OPERATIONAL SEQUENCER FOR ESS FACILITY

M. Celary*[1], A. Gorzawski[2], D. Spruce[2]
[1]S2Innovation, Kraków, Poland
[2]European Spallation Source, Lund, Sweden

Abstract

The Operational Sequencer is a software framework developed to automate complex and frequently executed high-level procedures that are essential to the operation of accelerator systems at the European Spallation Source (ESS). By transforming manual control room tasks into predefined sequences, the tool enhances process repeatability, reliability, and safety by reducing human error.

Designed within the ESS Integrated Control System Software group, the sequencer provides a unified architecture for defining, executing, and monitoring operational tasks. It supports multiple task execution types, scalable system design, and an intuitive graphical interface to facilitate operator oversight.

This paper presents the architectural concepts, implementation challenges, and scalability strategies of the Operational Sequencer, as well as lessons learned from its deployment during ESS commissioning. The framework represents a significant step toward efficient and reproducible operations, reducing both operator workload and time-to-beam for scientific experiments.

RATIONALE

The concept of an operational sequencer for ESS has been discussed for several years. In the earlier stages of beam production, only a limited part of the accelerator was in operation. As a result, the overall complexity remained manageable and many activities could be carried out manually. In this context, existing tools such as Save and Restore were sufficient for tracking configuration settings and machine states.

As the scope of operation expands, there is a growing need to organize higher-level procedures composed of actions typically performed by operators. These actions may include setting process variables (PVs), executing sequences available within the EPICS control system, or incorporating steps that require manual intervention. The Operational Sequencer is intended to support this type of structured workflow.

Experience from other facilities shows clear advantages of applying automated sequencing to common accelerator operations. Previously, procedures were often documented as detailed written instructions describing how to interact with GUI interfaces and which parameters to set in a specific order. Such an approach is susceptible to human error, as individual executions may deviate from the intended procedure, making reproducibility difficult to ensure. The Operational Sequencer provides a way to address these issues.

* mateusz.celary@gmail.com, linkedin.com/in/mateusz-celary/

TECHNICAL DESIGN AND IMPLEMENTATION

Existing Alternative Solutions

Numerous discussions during the design phase considered several alternative approaches. Among them were the "classic" EPICS sequencer [1] and the more recently introduced oac-tree developed at ITER [2]. The EPICS sequencer operates at the IOC level, with state transitions driven by changes in process variables. While effective for local control logic, this model is less suitable for procedures spanning multiple IOCs and large numbers of PVs. Its reliance on the State Notation Language (SNL), which must be compiled and deployed to IOCs, also makes it less convenient in environments where operational procedures evolve frequently. In addition, it does not support pvAccess, the preferred protocol at ESS.

Another commonly used alternative is to implement procedures as custom Python scripts or Jupyter notebooks. Although such solutions are flexible and can be developed quickly, they are generally more difficult to review, validate, and maintain in an operational setting. In practice, their logic is often primarily understood by the original author, which can introduce an unnecessary divide between technically specialized operators and regular users. This also makes it more difficult to assess whether a given implementation correctly reflects the intended operational procedure.

The oac-tree [2] represents a recent addition to the EPICS software ecosystem. Although it is not currently used in ESS operations, it may still offer value as an intermediate orchestration layer for coordinating sequences across multiple systems.

The ESS Operational Sequencer was therefore designed as a higher-level solution operating across multiple applications and beyond the strict EPICS layer. This makes it possible to integrate different sequencing approaches within a single operational framework and to combine their advantages in a manner better suited to the needs of ESS.

Architecture

The software architecture (Fig. 1) was intentionally kept simple and minimalistic, designed to satisfy the minimum known requirements without over-engineering. Sequence definitions are written in YAML with optional Jinja2 templates. This is a well-established format used by tools such as GitHub Actions, GitLab CI/CD, and Ansible. It is readable, parameterizable, and widely recognizable, which made it a natural design choice and allowed us to avoid creating a dedicated sequence definition language.

PREPRINT

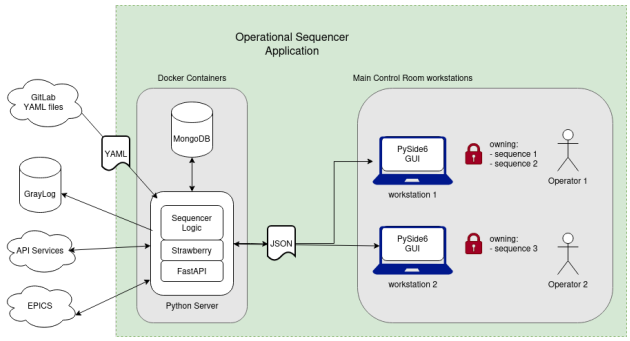

Figure 1: Architecture of the operational sequencer application. Lock icons indicate a single operator's ownership of the selected task. Connection to external services is indicated.

A GitLab repository was used to host the sequence definitions rather than a relational database, whose schema would inevitably require challenging migrations as the model evolved. GitLab provides built-in review workflows, allowing shift leaders to approve new sequences without additional tooling.

The nested YAML structure is mirrored throughout the full stack: a recursive task class tree in the backend, MongoDB for persistence, GraphQL as the transport layer, and a *QTreeView* widget in the frontend, which makes it easier for the developers to conceptualize the application layers.

