Arcetri Technical Report
SKA Project Series

Commands and Setting for CSP

C.Baffa, E.Giani M.Vela Nuñez

Abstract

During the normal set-up and programming of the many elements and sub-element of SKA it arises the necessity to program many parameter and to execute different commands. During the development of CSP.LMC prototype we designed a flexible schema to implement such functionality inside the TANGO Control paradigm used in SKA Control and Monitor. We sketch here our implementation proposal and analyse few use cases.

Arcetri Technical Report 2/2016 - 28 June 2016

1 Structure

The CSP.LMC prototype we developed, in its overall structure, follow as closely as possible the physical structure of CSP, adding some abstract-related classes as the Capability ones. Its logical components are illustrated in Figure 1.

Apart from the overall structure, the CSP.LMC prototype is composed by a number of specialized classes which share a common basic structure. In particular they all share the SKA CSP Guidelines status/state variable (see Appendix 1 : Mapping between SKA State and Mode and Tango state) implementing only the relevant ones. These classes belongs to few families:

1. Master classes, which manage a physical component and report its status and telemetry
2. Capability classes, which manage a logical component and report its status
3. LMC classes, which handles the command and monitoring of local LMC hardware
4. Alarm classes which manage alarms, up to complex behaviour as grouping, masking and censoring

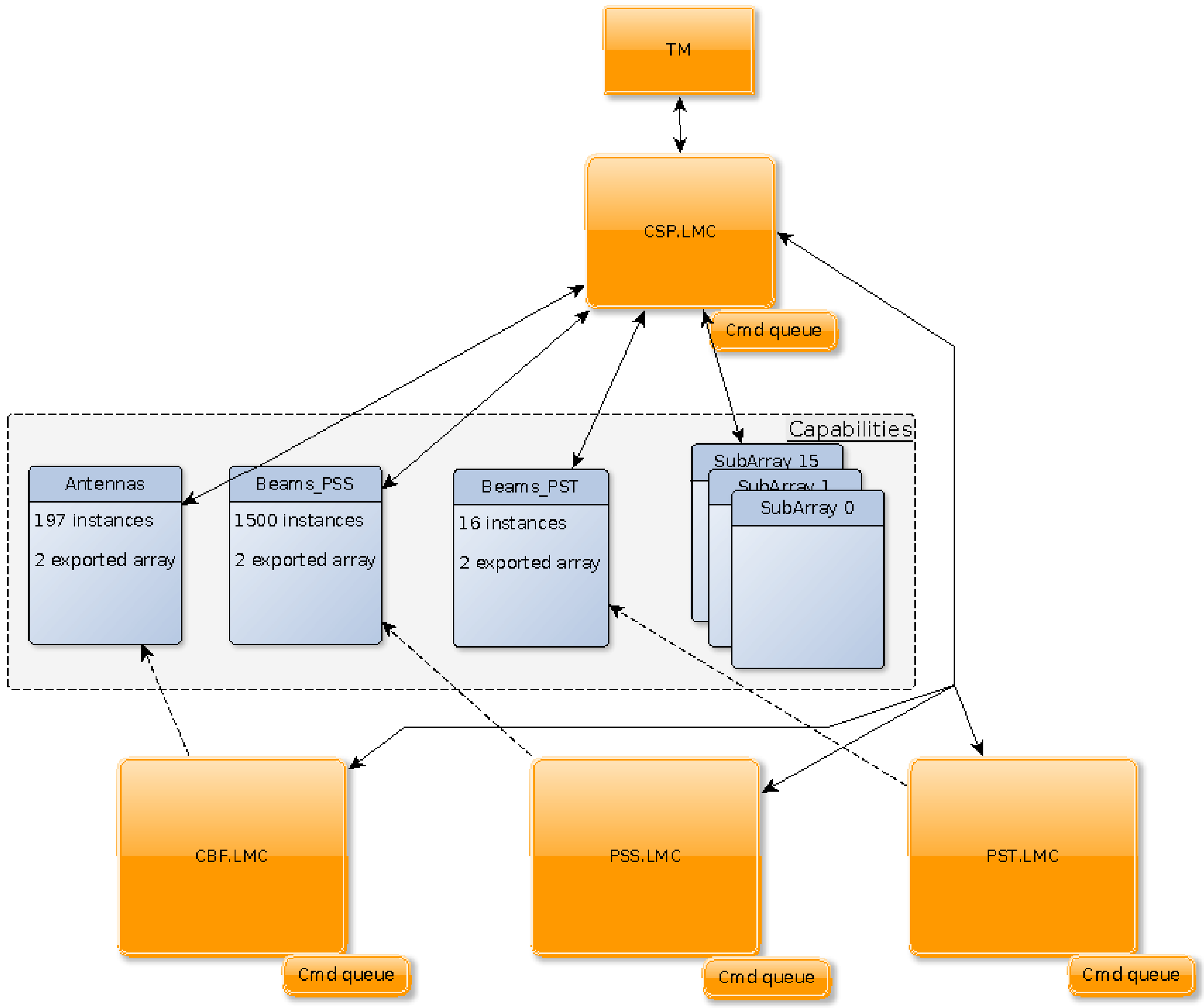

Figure 1: Logical structure of CSP.LMC prototype

2 Proposed Set/Command bundle

Our proposal for the set-up of a SKA component is based on a container approach. Consequently it can be implemented using progressively more complex structure.

In its simplest form we can only set a number of attributes on a single node, while in its most complex form it can perform the setting of a large number of parameters on a hierarchy of node and can execute commands on specific nodes.

We propose to implement a single command on all nodes. Let's call it *setParam*. This command behaviour is completely defined by a single self-describing text parameter. Let's assume a Json format for this parameter, but other formats can be easily implemented. In the present Tango implementation this string parameter can be passed along as command parameter. In future, we expect to exploit the the proposed TANGO REST interface.[1]

The program handling of a Json string can became trivial by means of the numerous library implementations available for all TANGO programming languages[2].

It its simplest form *setParam* command just set different attributes to specific values In this case :

```
{
        “scanTime”:  “34.12” ,          // scan (integration) time
        “beamBw”:  “2”,
        “accelerationRange” : “0”,
        “DispersionMeasure”:  “300”,...
}
```

In a more complex implementation, the attributes values can be specified also for lower level devices:

```
“CSP” : {
        “scanTime”:  “34.12” ,          // scan (integration) time
        “beamBw”:  “2”, …
}
“PSS” : {
        “accelerationRange” : “0”,
        “DispersionMeasure”:  “300”,...
}
```

In the most complete implementation, attribute setting can be mixed with command execution on a hierarchy of devices. Examples of this approach can be found as the possible implementation of uses cases in the following sections. A particularly suggestive example can be found in section 3.16.

