Comments:

Terminology:

1) Please avoid use of the term *sub-element*, depending on the context use the term 'CSP subsystems' or 'subordinate components. Later in the text one could use simply subsystem or component,
2) In this document the terms: device, component and sub-element are used interchangeably. That can be confusing, it is not immediately obvious are those the same or different items. Can we use the term 'CSP subsystem' to refer to one of: CBF, PSS, PST ? And 'CSP subsystems' (plural) to refer to all three.
3) Suggestion:

- the term subordinate component can be used to describe a hierarchy of components without naming anything specific (CBF, PSS and PST are subordinate components of CSP).

Class Component:
According to the description the class Component is a proxy for a subordinate component (or sub-system). In the case of CSP.LMC, there are 3 subordinate components: CBF, PSS and PST.
It is my understanding that the class Component supports execution of asynchronous commands, locally stores some of the data related to that component, for example: a) FQDN (or other access information), b) configuration (what the current configuration of the component should be), and c) status (the last reported values for the attributes of interest). It probably also monitors the status and subscribes to receive the events, warnings, alarms, etc.
Suggestion: the name Component is too general; it is already mentioned that this class is a proxy, can we call this class ComponentProxy or ComponentModel.
The problem is that we also have TANGO Device Proxies.

I have hard time believing that the same class can be used for Controller and Subarray, and for all 3 subsystems, but I will be glad if that can work.

So CSPController would have:
CbfControlProxy, PssControlProxy, PstControlProxy
CbfControlTDProxy, PssControlTDProxy, PstControlTDProxy

CSPSubarray would have:
CbfSubarrayProxy, PssSubarrayProxy, PsSubarrayProxy
CbfSubarrayTDProxy, PssSubarrayTDProxy, PstSubarrayTDProxy

I am sure you will find better names.

Capability Device Manager – not sure what is this responsible for.

Question: Is it true that the CSP Controller has one instance of the class EventManager – and a single instance subscribes with CBFController, PSS Controller and PSTController to all events of interest.

CT-255 Textual documentation for refactored code

E.Giani, G.Marotta, C.Baffa

Firenze 27 May 2021

1 Foreword

CSP.LMC code has been re-factored to improve the overall design and structure.
The current code is difficult to maintain, update and test for several reasons:

- it doesn't follow the object orient principles, resulting in a monolithic structure, with classes and methods too unwieldy to manipulate easily
- multiple changes are required in different part of the code in order for the changes to work properly
- a strict coupling with the TANGO layer with bad consequences on the testability of the system in isolation.

A drastic refactoring was clearly needed, and we prepared it for some time. The effective planning finally took place during PI 10, under the umbrella of SPO-1090. In this document we document the approach and the refactoring plan.

In the following with component, we mean one of the three CSP sub-elements: CBF, PSS, PST. As there is also a Component class, while capitalized, we intend the class, not the entity.

For brevity, in the following CSP-LMC Controller and CSP-LMC Subarray are referred by CSP Controller and CSP Subarray.

2 The overall approach

During the initial analysis, a number of assumptions were made: these apply to both the CSP-LMC Controller and the Subarray so the solutions adopted can be easily recycled in the development of both devices. These are inspired from the standard design patterns (Observer, Command, Aggregator/Mediator) [1]. In the following, assumptions used as starting point are proposed.

1. The CSP Controller communicates mainly with three sub-elements: CBF, PSS and PST. It must also access the CSP Subarrays and the Capabilities device manager to report the information on the resources. However the main operations are carried out into the three sub-elements controllers.
   The interaction between the CSP Controller and the sub-element devices can be mediated through a class that works as a *proxy (Component Class).* This approach has the advantage of abstraction. Specific operations on a sub-element can be done by specializing the proxy class for each sub-element and the corresponding functions are maintained in a specific part of the code.
2. A command issued on the CSP Controller (*controller command*) by a TANGO client or the TM, breaks up, nearly always, into three commands, one for each sub element. These three commands (*sub-commands* or *component commands*) are forwarded to the connected subelements.
   The CSP Controller has to be able to:
   A) invoke the command on a sub-element and monitors its execution, detecting its progress and its final status (success/failure).
   The operations to be performed are the same for all commands (On, Off, etc.):
   - check the initial device state to determine if the command is allowed

Commented [SV1]: Avoid the term sub-element. We could say CSP subsystems.

