
{
"messageTextContains" : <string> or null,
"idList" : <array of string> or null,
"sourceList" : <array of string> or null,
"severityList" : <array of string> or null // See Severity enum names in
eu.dariolucia.reatmetric.api.messages
}
```

The response returns in its body a key value according to the structure defined above.

The effect of this invocation is the creation, on the server side, of a subscription that will deliver the related messages via a GET request (see below).

Javascript library function: async registerToMessages(filter) : return the key value as string.

Javascript library function: messageFilter(messageTextContains, idList, sourceList, severityList) : return an operational message filter object.

Removal of subscription

```
DELETE  http://<host>:<port>/<system name>/messages/deregister/<key>
```

The effect of this invocation is the removal, on the server side, of the subscription linked to the specified key.

Javascript library function: async deregisterFromMessages(key) : **void**

Operational messages fetch

```
GET     http://<host>:<port>/<system name>/messages/get/<key>
```

The request returns the list of OperationalMessageData objects related to the subscription. Returned objects are removed from the server side queue. The array of objects is so defined.

```
{
"internalId" : <integer>,
"gentime" : <time string>, // Format as YYYY-MM-DD'T'hh:mm:ss.SSSZ
"id" : <string> or null,
"message" : <string>,
"source" : <string> or null,
"severity" : <string> // See Severity enum names in
```

```
eu.dariolucia.reatmetric.api.messages
}
```

Javascript library function: async getMessages(key) : return array of operational message data objects.

Model entity descriptor fetch

```
GET     http://<host>:<port>/<system name>/model/<path of system element, with /
separator instead of . separator>
```

If successful, the request returns an object with two properties: - element: it contains the descriptor of the entity referenced by the <path of system element>. If the path is not provided, then null is returned - children: it contains an array with the descriptors of the child elements. If the element has no children, an empty array is returned.

```
{
"element" : <descriptor>,
"children" : [
<descriptor>,
<descriptor>,
<descriptor>,
...
<descriptor>
]
}
```

The descriptors have the following structure.

Container

```
{
"type" : "CONTAINER",
"path" : <string> // full system entity path with . separator
}
```

Parameter

Defined as above ("Parameter descriptor list").

Event

Defined as above ("Event descriptor list").

Activity

Defined as above ("Activity descriptor list").

Javascript library function: async getDescriptor(path) : return the element-children information assigned to the provided path.

Model entity enable/disable

```
POST     http://<host>:<port>/<system name>/model/<path of system element, with /
separator instead of . separator>/[enable|disable]
```

This request enables or disables the processing linked to the specified system element.

Javascript library function: async enable(path) : **void**

Javascript library function: async disable(path) : **void**

Historical data retrieval

These operations can be used to retrieve archived parameter data, event data, activity occurrences and messages.

Parameter

```
POST    http://<host>:<port>/<system name>/parameters/retrieve?startTime=<time in ms
from UNIX epoch>&endTime=<time in ms from UNIX epoch>
```

The body of the request is a ParameterDataFilter in JSON format (see "Parameter state subscription"). The request returns an array of ParameterData in JSON format (see "Parameter state subscription").

Javascript library function: async retrieveParameters(startTime, endTime, filter) : return array of ParameterData objects.

Event

```
POST    http://<host>:<port>/<system name>/events/retrieve?startTime=<time in ms from
UNIX epoch>&endTime=<time in ms from UNIX epoch>
```

The body of the request is a EventDataFilter in JSON format (see "Event stream subscription"). The request returns an array of EventData in JSON format (see "Event stream subscription").

Javascript library function: async retrieveEvents(startTime, endTime, filter) : return array of EventData objects.

Raw Data

```
POST    http://<host>:<port>/<system name>/rawdata/retrieve?startTime=<time in ms from
UNIX epoch>&endTime=<time in ms from UNIX epoch>
```

The body of the request is a RawDataFilter in JSON format (see "Raw data stream subscription"). The request returns an array of RawData in JSON format (see "Raw data stream subscription").

Javascript library function: async retrieveRawData(startTime, endTime, filter) : return array of RawData objects.

Activities

```
POST    http://<host>:<port>/<system name>/activities/retrieve?startTime=<time in ms
from UNIX epoch>&endTime=<time in ms from UNIX epoch>
```

The body of the request is a ActivityOccurrenceDataFilter in JSON format (see "Activity stream subscription"). The request returns an array of ActivityOccurrenceData in JSON format (see "Activity stream subscription").

Javascript library function: async retrieveActivities(startTime, endTime, filter) : return array of ActivityOccurrenceData objects.

Messages

```
POST    http://<host>:<port>/<system name>/messages/retrieve?startTime=<time in ms
from UNIX epoch>&endTime=<time in ms from UNIX epoch>
```

The body of the request is a OperationalMessageDataFilter in JSON format (see "Message stream subscription"). The request returns an array of OperationalMessageData in JSON format (see "Message stream subscription").

Javascript library function: async retrieveMessages(startTime, endTime, filter) : return array of OperationalMessageData objects.

Connectors

Connector list fetch

```
GET     http://<host>:<port>/<system name>/connectors
```

The request returns the list of connectors defined in the system, as an array of objects so defined:

```
{
"name" : <string>,
"description" : <string>,
"alarmState" : <string>, // See AlarmState enum names
"status" : <string>, // See TransportConnectionStatus enum names
"rx" : <integer>, // RX data rate
"tx" : <integer>, // TX data rate
"autoreconnect" : <boolean> // true or false
```

```
}
```

Javascript library function: async getConnectors() : return a list of connector objects

Connector fetch

```
GET    http://<host>:<port>/<system name>/connectors/<connector name>
```

The request returns the connector status, using the same format defined for the connector list.

Javascript library function: async getConnector(name) : return the connector object

```
GET    http://<host>:<port>/<system name>/connectors/<connector name>/properties
```

The request returns the properties defined by the connector, as a JSON object (key-value map).

Javascript library function: async getConnectorProperties(name) : return the connector properties

Connector operations

```
POST    http://<host>:<port>/<system name>/connectors/<connector name/connect
```

The request connects the connector. No body defined.

Javascript library function: async connect(name) : **void**

```
POST    http://<host>:<port>/<system name>/connectors/<connector name/disconnect
```

The request disconnects the connector. No body defined.

Javascript library function: async disconnect(name) : **void**

```
POST    http://<host>:<port>/<system name>/connectors/<connector name/abort
```

The request aborts the connector. No body defined.

Javascript library function: async abort(name) : **void**

```
POST    http://<host>:<port>/<system name>/connectors/<connector name/reconnect
```

The request sets the reconnection flag of the connector. The body is:

```
{
"input" : <boolean> // true or false
}
```

```
POST     http://<host>:<port>/<system name>/connectors/<connector name/initialise
```

The request initialises the connector with the provided set of properties. The body is a JSON object (key-value map).

Scheduler

Scheduler state fetch

```
GET     http://<host>:<port>/<system name>/scheduler
```

The request returns a JSON structure containing the status of the scheduler (enabled, disabled) and the list of the scheduled items (only in state SCHEDULED, WAITING and RUNNING), so defined:

```
{
"enabled" : <boolean>, // true or false
"items" : <array of ScheduledActivityData>
}
```

The structure of a ScheduledActivityData is defined as follows:

```
{
"internalId" : <integer>,
"gentime" : <time string>, // Format as YYYY-MM-DD'T'hh:mm:ss.SSSZ
"request" : <JSON object of type ActivityRequest - see previous sections>,
"activity" : <long> or null, // activity occurrence ID if the activity request was
dispatched, otherwise null
"resources" : <array of string>,
"source" : <string> or null,
"externalId" : <string> or null,
"trigger" : <JSON object - see below>,
"latest" : <time string> or null, // Format as YYYY-MM-DD'T'hh:mm:ss.SSSZ
"startTime" : <time string> or null, // Format as YYYY-MM-DD'T'hh:mm:ss.SSSZ
"duration" : <integer>, // Estimated duration of the activity in milliseconds, -1 if
unknown
"conflict" : <string>, // See ConflictStrategy enum names in
eu.dariolucia.reatmetric.api.scheduler
"state" : <string> // See SchedulingState enum names in
eu.dariolucia.reatmetric.api.scheduler
}
```

The structure of a scheduling trigger is defined as follows:

```
    // Trigger execution as soon as the request is submitted
    {
    "type" : "now"
    }

    // Trigger execution at the specified time
    {
        "type" : "absolute",
        "startTime" : <time string>
    }

    // Trigger execution as soon as predecessors are completed, plus optional delay
    {
        "type" : "relative",
        "predecessors" : <array of string>, // external IDs of the predecessors
        "delay" : <integer> or null // in seconds
    }

    // Trigger execution as soon as event is raised, if enabled
    {
        "type" : "event",
        "path" : <string>, // path of the event
        "protection" : <integer> or null, // in milliseconds
        "enabled" : <boolean>
    }
```

Javascript library function: async getSchedulerState() : return the scheduler state

Scheduled item state fetch

```
GET     http://<host>:<port>/<system name>/scheduler/<scheduled item internal ID>
```

The request returns the scheduled item status, using the same format defined for the scheduler state as ScheduledActivityData. Only scheduled items in state SCHEDULED, WAITING and RUNNING can be retrieved.

Javascript library function: async getScheduledItem(ID) : return the scheduler item object

Scheduler operations

```
POST     http://<host>:<port>/<system name>/scheduler/enable
```

The request enables the scheduler. No body defined.

Javascript library function: async enableScheduler() : **void**

```
POST    http://<host>:<port>/<system name>/scheduler/disable
```

The request disables the scheduler. No body defined.

Javascript library function: async disableScheduler() : **void**

```
DELETE    http://<host>:<port>/<system name>/scheduler/<scheduled item internal ID>
```

The request removes the indicated scheduled item. No body defined.

Javascript library function: async removeScheduledItem(ID) : **void**

```
POST    http://<host>:<port>/<system name>/scheduler/<scheduled item internal
ID>?conflict=<creation conflict strategy>
```

The request updates the indicated scheduled item with the SchedulingRequest information provided in the body. The creation conflict strategy is one of the values defined by the CreationConflictStrategy enumeration in the eu.dariolucia.reatmetric.api.scheduler package. This method can be invoked as long as the request is in SCHEDULED state.

The body is:

```
{
"request" : <JSON object of type ActivityRequest - see previous sections>,
"resources" : <array of string>,
"source" : <string> or null,
"externalId" : <string> or null,
"trigger" : <JSON object - see previous section>,
"latest" : <time string> or null, // Format as YYYY-MM-DD'T'hh:mm:ss.SSSZ
"conflict" : <string>, // See ConflictStrategy enum names in
eu.dariolucia.reatmetric.api.scheduler
"duration" : <integer> // Estimated duration of the activity in milliseconds, -1 if
unknown
}
```

Javascript library function: async updateScheduledItem(ID, updatedRequest, creationConflictStrategy) : **void**

```
POST    http://<host>:<port>/<system name>/scheduler/schedule?conflict=<creation
conflict strategy>
```

The request requests the scheduling of an activity. The body is a SchedulingRequest object.

The response returns in its body an id value according to the structure defined below.

```
{
"id" : <integer> // The scheduled item internal id
}
```

Javascript library function: async schedule(newRequest, creationConflictStrategy) : the ID of the newly created scheduled item

Javascript library function: schedulingRequest(request, resources, source, externalId, trigger, latest, conflict, duration) : return a SchedulingRequest object.

```
POST     http://<host>:<port>/<system name>/scheduler/load?conflict=<creation conflict
strategy>&startTime=<time in ms from UNIX epoch>&endTime=<time in ms from UNIX
epoch>&source=<source>
```

The request updates the schedule by removing all scheduled items previously scheduled by the same source and populating such interval with the new scheduling requests. The body is an array of SchedulingRequests.

Javascript library function: async loadScheduleIncrement(startTime, endTime, source, schedulingRequests, creationConflictStrategy) : **void**

Configuration

Being a driver, the *httpserver* module must be registered as such in the system configuration file.

```
<ns1:core xmlns:ns1="http://dariolucia.eu/reatmetric/core/configuration">
    <name>Test System</name>
    <log-property-file>$HOME\Reatmetric\reatmetric_test\log.properties</log-property-
file>
    <definitions-location>$HOME\Reatmetric\reatmetric_test\processing</definitions-
location>
    <driver name="HTTP Driver" type=
"eu.dariolucia.reatmetric.driver.httpserver.HttpServerDriver"
    configuration="$HOME\Reatmetric\reatmetric_test\http"/>
</ns1:core>
```

The folder specified in the *configuration* attribute of the *driver* element must contain a file named *configuration.xml*, which defines the configuration properties of the driver.

The configuration structure of the eu.dariolucia.reatmetric.driver.httpserver module is defined in the package eu.dariolucia.reatmetric.driver.httpserver.definition. It is an XML file named *configuration.xml* using namespace definition *http://dariolucia.eu/reatmetric/driver/httpserver*.

An example of such file is presented below.

```
<ns1:httpserver xmlns:ns1="http://dariolucia.eu/reatmetric/driver/httpserver"
```

```
    host="127.0.0.1"
    port="8080"
    https="false"
    keystore-password=""
    keystore-type="JKS"
    keystore-location=""
    keymanager-password=""
    keymanager-algorithm="SunX509"
    trustmanager-algorithm="SunX509"
    ssl-protocol="TLS">
</ns1:httpserver>
```

The configuration is pretty self-explainable. When the attribute *https* is set to true, it is important to set the related attributes for the key store, key manager, trust manager and SSL protocol. The driver uses the com.sun.net.HttpServer and com.sun.net.HttpsServer implementations: refer to the related documentation for what concerns the use of each attribute and the possible allowed values. The code to handle the HTTPS configuration is derived from the SO answer here: https://stackoverflow.com/a/2323188/11023497

To generate a keystore, the following command can be used (update the placeholders)

```
$ keytool -genkeypair -keyalg RSA -alias self_signed -keypass <httpserver@keymanager-
password>
-keystore <httpserver@keystore-location> -storepass <httpserver@keystore-password>
```

Remote

Overview

The eu.dariolucia.reatmetric.driver.remote module provides a driver that allows to retrieve parameter, events and invoke activities on a remote ReatMetric system, and to repatriate such information into a local processing model.

Being a driver, a *remote* module must be registered as such in the system Core's configuration. The module provides an IActivityHandler implementation, capable to forward activity requests to be implemented by the remote system to the system itself and to monitor their progress.

The main idea implemented in ReatMetric for the monitoring and control of remote ReatMetric systems from a central ReatMetric system (called *master*) is the following (example):

- Each ReatMetric system (the remote ones and the master) have a processing model, with the hierarchical decomposition in system elements. For instance, the two remote systems might have the following:
  - Remote System A: SITE1.SYSTEM1.(...) and SITE1.SYSTEM2.(...)
  - Remote System B: SITE2.SYSTEM1.(...) and SITE2.SYSTEM2.(...) and SITE2.SYSTEM3.(...)
  - Master System: CENTRAL.(...) and two branches for the site 1 and site 2: CENTRAL.SITE1.(...)

and CENTRAL.SITE2.(…)

- The remote systems have no special configuration, besides the fact that their systems is configured allowing remote connections. See the documentation related to the module eu.dariolucia.reatmetric.remote.
- In order to repatriate the data from each remote system to the master, the master needs to know:
  - How to reach such systems in terms of name, IP and port: to achieve this, a specific configuration and related system variable must be initialised on the master system, according to the documentation provided in the eu.dariolucia.reatmetric.remoting.connector module. This configuration must contain the connection details for all remote systems, i.e. the configuration is a global one.
  - Where to map the root node of the processing model of each remote system into its own processing model: to achieve this, a specific configuration for each driver instance must be specified (see below for the details). In addition, such branches in the master system must be defined as mirrored branches.
- Upon start-up, the *remote* driver scans the configured branch, starting from the configured node, to detect all the defined parameters and events, and it subscribes for these in the remote system.

In order to repatriate the definition of the remote systems into the master system (assuming that they are contained in a single file), it is enough to copy the file into the master system, add the necessary global prefix (path_prefix attribute) and set the mirroring (mirrored attribute). If there is no interest to monitor specific parameter or events at master level, or there is the need to hide activities at master level, it is enough to remove such definitions in the copied file in the master system.

The main limitation of this approach is that **all system entities across all systems must have a unique ID**. Such limitation allows the implementation of a very robust and stable mechanism for system-of-systems deployment configurations with multiple tiers.

This module has the following internal dependencies:

- On eu.dariolucia.reatmetric.api
- On eu.dariolucia.reatmetric.core
- On eu.dariolucia.reatmetric.remoting.connector

This module has one external dependencies:

- On JAXB library, since the configuration is defined in XML files.

Configuration

Being a driver, the *remote* module must be registered as such in the system configuration file. You need to have a remote module registration for every remote system that you need to monitor and control.

```
<ns1:core xmlns:ns1="http://dariolucia.eu/reatmetric/core/configuration">
    <name>Master System</name>
    <log-property-file>$HOME\Reatmetric\reatmetric_test\log.properties</log-property-
file>
    <definitions-location>$HOME\Reatmetric\reatmetric_test\processing</definitions-
location>
    <driver name="Remote System 1" type=
"eu.dariolucia.reatmetric.driver.remote.RemoteDriver"
    configuration="$HOME\Reatmetric\reatmetric_test\system1"/>
    <driver name="Remote System 2" type=
"eu.dariolucia.reatmetric.driver.remote.RemoteDriver"
    configuration="$HOME\Reatmetric\reatmetric_test\system2"/>
</ns1:core>
```

The folder specified in the *configuration* attribute of the *driver* element must contain a file named *configuration.xml*, which defines the configuration properties of the driver.

The configuration structure of the eu.dariolucia.reatmetric.driver.remote module is defined in the package eu.dariolucia.reatmetric.driver.remote.definition. It is an XML file named *configuration.xml* using namespace definition *http://dariolucia.eu/reatmetric/driver/remote*.

An example of such file is presented below.

```
<ns1:remote xmlns:ns1="http://dariolucia.eu/reatmetric/driver/remote"
    remote-system-name="Test System Station 1"
    remote-path-prefix="CENTRAL."
    remote-path-selector="CENTRAL.SITE1">
</ns1:remote>
```

remote-system-name is the name of the remote system as exposed by the local remoting connector (local name).

remote-path-prefix indicates the prefix (parent path) in the master's processing model that must be added to remotely received updates and data, to construct their location in the master's processing model, and removed to outgoing activities to identify their correct path in the remote system. For instance: if the remote system reports an update for parameter SITE1.SYSTEM1.STATUS, such parameter's location is derived as concatenation of *remote-path-prefix* (CENTRAL.) and the remote location of the parameter, resulting in CENTRAL.SITE1.SYSTEM1.STATUS.

remote-path-selector indicates the system element in the master's processing model, which maps the root system element of the remote's processing model.

In order to work as expected, it is necessary to have a remoting configuration, which must be specified in the master system using the system variable *reatmetric.remoting.connector.config*. The content of the file pointed by such system variable is as follows.

```
<ns1:connectors xmlns:ns1=
"http://dariolucia.eu/reatmetric/remoting/connector/configuration">
```

```
    <connector local-name="Test System Station 1" remote-name="System 1" host=
"192.168.0.3" port="20000" />
    <connector local-name="Test System Station 2" remote-name="System 2" host=
"192.168.0.177" port="20000" />
</ns1:connectors>
```

For further details check the documentation related to the eu.dariolucia.reatmetric.remoting.connector module.

Socket

Overview

Configuration

Overview

The eu.dariolucia.reatmetric.driver.socket module provides a driver that allows the processing of socket-based protocols (TCP/P), in binary and ASCII format.

Being a driver, a *socket* module must be registered as such in the system Core's configuration.

As it must be able to support by configuration a large variety of ASCII and binary-based protocols, the driver configuration and design are overall complex. Nevertheless, by breaking down the configuration and the design smaller pieces, such complexity can be managed. The information provided in this document allows to understand the fundamentals of the design of the driver.

General Concepts

The *socket* driver is structure upon the following main concepts:

1. Messages: A message is a protocol data unit exchanged on one or more connections between the driver and the endpoint. Messages can be binary messages, defined using the eu.dariolucia.ccsds.encdec definitions, or ASCII, defined in the configuration file of the driver.
2. Connections: A connection is a communication channel, based on a network protocol, established between the driver and a remote host and port. For a given entity to monitor and control, the driver allows to specify several connections, each with its own characteristics: IP and port, initialisation of the connection, type of the protocol that is run over the connection (binary, ASCII), way to read the protocol data units from the connection. The driver supports TCP and UDP transport protocols.
3. Decodings: A decoding strategy is the way to read PDUs over a connection. For instance, in case of ASCII connections, the delimiter could be a specific character or sequence of characters. In case of binary data, there might be known prefixes, which identify the start of the PDU, or a structure with a length field, or the PDUs have fixed length.
4. Route configuration: A route configuration specifies which specific PDU is expected from a specific connection, how the information contained in the PDU must be mapped to the entities defined in the ReatMetric processing model, which events must be raised upon reception of

specific PDUs, how commands can be verified by reception of specific responses. Such protocol information are defined per connection.

When the driver is instantiated, all connections marked to be 'connector-driver' are grouped into a single ReatMetric *connector* and are globally controllable.

Configuration

Being a driver, the *socket* module must be registered as such in the system configuration file. You need to have a socket module registration for each endpoint that requires such driver.

```
<ns1:core xmlns:ns1="http://dariolucia.eu/reatmetric/core/configuration">
    <name>Test System</name>
    <log-property-file>$HOME\Reatmetric\reatmetric_test\log.properties</log-property-
file>
    <definitions-location>$HOME\Reatmetric\reatmetric_test\processing</definitions-
location>
    <driver name="Equipment 1 Driver" type=
"eu.dariolucia.reatmetric.driver.socket.SocketDriver"
    configuration="$HOME\Reatmetric\reatmetric_test\eqp1"/>
</ns1:core>
```

The folder specified in the *configuration* attribute of the *driver* element must contain a file named *configuration.xml*, which defines the configuration properties of the driver.

Main Configuration File

The configuration structure of the eu.dariolucia.reatmetric.driver.socket module is defined in the package eu.dariolucia.reatmetric.driver.socket.configuration. It is an XML file named *configuration.xml* using namespace definition *http://dariolucia.eu/reatmetric/driver/socket*.

In order to explain the different configuration parts in relation to ASCII and binary protocols, two examples are provided.

ASCII Example

An example of configuration file modelling an ASCII based protocol is presented below.

