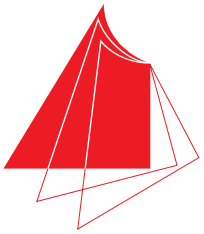

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES


Karlsruhe University of Applied Sciences
Faculty of Computer Science

Bachelor's Thesis

# Evaluation of Technologies for a Future Run Control System for the CMS Experiment at CERN

Author

Philipp Brummer
orcid.org/0000-0002-3857-3504

Contact

brph1014@hs-karlsruhe.de
philipp.maximilian.brummer@cern.ch

CERN-THESIS-2018-475
//2018

Supervisors

Prof. Dr. Thomas Fuchß
Hochschule Karlsruhe

Dr. Hannes Sakulin
CERN

February 2018

**Abstract**

The Run Control and Monitoring System (RCMS) of the Compact Muon Solenoid (CMS) experiment at the European Organization for Nuclear Research (CERN) is responsible for controlling and monitoring the data acquisition operations of the experiment [52]. The RCMS provides a high-level web-based interface to operators, allowing control of the experiment and the subsystems it is comprised of. As part of the RCMS, the Run Control provides a framework for developing Function Managers, user code that is implemented in accordance with the interfaces and behavior defined by the framework. The Run Control system is also responsible for executing said Function Managers in a distributed environment and handling communication between them and the resources they control.

The original Run Control and Monitoring System for the CMS experiment was developed to fulfill the requirements of the experiment for Phase-I and the first run of the Large Hadron Collider (LHC). Since then, various changes were necessary to keep up with the requirements for Run 2. The core of the system being over a decade old, making use of more recent developments in software technology, improved libraries and programming language features may simplify the implementation of the Run Control, making a rewrite or major update of the RCMS for Phase-II and Run 4 [6] of the LHC stand to reason. For this purpose, the evaluation of suitable technologies for a future Run Control System for the CMS experiment is required. This thesis concerns itself with documenting the functionality of the current RCMS, analyzing new requirements and possible improvements and evaluating technologies regarding their suitability to implement a future version of the Run Control system. To ease the comparison with the technologies currently in use, small demonstration projects and prototypes are implemented.

**Zusammenfassung**

Das Run Control and Monitoring System (RCMS) des Compact Muon Solenoid (CMS) Experiments an der europäischen Organisation für Kernforschung (CERN) hat die Aufgabe, die Datenakquisition des Experiments zu steuern und zu überwachen [52]. Das RCMS stellt eine webbasierte Schnittstelle auf hoher Ebene zur Verfügung, welche es Anwendern ermöglicht, das Experiment und seine Teilsysteme zu steuern. Als Teil des RCMS, stellt das Run Control Framework ein Rahmenwerk dar, das zur Entwicklung von Function Managern dient. Function Manager bestehen aus benutzerdefiniertem Quellcode, welcher ihr Verhalten definiert und sind in Übereinkunft mit den Schnittstellen des Frameworks implementiert. Das Run Control System ist ebenfalls für die verteilte Ausführung von Function Managern zuständig, insbesondere der Kommunikation zwischen Function Managern und den Ressourcen, welche von ihnen gesteuert werden.

Das ursprüngliche Run Control and Monitoring System des CMS Experiments wurde entwickelt, um die Anforderung für Phase-I und den ersten Run des Large Hadron Colliders (LHC) zu erfüllen. Seitdem waren vielseitige Änderungen am System notwendig, um den Anforderungen für Run 2 gerecht zu werden. Der Kern des Run Control Frameworks ist mehr als zehn Jahre alt und es besteht die Hoffnung, dass neuere Entwicklungen in der Software-Technologie, verbesserte Programmbibliotheken und Verbesserungen in der Funktionalität von Programmiersprachen erlauben, die Implementierung des Run Control Frameworks zu vereinfachen. Eine grundlegend neue Implementierung oder weitreichende Änderung am RCMS für Phase-II und Run 4 [6] sind denkbar und liegen nahe. Zu diesem Zweck ist es notwendig, geeignete Technologien für eine zukünftige Version des Run Control Systems zu evaluieren. Diese Abschlussarbeit dient der Dokumentation der Funktionalität des aktuellen RCMS, der Analyse neuer Anforderungen und möglicher Verbesserungen, sowie der Evaluation von Technologien in Anbetracht ihrer Eignung für die Implementierung einer zukünftigen Version des Run Control Systems. Zur Vereinfachung des Vergleichs mit den Technologien die sich momentan im Einsatz befinden, wurden kleine Demonstrationen und Prototypen implementiert.

# Contents

# Chapter 1

# Introduction

This chapter summarizes the context of the thesis by giving an overview of the experiment and the Run Control system's history. The work described in this thesis is conducted as part of the CMS experiment's Data Acquisition (DAQ) group.

## 1.1 Environment

The Run Control and Monitoring System is responsible for providing the tools required to control and monitor the data taking of the Compact Muon Solenoid experiment.

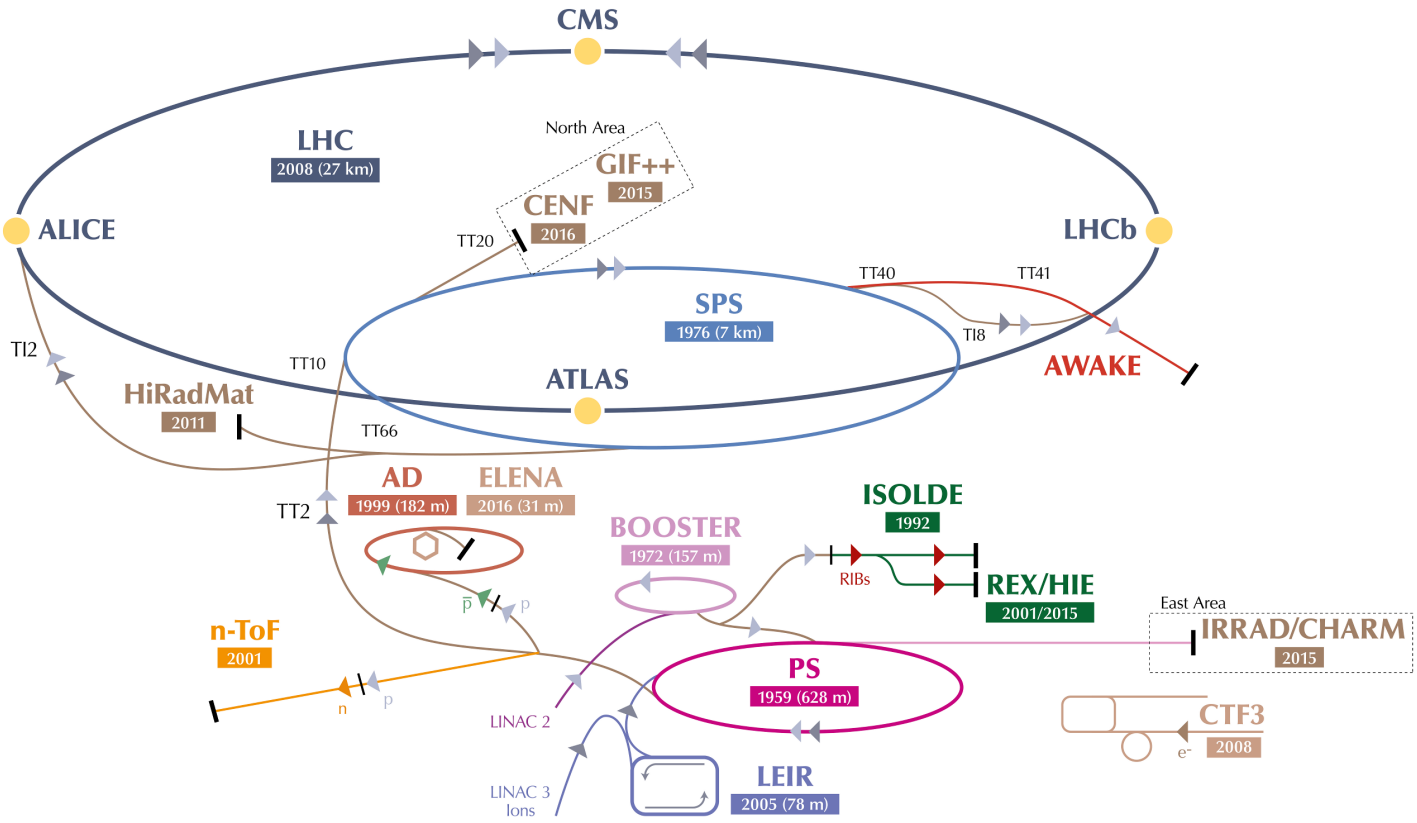


Figure 1.1: CERN Accelerator Complex [66]

### 1.1.1 The LHC at CERN

The Large Hadron Collider (LHC) at the European Organization for Nuclear Research (CERN) is a ring-shaped particle accelerator. In operation since September 2008, its purpose is to provide high-energy particle beams, travelling close to the speed of light. Two beams travel in opposite directions and are made to collide within the four detectors located at the LHC: ATLAS, CMS, ALICE, LHCb. The particles are accelerated and guided within the 27-kilometer long ring by the use of superconducting electromagnets. The environment required to achieve the desired particle energies includes an ultrahigh vacuum and magnet cooling to below negative 270 degrees Celsius [9].

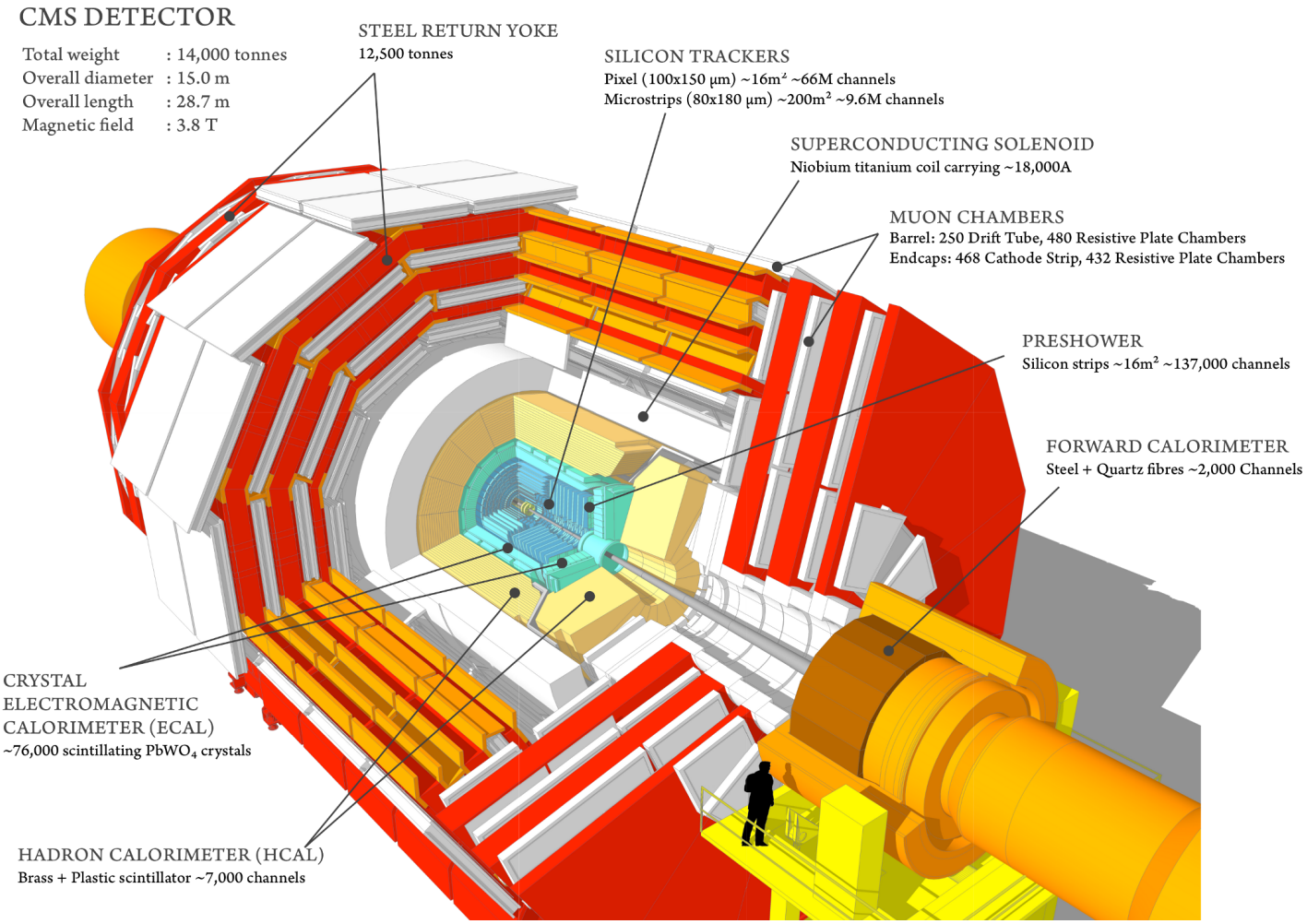


Figure 1.2: CMS Detector [11]

### 1.1.2 CMS experiment at the LHC

The Compact Muon Solenoid (CMS) detector is a general-purpose physics detector situated at one of the four collision points of the LHC. The 14,000-tonne detector acts as high-speed camera, capturing the energetic properties of the particles created by the beam collisions nearly 40 million times each second [12]. Various sub-detectors are responsible for measuring different properties of the particles produced during these collisions.

### 1.1.3 CMS Data Acquisition

The Data Acquisition (DAQ) of the CMS experiment concerns itself with taking and processing data from the 55 million (Run 1) readout channels of the experiment. Its main task is to reconstruct collision events using the data provided by the experiment's sub-detectors. With beams colliding at a rate of close to 40 million times each second, which translates to almost one billion particle collisions per second, new particle collisions happen every 25 nanoseconds, resulting in the particles from the previous collision not yet having left the detector when the next collision takes place. This makes it necessary to correlate the measurements from the different high-resolution sub-detectors [89].

While the experiment produces around 40 terabytes of data each second, not all of which is read out, a two-tiered trigger system samples the measurements and filters out only the most interesting events. In order to handle these high rates of data, the Level-1 (L1) Trigger is implemented as custom-built hardware and cuts the number of events down from around one billion to 100,000 (corresponding to 100 gigabyte at 1 megabyte event size) per second. The Level-2 (L2) Trigger or High Level Trigger (HLT), software running on thousands of ordinary computing cores, reduces the number of events further, down to a manageable 1000 events per second [89].

## 1.2 Run Control and Monitoring System

The Run Control and Monitoring System (RCMS) is responsible for configuring, controlling and monitoring the CMS experiment's data taking. It provides a high-level view of the experiment and allows operators to issue commands to the detector's and its sub-detectors' data acquisition systems.

### 1.2.1 Run 1

The first version of the RCMS was designed and implemented to meet the experiment's requirements for the first run of the LHC in 2008 [54]. Based on the GRIDCC [65] project, a preliminary design of the initial version of the RCMS was presented in 2003 [55]. A web-based solution making use of the Java Enterprise platform, Axis web services and JSP was chosen, as the requirements for the Run Control system were identified to be similar to those of modern web applications. In order to account for the thousands of resources needed for the DAQ system, the DAQ Configurator was introduced to automate the creation of configurations based on a description of the system's components [75].

### 1.2.2 Changes for DAQ2 (Run 2)

In order to accommodate for the changes made to the DAQ system for Run 2 in 2015 [6], the RCMS was adapted to the changed hardware configuration database format. This led to the creation of the DAQ2 Configurator, a tool based on the original DAQ Configurator and now external to the Run Control framework [77].

### 1.2.3 Plans for Run 3

For Run 3 in 2021 [6], no major changes in the RCMS are planned. Development will be focused on improving the monitoring and expert tools. Changes and fixes will be implemented as needed. A partial refactoring of the RCMS based on the conclusions of this thesis is possible. [76]

### 1.2.4 Phase-II and Run 4

For Phase 2 and Run 4 in 2026 [6], a redesign and new implementation of the RCMS is feasible. This thesis will concern itself with both a new implementation for Run 4 as well as the modernization of RCMS components for Run 3.

# Chapter 2

# The current Run Control System

This chapter describes the Run Control and Monitoring System (RCMS) as it was in use for Run 2 of the LHC. The RCMS is comprised of the Run Control Framework, tools for configuration management and various databases. The information presented in this chapter is based on the RCMS papers [75, 55, 54, 77] and the RCMS code [53], as well as the author's experience as a RCMS developer.

**Run** A Run identifies a state in which all active components of the experiment are ready to take data and ideally lasts for a couple of hours for a complete LHC fill. When the system in is a running state, the DAQ system processes and records the information it receives from the sub-detectors. A Run is not required to include all the experiment's Subsystems, which is why the Run Control discriminates between the Subsystem modes In, Out and Away, meaning that a Subsystem is included in the Run, Out of the Run or not included in the Configuration respectively. Run Control Runs are not to be confused with LHC Runs, which are used to describe the Large Hadron Collider's schedule [6].

## 2.1 Configurations

A configuration resource usually describes a service, on which machine it is run and includes an URL that can be used to access the service. It also contains various properties related to the service's configuration. The resources of the DAQ are organized in hierarchical configurations, describing the relationships between them. Configurations are used to both describe the hierarchy of Function Managers as well as the relationships between Function Managers and their child resources.

**Resource Service** Configurations are stored in the Resource Service database (described in section 2.4.2) and can be managed using the RS3 Manager (2.3.1). In addition, configurations for the central DAQ (CDAQ) are generated using the DAQ2 Configurator (2.3.2). The resource service's structure is similar to that of a file system: Configurations are stored in directories and are identified by their unique path. The Resource Service 3 stores a history of versions for each configuration.

**Global Configuration Maps** Global Configuration Maps (GCMs) are used to map configurations to a role. This is done to allow a Function Manager to specify a child FM as a subsystem by only specifying the role of the child FM. When the configuration is

started by the Run Control, the configuration's GCM key is used to find the actual resource registered for the subsystem's role. Changing a subsystem resource's configuration becomes as easy as registering a new configuration for the same role in the GCM and therefore does not require the parent configuration to be edited. GCMs are persisted in the GCM database (2.4.5).

## 2.2 Run Control Framework

The Run Control Framework is a framework for the development of Function Managers (FMs) and a platform for distributed execution thereof. It is based on SOAP web services, written in Java and runs on the Apache Tomcat software.