Commented [SV2]: Why lower-case?

Commented [SV3]: The previous sentence uses the term sub-element Controller, does this refer to the same object?

Commented [SV4]: In TANGO this would be a DeviceProxy class.
For a component that does not implement TANGO API a bespoke class.

Commented [SV5]: CSP subsystems or CBF, PSS and PST.

Commented [SV6]: 'is allowed' is usually implemented by the server not by a client or proxy. But in some circumstances a client may implement additional checks (limitations that server is not aware of).

- support synchronous and asynchronous execution
- wait for the final status (the one expected after the end of successful execution) and detect possible conditions of failures
- implement support for timeout

  These functionalities can be customized in a specific class (*ComponentCommand Class*), using an approach similar to the one described by the Command Pattern Design.

B) detect the end of all the sub-commands to be able to report the end of the *controller command*. A specific class *(CommandObserver Class)*, using the Observer Pattern Design, can be used to detect the *controller command* completion. Each sub-command is registered within the observer and notifies it when it has completed.

3. Management of the events can be rationalized delegating the subscription of events to a specific class (*Event Manager Class)*.
4. Provide a class working as interface to the TANGO system so that the TANGO API can be easily mocked during the tests *(Connector Class)*.
5. Use of Factories to instantiate concrete classes.

In Figure 1 the class diagram for CSP-LMC Controller is proposed, it shows all the classes and their relationships. The main Python classes are described in the following sections.

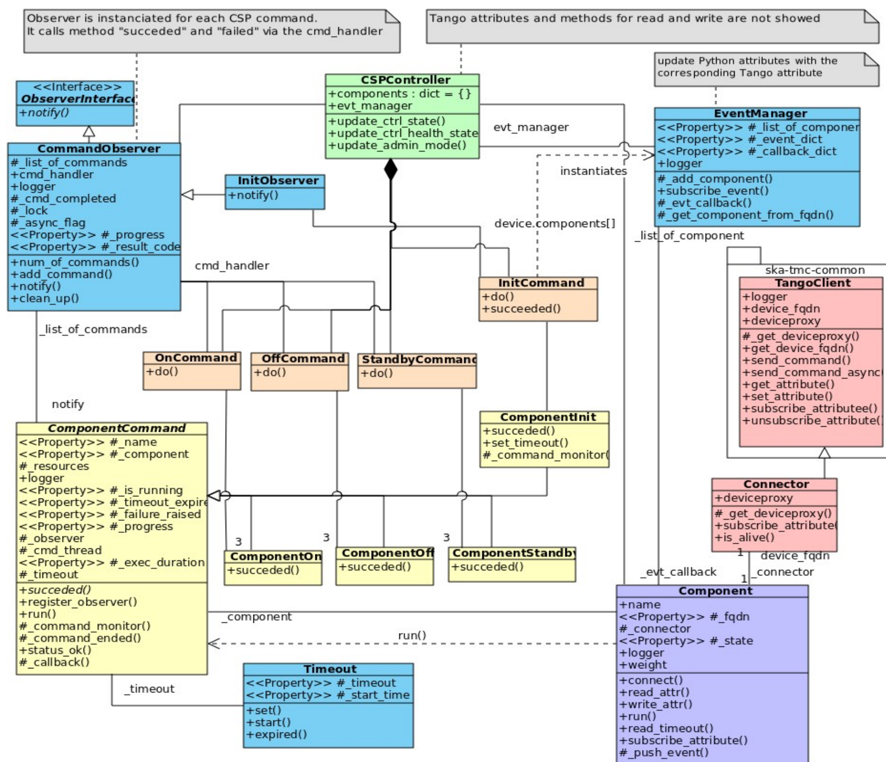

Figure 1: Class Diagram for CSP-LMC Controller

Commented [SV7]: In some cases it is OR.