---

1 Such approach is different and, in SKA case, more efficient than the SetParameters command. For the self-describing nature of Json coding error risk is reduced and also there is no more the necessity to transfer always ALL parameters

2 As an example, using C++/nlohmann implementation, (https://github.com/nlohmann/json), one can simply write: json j; j[“scanId”]=34; j[“sourceName”]=”CrabPulsar”;

3 Execution of Settings.

We analyse some setting commands. For brevity we put many commands in the same json packet (par 3.1). To have meaningful error reports it should be a good practice to put a single action on each command.

As an alternative approach, we can implement elementary commands for each action to be sent separately (cfr par 3.4).

3.1 Commands to initialize sub-array(s) – Slave SubArrays elementary commands

This use case is analysed using the most complete form of *setParam* command. Here we use two different calls.

```
Command:  createSubArray   From: TM       Destination: CSP.LMC (cspMaster).
Argument: Json  String {
“activationTime”: “10:30:00”, // should be a Unix time
“sourceId”: “TM”,
“commandId” : “123456”,     // identifies this execution
“createSubArray”: { // init
        “subArrayId”: “0”,
        “antennasList”: “0,1,2,3,4,10,100”,
        “creationDate”: “20160310 10:30:00”,
        “administrativeMode”: “enabled”,
        “observingMode”: “0”,         // idle
}
```

```
Command:  createSubArray   From: TM       Destination: CSP.LMC (cspMaster).
Argument: Json  String {
“activationTime”: “10:30:00”, // should be a Unix time
“sourceId”: “TM”,
“commandId” : “123457”,     // identifies this execution
“createSubArray”: { // init
        “createSubArray”: {   // we do not assume a single device
                “subArrayId”: “1”,
                “antennasList”: “10,11,12,13,14,20,100”,     // generate an error on 100!
                “creationDate”: “20160310 10:30:00”,
                “administrativeMode”: “enabled”,
                “observingMode”: “0”,         // idle
        }
}
```

3.2 Details of Execution

Here we will display the execution of the command above, split among the different LMC components.

CSP.LMC	SubArray{0-15}	Antennas
Receive the command from TM		
Acknowledge command 123456		
Identify the CSP section and parse it		
Verify if subArray 0 is already initialized and has Antennas allocated	Report status of subArray 0 and allocated Antennas	
Verify the availability of requested Antennas		Reports the availability of Antennas
If subArray 0 already has Antennas allocated Rise error → End of Command		
If subArray 0 is not IDLE Rise error → End of Command		
Allocate 0,1,2,3,4,10,100 to subArray 0	Register Antennas 0,1,2,3,4,10,100 on subArray 0	Register Antennas 0,1,2,3,4,10,100 to subArray 0
Write other attributes to SubArray (creationDate, administrativeMode, observingMode, ...)	Update other attributes to SubArray	
Acknowledge successful execution of command 123456		
Acknowledge command 123457		
Verify if subArray 1 is already initialized and has Antennas allocated	Report status of subArray 1 and allocated Antennas	
Verify the availability of requested Antennas		Reports the NON availability of Antennas
Detects conflicting Antenna Assignation Rise error		
Acknowledge unsuccessful execution of command 123457		

3.3 Discussion

This command assumes 16 Tango devices for subArrays. These devices are very simple (slave implementation).

In this scenario the CSP createSubArray command programs both SubArray and Antennas structures. This command is executed directly.

The conflict exception on antenna 100 will be raised by CSP.Master device.

In this approach each command receives its commandId from TM.

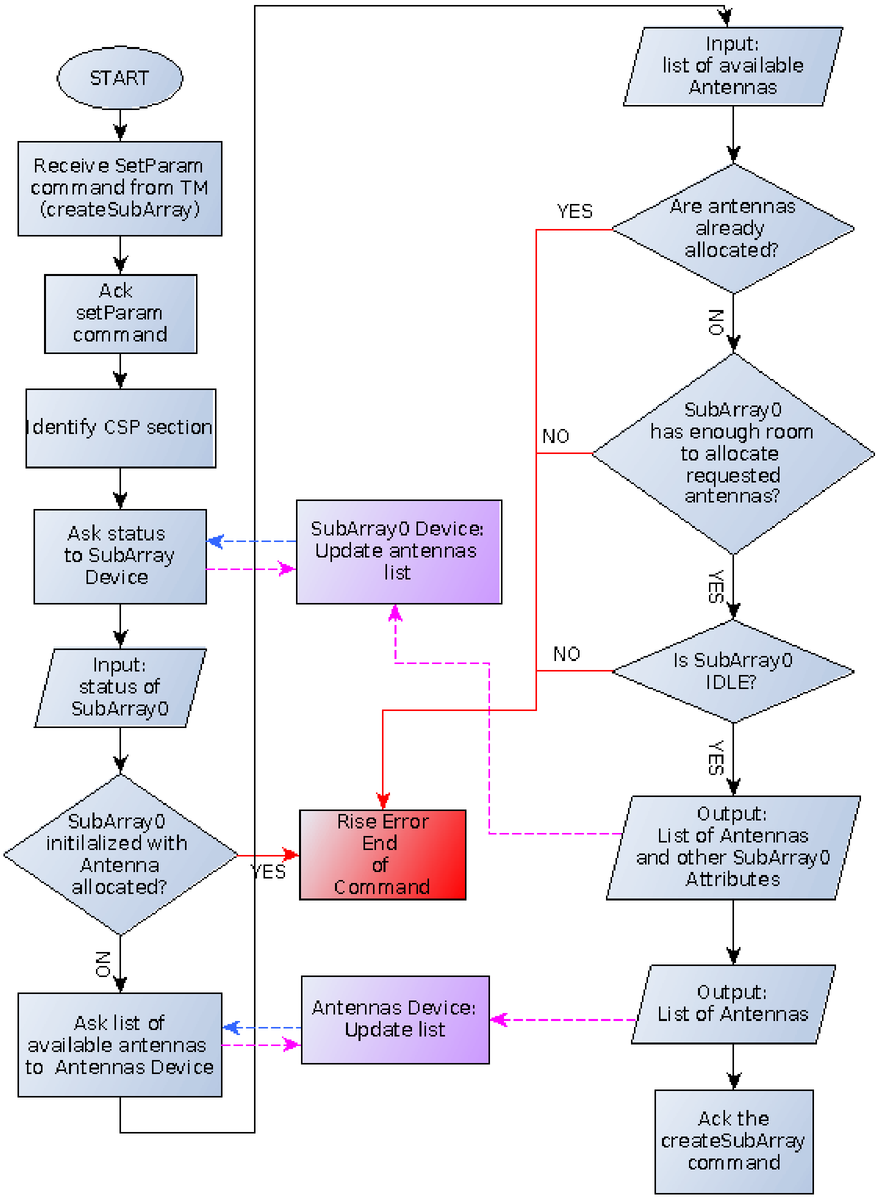

Figure 2: SubArray Initialization flow of operations. Errors in red, Capabilities in purple.

