Modular Experiment Control System packages for the CBM experiment

Pierre-Alain Loizeau[1,*]

[1]GSI Helmholtzzentrum für Schwerionenforschung GmbH, Planckstraße 1, 64291 Darmstadt, Germany

Abstract.
The Compressed Baryonic Matter (CBM) is a fixed-target experiment which will explore the QCD phase diagram through heavy-ions collisions using the beams from the SIS100 accelerator at FAIR. Its physics program characteristics led to a choice for a self-triggered and free-streaming data acquisition, followed by an online full reconstruction and selection chain in software. Such a system can operate reliably and efficiently only with a performant Experiment Control System (ECS) to ensure the synchronization and data quality of all sub-systems. The development of a CBM-specific Python based solution, focused only on the Experiment Controls and on the upper layer of Detector Controls (state and configuration propagation), was chosen after looking at existing solutions. It is divided in three packages, from an experiment independent modular core to user interfaces, in order to allow maximal quality checks of the core functions. This article presents the design choices for this ECS, the technical core package, the CBM ECS implementation package and the demonstrator GUI package based on it. All three packages are now available in demonstrator versions, with test coverage and typing coverage both above 90% for the core package. They will be deployed for validation in the CBM demonstrator, mini-CBM (mCBM).

1 Introduction

The Compressed Baryonic Matter (CBM) experiment at FAIR will explore the QCD phase diagram at high net-baryon densities through fixed-target heavy-ion collisions, using the beams provided by the SIS100 synchrotron in the energy range of 4.5-11 AGeV/c (fully stripped gold ions) [1]. The CBM physics program includes the measurement of rare probes with complex signatures, for which high interaction rates and a strong selection are needed to achieve the necessary statistics with reasonable storage needs. These requirements led to the technical decision for a self-triggered and free-streaming data acquisition, followed by online full reconstruction and selection chain in software. The experimental setup will consist of seven major detector systems, combined in various ways to target different physics (hadrons, di-electrons, di-muons). The ~3 million channels lead to an expected average data flow of 400 GB/s and peaks up to 600 GB/s [2]. By processing date only in software, the latency constraints are replaced by computing resource and software efficiency constraints.

The software stack on the data path will be composed of two levels: a first layer called FLES assembling and distributing so-called timeslices, fully contained data blocks with all

*e-mail: p.-a.loizeau@gsi.de

experiment raw data for a pre-selected period of time, and a second layer processing them up to event building and selection [2]. This is reflected in the computing hardware, with a first "entry" computing farm specific to CBM where the first layer will operate, connected by a set of optical fibers to the FAIR computing center, where a significant number of nodes will be reserved for CBM online processing during beamtime.

The software stack in the control path is the topic of the Experiment and Detector Controls (EDC) computing project within the CBM software organization. This project is less linear and monolithic than the data path, as it encompasses both the software elements controlling the detectors environment and supplies (typ. called Detectors Control Systems, DCS or slow controls), the software layers coordinating the accesses to the readout chain controls to avoid interfering with the data path, the central elements propagating high level commands and deriving states (typ. called Experiment Control System or ECS) and the auxiliary systems such as parameters storage or monitoring data storage, as well as all their interfaces.

The miniCBM (mCBM) experiment is a precursor setup for CBM, installed on a beamline of the existing SIS18 accelerator at GSI and meant to test and validate both the hardware and software of CBM. The setup is composed of demonstrator and/or pre-production elements of each major CBM detector system, with between 1 to 5 percent of the final channel count depending on the detector. A similarly scaled down version of the entry (DAQ) computing farm is available in a container close to the cave and partitions on the existing GSI batch-computing farm are used to emulate the future FAIR computing cluster. The already recorded and approved beamtime periods allow for a complete integration test up to attempts at online physics selection on the produced data. This is therefore where all elements of the EDC project will have to be validated before deployment in the CBM final setup.