Python serves as the single language for both client and server, keeping the codebase compact and maintainable. The frontend is implemented in PySide6, which provides a Qt-based Python interface comparable to PyQt while offering a more flexible licensing model. FastAPI was chosen for the backend due to its combination of high performance, clarity, and ease of development, which makes it well suited for building maintainable service APIs.

The backend represents sequences as a uniformly traversable tree, allowing operators to start, skip, or reset any individual task within a running sequence — including in deeply nested structures — without special-casing. The GUI follows an asynchronous, signal-driven design with a central controller as the sole coordinator (the application is following the MVVC pattern); widgets initialise lazily on first data arrival and incoming state changes bursts transported by web sockets are coalesced before rendering, keeping the interface responsive, even when operating on the sequences spanning thousands of nodes.

By adhering to the Open/Closed Principle of SOLID design, adding new task types and features as the application matured remained straightforward. Challenges arose only from larger refactoring efforts that addressed intentional shortcuts taken during the MVP stage.

Implemented Task Types

The main building blocks of the Operational Sequencer sequence definitions are "Tasks", which can be understood as basic actions performed by the system. Tasks constitute the sequence tree, which can have a nested structure. The task types that are currently implemented include:

- *PVPut* (to write to process variable(s) predefined values)
- *PVCheck* (to check an arbitrary mathematical condition on any number of process variables)
- *UserDialog* (displaying a communicate to the GUI user, with an option to confirm)
- *RestoreSnapshot* (to restore the previously saved state of multiple process variables at once)
- *CheckSnapshot* (to check if restoring from Save and Restore was successful or if the process variables didn't deviate from required values)
- *OLog* (create a generated entry in the Main Control Room electronic logbook service)
- *Include* (allows to include sequences from other files and render the parametrized templates)
- *WaitForCondition* (waiting for the condition to be met, e.g., waiting until the temperature is low enough)

This relatively small set of task types enables sophisticated hardware control and automation of most Main Control Room system interactions. However, the Operational Sequencer is a flexible framework that allows its capabilities to be extended quickly by introducing new task types. Some additional task types, developed as a proof of concept and not used at ESS, support basic interactions with the Tango Controls ecosystem.

User Requirements and Interface

From the beginning, the development process was guided by user needs and continuous feedback, which helped shape an application adapted to the specific working conditions of the Main Control Room.

The GUI (Fig. 2) was designed with readability prioritized over modern visual design, featuring large, clear icons, a live overview of the running sequence, and color-coded execution states that are intuitive to interpret. There is also a log area and task details area, which displays information about the selected task, such as its definition file. This provided operators with a much-needed level of situational awareness, even when the screens were viewed from a distance.

USAGE IN OPERATION

The operational sequencer became a permanent part of the Main Control Room toolset. A stable version is deployed to all workstations and has been extensively used during the commissioning run to deliver Beam on Dump. The automated procedures include:

- Accelerator startup, with the beam sent to the first destination.
- Accelerator Shutdown,
- RF and Klystrons startup/shutdown,
- Emergency RF shutdown,
- Insertable devices movement

The clearest demonstration of usefulness of the operational sequencer is the beam startup procedure. Beam startup

PREPRINT

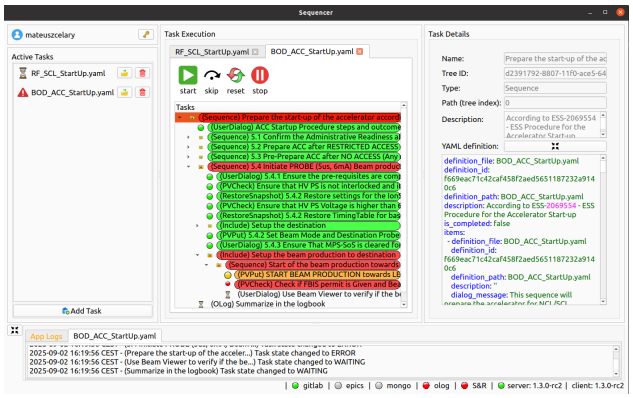

Figure 2: Screenshot of the GUI for the operational sequencer application. Colors of the tree entries indicate the state of the task execution.

can now be completed in less than an hour, although part of the process remains manual at the operators request. Previously, this procedure typically took more than four times longer and was often prone to operator error.

CONCLUSIONS AND FUTURE DIRECTIONS

The operational sequencer has established itself as a recognized and widely adopted tool in the Main Control Room, now following a regular release workflow. Its reception among operators has been positive, and usage has grown steadily since the initial deployment. A further testament to the application's growing recognition at ESS is the expansion of the team working on it, bringing in new contributors and sustaining development momentum - a natural sign of a project gaining institutional maturity.

The architecture has also demonstrated its long-term value. New task types and features were added with little friction, in line with the Open/Closed Principle that guided the original design. Even the migration of components from a more synchronous to a fully asynchronous execution model was carried out smoothly, confirming that the architectural decisions made early on were both sound and resilient to evolving requirements.

Looking ahead, there remain many potential applications of this software beyond the ESS use case. Since the concept is control system agnostic, other control systems e.g. Tango could be integrated with minimal effort. The GUI could also be embedded into an existing monitoring tool, with the server logic serving as a backend API. Creation of auxiliary tools such as a task tree editor has been suggested as well. All this makes the operational sequencer a compelling solution for other accelerator or industry facilities, and we encourage its adoption.

REFERENCES

[1] EPICS Sequencer, `https://epics-modules.github.io/sequencer/index.html`

[2] OACTree git repository, `https://github.com/oac-tree/`.

PREPRINT