Commented [SV8]: Add to this list: abort execution of the asynchronous command (does not need to be implemented immediately but will be required).

Commented [SV9]: This is true for almost every layer in the SKA CS Telescopes.

Commented [SV10]: What about the events that signal a command failure or completion?

Commented [SV11]: It would be useful to indicate which of these classes provide TANGO API (and are TANGO classes) and which are not. It is my understanding that the CSP Controller provides TANGO API to the clients above in the hierarchy (e.g. TMC), and that class TANGO Client is a DeviceProxy for one of the subordinate components (CBF Controller, PSS Controller, PST Controller or other TANGO device). Would it be too complicated to use color to differentiate TANGO from non-TANGO classes? And to move the TANGO Client to the bottom to suggest that it communicates downwards to subordinate components. Sorry, I know this is a class diagram not component diagram, still it would be easier to comprehend.

Commented [SV12]: After reading the whole document I suggest that instead of re-arranging this diagram you provide a Component and Connector diagram to show the software components during run-time: CSP Controller with the components for CBF, PSS and PST – only then the reader will understand how this works. Please show the object that provides CSP Controller TANGO API on the top, the objects that act as TANGO Clients for the CBF, PSS and PST controllers at the bottom and the non-tango components in the middle. I am convinced that would also help you understand this completely and perhaps catch missing bits.

3 Component Class

The interaction between the CSP Controller TANGO Device and a sub-element TANGO device, is realized via an instance of the Python *Component* class. This class is a new CSP specific class.
A Component object works as a proxy (or as a cache) to the corresponding TANGO device. The CSP Controller interacts directly with a Component to get/set attributes and run commands on the corresponding sub-element TANGO Device.
A Component instance is identified via its:

- name: *cbf-ctrl*, *pss-ctrl, pst-ctrl*
- address*:* the sub-element *FQDN*
- weight: a measure of the 'impact' of the component on the CSP Controller core functionalities.
  Value = 1 the component has a big impact on the CSP Controller. Failure of this component causes a failure of the CSP Controller *(State = FAULT)*
  Value = 0: failure in this component affects only a limited set of the CSP Controller functionalities and it is reflected in a degraded performance of CSP Controller (*healthState = DEGRADED*).

The Component object implements:

- the business logic to execute a command on the target sub-element device
- properties and methods to report information such as for example its State, adminMode and healthState
- subscription to events on the corresponding sub-element TANGO device

The big advantage of the Component class is that this class can be specialized for the specific subsystem and also for the Telescope, since Mid and Low sub-elements require different approaches as consequences of the different hardware.

4 Component Command Class

The *ComponentCommand* class models a command acting on a *Component* object. This class is a new CSP specific class.
This class declares an interface for all commands, providing a simple *execute* method which asks the receiver of the command (Component) to carry out the operation.

The *ComponentCommand* class, when instantiated for a specific command (On, Off , etc) contains all the information about the request such as: the input parameters (if any), the Component to act on, the conditions of success or failure, the timeout, etc.

When a TANGO method is invoked on the CSP Controller, the CSP Controller creates one *Component Command* instance for each connected Component, and sets the Component as receiver of the command.
When the CSP Controller invokes the *execute* method, each *Component Command* will run one (or more actions[1]) on the associated Component object.
The *ComponentCommand* does not perform the work on its own: the Component has the knowledge of what to do to carry out the request.

[1] This may be the case with some CSP Subarray commands, such as Restart, ObsReset and Off, which in certain circumstances may require the execution of several consecutive actions to drive the system to the desired final state.