2 CBM Experiment Control System

The CBM ECS, as the central coordination element in the control path, will have as main mission to ensure the reliability and efficiency of the experiment through the automatization of its configuration. This results in the following detailed missions: the determination and propagation of the experimental setup state (and therefore also its components states), the coordination of the flow of high and intermediate level commands, the propagation and resolution of system configurations as tags and sub-tags, the archiving of all changes to state and configuration. The first one will need a set of connected state machines, with the possibility for experts of any central (shared) system or detector to finely describe and control their specific hardware and software state, while still allowing to present a unified and simplified state to the non-experts operating the experiment during beamtime shifts. The second one is needed to ensure that commands from various user interfaces are properly propagated to specific targets and expanded without interfering between themselves or with automatic ones linked to the state machines. The third one is needed to avoid propagating full sets of parameter values within the software stack until the place where they are needed is reached, while allowing a human friendly, descriptive and flexible generation of the full setup configuration. The last one allows to remove any ambiguity about the run input parameters during data analysis, be it online, near-line or offline.

Existing frameworks were investigated at first to see if they could be taken as base, but they were either too costly, bound to specific operating systems or too large for CBM needs, with features already covered by other solutions in CBM. It was therefore decided to implement a CBM specific framework, focused on the ECS only.

The initial version of the ECS is fully developed in Python, while keeping open the option to convert part or all of it to a more performant language latter (e.g. C++). The ECS deployed

when CBM enters commissioning should be scalable from laboratory setups of individual detector systems to the full setup in the CBM cave. It should provide the flexibility to add and remove detector systems in an active state (outside of recording) to allow recovery of problematic detectors with minimal downtime. For the same reason, it should allow the flexibility to have parallel groups of detectors systems ("partitions") operating in an active or recording state, e.g. a main physics one and a commissioning or recovery one. Finally the ECS package(s) should possess a high level of code quality (QA), in order to maximize the recording efficiency and minimize doubts about it in case of errors or unexpected observations.

The current ECS prototype is made of three packages: a core package with base classes for state, command and configuration management, an implementation package with derived classes for all levels of the ECS and a Graphical User Interface (GUI) package.This allows to separate common parts of the code from application specific ones, with for example at some point one implementation for mCBM and one for full CBM, both relying on the same core. Different levels of QA are then possible, which can be verified both locally and through Continuous Integration (CI) when pushing changes to the GIT repository of each package. The following QA levels where targeted and achieved:

- All: clean lint checked with the flakeheaven tool [3]
- Core: > 90% covered with the pytest tool [4], > 90% typed with check by the mypy tool [5]
- Implementation: > 50-70% tested depending on class, > 90% typed
- GUI (QT [6]): no CI test, > 50% typed

The separation between the implementation package and the GUI package allows a flexibility in the choice of GUI framework(s), with eventually the possibility to have different GUIs connected to the same ECS instance. The separation between the core and implementation packages allows future upgrades of the core by expert software developers in a transparent manner for the applied package developed by detector experts and/or physicists. This could be for example a conversion of most of the core to C++ for performance reasons or updates of the external packages dependencies.

3 Core package

3.1 Design choices

A few technical specifications are defined beyond the usage of Python: ZeroMQ [7] (ZMQ) is to be used for all inter-process communication layers, and the code for commands management in ECS agents (emissions, reception, processing) should be separated from the one for state management (state machine(s), updates emission, reception and filtering) and the one for configuration management (tag request and resolution). The last technical specification is that the ECS is organized in multiple layers of processes called "Agents", each representing a system or group of systems, which form a so-called ECS topology through their interconnections. This provides flexibility in where each agent is deployed, ranging from having all ECS elements running on a single, dedicated ECS node to having a dedicated node for each ECS element. All other design choices are functional specifications.

On the command interface side, an agent executes only one command at a time (sequential execution). It always waits for a reply or timeout before emitting another command toward other agents, meaning that only one command is emitted per agent at a time. These two specifications aim at reducing instabilities or race-conditions in the state determination. A single agent may however still receive multiple commands from different emitters at close-by time. To address this, received commands are buffered and processed per emitter and command priority and per time of reception order. Finally, both commands and replies are routed from

source to destination to avoid the need for a-priori knowledge of the ECS topology when starting a given agent (see also Sec. 3.3).