```
<ns1:socket xmlns:ns1="http://dariolucia.eu/reatmetric/driver/socket"
    name="Double ASCII"
    description="Driver configuration for double ASCII TCP connection">
    <connections>
        <!-- Connection 1: TCP connection to get TM data from the endpoint -->
        <tcp name="Device TM TCP Connection" source="Device 2" protocol="ASCII" init=
"CONNECTOR"
             host="127.0.0.1" remote-port="35212" local-port="0" ascii-encoding="UTF8
">
            <!-- Decoding: messages are all terminated with a new line character -->
            <asciiDelimiterDecoding><delimiter>\n</delimiter></asciiDelimiterDecoding>
```

```
            <!-- Route configuration: definition of the mappings for inbound messages
-->
            <route name="Device 2 TM Route" entity-offset="0" command-lock="true">
                <activity-types>
                    <!-- List of activity types (from the processing model) supported
by this driver/route -->
                    <type>DEV2-ASCII-CMD</type>
                </activity-types>
                <!-- Mapping definition for incoming PDU: from parameters of message
TLM_SUB1 to the parameter IDs in the processing model -->
                <inbound id="TLM_SUB1_MP" message="TLM_SUB1">
                    <inject name="status_val" entity="40" />
                    <inject name="freq_val" entity="41" />
                    <inject name="temp_val" entity="42" />
                    <inject name="offset_val" entity="43" />
                    <inject name="mode_val" entity="44" />
                    <inject name="sweep_val" entity="45" />
                </inbound>
                <!-- Mapping definition for incoming PDU: from parameters of message
TLM_SUB2 to the parameter IDs in the processing model -->
                <inbound id="TLM_SUB2_MP" message="TLM_SUB2">
                    <inject name="status_val" entity="50" />
                    <inject name="freq_val" entity="51" />
                    <inject name="temp_val" entity="52" />
                    <inject name="offset_val" entity="53" />
                    <inject name="mode_val" entity="54" />
                    <inject name="sweep_val" entity="55" />
                </inbound>
                <!-- Mapping definition: TM Subsystem Registration -->
                <outbound id="POLL_SUB_MP" message="POLL_SUB" type="CONNECTION_ACTIVE
">

                </outbound>
            </route>
        </tcp>
        <!-- Connection 2: TCP connection to send commands to the endpoint -->
        <tcp name="Device TC TCP Connection" source="Device 2" protocol="ASCII" init=
"ON_DEMAND"
             host="127.0.0.1" remote-port="35213" local-port="0" ascii-encoding="UTF8
">
            <asciiDelimiterDecoding><delimiter>\n</delimiter></asciiDelimiterDecoding>
            <route name="Device 2 TC Route" entity-offset="0" command-lock="true">
                <activity-types>
                    <type>DEV2-ASCII-CMD</type>
                </activity-types>
                <!-- Common -->
                <inbound id="ACK_MP" message="ACK" />
                <inbound id="EXE_MP" message="EXE" />
                <inbound id="NOK_MP" message="NOK" />
                <!-- Common -->
                <outbound id="SET_MP" message="SET" type="ACTIVITY_DRIVEN" entity="2">
                    <argument name="device_subsystem" field="device_subsystem" />
```

```
                    <argument name="parameter" field="parameter" />
                    <argument name="new_value" field="new_value" />
                    <auto-increment counter-id="SEQ1" field="command_id" output-type=
"UNSIGNED_INTEGER"/>
                    <verification timeout="10">
                        <acceptance message="ACK" value-field="command_id" reference-
argument="command_id" result="OK" />
                        <acceptance message="NOK" value-field="command_id" reference-
argument="command_id" result="FAIL" />
                        <execution message="EXE" value-field="command_id" reference-
argument="command_id" result="OK" />
                        <execution message="NOK" value-field="command_id" reference-
argument="command_id" result="FAIL" />
                    </verification>
                </outbound>
                <!-- First Subsystem -->
                <outbound id="CMD_SUB1_RST_MP" message="CMD_RST" type="
ACTIVITY_DRIVEN" entity="46">
                    <argument name="device_subsystem" field="device_subsystem" />
                    <auto-increment counter-id="SEQ1" field="command_id" output-type=
"UNSIGNED_INTEGER"/>
                    <verification timeout="10">
                        <acceptance message="ACK" value-field="command_id" reference-
argument="command_id" result="OK" />
                        <acceptance message="NOK" value-field="command_id" reference-
argument="command_id" result="FAIL" />
                        <execution message="EXE" value-field="command_id" reference-
argument="command_id" result="OK" />
                        <execution message="NOK" value-field="command_id" reference-
argument="command_id" result="FAIL" />
                    </verification>
                </outbound>
                <outbound id="CMD_SUB1_SWP_MP" message="CMD_SWP" type="
ACTIVITY_DRIVEN" entity="47">
                    <argument name="device_subsystem" field="device_subsystem" />
                    <argument name="times" field="times" />
                    <auto-increment counter-id="SEQ1" field="command_id" output-type=
"UNSIGNED_INTEGER"/>
                    <verification timeout="10">
                        <acceptance message="ACK" value-field="command_id" reference-
argument="command_id" result="OK" />
                        <acceptance message="NOK" value-field="command_id" reference-
argument="command_id" result="FAIL" />
                        <execution message="EXE" value-field="command_id" reference-
argument="command_id" result="OK" />
                        <execution message="NOK" value-field="command_id" reference-
argument="command_id" result="FAIL" />
                    </verification>
                </outbound>
                <outbound id="CMD_SUB1_RBT_MP" message="CMD_RBT" type="
ACTIVITY_DRIVEN" entity="48">
```

```
                    <argument name="device_subsystem" field="device_subsystem" />
                    <argument name="delay" field="delay" />
                    <argument name="running" field="running" />
                    <auto-increment counter-id="SEQ1" field="command_id" output-type=
"UNSIGNED_INTEGER"/>
                    <verification timeout="10">
                        <acceptance message="ACK" value-field="command_id" reference-
argument="command_id" result="OK" />
                        <acceptance message="NOK" value-field="command_id" reference-
argument="command_id" result="FAIL" />
                        <execution message="EXE" value-field="command_id" reference-
argument="command_id" result="OK" />
                        <execution message="NOK" value-field="command_id" reference-
argument="command_id" result="FAIL" />
                    </verification>
                </outbound>
                <!-- Second Subsystem -->
                <outbound id="CMD_SUB2_RST_MP" message="CMD_RST" type="
ACTIVITY_DRIVEN" entity="56">
                    <argument name="device_subsystem" field="device_subsystem" />
                    <auto-increment counter-id="SEQ1" field="command_id" output-type=
"UNSIGNED_INTEGER"/>
                    <verification timeout="10">
                        <acceptance message="ACK" value-field="command_id" reference-
argument="command_id" result="OK" />
                        <acceptance message="NOK" value-field="command_id" reference-
argument="command_id" result="FAIL" />
                        <execution message="EXE" value-field="command_id" reference-
argument="command_id" result="OK" />
                        <execution message="NOK" value-field="command_id" reference-
argument="command_id" result="FAIL" />
                    </verification>
                </outbound>
                <outbound id="CMD_SUB2_SWP_MP" message="CMD_SWP" type="
ACTIVITY_DRIVEN" entity="57">
                    <argument name="device_subsystem" field="device_subsystem" />
                    <argument name="times" field="times" />
                    <auto-increment counter-id="SEQ1" field="command_id" output-type=
"UNSIGNED_INTEGER"/>
                    <verification timeout="10">
                        <acceptance message="ACK" value-field="command_id" reference-
argument="command_id" result="OK" />
                        <acceptance message="NOK" value-field="command_id" reference-
argument="command_id" result="FAIL" />
                        <execution message="EXE" value-field="command_id" reference-
argument="command_id" result="OK" />
                        <execution message="NOK" value-field="command_id" reference-
argument="command_id" result="FAIL" />
                    </verification>
                </outbound>
                <outbound id="CMD_SUB2_RBT_MP" message="CMD_RBT" type="
```

```
ACTIVITY_DRIVEN" entity="58">
                    <argument name="device_subsystem" field="device_subsystem" />
                    <argument name="delay" field="delay" />
                    <argument name="running" field="running" />
                    <auto-increment counter-id="SEQ1" field="command_id" output-type=
"UNSIGNED_INTEGER"/>
                    <verification timeout="10">
                        <acceptance message="ACK" value-field="command_id" reference-
argument="command_id" result="OK" />
                        <acceptance message="NOK" value-field="command_id" reference-
argument="command_id" result="FAIL" />
                        <execution message="EXE" value-field="command_id" reference-
argument="command_id" result="OK" />
                        <execution message="NOK" value-field="command_id" reference-
argument="command_id" result="FAIL" />
                    </verification>
                </outbound>
            </route>
        </tcp>
    </connections>
    <messages>
        <ascii id="TLM_SUB1">
            <template>TLM SUB1 ${{status_val}}$ ${{freq_val}}$ ${{temp_val}}$
${{offset_val}}$ ${{mode_val}}$ ${{sweep_val}}$\n</template>
            <symbol name="status_val" type="ENUMERATED" encode-null="" decode-empty-
null="true" />
            <symbol name="freq_val" type="UNSIGNED_INTEGER" encode-null="" decode-
empty-null="true" />
            <symbol name="temp_val" type="REAL" encode-null="" decode-empty-null="
true" />
            <symbol name="offset_val" type="SIGNED_INTEGER" encode-null="" decode-
empty-null="true" />
            <symbol name="mode_val" type="ENUMERATED" encode-null="" decode-empty-
null="true" />
            <symbol name="sweep_val" type="ENUMERATED" encode-null="" decode-empty-
null="true" />
        </ascii>
        <ascii id="TLM_SUB2">
            <template>TLM SUB2 ${{status_val}}$ ${{freq_val}}$ ${{temp_val}}$
${{offset_val}}$ ${{mode_val}}$ ${{sweep_val}}$\n</template>
            <symbol name="status_val" type="ENUMERATED" encode-null="" decode-empty-
null="true" />
            <symbol name="freq_val" type="UNSIGNED_INTEGER" encode-null="" decode-
empty-null="true" />
            <symbol name="temp_val" type="REAL" encode-null="" decode-empty-null="
true" />
            <symbol name="offset_val" type="SIGNED_INTEGER" encode-null="" decode-
empty-null="true" />
            <symbol name="mode_val" type="ENUMERATED" encode-null="" decode-empty-
null="true" />
            <symbol name="sweep_val" type="ENUMERATED" encode-null="" decode-empty-
```

```
null="true" />
        </ascii>
        <ascii id="POLL_SUB">
            <template>REQ SUB1 SUB2\n</template>
        </ascii>

        <ascii id="CMD_SWP">
            <template>CMD ${{device_subsystem}}$ ${{command_id}}$ SWP
${{times}}$\n</template>
            <symbol name="device_subsystem" type="CHARACTER_STRING" decode-empty-null
="false" />
            <symbol name="command_id" type="UNSIGNED_INTEGER" decode-empty-null="
false" />
            <symbol name="times" type="UNSIGNED_INTEGER" />
        </ascii>
        <ascii id="CMD_RST">
            <template>CMD ${{device_subsystem}}$ ${{command_id}}$ RST\n</template>
            <symbol name="device_subsystem" type="CHARACTER_STRING" decode-empty-null
="false" />
            <symbol name="command_id" type="UNSIGNED_INTEGER" decode-empty-null="
false" />
        </ascii>
        <ascii id="CMD_RBT">
            <template>CMD ${{device_subsystem}}$ ${{command_id}}$ RBT ${{delay}}$
${{running}}$\n</template>
            <symbol name="device_subsystem" type="CHARACTER_STRING" decode-empty-null
="false" />
            <symbol name="command_id" type="UNSIGNED_INTEGER" decode-empty-null="
false" />
            <symbol name="delay" type="UNSIGNED_INTEGER" />
            <symbol name="running" type="UNSIGNED_INTEGER" />
        </ascii>
        <ascii id="SET">
            <template>SET ${{device_subsystem}}$ ${{command_id}}$ ${{parameter}}$
${{new_value}}$\n</template>
            <symbol name="device_subsystem" type="CHARACTER_STRING" decode-empty-null
="false" />
            <symbol name="command_id" type="UNSIGNED_INTEGER" decode-empty-null="
false" />
            <symbol name="parameter" type="CHARACTER_STRING" />
            <symbol name="new_value" type="DERIVED">
                <type id="ENUMERATED" encode-format="%d" />
                <type id="UNSIGNED_INTEGER" encode-format="%d" />
                <type id="SIGNED_INTEGER" encode-format="%d" />
                <type id="REAL" encode-format="%.1f" />
            </symbol>
        </ascii>

        <ascii id="ACK">
            <template>ACK ${{device_subsystem}}$ ${{command_id}}$\n</template>
            <symbol name="device_subsystem" type="CHARACTER_STRING" decode-empty-null
```