3.4 Command to initialize sub-array(s) – Slave SubArrays Compounded Commands

In this analysis we have defined a general container command *setParam* which accept as argument a json string containing setting of parameters or commands to be executed on local or lower level servers.

Command: *setParam* From: TM Destination: CSP.LMC (cspMaster).
Argument: Json String {

```
“activationTime”: “10:30:00”, // should be a Unix time
“sourceId”: “TM”,
“commandId” : “123456”,     // identifies this execution
“CSP” : {
        “createSubArray”: { // init
                “subArrayId”: “0”,
                “antennasList”: “0,1,2,3,4,10,100”,
                “creationDate”: “20160310 10:30:00”,
                “administrativeMode”: “enabled”,
                “observingMode”: “0”,         // idle
                “commandId” : “123456/1”,  // identifies this execution
        }
        “createSubArray”: {   // we assume a single device
                “subArrayId”: “1”,
                “antennasList”: “10,11,12,13,14,20,100”,     // generate an error on 100!
                “creationDate”: “20160310 10:30:00”,
                “administrativeMode”: “enabled”,
                “observingMode”: “0”,         // idle
                “commandId” : “123456/2”,  // identifies this execution
        }
}
}
```

3.5 Details of Execution

Here we will display the execution of the command in section Error: Reference source not found split in the execution in the various software components.

CSP.LMC	subArray{0-15}	Antennas
Receive the command from TM		
Acknowledge command 123456		
Identify the CSP section and parse it		
Spawn the execution of first command (123456/1)		
Verify if subArray 0 is already initialized and has Antennas allocated	Report status of subArray 0 and allocated Antennas	
Verify the availability of requested Antennas		Reports the availability of Antennas
If subArray 0 already		

CSP.LMC	subArray{0-15}	Antennas
has Antennas allocated Rise error → End of Command		
If subArray 0 is not IDLE Rise error → End of Command		
Allocate 0,1,2,3,4,10,100 to subArray0	Register Antennas 0,1,2,3,4,10,100 on subArray0	Register Antennas 0,1,2,3,4,10,100 to subArray 0
Write other attributes to SubArray0 (creationDate, administrativeMode, observingMode, ...)	Update other attributes to SubArray0	
Acknowledge successful execution of command 123456/1		
Spawn the execution of second command (123456/2)		
Verify if subArray1 is already initialized and has Antennas allocated	Report status of subArray1 and allocated Antennas	
Verify the availability of requested Antennas		Reports the NON availability of Antennas
Detects conflicting Antenna Assignation Rise error		
Acknowledge unsuccessful execution of command 123456/2		
Acknowledge unsuccessful execution of command 123456		

3.6 Discussion

This command assumes 16 Tango devices for subArrays which are very simple (slave implementation).

In this scenario the CSP createSubArray command programs both SubArray and Antennas structures.

The conflict exception on antenna 100 will be raised by CSP.Master device.

This approach implements a single 'container' command (setParam) which acts as spawner for the CSP createSubArray commands (and many more).

In this approach each sub-command receives its commandId from TM.

3.7 Command to allocate beams to SubArrays

In this analysis we have defined a general container command *setParam* which accept as argument a json string containing setting of parameters or commands to be executed on local or lower level servers.

Command: *setParam* From: TM Destination: CSP.LMC (cspMaster).
Argument: Json String {
“activationTime”: “10:31:00”, // should be a Unix time
“sourceId”: “TM”,
“commandId” : “123456”, // identifies this execution
“CSP” : {
    “allocateBeams”: { // init
        “beamsType”: “PSS”,// it can be PSS, PST and VLBI
        “subArrayId”: “0”,
        “beamsCount”: “5”,
        “creationDate”: “20160310 10:31:00”,
        “commandId” : “123456/1”, // identifies this execution
    }
}
}

3.8 Details of Execution

Here we will display the execution of the command in section Error: Reference source not found split in the execution in the various software components.

CSP.LMC	subArray{0-15}	PssBeams Capability
Receive the command from TM		
Acknowledge command 123456		
Identify the CSP section and parse it		
Spawn the execution of command (123456/1)		
Verify if subArray0 is already initialized and has Antennas allocated	Report status of subArray0 and allocated Antennas	
If subArray 0 has not any Antennas allocated Rise error → End of Command		
If subArray 0 is not IDLE Rise error → End of Command		
Ask PssMaster list of available PSS Resources		
If avalable PSS Resource are less than requested beams Rise error → End of Command		
Ask CbfMaster list of available CBF Beams Resources		
If avalable CBF Beams Resource are less than requested beams Rise error → End of Command		

CSP.LMC	subArray{0-15}	PssBeams Capability
CSP.LMC creates a correspondence table between PSS Resource and CBF Beams		
CSP.LMC update PssBeam Capability		Update the PssBeam correspondence table
CSP.LMC update SubArray Capability	Update the implemented PssBeam table	
If avalable subArray PssBeam table is full Rise error →End of Command		
Write other attributes to SubArray0 (creationDate, administrativeMode, observingMode, ...)	Update other attributes to SubArray0	
Acknowledge successful execution of command 123456/1		
Acknowledge successful execution of command 123456		

3.9 Discussion

This command assumes 16 Tango devices for subArrays which are very simple (slave implementation) as is assumed a simple PssBeams Capability device.

In this scenario the CSP allocateBeams command programs both SubArray and PssBeams structures.

This approach implements a single 'container' command (setParam) which acts as spawner for the CSP allocateBeams commands (and many more).

In this approach each sub-command receives its commandId from TM.

Proposal: the acknowledges for a command are sent to TM, while the acknowledges for sub-commands are handled by CSP.