For the state interface, state updates are emitted whenever the agent state changes, but can also be requested through a command. Each update is then broadcast to all connected agents, which can either make use of it or filter it out upon reception. The agent full state is composed of a pair of a "Global State" and a "Local State". The set of "Global States" is common to all agents and they are used to trigger cascading state transitions within the ECS topology. The "Local States" are specific to each agent and provide some degree of granularity within a given "Global State", aiming to reduce eventual configuration cycle or recovery time while hiding this complexity from the higher levels. A transition table is mapping in each agent two full States (source and target) to a function handle. It is dynamic as it can be modified at runtime through commands to add and remove transitions, as long as the agent is in the startup state (called "Uninitialized" in Sec. 3.4). Two auxiliary states are used to characterize the recording mode, while not being involved in the main state determination: a "Configuration state" and a "Configuration Stability state" (see also Sec. 3.4).

The Configuration is managed with so-called "Tags", which are specific to a given agent and should be human readable, unique identifiers for a given set of parameter values. Only tags are propagated and resolved through the ECS topology, up to the point where the actual parameters are needed. The tag resolution happens through a request-reply pattern with a configuration server, the request being a pair of an agent identifier and a tag.

Each of these three interfaces uses its own Python threads to optimize the response time.

The last set of specifications concerns the structure of the ECS topology and the relation between "Agent" and so-called "Subagent". An agent state is defined by default as the lowest common state of its subagents, with this rule meant to be expanded or replaced by each system developer when implementing the derived agent class for their own system. Agents track their subagents availability through special "ping" commands, which are periodically emitted if no communication (command or reply) was received from it for a selected period of time. Aside from an as-of-yet undefined parameter interface allowing to get the parameter values for a given configuration tag (to be used within the agent), the configuration tag of a given agent can be resolved to a list of tags for each of its attached subagents. Finally there is always a "top" agent defined as the default master in each topology, in order to avoid "orphan" agents which could not be controlled anymore.

3.2 Package structure

The implementation of the core is organized in four levels of inheritance, as represented in Fig. 1. The first level is a base class owning an instance of a logging interface class, an instance of a ZMQ context to be used by all ZMQ sockets in the derived classes, and generic flags and context variables which are are common to the three core branches (e.g. a context object representing an agent master information). The next level comprises three independent classes, all inheriting from the foundation class, which implement respectively the command interface of an agent (Command Core Agent), the state machine and state interface of an agent (State Core Agent) and the configuration tag interface (Config Core Agent). The third level is two derived class of the Command Core. The first one adds base commands providing testing, monitoring and agents relations (master-subagent) commands. The second one inherits from both the Command Core and the State Core and interfaces new commands to the existing methods of the State Core. The last level is a single class inheriting from both the Base Command Agent, the State Command Agent and the Configuration Core Agent. It therefore provides the three core functionalities in a single class and is meant to be the base class of the agents in all layers of the CBM ECS.

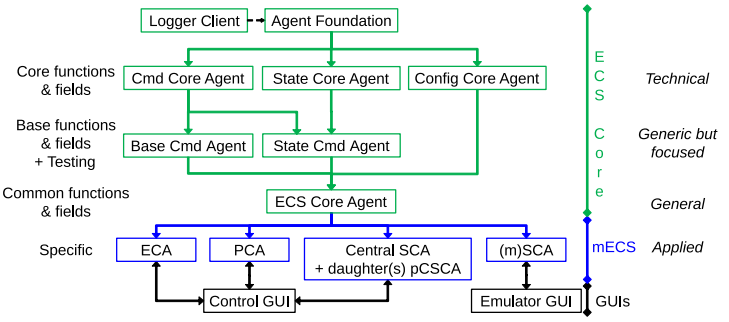

Figure 1. Packages and classes relations. See Section 3 for the core details, 4 for the mECS and 5 for the GUIs

Aside from this core, three single file executables provide central services, described in the next subsection, for each of the functionalities.

3.3 Central services

The three central services mirror the organization of the core Agent classes, with one for the command interface, one for the state interface and one for the configuration tag interface. Each service is using ZMQ for communication and is accessed by the Agent only using the methods from the corresponding "Core Agent" classes, as described in Fig 2.