```
="false" />
            <symbol name="command_id" type="UNSIGNED_INTEGER" decode-empty-null="
false" />
        </ascii>
        <ascii id="EXE">
            <template>EXE ${{device_subsystem}}$ ${{command_id}}$\n</template>
            <symbol name="device_subsystem" type="CHARACTER_STRING" decode-empty-null
="false" />
            <symbol name="command_id" type="UNSIGNED_INTEGER" decode-empty-null="
false" />
        </ascii>
        <ascii id="NOK">
            <template>NOK ${{command_id}}$\n</template>
            <symbol name="command_id" type="UNSIGNED_INTEGER" decode-empty-null="
false" />
        </ascii>
    </messages>
</ns1:socket>
```

Attribute <**name**> (mandatory, string): this element assigns the name of the driver instance, it does not play any role in the processing of the data.

Attribute <**description**> (mandatory, string): this element assigns a description of the driver instance.

Even if the configuration file first presents the connection definitions, with the associated decoding and route configuration, for a better understanding the element <**messages**> is introduced first. The sub-elements of a <**messages**> element are:

- <**ascii**> elements: it defines a single ASCII message, with associated template and symbol definitions;
- <**binary**> elements: it defines a collection of binary messages, with associated definition file and type of structure to consider within the definition file;

All <**ascii**> elements have the following structure:

- Attribute *id* (mandatory, string): the identifier of the message, which is used in the route configuration.
- Sub-element <**template**> (mandatory, string): the message template, including non printable characters (\n, \t, ...). The template is a sequence of characters and *symbols*. A symbol, to be recognised as such, must be enclosed between the *${{* and *}}$* escape fields.
- Sub-element <**symbol**> (optional, zero or more): it contains the definition of the symbol in terms of type and encoding/decoding characteristics. It defines the following attributes and sub-elements:
    - Attribute *name* (mandatory, string): it must match the name of the symbol as it appears in the template.
    - Attribute *type* (mandatory, enum from eu.dariolucia.reatmetric.api.value.ValueTypeEnum): the type of the field, DERIVED can also be used for messages representing commands.

- Attribute *radix* (optional, enum: BIN, OCT, DEC, HEX, default: DEC): the radix to be used for decoding and encoding ENUMERATION and INTEGER symbols.
- Attribute *encode-format* (optional, string, default: null): the encode format to be used for non-DERIVED types. The format syntax is the one defined by the java.util.Formatter class (e.g. %04X to print an integer as 4 digits hex number with pre-padded 0).
- Attribute *encode-null* (optional, string, default: null): the value to be used for the symbol when encoding a null value.
- Attribute *decode-empty-null* (optional, boolean, default: false): if true, then the Java null value is returned as decoded value, if the symbol is encoded as an empty string. If false, an empty string is returned.
- Sub-element <**type**> (optional, zero or more): this sequence of elements is used to specify how to encode a specific type in the case of DERIVED fields. It defines the following two attributes:
  - Attribute *id* (mandatory, enum from eu.dariolucia.reatmetric.api.value.ValueTypeEnum): the type of the field according to the provided value type (mapping defined as per ValueTypeEnum specification)
  - Attribute *encode-format* (mandatory, string): the encode format to be used for the specified type. The format syntax is the one defined by the java.util.Formatter class (e.g. %04X to print an integer as 4 digits hex number with pre-padded 0).

The example file defines 2 connections of type TCP, therefore the element <**connections**> contains 2 elements <**tcp**>.

Element <**tcp**>: this element defines a TCP/IP connection with the provided characteristics. Element <**udp**>: this element defines a UDP/IP connection with the provided characteristics.

The following attributes are defined for the two connection types:

- *name* (mandatory, string): the name of the connection.
- *source* (mandatory, string): the name of the endpoint, used as source when distributing the received messages as raw data to the Raw Data broker for distribution inside ReatMetric.
- *protocol* (mandatory, enum: ASCII, BINARY): the type of the protocol exchanged on top of this connection.
- *init* (optional, enum: CONNECTOR, ON_DEMAND, default: CONNECTOR): the way the establishment of the connection must be performed. If CONNECTOR is set, then the connection and disconnection is fully driven by the driver's *connector*. If ON_DEMAND is set, then the connection is established if the driver's *connector* is active and an outgoing message is requested to be sent on the connection. The connection remains open until the lifetime of the outgoing message is not expired.
- *host* (mandatory, string): the hostname or IP address of the endpoint.
- *remote-port* (mandatory, integer): the remote port, the connection shall associate to.
- *local-port* (mandatory, integer): the local port, the connection shall be bound to. Setting this value to 0 leaves to the operating system the decision of what to do.
- *ascii-encoding* (optional, enum: US_ASCII, ISO_8859_1, UTF_8, UTF_16, default: US_ASCII): the

ASCII encoding to be used when interpreting bytes from the connection and encoding messages.

- *timeout* (optional, integer, default: 5000): socket timeout in milliseconds, affecting socket reading and connection timeout.
- *tx-buffer* (optional, integer, default: 0): transmit buffer in bytes, 0 means use operating system default.
- *rx-buffer* (optional, integer, default: 0): reception buffer in bytes, 0 means use operating system default.

For <**tcp**> connections only, the following two attributes are additionally defined:

- *tcp-keep-alive* (optional, boolean, default: false): if true, activate TCP keep-alive mechanism.
- *tcp-no-delay* (optional, boolean, default: false): if true, activate TCP no-delay mechanism.

As decoding strategy, the following decoders are available as sub-elements of the <**tcp**> and <**udp**> elements:

- <**datagramDecoding**>: this element can only be used linked to <**udp**> connection types, to indicate that a received datagram must be interpreted as a PDU. No sub-elements/attributes are supported.
- <**fixedLengthDecoding**>: this decoder reads a fixed number of bytes from the underlying channel and interpret such data as a PDU. The following attribute is defined:
  - Attribute *length* (mandatory, integer): length of the PDU.
- <**lengthFieldDecoding**>: this decoder is configurable to read and interpret a specific field as length of the message, possibly shifting, masking and adding a value. The decoder can be configured to consider or not the header length or the size of the length field itself. The following attributes are defined:
  - Attribute *header-nb-bytes-to-skip* (optional, integer, default: 0): number of bytes to skip before the length field.
  - Attribute *field-length* (optional, integer, default: 0): size in bytes of the length field. Maximum value is 8.
  - Attribute *big-endian* (optional, boolean, default: true): length field endianness.
  - Attribute *field-mask* (optional, integer, default: -1): AND mask to be applied to the length field value.
  - Attribute *field-right-shift* (optional, integer, default: 0): number of bits to shift right the length field value, after application of the AND mask.
  - Attribute *field-value-offset* (optional, integer, default: 0): value to be added to the length field value, after AND mask and right shift.
  - Attribute *consider-skipped-bytes* (optional, boolean, default: false): if true, then the length field value counts also the skipped bytes (before the length field). This information is fundamental to ensure the complete and correct reading of the PDU from the connection.
  - Attribute *consider-field-length* (optional, boolean, default: false): if true, then the length field value counts also the size of the field length. This information is fundamental to ensure the complete and correct reading of the PDU from the connection.

- <**binaryDelimiterDecoding**>: this decoder reads all characters in the underlying channel and stops when the specified delimiter is found. The following attributes are defined:
  - Attribute *start-sequence* (mandatory, string): hex dump of the delimiter identifying the start of the PDU.
  - Attribute *end-sequence* (mandatory, string): hex dump of the delimiter identifying the end of the PDU.
- <**asciiDelimiterDecoding**>: This decoder reads all characters in the underlying channel and stops when the specified delimiter, as sequence of ASCII characters, is found. For instance, for ASCII protocols having a new line as message delimiter (as in the example file), the string "\n" shall be used. The following sub-element is defined.
  - Element <**delimiter**> (mandatory, string): string that identifies the end of message.

Each connection shall contain a <**route**> element, containing the mapping and protocol configuration. The purpose of the route configuration section is to:

- define inbound and outbound PDUs allowed for the connection;
- define a mapping between the fields contained in each inbound PDU to the corresponding parameters in the processing model;
- define a mapping between inbound PDUs and events to be raised in the processing model;
- define a mapping between inbound PDUs and verification stages linked to outbound PDUs;
- define the characteristics of the interlocking for outbound PDUs.

The <**route**> element has the following structure:

- Attribute *name* (mandatory, string): name of the route configuration.
- Attribute *entity-offset* (optional, integer, default: 0): value to be added to the defined mapping to derive the actual parameter/event ID in the processing model, and to be subtraced to the invoked activity ID, to identify the outbound message to encode and send as command.
- Attribute *command-lock* (optional, boolean, default: true): if true, the connection will not allow more than a single command to be sent and alive on the connection. The lifecycle of the command must complete, before the driver sends the following command. This type of configuration is typically used for communication patterns of type request/response or request/progress/response, where there is no way to identify the command in the protocol, for further verification with responses.
- Element <**activity-types**> (mandatory, one) with sub-elements <**type**> (string): this list identifies the types of the activity, as specified in the processing model, so that the driver can inform the processing model about its capability to encode and process such activities.
- List of <**inbound**> elements.
- List of <**outbound**> elements.

The <**inbound**> element specifies the processing for incoming PDUs, i.e. PDUs received from the endpoint through the connection, in terms of mapping of PDU's contained fields to parameters and mapping of PDU's reception to events. The element has the following structure:

- Attribute *id* (mandatory, string): the ID of the inbound message configuration, used for logging purposes.
- Attribute *message* (mandatory, string - IDREF): the ID of the mapped message as defined in the <**messages**> section.
- Attribute *secondary-id* (optional, string, default: empty string): in case of binary messages, this attribute must contain the ID of the binary structure defined in the structure definition file.
- Element <**command-match**> (optional): this element is a way to specify that the mapping is linked to an outbound message mapping, with optionally a specific argument matching a value expressed here. When such message is sent and a message is received, this mapping is effectively used only if it is linked to that command. This feature covers the situation when a request is asking for a given response, the request contains an ID of a device to query, but the response does not have such ID. The correlation to the ID, and therefore the mapping to the correct parameters to be injected in the processing model, must be derived by the fact that a previous command was sent. This approach of course works only for full synchronous request-response protocols. It specifies the following attributes:
  - *outbound-mapping* (mandatory, string - IDREF): the ID of the outbound message configuration.
  - *argument* (optional, string, default: null): the field in the outbound message, previously sent, to be checked to confirm the mapping. If null, the mapping is confirmed by default.
  - *value* (optional, string, default: null): the value that must be matched by the outbound message field indicated by the *argument* attribute. It is a mandatory if the *argument* attribute is set.
  - *type* (optional, enum from eu.dariolucia.reatmetric.api.value.ValueTypeEnum, default: null): the type of the value string specified in the *value* attribute. It is a mandatory if the *value* attribute is set.
  - *last-command* (optional, boolean, default: true): if true, the driver will use the actual very last PDU that was sent on this connection, regardless of the PDU linked to the associated outbound mapping, to verify if the inbound mapping must be selected. If false, the driver will use the last command sent with the specified outbound mapping.
- List of <**computed-field**> elements (optional): a computed field is an artificial field, computed from other fields (PDU value fields or other computed fields). It has the following structure:
  - Attribute *field*: name of the computed field.
  - Element value: Groovy expression to compute the value of the computed field. The expression can access all fields of the inbound message and also all the previously computed fields. Computed fields are calculated in the order they appear in the configuration. When binding the PDU fields and computed fields to the Groovy expression, characters ' ' (space) and '-' (minus) are replaced with '_' (underscore). This means e.g. that the value of the PDU field "device-id" will be mapped to a Groovy variable named "device_id".
- List of <**inject**> elements (optional): each element maps a field by name to the corresponding parameter in the processing model. It has the following structure:
  - Attribute *name* (mandatory, string): the name of the field to map.
  - Attribute *entity* (mandatory, integer): the ID of the parameter, as defined in the processing

model. The actual value of the parameter ID is computed by adding to this value the value of the *entity-offset* attribute defined in the parent <**route**> element.

- List of <**raise**> elements (optional): each element requests the raising of the corresponding event in the processing model. It has the following structure:
  - Attribute *entity* (mandatory, integer): the ID of the event, as defined in the processing model. The actual value of the event ID is computed by adding to this value the value of the *entity-offset* attribute defined in the parent <**route**> element.
  - Attribute *source* (optional, string, default: empty string): the source value to be associated to the request to raise the event.
  - Attribute *qualifier* (optional, string, default: null): the qualifier value to be associated to the request to raise the event. If specified, this value takes precedence over the *qualifier-reference*.
  - Attribute *qualifier-reference* (optional, string, default: null): the name of the field (PDU field or computed field) to be used as qualifier value. If the field is not defined, an empty string is used as qualifier.
  - Element <**condition**> (optional): if defined, the event is raised only if the equality condition specified by this element is satisfied. This element has the following structure:
  - Attribute *field* (mandatory, string): name of the field (PDU field or computed field) that must be checked.
  - Attribute *value* (mandatory, string): the value that must match the value of the specified field, to verify the condition.
  - Attribute *type* (mandatory, enum from eu.dariolucia.reatmetric.api.value.ValueTypeEnum): the type of the value string specified in the *value* attribute.

The <**outbound**> element specifies the processing for the encoding and sending of outgoing PDUs, i.e. PDUs assembled by the ReatMetric system and delivered to the endpoint through the connection. In terms of mapping to PDU fields, the outbound message element specifies how fields are computed from activity arguments, fixed values, auto incremental fields and computed fields. At the end of the processing, each field is mapped to the corresponding PDU field by name, and the resulting message is encoded. The element has the following structure:

- Attribute *id* (mandatory, string): the ID of the outbound message configuration, used for logging and mapping purposes in the *outbound-mapping* attribute of the <**command-match**> element of <**inbound**> elements.
- Attribute *message* (mandatory, string - IDREF): the ID of the mapped message as defined in the <**messages**> section.
- Attribute *secondary-id* (optional, string, default: empty string): in case of binary messages, this attribute must contain the ID of the binary structure defined in the structure definition file.
- Attribute *type* (mandatory, enum: ACTIVITY_DRIVEN, CONNECTION_ACTIVE, PERIODIC): the type of outbound message:
  - If ACTIVITY_DRIVEN: the PDU is encoded and dispatched after reception of an activity request, whose activity ID is mapped to this outbound mapping via the *entity* attribute value plus the offset defined in the *entity-offset* attribute defined in the parent <**route**> element.

    - If CONNECTION_ACTIVE: the PDU is encoded and dispatched every time the connection is established. The typical use of this feature is to send automated subscription requests upon connection establishment.
    - If PERIODIC: the PDU is encoded and dispatched periodically, as long as the connection is established. This value works only for connections configured to be connector-driven, i.e. with attribute *init* on the <**tcp**> or <**udp**> element to be set to CONNECTOR.
- Attribute *entity* (optional, integer, default: -1): the ID of the activity in the processing model, which shall trigger the encoding and dispatch of the associated message, if received by the driver. The actual value used to map the activity ID is computed by adding to this value the value of the *entity-offset* attribute defined in the parent <**route**> element. Only taken into account if the *type* of the outbound message is ACTIVITY_DRIVEN.
- Attribute *period* (optional, integer, default: 0): the period in seconds, for outbound messages with *type* PERIODIC.
- Attribute *max-waiting-time* (optional, integer, default: 2000): the maximum waiting time in milliseconds, that the driver can wait, in order to have a free connection to send the message in case the *command-lock* is used and active.
- Attribute *post-send-delay* (optional, integer, default: 0): the time in milliseconds that the driver waits, after sending this outbound message, before processing the other outbound messages.
- List of <**fixed-field**> elements (optional): a fixed field is an artificial field, defined in the outbound message configuration, which can be used for encoding purpose. The element contains the following attributes:
    - Attribute *field* (mandatory, string): name of the fixed field.
    - Attribute *value* (mandatory, string): the value that is assigned to the field.
    - Attribute *type* (mandatory, enum from eu.dariolucia.reatmetric.api.value.ValueTypeEnum): the type of the value string specified in the *value* attribute.
- List of <**argument**> elements (optional): the element specifies how activity argument fields are assigned to fields. The element contains the following attributes:
    - Attribute *field* (mandatory, string): name of the field.
    - Attribute *name* (mandatory, string): name of the activity argument.
- List of <**auto-increment**> elements (optional): this element assigns an incremental value to the specified field, every time a new outbound message is encoded. The element contains the following attributes:
    - Attribute *field* (mandatory, string): name of the field.
    - Attribute *counter-id* (mandatory, string): name of the counter. If different counters are needed, depending on the outbound messages, then different values must be specified here. If a single global counter is needed, then the same value on all outbound messages must be defined. Counters are route-specific and have signed integer resolution.
    - Attribute *output-type* (optional, enum from eu.dariolucia.reatmetric.api.value.ValueTypeEnum, default: ENUMERATED): output value of the counter, limited to: SIGNED_INTEGER, UNSIGNED_INTEGER, REAL, ENUMERATED, CHARACTER_STRING.

    - Attribute *string-format* (optional, string, default: null): the encode format to be used in case *output-type* is set to CHARACTER_STRING. The format syntax is the one defined by the java.util.Formatter class (e.g. %04X to print an integer as 4 digits hex number with pre-padded 0).
- List of <**computed-field**> elements (optional): a computed field is an artificial field, computed from other fields (argument fields, fixed value fields, auto-increment fields or other computed fields). The structure is the same one defined for <**inbound**> elements.
- Element <**verification**> (optional, default: null): the verification configuration for the outbound message. The element has the following structure:
    - Attribute *id-field* (optional, string, default: null): the name of the field of the outbound message, containing its identifier value. If null, this information is not used.
    - Attribute *timeout* (optional, integer, default: 5): overall timeout in seconds for the command to be fully verified, since its successful release (activity occurrence transitioned to TRANSMISSION).
    - List of <**acceptance**> elements (optional): a list of elements detailing the checks for confirming the acceptance of the outbound message by the endpoint. The checks are executed in order of definition.
    - List of <**execution**> elements (optional): a list of elements detailing the checks for confirming the execution of the outbound message by the endpoint. The checks are executed in order of definition.

Both the <**acceptance**> and <**execution**> elements have the following structure:

- Attribute *message* (mandatory, string - IDREF): the ID of the message that contains the verification information.
- Attribute *secondary-id* (optional, string, default: null): the secondary ID of the message that contains the verification information. Shall be null if not used, i.e. for ASCII messages.
- Attribute *id-field* (optional, string, default: null): the name of the field of the inbound message, containing the identifier value of the corresponding outbound message (for mapping purposes). If null, it is not used.
- Attribute *value-field* (optional, string, default: null): the name of the field of the inbound message, to be checked to understand the result of the operation. If null, it is not used.
- Attribute *expected-fixed-value* (optional, string, default: null): the value that it is compared with the one defined in the *value-field* attribute.
- Attribute *expected-fixed-value-type* (optional, enum from eu.dariolucia.reatmetric.api.value.ValueTypeEnum, default: CHARACTER_STRING): the type of the value defined in the *expected-fixed-value* attribute.
- Attribute *reference-argument* (optional, string, default: null): the name of the field of the outbound message, that it is compared with the one defined in the *value-field* attribute of the inbound message.
- Attribute *result* (mandatory, enum from eu.dariolucia.reatmetric.api.activity.ActivityReportState): the result to be reported for the specific stage (acceptance, execution) if all defined criteria are matched.

Therefore, the verification approach for a released outbound command pending verification works as follows:

1. An inbound message is received.
2. If there is a registered outbound message declaring the message and secondary ID of the inbound message in one of the verification stages, then the inbound message is further processed for its verification.
3. If an id-field is defined and its value in the inbound message matches the value of the outbound message under verification, then the inbound message is further processed for the verification of the specific outbound message. The same happens if no id-field is specified.
4. If a value-field is defined, then its value is checked against either the fixed value or against the specified outbound message field. If the values are the same, then the inbound message is further processed for the verification of the specific outbound message. The same happens if the value-field is not specified.
5. The result as reported in the *result* attribute is delivered to the processing model in case of activity mapping.

Once the acceptance stage is verified, the defined **<acceptance>** elements are not processed anymore. The same happens for the execution stage. It is possible for an outbound message to have both acceptance and execution elements, only acceptance, only execution or no verification element.

Main Configuration File - Binary

An example of configuration file modelling a binary protocol is presented below.

```
<ns1:socket xmlns:ns1="http://dariolucia.eu/reatmetric/driver/socket"
    name="Single Binary"
    description="Driver configuration for single Binary TCP connection">
    <connections>
        <tcp name="Device TC TCP Connection" source="Device 4" protocol="BINARY" init
="CONNECTOR"
             host="127.0.0.1" remote-port="37280" local-port="0" >
            <lengthFieldDecoding header-nb-bytes-to-skip="4" field-length="4" big-
endian="true"
                                 consider-skipped-bytes="true" consider-field-length=
"true" />
            <route name="Device 4 Route" entity-offset="0" command-lock="false">
                <activity-types>
                    <type>DEV4-BIN-CMD</type>
                </activity-types>
                <!-- Common -->
                <inbound id="ACK_POS_IN" message="CMD_DEFINITION" secondary-id=
"ACK_POS" />
                <inbound id="ACK_NEG_IN" message="CMD_DEFINITION" secondary-id=
"ACK_NEG" />
                <inbound id="EXE_POS_IN" message="CMD_DEFINITION" secondary-id=
"EXE_POS" />
```

```
                <inbound id="EXE_NEG_IN" message="CMD_DEFINITION" secondary-id=
"EXE_NEG" />
                <inbound id="NOK_IN" message="CMD_DEFINITION" secondary-id="CMD_NOK"
/>

                <!-- TM First Subsystem -->
                <inbound id="TLM_SUB1_MP" message="CMD_DEFINITION" secondary-id=
"TLM_SUB1">
                    <inject name="status_val" entity="170" />
                    <inject name="freq_val" entity="171" />
                    <inject name="temp_val" entity="172" />
                    <inject name="offset_val" entity="173" />
                    <inject name="mode_val" entity="174" />
                    <inject name="summary_val" entity="175" />
                    <inject name="sweep_val" entity="176" />
                </inbound>
                <!-- TM Second Subsystem -->
                <inbound id="TLM_SUB2_MP" message="CMD_DEFINITION" secondary-id=
"TLM_SUB2">
                    <inject name="status_val" entity="180" />
                    <inject name="freq_val" entity="181" />
                    <inject name="temp_val" entity="182" />
                    <inject name="offset_val" entity="183" />
                    <inject name="mode_val" entity="184" />
                    <inject name="summary_val" entity="185" />
                    <inject name="sweep_val" entity="186" />
                </inbound>

                <!-- EVT First Subsystem -->
                <inbound id="EVT_SUB1_MP" message="CMD_DEFINITION" secondary-id=
"EVT_SUB1">
                    <raise entity="200" source="Device 4" qualifier-reference=
"message_val">
                        <condition field="alarm_val" value="0" type="ENUMERATED" />
                    </raise>
                    <raise entity="201" source="Device 4" qualifier-reference=
"message_val">
                        <condition field="alarm_val" value="1" type="ENUMERATED" />
                    </raise>
                    <raise entity="202" source="Device 4" qualifier-reference=
"message_val">
                        <condition field="alarm_val" value="2" type="ENUMERATED" />
                    </raise>
                    <raise entity="203" source="Device 4" qualifier-reference=
"message_val">
                        <condition field="alarm_val" value="3" type="ENUMERATED" />
                    </raise>
                </inbound>
                <!-- EVT Second Subsystem -->
                <inbound id="EVT_SUB2_MP" message="CMD_DEFINITION" secondary-id=
"EVT_SUB2">
```

```
                    <raise entity="300" source="Device 4" qualifier-reference=
"message_val" />
                </inbound>

                <!-- TM Subsystem Registration -->
                <outbound id="POLL_SUB1_MP" message="CMD_DEFINITION" secondary-id=
"POLL_SUB" type="CONNECTION_ACTIVE">
                    <fixed-field field="device_subsystem" value="1" type=
"UNSIGNED_INTEGER" />
                    <auto-increment counter-id="SEQ1" field="request" output-type=
"UNSIGNED_INTEGER"/>
                    <verification timeout="6" id-field="request">
                        <execution message="CMD_DEFINITION" secondary-id="TLM_SUB1"
id-field="request" result="OK" />
                        <execution message="CMD_DEFINITION" secondary-id="
TLM_SUB1_NOK" id-field="request" result="FAIL" />
                        <execution message="CMD_DEFINITION" secondary-id="CMD_NOK"
result="FAIL" />
                    </verification>
                </outbound>
                <outbound id="POLL_SUB2_MP" message="CMD_DEFINITION" secondary-id=
"POLL_SUB" type="CONNECTION_ACTIVE">
                    <fixed-field field="device_subsystem" value="2" type=
"UNSIGNED_INTEGER" />
                    <auto-increment counter-id="SEQ1" field="request" output-type=
"UNSIGNED_INTEGER"/>
                    <verification timeout="6" id-field="request">
                        <execution message="CMD_DEFINITION" secondary-id="TLM_SUB2"
id-field="request" result="OK" />
                        <execution message="CMD_DEFINITION" secondary-id="
TLM_SUB2_NOK" id-field="request" result="FAIL" />
                        <execution message="CMD_DEFINITION" secondary-id="CMD_NOK"
result="FAIL" />
                    </verification>
                </outbound>

                <!-- EVT Subsystem Registration -->
                <outbound id="EVT_SUB1_MP" message="CMD_DEFINITION" secondary-id=
"EVT_SUB" type="CONNECTION_ACTIVE">
                    <fixed-field field="device_subsystem" value="1" type=
"UNSIGNED_INTEGER" />
                    <auto-increment counter-id="SEQ1" field="request" output-type=
"UNSIGNED_INTEGER"/>
                    <verification timeout="6" id-field="request" >
                        <execution message="CMD_DEFINITION" secondary-id="EVT_SUB1"
id-field="request" result="OK" />
                        <execution message="CMD_DEFINITION" secondary-id="
EVT_SUB1_NOK" id-field="request" result="FAIL" />
                        <execution message="CMD_DEFINITION" secondary-id="CMD_NOK"
result="FAIL" />
                    </verification>
```

```
                </outbound>
                <outbound id="EVT_SUB2_MP" message="CMD_DEFINITION" secondary-id=
"EVT_SUB" type="CONNECTION_ACTIVE">
                    <fixed-field field="device_subsystem" value="2" type=
"UNSIGNED_INTEGER" />
                    <auto-increment counter-id="SEQ1" field="request" output-type=
"UNSIGNED_INTEGER"/>
                    <verification timeout="6" id-field="request">
                        <execution message="CMD_DEFINITION" secondary-id="EVT_SUB2"
id-field="request" result="OK" />
                        <execution message="CMD_DEFINITION" secondary-id="
EVT_SUB2_NOK" id-field="request" result="FAIL" />
                        <execution message="CMD_DEFINITION" secondary-id="CMD_NOK"
result="FAIL" />
                    </verification>
                </outbound>

                <!-- Common -->
                <outbound id="SET_MP_LONG" message="CMD_DEFINITION" secondary-id=
"CMD_SET_LONG" type="ACTIVITY_DRIVEN" entity="103">
                    <!-- Not needed, can go in the definition, used for testing-->
                    <fixed-field field="length" value="36" type="UNSIGNED_INTEGER" />
                    <argument name="device_subsystem" field="device_subsystem" />
                    <argument name="parameter" field="parameter" />
                    <argument name="new_value" field="new_value" />
                    <auto-increment counter-id="SEQ1" field="request" output-type=
"UNSIGNED_INTEGER"/>
                    <verification timeout="10" id-field="request">
                        <acceptance message="CMD_DEFINITION" secondary-id="ACK_POS"
id-field="request" result="OK" />
                        <acceptance message="CMD_DEFINITION" secondary-id="ACK_NEG"
id-field="request" result="FAIL" />
                        <execution message="CMD_DEFINITION" secondary-id="EXE_POS" id-
field="request" result="OK" />
                        <execution message="CMD_DEFINITION" secondary-id="EXE_NEG" id-
field="request" result="FAIL" />
                        <execution message="CMD_DEFINITION" secondary-id="CMD_NOK"
result="FAIL" />
                    </verification>
                </outbound>
                <outbound id="SET_MP_INT" message="CMD_DEFINITION" secondary-id=
"CMD_SET_INT" type="ACTIVITY_DRIVEN" entity="104">
                    <fixed-field field="length" value="32" type="UNSIGNED_INTEGER" />
                    <argument name="device_subsystem" field="device_subsystem" />
                    <argument name="parameter" field="parameter" />
                    <argument name="new_value" field="new_value" />
                    <auto-increment counter-id="SEQ1" field="request" output-type=
"UNSIGNED_INTEGER"/>
                    <verification timeout="10" id-field="request">
                        <acceptance message="CMD_DEFINITION" secondary-id="ACK_POS"
id-field="request" result="OK" />
```

```
                        <acceptance message="CMD_DEFINITION" secondary-id="ACK_NEG"
id-field="request" result="FAIL" />
                        <execution message="CMD_DEFINITION" secondary-id="EXE_POS" id-
field="request" result="OK" />
                        <execution message="CMD_DEFINITION" secondary-id="EXE_NEG" id-
field="request" result="FAIL" />
                        <execution message="CMD_DEFINITION" secondary-id="CMD_NOK"
result="FAIL" />
                    </verification>
                </outbound>
                <outbound id="SET_MP_DOUBLE" message="CMD_DEFINITION" secondary-id=
"CMD_SET_DOUBLE" type="ACTIVITY_DRIVEN" entity="105">
                    <fixed-field field="length" value="36" type="UNSIGNED_INTEGER" />
                    <argument name="device_subsystem" field="device_subsystem" />
                    <argument name="parameter" field="parameter" />
                    <argument name="new_value" field="new_value" />
                    <auto-increment counter-id="SEQ1" field="request" output-type=
"UNSIGNED_INTEGER"/>
                    <verification timeout="10" id-field="request">
                        <acceptance message="CMD_DEFINITION" secondary-id="ACK_POS"
id-field="request" result="OK" />
                        <acceptance message="CMD_DEFINITION" secondary-id="ACK_NEG"
id-field="request" result="FAIL" />
                        <execution message="CMD_DEFINITION" secondary-id="EXE_POS" id-
field="request" result="OK" />
                        <execution message="CMD_DEFINITION" secondary-id="EXE_NEG" id-
field="request" result="FAIL" />
                        <execution message="CMD_DEFINITION" secondary-id="CMD_NOK"
result="FAIL" />
                    </verification>
                </outbound>
                <outbound id="SET_MP_PSTRING" message="CMD_DEFINITION" secondary-id=
"CMD_SET_PSTRING" type="ACTIVITY_DRIVEN" entity="106">
                    <argument name="device_subsystem" field="device_subsystem" />
                    <argument name="parameter" field="parameter" />
                    <argument name="new_value" field="new_value" />
                    <auto-increment counter-id="SEQ1" field="request" output-type=
"UNSIGNED_INTEGER"/>
                    <!-- Declaration order matters -->
                    <computed-field field="length">
                        return 28 + 4 + ((String) new_value).length();
                    </computed-field>
                    <computed-field field="new_value_length">
                        return ((String) new_value).length();
                    </computed-field>
                    <verification timeout="10" id-field="request">
                        <acceptance message="CMD_DEFINITION" secondary-id="ACK_POS"
id-field="request" result="OK" />
                        <acceptance message="CMD_DEFINITION" secondary-id="ACK_NEG"
id-field="request" result="FAIL" />
                        <execution message="CMD_DEFINITION" secondary-id="EXE_POS" id-
```