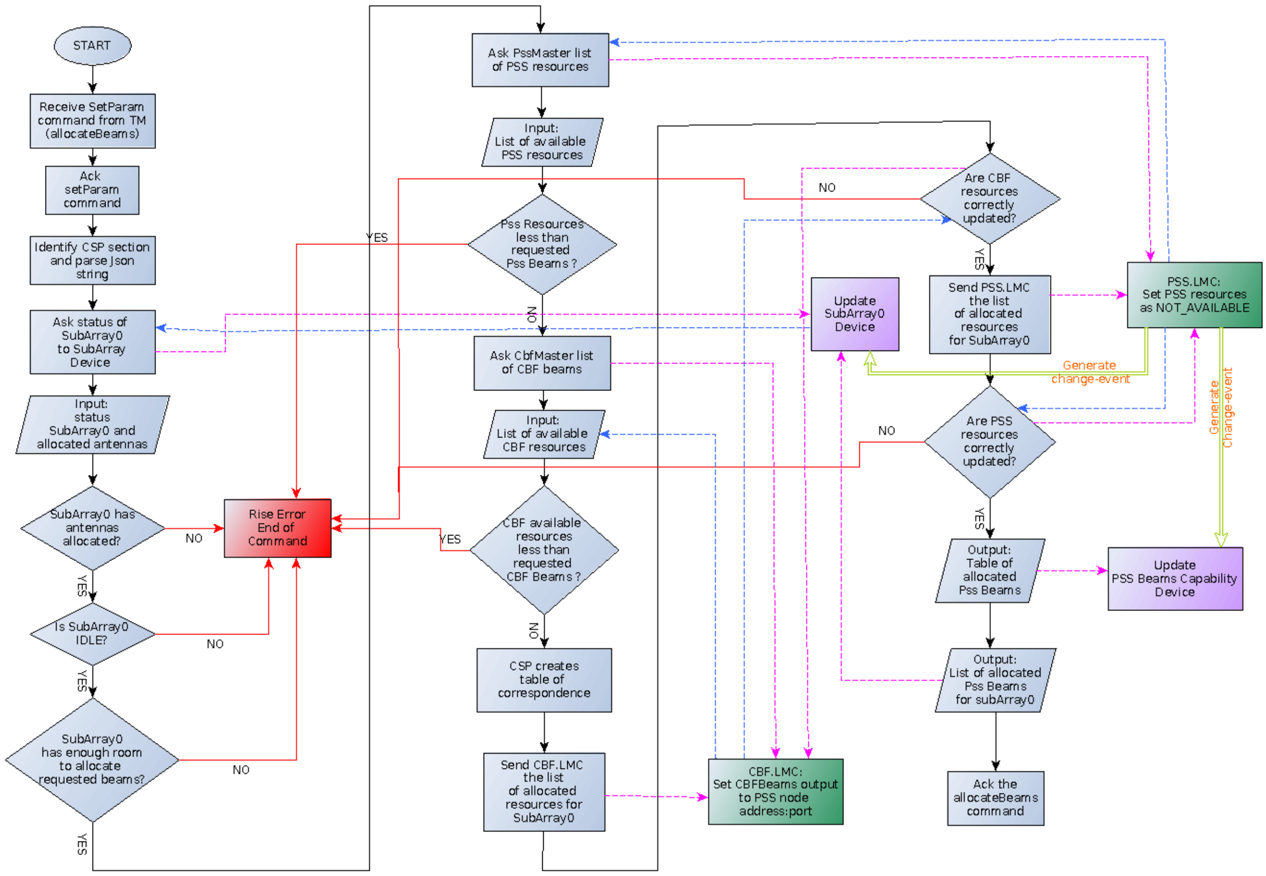

Figure 3: Graphic flow of beam allocation operations. Error handling in red, Capabilities in purple and SubDevices in green

3.10 Command to remove an antenna from a sub-array

In this analysis we have defined a general container command *setParam* which accept as argument a json string containing setting of parameters or commands to be executed on local or lower level servers.

Command: *setParam* From: TM Destination: CSP.LMC (cspMaster).
Argument: Json String {
“activationTime”: “10:30:00”, // should be a Unix time
“sourceId”: “TM”,
“commandId” : “123456”, // identifies this execution
“CSP” : {
    “removeAntennas”: { // init
        “subArrayId”: “0”,
        “antennasList”: “0,1”,
        “creationDate”: “20160310 10:30:00”,
        “commandId” : “123456/1”, // identifies this execution
    }
}
}

3.11 Details of Execution

Here we will display the execution of the command in section Error: Reference source not found split in the execution in the various software components.

CSP.LMC	subArray{0-15}	Antennas
Receive the command from TM		
Acknowledge command 123456		
Identify the CSP section and parse it		
Spawn the execution of first command (1234561)		
Verify if subArray0 is already initialized and has Antennas allocated	Report status of subArray0 and allocated Antennas	
Verify the availability of requested Antennas		Reports the availability of Antennas
If subArray 0 does not have specified Antennas allocated Rise error → End of Command		
Deallocate Antennas 0,1from subArray0	Deallocate Antennas 0,1 from subArray0	Deallocate Antennas 0,1 from subArray 0
Acknowledge successful execution of command 123456/1		
Acknowledge successful execution of command 123456		

3.12 Discussion

This command assumes 16 Tango devices for subArrays which are very simple (slave implementation).

In this scenario the CSP removeAntennas command programs both SubArray and Antennas structures.

This approach implements a single 'container' command (setParam) which acts as spawner for the CSP createSubArray commands (and many more).

In this approach each sub-command receives its commandId from TM. The same analysis apply to the command addAntennas.

Proposal: the acknowledges for a command are sent to TM, while the acknowledges for sub-commands are handled by CSP.