### 2.2.1 Function Managers

A Function Manager (FM) describes a piece of software which provides interfaces and implements behavior according to the requirements and conventions of the Run Control framework. FMs are distributed as Java Archive (JAR) files and are loaded by the Run Control System on the Function Manager's instantiation. The logic implemented by Function Managers may vary - from a high-level overview such as the Level-0-FM to communication with XDAQ applications. Function Managers are implemented by the DAQ group as well as by groups responsible for the experiment's subsystems. Function Managers may control a variety of child resources, usually other function managers or remote applications using the XDAQ online software.

Levels are assigned to Function Managers depending on their position in the hierarchy of the central DAQ setup's configuration. The Level-0-FM (2.6) is the top-most Function Manager and provided by the DAQ group.

**State Machines** Function Managers implement a state machine which is registered with the Run Control instance executing them. The state machine definition follows certain conventions and includes states, inputs and transitions. Since the state machine definition is part of the Function Manager user code, following the conventions is not enforced and FMs may decide to implement their own states and inputs.

**Transitional States** In order for Function Managers to react to inputs sent to their state machine by themselves, another resource or an operator using their GUI, FM developers may define transitional states which are located in between stable states in the state machine definition. They are used instead of state transition actions to ensure that the FM remains in a defined state while executing actions that may fail. Transitional states lead back to a stable or error state depending on the outcome of the action.

**Parameters** Function Managers have a set of parameters which is registered with the framework. Parameters can be modified by the FM itself, by the operator using the FM's GUI or by external resources using the Run Control framework's web services.

Parameter types are predefined by the Run Control framework and include wrappers for Java primitives and unsigned equivalents as well as basic collections, namely vectors and maps. The framework implements serialization and deserialization of these parameters to and from XML for transporting them via the SOAP web services. Furthermore, support for JSON (de)serialization of parameter collections for communication with the Function Manager web-GUIs is provided.

**Lifecycle** The lifecycle of a Function Manager is defined by the Run Control framework, with FMs implementing code to be executed during lifecycle changes. In particular, FMs can be created and destroyed, with the creation consisting of the preparation done by the framework, such as downloading the FM JAR and unpacking the custom GUI, and custom setup code defined by the Function Manager implementation itself. Similarly, destroying a Function Manager involves the shutdown and cleanup of central framework services provided to the FM. Again, a FM is able to perform its own cleanup operations within its custom destroy code.

**Event and Notification Handling** The Run Control framework provides Function Manager developers with the possibility to register custom event handlers, allowing the user code to react to various events such as state entries, state notifications from child resources, parameter changes and parameter notifications. Parameter changes describe external changes to the Function Manager's own parameter set, whereas parameter notifications may contain arbitrary parameter sets sent by other resources for the purpose of information exchange.

**Asynchronous Messaging** Since Function Managers may communicate with a large number of heterogeneous child resources, synchronous blocking communication would have a negative and unpredictable impact on a FM's performance. Therefore, most communication between Function Managers and their resources is done asynchronously. Responses to commands and state changes caused by state machine inputs sent to other resources arrive as asynchronous notifications, are routed to their target FM by the Run Control and then processed by the FM's event and notification handlers. The Run Control Framework allows for communication with other Function Managers, XDAQ resources and PSX, the interface to the Detector Control System.

**Message Queues** For incoming messages and notifications, the Run Control manages a set of message queues for each Function Manager. These message queues have different priorities and are accessible by the Function Manager's user code.

**Graphical User Interface** The Run Control provides a default web-based interface for Function Managers. Operators may use this interface to monitor the FM's state and parameters, issue state machine inputs and modify the FM's parameter set.

**Custom GUI** Utilizing Java Server Pages (JSP), HTML, CSS and JavaScript, Function Managers may define their own custom GUI, which replaces the default GUI provided by the Run Control. The custom GUI is shipped together with the FM's user code and unpacked from the JAR file by the Run Control once the FM is instantiated. Custom GUIs may use JSP tags and client-side code provided by the Run Control Framework

in order to subscribe to state and parameter changes of the monitored FM. The way in which states and parameters are visualized and what options are made available to the operator is entirely up to the developer of the custom GUI.

**Access Control** Since Function Managers are arranged in a hierarchical order and parent FMs control and communicate with their child FMs, an access control system prevents direct inputs and parameter changes to child Function Managers. This is done to safeguard against accidental interference with Function Managers that are supposed to be controlled by their parent FM only. The parent Function Manager is able to unlock its children at runtime, enabling direct control of a child Function Manager, for example by using its GUI.

**Monitoring** In addition to running configurations and executing their Function Managers' code, the Run Control and Monitoring System is also responsible for monitoring Function Managers and their child resources. Monitoring of FMs includes error logging, notification queue statistics and input, notification and asynchronous command reply logs. In order to clean up crashed Function Managers, a destroy backdoor is available, which bypasses the access control.

**Utility** In addition to the functionality described in the previous paragraphs, the Run Control framework also provides Function Managers with various utilities. This includes parallelized access to child resources, utilities to simplify database access and the ability to define task sequences. Task sequences are lists of tasks that are executed in order, can be set to wait for certain states and may process incoming notifications while they are being executed.

The Run Control framework allows Function Managers to access their group (2.2.4), and the resources contained in it.

### 2.2.2 Web Services

The Run Control Framework provides various JAX-RPC (Java API for XML-based Remote Procedure Calls) web services for the interaction between Run Control instances, Function Managers and their resources and for monitoring purposes, as well as for access by external tools. The web services are implemented using the Apache Axis library [47], an open-source JAX-RPC and SOAP implementation. Communication between Function Managers and resources that are not Function Managers themselves (e.g. XDAQ applications) is handled by separate endpoints in the form of Java Servlets, which act as a proxy to the web services.

**Command Service** The Command web service allows clients to send parameterized command Inputs to Function Managers.

**Error Receiver Service** The Error Receiver web service can be used to send errors to, allowing child FMs to propagate errors to their parent Function Managers.

**Lifecycle Service** The Lifecycle web service allows clients to control the lifecycle (creation, destruction) of Function Managers.

**Parameter Controller Service** The Parameter Controller Web Service may be used to retrieve and set Function Manager parameters.

**Parameter Receiver Service** The Parameter Receiver can be used to send Parameter Notifications to, allowing for event-based communication between Function Managers.

**Reply Command Receiver Service** The Reply Command Receiver is a web service XDAQ applications can send replies to asynchronous commands to.

**State Receiver Service** The State Receiver web service is used to propagate Function Manager states between RC instances, informing parent Function Managers about state changes of their child FMs.

**RS Retrieve Service** The RS Retrieve web service exposes the Resource Service accessible by the RC instance, allowing a client to obtain information about the directory structure and groups available.

**Notification Service** The Notification web service provides a publish/subscribe infrastructure for notification events.

**PSX Receiver Servlet** The PSX Receiver Servlet receives notifications sent by the Detector Control System (DCS) via the PSX (PVSS SOAP eXchange).

**Error Receiver Servlet** The Error Receiver Servlet receives errors from XDAQ resources. It decodes the SOAP messages and routes them to the target FMs.

### 2.2.3 XDAQ

XDAQ, pronounced "Cross-DAQ", is the CMS Online Software (CMSOS) and a software platform for the development of distributed data acquisition (DAQ) systems [62]. It is extensively used across the CMS experiment as a communication middleware, connecting distributed applications. XDAQ is written in C++ and provides developers with libraries and interfaces to implement their own XDAQ applications, including state machine support and hardware access libraries (HAL) as an abstraction layer for communication with hardware.

**Executives** XDAQ executives correspond to a single process and execute XDAQ applications in separate threads. Executives describe a XDAQ process uniquely identified by a host and port. A single executive can run multiple applications concurrently.

**JobControl** JobControl is a XDAQ application running as a service on all machines. It is used by the Run Control framework to start and stop other XDAQ applications (by starting and stopping the respective executives) [70]. Communication between the Run Control and the remote JobControl is realized using SOAP.

**Service Applications** Unlike regular XDAQ applications which are started by Run Control (using JobControl) when their parent Function Manager is created and which are stopped when the FM is destroyed, XDAQ service applications are running continuously. In order to avoid a service application being used by multiple Function Managers concurrently, a lease system is implemented, allowing only the lease owner to control the application. A lease has to be renewed periodically and expires eventually if it is not renewed.

**XDAQ Configuration** XDAQ configuration files are XML files that describe the XDAQ executives and their applications, as well the endpoints provided by and the connections between them. The description of an application also includes application parameters [1].

**XML Message Format** The communication between XDAQ and Run Control is based on XML over SOAP. XDAQ defines message formats [1] for certain message types. In particular, the CMS error and state notification message formats need to be supported by the RCMS in order to be notified of state changes or error messages of remote XDAQ applications.

### 2.2.4 Resource Model

The Run Control framework makes use of a resource model for representing the configuration structures stored in the database.

**Groups** A group represents part of a configuration's hierarchy, consisting of a Function Manager and its child resources. A group may have a single parent group, corresponding to its Function Manager having a parent FM. A group without a parent group is called the top group of the configuration. In addition to child resources, groups may therefore have child groups, describing child Function Managers and their resources.

**Resource Types** The resource service identifies different types of resources. A differentiation between the resource type in the database and its purpose and behavior is made in the form of the "qualified resource type" attribute. Qualified resource types are used to describe the behavior and interface of a resource. The identified qualified resource types and corresponding resource types in the model are listed in table 2.1.

| Qualified Resource Type | Model Type | Remarks |
|---|---|---|
| Function Manager | Function Manager | a top FM or a child FM of another FM in the same configuration |
| Subsystem | Function Manager | a Function Manager that is a child of another FM but not part of the parent FM's configuration; instead, it has a mapping for the respective subsystem role in the Global Configuration Map (GCM) |
| XDAQ Executive | XDAQ Executive | |
| XDAQ Application | XDAQ Application | |
| XDAQ Service Application | XDAQ Application | a continuously running XDAQ application that can be leased |
| JobControl | JobControl | a XDAQ application used to start arbitrary applications |
| Generic | Generic | |
| Psx | Generic | only used for PVSS State Machine (SMI) notifications |
| any | Virtual Resource | not included in the configuration hierarchy; attached to the top group |

Table 2.1: Resource types as identified by the resource service

## 2.3 Tools for configuration management

The RS3 Manager tool allows users to create, edit, migrate and manage configurations. To accommodate for the size of configurations for the central DAQ system and to automate their creation, the DAQ Configurator was created.

### 2.3.1 RS3 Manager

The RS3 Manager is a management tool for the Resource Service 3 and can be used to create and modify configurations. The RS3 Manager is also able to export and import configurations to and from XML and provides functionality to migrate configurations from the legacy Duck DB to the Resource Service 3.

Like the DAQ2 Configurator, the RS3 Manager is capable of creating XDAQ XML configuration (2.2.3) files for configurations.

### 2.3.2 DAQ2 Configurator

Configurations that are executed using the Run Control can represent large hierarchies of resources. For this reason, the DAQ2 Configurator was created to automate the creation of configurations. It is a successor to the DAQ Configurator [74] and the DuckCAD [78] and tailored to the creation of configurations for the central DAQ system.

The DAQ2 Configurator operates on the Hardware Configuration (HWCfg, 2.4.1) and Software Template (SWT) databases to create DAQ configurations and writes them the Resource Service 3 (RS3, 2.4.2), optionally registering them in a Global Configuration Map (GCM, 2.4.5).

**Software Templates** The DAQ Configurator makes use of Software Templates (SWT), which contain definitions of units and their control structures, including the hierarchy of Function Managers as well as the hierarchy of their resources. Software Templates also contain a collection of site- and user-specific variables and settings that are used during the automated creation of configurations.

**Blacklist** During the creation of configurations, a blacklist may be used to exclude certain machines from being used in the configuration due to ongoing maintenance. This feature is extensively used by system experts to quickly modify configurations in the case of failing machines.

## 2.4 Databases

In order to persist data, the RCMS uses a variety of databases. Database connections are maintained and monitored by the Run Control framework, which provides connectors that abstract from the underlying SQL databases and perform automatic recovery of erroneous connections.

### 2.4.1 Hardware Configuration

The Hardware Configuration (HWCfg) database stores information about the DAQ system's hardware components and the physical connections between them.

### 2.4.2 Resource Service 3

The Resource Service 3 (RS3) stores configurations (2.1), their hierarchy and properties, as well as information about the resources that are part of them.

### 2.4.3 Run Info

Originally, the Run Info database contained one set of frequently updated entries per run which reflected the current state of the experiment. However, the Run Info connectors were modified to support updating this information while retaining a history, effectively turning the Run Info into a logging database.

### 2.4.4 Log Session

The Log Session database is responsible for storing unique session identifiers (SIDs), which identify a set of runs that was taken with the same instance of a function manager. The Run Control framework provides a connector to start new sessions and close existing ones. Sessions are usually obtained by the topmost Function Manager, which passes the session id down to its child FMs. Session IDs are stored along with a user and database sequence name, identifying the Run Control instance for which the session was requested.

### 2.4.5 Global Configuration Map

Global Configuration Maps (GCMs, 2.1) are persisted in a database and can be accessed using a unique key that follows the semantics of a filesystem path.

### 2.4.6 HLT Configuration

The HLT Configuration database persists High Level Trigger (HLT) configurations. The HLT Configuration database is external to the RCMS.

### 2.4.7 L1 / HLT

The L1/HLT (Level 1/High Level Trigger) database contains so-called L1/HLT keys, which describe compatible sets of Level 1 and HLT configurations. Based on these keys, HLT configurations are loaded from the HLT Configuration database and supplied to the High Level Trigger.

### 2.4.8 Run Mode

Run Modes are persisted in the Run Mode database and include a L1/HLT key in addition to other configuration parameters. The Run Mode can be automatically selected by the Level-Zero Function Manager based on the Run Mode Matrix and the machine and beam modes of the LHC.

## 2.5 Log Collector

The Log Collector (or Logging Collector) of the RCMS is a service separate from the Run Control. It provides network endpoints for sending log messages to, which it persists to a database or file or distributes using a publish/subscribe system.

The Log Collector not only receives messages from Function Managers and the Run Control, but also from remote XDAQ applications, which send log messages to the Run Control their parent FM is running on.

The Log Collector supports XML-based Apache Log4J and log4cplus log formats [75]. Produced logs can be viewed using Apache Chainsaw or, due to performance considerations, a compatible command-line tool called Handsaw.

## 2.6 Level-0 Function Manager

The Level-0-FM (LV0) is the top-most Function Manager, a parent to the Level-1 FMs such as those of the Timing and Control Distribution System (TCDS), Detector Control Systems (DCS) and the Subsystem FMs, which in turn may have children FMs of their own. The Level-0-FM provides operators with a high-level interface to control and monitor the experiment's data taking operations.

## 2.7 Standalone Monitoring Tools

In addition to the monitoring tools built into the Run Control, the RCMS also offers monitoring tools external to the Run Control Framework.

**RunInfo Servlet** The RunInfo Servlet presents the information from the Run Info database (2.4.3) in a tabular format. Logged information can be accessed by run number or session ID.

**RunInfo Timeline** The RunInfo Timeline visualizes the information contained in the Run Info database on an interactive timeline, giving an overview of the experiment's Function Manager states and the actions executed by operators and automated systems.

## 2.8 Automation

With the number of system experts being limited and the system operators having to keep an eye on a multitude of monitoring tools, automating system recoveries to reduce downtime and data loss and to facilitate the work of operators stands to reason.

**DAQ Doctor** The DAQ Doctor was developed as a monitoring tool with expert knowledge, collecting information from various monitoring sources to determine the cause of system downtime according to predefined rules. Since it was implemented for the DAQ system of Run 1 and not adapted to the changes of the DAQ system for Run 2, the DAQ Doctor was replaced with the DAQ Expert.

**DAQ Expert** As a successor to the DAQ Doctor, the Expert takes a more modular approach to detecting problems with the DAQ system. The collection of data is now done by a separate service, the DAQ Aggregator, which aggregates monitoring data from various sources and persists them into snapshots. These snapshots are in turn used by the expert and other monitoring tools.

**LV0 Automator** The Level-0 Automator (LV0A) is a Function Manager that provides assistance to operators and an interface for external systems such as the Expert. As the name suggests, it automates certain actions that otherwise need to be done manually in the Level-0-FM (2.6). It is implemented as a parent FM of the LV0, while still allowing operators to control the LV0 FM directly.

# Chapter 3

# Improvements

This chapter describes possible improvements that have been identified for the current RCMS and for the implementation of which it is assumed that the foundation and concepts of the RCMS remain unchanged.

The improvements described in this chapter are based on the experience of the RCMS developers, as well as feedback from experts and users. A dedicated meeting with the DAQ group in order to gather feedback and suggestions was held.

## 3.1 Functional

This section describes functional improvements which affect the functionality of the RCMS itself, ranging from substituting libraries to rewriting existing features.

### 3.1.1 Communication

One of the main features of the RCMS is handling the local and remote communication between Function Managers and between Function Managers and their child resources. Most communication is done via webservices implemented using the outdated Axis library and JAX-RPC standard. Communication with XDAQ requires additional Servlets. Message formats are not uniform and FM communication requires conversion between Java Primitives and custom Parameter Types, while using the webservices requires conversion to and from Beans.

**Library** Modern libraries are able to serialize and deserialize regular Java objects (POJOs) to and from XML and JSON without requiring any configuration, while still allowing their behavior to be influenced using Java Annotations. Using such a library would make the conversion to and from Beans obsolete and also allow for transparent conversion of data into different formats for communication between Run Control instances and with the FM GUIs.

**Message Broker** As an alternative or in addition to direct communication between Run Control instances, a message broker could be used, allowing reliable delivery of commands and notifications. Using a publish/subscribe system (topics) to publish state and parameter notifications would allow services to transparently attach to the message

broker to monitor the exchanged messages. One-to-many communication would become trivial and no longer be a concern of the RCMS implementation. Adapter services for XDAQ communication could be used to publish messages from XDAQ to the broker.

### 3.1.2 Function Managers

Function Manager improvements mainly concern themselves with simplifying the development of FMs and providing solutions for common problems that need to be solved by Function Manager developers.