```
field="request" result="OK" />
                        <execution message="CMD_DEFINITION" secondary-id="EXE_NEG" id-
field="request" result="FAIL" />
                        <execution message="CMD_DEFINITION" secondary-id="CMD_NOK"
result="FAIL" />
                    </verification>
                </outbound>
                <!-- First Subsystem -->
                <outbound id="CMD_SUB1_RST_MP" message="CMD_DEFINITION" secondary-id=
"CMD_RST" type="ACTIVITY_DRIVEN" entity="177">
                    <fixed-field field="device_subsystem" value="1" type=
"UNSIGNED_INTEGER" />
                    <auto-increment counter-id="SEQ1" field="request" output-type=
"UNSIGNED_INTEGER"/>
                    <verification timeout="10" id-field="request">
                        <acceptance message="CMD_DEFINITION" secondary-id="ACK_POS"
id-field="request" result="OK" />
                        <acceptance message="CMD_DEFINITION" secondary-id="ACK_NEG"
id-field="request" result="FAIL" />
                        <execution message="CMD_DEFINITION" secondary-id="EXE_POS" id-
field="request" result="OK" />
                        <execution message="CMD_DEFINITION" secondary-id="EXE_NEG" id-
field="request" result="FAIL" />
                        <execution message="CMD_DEFINITION" secondary-id="CMD_NOK"
result="FAIL" />
                    </verification>
                </outbound>
                <outbound id="CMD_SUB1_SWP_MP" message="CMD_DEFINITION" secondary-id=
"CMD_SWP" type="ACTIVITY_DRIVEN" entity="178">
                    <fixed-field field="device_subsystem" value="1" type=
"UNSIGNED_INTEGER" />
                    <argument name="times" field="times" />
                    <auto-increment counter-id="SEQ1" field="request" output-type=
"UNSIGNED_INTEGER"/>
                    <verification timeout="10" id-field="request">
                        <acceptance message="CMD_DEFINITION" secondary-id="ACK_POS"
id-field="request" result="OK" />
                        <acceptance message="CMD_DEFINITION" secondary-id="ACK_NEG"
id-field="request" result="FAIL" />
                        <execution message="CMD_DEFINITION" secondary-id="EXE_POS" id-
field="request" result="OK" />
                        <execution message="CMD_DEFINITION" secondary-id="EXE_NEG" id-
field="request" result="FAIL" />
                        <execution message="CMD_DEFINITION" secondary-id="CMD_NOK"
result="FAIL" />
                    </verification>
                </outbound>
                <outbound id="CMD_SUB1_RBT_MP" message="CMD_DEFINITION" secondary-id=
"CMD_RBT" type="ACTIVITY_DRIVEN" entity="179">
                    <fixed-field field="device_subsystem" value="1" type=
"UNSIGNED_INTEGER" />
```

```
                    <argument name="delay" field="delay" />
                    <argument name="running" field="running" />
                    <auto-increment counter-id="SEQ1" field="request" output-type=
"UNSIGNED_INTEGER"/>
                    <verification timeout="10" id-field="request">
                        <acceptance message="CMD_DEFINITION" secondary-id="ACK_POS"
id-field="request" result="OK" />
                        <acceptance message="CMD_DEFINITION" secondary-id="ACK_NEG"
id-field="request" result="FAIL" />
                        <execution message="CMD_DEFINITION" secondary-id="EXE_POS" id-
field="request" result="OK" />
                        <execution message="CMD_DEFINITION" secondary-id="EXE_NEG" id-
field="request" result="FAIL" />
                        <execution message="CMD_DEFINITION" secondary-id="CMD_NOK"
result="FAIL" />
                    </verification>
                </outbound>
                <!-- Second Subsystem -->
                <outbound id="CMD_SUB2_RST_MP" message="CMD_DEFINITION" secondary-id=
"CMD_RST" type="ACTIVITY_DRIVEN" entity="187">
                    <fixed-field field="device_subsystem" value="2" type=
"UNSIGNED_INTEGER" />
                    <auto-increment counter-id="SEQ1" field="request" output-type=
"UNSIGNED_INTEGER"/>
                    <verification timeout="10" id-field="request">
                        <acceptance message="CMD_DEFINITION" secondary-id="ACK_POS"
id-field="request" result="OK" />
                        <acceptance message="CMD_DEFINITION" secondary-id="ACK_NEG"
id-field="request" result="FAIL" />
                        <execution message="CMD_DEFINITION" secondary-id="EXE_POS" id-
field="request" result="OK" />
                        <execution message="CMD_DEFINITION" secondary-id="EXE_NEG" id-
field="request" result="FAIL" />
                        <execution message="CMD_DEFINITION" secondary-id="CMD_NOK"
result="FAIL" />
                    </verification>
                </outbound>
                <outbound id="CMD_SUB2_SWP_MP" message="CMD_DEFINITION" secondary-id=
"CMD_SWP" type="ACTIVITY_DRIVEN" entity="188">
                    <fixed-field field="device_subsystem" value="2" type=
"UNSIGNED_INTEGER" />
                    <argument name="times" field="times" />
                    <auto-increment counter-id="SEQ1" field="request" output-type=
"UNSIGNED_INTEGER"/>
                    <verification timeout="10" id-field="request">
                        <acceptance message="CMD_DEFINITION" secondary-id="ACK_POS"
id-field="request" result="OK" />
                        <acceptance message="CMD_DEFINITION" secondary-id="ACK_NEG"
id-field="request" result="FAIL" />
                        <execution message="CMD_DEFINITION" secondary-id="EXE_POS" id-
field="request" result="OK" />
```

```
                        <execution message="CMD_DEFINITION" secondary-id="EXE_NEG" id-
field="request" result="FAIL" />
                        <execution message="CMD_DEFINITION" secondary-id="CMD_NOK"
result="FAIL" />
                    </verification>
                </outbound>
                <outbound id="CMD_SUB2_RBT_MP" message="CMD_DEFINITION" secondary-id=
"CMD_RBT" type="ACTIVITY_DRIVEN" entity="189">
                    <fixed-field field="device_subsystem" value="2" type=
"UNSIGNED_INTEGER" />
                    <argument name="delay" field="delay" />
                    <argument name="running" field="running" />
                    <auto-increment counter-id="SEQ1" field="request" output-type=
"UNSIGNED_INTEGER"/>
                    <verification timeout="10" id-field="request">
                        <acceptance message="CMD_DEFINITION" secondary-id="ACK_POS"
id-field="request" result="OK" />
                        <acceptance message="CMD_DEFINITION" secondary-id="ACK_NEG"
id-field="request" result="FAIL" />
                        <execution message="CMD_DEFINITION" secondary-id="EXE_POS" id-
field="request" result="OK" />
                        <execution message="CMD_DEFINITION" secondary-id="EXE_NEG" id-
field="request" result="FAIL" />
                        <execution message="CMD_DEFINITION" secondary-id="CMD_NOK"
result="FAIL" />
                    </verification>
                </outbound>
            </route>
        </tcp>
    </connections>
    <messages>
        <binary id="CMD_DEFINITION" location="binary_single_messages.xml">
            <type-marker>DEV4-BIN-CMD</type-marker>
        </binary>
    </messages>
</ns1:socket>
```

Other example configuration are provided in the test resources of the eu.dariolucia.reatmetric.driver.socket module.

Spacecraft

Overview

The eu.dariolucia.reatmetric.driver.spacecraft module provides a driver that allows the processing of telemetry data and the sending of telecommand to satellites compliant to the CCSDS (http://www.ccsds.org) standards.

Being a driver, a *spacecraft* module must be registered as such in the system Core's configuration.

As it must be able to support primarily by configuration the basic monitoring and control of CCSDS-compliant satellites, the driver configuration and design are overall complex and require preliminary knowledge of CCSDS and ECSS standards. Nevertheless, by breaking down the configuration and the design smaller pieces, such complexity can be managed. The information provided in this document allows to understand the fundamentals of the design of the driver, without pretending to be an exhaustive guide to satellite monitoring and control.

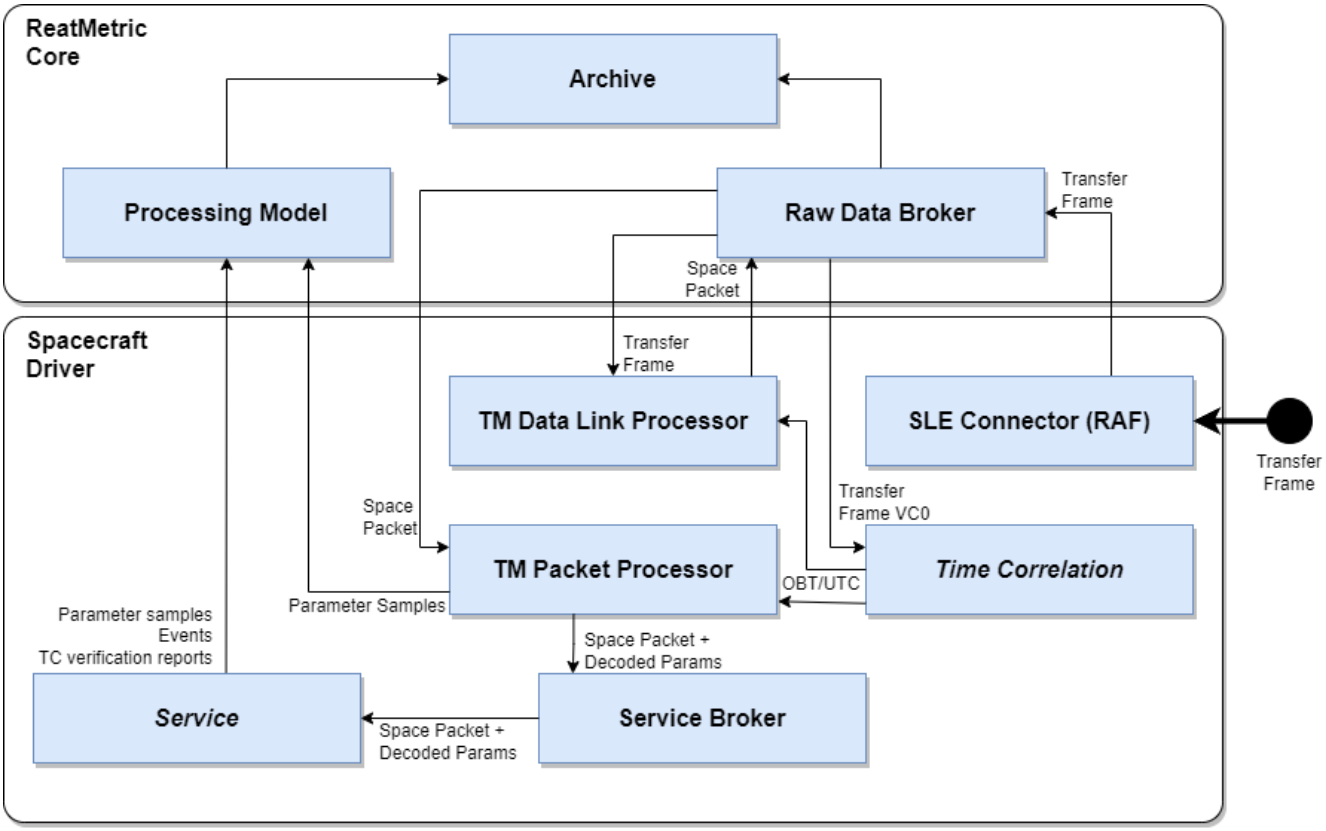
Design

Telemetry

The telemetry processing chain is composed by the following processing entities:

1. Interface that are able to receive TM data from the spacecraft by means of *connectors*. The spacecraft driver implements the support for CCSDS SLE RAF (Return All Frames) and RCF (Return Channel Frames) protocols. These protocols deliver TM or AOS Transfer Frames to the ReatMetric Raw Data Broker, for further dissemination and processing. Additional interfaces can be added by leveraging the driver extensions (see Command Connectors). De-randomisation is also performed by this entity, if configured.
2. A *TM Data Link Processor* that is subscribed to the Raw Data Broker for TM or AOS transfer frames (as defined in the driver configuration) and extracts TM packets from the transfer frames. If the security layer is configured, this entity also makes sure to decrypt the frame before extracting the TM packets. Extracted TM packets are identified according to the packet definitions defined as part of the driver configuration, time-correlated with the currently availabile time correlation coefficients, and forwarded to the Raw Data Broker for further dissemination and processing.
3. A *TM Packet Processor* that is subscribed to the Raw Data Broker to TM packets and decodes the TM parameters contained in each packet according to the packet structure definition defined as part of the driver configuration. The extracted TM parameters are time-correlated (depending on the structure definition of the packet) and mapped to ReatMetric's processing model parameters and injected into the processing model for processing (validity checking, calibration, alarm checking). Each TM packet together with the extracted TM parameters is also forwarded to the *Service Broker*.
4. The *Service Broker* is an entity responsible to distribute TM packets and telecommand (both TC packets and VC data units) lifecycle information to so-called *services*, as well to act as a registry for some specific internal interface implementations. A *service* in this context has the same meaning of that defined by the ECSS Packet Utilisation Standard. The driver provides basic (and sometimes partial) implementations of the following services, but the service support can be extended by means of extensions and the provided implementations easily replaced (see Services):
    a. Time service (PUS service 9), responsible for time correlation with a linear regression implementation;
    b. On-board event service (PUS service 5);
    c. On-board operations scheduling service (PUS service 11), with limited support to 11,4 telecommands: only one inner TC) and no support for (11,1), (11,2) and status reports;

d. Command verification service (PUS service 1).



Telecommand

The telecommand processing chain is composed by the following processing entities:

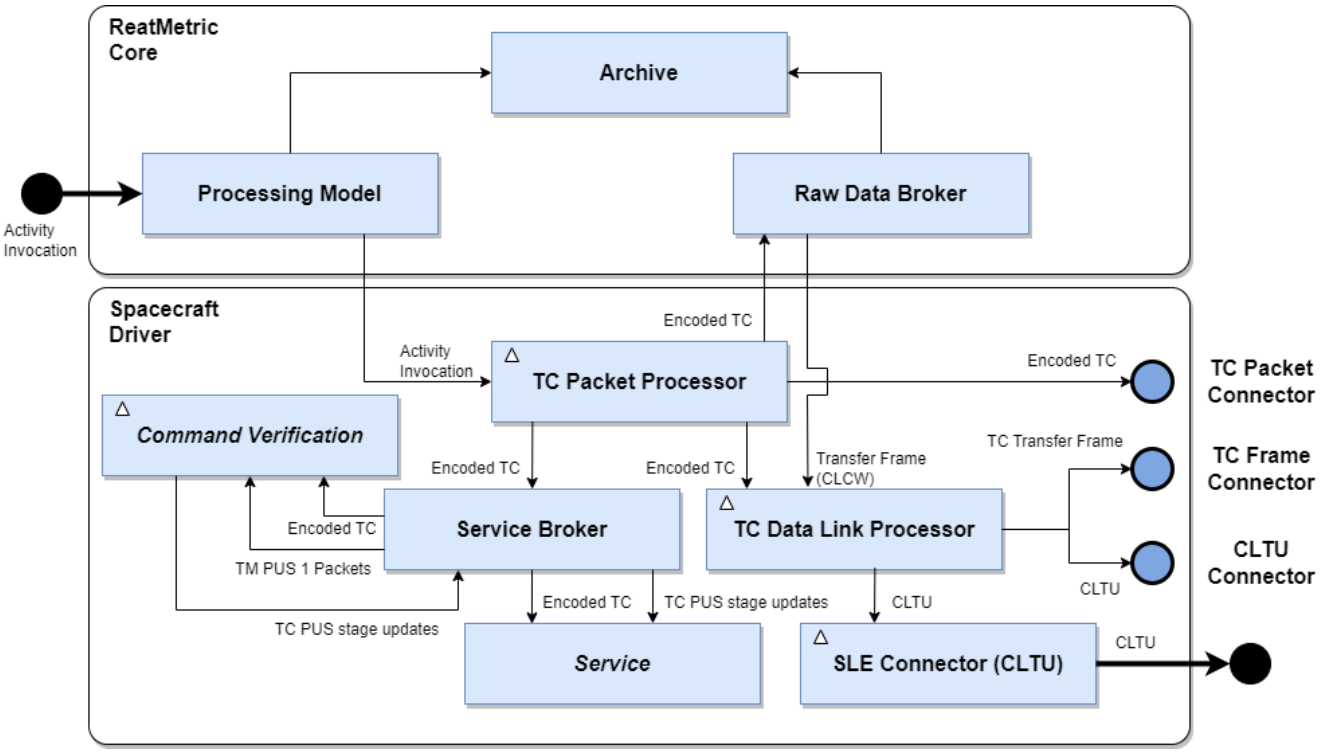
1. A *TC Packet Processor* that is an *Activity Handler* of the ReatMetric processing model: when an activity is invoked, and this activity is mapped to a telecommand packet or to a VC data unit (depending on the telecommand definition), the corresponding activity occurrence is forwarded to this element. This element encodes the telecommand, forwards it to the Raw Data Broker and to the *Service Broker*. If there is a service that fully takes over the processing of the telecommand (e.g. the on-board operations scheduling service), then the TC Packet Processor does not perform any additional processing. If instead there is no service taking over the processing of the telecommand, then the TC Packet Processor, depending on the selected route, might forward the telecommand to an external, TC packet-based interface (in case of TC packets) or to the TC Data Link Processor.
2. A *TC Data Link Processor* that receives TC packets and VC data units, and constructs TC frames out of them. This entity also manages the COP-1 protocol for all the possible TC VC IDs, as specified in the configuration. Selection of the AD/BD mode, segmentation, MAP-ID and grouping is achieved by configuration, and it can be overridden using the *activity properties* described further on. Depending on the selected route, the TC Data Link Processor forwards the TC frame to an external, TC frame-based interface or to an external, CLTU-based interface. In the latter case, this entity takes care of encoding the CLTU. Randomisation is also performed by this entity, if configured.
3. Interfaces that are able to send telecommands in TC packet, TC frame or CLTU form by means of *connectors*. The spacecraft driver implements the support for CCSDS SLE CLTU protocol, as well as for a simple TCP/IP CADU/CLTU connection.

An important aspect related to telecommand handling is the management of telecommand verification. ReatMetric Spacecraft driver defines a telecommand lifecycle, partitioned in *phases*, mapped to the ReatMetric processing model activity lifecycle:

1. **ENCODED**: the telecommand is encoded and announced as such to the service broker, so that all services involved in the telecommand handling are aware that a new telecommand is encoded and ready to be released (but not released yet).
2. **RELEASED**: the telecommand is possibly encoded into a TC frame (and possibly in a CLTU) and sent for release. The service broker is informed that the telecommand left the ReatMetric system and it is on its way to the satellite. In case of SLE CLTU being used, this phase corresponds to the reception of a positive CLTU-TRANSFER-DATA confirmation by the ground station or intermediate system. Other interfaces (e.g. TCP/IP CADU/CLTU connection) might report this phase if the connection is active and if it was possible to write the telecommand (including its envelope, i.e. TC frame/CLTU) to the connection. The linked activity occurrence is in TRANSMISSION.
3. **UPLINKED**: the telecommand has been transmitted from the ground station or intermediate system to the satellite. In case of SLE CLTU being used, this phase corresponds to the reception of an ASYNC-NOTIFY operation, reporting the radiation of the CLTU. Other interfaces (e.g. TCP/IP CADU/CLTU connection) might report this phase if the connection is active and if it was possible to write the telecommand (including its envelope, i.e. TC frame/CLTU) to the connection. This behaviour is obviously protocol- and intermediate system-dependant. The linked activity occurrence is in TRANSMISSION.
4. **RECEIVED_ONBOARD**: the telecommand is assumed or confirmed to be received on-board. In case of telecommand dispatch in TC frames using AD mode, this information is reported by the FOP-1 engine. In case of BD mode, the propagation delay, as defined as part of the driver configuration, is taken into account to announce the *expected* reception of the telecommand on-board. It must be kept in mind that this phase simply means that the telecommand should be at the satellite: it does not mean that the telecommand is in the availability of the on-board software for execution. In fact, this depends whether the telecommand is for direct execution or if it is for later, scheduled execution (in PUS terms, it is wrapped in a PUS 11,4 telecommand). The linked activity occurrence is in TRANSMISSION.
5. **SCHEDULED**: the telecommand is scheduled on-board for future execution. In the current driver implementation, this phase is announced by the On-board Operations Scheduling service, upon completion of the containing 11,4 telecommand. The linked activity occurrence is still in TRANSMISSION.
6. **AVAILABLE_ONBOARD**: the telecommand is assumed to be in the availability of the on-board software for immediate execution. This phase is announced by the PUS 1 service immediately after the RECEIVED_ONBOARD phase for immediate telecommands, or by the PUS 9 service after the SCHEDULED phase for time-tagged commands. The announcement of this phase triggers the Command Verification service to register the telecommand and start processing incoming PUS 1 TM packets, to receive and process the PUS 1 reports related to the telecommand. The linked activity occurrence is in EXECUTION.
7. **STARTED**: the telecommand is started its on-board execution. This phase is announced by the Command Verification service upon reception of the related PUS START success report (1,3). There is no such verification for VC data units telecommands.

8. **COMPLETED**: the telecommand completed its on-board execution. This phase is announced by the Command Verification service upon reception of the related PUS COMPLETED success report (1,7), or by any success report that is marked to be the last expected PUS report linked to the telecommand. There is no such verification for VC data units telecommands.
9. **FAILED**: at any time during the lifecycle of the telecommand, a transition to the FAILED phase means that the telecommand lifecycle must be stopped and a failure must be reported for the corresponding activity.

It could be observed that the PUS 1 reports related to the on-board acceptance and on-board progress are not reflected in the telecommand lifecycle phases. The reports are redundant to describe the lifecycle of a telecommand from the point of view of the ReatMetric design, but they are in any case forwarded as activity progress reports to the ReatMetric processing model, so no monitoring information is skipped.



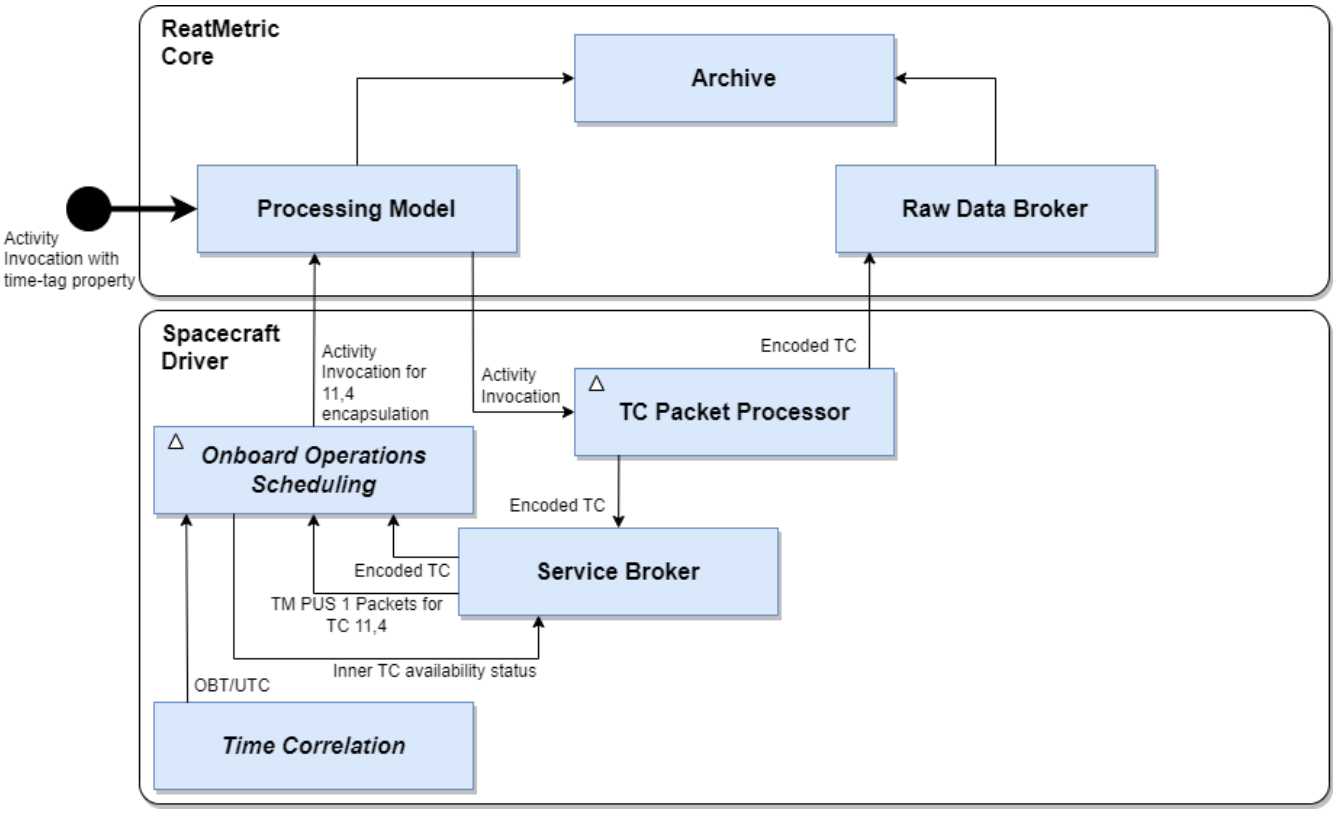
The handling of time-tagged telecommands deserves a dedicated explanation. In order to request the execution of an activity to be scheduled on-board, the Spacecraft driver defines two special properties that can be added to the activity invocation request:

- *tc-scheduled-time*: mandatory. It specifies the UTC execution time of the activity
- *onboard-sub-schedule-id*: optional. If specified, overrides the sub-schedule ID for telecommands wrapped into a PUS 11,4 packet

When an activity is invoked with the *tc-scheduled-time* specified, the encoded telecommand is declared to be fully taken over by the Onboard Scheduling service. The service will then use the encoded telecommand as argument for the invocation of the configured PUS 11,4 activity. From the point of view of the ReatMetric processing model, two activity occurrences will then be handled:

- the one linked to the telecommand being invoked and expected to be scheduled on-board
- the one linked to the PUS 11,4 corresponding activity

The processing of the PUS 11,4 activity occurrence is performed as a normal, immediate telecommand. Nesting of PUS 11,4 telecommands into PUS 11,4 telecommands is not supported by the driver implementation.



COP-1

The *TC Data Link Processor* entity instantiates one FOP engine for each configured TC virtual channel. This entity can execute activities defined with type *COP-1*. These activities are:

- **SET_AD**: it allows to set the type of service to be used for TC frame-based telecommand, changing the setting provided in the configuration.
- **DIRECTIVE**: it allows to request the execution of a directive to the specified FOP engine (TC VCID). This activity is particularly important, as it allows to initialise the COP-1 Sequence Controlled service.

In order to use these two activities, the fop_model.xml file (or its contents) must be included in the processing model. The names, types and arguments of the activities shall not be changed, as they are used by the implemented code. The locations, descriptions and entity IDs can be updated to fit the needs of the users.

In order to receive CLCW information to handle the Sequence Controlled service, the *TC Data Link Processor* is subscribed to the Raw Data Broker to receive TM frames from the configured satellite.

Services

The concept of *service* is a bit broader than the one specified by the ECSS PUS standard: a service in the ReatMetric Spacecraft driver term is simply an entity that can be notified of telecommands and telemetry packets/parameters disseminated in the system. A service is identified by its service number: the numbers between 1 and 255 are reserved for PUS services. Other numbers can be used

to register extension services, not necessarily linked to the service type of a PUS on-board implementation. What a service does with such information is service-implementation specific: it could maintain an internal model based on the received data, it can generate reports, it can communicate with external systems or with a database, and so on. The services provided by the driver implementation are (partial) implementations of some PUS-defined services.

The **Command Verification Service** (service number 1) is a full implementation of the PUS 1 service, and it manages the telecommand execution verification reports as generated by the satellite. This service does not require configuration.

Type class: *eu.dariolucia.reatmetric.driver.spacecraft.services.impl.CommandVerificationService*

The **Time Correlation Service** (service number 9) is a full implementation of the PUS 9 service, and it provides an implementation of the *ITimeCorrelation* interface. This service uses time packets to correlate the generation of VC0, VCC0 frames to UTC time and builds the related time couples. Depending on the number of time couples, time correlation coefficients are generated by direct interpolation or by linear regression. Upon updating the time correlation coefficients, the service distributes them via the Raw Data Broker. In this way, the coefficients can be stored and reloaded when the system is started again. This service requires configuration, as specified in the eu.dariolucia.reatmetric.driver.spacecraft.definition.services package, class TimeCorrelationServiceConfiguration.

Type class: *eu.dariolucia.reatmetric.driver.spacecraft.services.impl.TimeCorrelationService*

The **Direct Link Time Correlation Service** (service number 9) is an alternative implementation for time correlation, replacing the one based on time packets. The documentation of class DirectLinkTimeCorrelationService explains the behaviour of this implementation. It is useful in cases where the system is used with simulators with unreliable OBT generation or with missing time packet information, but it should not be used in real-case scenarios. This service does not require configuration.

Type class: *eu.dariolucia.reatmetric.driver.spacecraft.services.impl.DirectLinkTimeCorrelationService*

The **Onboard Event Service** (service number 5) is a full implementation of the PUS 5 service. Upon reception of a PUS 5 report, the service raises a corresponding event in the ReatMetric processing model. This service does not require configuration.

Type class: *eu.dariolucia.reatmetric.driver.spacecraft.services.impl.OnboardEventService*

The **Onboard Operations Scheduling Service** (service number 11) is a partial implementation of the PUS 11 service. This service requires configuration, as specified in the eu.dariolucia.reatmetric.driver.spacecraft.definition.services package, class OnboardOperationsSchedulingServiceConfiguration.

Type class: *eu.dariolucia.reatmetric.driver.spacecraft.services.impl.OnboardOperationsSchedulingService*

Typically, services are stand-alone and communicate among themselves via telecommand notifications and TM information, delivered by the service broker. However, direct service-to-service communication via method calls is possible: all registered services are subject to be located

by means of interfaces. An example can be seen in the code with the ITimeCorrelation interface: this interface is provided by one service (Time Correlation Service) and used by another service (Onboard Operations Scheduling Service). The latter looks up the interface provider via the service broker. In this way there is full decoupling between services. Custom services can leverage this system to register and look up custom interface and related implementations.

Security

The Spacecraft driver provides design support to include security decryption for telemetry frames and encryption for TC frames, based on the Space Data Link Security Protocol (CCSDS standard CCSDS 355.0-B-2). The support is provided by implementing the *ISecurityHandler* interface and registering such implementation in the Spacecraft driver configuration (see Configuration section below) as a service.

The contract specified by the *ISecurityHandler* interface is straightforward:

- As a service, it has access to all TC and TM via the service broker, which is particularly relevant in case of security management operations. Access to TM frames and TC frames is also possible via the Raw Data Broker;
- The implementation must provide the length of the security header and trailer fields for a given spacecraft id, virtual channel and type of frame;
- The implementation must provide the encryption of a given transfer frame, by returning the encrypted version of the transfer frame;
- The implementation must provide the decryption of a given transfer frame, by returning the decrypted version of the transfer frame.

As for all services, upon instantiation the context information and the driver configuration are provided by means of the *initialise()* method. Clean-up operations must be implemented inside the *dispose()* method.

As example, the driver provides one implementation of the *ISecurityHandler* interface, which uses AES as encryption/decryption algorithm and supports a set of keys specified as part of the extension configuration (package eu.dariolucia.reatmetric.driver.spacecraft.definition.security). The implementation uses a parameter to be specified in the processing model, to keep track of which key must be used for the encryption of TC frames. This parameter can be manually changed by the user. The structure is provided in the security_model.xml file.

Type class: *eu.dariolucia.reatmetric.driver.spacecraft.services.impl.AesEncryptionService*

Command Connectors

The driver provides the following points for external connection extensions:

- CLTU-based connectors, implementing the *ICltuConnector* interface;
- TC frame-based connectors, implementing the *ITcFrameConnector* interface;
- TC Packet-based connectors, implementing the *ITcPacketConnector* interface;
- Telemetry-only connectors, implementing the *IReceptionOnlyConnector* interface.

All interfaces extend the *ITransportConnector* ReatMetric interface, and therefore will be controllable as ReatMetric *connectors*. It is suggested to extend the abstract class *AbstractTransportConnector*, which provides already a skeleton for implementation. A connector implemented in this way shall take care of:

- Receiving TM data and providing it to the Raw Data Broker in a way that can be processed by the other elements of the TM processing chain;
- Forwarding TC data to the related external endpoint and handling the lifecycle of the specific data unit.

Three examples (CLTU/CADU-based connector, TM/TC packet connector) can be checked in the package eu.dariolucia.reatmetric.driver.spacecraft.connectors.

Type: *eu.dariolucia.reatmetric.driver.spacecraft.connectors.CltuCaduTcpConnector* Configuration: a string so defined: <IP address to connect to>:<TCP port to connect to>:<total block length>:<sync word length>

Type: *eu.dariolucia.reatmetric.driver.spacecraft.connectors.SpacePacketTcpConnector* Configuration: a string so defined: <IP address to connect to>:<TCP port to connect to>

In addition, a connector can declare to implement no support for commanding. It is typically use to implement connectors for pure TM reception support. An example provided by the Spacecraft driver is the Replay connector, which can read TM frames from an external archive and ingest such data in the running ReatMetric instance.

Type: *eu.dariolucia.reatmetric.driver.spacecraft.connectors.TmPacketReplayConnector* Configuration: not required

Activity Invocation Properties

Packet level

pus-ack-override

Override the PUS ack flags specified in the packet definition.

Format: `[X|-][X|-][X|-][X|-]`

Example: `X-XX`

pus-source-override

Override the PUS source ID specified in the packet definition or configuration.

Format: `[0-9]+`

Example: `14`

map-id-override

Override the Map ID specified in the packet definition.

Format: `[0-9]+`

Example: `2`

tc-scheduled-time

Specify the on-board execution time of the telecommand. If this property is specified, the encoded telecommand will be wrapped into the configured PUS 11,4 command.

Format: `ISO-8601 instant format`

Example: `2011-12-23T10:15:30Z`

tc-vc-id-override

Override the TC VC ID specified in the configuration for generated TC frames.

Format: `[0-7]`

Example: `1`

use-ad-mode-override

Override the currently specified TC frame transfer mode for generated TC frames. If set to 'true', the TC frame will have the bypass flag unset.

Format: `true|false`

Example: `true`

group-tc-name

Inform the TC Data Link processor that the TC packet is part of a group and shall be encoded inside a single frame with other commands. The string set here identifies the name of the group.

Format: `[0-9a-zA-Z]+'`

Example: `Group1`

group-tc-transmit

Inform the TC Data Link processor that the TC packet is the last one of the group identified with the group-tc-name property. The group is closed, encoded and the resulting frame transmitted.

Format: `true|false`

Example: `true`

onboard-sub-schedule-id

Override the sub-schedule ID for telecommands wrapped into a PUS 11,4 packet.

Format: `[0-9]+`

Example: 2

linked-scheduled-activity-occurrence

Allow to keep tracking between a scheduled activity occurrence and the PUS 11,4 TC. The implementation of the PUS 11 in this module supports a single TC per PUS 11,4.

Format: *internal*

Configuration

Being a driver, the *spacecraft* module must be registered as such in the system configuration file. You need to have a spacecraft module registration for each satellite that you need to process.