Commented [SV13]: Also attributes – I would say the ComponentProxy locally stores the configuration parameters (attributes) and the current status of the attributes of interest.
That way, after the communication with the component is lost (and again established) or the component has been restarted, the ComponentProxy can restore previous configuration. (I would say this is desired in general, but I am not sure do we want CSP.LMC to automatically configure CBF, PSS and/or PST after the re-start – perhaps we should implement that and de-activate later if we find that it is dangerous when the full system is installed). Sorry for the confusion, I'll add this to the guidelines.

5 Command Observer

When a command is invoked on the CSP Controller by the TM or any other TANGO client, the same command is forwarded to each connected sub-element controller device. This class is a new CSP specific class.
The CSP Controller command is considered completed when all the forwarded commands have ended. A CSP Controller command is managed via an instance of the *CommandObserver* class.
The *CommandObserver* class has been developed adopting the Observer Design Pattern [].
The CommandObserver declares an interface with the *notify* method. A reference to the *CommandObserver* is maintained by each *ComponentCommand* that, on completion, notifies its state change to the *CommandObserver* by invoking its *notify* method.
The *CommandObserver* keeps also track of the commands forwarded to the sub-element, to be able to detect the completion of all the Component commands and transition the CSP controller to a proper final state.

The *CommandObserver* implements also some attributes to report the status of each Component command, the progress, the running state etc.

6 EventManager Class

The CSP Controller relies on one instance of this class to subscribe to events on the controlled subelement devices. This class is a new CSP specific class.
The *EventManager* is instantiated and registered inside the CSP Controller at initialization. On initialization completion (when the connection with the ~~running~~ sub-element devices has been established) the CSP Controller selects which events on which sub-elements are to be monitored and delegate the subscription to the *EventManager*.
The *EventManager* instance works on the behalf of the CSP Controller to:

- subscribe events on the connected Component
- retrieve the value or errors reported by the callback registered with the events
- adopt backup strategies when an event reports a failure: for example a direct read of the attribute generating the event.

Commented [SV14]: Question: Is it true that the CSP Controller has one instance of this class – and a single instance subscribes with CBFController, PSS Controller and PSTController to all events of interest.

Commented [SV15]: Suggestion: delete. The CSP Controller cannot register with the subsystems which are not responsive. If a subsystem is unresponsive, I would expect that the CSP Controller would report that as alarm, unless admin mode for the subsystem is OFFLINE or NOT-FITTED. Added after reading the next section: Event Manager can subscribe with the ComponentProxy even when the component (e.g. CBFController) is not responsive.

The *EventManager* does not subscribe directly to the sub-element TANGO devices, but relies on the corresponding Component objects to perform such work.
The events received by the *EventManager* from each Component, are pushed back to the CSP Controller via callbacks registered by the CSP Controller at subscription time.

Commented [SV16]: For discussion: So EventManager relies on the ComponentProxy to report events. I would like to discuss this with you.

The introduction of this class decouples the CSP Controller from the sub-elements: all communication takes place via the *EventManager.* The advantage of this solution is that any changes to modify the behavior or improve performances are performed in one place: in the EventManager code.
The EventManager can also be configured by the CSP Controller to carry out particular policies of aggregation on attributes, reducing the load of information traveling to the controller.

As a final remark this object must be well designed to prevent it from becoming a bottleneck in events management. In this context, profiling analysis on this object are required.

7 Connector Class

The current project relies on the *ska-tmc-common* python package developed by the NCRA to implement an abstract layer towards the TANGO Client and Server APIs via the TangoClient and TangoServerHelper classes.

Commented [SV17]: It is not clear will this dependency continue when the new design is implemented?

In particular, the CSP.LMC software provides a *Connector* class that inherits from the TangoClient but overrides and add some methods to model specific features. In particular, the method *is_alive()* will be added to check on device in the initialization process.