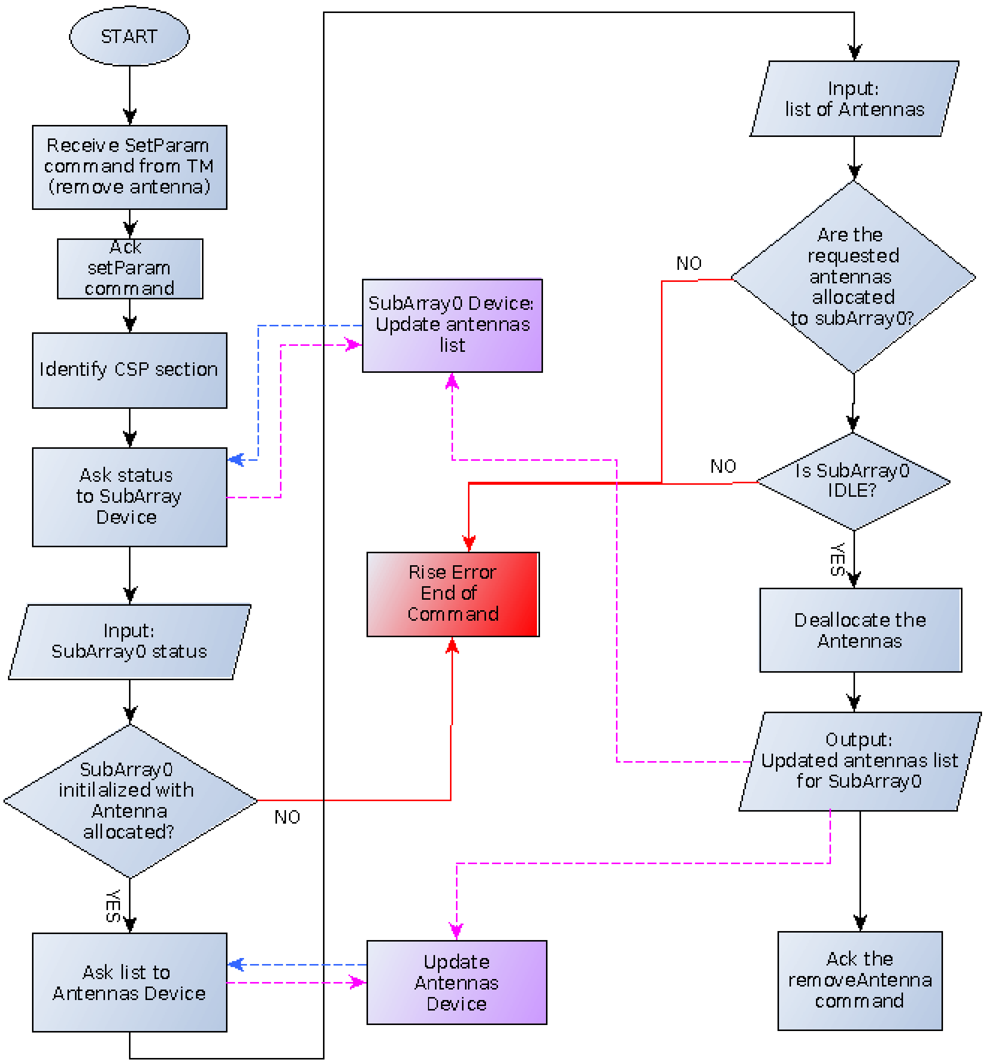

Figure 4: Fow of remove Antenna execution. Capabilities in purple, Errors in red.

3.13 Set up of an image observation

In this analysis we have defined a general container command *setParam* which accept as argument a json string containing setting of parameters or commands to be executed on local or lower level servers.

```
Command:  setParam          From: TM       Destination: CSP.LMC (cspMaster).
Argument: Json  String {
“activationTime”: “10:31:00”, // should be a Unix time
“sourceId”: “TM”,
“commandId” : “123456”,     // identifies this execution
“CSP” : {
        “GlobalValues”: { // init of internal variable common to all subsystems
                “subArrayId”: “5”,
                “ObservingMode”: “1”, // imaging
                “commandId” : “123456/1”,  // identifies this execution
        }
}
“CBF.Master”: {
        “setSubArray”:{       // specialized command
                “subArrayId”:  “5”,    // in the fifth slot we host subArray 5
                “ObservingMode” : “1”, // imaging (this will be updated automatically?)
                “scanId”:  “AB45-34”,          // We store scanId for subArray 5
                “scanTime”:  “34.12” ,         // scan (integration) time
                “subArrayObsMode”:  “1”, // imaging
                “programming Parameters” : { … }  // hardware related parameters
                “commandId” : “123456/2”,  // identifies this execution
        }
}
}
```

3.14 Details of Execution

Here we will display the execution of the command in section Error: Reference source not found split in the execution in the various software components.

CSP.LMC	subArray{0-15}	CBF.Master
Receive the command from TM		
Acknowledge command 123456 received		
Identify the CSP section and parse it		
Spawn the execution of command (123456/1)		
Verify if subArray5 is already initialized and has Antennas allocated	Report status of subArray5 and allocated Antennas	
If subArray 5 has not any Antennas allocated Rise error → End of Command		
If subArray 5 is not IDLE Rise error →End of Command		
Acknowledge successful execution of		

CSP.LMC	subArray{0-15}	CBF.Master
command 123456/1		
Identify the CBF section, add common values, if applicable, and send result to CBF.Master		Receive and parse the command 1234562
		Identify the target slot. If subArray 5 is not IDLE Rise error → End of Command
		Update local variable on subArray 5
	Set the local variable in accordance to the CBF values	Update the subArray 5 variables in accordance to local programmed values
		Program the sub components in accordance of local parameters
		If any errors on sub components programming Rise error → End of Command
		Add ObservingMode 1 (Imaging) to ObservingMode
Add ObservingMode 1 (Imaging) to global ObservingMode		
		Acknowledge successful execution of command 123456/2
Acknowledge successful execution of command 123456 (success of 123456/1 & 123456/2)		

3.15 Discussion

This command assumes 16 Tango devices for subArrays which are very simple (slave implementation) as is very simple also the PssBeams Capability device.

In this scenario the CSP execute only the CSP portion of the command, and spawn to each sub-element specified the relative section. A 'global' setParam section can be implemented in order to send the same command/set to all sub-devices.

This approach implements a single 'container' command (setParam) which acts as spawner for the CSP allocateBeams commands (and many more). Inside CSP the Sub arrays parameteres are internally stored as an array of classes. Selected values of these classes can be accessed as array attributes of Tango numerical types.

Proposal: the acknowledges for a command are sent to TM, while the acknowledges for sub-commands are handled by CSP. In this approach each sub-command receives its commandId from TM.

Successful acknowledge execution of command 123456 depends on success of commands 123456/1 and 123456/2.

The global ObservingMode update is the result of succesful update od this observing mode on both sub-element.

3.16 Set up of a PSS observation

In this approach we have defined a general container command *setParam* which accept as argument a json string containing setting of parameters or commands to be executed on local or lower level servers.