```
<ns1:core xmlns:ns1="http://dariolucia.eu/reatmetric/core/configuration">
    <name>Test System</name>
    <log-property-file>$HOME\Reatmetric\reatmetric_test\log.properties</log-property-
file>
    <definitions-location>$HOME\Reatmetric\reatmetric_test\processing</definitions-
location>
    <driver name="Spacecraft Driver" type=
"eu.dariolucia.reatmetric.driver.spacecraft.SpacecraftDriver"
    configuration="$HOME\Reatmetric\reatmetric_test\spacecraft"/>
</ns1:core>
```

The folder specified in the *configuration* attribute of the *driver* element must contain a file named *configuration.xml*, which defines the configuration properties of the driver, and a file named *tmtc.xml*, which contains the definition of the TM and TC packets and VC data units.

In addition, the folder might contain a subfolder called *sle*, which shall contain the SLE configuration files to configure the SLE service instances. For further information about these files, please check the Javadoc and examples provided in the eu.dariolucia.ccsds.sle.utl module.

Main Configuration File

The configuration structure of the eu.dariolucia.reatmetric.driver.spacecraft module is defined in the package eu.dariolucia.reatmetric.driver.spacecraft.definition. It is an XML file named *configuration.xml* using namespace definition *http://dariolucia.eu/reatmetric/driver/spacecraft*.

An example of such file is presented below. As it covers all the TM/TC characteristics of a CCSDS/ECSS spacecraft, except the packet structure and mapping, the configuration is not easy to understand. Therefore, the following example provide extensive comments covering all elements and configuration aspects.

```
<ns1:spacecraft xmlns:ns1="http://dariolucia.eu/reatmetric/driver/spacecraft">
    <!-- Spacecraft name -->
    <name>TEST</name>
    <!-- Spacecraft ID -->
    <id>123</id>
```

```
    <!-- Agency epoch -->
    <obt-epoch>2000-01-01T00:00:00.000Z</obt-epoch>
    <!-- Propagation delay -->
    <propagation-delay>0</propagation-delay>
    <!-- TC datalink configuration -->
    <tc randomize="true" fecf="true" ad-mode-default="true">
        <!-- TC VC configuration -->
        <tc-vc-descriptor tc-vc="0" segmentation="true" default-tc-vc="true" system-
entity-path="SPACE.FOP.TCVC0"
            use-security="true" access-mode="PACKET" />
        <tc-vc-descriptor tc-vc="1" segmentation="false" default-tc-vc="false" system-
entity-path="SPACE.FOP.TCVC1"
            use-security="true" access-mode="VCA" />
    </tc>
    <!-- TM datalink configuration -->
    <tm fecf="true" derandomize="false" frame-length="1115">
        <!-- TM VC configuration -->
        <tm-vc-descriptor id="0" process-type="PACKET" />
        <tm-vc-descriptor id="1" process-type="PACKET" />
        <tm-vc-descriptor id="2" process-type="VCA" />
        <tm-vc-descriptor id="7" process-type="IGNORE" />
        <!-- Protocol type -->
        <type>TM</type>
    </tm>
    <!-- TM packet configuration -->
    <tm-packet parameter-id-offset="100000">
        <!-- Default PUS configuration -->
        <default-pus-configuration
            destination-field-length="0"
            packet-subcounter-present="false"
            tm-spare-length="0"
            tm-pec-present="NONE">
            <!-- OBT time field format in PUS header: CUC or CDS -->
            <obt-cuc-config explicit-p-field="false" coarse="4" fine="2" />
            <!-- <obt-cds-config explicit-p-field="false" short-days="true" subtime-
byte-res="2" /> -->
        </default-pus-configuration>
        <!-- It is possible to override the default PUS configuration for specific
APIDs, as per following block.
        More sub-elements can be added. -->
        <!--
        <apid-pus-configuration apid="103" destination-field-length="8"
            packet-subcounter-present="true"
            tm-spare-length="0"
            tm-pec-present="CRC">
            <obt-cuc-config explicit-p-field="false" coarse="4" fine="3" />
        </apid-pus-configuration>
        -->
    </tm-packet>
    <!-- TC packet configuration -->
    <tc-packet
```

```
        telecommand-id-offset="0"
        activity-tc-packet-type="TC"
        pus-source-id-default-value="123"
        pus-source-id-length="8"
        pus-spare-length="0"
        tc-pec-present="CRC">
    </tc-packet>
    <!-- Service configuration -->
    <services>
        <service type=
"eu.dariolucia.reatmetric.driver.spacecraft.services.impl.TimeCorrelationService"
            configuration=
"$HOME\Reatmetric\reatmetric_test\test\services\time_correlation.xml" />
        <!-- <service
type="eu.dariolucia.reatmetric.driver.spacecraft.services.impl.DirectLinkTimeCorrelati
onService"