**Annotations** Instead of, or in addition to, providing interfaces for Function Managers to implement, Java Annotations may be provided, allowing for more flexibility when implementing functionality required by the Run Control. Together with automatic discovery of annotated classes and methods, this would allow interaction points between FMs and the Run Control to be defined more loosely.

**Websocket Endpoints for GUIs** In order to allow for easier communication between a Function Manager and its GUI, WebSockets may be used, allowing for bidirectional exchange of messages over a persistent connection. A suitable protocol for the exchange would be STOMP [86] in combination with a message broker, allowing the GUI to subscribe to and be notified of relevant FM parameter changes and events.

**Thread Pool and Scheduler** It might be useful for the Run Control framework to provide Function Managers with a dedicated thread pool and scheduled thread pool, as some FMs use custom implementations for scheduling various tasks that need to be executed periodically or with a delay. Framework-provided thread pools can be shut down cleanly on destruction of the FM, assuming the user code handles interrupts correctly.

**Scripting Languages** To allow developers more freedom when implementing a Function Manager, it might be interesting to allow FMs to be written in scripting languages, such as JavaScript, Python and Groovy. This conflicts with other suggested improvements, namely with a unified build system for FMs, Java Annotations and the simplification of framework-provided functionality.

**Template System for GUIs** In order to separate a Function Manager's logic from its GUI, technologies such as JSP should be abandoned. Instead, client-side rendering with template systems should be considered.

### 3.1.3 Logging

One of the Run Control's purposes is the collection of log messages produced by Function Managers and their XDAQ child resources. The collected messages are written to a single file, providing a starting point for debugging.

**Modern Logging API** Currently, the Run Control framework uses Apache Log4J as a Java logging library, while XDAQ makes use of log4cplus, a compatible logging solution

for C++. Moving forward, the successor Log4J 2 should be used due to increased performance, stability and a new API which is separated from the implementation, making it possible to replace logging libraries transparently.

**Unified Logging System** The Run Control framework provides and uses Log4J loggers directly, as well as its own wrapper classes. Since the functionality isn't entirely the same, this may cause confusion for developers. Since the Log4J 2 API is a separate project and provides a standardized interface, it is suitable to replace the custom logger classes while maintaining modularity and independence from the logger implementation.

### 3.1.4 Configurations

The structure of configurations and components and the databases they are stored in, evolved with changes in requirements and changes in the structure of the DAQ system. Improvements may be made to how system components and configurations are stored, with the format heavily depending on the components and layout of the DAQ for Run 3 and 4.

The configuration management aspect was identified to be the single most complex component of the RCMS and one of the most important points of improvement in order to ease the work of system experts. The structure of the configuration management suggests that it evolved as the requirements for it changed with the DAQ system and operational experience gained.

**Full and Light Configurations** The RCMS distinguishes between full and light configurations, with full configurations being stored in a normalized format in the database. Since this approach led to poor performance when loading and saving large configurations, light configurations were introduced. The resource hierarchy of light configurations is stored as compressed BLOBs (binary large objects) in the database. Due to denormalization, editing light configurations has certain limitations, especially when it comes to regenerating XDAQ XML configuration files for them. Ideally, a single fully supported and performant configuration format should be implemented.

**Configuration Library** Since the configuration management tools, the Run Control framework and external tools make use of the Java configuration structure, this abstraction from the database should not be part of the Run Control framework project as it is now, but instead be contained in its own project and included where needed as a library. This represents a more modular approach, where transparent changes can be made to how configurations are stored.

### 3.1.5 Configuration Management Tools

Improvements to the configuration management tools are partially dependent on the databases they operate on, yet usability by system experts is also a criterion to keep in mind when wanting to improve or rewrite the tools for configuration management.

**Web-based tools** Instead of distributing the configuration management tools as standalone Java applications started using Web Start technology, a web-based interface could be used. This would ensure that the latest version of the tools is in use and possibly remove the need for the user to organize their own setting files.

**Scriptable configurations** Similar to how the Hardware Configuration database fillers work today, entire configurations could be described by source code operating on the resource model, allowing for arbitrary logic and great flexibility when creating them. Conversely, the person creating the configuration would be required to have knowledge of the programming language used, making this more of an additional feature than a replacement for the current tools.

## 3.2 Non-Functional

Non-functional improvements are mostly improvements to the development of the Run Control Framework, including the build system, version control, code style guidelines, tooling and others.

### 3.2.1 Function Managers

In addition to the functional improvements to the Function Manager framework described earlier, this section concerns itself with non-functional improvements.

**Versioning Scheme** Function Managers and their GUIs should follow a unified versioning scheme, allowing operators to verify which version of a FM is being used. Version information should be provided in a normalized format, possibly a separate configuration file describing the FM, to allow the Run Control to obtain and display this information to users.

**Build System** A unified build system for Function Managers and FM GUIs should be provided. A central continuous integration service may also be configured to allow for automated builds and additional tests.

### 3.2.2 Security

Besides identifying users for logging purposes and adding safeguards against accidentally issued commands, the security of the Run Control may be improved further. While access from the internet is possible using SSH tunnels, the Run Control system is contained in a private network and not exposed to the internet.

**HTTPS** Providing access to web-based interfaces using HTTPS stands to reason, as TLS is widely supported by browsers. The slight performance impact due to connection overhead is negligible considering the low frequency of new connections to web-based interfaces.

### 3.2.3 Logging

Apart from unified and reliable logging, non-functional improvements to the log system were identified.

**Log Message Style** Guidelines for a common log message style should be agreed upon, including the format and log level of messages. This should be done to make the logs easier to filter and analyze, as well as to simplify automated processing.

### 3.2.4 Version Control

With going back to previous versions of the source code and having a history of changes being a requirement for the development of the Run Control framework, a version control system is to be used. Currently, the RCMS source code is kept in a SVN repository provided by the CERN IT department [27].

**Git** Since CERN IT recommends the migration from SVN to CERN's GitLab service [25], the RCMS repository should be migrated to Git eventually.

**Commit Message Guidelines** To make the version history useful, consistent and meaningful commit messages should be written. To support this, a common message style, such as semantic commit messages [26], may be adopted by all developers.

### 3.2.5 Build System

In order to create builds of the different components of the RCMS, various build tools are in use today. The Run Control framework, configuration management tools and central Function Managers use Ant for compiling and packaging, which is the conventional build tool for subsystem Function Managers as well. Deployable RPM packages for the different setups of the RCMS are created by a script written in Perl.

**Maven** As a convention-based build system, Maven might be preferable to Ant, allowing for consistent build processes with minimal configuration, as well as a significant simplification of dependency management. With Maven being based on conventions, it lacks the flexibility of Ant.

**Gradle** While Gradle follows conventions and has support for Maven's dependency management, it still provides developers with means to fully customize the build process and could therefore be seen as a compromise between the Ant build system currently in use and Maven.

**Continuous Integration** Continuous integration (CI) allows for the automated execution of tests and builds, as well as packaging. For the RCMS, continuous integration might be a suitable option to speed up the release cycle. Separate builds could be used to facilitate the deployment of test instances.

### 3.2.6 Tooling

Making use of tools can greatly improve development efficiency and help with finding issues as well as improving code quality. Apart from a state of the art integrated development environment (IDE), other tools can be identified to support the development of the RCMS.

**Static Analysis** Static analysis tools, also built into modern IDEs, can be used to find common errors and misconceptions in the source code. They can prove especially useful to hint at difficult to debug timing issues caused by improperly synchronized concurrency.

**Ticket System** In order to document the identified defects and enhancements of the system, a ticket system may be used, with the tickets being referenced by related commits to the version control. The current RCMS makes use of Trac [27]. When considering Git and the services available at CERN, systems such as GitLab or JIRA may be used instead.

### 3.2.7 Tests

In order to verify the continued functionality of the software, especially after changes were made to it, tests should be provided. Currently, the RCMS defines tests for some of its modules.

**Unit Tests** Unit tests should be provided for all modules of the software, covering the known use cases as well as the the fault tolerance of the system. Unit tests may be executed automatically by a continuous integration tool on every build or whenever a new commit to the version control is made.

**Integration Tests** Integration tests should be provided to ensure that critical components of the Run Control work together. Like unit tests, integration tests may be run by the CI service.

**Test Coverage** A certain level of test coverage should be agreed upon. Critical parts of the software should be tested rigorously with more comprehensive test cases.

### 3.2.8 Code Style

A readable and uniform code style can improve the maintainability of source code significantly. Due to many different developers working on the RCMS over the years and due to the lack of guidelines, the code style is partially inconsistent.

**Modern Data Structures** Throughout the RCMS, Java data structures such as Vectors and Hashtables are used, with more appropriate alternatives such as Lists and (Concurrent) Hash Maps available in newer versions of Java. Often, the behavior and interfaces of these alternatives are more desirable. While the mentioned modern data structures are also widely used by the Run Control Framework, the choice of data structures is not consistent across the framework. Refactoring the current RCMS to replace the data structures used may not be worth the effort, yet a new implementation should consistently make use of them.

**Code Style Guidelines** In order to unify the code style throughout the RCMS, code style guidelines should be agreed upon. These should preferably follow proven guidelines and recommendations, such as the ones provided by Google [49].

**Checkstyle Rules** To enforce the agreed upon code style guidelines, tools such as checkstyle [73] may be used. Modern IDEs and build systems usually provide code style presets and custom rules or plugins to run external style checks.

**Code Reviews** Code reviews can help to detect possible issues with changes and additions that are made to the source code. Especially when changes are made by new developers who lack expert knowledge about the system and if requirements are not fully understood, code reviews may help to find problems before the software or changes to it are deployed.

**Modularity and Separation of Concerns** Modularity and separation of concerns should be employed wherever possible in order to improve maintainability, with modularity ideally allowing for transparent modification and even replacement of components. Separation of concerns should be employed to split the Run Control framework into multiple projects and to design a clear interface between the system's backend and the graphical user interfaces and external tools.

### 3.2.9 Documentation

While documentation on the current RCMS exists in the form of introductory texts, mostly changed and added functionality was documented in a developer-friendly and condensed format. The documentation of the RCMS should be designed to introduce new Function Manager developers to all aspects of the framework, including examples on how to make use of and interface with the provided functionality. Demonstrating the intended way of using the framework will also avoid Function Manager developers finding non-standard and unforeseen solutions to common problems.

### 3.2.10 Deployment

Deployment of the Run Control and Log Collector is done using RPM (RPM Package Manager) packages. Improvements may be made to the ease of deployment, especially for test setups. Automated packaging and largely automatic deployment for production setups is desirable.

**Reverse Proxy** Performant TLS termination for HTTPS may be handled by a separate reverse proxy service, separating concerns further and ensuring efficient connection handling. Slow requests originating from remote connections may be buffered by an efficient event-based reverse proxy server to avoid consuming resources of the Run Control backend.

**Virtualized deployment** To ease the deployment for test setups, virtual machine images or Docker containers may be provided, containing a preconfigured Run Control system and the services required to use it.

**Database support** The current RCMS only supports Oracle databases, with MySQL support having been discontinued in favor of reducing maintenance work. However, test setups often don't have access to commercial database services, so supporting at least one freely available database such as PostgreSQL or MySQL would ease the deployment of the RCMS. In combination with using an ORM library for database access, the maintenance work required to support multiple database services might be negligible.

# Chapter 4

# New Functionality and Concepts

This section describes new features that are desirable to be added to the current RCMS or implemented in case of a rewrite of the RCMS, as well as possible changes to the core concepts of the Run Control system for Run 4.

Like the improvements described in the previous chapter, new concepts were devised from ideas and feedback of RCMS developers as well as DAQ experts and RCMS users.

## 4.1 New Functionality

In addition to the improvements described in chapter 3, possible new functionality to be added to the RCMS was identified and discussed.

### 4.1.1 Security and Safety

While the production Run Control instances run in a private network and are not accessible from the internet, operators and experts may still connect to them from outside CERN using tunnels to the CERN LXPLUS [24] service and CMS cluster head nodes [88]. Improving the security and safety of the system should be considered to protect the production system from operational accidents.

**Identifying Users and Restricting Access** Currently, the Run Control identifies users by the username specified when logging in to the Run Control instance. This account mainly determines which configurations can be loaded. The available accounts and passwords are known to most users and are therefore not suited or intended to identify the person operating the instance. Instead, users should be identified using their existing CERN account, which can be accomplished by interfacing with the CERN single sign-on service. Actions performed can then be mapped to the person operating the system.

Once users are identified, their access may be restricted based on their role. While expert users may require full access to the system at all times, limited time-based access for shifters could reduce the danger of operational accidents.

The DAQ group assumes user authentication to have negative impact on operations. Existing access control restrictions (2.2.1) and GUI locks employed in top Function Managers suffice to prevent operators from issuing commands on accident. Limitations arising from role-based permissions may lead to operators being unable to control the system when they need to, due to unforeseen changes in the shift schedule or issues that prevent an operator from authenticating.

API access by external systems may be limited using identification tokens, such as session-identifiers or secrets only known to the service using the API and regulated by defining permissions to be granted to each service.

## 4.2 New Concepts

In contrast to previous statements, this section concerns itself with describing conceptual changes to the Run Control and Monitoring System. Changes described here may not be possible to implement without rewriting significant parts of the RCMS.

### 4.2.1 Configurations

**Storage as XDAQ XML** Instead of storing Configurations in the Resource Service database, storing them in a superset of the XDAQ configuration XML format (2.2.3) may be possible, reducing the number of databases required by the RCMS. The XDAQ configuration XML format is already being used by the RCMS and XDAQ XML is maintained as part of the database configuration format. Since XDAQ has no concept of Function Managers, the RCMS would need to use an extended XML format, with XDAQ-compatible nodes to describe the resources that are known to XDAQ. Changing the configuration format to be file-based would also unify full and light configuration variants (3.1.4) into a single format which is ready to be exported and imported to and from files. Providing configurations for XDAQ resources becomes as easy as extracting sub-nodes of the RCMS configuration XML.

Due to the advances in computing performance since the start of the CMS experiment, the number of machines in use by the DAQ were reduced significantly. Therefore, XDAQ configuration files are smaller and need to be distributed to far less machines than before. The current RCMS splits the configuration file for all XDAQ executives and applications into smaller, machine-specific parts to reduce the startup time of the system. This is no longer required due to the aforementioned reduction in the number of machines and the significantly increased bandwidth available. A single configuration file may be passed to all XDAQ resources, with XDAQ extracting the relevant parts, easing the configuration management of the RCMS further.

Conversion between the Java configuration model and XML should be taken care of by a library and require as little XML construction and configuration by the RCMS as possible, while still remaining compatible with the format expected by XDAQ.

### 4.2.2 Runs

As described in section 2, the Run Control uses Runs to describe the state in which the components of the experiment are ready to take data. Instead of synchronizing all components of the experiment, it may be feasible to allow partial Runs, during which some of the sub-detectors are not ready to take data. The recorded data would have to contain information about which detectors contributed to it and a different synchronization mechanism for the sub-detectors would have to be implemented in systems external to the RCMS. Abandoning the strict requirements of Runs may allow for faster system recovery and therefore reduce downtime, by allowing data to be taken even while some of the subsystems are faulty and need to be recovered.

# Chapter 5

# Requirements for a future Run Control framework

This chapter concerns itself with formalizing the requirements for a future Run Control framework, as other aspects of the RCMS, such as configuration management, are out of scope of this thesis. The requirements described are high-level requirements, documenting the well-understood functionality of the Run Control framework and serving as a basis for future developments and changes to the current system. The chosen approach for documenting requirements is based on the Unified Process as taught as part of a Software-Engineering university course [48].

Requirements will be divided in functional and non-functional requirements, describing functionality as well as the environment and properties of the system. Only the most important requirements are included for later reference by prototype implementations.

Use cases will be employed to describe the system functionality from an operator's and Function Manager developer's perspective. This top-down approach was chosen to support agile development as well as independence from the technologies used.

## 5.1 Functional Requirements

Functional requirements describe the functionality that needs to be provided by the system. The Run Control framework's functional requirements are well-understood, yet underwent slight changes since their initial implementation. Eliciting the functional requirements is done as part of the analysis of the problems the Run Control framework is designed to solve and will serve as a point of reference for prototypical designs and implementations later in the thesis.

Functional requirements are grouped into categories of concern, namely general functionality, configurations, network communication, Function Managers, monitoring and safety.