8 CSP Controller initialization

The CSP Controller no longer relies on the Kubernetes deployment procedure to perform a coherent startup of its classes and of the component's ones, but implements a non-blocking mechanism in the code to wait until the sub-element devices are ready for connection. This approach has the advantage of a much faster start-up and to avoid the possibility of deadlocks, where devices of different levels wait for each other.
A sequence diagram for a successful initialization is presented in Figure 2.

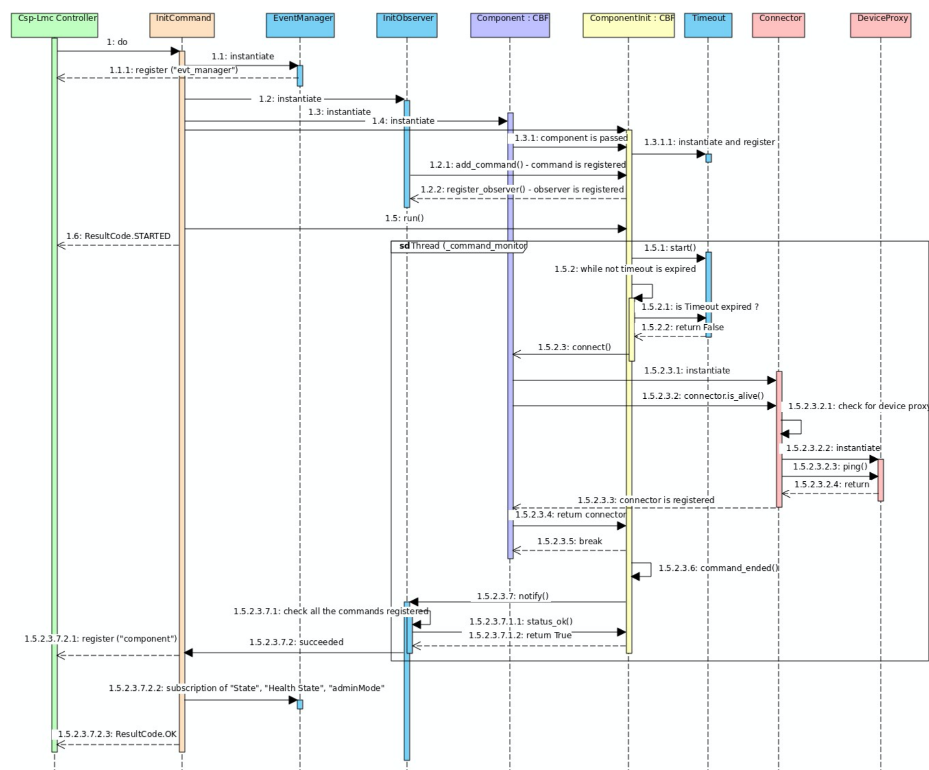

Figure 2: Sequence diagram for the initialization od CSP-LMC Controller

The CSP Controller Init command try to establish a connection with each sub-element component: this operation is performed concurrently. The connection procedure implements also a retry mechanism whose maximum duration is configurable through the *ConnectionTimeout* device property of the CSP Controller. After this period, the connection is considered as failed.

The connection retry mechanism is performed only when the sub-element devices registered within the TANGO DB and have the administrative mode ONLINE or MAINTENANCE.

Connection to a sub-element device not registered into the TANGO DB or registered into the TANGO DB but with administrative mode disabled, is immediately reported as failed.

When the connection with a sub-element component is established, the corresponding Component instance is registered within the CSP Controller: the device maintains a list with all the active (connected) components and when it receives a command from the TM or another TANGO Client, it forwards the command only to the components that are registered within this list.
One at this stage, the CSP Controller instructs the *EventManager* (instantiated at initialization startup) to subscribe a set of attributes on the connected Components. In Figure 2 the behavior of Event Manager is shown.