configuration="$HOME\Reatmetric\reatmetric_test\test\services\time_correlation.xml" />
-->
        <service type=
"eu.dariolucia.reatmetric.driver.spacecraft.services.impl.OnboardOperationsSchedulingS
ervice"
            configuration=
"$HOME\Reatmetric\reatmetric_test\test\services\onboard_scheduling.xml" />
        <service type=
"eu.dariolucia.reatmetric.driver.spacecraft.services.impl.OnboardEventService"
configuration="" />
        <service type=
"eu.dariolucia.reatmetric.driver.spacecraft.services.impl.CommandVerificationService"
configuration="" />
        <service type=
"eu.dariolucia.reatmetric.driver.spacecraft.services.impl.AesEncryptionService"
         configuration="$HOME\Reatmetric\reatmetric_test\test\security\security.xml"
/>
    </services>
    <!-- External connectors -->
    <external-connectors>
        <external-connector type=
"eu.dariolucia.reatmetric.driver.spacecraft.connectors.CltuCaduTcpConnector"
configuration="127.0.0.1:23532:1279:4" data-unit-type="CLTU" />
        <external-connector type=
"eu.dariolucia.reatmetric.driver.spacecraft.connectors.SpacePacketTcpConnector"
configuration="127.0.0.1:33532" data-unit-type="TC_PACKET" />
        <!--
        <external-connector
type="eu.dariolucia.reatmetric.driver.spacecraft.connectors.TmPacketReplayConnector"
configuration="" data-unit-type="NONE" />
        -->
    </external-connectors>
</ns1:spacecraft>
```

Element <**name**> (mandatory, string): this element assigns the name of the driver instance, it does not play any role in the processing of the spacecraft data.

Element <**id**> (mandatory, integer): this element must be set to the spacecraft ID as present in the TM frames, and it will be used as value for the spacecraft ID when constructing TC frames. TM frames containing a different spacecraft ID will not processed by this driver instance.

Element <**obt-epoch**> (optional, string with XML datetime format): the value is used to correctly interpret the time information when delivered with CUC or CDS format. When not specified it is set to 1st Jan 1958 00:00:00.

Element <**propagation-delay**> (optional, integer): this value (in microseconds) is used to derive the actual frame generation time (from the Earth-received time) when performing time correlation calculations, and to open verification windows for telecommand verification purposes, taking into account the time required by the telecommand to travel and reach the spacecraft. When not specified it is set to 0.

Element <**tc**> (mandatory): this element lists the available TC data link properties and their characteristics. At least one tc-vc-descriptor child element must be present beneath this element. The following attributes are defined:

- *randomize* (optional, boolean, default: true): TC frames will be randomized using the polynomial defined by the CCSDS standard. To disable this behaviour, the attribute shall be set to false.
- *fecf* (optional, boolean, default: true): TC frames will contain the Frame Error Control Field at the end (2 bytes). To disable the addition of the FECF, the attribute shall be set to false.
- *ad-mode-default* (optional, boolean, default: false): if true, TC frames will have the Sequence Controlled flag set. The default setting implies that all TC frames are sent in Bypass mode, i.e. they do not require the COP-1 to be active. To require the use of the Sequence Controlled mode by default, this attribute shall be set to true and the COP-1 shall be activated using the standard-specified activation procedure and related directives.

Sub-element <**tc**><**tc-vc-descriptor**> (mandatory, at least one defined). This element defines the characteristics of a TC virtual channel. The following attributes are defined:

- *tc-vc* (mandatory, integer): the virtual channel ID of the TC VC.
- segmentation (optional, boolean, default: false): if set, the virtual channel will use TC segments to deliver telecommands.
- *default-tc-vc* (optional, boolean, default: false): if set, the virtual channel will be used for all TC frames, whose related activity invocations do not specify explicitly a TC VC to be used (see the activity invocation properties).
- *system-entity-path* (optional, string, default: null): if specified, and if the specified processing model element exists with the right structure (see the fop_model.xml file), then the driver will publish status information related to the FOP linked to the defined TC virtual channel. If not present or if with wrong structure, the system will raise warnings at startup, but it will not block COP operations.
- *use-security* (TODO: implement support, optional, string, default: false): if set, all TC frames on this virtual channel will be encrypted using the registered ISecurityHandler service.

- *access-mode* (optional, enumeration: PACKET, VCA, default: PACKET): access mode of the virtual channel. If PACKET is selected, then only TC space packets can be delivered on the virtual channel. If VCA is selected, then only raw VC data units can be delivered on the virtual channel. Attempting to send the wrong data unit to the virtual channel will cause the telecommand to fail upon release.

Element <**tm**> (mandatory). This element configures TM data link properties and their characteristics. All VCs possibly defined by the selected protocol (see type element below) are processed with the default configuration, unless the virtual channel is configured ad-hoc in the list of tm-vc-descriptor elements. The following attributes are defined:

- *type* (optional, enumeration: TM, AOS, default: TM): the type of transfer frames delivering telemetry data. It applies to all virtual channels.
- *fecf* (optional, boolean, default: false): if set, the Frame Error Control Field is assumed present in all transfer frames.
- *ocf* (optional, boolean, default: true): if set, the OCF is assumed present in all transfer frames.
- *derandomize* (optional, boolean, default: false): if set, received frames shall be derandomised by the receiving connector, before distributing the frames in the Raw Data broker. This processing depends on the connector: for instance, SLE connectors do not provide any derandomization, as this is not specified by the SLE standard.
- *aos-fhec* (optional, boolean, default: false): if set, the AOS Frame Header Error Control is assumed present in AOS frames. Only considered if type = AOS.
- *aos-insert-zone-length* (optional, integer, default: 0): the length of the Insert Zone for AOS frames. Only considered if type = AOS.
- *frame-length* (optional, integer, default: -1): the length of the TM/AOS transfer frame. This information is available to connectors that require it, e.g. to read CADUs and frames from external sources. SLE connectors do not require this information. It is **strongly** suggested to set this value, to avoid undefined behaviours when using connectors that are not SLE connectors, which might rely on this setting.

Sub-element <**tm**><**tm-vc-descriptor**> (optional). This element defines the characteristics of a TM virtual channel. The following attributes are defined:

- *id* (mandatory, integer): the virtual channel id.
- *process-type* (optional, enumeration: PACKET, VCA, IGNORE, default: PACKET): the data delivered by the virtual channel. If PACKET is set, then space packets are assumed to be delivered by the virtual channel. If VCA is set, then it is assumed that raw data is written in each transfer frame, not encapsulated in space packets. If IGNORE is set, then transfer frames are not further processed.

Sub-element <**tm**><**type**> (optional, enumeration: TM, AOS, default: TM): data link protocol used to deliver telemetry data.

Element <**tm-packet**> (mandatory). This element provides information related to the mapping of packet parameters to processing model parameters, and information about properties defined for PUS packet secondary headers. The following attributes are defined:

- *parameter-id-offset* (optional, integer, default: 0): when a telemetry packet or VC data unit is decoded, i.e. parameters are extracted, they are assigned to the defined external-id as per structure definition. The value specified in this field is then added to the parameter's external-id to compute the ID of the parameter in the ReatMetric processing model. This approach simplifies the configuration in case of identical spacecraft (sharing the TM/TC structure definition), which can be both included in a single processing model, keeping the same structure definition but using different offsets.

Sub-element <**tm-packet**><**default-pus-configuration**> (optional): if this element is not provided, and no APID-specific configuration is provided, then no PUS secondary header will be attempted to be extracted and processed, in case the TM packet indicates the presence of a secondary header. This element has a sub-element to define the format of the OBT time field (CUC or CDS) as encoded in the PUS header. Support for other, non-PUS secondary headers is not present in the current implementation. The following attributes are defined:

- *destination-field-length* (optional, integer, default: 0): length in bits of the PUS header destination field.
- *packet-subcounter-present* (optional, boolean, default: false): if set, the PUS packet sub-counter is assumed present.
- *tm-spare-length* (optional, integer, default: 0): length in bits of the PUS header spare field.
- *tm-pec-present* (optional, enumeration: NONE, CRC, ISO, default: NONE): presence and type of the packet error control field (2 bytes).

Sub-element <**tm-packet**><**default-pus-configuration**><**obt-cuc-config**> (optional, mandatory choice with obt-cds-config): this element defines the characteristics of a CUC-encoded time field as per CCSDS 301.0-B-4. The following attributes are defined:

- *explicit-p-field* (optional, boolean, default: false): whether the P-field is present or not.
- *coarse* (optional, integer, default: 4): number of bytes for the coarse CUC field.
- *fine* (optional, integer, default: 3): number of bytes for the fine CUC field.

Sub-element <**tm-packet**><**default-pus-configuration**><**obt-cds-config**> (optional, mandatory choice with obt-cuc-config): this element defines the characteristics of a CDS-encoded time field as per CCSDS 301.0-B-4. The following attributes are defined:

- *explicit-p-field* (optional, boolean, default: false): whether the P-field is present or not.
- *short-days* (optional, boolean, default: true): if set, 16 bits are considered for the days field. If false, 24 bits.
- *subtime-byte-res* (optional, integer, default: 2): possible values are 0 (no sub-milli field), 2 (number of microseconds in millisecond), or 4 (number of picoseconds in millisecond).

Sub-element <**tm-packet**><**apid-pus-configuration**> (optional): this element has the same structure of the **default-pus-configuration**, with the additional presence of the following attribute:

- *apid* (mandatory, integer): APID of the TM packets, for which the configuration applies.

Element <**tc-packet**> (mandatory): this element defines the characteristics of the TC packets that

are encoded by the Spacecraft driver. The following attributes are defined:

- *telecommand-id-offset* (optional, integer, default: 0): the offset to be applied to the activity ID to derive the packet ID as specified in the packet structure definition.
- *activity-tc-packet-type* (optional, string, default: "TC"): the type of TC packets as defined in the packet structure definition.
- *pus-source-id-default-value* (optional, integer, default: null): the source ID to be set in the corresponding PUS header field, in case it is not overriden by the activity invocation, and in case the structure definition does not contain the value of the corresponding field. If this attribute is null, and no source id is specified in the activity invocation and in the packet structure definition, the source ID field will not be encoded.
- *pus-source-id-length* (optional, integer, default: 0): the length in bits of the source ID field in the PUS header.
- *pus-spare-length* (optional, integer, default: 0): the length in bits of the spare field in the PUS header.
- *tc-pec-present* (optional, enumeration: CRC, ISO, NONE, default: CRC): the type of 2-bytes checksum to be put at the end of the TC packet. If set to NONE, the checksum field will not be encoded.

Element <**services**> (mandatory): this element contains a list of zero or more <service> sub-elements.

Sub-element <**services**><**service**> (optional): this element register the implementation of a service in the Spacecraft driver. The service implementation must implement the interface IService and it must be registered as exposed Java service in the corresponding module-info file. The following attributes are defined:

- *type* (mandatory, string): the fully qualified name of the Java class implementing the service.
- *configuration* (mandatory, string): a string identifying the configuration to be used for the implementation of the service. Typically, it is a path to a file, but it is implementation-dependant.

Element <**external-connectors**> (mandatory): this element contains a list of zero or more <external-connector> sub-elements.

Sub-element <**external-connectors**><**external-connector**> (optional): this element register the implementation of an external connector in the Spacecraft driver. The connector implementation must implement the interface ICltuConnector (in case it sends CLTUs to external systems), ITcFrameConnector (in case it sends TC frame to external systems), ITcPacketConnector (in case it sends TC packets to external systems), or IReceptionOnlyConnector (in case it has no support for telecommands), and it must be registered as exposed Java service in the corresponding module-info file. The following attributes are defined:

- *type* (mandatory, string): the fully qualified name of the Java class implementing the connector.
- *configuration* (mandatory, string): a string identifying the configuration to be used for the implementation of the service. Typically, it is a path to a file, but it is implementation-dependant.

- *data-unit-type* (mandatory, enumeration: CLTU, TC_FRAME, TC_PACKET, NONE): the telecommand type that the connector handles. It must match with the interface implemented by the connector.

TM/TC Structure Definition File

The TC and TM packet structures are defined inside the tmtc.xml file. Such file uses the format defined by the eu.dariolucia.ccsds.encdec module. The description of such file in this documentation is explained by an example, and only focuses to the parts linked to the processing of the Spacecraft driver, assuming a spacecraft that uses the ECSS PUS standard. The concept can be extended also for non-PUS spacecraft.

For further information, it is advised to check the Javadoc of the classes of the package eu.dariolucia.ccsds.encdec.definition.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:packet_definitions xmlns:ns2="http://dariolucia.eu/ccsds/encdec">
    <id_fields>
        <field id="I-APID" offset="0" len="2" and="2047" or="0" lshift="0" rshift="0
"/>
        <field id="I-PUS-TYPE" offset="7" len="1" and="-1" or="0" lshift="0" rshift="
0"/>
        <field id="I-PUS-SUBTYPE" offset="8" len="1" and="-1" or="0" lshift="0"
rshift="0"/>
        <field id="I-18-32" offset="18" len="4" and="-1" or="0" lshift="0" rshift="0
"/>
        <field id="I-22-24" offset="22" len="3" and="16777215" or="0" lshift="0"
rshift="0"/>
    </id_fields>
    <packets>
        <packet id="TMPACKET1" external_id="114001" description="Service Packet
(106,12)" type="TM">
            <identification>
                <match field="I-APID" value="33"/>
                <match field="I-PUS-TYPE" value="106"/>
                <match field="I-PUS-SUBTYPE" value="12"/>
                <match field="I-18-32" value="1440852322"/>
                <match field="I-22-24" value="129723"/>
            </identification>
            <structure>
                <parameter id="EI-DMT70003-000">
                    <location_absolute bit_absolute="200"/>
                    <type_fixed type="UNSIGNED_INTEGER" length="8"/>
                    <time offset="0"/>
                    <parameter_fixed parameter="DMT70003"/>
                </parameter>
                <parameter id="EI-DMT70004-000">
                    <location_absolute bit_absolute="208"/>
                    <type_fixed type="UNSIGNED_INTEGER" length="16"/>
                    <time offset="0"/>
```

```
                    <parameter_fixed parameter="DMT70004"/>
                </parameter>
                <parameter id="EI-DMT70005-000">
                    <location_absolute bit_absolute="232"/>
                    <type_fixed type="UNSIGNED_INTEGER" length="8"/>
                    <time offset="0"/>
                    <parameter_fixed parameter="DMT70005"/>
                </parameter>
                <parameter id="EI-DMT70006-000">
                    <location_absolute bit_absolute="224"/>
                    <type_fixed type="UNSIGNED_INTEGER" length="8"/>
                    <time offset="0"/>
                    <parameter_fixed parameter="DMT70006"/>
                </parameter>
                <parameter id="EI-YFTRW107-000">
                    <location_absolute bit_absolute="240"/>
                    <type_fixed type="UNSIGNED_INTEGER" length="32"/>
                    <time offset="0"/>
                    <parameter_fixed parameter="YFTRW107"/>
                </parameter>
            </structure>
        </packet>
        [...]
        <packet id="TCPACKET1" external_id="116873" description="(102,17) Modify Limit
Monitoring Check Def for float parameter" type="TC">
            <identification/>
            <structure>
                <parameter id="PARAMETER_ID">
                    <type_fixed type="UNSIGNED_INTEGER" length="32"/>
                </parameter>
                <parameter id="MIN_VALUE">
                    <type_fixed type="REAL" length="1"/>
                </parameter>
                <parameter id="MAX_VALUE">
                    <type_fixed type="REAL" length="1"/>
                </parameter>
            </structure>
            <extension>APID=231.PUSTYPE=102.PUSSUBTYPE=17.ACKS=XX-
X.CRITICAL=true</extension>
        </packet>
    </packets>
    <parameters>
        <parameter id="DMT70003" external_id="2634" description="PARARR_INDEX">
            <type type="UNSIGNED_INTEGER" length="8"/>
        </parameter>
        <parameter id="DMT70004" external_id="2639" description="PARARR_PTC_PFC">
            <type type="UNSIGNED_INTEGER" length="16"/>
        </parameter>
        [...]
    </parameters>
```

```
</ns2:packet_definitions>
```

The first element <**id_fields**> identifies all the fields in a packet, which contribute to its identification. For PUS-defined spacecraft, these are typically the APID, PUS type, PUS subtype fields, plus some addition fields identifying the specific ID of a PUS report.

The definition of each <**field**> includes:

- the *id* of the field, which is then referenced by the <match> element in the packet definition;
- the *offset* of the field in bytes from the start of the packet;
- the *length* of the field in bytes: 1, 2, 4 and 8 are supported;
- the *and* value (default: -1, i.e. all 1 in binary), which is applied to the extracted field value;
- the *or* value (default: 0, i.e. all 0 in binary), which is applied to the extracted field value, after the *and* value;
- the *lshift* value in bits (default: 0, i.e. no left shift), which is applied to the extracted field value, after the *and* and *or* values;
- the *rshift* value in bits (default: 0, i.e. no left shift), which is applied to the extracted field value, after the *and*, *or* and *lshift* values.

The second element <**packets**> contains the list of packets (TM and TC). For each <**packet**>, the following attributes:

- attribute *id* (mandatory, string): identifies the packet definition;
- attribute *external_id* (mandatory, integer): used to map a packet to an external entity (e.g. in case of PUS 5 events, this value is equal to the event object in the ReatMetric processing model mapped to such on-board event; in case of TC packets, this value must match the value of the ID of the corresponding activity plus offset, as defined in the configuration.xml file);
- attribute *description* (optional, string, default:): description of the packet
- attribute *type* (mandatory, string): ReatMetric uses the following hardcoded convention: "TM" for TM packets, "TC" for TC packets.

Each <**packet**> element contains:

- the identification criteria, as a list of <**match**> elements, each of them referring to one identification field and providing the expected value for such field. Note that each packet might specify only a subset of the defined identification fields. These fields are needed for TM packets only, since TC packets are encoded by the ReatMetric driver and do not need an identification.
- the structure of the packet, as a sequence of <**parameter**> elements and related sub-elements. Refer to the eu.dariolucia.ccsds.encdec Javadoc for the detailed explanation of each sub-element. For telemetry parameters, each structure parameter must be mapped to a top level parameter
- an extension string, used for TC packets, composed by a sequence of key-value pairs separated by a dot. The following keys can be specified:
  - APID: the APID value to be set in the encoded TC packet;

- PUSTYPE: the PUS type value to be set in the encoded TC packet;
- PUSSUBTYPE: the PUS subtype value to be set in the encoded TC packet;
- ACKS: a string with 4 characters, either X or -, for the PUS verification stages Acceptance, Start, Progress, Completed
- SOURCEID: an integer to be used as source ID in the PUS header (if the source field is configured)
- MAPID: an integer to overwrite the default MAP ID of the TC segment (if segmentation is configured)
- Other values can be used but are not taken into account by the current Spacecraft driver implementation.

The third element <**parameters**> contains the definition of top level parameters, which are ultimately mapped to parameters in the ReatMetric processing model via the *external_id* attribute value, plus the offset configured inside the Spacecraft driver configuration. When a packet is received, the packet is first identified, then the structure parameters are extracted and mapped to top-level parameters, which are finally forward to the processing model for calibration and checking.

Time Correlation Configuration File

The Time Correlation Service implementation requires a configuration file, as specified below.

```
<ns1:time-service xmlns:ns1="http://dariolucia.eu/reatmetric/driver/spacecraft/time-
service"
            generation-frame-period="256"
            on-board-delay="0"
            num-time-couples="10"
            generation-period-reported="true"
            maximum-frame-time-delay="1200000000000000">
    <time-format explicit-p-field="true" coarse="4" fine="2" />
</ns1:time-service>
```

The attributes have the following usage: - *generation-frame-period* (optional, integer, default: 256): VC0 frame counter modulo that triggers the generation of a time packet; - *on-board-delay* (optional, integer, default: 0): on-board delay in microseconds, to be taken into account when computing the propagation time; - *num-time-couples* (optional, integer, default: 2): number of latest time couples to be considered by the linear regression (minimum value is 2); - *generation-period-reported* (optional, boolean, default: false): whether the generation period field is contained in the time packet; - *maximum-frame-time-delay* (optional, long, default: 1200000000L): the maximum delay in microseconds that it is allows to match a frame identified by the generation period with the corresponding time packet. - *initial-coefficient-m* (optional, double, default: 1.0): initial m coefficient - *initial-coefficient-q* (optional, double, default: 0.0): initial q coefficient

The <**time-format**> sub-element specifies the CUC format of the time field contained in the time packet. CDS format is not supported.

Onboard Operations Scheduling Configuration File

The On-Board Operations Scheduling Service implementation requires a configuration file, as specified below.

```
<ns1:onboard-scheduling-service xmlns:ns1=
"http://dariolucia.eu/reatmetric/driver/spacecraft/onboard-scheduling-service"
    schedule-activity-path="SPACE.SC.SCHEDULE.SCHEDULEA0000"
    sub-schedule-id-name="Subschedule-ID"
    array-used="false">
</ns1:onboard-scheduling-service>
```

The attributes have the following usage: - *schedule-activity-path* (mandatory, string): the path to the activity mapped to the 11,4 TC; - *sub-schedule-id-name* (optional, string, default: null): the name of the 11,4 activity argument that is used to specify the sub-schedule ID; - *array-used* (optional, boolean, default: false): whether an array for the 11,4 TC field is used; - *num-commands-name* (optional, string, default: null): the name of the 11,4 activity argument that is used to specify the number of TC commands in the TC 11,4 packet.

Onboard Event Configuration File

The On-Board Event Service implementation optionally requires a configuration file, as specified below.

```
<ns1:event-service xmlns:ns1="http://dariolucia.eu/reatmetric/driver/spacecraft/event-
service"
    event-id-offset="0">
</ns1:event-service>
```

The attributes have the following usage: - *event-id-offset* (optional, integer, default: 0): the event offset to be added to the TM packet external ID, as specified in the packet definition, to compute the event object in the processing model, to be raised.

AES Encryption Configuration File

The AES Encryption implementation requires a configuration file, as specified below. Note that the implementation is purely an example and not an operational one. It is strongly advised to check the implementation code of the AesEncryptionService class (and the related constraints), to understand how to use this service.

This implementation uses a security header length of 18 bytes: 2 for the SPI identification, 16 for the initialisation vector. The security trailer is made of 8 bytes, containing the 8 least significant bytes of the SHA-256 checksum, computed over the concatenation of the bytes in scope, as specified by the CCSDS standard.

When encrypting TC frames, the SPI to be used is taken from the (integer) value of the parameter indicated by the *tc-spi-parameter-path* attribute.

```
<ns1:aes-security-handler xmlns:ns1=
"http://dariolucia.eu/reatmetric/driver/spacecraft/security/aes"
tc-spi-parameter-path="SPACE.SC.SECURITY.TCSPI"
default-tc-spi="1">
    <salt>AABBCCDDEEFF1122</salt>
    <tm-spi id="1" password="TestPassword1" />
    <tm-spi id="2" password="Short" />
    <tm-spi id="3" password="V3ryL0ngW31rdP455w0rd" />
    <tc-spi id="1" password="TcCommandPW1!" />
    <tc-spi id="2" password="TcCommandPW2!" />
    <tc-spi id="3" password="TcCommandPW3!" />
</ns1:aes-security-handler>
```

Serial

Overview

The eu.dariolucia.reatmetric.driver.serial module provides a driver that allows sending real-time monitoring data (currently parameters and log message) to devices connected to a specified serial port.

Being a driver, a *serial* module must be registered as such in the system Core's configuration.

The communication protocol implemented by this driver has been designed to allow visualisation of ReatMetric real-time data using very low power/low resources terminals, such as the Atari Portfolio™ (1989) via serial or parallel ports, or basic serial terminals. It is not supposed to be used for fully fledged data transfers. Use the available remoting connector for that, or write your own driver to match your needs.

The protocol is fully synchronous, with the knowledge of the session kept on the server side (i.e. on this driver). Interactions are always started by the client. The server always waits for client instructions. All messages are US-ASCII encoded. The client **must** wait for the server to reply to its command and complete the response, before sending the next command.

Server-side, the protocol has two main states:

DEREGISTERED: all messages received when the server is in this state are rejected with message ABORT\r and the server remains in the DEREGISTERED state, except if the HELLO\r message is received. An HELLO\r message transitions the server into the REGISTERED state.

REGISTERED: HELLO\r messages received when the server is in this state are rejected with message ABORT\r and transitions the server into the DEREGISTERED state. A BYE\r message transitions the server into the DEREGISTERED state.

In all states, unrecognised messages send back an ABORT\r and the server transitions in the DEREGISTERED state, if not there already.

This module has the following internal dependencies:

- On eu.dariolucia.reatmetric.api
- On eu.dariolucia.reatmetric.core

This module has two external dependencies:

- On JAXB library, since the configuration is defined in XML files.
- On jSerialComm library, used to handle the serial ports.

Protocol

Registration

client > server

```
HELLO <name>\r
```

server > client

```
HI RTM\r
```

Deregistration

client > server

```
BYE\r
```

server > client

```
CYA\r
```

Keep-Alive

The server expects to receive a message from the connected and registered client every X seconds. If no message is received, the server assumes that the client is disconnected. This is a server-side feature: compatible servers must be able to reply to a PING, but they might not have a connection timeout implemented.

client > server

```
PING\r
```

server > client

```
PONG\r
```

Set length of value strings in parameter update

client > server

```
SET_VALUE_LEN <length of value strings>\r
```

server > client (if OK)

```
OK\r
```

server > client (if not OK, i.e. wrong number/wrong value)

```
KO\r
```

Register monitoring parameter

client > server

```
REG_PARAM <param name>\r
```

server > client (if OK)

```
OK <param #, 2 digits number>\r
```

server > client (if NOK, i.e. parameter not found)

```
KO\r
```

The protocol allows a maximum of 99 parameters to be registered at a given point in time.

Deregister monitoring parameter

client > server

```
DEREG_PARAM <param #, 2 digits number>\r
```

server > client (if OK)

```
OK\r
```

server > client (if NOK, i.e. parameter number not existing)

```
KO\r
```

Deregister all monitoring parameters

client > server

```
DEREG_PARAM_ALL\r
```

server > client

```
OK\r
```

Request update of registered parameters

client > server

```
UPDATE_PARAM\r
```

server > client

```
<# records, 2 digits number>\r
# records lines, each formatted as:
<param #, 2 digits> <hh:mm:ss> <value as string> <V|I|E|U|D>
<ALM|WRN|VIO|NOM|N/A|N/C|ERR|UNK|IGN>\r
OK\r
```

If no parameters are registered, the answer will be:

```
00\r
OK\r
```

Example (assuming that the value length is set to 10)

```
04\r
01 08:12:33   56.42234 V NOM
   02 08:12:32   2256.422 V WRN
   03 08:11:58 Testing ac V NOM
   04 08:12:08          0 I N/A
```

```
   OK\r
```

Note that the full length of the variable block (the list of parameters) can be derived as soon as the number of entries is known.

Set max number of log messages in log update

client > server

```
SET_MAX_LOG <# max log events, 2 digits number>\r
```

server > client (if OK)

```
OK\r
```

server > client (if NOK, i.e. number too large)

```
KO\r
```

The protocol allows a maximum number of 99 log messages to be delivered in a single go.

Set length of message strings in log update

client > server

```
SET_LOG_LEN <length of message strings, 2 digits number>\r
```

server > client (if OK)

```
OK\r
```

Request update of log updates

client > server

```
UPDATE_LOG\r
```

server > client

```
<# records, 2 digits number>\r
# records lines, each formatted as:
<hh:mm:ss> <ALM|WRN|INF|ERR|UNK> <message>\r
```

```
OK\r
```

If no new logs are raised from the previous call, the answer will be:

```
00\r
OK\r
```

Note that the full length of the variable part becomes known as soon as the number of entries is known.

Configuration

Being a driver, the *serial* module must be registered as such in the system configuration file. You need to have a serial module registration for every terminal that you need to send data to via a serial port.

```
<ns1:core xmlns:ns1="http://dariolucia.eu/reatmetric/core/configuration">
    <name>Test System</name>
    <log-property-file>$HOME\Reatmetric\reatmetric_test\log.properties</log-property-
file>
    <definitions-location>$HOME\Reatmetric\reatmetric_test\processing</definitions-
location>
    <driver name="Serial Driver" type=
"eu.dariolucia.reatmetric.driver.serial.SerialDriver"
    configuration="$HOME\Reatmetric\reatmetric_test\serial"/>
</ns1:core>
```

The folder specified in the *configuration* attribute of the *driver* element must contain a file named *configuration.xml*, which defines the configuration properties of the driver.