### 5.1.1 General functionality

Functional requirements that do not fit into a single other category yet are essential to the Run Control framework are listed here.

| ID | Name | Description | Visible | Relevance |
|---|---|---|---|---|
| F-B01 | maintain database connections | maintain connections to all required databases | no | required |

### 5.1.2 Configurations

Functional requirements that are related to how the Run Control framework allows operators and Function Managers to interact with configurations. This does not include the functional requirements of the RCMS regarding the creation and modification of configurations.

| ID | Name | Description | Visible | Relevance |
|---|---|---|---|---|
| F-C01 | browse configurations | display available configurations for the current instance and user | yes | required |
| F-C02 | run configuration | start a configuration and its associated function manager | yes | required |
| F-C03 | attach to configuration | attach to an already running configuration | yes | required |
| F-C04 | list running configurations | display configurations running on the current instance and their respective top FM | yes | required |
| F-C05 | view function manager GUI | display default or custom GUI of a function manager | yes | required |

### 5.1.3 Network Communication

Requirements for the Run Control Framework regarding network communication between distributed Function Managers, Function Managers and their resource and Function Managers and their GUIs.

| ID | Name | Description | Visible | Relevance |
|---|---|---|---|---|
| F-N01 | send input to function manager | send a parameterized state machine input to a function manager | no | required |
| F-N01a | send command to function manager | send a parameterized command to a function manager | no | required |
| F-N02 | read function manager parameters | read a set of parameters from a function manager | no | required |
| F-N03 | poll resource state | poll the state of a local or remote resource | no | required |
| F-N04 | route notifications to function managers | route incoming notifications to the correct function manager | no | required |
| F-N05 | provide message queues | provide message queues with different priorities for incoming notifications to FMs | no | required |
| F-N06 | provide endpoints for receiving messages | provide endpoints for external resources and systems to send messages to | no | required |
| F-N07 | provide endpoints for FM GUIs | provide endpoints for FM GUIs to send commands and receive parameter updates | no | required |
| F-N08 | send commands to XDAQ applications | provide interface to send messages to XDAQ | no | required |
| F-N08a | send commands to job-control | provide higher-level interface to send commands to job control | no | required |
| F-N09 | change FM parameter | change a FM's parameter | no | required |

### 5.1.4 Function Managers

Requirements for the Run Control Framework for functionality provided to Function Managers, including essential utilities, interaction with Configurations, the definition of message handlers, FM GUIs and registration of parameters and state machines.

| ID | Name | Description | Visible | Relevance |
|---|---|---|---|---|
| F-FM01 | custom GUI | allow function managers to define a custom GUI which is displayed on attach | yes | required |
| F-FM02 | provide default GUI | provide a default GUI allowing control of a FM's state machine and displaying its parameters | yes | required |
| F-FM03 | database access | provide a FM with database connectors to required databases | no | required |
| F-FM04 | send messages to resources | allow a FM to send messages to its child resources | no | required |
| F-FM05 | process incoming notifications | allow FMs to provide event handlers for incoming notifications | no | required |
| F-FM06 | definition of state machines | provide a framework for the definition of state machines: states, inputs, transitions and actions | no | required |
| F-FM07 | definition of parameters | allow a FM to define parameters that are made accessible to the FM's GUI and to other resources | no | required |
| F-FM08 | access to configuration | allow a FM to access information about its own configuration group (child resources etc.) | no | required |
| F-FM09 | process incoming commands | allow FMs to provide event handlers for commands (and inputs) | no | required |
| F-FM10 | process parameter changes | allow FMs to provide parameter change handlers | no | required |
| F-FM11 | send notifications to GUIs | allow FMs to actively send notifications to all attached GUIs | no | important |

### 5.1.5 Monitoring

Monitoring functionality describes the requirements for monitoring tools integrated into the Run Control Framework itself. Requirements for external tools that are part of the RCMS are not described.

| ID | Name | Description | Visible | Relevance |
|---|---|---|---|---|
| F-M01 | central log | provide a central log for the instance, local function managers and their XDAQ resources | yes | required |
| F-M02 | display routing rules | display the active routing rules for incoming notifications | yes | important |
| F-M03 | list running FMs | display a list of all FMs running on the current instance | yes | important |
| F-M04 | force destroy FM | forcefully destroy a FM | yes | required |
| F-M05 | display Inputs sent to FMs | display a log of Inputs sent to the function managers of the local instance | yes | important |
| F-M06 | display received notifications | display incoming notifications along with the FM they have been routed to | yes | important |
| F-M07 | display a list of FM errors | display a list of errors for all local FMs | yes | important |
| F-M08 | display sent notifications | display a list of sent and answers to received notifications | yes | important |
| F-M09 | display statistics for message queues | display the state of various message queues for each FM | yes | important |
| F-M10 | monitor database connections and queries | monitor database connections and queries for errors and duration | no | important |

### 5.1.6 Safety

Requirements for the Run Control Framework regarding safety, intended to safeguard against operational accidents, such as accidentally issued commands.

| ID | Name | Description | Visible | Relevance |
|---|---|---|---|---|
| F-S04 | access control | prevents unwanted actions from being executed by limiting access to the GUIs depending on the experiment's state and the Function Manager hierarchy | no | required |

## 5.2 Non-functional Requirements

The non-functional requirements described here incorporate some of the possible improvements identified earlier, but also include the requirements of the current Run Control framework that are relevant to provide measures for the suitability of the technologies

used to implement the prototypes. Non-functional requirements also describe properties that the software is required to fulfill, as well as environmental properties that the software requires to function reliably.

### 5.2.1 General Requirements

General non-functional requirements that do not fit into any of the other categories.

| ID | Name | Description | Relevance |
|---|---|---|---|
| N-B01 | web interface | all visible functionality must be available using a web interface | required |

### 5.2.2 Deployment

Requirements regarding the deployment of the Run Control Framework.

| ID | Name | Description | Relevance |
|---|---|---|---|
| N-DP01 | RPM distribution | distribution as RPM packages | required |
| N-DP02 | automated builds | automated builds by a CI service | wanted |

### 5.2.3 Network

Network requirements of the Run Control Framework for the distributed execution of Function Managers and the communication with resources.

| ID | Name | Description | Relevance |
|---|---|---|---|
| N-N01 | high-speed network | instances must be connected by a high-speed network (e.g. Gigabit Ethernet) | important |
| N-N02 | reliable network | if the network is partitioned, nominal operation can not be guaranteed | required |

### 5.2.4 Function Managers

Functionality for Function Manager development that is not part of the Run Control Framework itself, but is required in order to interact with the Framework.

| ID | Name | Description | Relevance |
|---|---|---|---|
| N-FM01 | GUI framework | provide a framework for implementing custom FM GUIs | required |

### 5.2.5 Environment

Environmental requirements for the Run Control Framework, describing where it needs to be able to run.

| ID | Name | Description | Relevance |
|---|---|---|---|
| N-E01 | CC7 compatibility | runnable on CentOS CERN 7 | required |

### 5.2.6 Version Control

Agreed upon and recommended requirements regarding the usage of version control for the development of the Run Control Framework.

| ID | Name | Description | Relevance |
|---|---|---|---|
| N-V01 | git | git as version control system | required |
| N-V02 | commit message guidelines | guidelines for the format and content of commit messages | important |

### 5.2.7 Source Code

Non-functional requirements regarding the source code of the framework.

| ID | Name | Description | Relevance |
|---|---|---|---|
| N-C01 | maintainability | maintainability through code reuse, usage of libraries and modularity | required |
| N-C03 | test coverage | test coverage for the most critical parts of the system | important |
| N-C04 | code reviews | implementation and modifications should be reviewed by at least one other developer | important |
| N-C05 | code style guidelines | guidelines regarding the formatting of code | important |

### 5.2.8 Tooling

Requirements to be met by tooling used during development.

| ID | Name | Description | Relevance |
|---|---|---|---|
| N-T01 | static analysis | static analysis tools to find common bugs | important |
| N-T02 | code style check | usage tools to check if the code style complies with the agreed upon guidelines | important |

### 5.2.9 Security

Security requirements which are not directly related to the functionality of the framework and do not affect its implementation.

| ID | Name | Description | Relevance |
|---|---|---|---|
| N-S01 | HTTP over TLS | encrypt connections to the web interface and custom GUIs | important |

### 5.2.10 Documentation

Requirements of the framework regarding documentation that is made available to operators and Function Manager developers.

| ID | Name | Description | Relevance |
|---|---|---|---|
| N-D01 | ticket system | document all changes in a ticket system | required |
| N-D02 | FM development | provide documentation for Function Manager developers, describing the functionality provided by the framework and how to integrate with it | required |

## 5.3 Use Cases

The use cases described in this section are high-level use cases and exclude details regarding interaction with other components of the RCMS such as the configuration management. Use cases are loosely grouped and put into relation using use case diagrams.

A use case consist of an identifying name, a list of actors that participate in the use case, a priority describing its importance for the system and in relation to other use cases and a textual description of the use case's procedure. Additionally, it contains preconditions that have to be met for the use case to be feasible, references to functional and non-functional requirements that stem from the use case and possible points which require clarification.

**Use Case Template** The following template is used to formalize all use cases for the Run Control Framework that have been identified.

| **Use Case** | **"Use Case Name"** |
|---|---|
| Actors | Users and external components involved in the use case. |
| Priority | Importance of this use case for the system. |
| Description | A description of the actions performed as part of the use case. |
| Preconditions | Preconditions that have to be met for the use case to be performed. |
| Functional references | References to functional requirements. |
| Non-funct. references | References to non-functional requirements. |
| Clarification needed | Uncertainties and remarks regarding the use case. |

Table 5.1: Use Case Template

### 5.3.1 Preface: A good use case

As part of the problem analysis, use cases are a technique to specify functional requirements, grouping them into categories and putting them into relation. The purpose of use cases is specified as part of the UML 2.0 standard [56]. Considering the requirements listed previously are very broad, some use cases translate into a single functional requirement.

Actors describe users or systems external to the system that is being developed. Use cases produce a result for the actors involved and therefore describe functionality that in itself is valuable to the actors. This also means that use cases can stand on their own and, while relations between use cases are common, are able to produce a distinct result. [56]

Use cases describe a flow of actions from the perspective of one or more actors and, depending on the level of detail the use case is described with, also describe alternative flows in case of differing conditions or exceptions [56]. Use cases can also provide context for user stories, grouping them together and serving as a point of reference, allowing agile development to benefit from use cases as well [71].

**Use case diagrams** Use case diagrams put use cases into relation in the context of the entire system or a component thereof. They allow the specification of usages and specializations between use cases. They visualize the interaction between actors and the respective system component. [56]

**Function Manager development** Use cases from a Function Manager's perspective treat the Function Manager user code as an actor in the system. While Function Managers are executed by the Run Control framework, they are not part of it, resulting in use cases that can not stand on their own from an operator's perspective.

### 5.3.2 Configuration Use Cases

This section describes the Run Control Framework use cases which allow an operator to interact with the configurations available to the system.

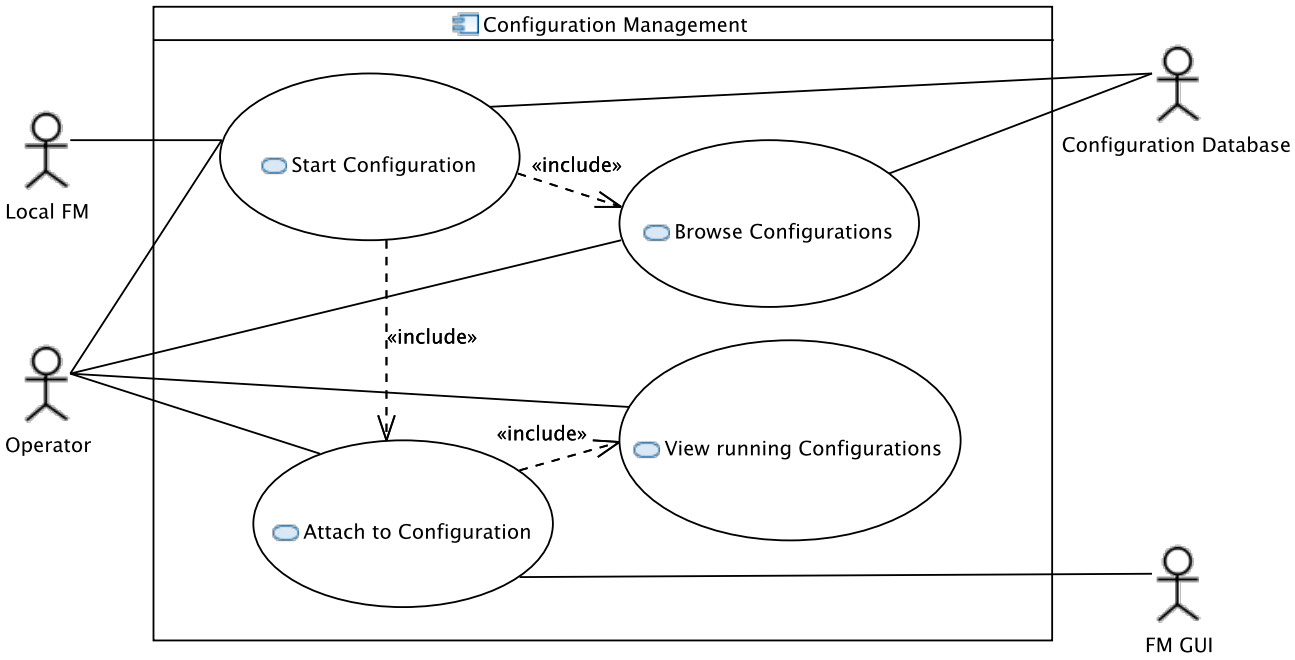


Figure 5.1: Configuration use case diagram

| **Use Case** | **"Start Configuration"** |
|---|---|
| Actors | Operator, Configuration Database |
| Priority | High |
| Description | The operator selects a configuration from the browser and starts it. After the RC creates the top function manager of the configuration, the operator is redirected to the FM's GUI. |
| Preconditions | The selected configuration is not yet running. |
| Functional references | F-C02 |

| **Use Case** | **"Attach to Configuration"** |
|---|---|
| Actors | Operator, FM GUI |
| Priority | High |
| Description | The GUI of the FM is displayed to the operator. |
| Preconditions | The Function manager is running. |
| Functional references | F-C03, F-C05, F-FM01, F-FM02 |

| **Use Case** | **"Browse Configurations"** |
|---|---|
| Actors | Operator, Configuration Database |
| Priority | High |
| Description | The operator browses a list of configurations that is available to them, the user they selected or the RC instance. |
| Functional references | F-C01 |
| Clarification needed | Is this a proper use case on its own? |

| **Use Case** | **"View running configurations"** |
|---|---|
| Actors | Operator |
| Priority | High |
| Description | The operator views a list of configurations running on the RC instance along with their top FM's state. |
| Functional references | F-C04, F-N03 |

### 5.3.3 Function Manager Use Cases

Use cases of the Run Control Framework providing functionality to Function Manager user code, relevant to a Function Manger developer when implementing their custom FM.

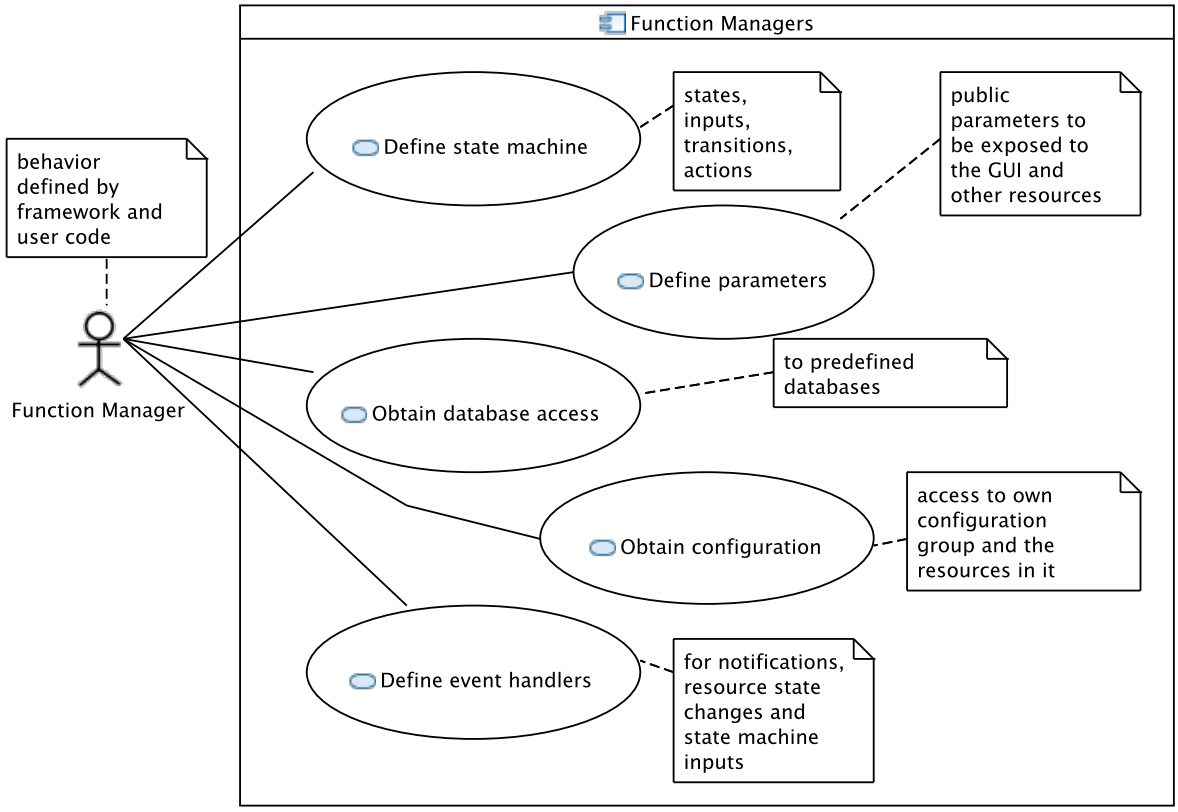


Figure 5.2: Function Manager use case diagram

| | |
|---|---|
| **Use Case** | **"Define state machine"** |
| Actors | Function Manager |
| Priority | High |
| Description | The RC provides a framework for the function manager to define its state machine and register it with the RC. This includes defining states, transitions and actions that are to be executed during transitions, when states are entered or left. In the default GUI, the state machine determines the available actions an operator can execute. |
| Functional references | F-FM06 |

| | |
|---|---|
| **Use Case** | **"Define parameters"** |
| Actors | Function Manager |
| Priority | High |
| Description | The function manager defines the parameters that are to be exposed by the RC. These parameters can be read and modified by the FM's GUI and other resources. |
| Functional references | F-FM07 |

| | |
|---|---|
| **Use Case** | **"Obtain database access"** |
| Actors | Function Manager |
| Priority | High |
| Description | The RC provides database access to the Function Manager. This is done for the databases commonly used by Function Managers. |
| Functional references | F-B01, F-FM03 |
| Clarification needed | Is the RC supposed to expose database connections to the FM or wrap all functionality? |

| | |
|---|---|
| **Use Case** | **"Access configuration"** |
| Actors | Function Manager |
| Priority | High |
| Description | The RC provides a function manager with the ability to access the configuration it belongs to. This allows the FM access to parent and child resources. |
| Functional references | F-FM08 |

| **Use Case** | **"Define event handlers"** |
|---|---|
| Actors | Function Manager |
| Priority | High |
| Description | The RC allows FMs to define handlers for various messages, events and notifications sent to them. After incoming messages are pre-processed by the RC and routed to their destination FM, the respective handler of the FM will be executed by the RC. |
| Functional references | F-N04, F-FM05, F-FM09, F-FM10 |

### 5.3.4 Function Manager GUI Use Cases

Use cases which describe the functionality provided by the Run Control Framework for Function Manager GUI development and operation.