Commented [SV18]: What happens if the communication with one of the registered components is lost ? The DeviceProxy would detect that and DeviceProxy would report state as UNKNOWN (or FAULT), but are any of the objects destroyed? I would expect not. 2. What if a component which was not available when the CSP Controller was initiated later comes up and wants to register with the CSP Controller. It is quite possible that the CBF, PSS and PST Controllers will be running each on different server (as per baseline design).

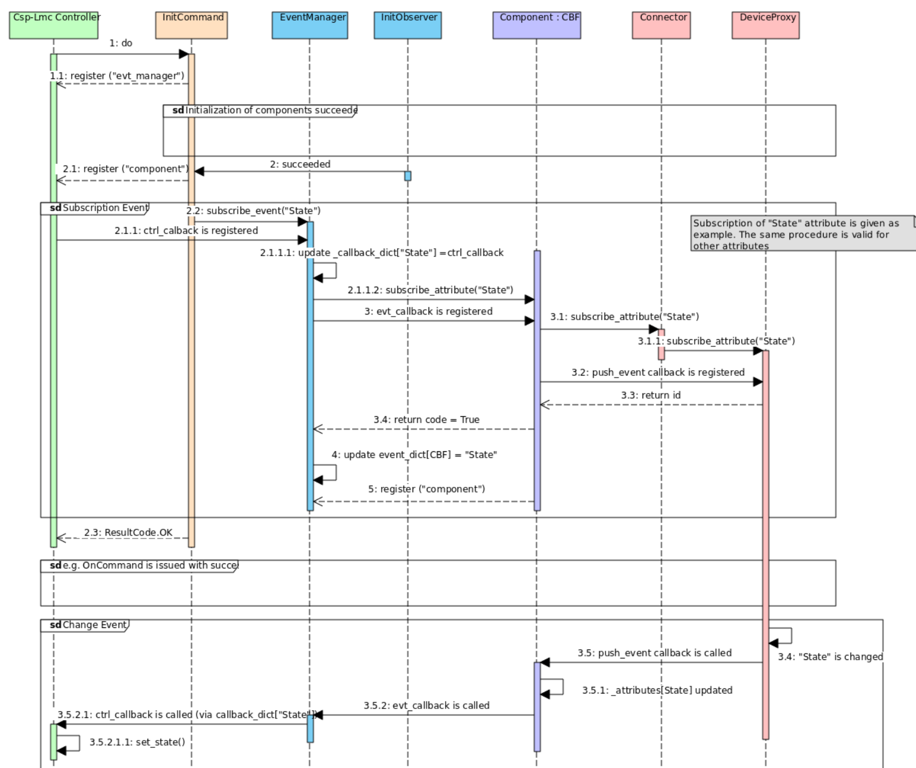

Figure 3: Sequence diagram of the functioning of EventManager: subscription of events and update of attributes

9 CSP.LMC Subarray

The set of classes described before, can be reused to implement a CSP.LMC Subarray. In Figure 4 the Class Diagram for Subarray is proposed. Class Circled in red are the same shown in Figure 1 in the diagram of the Controller.