The configuration structure of the eu.dariolucia.reatmetric.driver.serial module is defined in the package eu.dariolucia.reatmetric.driver.serial.definition. It is an XML file named *configuration.xml* using namespace definition *http://dariolucia.eu/reatmetric/driver/serial*.

An example of such file is presented below.

```
<ns1:serial xmlns:ns1="http://dariolucia.eu/reatmetric/driver/serial"
    device="COM3"
    timeout="10"
    baudrate="4800"
    parity="EVEN"
    data-bits="7"
    stop-bits="ONE"
    flow-control="NONE">
</ns1:serial>
```

device is the name of the serial port (e.g. "COM3" in Windows).

timeout is the serial port reading timeout in seconds.

baudrate is the speed of the serial port.

parity is the serial parity: it can be "EVEN", "ODD", "NO", "MARK" or "SPACE".

data-bits is the number of data bits: typically 7 or 8.

stop-bits is the use of stop bits: it can be "ONE", "ONEDOTFIVE", "TWO".

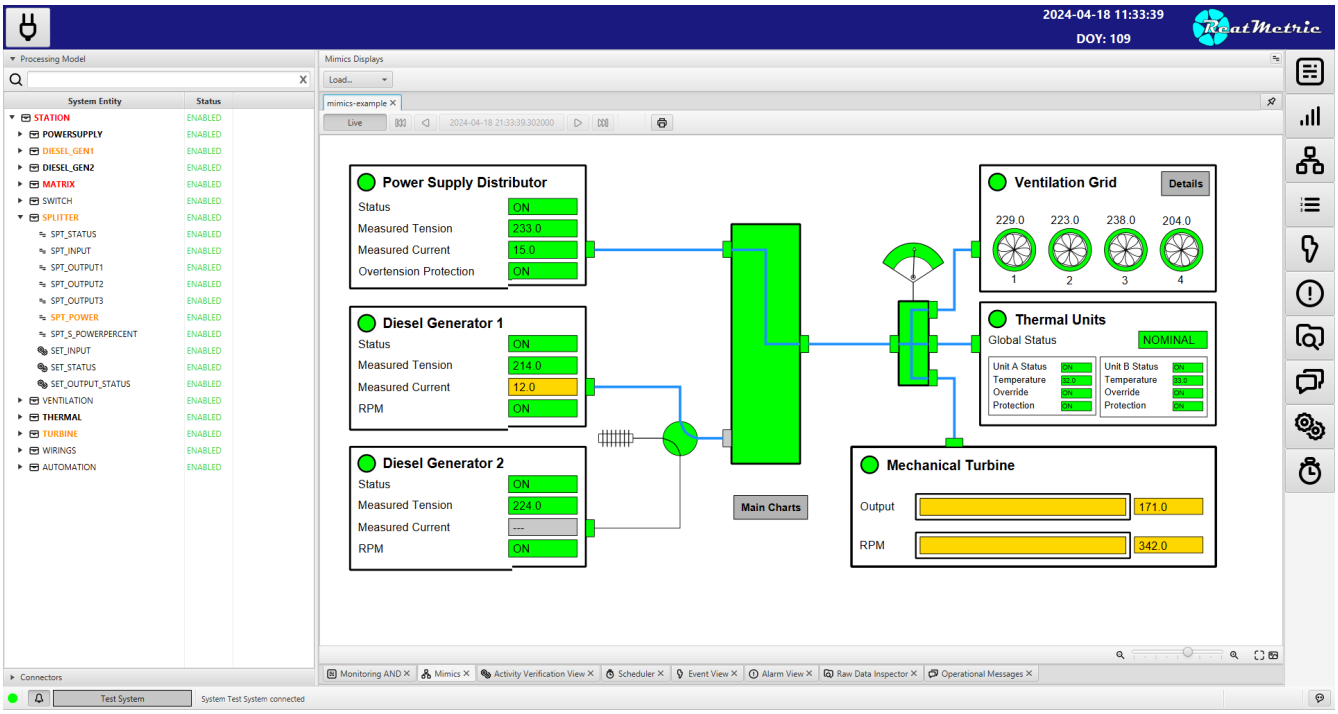
flow-control is the type of flow control used by the serial port: it can be "NONE", "CTS", "RTS_CTS", "DSR", "DTR_DSR", "XON_XOFF".

User Interface

Overview

The eu.dariolucia.reatmetric.ui module provides an implementation of a graphical User Interface (UI) to operate a ReatMetric system. In terms of technology, it is primarily based on JavaFX.

Depending on the deployment configuration, the ReatMetric UI can be deployed with an embedded ReatMetric backend, or it can be used to remotely connect to a backend running on a separate host, via a ReatMetric Remoting Connector.



The UI is very intuitive to use and it is composed by five main areas.

- The top level bar, containing the button to connect to and disconnect from configured ReatMetric systems, the system clock and the information about the UI version.
- The right panel, containing the buttons to open the various displays in the main display area of the UI.
- The left panel, which contains two areas:
  - The hierarchical view of the processing model.
  - The list of connectors registered in the system.
- The bottom toolbar, which contains:
  - An indicator (green, orange, red) to display the initialisation status of the back-end and, by pressing on it, debug information of the back-end processing status.
  - An indicator with the name of the connected system, which blinks in case of outstanding alarms. By pressing on it, a display shows the outstanding alarms, and provides a way to selectively acknowledge them.
  - A system message label and a progress bar, for long running operations.

    - A button to open the chat window.
- The main display area, where the different displays are opened upon click on the right panel button.

ReatMetric UI provides several displays to visualise specific aspects of the connected ReatMetric system. All displays can work in live as well as in historical mode and are detachable. The displays are:

- AND Displays, showing a set of parameters, partitioned in groups, with related value, alarm state, validity.
- Mimics Displays, showing the rendering of real-time updated SVG-based mimics.
- Chart/Plot Displays, showing line, area, bar and scatter charts (for events).
- Parameter Log Display, showing all (or filtered) parameter samples as processed by the processing model.
- Alarm Log Display, showing all (or filtered) alarms as generated by the processing model.
- Event Log Display, showing all (or filtered) events as raised by the processing model.
- Raw Data Display, showing all (or filtered) raw data distributed in the system by the raw data broker.
- Message Display, showing all (or filtered) messages distributed in the system by the operational message broker.
- Activity Display, showing all activities and related verification stages as managed by the processing model.
- Schedule Display, showing all scheduled activities and bots as managed by the scheduler.

This module has the following dependencies:

- On eu.dariolucia.reatmetric.api

This module has the following external dependencies:

- On JavaFX libraries, as main UI library.
- On ControlsFX, for add-ons and enhancements built on top of JavaFX.
- On json-path library, since the storage of AND and chart displays is in JSON format.
- On eu.dariolucia.jfx.timeline, for the timeline chart rendering of the Scheduler Display View.

Start-up

It is assumed that all depending modules are located inside the "deps" folder in the current working directory, from where the ReatMetric UI is started.

In case of ReatMetric system embedded in the UI, the command line to start the UI are the followings:

(Windows)

```
    java
    --module-path="deps"
    -Dreatmetric.core.config=<path to ReatMetric>\configuration.xml
    --add-exports javafx.base/com.sun.javafx.event=org.controlsfx.controls
    -m eu.dariolucia.reatmetric.ui/eu.dariolucia.reatmetric.ui.ReatmetricUI
```

(Linux)

```
    java
    --module-path="deps"
    -Dreatmetric.core.config=<path to ReatMetric>/configuration.xml
    --add-exports javafx.base/com.sun.javafx.event=org.controlsfx.controls
    -m eu.dariolucia.reatmetric.ui/eu.dariolucia.reatmetric.ui.ReatmetricUI
```

In case of ReatMetric system running on a separate host, the command line to start the UI are the followings:

(Windows)

```
    java
    --module-path="deps"
    -Djava.rmi.server.hostname=<server IP to use for local connections>
    -Dreatmetric.remoting.connector.config=<path to ReatMetric
remoting>\configuration.xml
    --add-exports javafx.base/com.sun.javafx.event=org.controlsfx.controls
    -m eu.dariolucia.reatmetric.ui/eu.dariolucia.reatmetric.ui.ReatmetricUI
```

(Linux)

```
    java
    --module-path="deps"
    -Djava.rmi.server.hostname=<server IP to use for local connections>
    -Dreatmetric.remoting.connector.config=<path to ReatMetric
remoting>/configuration.xml
    --add-exports javafx.base/com.sun.javafx.event=org.controlsfx.controls
    -m eu.dariolucia.reatmetric.ui/eu.dariolucia.reatmetric.ui.ReatmetricUI
```

In case the CSS stylesheet requires updates for UI customisation, the UI application must be started with the following VM option:

```
    -Dreatmetric.css.default.theme=<path to CSS file>
```

Two examples of such file (which can be copied and modified) can be found inside the reatmetric.ui module, src/main/resources/eu/dariolucia/reatmetric/ui/fxml/css. If no property is specified, the UI application will start with the reatmetric_clear.css theme by default, which in turns uses the

standard OpenJFX CSS file Modena.css. The latest version of such file can be found by following this link: https://github.com/openjdk/jfx/blob/master/modules/javafx.controls/src/main/resources/com/sun/javafx/scene/control/skin/modena/modena.css

The file contains a lot of explanations. To make a customisation, it is enough to overwrite the desired property in the custom CSS file. Example: the reatmetric_dark style overwrites the property -fx-base of the class .root. This property is used by the Modena style to derive all the gradients of background automatically.

The reference guide for OpenJFX styling is provided here: https://openjfx.io/javadoc/19/javafx.graphics/javafx/scene/doc-files/cssref.html

Configuration

The UI does not require specific configuration. During its execution, it stores specific user preferences inside the folder $HOME/Reatmetric UI. Artefacts such as ANDs and charts/plots can be created directly via the UI. The only artefacts that require external preparation are the mimics. ReatMetric mimics are built by means of SVG files. In order to link the defined SVG elements to parameters, ReatMetric prescribes the use of custom data-* SVG attributes.

The attributes defined and/or used by ReatMetric, their meaning and the expected syntax value are listed in this section.

Condition-Expression

Unless differently specified, ReatMetric custom attributes contains 'condition-expression' values. A condition-expression value is a string, composed by (guess eh..) a condition and an expression.

```
condition-expression ::= [<condition>' ']':= '<expression>
```

A condition is a single boolean expression, which is composed by a reference, a boolean operator and a reference value. If it is omitted, then it is assumed that the condition is always met (i.e. is always evaluated to *true*).

```
<condition> ::= <reference>' '<operator>' '<reference value>'
```

A reference can be one of the followings:

```
<reference> ::= '$eng'|'$raw'|'$alarm'|'$validity'
```

$eng is the engineering value of the bound parameter $raw is the raw (source) value of the bound parameter $alarm is the global alarm state of the bound parameter $validity is the validity state of the bound parameter

```
<operator> ::= LT|GT|LTE|GTE|EQ|NQ
```

```
<reference value> ::= any string literal (parameter reference-type dependant)|'##NULL##'
```

Depending on the selected reference, ReatMetric can infer the correct type and can apply the correct comparison function to the derived values. To indicate the null value, the following reserved string must be used: *##NULL##*

```
<expression> ::= any string literal (SVG attribute dependant)
```

An expression exact value depends on the SVG attribute and the allowed values are described in this documentation. For instance, the *data-rtmt-visibility-00* attribute accepts as expression either the string *visible* or the string *hidden*. However, the ReatMetric framework allows to use $eng, $raw, $validity and $alarm also in expressions: the placeholder is replaced with the correct value if the transformation must be applied.

Some examples of condition-expression values are provided hereafter.

```
data-rtmt-fill-color-00="$alarm EQ WARNING := #AA3344FF"
```

Explanation: if the alarm state is WARNING, set the fill color to #RGBA.

```
data-rtmt-visibility-00="$validity EQ INVALID := hidden"
```

Explanation: if the validity is INVALID, set the visibility to hidden (SVG element is not displayed).

```
data-rtmt-text-00="$validity EQ VALID := $eng"
```

Explanation: if the validity is VALID, set the text of the SVG element to the engineering value of the parameter.

Attributes

data-rtmt-binding-id

This attribute contains the path of the parameter that is bound to the SVG element. This attribute is mandatory.

```
Example: data-rtmt-binding-id="ROOT.ELEMENT.PARAM1"
```

data-rtmt-visibility-[nn]

This attribute is used to set the visibility of the SVG element. Its value is defined by a ReatMetric condition-expression. This attribute can be present several times attached to a single SVG element. If so, such attribute list is evaluated in lexicographical order. As soon as one item's evaluation is successful, the list evaluation stops.

Allowed expression values: *collapse*, *hidden* or *visible*. Null value *##NULL##* removes the attribute.

data-rtmt-fill-color-[nn]

This attribute is used to set the fill color of the SVG element. Its value is defined by a ReatMetric condition-expression. This attribute can be present several times attached to a single SVG element. If so, such attribute list is evaluated in lexicographical order. As soon as one item's evaluation is successful, the list evaluation stops.

Allowed expression values: #RRGGBBAA. Null value *##NULL##* removes the attribute.

data-rtmt-stroke-color-[nn]

This attribute is used to set the stroke color of the SVG element. Its value is defined by a ReatMetric

condition-expression. This attribute can be present several times attached to a single SVG element. If so, such attribute list is evaluated in lexicographical order. As soon as one item's evaluation is successful, the list evaluation stops.

Allowed expression values: #RRGGBBAA. Null value *##NULL##* removes the attribute.

data-rtmt-stroke-width-[nn]

This attribute is used to set the stroke width of the SVG element. Its value is defined by a ReatMetric condition-expression. This attribute can be present several times attached to a single SVG element. If so, such attribute list is evaluated in lexicographical order. As soon as one item's evaluation is successful, the list evaluation stops.

Allowed expression values: a real number. Null value *##NULL##* removes the attribute.

data-rtmt-width-[nn]

This attribute is used to set the width of the SVG element. Its value is defined by a ReatMetric condition-expression. This attribute can be present several times attached to a single SVG element. If so, such attribute list is evaluated in lexicographical order. As soon as one item's evaluation is successful, the list evaluation stops.

Allowed expression values: a real number. Null value *##NULL##* removes the attribute.

data-rtmt-height-[nn]

This attribute is used to set the height of the SVG element. Its value is defined by a ReatMetric condition-expression. This attribute can be present several times attached to a single SVG element. If so, such attribute list is evaluated in lexicographical order. As soon as one item's evaluation is successful, the list evaluation stops.

Allowed expression values: a real number. Null value *##NULL##* removes the attribute.

data-rtmt-text-[nn]

This attribute is used to set the text of the SVG element. Its value is defined by a ReatMetric condition-expression. It can be attached only to SVG <*text*> elements. <*text*> elements shall not contain any <*tspan*> element, as it is not supported. This attribute can be present several times attached to a single SVG element. If so, such attribute list is evaluated in lexicographical order. As soon as one item's evaluation is successful, the list evaluation stops.

Allowed expression values: any string. Null value *##NULL##* is treated as empty string.

data-rtmt-transform-[nn]

This attribute is used to set the transformation of the SVG element. Its value is defined by a ReatMetric condition-expression. This attribute can be present several times attached to a single SVG element. If so, such attribute list is evaluated in lexicographical order. As soon as one item's evaluation is successful, the list evaluation stops.

Allowed expression values: string, syntax as per https://developer.mozilla.org/en-US/docs/Web/SVG/

Attribute/transform. Null value *##NULL##* removes the attribute.

data-rtmt-blink-[nn]

This attribute is used to set whether an SVG object shall blink. Its value is defined by a ReatMetric condition-expression. This attribute can be present several times attached to a single SVG element. If so, such attribute list is evaluated in lexicographical order. As soon as one item's evaluation is successful, the list evaluation stops.

Allowed expression values: #RRGGBBAA or *none* to disable blinking.

If set to a colour, the fill attribute value is taken from the specified color and the tone is decreased by half. The *animate* tag attached to the SVG element is (example):

```
<animate attributeType="XML" attributeName="fill" values="#800;#f00;#800;#800" dur="1.0s" repeatCount="indefinite"/>
```

data-rtmt-rotate-[nn]

This attribute is used to set whether an SVG object shall rotate. Its value is defined by a ReatMetric condition-expression. This attribute can be present several times attached to a single SVG element. If so, such attribute list is evaluated in lexicographical order. As soon as one item's evaluation is successful, the list evaluation stops.

Allowed expression values: `<rotation time in milliseconds> <rotation center x> <rotation center y>` or `none` to disable the rotation.

Actions

Actions can be linked to SVG elements by means of the `onclick` attribute. In addition to the standard javascript functions, ReatMetric mimics can use the following functions. In order to indicate the presence of a clickable action, it is suggested to also use the attribute `cursor=pointer`.

reatmetric.and('name')

Open the specified AND.

reatmetric.chart('name')

Open the specified chart.

reatmetric.mimics('name')

Open the specified mimics.

reatmetric.exec('path')

Open the dialog window to request the execution of the specified activity.

reatmetric.set('path')

Open the dialog window to request the set of the specified parameter.

Extending ReatMetric

Implement a new driver

This part of the documentation provides a step-by-step guide on how to create, implement and deploy a new ReatMetric driver from scratch. This guide will assume adequate knowledge of Java and Maven.

The example driver presented here is very simple:

- it implements a connector, that can be started and stopped.
- When the connector is started, the driver publishes a single parameter value every second to the ReatMetric processing model, increasing its value by one;
- it raises one event when the counter is a multiple of 10;
- it handles an activity that resets the counter to 0.

Step 1: Create a new Maven project

Create an empty folder and put inside this folder the pom file and the necessary Maven folders. As a minimum, the folder src/main/java needs to exist. The pom file must contain as a minimum the dependencies to the API and Core modules.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>eu.dariolucia.reatmetric.example</groupId>
    <artifactId>eu.dariolucia.reatmetric.driver.example</artifactId>
    <name>REATMETRIC DRIVER - Example driver</name>
    <version>1.0.0</version>
    <description>REATMETRIC driver module for documentation purposes</description>
    <packaging>jar</packaging>

    <properties>
        <!-- Encoding -->
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    </properties>

    <build>
        <plugins>
            <!-- Maven version to use -->
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>versions-maven-plugin</artifactId>
                <version>2.5</version>
```

```
                <configuration>
                    <generateBackupPoms>false</generateBackupPoms>
                </configuration>
            </plugin>
            <!-- Set the source file version to Java 11-->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.11.0</version>
                <configuration>
                    <release>11</release>
                </configuration>
            </plugin>
        </plugins>
    </build>

    <dependencies>
        <dependency>
            <groupId>eu.dariolucia.reatmetric</groupId>
            <artifactId>eu.dariolucia.reatmetric.api</artifactId>
            <version>1.0.0</version>
        </dependency>
        <dependency>
            <groupId>eu.dariolucia.reatmetric</groupId>
            <artifactId>eu.dariolucia.reatmetric.core</artifactId>
            <version>1.0.0</version>
        </dependency>
        <dependency>
            <groupId>eu.dariolucia.reatmetric</groupId>
            <artifactId>eu.dariolucia.reatmetric.processing</artifactId>
            <version>1.0.0</version>
        </dependency>
    </dependencies>
</project>
```

By running:

```
mvn clean install
```

the project should build without errors.

Step 2: Create the driver module file and the class entrypoint

Create the file module-info.java with the information about the module dependencies. As a minimum, the following dependencies shall be listed.

```
open module eu.dariolucia.reatmetric.driver.example {
    requires java.rmi;
```

```
    requires eu.dariolucia.reatmetric.api;
    requires eu.dariolucia.reatmetric.core;
}
```

Create a class named ExampleDriver inside package eu.dariolucia.reatmetric.driver.example, extending from AbstractDriver, as shown below (imports are omitted).

```
public class ExampleDriver extends AbstractDriver {

    public List<DebugInformation> currentDebugInfo() {
        return Collections.emptyList();
    }

    protected SystemStatus startProcessing() throws DriverException {
        return SystemStatus.NOMINAL;
    }

    protected SystemStatus processConfiguration(String driverConfiguration,
ServiceCoreConfiguration coreConfiguration, IServiceCoreContext context) throws
DriverException {
        return SystemStatus.NOMINAL;
    }
}
```

Update the module-info.java to export the driver implementation as Java service.

```
open module eu.dariolucia.reatmetric.driver.example {
    requires java.rmi;
    requires eu.dariolucia.reatmetric.api;
    requires eu.dariolucia.reatmetric.core;
    exports eu.dariolucia.reatmetric.driver.example;
    provides eu.dariolucia.reatmetric.core.api.IDriver with eu.dariolucia.reatmetric
.driver.example.ExampleDriver;
}
```

Step 3: Initialise the driver from the configuration

The driver needs to know which are the IDs of the related parameter and event objects in the processing model, as well as to know the ID of the supported activity. The approach used for this example is very simple: the configuration string is the path of the parent system element and the child system elements are expected to be present there. The IDs of such element are then retrieved during the starting phase.

```
public class ExampleDriver extends AbstractDriver {

    private final String PARAMETER_NAME = "Example-Parameter";
    private final String EVENT_NAME = "Example-Event";
```

```
    private final String ACTIVITY_NAME = "Example-Activity";

    private SystemEntityPath parentSystemElement;

    private int parameterId;
    private int eventId;
    private int activityId;

    public List<DebugInformation> currentDebugInfo() {
        return Collections.emptyList();
    }

    protected SystemStatus startProcessing() throws DriverException {
        // Resolve the paths into IDs
        try {
            this.parameterId = getContext().getProcessingModel().getDescriptorOf(this
.parentSystemElement.append(PARAMETER_NAME)).getExternalId();
            this.eventId = getContext().getProcessingModel().getDescriptorOf(this
.parentSystemElement.append(PARAMETER_NAME)).getExternalId();
            this.activityId = getContext().getProcessingModel().getDescriptorOf(this
.parentSystemElement.append(PARAMETER_NAME)).getExternalId();
        } catch (ReatmetricException e) {
            throw new DriverException(e);
        }
        return SystemStatus.NOMINAL;
    }

    protected SystemStatus processConfiguration(String driverConfiguration,
ServiceCoreConfiguration coreConfiguration, IServiceCoreContext context) throws
DriverException {
        this.parentSystemElement = SystemEntityPath.fromString(driverConfiguration);
        return SystemStatus.NOMINAL;
    }
}
```

Step 4: Add a connector, implement data injection in the processing model

Create a class in the package eu.dariolucia.reatmetric.driver.example, named ExampleConnector and extending from the AbstractConnector class, as shown below. This class is used to control the start and stop of the parameter publication.