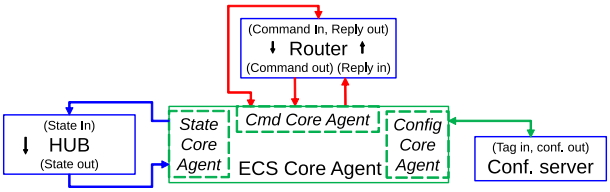

Figure 2. Central services interfaces with an ECS Core Agent instance

The Command Router is based on three "Router" ZMQ sockets and will asynchronously route received command to their target agents and received replies to the command emitter. This avoids needing a discovery mechanism for the command interface, as the only network information needed at startup by an agent is the address of the router and the ports of the three sockets. This feature enhances the capability to expand the ECS topology by adding agents on-the-fly. It also makes the ECS topology more robust against cascading agent failures, as each agent is directly connected only to the router which has limited failure modes.
The State Hub is based on one "PULL" and one "PUB" ZMQ socket for input and output, respectively. Each state update received on the input is broadcast to all agents connected to the output. Similarly to the command router, by avoiding direct connections between agents, it removes the need for knowing the complete agents network at startup and improves both the expansion flexibility and the robustness of the system. It also improves the reliability in case of bursts of state updates, as each update is buffered and published in the order it is received by the hub. The broadcast feature is needed to allow monitoring agents (e.g. GUIs) to operate in parallel to the more hierarchical control topology of agents.
The Configuration Server has the simplest agent-facing interface, as it is mainly using a single "REP" ZMQ socket. It may load on startup the available configurations for a variety of agents either from a JSON file (implemented) or from a database (still to be defined). Agents can then request either the list of agents for which at least one tag is known, the list of tags for a given agent or the content of a given agent-tag pair. The server also has an optional ZMQ "REP" socket for sending edition commands, to which agents may connect if needed.

Currently available commands are the addition and removal of tags as well as the dumping of the current known configurations to a new JSON file.

3.4 Default states

The state sequence provided by default in the core package is shown in Fig 3. The Uninitialized state is the initial state, where editing the transition table is still allowed, and no guarantee is provided regarding the current state of the system represented by the agent. During the transition to "Initialized", the agent should ensure that all required resources are available and establish all required connections to hardware or software elements. In the "Configuring" state, the system controlled by the agent is prepared up to a point where it is either generating (detectors), transporting (FLES), processing (online farm) or supporting (supply systems) a data flow. This could be seen as a transition but is made equivalent to a state due to its potentially macroscopic duration and eventual internal "Local" states (see end of this section). The "Active" state is the one of a running system, ready for full operation of the experimental setup. An additional "Recording" state is used for the two systems were it is relevant: FLES (raw data flow) and online processing (processed and selected data flow). The "Error" state possesses a sub-state linking to the original state, allowing either recovery to it or cascading to a lower error state, down to the level where a system is back to "Uninitialized".

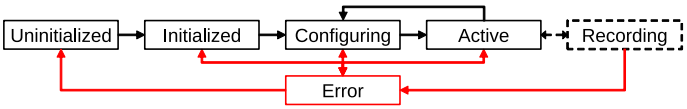

Figure 3. Default global states and transitions

As mentioned previously the state machine is implemented through a transition table which can be edited when in the "Uninitialized" state, which is the only one which cannot be removed. There is however for now no check for dead-ends or "final" states.

Two additional states are used to determine the expected data quality when entering and exiting the "Recording" state. The "Configuration state" represents how the current configuration and parameters relate to the ones linked to the currently set configuration tag (e.g. "clean tag", "user edited", ...). The "Stability state" represents how strict the ranges in which automated processes are allowed to tune the current parameters are, which for example could be a large window during detector commissioning and a smaller one during standard operation. This feature should allow a classification of recording runs on a scale from direct usage to cautious analysis. Table 1 shows an example with two values for each state, leading to four grades for the runs (eight if state at beginning and end of run taken into account).

By default no local states are defined and all agents full state are defined as the pair "Global State, undefined". Each agent can however add its own local states, for example expanding the "Configuring" state into multiple sub-steps corresponding to the various hardware levels in the readout chain (data aggregator, opto-converter, front-end, ...) in order to perform only necessary work when reconfiguring from the "Active" state.