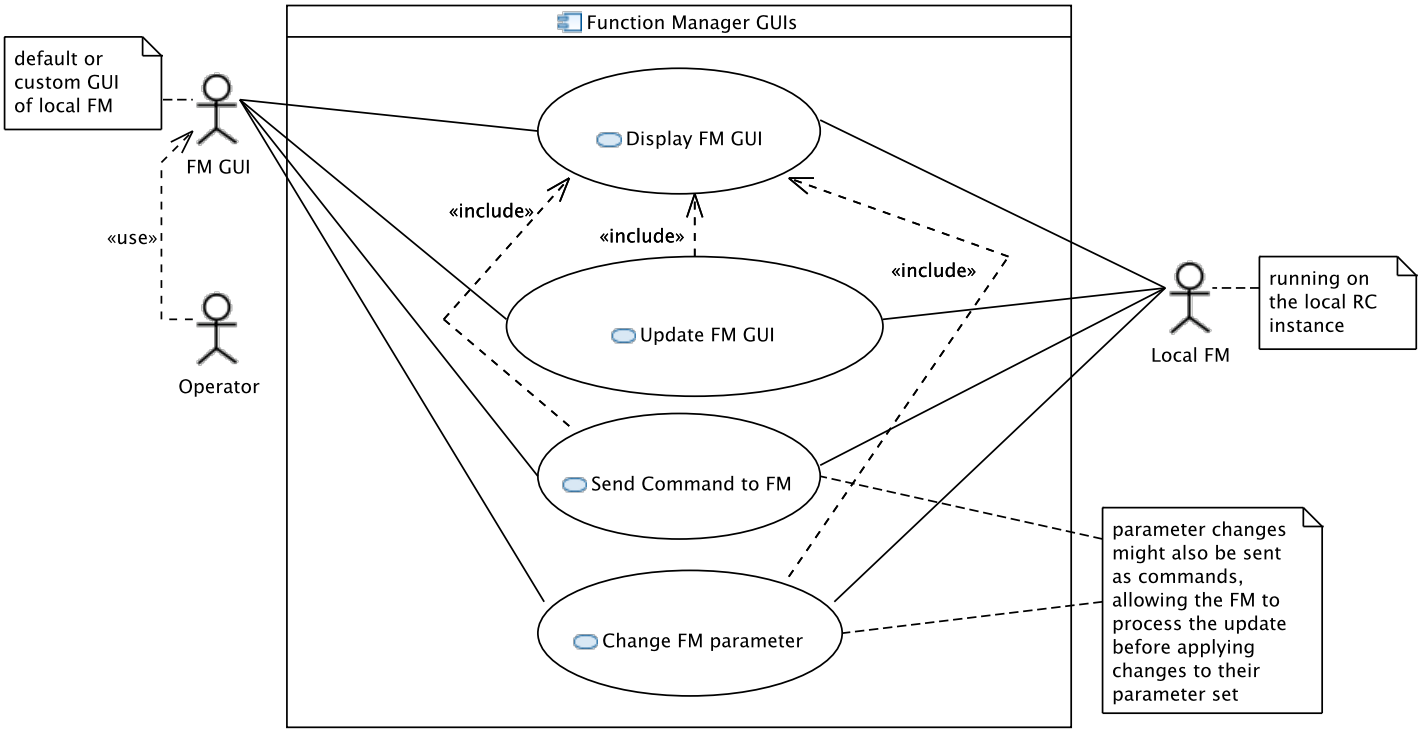


Figure 5.3: Function Manager GUI use case diagram

| **Use Case** | **"Display FM GUI"** |
|---|---|
| Actors | Operator using the FM GUI, Local FM via its network endpoint |
| Priority | High |
| Description | The FM GUI obtains the parameters of the FM it controls and renders the page accordingly, giving an overview of the status the FM is in, as well as giving options to send commands or change parameters. |
| Preconditions | The target function manager is running and the custom GUI, if defined, is deployed and accessible. |
| Functional references | F-C05, F-N02, F-N03, F-N07, F-FM01, F-FM02, F-FM07 |
| Non-funct. references | N-B01, N-FM01 |

| **Use Case** | **"Send Command to FM"** |
|---|---|
| Actors | Operator using the FM GUI, Local FM via its network endpoint |
| Priority | High |
| Description | Send a command to the FM, triggered by the operator using a control element in the GUI. Consequently, the FM's command event handler is executed or the FM's state machine transitions based on the received input. |
| Preconditions | The FM GUI is displayed to the operator. |
| Functional references | F-C05, F-N01, F-N01a, F-N07, F-FM01, F-FM02, F-FM07, F-FM09 |
| Non-funct. references | N-B01, N-FM01 |
| Clarification needed | Does it make sense to differentiate between commands and state machine inputs? A command might not result in a FM state change. |

| **Use Case** | **"Update FM GUI"** |
|---|---|
| Actors | The FM GUI, Local FM via its network endpoint |
| Priority | High |
| Description | Update of the FM GUI, usually triggered by an update from the FM's network endpoint due to a change in parameters or state. |
| Preconditions | The FM GUI is displayed. |
| Functional references | F-C05, (F-N02), (F-N03), F-N07, F-FM01, F-FM02, F-FM11 |
| Non-funct. references | N-B01, N-FM01 |
| Clarification needed | Is the FM state treated like any other parameter, or are different types of update notifications required? This depends on how the state is modelled in the FM framework and the user code itself. If the parameter set is provided by the framework, a state parameter which mirrors the FM state machine's state may be added and updated by the framework. |

| **Use Case** | **"Change FM parameter"** |
|---|---|
| Actors | Operator using the FM GUI, Local FM via its network endpoint |
| Priority | High |
| Description | The operator uses the FM GUI to change a FM parameter, usually by changing a value in the GUI. |
| Preconditions | The FM GUI is displayed. |
| Functional references | F-C05, F-N07, F-N09, F-FM01, F-FM02, F-FM07, F-FM10 |
| Non-funct. references | N-B01, N-FM01 |
| Clarification needed | Are parameter updates sent as commands? This would allow the FM to process and check the update before making changes to its parameter set. |

### 5.3.5 Network Communication Use Cases

Use cases which describe network functionality provided to Function Managers by the Run Control Framework.

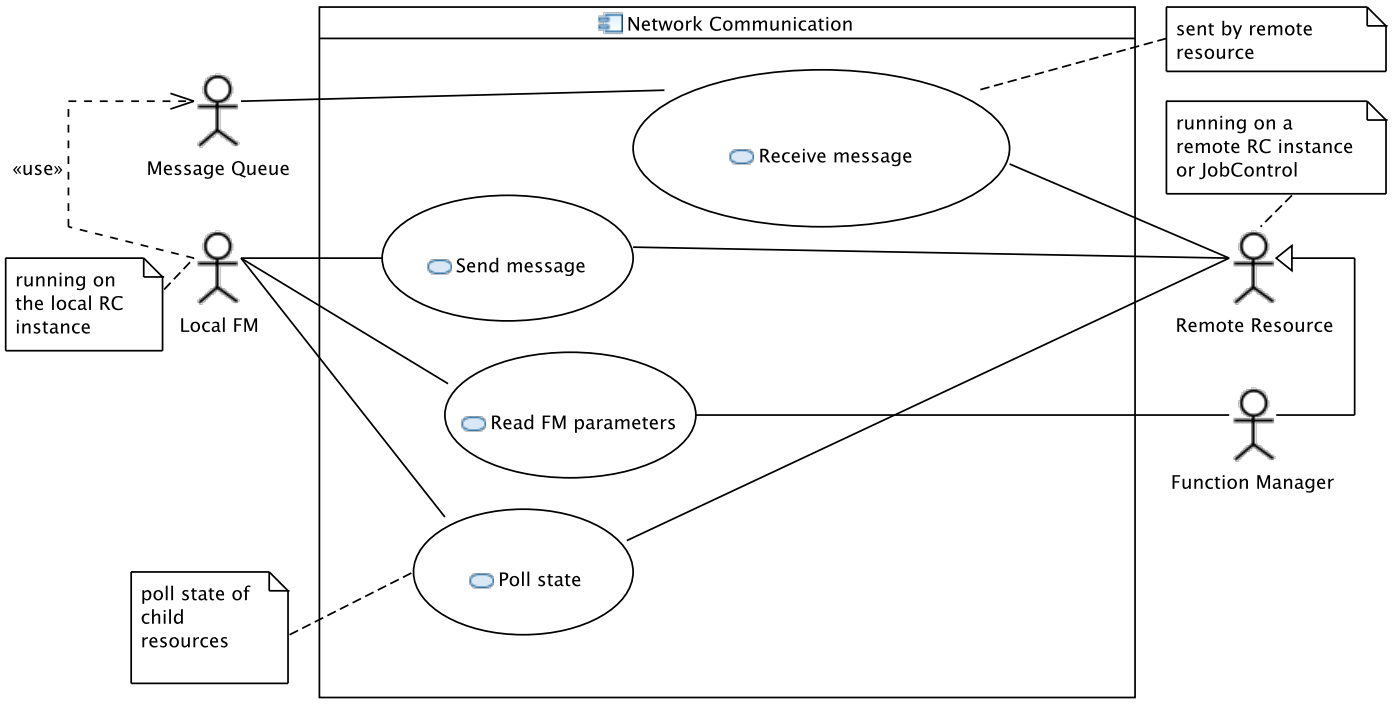


Figure 5.4: Network communication use case diagram

| | |
|---|---|
| **Use Case** | **"Receive message"** |
| Actors | Local FM via its message queue, Remote Resource |
| Priority | High |
| Description | The RC must provide a function manager with means to receive messages from its local and remote child resources. The content and meaning of these messages is resource- and FM-specific. The routing to the local FM is done by the RC based on routing rules or subscriptions. These rules are tuples of the form (sender, local FM), whereas the sender is the remote resource the message is sent by and the local FM is the function manager the message is forwarded to. |
| Preconditions | The local FM and target resource are running and a routing entry from the remote resource to the local FM exists. |
| Functional references | F-N04, F-N05, F-N06, F-FM05, (F-FM08) |
| Clarification needed | The routing might be done automatically by the RC framework, based on the parent-child relation between a FM and its resources. It might also be desirable to allow FMs to subscribe to messages from resources in their user code. Which format do these messages have (XML, JSON)? Which protocol is being used (REST, SOAP)? Is authorization of remote resources required? |

| | |
|---|---|
| **Use Case** | **"Send message"** |
| Actors | Local FM, Remote Resource |
| Priority | High |
| Description | The RC framework allows local FMs to send messages to remote resources or other local FMs. The content and meaning of these messages is resource- and FM-specific. |
| Preconditions | The local FM and target resource are running. |
| Functional references | F-N01, F-N01a, F-N08, F-N08a, F-FM04 |
| Clarification needed | Which format do these messages have (XML, JSON)? Which protocol is being used (REST, SOAP)? Are state machine inputs sent differently? |

| | |
|---|---|
| **Use Case** | **"Read FM parameters"** |
| Actors | Local FM, FM |
| Priority | High |
| Description | Allow local FMs to read the parameters of other local or remote FMs. |
| Preconditions | The local FM and target FM are running. |
| Functional references | F-N02, F-FM07, F-FM10 |
| Clarification needed | Is this different from the GUI reading the FM parameters? If yes, how does it differ? |

**Note** It should be clarified if Function Managers need to be able to directly set parameters of other Function Managers, as parameters between FMs are usually passed together with Inputs.

| **Use Case** | **"Poll state"** |
|---|---|
| Actors | Local FM, Remote Resource |
| Priority | High |
| Description | Allow local FMs to poll the state of another local FM or remote resource. This implies sending a synchronous message to obtain the current state of the target resource. |
| Preconditions | The local FM and the target resource are running. |
| Functional references | F-N03 |
| Clarification needed | Since state changes are supposed to be propagated using notifications, what are the applications for polling the state? It can be used to poll states of resources that do not send notifications to the RC, e.g. XDAQ service applications that are not yet leased. |

**Note** It may be sensible to unify communication to only send one type of message between FMs and FMs and their GUI. Combining this with sending parameter changes and inputs as commands, the message handling logic could be simplified.

### 5.3.6 Monitoring Use Cases

Use cases which provide monitoring information about the running Function Managers to an operator. Monitoring functionality is provided by a Run Control Framework instance for the Function Managers running locally on the instance. Monitoring as described by the following use cases provides means to verify that the framework and especially FM user code is working as intended.

| **Use Case** | **"View routing rules"** |
|---|---|
| Actors | Operator, Routing Table |
| Priority | Medium |
| Description | The operator retrieves a table of routing rules or subscriptions, which reflect the RC's routing behavior regarding incoming messages. This information can be used in order to verify the correct routing of messages and diagnose problems with the routing rules or subscriptions of function managers. |
| Functional references | F-N04, F-M02 |

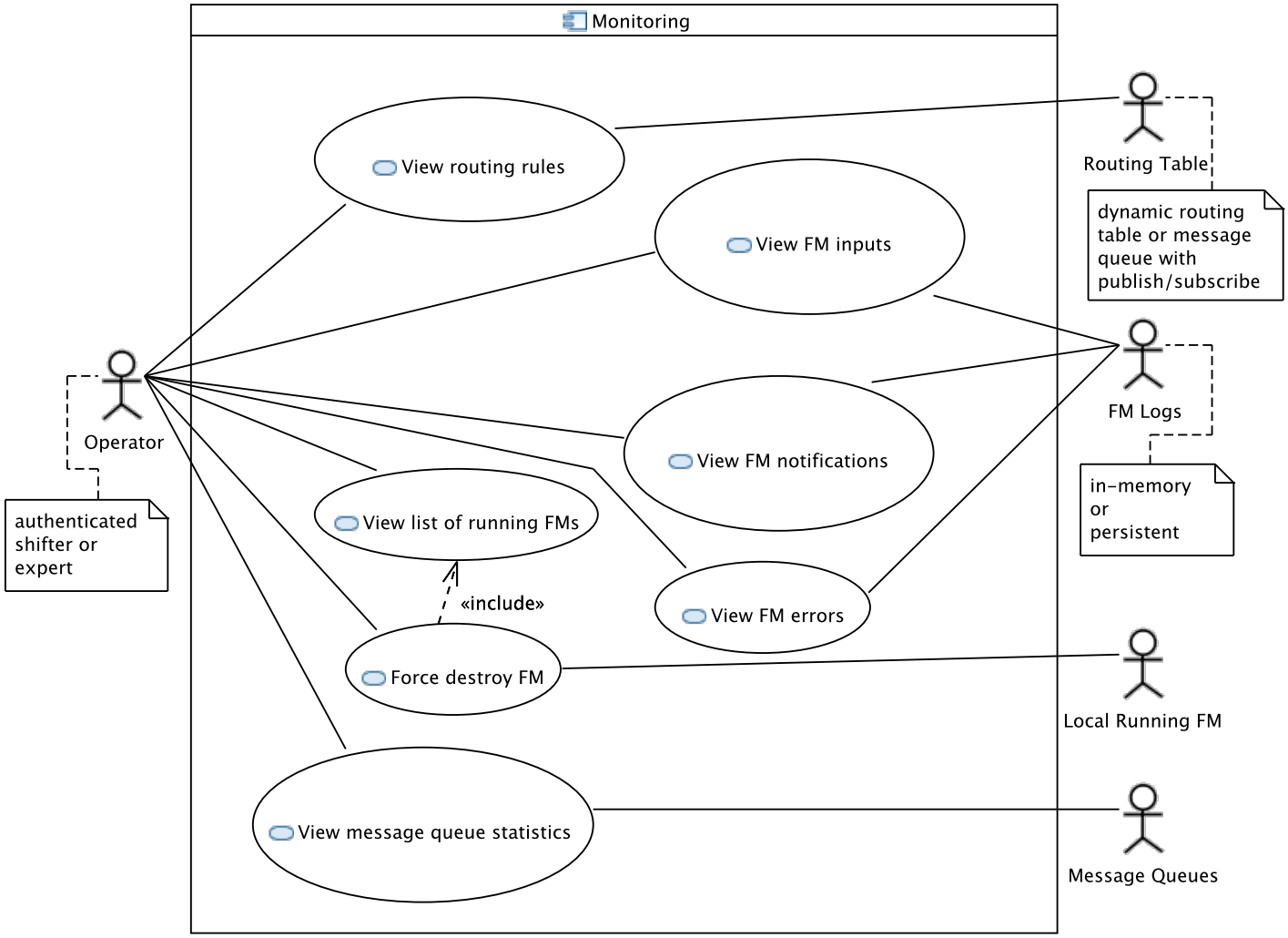


Figure 5.5: Monitoring use case diagram

| **Use Case** | **"View FM inputs"** |
|---|---|
| Actors | Operator, FM Input Log |
| Priority | Medium |
| Description | The operator retrieves a table giving an overview of the inputs and commands that were sent to function managers. |
| Functional references | F-M05 |
| Clarification needed | Is the input log written to disk or only kept in-memory? |

| **Use Case** | **"View FM notifications"** |
|---|---|
| Actors | Operator, FM Notification Log |
| Priority | Medium |
| Description | The operator retrieves a table giving an overview over messages received by the RC and which FM they were routed to and over messages sent by the local FMs to remote resources. |
| Functional references | F-M06, F-M08 |

| **Use Case** | **"View FM errors"** |
|---|---|
| Actors | Operator, FM Error Log |
| Priority | High |
| Description | The operator retrieves a table of errors encountered during function manager execution. |
| Functional references | F-M07 |

| **Use Case** | **"View list of running FMs"** |
|---|---|
| Actors | Operator |
| Priority | High |
| Description | The operator retrieves a list of function managers running on the local RC instance. This is different from a list of running configurations, since the top FM of a configuration might have child FMs in the same configuration and the running configuration view only shows the top FM. |
| Functional references | F-M03 |

| **Use Case** | **"Force destroy FM"** |
|---|---|
| Actors | Operator, local FM |
| Priority | High |
| Description | The operator destroys a local function manager forcefully, bypassing access control and ignoring parent-child relationships between FMs. This option is available from the list of running FMs. |
| Preconditions | The local FM to be destroyed is running. |
| Functional references | F-M04 |
| Clarification needed | This was implemented as "destroy backdoor" in the current RCMS. Is it really needed? Its main use is to clean up crashed FMs that no longer respond to regular actions. |

| **Use Case** | **"View message queue statistics"** |
|---|---|
| Actors | Operator, message queues |
| Priority | Medium |
| Description | Allows the operator to see the status of various message queues for incoming messages and notifications. In particular, a table of queues with differing priorities is displayed for each running local FM. |
| Functional references | F-M09 |

# Chapter 6

# Technologies and Prototypes

This chapter focuses on the evaluation of technologies. As evaluating every potentially suitable technology is out of the scope of this thesis, only technologies compatible with the current Run Control Framework are used to implement prototypes.