```
public class ExampleConnector extends AbstractTransportConnector {

    private final ExampleDriver driver;
    private Thread countingThread;
    private volatile boolean started = false;
    private final AtomicLong counter = new AtomicLong(0);

    public ExampleConnector(String name, String description, ExampleDriver driver) {
```

```
        super(name, description);
        this.driver = driver;
    }

    @Override
    protected Pair<Long, Long> computeBitrate() {
        return null; // No TX,RX data rate computed
    }

    @Override
    protected synchronized void doConnect() throws TransportException {
        // If the counting thread is not started, start the thread
        if(this.countingThread == null) {
            updateAlarmState(AlarmState.NOT_APPLICABLE);
            updateConnectionStatus(TransportConnectionStatus.CONNECTING);
            this.started = true;
            this.countingThread = new Thread(this::countingLoop);
            this.countingThread.setDaemon(true);
            this.countingThread.start();
        }
    }

    private void countingLoop() {
        updateConnectionStatus(TransportConnectionStatus.OPEN);
        while(started) {
            long toDistribute = this.counter.getAndIncrement();
            this.driver.newValue(toDistribute);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // No action needed here
            }
        }
        updateConnectionStatus(TransportConnectionStatus.IDLE);
    }

    @Override
    protected synchronized void doDisconnect() throws TransportException {
        // If the counting thread is started, stop the thread
        if(this.countingThread != null) {
            updateConnectionStatus(TransportConnectionStatus.DISCONNECTING);
            this.started = false;
            this.countingThread.interrupt();
            try {
                this.countingThread.join();
            } catch (InterruptedException e) {
                // Nothing to be done here
            }
            this.countingThread = null;
        }
    }
```

```
    @Override
    protected void doDispose() {
        // Nothing to be done here
    }

    @Override
    public void abort() throws TransportException, RemoteException {
        disconnect();
    }
}
```

The ExampleDriver class must now be extended to:

- Create the connector in the starting phase;
- Return the connector as supported connector;
- Implement the newValue(long) method.

```
public class ExampleDriver extends AbstractDriver {

    private final String PARAMETER_NAME = "Example-Parameter";
    private final String EVENT_NAME = "Example-Event";
    private final String ACTIVITY_NAME = "Example-Activity";

    private SystemEntityPath parentSystemElement;

    private int parameterId;
    private int eventId;
    private int activityId;

    private ExampleConnector connector;

    public List<DebugInformation> currentDebugInfo() {
        return Collections.emptyList();
    }

    protected SystemStatus startProcessing() throws DriverException {
        // Resolve the paths into IDs
        try {
            this.parameterId = getContext().getProcessingModel().getDescriptorOf(this
.parentSystemElement.append(PARAMETER_NAME)).getExternalId();
            this.eventId = getContext().getProcessingModel().getDescriptorOf(this
.parentSystemElement.append(PARAMETER_NAME)).getExternalId();
            this.activityId = getContext().getProcessingModel().getDescriptorOf(this
.parentSystemElement.append(PARAMETER_NAME)).getExternalId();
        } catch (ReatmetricException e) {
            throw new DriverException(e);
        }
        // Create the connector
```

```
        this.connector = new ExampleConnector("Example Connector", "Example Connector
Description", this);
        // The connector prepare() must be called before being able to use it
        this.connector.prepare();
        // If we are here, all fine
        return SystemStatus.NOMINAL;
    }

    protected SystemStatus processConfiguration(String driverConfiguration,
ServiceCoreConfiguration coreConfiguration, IServiceCoreContext context) throws
DriverException {
        this.parentSystemElement = SystemEntityPath.fromString(driverConfiguration);
        return SystemStatus.NOMINAL;
    }

    @Override
    public List<ITransportConnector> getTransportConnectors() {
        return Collections.singletonList(this.connector);
    }

    public void newValue(long toDistribute) {
        // Parameter injection
        ParameterSample sample = ParameterSample.of(this.parameterId, toDistribute);
        getContext().getProcessingModel().injectParameters(Collections.singletonList
(sample));
        // Event injection
        if(toDistribute % 10 == 0) {
            EventOccurrence event = EventOccurrence.of(this.eventId);
            getContext().getProcessingModel().raiseEvent(event);
        }
    }
}
```

Step 5: Add an activity handler

Create a class in the package eu.dariolucia.reatmetric.driver.example, named ExampleHandler and implementing the IActivityHandler interface, as shown below. This class is used to manage the requests of activity executions.

```
public class ExampleHandler implements IActivityHandler {

    private final ExampleDriver driver;

    public ExampleHandler(ExampleDriver driver) {
        this.driver = driver;
    }

    @Override
    public void registerModel(IProcessingModel model) {
```

```
        // Not needed
    }

    @Override
    public void deregisterModel(IProcessingModel model) {
        // Not needed
    }

    @Override
    public List<String> getSupportedRoutes() {
        return Collections.singletonList(ExampleDriver.ROUTE_NAME);
    }

    @Override
    public List<String> getSupportedActivityTypes() {
        return Collections.singletonList(ExampleDriver.ACTIVITY_TYPE);
    }

    @Override
    public void executeActivity(ActivityInvocation activityInvocation) throws
ActivityHandlingException {
        // Check if the connector is active
        if(!driver.isConnectorStarted()) {
            throw new ActivityHandlingException("Connector not started");
        }
        // Check if the route is OK
        if(!activityInvocation.getRoute().equals(ExampleDriver.ROUTE_NAME)) {
            throw new ActivityHandlingException("Route mismatch");
        }
        // Check if the activity is the one you expect (ID and path are matching)
        if(!driver.isActivitySupported(activityInvocation.getPath(),
activityInvocation.getActivityId())) {
            throw new ActivityHandlingException("ID/Path mismatch");
        }
        // If so, inform that the RELEASE is done and invoke the request
asynchronously to the connector
        // (a service executor would help, but this is an example)
        new Thread(() -> {
            driver.executeCounterReset(activityInvocation);
        }).start();
    }

    @Override
    public boolean getRouteAvailability(String route) throws ActivityHandlingException
{
        return route.equals(ExampleDriver.ROUTE_NAME) && driver.isConnectorStarted();
    }

    @Override
    public void abortActivity(int activityId, IUniqueId activityOccurrenceId) throws
ActivityHandlingException {
```

```
        // Not supported for this driver
        throw new ActivityHandlingException("Operation not supported");
    }
}
```

The ExampleDriver class must now be extended to:

- Create the activity handler in the starting phase;
- Return the activity handler as supported connector;
- Implement the necessary methods to implement the activity occurrence lifecycle.

```
public class ExampleDriver extends AbstractDriver {

    public static final String ROUTE_NAME = "Example Route";
    public static final String ACTIVITY_TYPE = "Example Activity Type";
    public static final String RESET_EXECUTION_NAME = "Reset Execution";
    private final String PARAMETER_NAME = "Example-Parameter";
    private final String EVENT_NAME = "Example-Event";
    private final String ACTIVITY_NAME = "Example-Activity";

    private SystemEntityPath parentSystemElement;

    private int parameterId;
    private int eventId;
    private int activityId;

    private ExampleConnector connector;
    private ExampleHandler handler;

    public List<DebugInformation> currentDebugInfo() {
        return Collections.emptyList();
    }

    protected SystemStatus startProcessing() throws DriverException {
        // Resolve the paths into IDs
        try {
            this.parameterId = getContext().getProcessingModel().getDescriptorOf(this
.parentSystemElement.append(PARAMETER_NAME)).getExternalId();
            this.eventId = getContext().getProcessingModel().getDescriptorOf(this
.parentSystemElement.append(EVENT_NAME)).getExternalId();
            this.activityId = getContext().getProcessingModel().getDescriptorOf(this
.parentSystemElement.append(ACTIVITY_NAME)).getExternalId();
        } catch (ReatmetricException e) {
            throw new DriverException(e);
        }
        // Create the connector
        this.connector = new ExampleConnector("Example Connector", "Example Connector
Description", this);
        // The connector prepare() must be called before being able to use it
```

```
        this.connector.prepare();
        // Create the activity handler
        this.handler = new ExampleHandler(this);
        // If we are here, all fine
        return SystemStatus.NOMINAL;
    }

    protected SystemStatus processConfiguration(String driverConfiguration,
ServiceCoreConfiguration coreConfiguration, IServiceCoreContext context) throws
DriverException {
        this.parentSystemElement = SystemEntityPath.fromString(driverConfiguration);
        return SystemStatus.NOMINAL;
    }

    @Override
    public List<IActivityHandler> getActivityHandlers() {
        return Collections.singletonList(this.handler);
    }

    @Override
    public List<ITransportConnector> getTransportConnectors() {
        return Collections.singletonList(this.connector);
    }

    public void newValue(long toDistribute) {
        // Parameter injection
        ParameterSample sample = ParameterSample.of(this.parameterId, toDistribute);
        getContext().getProcessingModel().injectParameters(Collections.singletonList
(sample));
        // Event injection
        if(toDistribute % 10 == 0) {
            EventOccurrence event = EventOccurrence.of(this.eventId);
            getContext().getProcessingModel().raiseEvent(event);
        }
    }

    public boolean isConnectorStarted() {
        return this.connector.getConnectionStatus() == TransportConnectionStatus.OPEN;
    }

    public boolean isActivitySupported(SystemEntityPath path, int requestedActivity) {
        return path.equals(this.parentSystemElement.append(ACTIVITY_NAME)) &&
requestedActivity == this.activityId;
    }

    public void executeCounterReset(IActivityHandler.ActivityInvocation
activityInvocation) {
        // Informing that we are proceeding with the release of the activity
occurrence, and that, if it works, we go
        // directly in the EXECUTION state
        reportActivityState(activityInvocation.getActivityId(), activityInvocation
```

```
.getActivityOccurrenceId(), Instant.now(),
                ActivityOccurrenceState.RELEASE, ActivityOccurrenceReport
.RELEASE_REPORT_NAME, ActivityReportState.PENDING,
                ActivityOccurrenceState.EXECUTION);
        if(!isConnectorStarted()) {
            // Connector not started, release failed
            reportActivityState(activityInvocation.getActivityId(),
activityInvocation.getActivityOccurrenceId(), Instant.now(),
                    ActivityOccurrenceState.RELEASE, ActivityOccurrenceReport
.RELEASE_REPORT_NAME, ActivityReportState.FATAL,
                    ActivityOccurrenceState.RELEASE);
            // That's it
            return;
        } else {
            // Connector started, release OK, pending execution
            reportActivityState(activityInvocation.getActivityId(),
activityInvocation.getActivityOccurrenceId(), Instant.now(),
                    ActivityOccurrenceState.RELEASE, ActivityOccurrenceReport
.RELEASE_REPORT_NAME, ActivityReportState.OK,
                    ActivityOccurrenceState.EXECUTION);
            reportActivityState(activityInvocation.getActivityId(),
activityInvocation.getActivityOccurrenceId(), Instant.now(),
                    ActivityOccurrenceState.EXECUTION, RESET_EXECUTION_NAME,
ActivityReportState.PENDING,
                    ActivityOccurrenceState.VERIFICATION);
        }
        // Execution of the activity
        boolean resetCounter = this.connector.resetCounter();
        if(resetCounter) {
            // Good, activity finished OK
            reportActivityState(activityInvocation.getActivityId(),
activityInvocation.getActivityOccurrenceId(), Instant.now(),
                    ActivityOccurrenceState.EXECUTION, RESET_EXECUTION_NAME,
ActivityReportState.OK,
                    ActivityOccurrenceState.VERIFICATION);
        } else {
            // Bad, activity finished with error
            reportActivityState(activityInvocation.getActivityId(),
activityInvocation.getActivityOccurrenceId(), Instant.now(),
                    ActivityOccurrenceState.EXECUTION, RESET_EXECUTION_NAME,
ActivityReportState.FATAL,
                    ActivityOccurrenceState.EXECUTION);
        }
    }
}
```

Finally, the ExampleConnector must be extended to implement the method resetCounter, which is trivial.

```
public class ExampleConnector extends AbstractTransportConnector {
```

```
    private final ExampleDriver driver;
    private Thread countingThread;
    private volatile boolean started = false;
    private final AtomicLong counter = new AtomicLong(0);

    public ExampleConnector(String name, String description, ExampleDriver driver) {
        super(name, description);
        this.driver = driver;
    }

    @Override
    protected Pair<Long, Long> computeBitrate() {
        return null; // No TX,RX data rate computed
    }

    @Override
    protected synchronized void doConnect() throws TransportException {
        // If the counting thread is not started, start the thread
        if(this.countingThread == null) {
            updateAlarmState(AlarmState.NOT_APPLICABLE);
            updateConnectionStatus(TransportConnectionStatus.CONNECTING);
            this.started = true;
            this.countingThread = new Thread(this::countingLoop);
            this.countingThread.setDaemon(true);
            this.countingThread.start();
        }
    }

    private void countingLoop() {
        updateConnectionStatus(TransportConnectionStatus.OPEN);
        while(started) {
            long toDistribute = this.counter.getAndIncrement();
            this.driver.newValue(toDistribute);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // No action needed here
            }
        }
        updateConnectionStatus(TransportConnectionStatus.IDLE);
    }

    @Override
    protected synchronized void doDisconnect() throws TransportException {
        // If the counting thread is started, stop the thread
        if(this.countingThread != null) {
            updateConnectionStatus(TransportConnectionStatus.DISCONNECTING);
            this.started = false;
            this.countingThread.interrupt();
            try {
```

```
                this.countingThread.join();
            } catch (InterruptedException e) {
                // Nothing to be done here
            }
            this.countingThread = null;
        }
    }

    @Override
    protected void doDispose() {
        // Nothing to be done here
    }

    @Override
    public void abort() throws TransportException, RemoteException {
        disconnect();
    }

    public boolean resetCounter() {
        if(getConnectionStatus() != TransportConnectionStatus.OPEN) {
            return false;
        }
        this.counter.set(0);
        return true;
    }
}
```

Step 6: Prepare the processing model definition

Create an XML file with the following content:

```
<ns1:processing xmlns:ns1="http://dariolucia.eu/reatmetric/processing/definition">
    <parameters>
        <!-- Counter -->
        <parameter id="#100" location="EXAMPLE.SYSTEM.DRIVER.Example-Parameter"
                   description="Example parameter - counter"
                   raw_type="SIGNED_INTEGER" eng_type="SIGNED_INTEGER" eng_unit="" />
    </parameters>
    <events>
        <!-- Event for % 10 condition -->
        <event id="#101" location="EXAMPLE.SYSTEM.DRIVER.Example-Event" description=
"Example event"
               severity="INFO" type="EXAMPLE_EVENT_TYPE"/>
    </events>
    <activities>
        <!-- Activity for counter reset -->
        <activity id="#102" location="EXAMPLE.SYSTEM.DRIVER.Example-Activity"
                  description="Example activity - reset counter" type="Example
Activity Type"
                  verification_timeout="5" >
```

```
            <verification>
                <!-- Check that the value of the counter goes to 0 -->
                <expression>COUNTER == 0</expression>
                <symbol name="COUNTER" reference="#100" binding="ENG_VALUE" />
            </verification>
        </activity>
    </activities>
</ns1:processing>
```

It is possible to recognise the three system entities (parameter, event and activity), plus a special post-execution verification for the activity, which confirms the reset using the value provided by the parameter.

The driver is completed. The next section explains how to create a new ReatMetric deployment, and to configure it to use this driver.

The full driver source code can be found in the module eu.dariolucia.reatmetric.driver.example.

Generate a new deployment package

This part of the documentation provides a step-by-step guide on how to create a new ReatMetric deployment from scratch. This guide will assume adequate knowledge of Java and Maven.

The example deployment presented here is very simple, it generates a ReatMetric software deployment including:

- The example driver implemented in the previous section;
- The automation, HTTP driver and scheduler;
- The persist implementation based on Apache Derby;
- The UI module.

The deployment is assumed to be on Windows systems. In case it is done on Unix systems, obviously the Unix path separator should be used, i.e. / instead of \.

Step 1: Create a new Maven project

Create an empty folder and put inside this folder the pom file. The pom file must contain the dependencies that are required for the deployment.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>eu.dariolucia.reatmetric.example</groupId>
    <artifactId>eu.dariolucia.reatmetric.driver.example.test</artifactId>
    <name>REATMETRIC - Example Test</name>
    <version>1.0.0</version>
```

```
    <description>REATMETRIC Example Test for deployment</description>

    <properties>
        <!-- Encoding -->
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    </properties>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-dependency-plugin</artifactId>
                <executions>
                    <execution>
                        <id>copy-dependencies</id>
                        <phase>prepare-package</phase>
                        <goals>
                            <goal>copy-dependencies</goal>
                        </goals>
                        <configuration>
                            <outputDirectory>target/deps</outputDirectory>
                            <overWriteIfNewer>true</overWriteIfNewer>
                            <excludeGroupIds>

org.junit.jupiter,org.apiguardian,org.junit.platform,org.opentest4j
                            </excludeGroupIds>
                        </configuration>
                    </execution>
                </executions>
                <configuration>
                    <verbose>true</verbose>
                    <detail>true</detail>
                    <outputDirectory>${project.build.directory}</outputDirectory>
                </configuration>
            </plugin>
        </plugins>
    </build>

    <dependencies>
        <dependency>
            <groupId>eu.dariolucia.reatmetric</groupId>
            <artifactId>eu.dariolucia.reatmetric.api</artifactId>
            <version>1.0.0</version>
        </dependency>
        <dependency>
            <groupId>eu.dariolucia.reatmetric</groupId>
            <artifactId>eu.dariolucia.reatmetric.core</artifactId>
            <version>1.0.0</version>
        </dependency>
        <dependency>
```

```
            <groupId>eu.dariolucia.reatmetric</groupId>
            <artifactId>eu.dariolucia.reatmetric.scheduler</artifactId>
            <version>1.0.0</version>
        </dependency>
        <dependency>
            <groupId>eu.dariolucia.reatmetric</groupId>
            <artifactId>eu.dariolucia.reatmetric.ui</artifactId>
            <version>1.0.0</version>
        </dependency>
        <dependency>
            <groupId>eu.dariolucia.reatmetric</groupId>
            <artifactId>eu.dariolucia.reatmetric.driver.automation.groovy</artifactId>
            <version>1.0.0</version>
        </dependency>
        <dependency>
            <groupId>eu.dariolucia.reatmetric</groupId>
            <artifactId>eu.dariolucia.reatmetric.persist</artifactId>
            <version>1.0.0</version>
        </dependency>
        <dependency>
            <groupId>eu.dariolucia.reatmetric</groupId>
            <artifactId>eu.dariolucia.reatmetric.processing</artifactId>
            <version>1.0.0</version>
        </dependency>
        <dependency>
            <groupId>eu.dariolucia.reatmetric</groupId>
            <artifactId>eu.dariolucia.reatmetric.driver.httpserver</artifactId>
            <version>1.0.0</version>
        </dependency>
        <!-- The new driver -->
        <dependency>
            <groupId>eu.dariolucia.reatmetric.example</groupId>
            <artifactId>eu.dariolucia.reatmetric.driver.example</artifactId>
            <version>1.0.0</version>
        </dependency>
    </dependencies>
</project>
```

Step 2: Run Maven

By running:

```
mvn clean install
```

the project should build without errors. Inside the target/deps folder you should find all JAR files that are needed to start the system.

Step 3: Create the system configuration

Let's assume that you want to install the system, including the configuration, inside the folder $HOME\reatmetric_example. So first create folder $HOME\reatmetric_example and copy all the contents of the deps folder into a folder beneath this newly create folder, e.g. $HOME\reatmetric_example\bin.

As second step, create the $HOME\reatmetric_example\configuration.xml file, needed to configure the ReatMetric Core module.

```
<ns1:core xmlns:ns1="http://dariolucia.eu/reatmetric/core/configuration">
    <name>Example System</name>
    <log-property-file>$HOME\reatmetric_example\log.properties</log-property-file>
    <archive-location>$HOME\reatmetric_example\archive</archive-location>
    <definitions-location>$HOME\reatmetric_example\model</definitions-location>
    <scheduler-configuration>$HOME\reatmetric_example\scheduler-
configuration.xml</scheduler-configuration>
    <driver name="Example Driver"
        type="eu.dariolucia.reatmetric.driver.example.ExampleDriver"
        configuration="EXAMPLE.SYSTEM.DRIVER" />
    <driver name="Automation Driver"
        type="
eu.dariolucia.reatmetric.driver.automation.groovy.GroovyAutomationDriver"
        configuration="$HOME\reatmetric_example\automation" />
    <driver name="HTTP Driver"
        type="eu.dariolucia.reatmetric.driver.httpserver.HttpServerDriver"
        configuration="$HOME\reatmetric_example\http" />
    <autostart-connectors startup="true" reconnect="true" />
</ns1:core>
```

Now you have to create the files and folders pointed by this configuration file:

- $HOME\reatmetric_example\model folder
- $HOME\reatmetric_example\automation folder
- $HOME\reatmetric_example\http folder

Create the log.properties file in the $HOME\reatmetric_example, example:

```
handlers = java.util.logging.ConsoleHandler, java.util.logging.FileHandler

.level = OFF
eu.dariolucia.level = INFO

java.util.logging.ConsoleHandler.level = ALL

java.util.logging.FileHandler.level = ALL
java.util.logging.FileHandler.pattern=C:\\temp\\reatmetric.log
java.util.logging.FileHandler.limit=5000000
```

```
java.util.logging.FileHandler.count=1
java.util.logging.FileHandler.formatter=java.util.logging.SimpleFormatter
```

Create the configuration.xml folder in the $HOME\reatmetric_example\automation, example:

```
<ns1:automation xmlns:ns1="http://dariolucia.eu/reatmetric/driver/automation">
    <script-folder>$HOME\reatmetric_example\automation\scripts</script-folder>
</ns1:automation>
```

Create the folder $HOME\reatmetric_example\automation\scripts.

Create the configuration.xml folder in the $HOME\reatmetric_example\http, example:

```
<ns1:httpserver xmlns:ns1="http://dariolucia.eu/reatmetric/driver/httpserver"
 host="127.0.0.1"
 port="8081">
</ns1:httpserver>
```

Copy the example_model.xml of the driver into the $HOME\reatmetric_example\model folder.

Step 4: Create a launcher script

Create a launcher script inside the $HOME\reatmetric_example folder. The script assumes that the java executable is in the path.

```
    java
    --module-path="bin"
    -Dreatmetric.core.config=<path to reatmetric_example folder>\configuration.xml
    --add-exports javafx.base/com.sun.javafx.event=org.controlsfx.controls
    -m eu.dariolucia.reatmetric.ui/eu.dariolucia.reatmetric.ui.ReatmetricUI
```

Step 5: Create a launcher script

Execute the script and enjoy your ReatMetric instance running.