		Configuration State	
		Manual	Clean
Stability State	Unstable	Technical	Unstable
	Stable	Dirty	Production

Table 1. Example of auxiliary states values and derived recording states

3.5 Test topology

Each element in the core package is tested using variations of the test topology shown in Fig. 4. For example tests of the State Core Agent are done with a single instance of the

class and an instance of the Hub, while tests of the Command Core Agent are done using an instance of the Command Router and between one and three layers of instances of the class. The final validation test of the package in the CI, which is required for approval of any merge request, is done with the full topology and instances of the ECS Core Agent class.

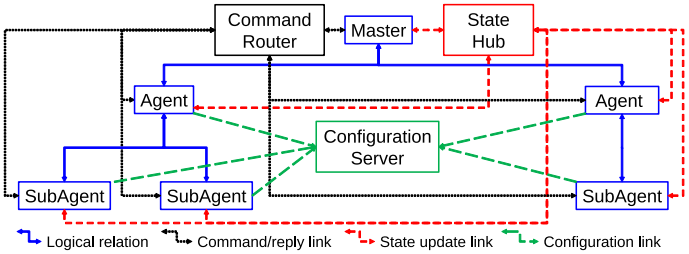

Figure 4. Agents and services topology for testing

4 CBM ECS prototype implementation package

The minimal topology for a running ECS comprises the following agents, each being an instance of a class derived from EcsCoreAgent: one Experiment Control Agent (ECA), as many Central System Control Agents (CSCA) as needed, one Partition Control Agent (PCA), one System Control Agent (SCA), one partition CSCA (pCSCA) for each SCA. The number of SCAs is dynamic and defined by the detectors present and the topology. Each SCA can participate in only one partition. The number of PCAs is dynamic. The number of pCSCA is dynamic and, as each represent the share of a given CSCA resource assigned to a PCA, equal to the number of PCA times the number of CSCA.

The corresponding derived classes provide the implementation of the state transitions for each of these agents and are gathered in the prototype implementation package, called for now "mECS implementation" (mECS standing for "mCBM ECS"). For now only generic versions of the CSCA, pCSCA and SCA are available (base classes), as shown in the package class relations in Fig 1. Each of the four central systems are then using instances of the plain CSCA and each of the seven detector systems are using instances of the plain SCA.

As an example, Fig. 5 shows a two partitions topology with a single central system and one detector system per partition.

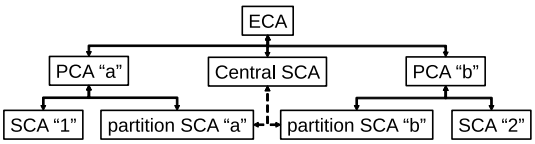

Figure 5. Example of minimal two partitions topology

In a second phase derived classes of the SCA will be introduced for each of the detectors and central systems present in mCBM, allowing to build a topology dedicated to the control of this setup. For most of mCBM usage, a single partition will be used given the reduced size of the setup. The experience gained with this topology will be used to create a second generation of derived classes, dedicated to the systems deployed in the main CBM setup. This CBM topology will be used with various partitions to support the installation phase, the commissioning phase and the first data taking with beam.

5 Demonstrator/Emulator GUI package

The last package needed to validate the prototype is a set of GUIs with two main usages: as a demonstrator where all interactions between agents can be tested in a wider range of states

and time sequences, and as an emulator where the connection to the actual systems (state transition execution) is replaced by some user interactions. In the first case all state transitions in the CSCAs and SCAs are replaced by simple few seconds sleeps (automatic transitions), while in the second case pop-ups should be presented to the user to decide between ignoring the transition request, failing it (error state) or accepting it.

The GUIs are implemented using the "QT6" library with the "Pyside6" Python bindings and are all derived from the ECS Core Agent class, but with an empty state table. One GUI is prepared to control the ECA, all four central systems and the eventual partitions, see Fig 6 left. It allows to validate the effect of adding or removing systems in a partition in various states, the proper listing, choice and application of configuration tags from the top level to the lowest one and some dependencies between states of some systems and the actions available in others (e.g. no partition creation while the central systems are not "Active"). The second GUI handles a single SCA and is meant as the interface presented to the detector system shifters when operating mCBM, see Fig 6 right. It is the place where the emulator and demonstrator modes are implemented. The selection of a given mode is done by a startup flag of the GUI, which is propagated to the underlying agent. Both GUIs can be disconnected from the agents running in background processes and reconnected to them, including when running on different nodes. The relation between these two GUIs and the implementation classes can be seen in Fig 1.