## 6.1 A look at other experiments

Experiments at CERN, at other institutes and in the domain of High Energy Physics (HEP) in general, face similar challenges in their Run Control software. This section gives a brief overview of the technologies used in experiments other than the CMS, including solutions designed for future upgrades which have not yet been deployed. While most information presented in this section can be found in papers published by the experiments, the information on preliminary plans was mostly obtained during meetings and talks at CERN and the ISOTDAQ 2018 [59].

In the following paragraphs, the technologies used to implement the Run Control and related services for various experiments is described. The information presented here was made available in scientific publications and might not be current or complete. For example, WinCC Open Architecture (WinCC OA) is often used in conjunction with SMI++ (State Machine Interface) [5] and JCOP (Joint Controls Project) [8], even if not explicitly mentioned in the respective publications.

**ATLAS (A Toroidal LHC ApparatuS)** The ATLAS experiment uses a C++-based Run Control system controlled by a Java GUI (IGUI). Their expert system is implemented in Java and uses the Esper [29] complex event processing (CEP) software for the definition and evaluation of rules against the monitoring data. For configuration management, ATLAS utilizes XML files and provides database wrappers for C++, Java and Python. Inter-process communication is realized using CORBA [68] middleware. The ATLAS Run Control utilizes LDAP (Lightweight Directory Access Protocol) to implement role-based access control. [3, 15]

**ALICE (A Large Ion Collider Experiment)** For their Control System, ALICE uses WinCC Open Architecture (WinCC OA, previously PVSS). SMI++ is used to implement state machine logic. [13, 14]

**LHCb (Large Hadron Collider beauty)** Like ALICE, LHCb uses WinCC OA together with JCOP for their Run Control [16].

**COMPASS (COmmon Muon Proton Apparatus for Structure and Spectroscopy)** The Run Control of the COMPASS experiment is implemented in C++ and controlled using a Qt [18] GUI. The configuration management of the experiment is web-based and makes use of technologies such as PHP, JavaScript and MySQL. [64, 2]

**NA62 (North Area; Kaon Factory)** The NA62 Run Control is implemented using WinCC OA [67, 61].

**Observations regarding Run Control systems of other experiments** It is notable that state machines are an established method of modelling and synchronizing the resources of experiments, usually in the form of a hierarchical structure, where the state of a parent resource is comprised of those of its child resources, while state machine inputs to a parent resource are propagated to its children.

Most experiments aim to automate the detection and recovery of common faults with so-called expert systems, providing the operators with expert knowledge without requiring the intervention of actual system experts.

In some of the experiments, the Run Control system directly communicates with the DAQ machines, processes and hardware, which makes technology choices such as WinCC OA and C++ more suitable compared to languages further away from the hardware level, like Java. For the CMS Run Control, being able to interface with low-level software or hardware is not important, as the XDAQ CMS online software represents an additional layer of abstraction below the Run Control.

While not all experiments provide a web-based interface, there are efforts to implement such interfaces to allow remote control of the experiment, especially by experts.

## 6.2 Technologies

In this section a non-exhaustive list of potential technologies for implementing a successor to the current CMS Run Control Framework is presented. Java-compatible technologies may also be used in order to implement improvements to the current framework.

**Web Technologies** With a web-based interface being a requirement of the CMS Run Control, established technologies such as HTML, CSS and JavaScript can be used to implement client-side interfaces. Web frameworks such as Angular (not to be confused with AngularJS), may improve the development life cycle of graphical user interfaces, with extensive tooling and documentation available. As a client-side web framework, Angular has excellent support for interfacing with REST or WebSocket-based web services. [4]

**Java with Spring Framework** The Spring Framework [82] is a Java framework and inversion of control (IoC) container. Spring realizes inversion of control in the form of dependency injection (DI). Beans and components created and managed by the Spring container can be automatically injected into other Spring-managed components. The Spring Framework also offers aspect-oriented programming capabilities and validation of bean constraints. Spring offers modules for web application development, providing MVC (Model View Controller), WebSocket (also with STOMP, the Simple Text Oriented Messaging Protocol) and REST (REpresentational State Transfer) web service support [83]. Using Spring AMQP (Advanced Message Queuing Protocol), Spring applications may interface with message brokers such as RabbitMQ [72]. Support for other message brokers is available.

**WinCC Open Architecture** SIMATIC WinCC OA (previously PVSS; not to be confused with WinCC) [79] is a SCADA (Supervisory Control and Data Acquisition) system by Siemens. It is platform-independent and suitable for the control of highly distributed systems. The CMS Detector Control System (DCS) was implemented in PVSS with JCOP. As mentioned earlier, many experiments also implemented their Run Control using WinCC OA, in combination with JCOP and SMI++ to define their state machine logic. WinCC OA has optional support for web interfaces [80]. In addition to graphical and dialogue-based programming, scripting using a C# API and the ANSI C-based "Control" language is supported. CERN offers comprehensive support for WinCC OA, including training courses [10].

**Python** If the Run Control Framework is to be rewritten, choosing another language besides Java is a possibility. As the CMS DAQ group's scripting language of choice, Python may be a suitable language to do so, with frameworks such as django [33] easing the development of web-based applications. Due to a lack of static typing, large Python projects may be prone to errors that can not be detected at compile-time.

**C++ and XDAQ** Alternatively, C++ may be used to implement the Run Control Framework, with XDAQ being written in C++ supporting this option. XDAQ's hyperdaq [1] may be used to expose web interfaces or APIs. Interfacing with XDAQ like the current Run Control does, would become easier if it itself was written in XDAQ.

**Go** Go [19] is a relatively young language, which is statically typed and compiles to native code, yet employs a garbage collector. Go is geared towards concurrent and distributed applications, making it a suitable choice for implementing a Run Control system.

**Communication and RPC** For the Java ecosystem, successors to the currently employed JAX-RPC and Axis are available, with more details being provided in the descriptions of the prototypes below. As an alternative to SOAP- and REST-based solutions, technologies such as gRPC [51] or Protocol Buffers [50] may be utilized for communication between Run Control instances. While not as suitable for consumption by client-side web applications compared to a REST web service, these solutions offer lower latency and better performance due to reduced protocol overhead. Message-based systems also favor asynchronous event-based communication, which is widely employed by the Run Control Framework.

**Message Broker** In addition to or instead of a web service, a message broker may be used as a communication middleware, to handle both communication between Run Control instances as well as between the Run Control Framework and client-side GUIs. Message brokers, not unlike CORBA, promote asynchronous communication. Messages are usually sent to queues (point-to-point) or topics (publish-subscribe), with the exact naming of the mechanics differing between broker protocols. In addition to asynchronous messaging, brokers also support synchronous RPC requests by means of waiting for response messages from the called service. When using synchronous requests with response callbacks, the remote procedure is recommended to be idempotent, as in the case of a server failure, the request message may still be stored in message broker and therefore processed again by the service. In case of the CMS Run Control, a reliable network can be assumed, as it is a non-functional requirement. With a publish-subscribe architecture, it would be trivial to attach monitoring applications to transparently record the messages exchanged between Run Control instances.

## 6.3 About the Prototypes

In order to evaluate the suitability of technologies, small prototypes have been implemented, with the findings presented in this chapter. The prototypes focus only on fulfilling a single or a subset of requirements of the Run Control Framework and are compared to the solutions currently in place, building on the suggested improvements described previously.

The prototypes were developed using IntelliJ IDEA and use Ant, Maven and Gradle as build systems. With the Axis and Jersey prototypes being deployed on a local Tomcat instance, the IDE was configured to attach its debugger to the Tomcat using the server's JPDA (Java Platform Debugger Architecture) interface.

The source code of the prototypes can be found in the following repository on the CERN GitLab: gitlab.cern.ch/pbrummer/bachelor-thesis-prototypes

## 6.4 Network Communication Prototypes

The current Run Control Framework uses Apache Axis [47] as a library for implementing communication between instances. Axis provides web services based on the JAX-RPC (Java API for Remote Procedure Calls) standard, which was superseded by the JAX-WS (Java API for Web Services) standard. In order to make use of more recent technologies and in an effort to simplify the web services of the Run Control Framework, multiple Java libraries targeted at the creation of web services have been evaluated.

The prototypes presented in this section concern themselves with implementing the network communication use cases for communication between Run Control instances and Function Managers and their GUIs. While the current Run Control Framework has an additional wrapper around its Axis web service for interfacing with the web-based FM GUIs, this additional layer is not required when communicating with client-side GUIs based on modern web technologies, assuming the web services are implemented using a web-accessible architecture such as REST.

**Example Web Service** The prototypes implement a simple state machine web service, allowing querying the current state as well as sending an input to change the state. The Java interface is defined as follows:

```
1 public interface StateMachineService {
2     String getState();
3     void processInput(String input);
4 }
```

Listing 6.1: State Machine Service Java Interface

**Implementation** The implementation stores the current state and maps inputs to target states. When an input is sent, the state changes to the target state as defined in the transition map or to "Undefined" if no transition for the input is defined.

### 6.4.1 Axis 2

As a successor to Axis, Apache Axis2 [34] implements the newer JAX-WS standard. While there are shared concepts between Axis and Axis2, the latter is rather a different project than a new version of Axis and was built on a redesigned architecture [34]. A migration guide from Axis to Axis2 is provided [46].

Like Axis, Axis2 supports the exchange of messages using SOAP and XML, although Axis2 additionally supports exposing RESTful web services [38], as well as JSON encoding for messages [37]. The evaluation will focus on SOAP and XML messages, as other prototypes will concern themselves with libraries targeted at REST web services.

**Axis2 Deployment** Axis2 can be deployed as a standalone application with an embedded web server or as a web application in a servlet container such as Tomcat [35]. For the Run Control Framework, the deployment of Axis2 in a servlet container is largely similar to the current deployment of Axis. For the purposes of the evaluation, Axis2 was deployed as a web application on a local Apache Tomcat 8.5 servlet container, with Axis being available at "http://localhost:8080/axis2/" and services deployed under the "services/" path.

**Configuration** The examples developed as part of the evaluation make use of the default configuration included with the Axis2 web application distribution. However, Axis2 offers fine-grained configuration of its components [42], which possibly implies significant configuration work for production usage as part of the Run Control Framework.

**Web service development** Axis2 allows for a top-down approach to developing web services by describing the functionality exposed by the service in a WSDL (Web Service Description Language) file. Axis2 supplies tools for generating Java source code and a service descriptor (services.xml) for the server as well as client stubs from the WSDL description. For this approach, multiple options to generate the Java code are available [39].

Alternatively, a bottom-up approach can be utilized, allowing developers to implement a POJO ("Plain Old Java Object") and generate a WSDL file from it. This WSDL file can in turn be used to generate the service descriptor and client stubs [39].

**JAX-WS** Axis2 supports JAX-WS annotations [36], making it able to serve web services implemented without any Axis2-specific configuration (such as a service descriptor) or library dependency. With the JDK-provided tool wsgen, a WSDL file can be generated from a JAX-WS annotated service, while wsimport can generate JAX-WS portable artifacts (client stubs; dynamic proxy client) from the WSDL file. Alternatively, a dispatch client can be used to communicate with the service using custom-built SOAP messages [36]. As JAX-WS is a specification and independent from actual implementations, changing the web service engine after implementing the service remains a possibility.

JAX-WS can be used to define services on the level of functionality provided by the service (using the @WebService annotation) or on the message level (using the @WebServiceProvider annotation and implementing the Provider interface), the latter allowing for access to the messages exchanged by the web service, which may be used to dispatch requests to services dynamically [36]. For the Run Control Framework, functionality-level definitions are more appropriate, as all service functionality is static and manual message handling and service dispatching adds another layer of complexity.

**Web service deployment** When developing a web service from a WSDL (top-down) or POJO (bottom-up), the service along with the generated or manually created service descriptor can be packaged into an AAR (Axis2 service archive) file and copied into the "WEB-INF/services" directory of the Axis2 installation, where it will be automatically picked up and deployed by Axis2 [39].

When using JAX-WS annotations, an Axis2-specific service descriptor is not required. Instead, the service is packaged into a JAR (Java Archive) file and copied into the "WEB-INF/servicejars" directory of the Axis2 installation. Axis2 will automatically scan the JAR file for classes containing JAX-WS annotations and deploy the services they provide [36].

**Dependency Injection** Utilizing Resource-Annotations, Axis2 can be used to inject dependencies such as the Web Service Context into the the service, making it available to the implementation [36]. More information on dependency injection will be provided with the Jersey and Spring prototypes.

### Prototype: POJO with service descriptor

For this prototype, the previously mentioned state machine service was implemented as a POJO with a service descriptor and deployed as an Axis2 service archive (AAR).

**Service Descriptor** In order to provide Axis2 with information on how to deploy the service, a service descriptor (META-INF/service.xml) [43] is required. The service descriptor used for the prototype is listed below:

```
1  <service name="StateMachineService" scope="application" targetNamespace="[...]">
2      [...]
3      <messageReceivers>
4          [...]
5      </messageReceivers>
6      <schema schemaNamespace="[...]"/>
7      <parameter name="ServiceClass">
8          axis2.service.StateMachineServiceImpl
9      </parameter>
10 </service>
```

Listing 6.2: Axis2 service.xml service descriptor

Noteworthy definitions include the scope (line 1), which is used to tell Axis2 whether to use a new service object per request or session or reuse the same object for the lifetime of the application. Message receivers (line 3ff) define how the service is exposed (e.g. SOAP, HTTP). The namespace definitions (line 1 and 6) have to match the namespaces defined in the WSDL. The service class parameter (line 7-9) tells Axis which implementation provides the described service. More options are available to be defined in the service descriptor [43].

**HTTP Binding** If the service is exposed by a HTTP binding in addition to a SOAP binding, the service is also accessible using regular HTTP requests, for example by the browser. This is especially useful for debugging purposes. In order to send an input to the service, the following request may be used (here, a "Start" input is sent):

```
http://localhost:8080/axis2/services/StateMachineService/processInput?input=Start
```

To query the state

```
http://localhost:8080/axis2/services/StateMachineService/getState
```

will result in a XML response containing the current state:

```
1  <ns:getStateResponse>
2      <ns:return>
3          Running
4      </ns:return>
5  </ns:getStateResponse>
```

In the default configuration, Axis2 exposes service with a HTTP binding, with the exception of JAX-WS services, where an explicit annotation is required:

```
@BindingType(value = HTTPBinding.HTTP_BINDING)
```

**Generated WSDL and XSD** Axis2 provides a generated WSDL description and XSD schema for deployed web services, which can be accesses by suffixing the service URL with "?wsdl" and "?xsd" respectively. For the purpose of the prorotype, the WSDL was explicitly generated using the java2wsdl script provided with Axis2 by running the following command:

```
1 java2wsdl.sh \
2     -cp build/classes \
3     -cn axis2.service.StateMachineService
```

Listing 6.3: Axis2 WSDL generation from Java POJO

Together with the classpath (line 2) containing the compiled Java class bytecode, the name of the class implementing the service (line 3) needs to be specified to the script. Additionally, the namespaces may be specified and have to match the ones specified in the service descriptor. Other options for the conversion are available [44].

Generated WSDLs are limited in their ability to describe the constraints of the service. For example, object parameters are marked as nillable by default, which may not be desired. If a bottom-up approach is used, manual editing of the generated WSDL files may be required. Additionally, parameters may be checked for invalid values by the service implementation.

**Client** The client stub for the prototype was generated using the wsdl2java script provided by Axis2:

```
1 wsdl2java.sh \
2     -uri ../StateMachineService.wsdl \
3     -p axis2.client \
4     -d adb \
5     -s
```

Listing 6.4: Axis2 client stub generation from WSDL

The path of the WSDL file is specified as an URI and may point to a website, allowing generation of client stubs directly from an Axis2-provided WSDL. Additionally, the package name of the generated client stubs may be specified (line 3), along with the databinding to use (line 4). The script may generate synchronous, asynchronous or both types of stubs. For the purposes of the prototype, only synchronous stubs (line 5) were generated, matching the behavior of the current Run Control Framework's web services. Other options for the conversion are available [44].

An example client demonstrating the usage of the generated client stub:

```
1 StateMachineServiceStub serviceStub = new StateMachineServiceStub();
2
3 StateMachineServiceStub.GetState getState = new
      StateMachineServiceStub.GetState();
4 String state = serviceStub.getState(getState).get_return();
5
6 StateMachineServiceStub.ProcessInput processInput = new
      StateMachineServiceStub.ProcessInput();
7 processInput.setArgs0("Start");
8 serviceStub.processInput(processInput);
```

Listing 6.5: Axis2 client example

First, a new instance of the client stub is created (line 1). The default constructor assumes that the service is running locally. In order to query the state, a new "GetState" request wrapper instance (line 3) is created, which is then passed to the web service (line 4). A response wrapper ("GetStateResponse") object is returned, with the result of the web service call being obtainable by calling the "get_return" method on the response. Similarly, the "processInput" method of the web service can be used by creating a request wrapper object (line 6), setting the request argument (line 7) corresponding to the "input" function parameter and using the client stub to send the request to the web service (line 8). In order to change the parameter names in the request and response wrappers from "args0" and "return" to something more sensible like "input" and "state", the WSDL file needs to modified manually, changing the parameter and return value name attributes from the default. After the change and regenerating the client stubs, the methods are named "setInput" (line 7) and "getState" (line 4) respectively.

### Prototype: JAX-WS service

The JAX-WS prototype implements the same web service as the previous prototype, using JAX-WS instead of an Axis2-specific service descriptor. A bottom-up approach is followed like before, the metadata usually contained in the service descriptor and WSDL description being expressed using Java annotations instead.

**Web service interface** The web service interface needs to be marked as a web service using an annotation as follows:

```
1 @WebService
2 public interface StateMachineService {
```

Listing 6.6: JAX-WS State Machine Service interface

**Web service implementation** The implementation can be annotated with additional attributes.

```
1 @WebService(
2     endpointInterface = "axis2.service.StateMachineService",
3     serviceName = "StateMachineService",
4     wsdlLocation = "META-INF/StateMachineService.wsdl")
5 public class StateMachineServiceImpl implements StateMachineService {
```

Listing 6.7: JAX-WS State Machine Service implementation

Using the "WebService"-annotation, the service interface that is implemented can be specified (line 2), as well as an optional path to the WSDL file (line 4). In this example, the WSDL file is packaged into the produced JAR file's "META-INF" directory. If a WSDL description is supplied, Axis2 delivers the WSDL file shipped with the service instead of dynamically generating one.