Command: *setParam* From: TM Destination: CSP.LMC (cspMaster).
Argument: Json String {
“activationTime”: “10:31:00”, // should be a Unix time
“sourceId”: “TM”,
“commandId” : “123456”, // identifies this execution
“GlobalValues”: { // init of internal variables common to all subsystems
    “subArrayId”: “4”,
    “ObservingMode”: “2”, // PSS
    “scanId”: “AB45-34”, // We store scanId for subArray 4
    “numberOfBeams”: “500”
}
“CSP” : {
    // CSP specific parameters
    “PSSBeamID” : [“AB45-34/1”, “AB45-34/2”, … “AB45-34/500”] // 500 values
    “PSSPointingCoord” : […] // 500 values
    “PSSDestinationAddress” : [“10.1.1.1:4000”, … “10.1.50.10:4000”] // 500 values
}
“CBF.Master”: {
    “setSubArray”:{ // specialized command
        “scanTime”: “34.12” , // scan (integration) time
        “subArrayObsMode”: “2”, // PSS
        “numberOfChannels”: “4096”, // PSS
        “beamBw”: “2”, // PSS
        “bitPerSample”: “8”, // PSS
        “Filter Banks Parameters” : { … ,} // many hardware related parameters
        “Delay Model Parameters” : { … },
        “commandId” : “123456/2”, // identifies this execution
    }
    “setBeams”:{ // specialized command
        “numberOfChannels”: “4096”,
        “PSSBeamID” : [“AB45-34/1”, “AB45-34/2”, … “AB45-34/500”] // 500 values
        “Beam Ponting Parameters” : { … }, // many hardware related parameters
        “commandId” : “123456/3”, // identifies this execution
    }
}
“PSS.Master”: {
    “setSubArray”:{ // specialized command
        “subArrayId”: “4”, // in the fourth slot we host subArray 4
        “scanTime”: “34.12” , // scan (integration) time
        “subArrayObsMode”: “2”, // PSS
        “beamBw”: “2”,
        “accelerationRange” : “0”,
        “DispersionMeasure”: “300”,
        “programming Parameters” : { … } // many hardware related parameters
        “commandId” : “123456/4”, // identifies this execution
    }

“setBeams”:{// specialized command
“beamBw”: “2”,
“accelerationRange” : “0”,
“DispersionMeasure”: “300”,
“PSSBeamID” : [“AB45-34/1”, “AB45-34/2”, … “AB45-34/500”] // 500 values
“programming Parameters” : { … } // many hardware related parameters
“commandId” : “123456/5”, // identifies this execution
}

}
}

3.17 Details of Execution

Here we will display the execution of the command in section 3.16 split in the execution in the various software components.

CSP.LMC	subArray{0-15}	PSSBeams	CBF.Master	PSS.Master
Receive the command from TM				
Acknowledge command 123456				
Identify the CSP section and parse it				
Spawn the execution of command (123456/1)				
Verify if subArray 4 is already initialized and has Antennas allocated	Report status of subArray 4 and allocated Antennas			
If subArray 4 has not any Antennas allocated Rise error → End of Command				
If subArray 5 is not IDLE Rise error → End of Command				
Acknowledge successful execution of command 123456/1				
Identify the CBF section, add common values, if applicable, and send result to CBF.Master			Receive and parse the command 1234562	
Identify the PSS section, add common values, if applicable, and send result to PSS.Master			Identify the target slot. If subArray 4 is not IDLE Rise error → End of Command	Receive and parse the command 1234564

CSP.LMC	subArray{0-15}	PSSBeams	CBF.Master	PSS.Master
Update the BeamPSS variables according to local programmed values		Set the local variable in accordance to the CSP.Master values	Update local variables on subArray 4	Identify the target slot. If subArray 4 is not IDLE Rise error →End of Command
	Set the local variable in accordance to the CBF.Master values		Update the subArray 4 variables according to local programmed values	Update local variables on subArray 4
	Set the local variable in accordance to the PSS.Master values		Program the sub components in accordance of local parameters	Update the subArray 4 variables according to local programmed values
			If any errors on sub components programming Rise error →End of Command	Acknowledge successful execution of command 123456/4
			Acknowledge successful execution of command 123456/2	Receive and parse the command 123456/5
			Receive and parse the command 123456/3	Identify the target slot. If subArray 4 is not IDLE Rise error →End of Command
			Identify the target beams. If beams are not IDLE Rise error →End of Command	Update local variables on Beams
		Set the local variable in accordance to the PSS.Master values	Update local variables on beams	Update the BeamPSS variables according to local programmed values
		Set the local variable in accordance to the CBF.Master values	Update the beamsPSS variables according to local programmed values	Program the sub components in accordance of local parameters
			Program the sub components in accordance of local parameters	If any errors on sub components programming Rise error →End of Command
			If any errors on sub components programming Rise error →End of Command	Add Mode 2 (PSS) to ObservingMode
Add ObservingMode 2 (PSS) to global ObservingMode			Add Mode 2 (PSS) to ObservingMode	Acknowledge successful execution of command 123456/5

CSP.LMC	subArray{0-15}	PSSBeams	CBF.Master	PSS.Master
Acknowledge successful execution of command 123456			Acknowledge successful execution of command 123456/3	

3.18 Discussion

This command assumes 16 Tango devices for subArrays which are very simple (slave implementation) as is very simple also the PssBeams Capability device.

In this scenario the CSP execute only the CSP portion of the command, and spawn to each sub-element specified the relative section. A 'global' setParam section can be implemented in order to send the same command/set to all sub-devices.

This approach implements a single 'container' command (setParam) which acts as spawner for the CSP allocateBeams commands (and many more). Inside CSP the Sub arrays parameteres are internally stored as an array of classes. Selected values of these classes can be accessed as array attributes of Tango numerical types.

The global ObservingMode update is the result of succesful update od this observing mode on both sub-element.

In this approach each sub-command receives its commandId from TM.

Proposal: the acknowledges for a command are sent to TM, while the acknowledges for sub-commands are handled by CSP.

Successful acknowledge execution of command 123456 depends on success of commands 123456/1, 123456/2, 123456/3, 123456/4.

At the end of scan the Beam-PSS and CBF automatically mark free the allocated resources, (can be implemented by an event handler inside CSP-Master).