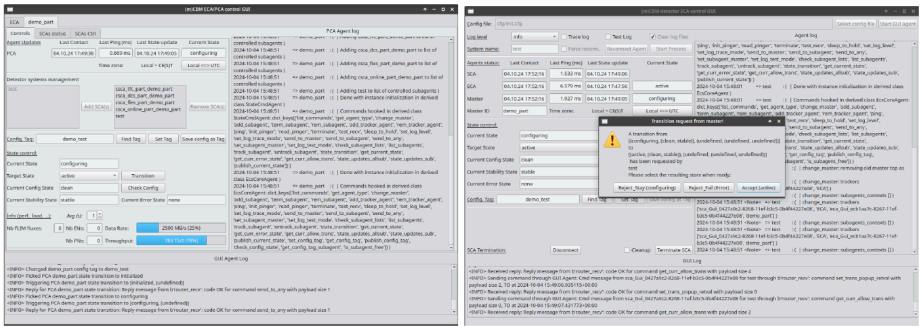

Figure 6. Screenshots of the two GUIs: left for the central elements, right the SCA

As all detector systems will not have their SCA implemented at the same time, the mCBM ECS will operate in a mixed mode for beamtimes in the coming years, with some detectors having full-feature automatized agents and others the user pop-up placeholder. The deadline for each detector SCA will be their installation and commissioning in the CBM cave.

6 Conclusion

A framework prototype for the CBM ECS is implemented and available in the CBM git repository [8] as three packages depending on each other. The "ecs-core" package provides the base classes for the three main components of an ECS, which are a command interface, a state machine and state interface and a configuration interface. These three components are combined in a class called "EcsCoreAgent" which should be the base of all elements in the CBM ECS. This package is following strict QA goals to minimize uncertainties when including it as dependency, with 92% testing coverage. One of those tests ensures that the command core can emit, receive and process at least 100 *pings* (minimal execution) per second between two agents. An "implementation" package oriented toward an ECS for mCBM

is built upon it and provides default classes for the four types of agents composing a CBM ECS: Experiment Control Agent, Partition Control Agent and System Control Agent for both central and detector systems. The last package provides GUIs built using QT6 and allowing both to check the agents usage flow (demonstrator) and to help with the operation of the next mCBM beamtime campaign (emulator).

The next step for the validation of the prototype will be a first deployment for the mCBM 2025 spring campaign, with only a few central systems implementing their state transitions. It will be followed by the integration of all detector systems in 2025 and 2026 through a gradual implementation of their SCAs, with mCBM used as test-bench. The first stable release version for CBM is expected at latest in 2027 for the installation and commissioning phase.

References

[1] T. Ablyazimov, A. Abuhoza, R.P. Adak et al., Challenges in QCD matter physics – The scientific programme of the Compressed Baryonic Matter experiment at FAIR, Eur. Phys. J. A **53**, 60 (2017). 10.1140/epja/i2017-12248-y

[2] J. de Cuveland, D. Emschermann, V. Friese, I. Fröhlich, P. Gasik, D. Hutter, W. Müller, C. Sturm (CBM Collaboration), Technical Design Report for the CBM Online Systems – Part I, DAQ and FLES Entry Stage, CBM Technical Design Report (GSI Helmholtzzentrum für Schwerionenforschung GmbH, Darmstadt, 2023), https://doi.org/10.15120/GSI-2023-00739

[3] flakheaven in pypi, https://pypi.org/project/flakeheaven/ (2024), accessed: 2025-04-29

[4] Pytest in pypi, https://git.cbm.gsi.de/ecs (2024), accessed: 2025-04-29

[5] mypy in pypi, https://pypi.org/project/mypy/ (2024), accessed: 2025-04-29

[6] Pyside6, qt6 bindings in pypi, https://pypi.org/project/PySide6/ (2024), accessed: 2025-04-29

[7] Zeromq bindings in pypi, https://pypi.org/project/pyzmq/ (2024), accessed: 2025-04-29

[8] CBM, ECS group in the CBM Gitlab repository, https://git.cbm.gsi.de/ecs (2024), accessed: 2025-04-29