**Web service methods** By default, all methods of the annotated service are exposed. Optionally, methods in the web service interface can be annotated as well, providing additional information about them to the web service engine. Since JAX-WS annotations are retained during runtime, they can be read from the compiled Java classes, allowing the

WSDL generation tool to use the information specified using annotations. An example follows.

```
1 @WebMethod(operationName = "getState")
2 @WebResult(targetNamespace = "http://service.axis2/", name = "state")
3 String getState();
4
5 @WebMethod(operationName = "processInput")
6 void processInput(
7         @WebParam(name = "input", targetNamespace = "http://service.axis2/")
8                 String input
9 );
```

Listing 6.8: JAX-WS State Machine Service interface methods

Specifying the result (line 2) and parameter (line 7) names allows the WSDL generator to include them in the WSDL file and makes it possible to improve the naming of methods of the generated client stub without requiring manual editing of the WSDL file (as was the case with the Axis2 client example). In addition, "RequestWrapper" and "ResponseWrapper" can be specified in order to reference Java classes used to wrap service requests and responses. These classes do not have to be created manually, but are instead generated with the "wsgen" command provided by the JDK:

```
1 wsgen -keep \
2     -cp target/classes \
3     axis2.service.StateMachineServiceImpl \
4     -d target/classes \
5     -s src/main/java \
6     -wsdl
```

Listing 6.9: JAX-WS wrapper classes and WSDL generation

The command is executed on the compiled Java classes, with the classpath (line 2) and the service implementation class name (line 3) specified. Class (line 4) and Java source code (line 5) output directories can be specified. Optionally, a WSDL may be generated (line 6).

**Client** If a WSDL was generated using the "wsgen" command, it may be used to generate a JAX-WS client stub for the web service using the "wsimport" command:

```
1 wsimport -keep \
2     -p axis2.service.client \
3     -d target/classes \
4     -s src/main/java \
5     src/main/resources/META-INF/StateMachineService.wsdl
```

Listing 6.10: JAX-WS client stub generation

In addition to the target package for the client stub Java classes (line 2), the class and source code destination directories are specified (line 3 and 4), followed by a path to the WSDL file. Alternatively, an URL pointing to the WSDL may be specified.

An example client using the generated JAX-WS client stub:

```
1 URL endpoint = new
      URL("http://localhost:8080/axis2/services/StateMachineService");
2 QName qname = new QName("http://service.axis2/","StateMachineService");
3
4 StateMachineService_Service service = new StateMachineService_Service(endpoint,
      qname);
5 StateMachineService serviceImpl = service.getStateMachineServiceImplPort();
6
7 serviceImpl.processInput("Start");
8 String state = serviceImpl.getState();
```

Listing 6.11: JAX-WS State Machine Service client example

Using the default constructor, the service URL is read from the service's WSDL file. Alternatively, an endpoint can be specified explicitly (line 1 and 2). When using the endpoint to instantiate the client stub (line 4), the endpoint URL will be queried to obtain the WSDL provided by the web service engine. Afterwards, a port instance implementing the service interface can be obtained (line 5) and consequently utilized to use the web service (line 7 and 8). The example client interfaces with the service on a method level. Alternatively, a dispatch client can be developed without requiring the service's WSDL and a generated client stub.

### Complex data types

While the service implementations so far only dealt with Java Strings, more complex data types are in use by the Run Control framework's web service. The following example will demonstrate how to implement web services with complex method parameters and return types using JAX-WS together with the wsgen and wsimport tools.

The previous state machine web service interface is modified to contain methods with complex return and parameter types:

```
1 State getState();
2 void processInput(Input input);
```

Listing 6.12: Web service interface with complex data types

with the State class consisting of a single String attribute representing the state's name:

```
1 public class State {
2     private String name;
```

Listing 6.13: State class

while the Input class consists of a name as well as a set of parameters:

```
1 public class Input {
2     private String name;
3     private Set<InputParameter> parameters;
```

Listing 6.14: Input class

with InputParameter being a static inner class of Input, representing a simple String-based key-value pair:

```
1 public static class InputParameter {
2     private String name;
3     private String value;
```

Listing 6.15: InputParameter class

Constructors, getters and setters were omitted to keep the listings brief.

Like before, the WSDL specification and request wrappers were generated using the "wsgen" command, followed by the "wsimport" command to generate a client stub from the WSDL file. The web service description generated by the "wsgen" command only takes into account public fields as well as variables accessible by both a getter and setter. Additionally, annotations may be used to influence the generated XML schema and to declare which fields to include [28].

In order for the web service and client stub to work, all complex types are required to have a default constructor with no arguments. This is due to how JAXB (Java Architecture for XML Binding) works and can be worked around by developing a XmlAdapter, allowing for conversion between types and adapted types, effectively requiring multiple implementations of the same type. Fields that require an adapter, need to be annotated with the "XmlJavaTypeAdapter" annotation, specifying the adapter to use when converting to/from XML.

**Collections** Despite the Input class specifying a set of parameters, the client stub generated from the WSDL file specifies a list instead. This is due to how collections are expressed in XML schemes and the lack of a constraint signifying uniqueness of elements in a collection. In generated client stubs, collections have no setters, but are initialized and expected to be modified using the reference returned by the getter [87]. An example of this can be seen in line 7 of listing 6.16.

**Example client** Only empty constructors are available for the generated client stub classes. Additional convenience constructors may be specified by manually modifying the generated code. An example client using the client stub generated for the service using complex types follows.

```
1  Input startInput = new Input();
2  startInput.setName("Start");
3
4  InputParameter param1 = new InputParameter();
5  param1.setName("param1");
6  param1.setValue("value1");
7  startInput.getParameters().add(param1);
8
9  serviceImpl.processInput(startInput);
10
11 State state = serviceImpl.getState();
```

Listing 6.16: Complex type example client

First, a new "Start" Input is instantiated (line 1 and 2). Next, a parameter (key-value pair) is created and added to the Input object (line 4-7). Afterwards, the parameterized input is sent to the web service (line 9), after which the current state is obtained (line 11).

**Conclusion**

While the configuration options of Axis2 are different from those of Axis and Axis2's functionality and modularity make it a highly flexible web service engine, most of these advanced features are not required by the Run Control Framework's web service, making Axis2 an unnecessarily complex choice. Migrating from Axis to Axis2 when using a top-down approach and starting with the WSDL might only yield performance advantages, while no significant simplifications in the configuration or usage can be expected. Implementing the RC web services with JAX-WS, starting from annotated Java classes, would result in a simplified, standardized and engine-independent web service. However, bottom-up approaches with generated WSDL files often require manual editing of said WSDLs, for example to implement constraints on the level of the web service description. Not many options are available when generating a client stub from a WSDL, although manual changes to the generated Java source may still not be necessary. If the decision is made to implement the Run Control's web service in compliance with JAX-WS, there might be more suitable web service engines available.

**Apache CXF** Compared to Axis2, Apache CXF [40] has a simpler API and is more focused on supporting JAX-WS and related standards. While Axis2 favors standalone web services, CXF provides easier integration into existing applications and frameworks such as Spring. CXF emphasizes a bottom-up design approach and allows most configuration to be done using annotations and code, instead of writing and editing XML files. [45, 41]

**Spring WS** Spring WS is another alternative to Axis2 and Apache CXF. It favors a top-down contract-first approach and does not adhere to the JAX-WS standard. Opposed to JAX-WS, it allows for other data bindings than JAXB. As a Spring project, Spring WS is easy to integrate with Spring Core and Spring MVC and can leverage Spring Boot for easy standalone deployment. [84]

### 6.4.2 REST, JAX-RS and Jersey

As an alternative to SOAP-based web services, REST (representational state transfer) or RESTful web services most commonly make use of the widely supported HTTP protocol and facilitate the development of stateless APIs and loosely coupled distributed applications.

JAX-RS (Java API for RESTful web services) [60] is a defined standard for developing REST web services in Java and as such, an alternative to the JAX-WS and the deprecated JAX-RPC standards.

When using the REST architectural style with HTTP as transport protocol, any HTTP client is capable of consuming the web service, making server and client implementations largely independent. With WADL (Web Application Description Language) definitions, REST web services can be described similarly to how SOAP web services are described using WSDL. However, WADL definitions are usually far less concise and not sufficient to generate a full client from, making the loose coupling come at the expense of necessitating the investment of more work into implementing the client, at least when using a bottom-up code-first approach to designing the web service. As a top-down approach, advanced description languages such as Swagger [85] may be used to generate web services that conform with the OpenAPI specification (OAS) [58] or otherwise allow the generation of client- and server-side code. The OAS itself does not emphasize a top-down design approach and may also be adhered to when developing REST services code-first. Apart from WADL and advanced description languages, RESTful web services may also be described using WSDL 2.0 [63].

While SOAP is an appropriate choice of communication protocol for the Run Control when it comes to exchanging data between RC instances, the arising need to provide an API that is accessible by external tools and even browser clients, makes RESTful web services worth looking into. Together with a description language like Swagger, a more concise contract between the client and server can be defined, while retaining the flexibility of RESTful web services. REST is often mentioned in connection to the HATEOAS (Hypermedia As The Engine Of Application State) constraint, however, dynamic service discovery is neither required nor appropriate for the Run Control's web services.

**Jersey** Jersey is a reference implementation of the JAX-RS standard, yet provides an extended API, offering more functionality than defined in the standard. [20, 23]

**Deployment** Jersey emphasizes embedded deployment. Added as a dependency or using a Maven quickstart archetype, Jersey is deployed as a servlet in an application container such as Tomcat. While the Jersey servlet container is responsible for exposing the RESTful web services, it can easily be embedded as part of an existing application.

**Configuration** In order for the service implementations to be discovered, a parameter to the Jersey servlet containing the package name to be searched must be specified:

```
1 <init−param>
2     <param−name>jersey.config.server.provider.packages</param−name>
3     <param−value>cern.cms.rcms_jersey_example.api</param−value>
4 </init−param>
```

Alternatively, the fully qualified name of a JAX-RS Application extending the "ResourceConfig" class may be specified:

```
1 <init−param>
2     <param−name>javax.ws.rs.Application</param−name>
3     <param−value>cern.cms.rcms_jersey_example.StateMachineService</param−value>
4 </init−param>
```

which in turn defines the packages to search in its implementation by calling a Java method (with "true" signifying recursive service discovery):

```
1 packages(true, "cern.cms.rcms_jersey_example.api");
```

Listing 6.17: Jersey web service discovery

Additionally, the mapping of the Jersey servlet may be specified, determining the path under which the REST resources will be exposed:

```
1 <servlet-mapping>
2     <servlet-name>Jersey Web Application</servlet-name>
3     <url-pattern>/webapi/*</url-pattern>
4 </servlet-mapping>
```

**WADL** Similarly to how Axis2 exposes WSDL files for the deployed services, Jersey provides a WADL file for all deployed services, available at:

```
1 http://localhost:8080/rcms-jersey-example-service/webapi/application.wadl
```

with "rcms-jersey-example-service" being the name of the deployed webapp. The WADL may be used to generate a client skeleton, it however does not contain any information about the complex data types used by the service.

**Dependency Injection** Jersey uses HK2 [69] for dependency injection. Since the web services are instantiated and made accessible by the Jersey container servlet, resulting in inversion of control (IoC), dependency injection may be used to obtain objects required by the service. The dependency injection has to be made aware of classes it is supposed to inject, requiring a so-called binder to be registered. [22]

```
1 public class DependencyBinder extends AbstractBinder {
2     protected void configure() {
3         bind(this.stateMachineRepository).to(StateMachineRepository.class);
```

Listing 6.18: Dependency Injection Binder example

This binder registers an instance to a class, leading to HK2 injecting the registered instance whenever an injection for the class is requested. This corresponds to the singleton pattern, as the same object is injected for all requests. Alternatively, classes and factories can be bound, resulting in a new instance being created for each injection or a method being called to obtain the instance respectively.

**Validating Constraints** In Java, constraints such as "NotNull" may be expressed using annotations. With bean validation, Jersey supports validating these constraints, allowing web services to define parameters as not-nullable and making the web service engine reject requests which violate these constraints. In order to enable bean validation, an additional dependency is required. [21]

**Example Service** With the service interface being the same as for the Axis2 JAX-WS example with complex data types, "NotNull"-annotations have been added to the parameters of the service methods. An example of the service implementation follows:

```
1  @Path("state-machine")
2  public class StateMachineResourceImpl implements StateMachineResource {
3
4      @Inject
5      public StateMachineResourceImpl(StateMachineRepository repository) [...]
6
7      @Path("state/{sm}")
8      @GET
9      @Produces(MediaType.APPLICATION_JSON)
10     public State getState(
11         @NotNull @PathParam(value="sm") String stateMachine) [...]
12
13     @POST
14     @Path("{sm}")
15     @Consumes(MediaType.APPLICATION_JSON)
16     public void processInput(
17         @NotNull @PathParam(value="sm") String stateMachine,
18         @NotNull Input input) [...]
```

Listing 6.19: Jersey web service example

The path the web service is made available under is defined using the "Path"-annotation (line 1). Further path annotations on methods (line 7 and 14) are relative to the service path. In the constructor, an object instance is injected (line 4f) and made available to the service. Injecting instances in constructors compared to injecting the fields directly has the advantage of allowing for the creation of unit tests, where dependency injection is usually not available. The actual value of the "sm" placeholder in the method paths (line 7 and 14) is passed to the method as an argument (line 11 and 17) annotated with the "PathParam" annotation. The HTTP verb annotations (line 8 and 13) define which kind of request to the service causes a method to be called, while the "Produces" and "Consumes" annotations (line 9 and 15 respectively) inform Jersey that responses and requests need to be serialized and deserialized to and from JSON. Other message formats, such as XML, are available. Regular arguments such as the input parameter (line 18), are parsed from the request body.

**Example Client** The example client was implemented without code generation, demonstrating Jersey's REST client capabilities.

```
1  public State getState(String stateMachine) {
2      return this.client.target(this.restEndpoint)
3              .path("state").path(stateMachine)
4              .request(MediaType.APPLICATION_JSON)
5              .get(State.class);
6
7  public void processInput(String stateMachine, Input input) {
8      this.client.target(this.restEndpoint)
9              .path(stateMachine)
10             .request(MediaType.APPLICATION_JSON)
11             .post(Entity.entity(input,MediaType.APPLICATION_JSON));
```

Listing 6.20: Jersey example client

In this case, the client implements the same interface as the service itself and also shares the complex "State" and "Input" types with the service. Paths and path parameters are constructed using builder methods (line 3 and 9), with the request types being specified in line 5 and 11. Conversion from and to JSON is handled automatically by Jersey. Using the client is as simple as specifying the URL to the REST endpoint and calling the methods. Error handling is not implemented in this example.

**Complex data types** Due to the largely automated conversion between Java objects and JSON, using complex data types is straightforward with Jersey. Empty constructors, which may be private, are still required in order for Jackson to deserialize complex types from JSON. Alternatively, the Java classes may be supplemented with Jackson annotations, defining methods and properties to be used during instantiation by the JSON parser. Available Jackson annotations can be found at [30].

**RESTeasy** RESTeasy is an alternative solution to Jersey, which works largely similar. [17]

## 6.5 Backend Prototype: Spring

The Spring prototype serves to evaluate the suitability of available Spring components and modules for the implementation of a Run Control system. As a communication architecture, REST was chosen. The prototype is deployed locally using Spring Boot [81] and made available using its embedded Tomcat servlet container. The Spring prototype is built using Gradle [57].

**Use cases** The uses cases partially covered and explored by this prototype are the Function Manager use cases "Define state machine", "Define parameters" and "Define event handlers". Furthermore, the network communication use cases "Read FM parameters", "Poll state", "Receive message" and "Send message" are implemented.

**Function Manager Hierarchy** The prototype consists of a Top Function Manager with two child (L1) Function Managers. All Function Managers are executed within the same Spring Boot instance, however communication between them is limited to the REST API, effectively emulating distributed execution.

**State Machine** The Function Managers implement the same state machine, allowing for states Ready, Configured, Running, Error and inputs Configure, Halt, Start, Stop, Reset.

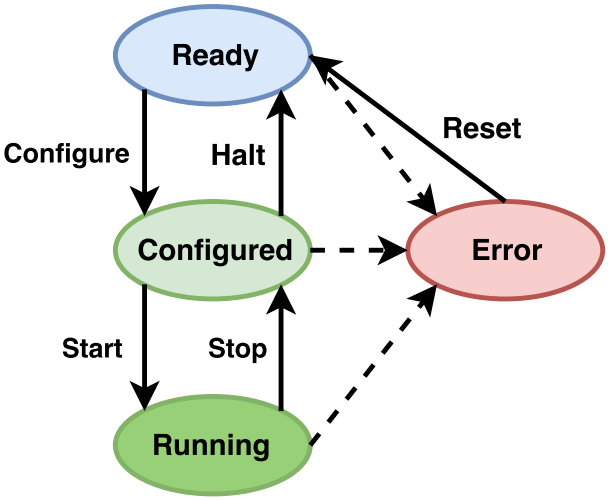


Figure 6.1: Spring Prototype: Function Manager state machine

State machine transitions are realized as a 3-tuple (current state, input, target state) and stored in a map of maps, with the key of the first map being the current state, the key of the inner map being the input and the value being the target state. Transition handlers are defined using the "Transition" method annotation, which optionally supports defining an origin and target state, as well as an input state on which to execute.

```
1 @Transition(from = "Running", input = "Stop", to = "Configured")
2 public void stopAction(State from, Input input, State to) throws
      TransitionException {
```

Listing 6.21: Spring Prototype: Transition annotation example

Since Java annotations do not support custom types as arguments (as of Java 9), Strings are used. This makes specifying handler methods error-prone, as reflection is required to detect annotated methods and mistyping a state or input name does not result in a compile-time error. To work around this limitation, state enums may be used, if compatible with the state machine implementation.

**Communication** The Top Function Manager controls the L1 FMs by forwarding inputs to them, adding configuration parameters where appropriate. The L1 FMs report state changes to their parent, allowing the Top FM to react to changes in its child FMs' states without polling them periodically.