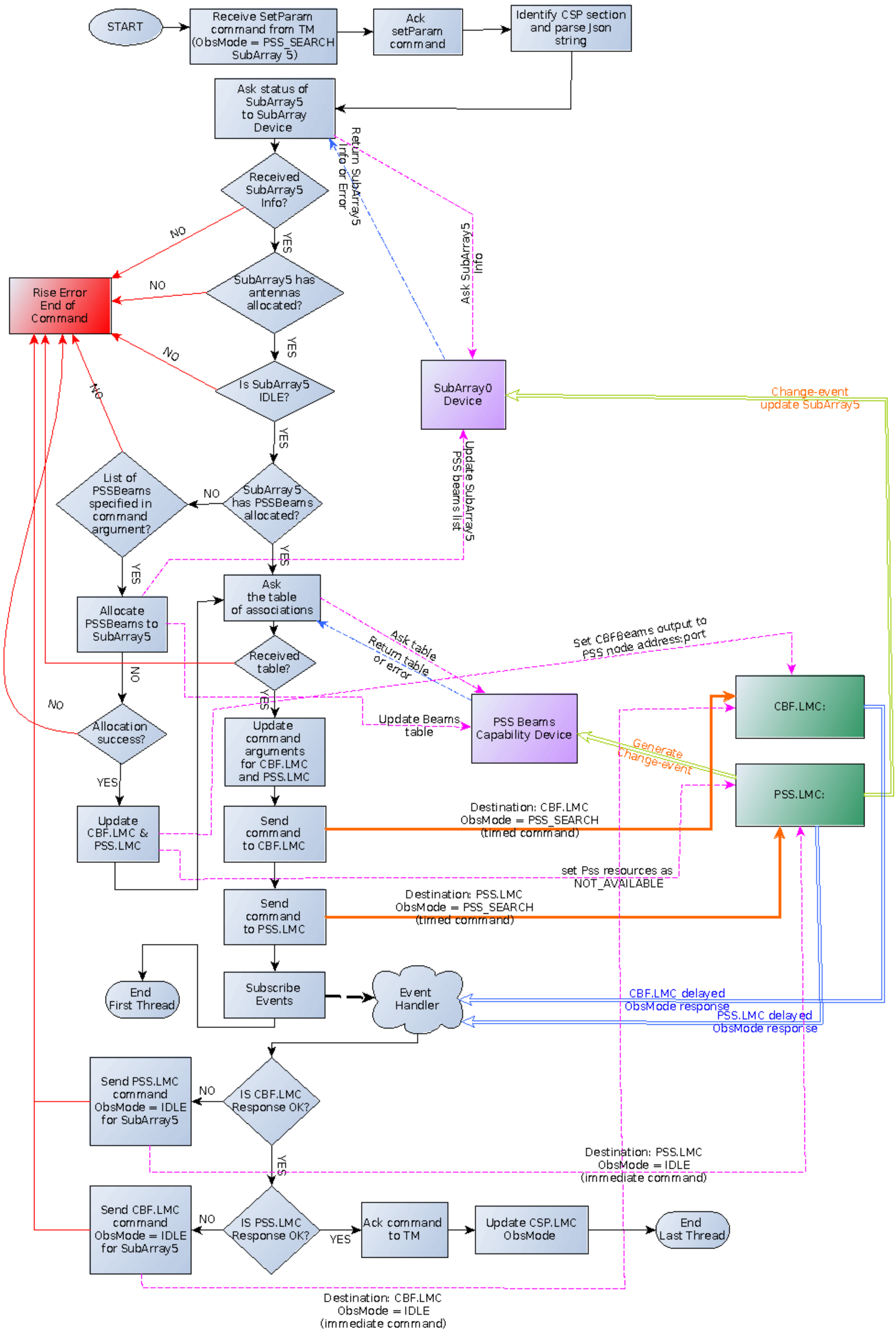

Figure 5: Flow of operations for set-up of a PSS Observation.

Appendix 1 : Mapping between SKA State and Mode and Tango state

In SKA environment there are some variables which define the global status of each Element. The main internal variable is the Operating State which has a direct mapping to the Tango State attribute.

The full list of SKA status variable is:

- Control Mode;
- Operational Mode;
- Administrative State/Operating State;
- Health State;
- Usage State;
- Simulated State;
- Sub-array State: this is applicable for sub-arrays only. It assumes only a sub-set of the Operational State values, so it can be implement in the same way as Operational State.

Here we propose a schema for the implementation of SKA status as Tango attributes.

Operating State (SKA) versus State (Tango) attributes.

In Tango, there are two variables which refer to the logical state of the device (State) and a string description of the current state value (Status). In SKA the corresponding value has the name of Operational State.

SKA Operational State has eight possible values, while Tango state has 14 possible values.

In Table 1 we propose a possible mapping between the SKA values to the nearest Tango ones.

We have two possible alternatives:

1. To implement the Operational State as a DevState Tango type. This approach guarantees the availability of the Tango State Machine, a facility to allow/deny the execution of commands when the Element is in a particular state.
2. To implement the Operational State as a Tango *short* type attribute. At present, we are going to implement into the CSP.LMC prototype.

   This choice permits the use the same SKA values for this attributes, using the C++ 11 way of enum declaration but, at the same time, we loses the Tango State Machine functionality.

SKA		Tango
OFF	This is a Powered off state.	**OFF**
READY	This suggests that the Element is ready to operate	**ON**
SHUTTING-DOWN	This is a transient state in which the Element is shutting down.	**MOVING**
HYBERNATE	Special non-operational state in which Entity has been placed after intialization. From this state it can transit to OFF or SLEEP.	**DISABLE**
SLEEP	Special non-operational state in which Entity has been placed to reduce power consumption. From this state it can transit to HYBERNATE or READY.	**STANDBY**
FAILED	An Element reports an 'Error' state when it detects a problem that affects its ability to accept certain commands or execute certain processes/operations.	**FAULT**
UNKNOWN:	TM is not aware of actual state of Element.	**UNKNOWN**
INITIALIZING:	This is a transient state in which the Element exists when it is starting up its processes.	**INIT**

Table 1: Possible correspondence between SKA Operating State and Tango State.

Status and Mode variables

In the SKA framework, beside the Operating State, there are 6 others main variables which describe the basic properties of an Element (and, lower level entities). <u>We propose to implement them as Tango attributes of *short* type</u>. The other alternative is to use the more natural enum types. We believe, however, that this feature, recently added to Tango 9, don't fit well to our scope.

As support to this choice we note that the ObservingMode attribute has to be implemented as a bit-mask type, because several types of observations can be done in parallel. So the overall observing status is the combination of different values. The Tango enum type can assume only consecutive values, so it can't be used to represent the value of a mask.
Another problem arise as Tango enum type cannot be declared as Array. In the CSP there are many examples of device whose instances should be handled in parallel, so we definitively needs array of enum.

Moreover, the Tango developers strongly suggest the use of Pogo as code generator of a Tango Device. We have verified that Pogo still handles enum type attributes in a way unsuitable for our purposes.