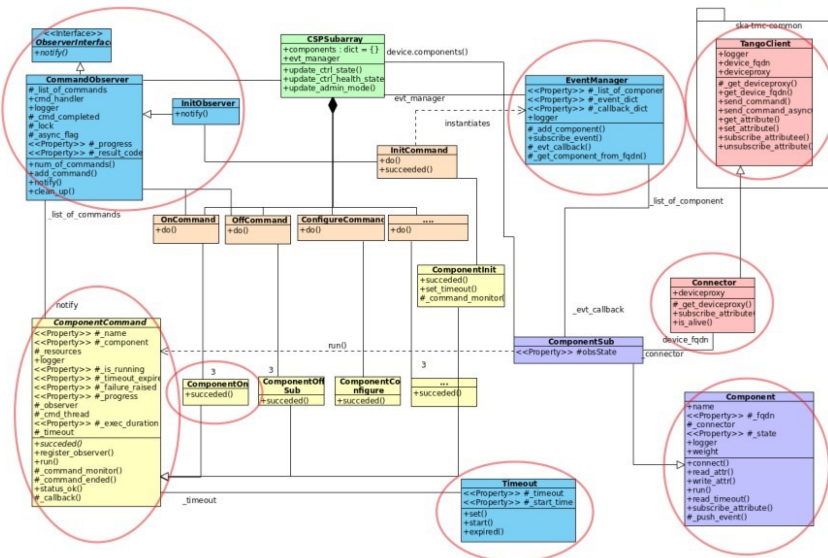

Figure 4: Class diagram for Subarray. CIrcled class are the same of those in Controller scheme

In the following sections a description for several Subarray functionalities will be provided,

CSP Subarray initialization

CSP Subarray initialization works in the same way as for the CSP Controller. The main difference is that the PST sub-element does not provide a Subarray device.
The CSP Subarray does not interact directly with the sub-system TANGO Devices, as the CSP Controller, but through concrete instances of the Component class that act as a proxy to the real devices.
At initialization the CSP subarray attempt a connection to PSS and CBF subarrays and also to all the PST beams. For the latter, it executes a read call of the `beam` subarray affiliation attribute to determine which beams belong to it and adds them to the list of controlled components.
In this way, the CSP Subarray is able to handle both the first initialization and the re-initialization.

The CSP Subarray maintains a list of connected sub-element observing components and every time it receives a requests, it verifies if the targets of the request belong to the list of connected components. If it's not the case, the CSP Subarray connects to those devices not belonging to it, and update the list of the connected components. This behavior make possible to synchronize the the [re]initializing subarray component list with the controlled entities internal records.

At initialization completion, the CSP Subarray subscribes, via the EventManager, the State and Control Mode attributes on each connected device to maintain updated information about their status.

CSP Subarray commands execution

The Subarray commands can be quite different in nature and execution patterns. As can be easily understood this requires special care. In the following we will describe the main examples. We are

confident to have covered the trickiest ones, but as the SKA system is in fast evolution, we are aware some important points will not be covered here.
In Figure 5 the sequence diagram for Configure Command is proposed.

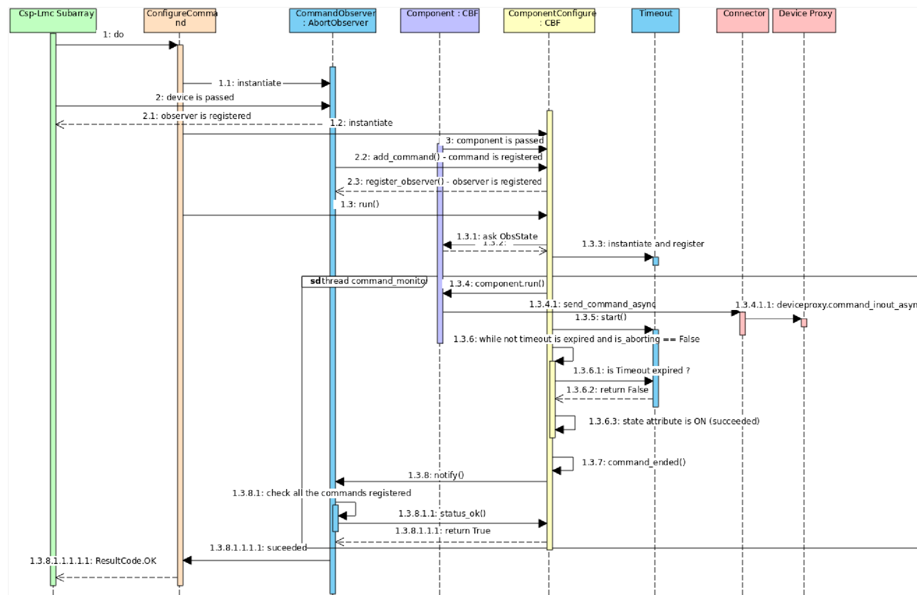

Figure 5: Sequence diagram for Configure Command of Subarray

CSP Subarray Restart/Reset commands

Restart and Reset commands are issued on a CSP Subarray to exit from a FAULT or ABORTED state. Restart command transitions the subarray to an EMPTY state (no resource assigned) while the second one to an IDLE state (resources assigned but not configured).