SKA			Tango name
Administrative Mode (alternative to operating Mode)	*Enabled*	Element can be administered	Administrative Mode
	Disabled	(Default) Normal Operations	
	Maintenance	Element under maintenance	
	Not-Fitted	the Element/sub-element is not fitted.	
Simulated Mode	*Real*	The element is a real hardware	Simulated Mode
	Local	The element is simulated	
Observing Mode	*Idle*	The element is idle.	Observing Mode MASK of bits
	Imaging	The element is observing images	
	PSS	The element is performing PSS	
	PST	The element is performing PST	
	VLBI	Performing VLBI observations	
	Transient search	The element performs *transient search*	
Test Mode	*Normal*	Element is in normal working condition	Test Mode
	Test	Element is under test	
Redundancy State	*Active*	Redundancy is enabled	Redundancy State
	Standby	Element is in standby	

Table 2: Proposed status variable

Health State	*Normal*	Element is in normal working condition	Health State
	Degraded	Element is functioning in degraded condition when subset of its functionality is compromised or unavailable.	
	Failed	Implies when there is major failure that prevents Element to perform its function.	
	NotOperable	Element is not available for observations due to missing dependencies.	

Table 3: Synopsis of proposed mapping of SKA Status and Mode variable to Tango Attributes.

Appendix 2: Investigate beams/capabilities

Here we try to investigate how the PSSBeams Capability and the PSS-Resources on PSS Master overlap in their description attributes.

The PSSBeams Capability is, for us, only a more organized way of seeing the elementary components of the PSS. For this reason we suggest to implement the PSSBeam Capability as a Tango Device that only reports information.

We discuss the Mid scenario, while the Low case can be extrapolated.

One PSS node can process from 2 to up 12 CBF-beams.

N PSS.MID nodes with their M parallel data processing pipeline form the N * M = 1500 PSS-Resources.

256 FPGA boards of the CBF.MID can form up to 1536 CBFBeams used by the PSS.MID.

A resource at PSS level corresponds to a single software data pipeline processing data coming from the associated CBF-beam.

Each PSS-Resource is identified by:

- a name corresponding to the Tango device name (running on a PSS node).
- an IP Address-port combination for data input
- a SDP IP Address-port combination for output products.
- A symbolic name, for instance: Node_RR_PCX where RR is the rack, PC is the PC sequential number, and X is the pipeline identifier

After initialization, the CBF.LMC and PSS.LMC communicate to CSP.LMC (asynchronously or on CSP.LMC direct request, TBD) the list with their available resources.

The association PSS-resource → CBF-beam is done by the CSP.LMC when, on TM request, it is asked to allocate a number of PSS resources to a sub-array.

This association represents what we call a PSSBeam.

Our plan is to implement the PSSBeams Capability as a single Tango Device that exports towards the Tango Clients a limited set of attributes (HealthState, ProgressStatus, …) and implements, as private attributes, 1500 instances of a PSSBeam Class which collects all the basic information to fully describe the PSSBeams Capability, as for example the table with the complete association between the PSS-Resources and the CBFBeams.

In our view, this device is comparable to an up-to-date register containing all the needed information about the PSSBeams.

The PSSBeams Capability Tango Device communicates with:

- CSP.LMC: PSSBeam Capability device works as a client. The CSP can read the Health Status of the PSS Beams and can ask the list with the association between PSSResourcess-CBF.Beams.

- CBF.LMC: the PSSBeam Capability device subscribes on it a number of attributes (HealthState of the FPGA boards) to get the updated value of the CBFBeams.
- PSS.LMC: the PSSBeam Capability device subscribes on it a number of attributes (HealthState of the PSSBeams and/or PSS Beams node, pipeline data processing progress...) to get the updated value of the PSSBeams.

Attribute	Pss-Resource	Pss-Beam capability
OperationalState	It is the operational state of the data pipeline	Capabilities do not implement the Operational State. It can be reported the information about the progress status of data processing.
ObservingMode	Is the observing Mode setted by the SubArray In this case we can have a combination of PULSAR_SEARCH and TRANSIENT_SEARCH	The same
Health Status	The health status is linked to the status of the harware/software processing the data **Normal**: if the parallel processor and the pipeline are ok **Not-operable**: the processor is not operable and/or the pipeline is not running **Degraded:** For example if the PSS pipeline can't ouput the products to SDP (?)	The health is a composition of: - the status of the hardware and software of the CBF FPGA boards - the status of the PSS node hardware and software - the status of the single parallel processor and pipeline software The PSSBeams Capability device has to subscribes a change-event both on the health status of the PSSResources and on the health status of the CBFBeams.
Administrative Mode	Is set by TM ENABLE DISABLED MAINTENANCE NOT FITTED The cases disabled and not-fitted has to be reported by PSS.LMC as PssResource not available. In this case the CSP can't use it to build the corrispondence. For maintance I don't know. If TM changes the administrative mode of a PSS resource, the list of PSSResources has to be updated	Read Only Same states as the PSSResource, except for Maintenance. This is a result of the the composition of two available resources. Proposal: Internally set to ENABLE if all components are ENABLED, else set to DISABLED

	and the CSP.LMC has to rebuild the list of available resources. The same for CBF beams.	
Usage State	IDLE : is not processing data USED: if Observing Mode is PULSAR_SEARCH and/or TRANSIENT_SEARCH	For a capability this status is reported by the ObservingMode. IDLE: if no observation required USED: if ObservingMode is PULSAR_SEARCH and/or TRANSIENT_SEARCH
Belongs to subarray/ Used by sub-arrays	Each PSS resource processes data coming from only one CBFBeam that belongs to only one subarray. It reports the ID of the sub-array	Each PSS Beam belongs to only one sub-array. It reports the ID of the sub-array.
Monitoring point	For a PSSResource the monitoring points are: - the parallel process temperature, voltage, frequency - the pipeline process status. A failure of the processor and/or the pipeline generates an alarm. If the node on which a PSS-resource data pipeline is running fails, an aggregated alarm has to be generated because in this case M resources fail all together. The failure is signalled to CSP.LMC that has to update the matching list between the available PSS-resources and CBF-beams.	The monitoring points in this case are the union of the monitoring points of the PSSResource and of the CBF Beam. Proposal: a link to low level summary of these.
Used by Capabilities	A PSS resource can be used only by one sub-array at time. Reports the ID of the sub-array	The same
List of command/messages in progress and in waiting or revoked commands	This list can be reported by the PSS.LMC that should implement a command queue. The single PSS-Resource does not implement any command queue capability	The same
List of components	The components of the node to which the PSS resource belongs.	The union of the components generating the PSS resource and the CBFBeam
List of active alarms List of implemented alarms		