Restart and Reset commands are often issued on odd system situation, so a lot of care should be taken to have robust implementations.

A not coherent situation happens if the CSP Subarray is in FAULT/ABORTED but not all the subelement observing devices[2] are in that state, due to slow communications or some errors in controlled elements. This situation requires a special action: if command Restart (or Reset) issued on the CSP Subarray is simply forwarded to all the connected sub-elements, the command could fail because the sub-element observing devices not in fault/aborted are not allowed to process the received command.
The solution to this odd situation requires that the Restart/Reset command is executed in a different way on the different sub-element observing devices: the device in fault will receive the

[2] The term *observing device* is used to refer to a sub-element subarray as well as a PST beam (the PST does not rely on the subarray device to handle an observation).

Restart/Reset command while the others receive a sequence of commands (for example Abort + Restart) to transition the associated component to the expected final state.
The current implementation of the ComponentCommand class, helps to handle such situations: the ComponentCommand class instantiated to execute the Restart/Reset command on a component, can be set up to run a different set of actions depending on the state of the component it acts on.

CSP Subarray Abort command

The Abort command is issued on a CSP Subarray to stop abruptly the current running operation.
On command success, the device transitions to ABORTED state.
Abort command is often issued on odd system situation, so a lot of care should be taken to have robust implementations.
In Figure 6 the sequence diagram for Abort Command is proposed.

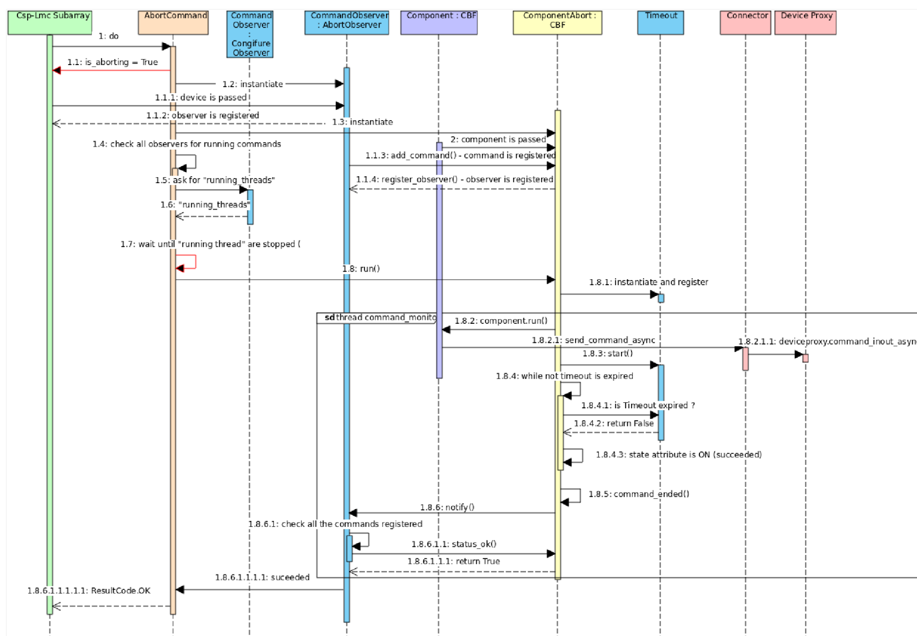

Figure 6: Sequence diagram for Abort Command

10 References

[1] GAMMA, E., HELM, R., JOHNSON, R. AND VLISSIDES, J., Design patterns: Elements of reusable object-oriented software 1995 - Addison-Wesley - Reading, Mass.