**REST Controller** The definition of Spring REST controllers is very similar to that of JAX-RS web services as demonstrated in the Jersey example. While the names of the annotations differ, the description of the behavior is essentially the same. By default, Spring MVC REST controllers produce and consume JSON, yet can be configured to produce or consume other formats, including XML. Multiple formats can be specified in parallel, allowing the web service to respond using the data format requested in the HTTP request's headers. As with the Jersey prototype, Spring uses Jackson [31] for JSON serialization and deserialization.

**REST Clients** In order for Function Managers to consume the REST services to communicate with other FMs, a REST client was implemented using Spring's RestTemplate,

which is comparable to the Jersey REST client described above, with mostly syntactical differences.

**Parameters** The Function Manager and state machine input parameters are realized as a generic Java class, able to hold values of arbitrary type:

```
1 @JsonTypeInfo(
2         use = JsonTypeInfo.Id.NAME,
3         include = JsonTypeInfo.As.PROPERTY,
4         property = "javaType")
5 public class Parameter<T> {
```

Listing 6.22: Spring Prototype: Generic parameter type

Allowing any type to be used as a parameter value is problematic, as the deserialization of JSON on the receiver side becomes non-trivial. To circumvent this restriction, Jackson annotations can be used to include Java type information in the produced JSON [32]. Since Java does not retain generic type parameters at runtime, generic type information is not available to be included in the JSON, making collections and maps especially difficult to correctly deserialize and requiring type information for every object in the collection or detailed type information on the receiver side.

The current Run Control Framework solves this problem by defining wrapper classes for the allowed parameter types, consequently restricting developers and requiring additional wrapping and unwrapping, as well as adding complexity to the web service. While Jackson can be instructed to include type information in the produced JSON for simple types, for collections and maps, including the fully qualified Java class name for parameter objects would be required, along with implementing a custom deserializer. Since the generic type can not be obtained at runtime due to Java's type erasure, the type of a collection must be known in advance. In this case, collections may only contain objects of the same type. Alternatively, collections containing only parameters may be used, ensuring that each element in the collection includes its own type information when serialized. Automatic wrapping and unwrapping of simple types may also be provided by the framework.

Another option would be to require exact parameter types to be known on the receiving side in advance. This can be assumed either way, as the logic dealing with another Function Manager's or an input's parameters must be aware of their type. In this case, a lazy deserializer may be implemented, retaining the JSON and only producing Java objects when explicitly requested, with the request required to include the type that the parameter is expected to have. In this case, deserialization would not be done by the framework automatically, but only when a Function Manager's user code explicitly requests the value of a parameter.

An expansion on the previous idea would be to employ parameter models. If all parameter types of the monitored Function Manager are known, its parameter set can be reconstructed by the receiver. In this case, the framework could automatically create and maintain a parameter set for each child Function Manager, allowing the parent FM to access the parameter set directly as if it was a local object. This solution is employed in the Level-0 Automator FM [7] to model the parameters of the Level-0 FM and its

Subsystems. Since parameter change notifications are asynchronous and may not arrive in order, timestamps should be employed to avoid updating the parameter model with outdated information.

**WebSocket and STOMP** A Spring module for WebSocket support is available. To handle publish-subscribe messaging with STOMP over WebSocket. To accomplish this, an integrated message broker is used. [84]

**Comparison with Jersey** In the context of the minimal Spring Prototype, Jersey could have provided the same functionality regarding the REST web services and dependency injection. However, the Spring prototype only makes use of a small fraction of the Spring Framework's capabilities. The Spring Framework also provides integration options for Jersey, allowing its usage as a JAX-RS compatible REST service provider.

### 6.5.1 Spring with Message Broker

With a dedicated message broker like RabbitMQ, different approaches are feasible. While a message broker is capable of replacing the REST API, it may also be used as a complement, handling asynchronous communication such as notifications while the web service remains in use for synchronous communication such as sending state machine inputs or manipulating parameters.

**Setup** Different message broker setups are imaginable. A central message broker for each Run Control setup (production, integration etc.) would loosen the coupling between the distributed Run Control instances, removing the need to specify hostnames and ports of remote instances in the configuration. However, a centralized message broker would slow down communication between Function Managers running on the same RC instance, since in this case, direct communication would be an option. However, if monitoring tools are to be attached to the broker, local notifications would have to be replicated to a central broker in any case. Another option would be to run one message broker service per Run Control instance and configure the message brokers to automatically deliver messages to the appropriate destination broker.

**GUI** Message brokers such as RabbitMQ also offer exposing their message exchanges via WebSockets, allowing client-side GUIs to directly attach to the broker to obtain information about parameter and state changes of the monitored Function Manager.

# Chapter 7

# Summary and Outlook

The goal of this thesis was to formalize the core requirements of the Run Control Framework and evaluate technologies regarding their suitability to implement a new Run Control Framework for the CMS experiment or their suitability to significantly simplify the current Run Control Framework. To this end, a number of technologies were presented and evaluated, with the result of many promising options for simplification being available. As such, the results of this thesis are only to be seen as a starting point for further investigation, prototyping and informed design.

One of the largest potentials for simplification stems from implementing more rigorous separation of concerns, which is supported by modern web technologies. Implementing the Run Control Framework as a backend-only service with a web-accessible API allows for decoupled development and deployment of user interfaces. An extension of this approach would be to explore implementing the Run Control Framework following a microservice architecture.

In order to implement a web service which is able to serve Function Manager GUIs as well as remote Run Control instances, architectures such as REST are well-suited. However, the asynchronous nature of the Run Control may be favored by approaches making use of message brokers and WebSockets as an alternative to blocking long polling. Providing a synchronous web service API together with an asynchronous messaging solution allows the Run Control to profit from the advantages of both technologies.

As seen by taking a look at other experiments, there is a discernable trend towards making use of web technologies to provide user interfaces for monitoring and control purposes. This supports the initial design decision to use web technologies as a foundation for the Run Control Framework. There is also a clear trend towards improving usability, much effort spent on providing operators with useful and efficient tools to reduce the number of interventions where system experts need to get involved.

**Outlook** The outlook provides suggestions to complement the insights gained from the thesis, as well as further steps to take when choosing the technologies to implement the Run Control Framework in.

Due to the limited scope of this thesis, the area of configuration management was not analyzed, even though this particular area was identified to have great potential for

simplification of the Run Control system. Before developing a new Run Control Framework, options for simplifying configurations should be looked into, including configuration formats, ways of storing configurations and options to ease generating and editing them.

While the prototypes presented in this thesis were focused on technologies compatible with today's Run Control Framework, other options should be evaluated as well, especially non-Java technologies such as WinCC OA and Go.

With today's network bandwidth and the computing power of individual machines, the differences in performance of the various communication solutions appear to be negligible. However, as an extension to the results of this thesis, performance benchmarks regarding message throughput and latency may prove to be useful when deciding on which communication technologies to use.

In addition to prototyping backend technologies, technologies to implement the graphical user interfaces for Run Control instances and Function Managers should be investigated, as they not only have a great impact on the usability of the system, but may also influence the choice of communication technology used by the Run Control Framework.

# List of Figures

# List of Tables

# Listings

# Bibliography


[1] Luciano Orsini et al. *CMS Online Software - Developer's Manual*. URL: https://edms.cern.ch/file/998717/1.0/CMSOSDM_D_V2.2.pdf (visited on 30/11/2017).

[2] M. Bodlak et al. *Development of new data acquisition system for COMPASS experiment*. URL: https://cds.cern.ch/record/2263114/files/10.1016_j.nuclphysbps.2015.09.153.pdf?version=1 (visited on 24/02/2018).

[3] G Anders et al. *Intelligent operations of the data acquisition system of the ATLAS experiment at LHC*. URL: http://inspirehep.net/record/1372959/files/10.1088_1742-6596_608_1_012007.pdf (visited on 24/02/2018).

[4] Powered by Google Angular Developers. *Angular*. URL: https://angular.io/ (visited on 24/02/2018).

[5] B.Franek, C.Gaspar and CERN. *SMI++ - State Manager Interface*. URL: http://smi.web.cern.ch/smi/ (visited on 24/02/2018).

[6] Frederick Bordry. *LHC schedule beyond LS1*. URL: https://lhc-commissioning.web.cern.ch/lhc-commissioning/schedule/LHC%20schedule%20beyond%20LS1%20MTP%202015_Freddy_June2015.pdf (visited on 27/11/2017).

[7] Philipp Brummer. *LV0 Automator FM - 2.2 Parameter replication and data models*. 24th Feb. 2016.

[8] CERN. *JCOP Framework*. URL: http://jcop.web.cern.ch/jcop-framework (visited on 24/02/2018).

[9] CERN. *The Large Hadron Collider*. URL: https://home.cern/topics/large-hadron-collider (visited on 27/11/2017).

[10] CERN. *WinCC-OA Service*. URL: https://readthedocs.web.cern.ch/display/ICKB/WinCC-OA+Service (visited on 24/02/2018).

[11] The CMS Experiment at CERN. *How CMS Works*. URL: https://cms.cern/detector (visited on 24/02/2018).

[12] CMS Collaboration. *About CMS*. URL: https://cms.cern/detector (visited on 27/11/2017).

[13] The ALICE collaboration. *ALICE Data Acquisition - DATE*. URL: https://alice-daq.web.cern.ch/products/date (visited on 24/02/2018).

[14] The ALICE collaboration. *The Evolution of the ALICE Detector Control System*. URL: http://cds.cern.ch/record/2213517?ln=en (visited on 24/02/2018).

[15] The ATLAS TDAQ Collaboration. *The ATLAS Data Acquisition and High Level Trigger system*. URL: http://iopscience.iop.org/article/10.1088/1748-0221/11/06/P06008 (visited on 24/02/2018).

[16] The LHCb collaboration. *LHCb Experiment Control System*. URL: https://lhcb-online.web.cern.ch/lhcb-online/ecs/default.htm (visited on 24/02/2018).

[17] JBoss Community and redhat. *RESTEasy*. URL: http://resteasy.jboss.org/ (visited on 09/02/2018).

[18] The Qt Company. *Qt*. URL: https://www1.qt.io/developers/ (visited on 24/02/2018).

[19] Go contributors and Google. *The Go Programming Language*. URL: https://golang.org/ (visited on 24/02/2018).

[20] Oracle Corporation. *Jersey*. URL: https://jersey.github.io/ (visited on 08/02/2018).

[21] Oracle Corporation. *Jersey - Bean Validation Support*. URL: https://jersey.github.io/documentation/latest/bean-validation.html (visited on 09/02/2018).

[22] Oracle Corporation. *Jersey - Custom Injection and Lifecycle Management*. URL: https://jersey.github.io/documentation/latest/ioc.html (visited on 09/02/2018).

[23] Oracle Corporation. *Jersey User Guide*. URL: https://jersey.github.io/documentation/latest/index.html (visited on 08/02/2018).

[24] CERN IT Department. *LXPLUS Service*. URL: http://information-technology.web.cern.ch/services/lxplus-service (visited on 12/12/2017).

[25] CERN IT Department. *SVN Service*. URL: http://information-technology.web.cern.ch/services/svn-service (visited on 20/12/2017).

[26] AngularJS developers. *Git Commit Guidelines*. URL: https://github.com/angular/angular.js/blob/master/DEVELOPERS.md#commits (visited on 20/12/2017).

[27] RCMS Developers. *RCMS Trac*. URL: https://svnweb.cern.ch/trac/rcms (visited on 20/12/2017).

[28] docs.oracle.com. *javax.xml.bind.annotation*. URL: https://docs.oracle.com/javase/8/docs/api/javax/xml/bind/annotation/package-frame.html (visited on 06/02/2018).

[29] EsperTech. *Esper*. URL: http://www.espertech.com/esper/ (visited on 24/02/2018).

[30] FasterXML. *Jackson Annotations*. URL: https://github.com/FasterXML/jackson-annotations/wiki/Jackson-Annotations (visited on 09/02/2018).

[31] FasterXML. *Jackson Core*. URL: https://github.com/FasterXML/jackson-core (visited on 08/12/2017).

[32] FasterXML. *Polymorphic Type Handling*. URL: https://github.com/FasterXML/jackson-docs/wiki/JacksonPolymorphicDeserialization (visited on 26/02/2018).

[33] Django Software Foundation and individual contributors. *django*. URL: https://www.djangoproject.com/ (visited on 24/02/2018).

[34] The Apache Software Foundation. *Apache Axis2*. URL: http://axis.apache.org/axis2/java/core/index.html (visited on 30/01/2018).

[35] The Apache Software Foundation. *Apache Axis2 - Axis2 Installation Guide*. URL: https://axis.apache.org/axis2/java/core/docs/installationguide.html (visited on 30/01/2018).

[36] The Apache Software Foundation. *Apache Axis2 - JAX-WS Guide*. URL: http://axis.apache.org/axis2/java/core/docs/jaxws-guide.html (visited on 30/01/2018).

[37] The Apache Software Foundation. *Apache Axis2 - JSON Support in Axis2*. URL: https://axis.apache.org/axis2/java/core/docs/json_support.html (visited on 30/01/2018).

[38] The Apache Software Foundation. *Apache Axis2 - RESTful Web Services Support*. URL: https://axis.apache.org/axis2/java/core/docs/rest-ws.html (visited on 30/01/2018).

[39] The Apache Software Foundation. *Apache Axis2 User's Guide - Building Services*. URL: http://axis.apache.org/axis2/java/core/docs/userguide-buildingservices.html (visited on 30/01/2018).

[40] The Apache Software Foundation. *Apache CXF$^{TM}$: An Open-Source Services Framework*. URL: http://cxf.apache.org/ (visited on 02/02/2018).

[41] The Apache Software Foundation. *Apache CXF Software Architecture Guide*. URL: http://cxf.apache.org/docs/cxf-architecture.html (visited on 02/02/2018).

[42] The Apache Software Foundation. *Axis2 Configuration Guide*. URL: http://axis.apache.org/axis2/java/core/docs/axis2config.html (visited on 30/01/2018).

[43] The Apache Software Foundation. *Axis2 Configuration Guide*. URL: http://axis.apache.org/axis2/java/core/docs/axis2config.html#Service_Configuration (visited on 31/01/2018).

[44] The Apache Software Foundation. *Axis2 Reference Guide*. URL: http://axis.apache.org/axis2/java/core/docs/reference.html (visited on 31/01/2018).

[45] The Apache Software Foundation. *CXF User's Guide*. URL: ttp://cxf.apache.org/docs/index.html (visited on 02/02/2018).

[46] The Apache Software Foundation. *Migrating from Apache Axis 1.x to Axis2*. URL: http://axis.apache.org/axis2/java/core/docs/migration.html (visited on 30/01/2018).

[47] The Apache Software Foundation and The Axis Development Team. *WebServices - Axis*. URL: https://axis.apache.org/axis/ (visited on 28/11/2017).

[48] Prof. Dr. Thomas Fuchß. *Hochschule Karlsruhe: Software-Engineering*.

[49] Google. *Google Java Style Guide*. URL: https://google.github.io/styleguide/javaguide.html (visited on 20/12/2017).

[50] Google. *Protocol Buffers*. URL: https://developers.google.com/protocol-buffers/ (visited on 25/02/2018).

[51] Inc. Google and Cloud Native Computing Foundation. *gRPC*. URL: https://grpc.io/ (visited on 25/02/2018).

[52] CMS TriDAS Group. *CMS Run Control and Monitoring System - "Mission Statement"*. URL: http://cmsdoc.cern.ch/cms/TRIDAS/RCMS/ (visited on 27/11/2017).

[53] CMS TriDAS Group. *RCMS Download page*. URL: http://cmsdoc.cern.ch/cms/TRIDAS/RCMS/Downloads/downloadsIndex.html (visited on 29/11/2017).

[54] Michele Gulmini. *CMS and Agata Run Control*. URL: https://agenda.infn.it/getFile.py/access?contribId=14&resId=1&materialId=slides&confId=913 (visited on 28/11/2017).

[55] Michele Gulmini. *Run Control and Monitor System for the CMS Experiment*. URL: http://cmsdoc.cern.ch/cms/TRIDAS/RCMS/Docs/RCMSPapers/proceedings/2003_CHEP_proceedings_Bellato_RCMS_0306110.pdf (visited on 28/11/2017).

[56] James Heumann. *Tips for writing good use cases.* URL: ftp://ftp.software.ibm.com/software/rational/web/whitepapers/RAW14023-USEN-00.pdf (visited on 29/01/2018).

[57] Gradle Inc. *Gradle Build Tool*. URL: https://gradle.org/ (visited on 26/02/2018).

[58] Open API Initiative and The Linux Foundation. *Open API Initiative*. URL: https://www.openapis.org/ (visited on 08/02/2018).

[59] *ISOTDAQ 2018 - International School of Trigger and Data AcQuisition*. URL: https://indico.cern.ch/event/643308/ (visited on 24/02/2018).

[60] *Java™ API for RESTful Web Services (JAX-RS)*. URL: https://github.com/jax-rs (visited on 08/02/2018).

[61] Nicolas Lurkin, Cristina Lazzeroni and Evgueni Goudzovski. *Neutral Pion Transition Form Factor Measurement and Run Control at the NA62 experiment*. URL: http://cds.cern.ch/record/2280295?ln=en (visited on 24/02/2018).

[62] XDAQ Maintainers. *cmsos - XDAQ CMS Online Software*. URL: https://svnweb.cern.ch/trac/cmsos (visited on 27/11/2017).

[63] Lawrence Mandel. *Describe REST Web services with WSDL 2.0*. URL: https://www.ibm.com/developerworks/webservices/library/ws-restwsdl/ (visited on 09/02/2018).

[64] Bodlák Martin et al. *New data acqusition system for the COMPASS experiment*. URL: https://indico.cern.ch/event/170595/contributions/266303/attachments/212115/297315/Poster_OXFORD.pdf (visited on 24/02/2018).

[65] Elettra Software for Measurements Group. *GRIDCC Project Home Page*. URL: https://web.archive.org/web/20071225210247/http://www.gridcc.org:80/ (visited on 28/11/2017).

[66] Esma Anais Mobs and CERN. *The CERN accelerator complex*. URL: http://cds.cern.ch/record/2225847/ (visited on 24/02/2018).

[67] Cristina Lazzeroni on behalf of the NA62 collaboration and Nicolas Lurkin. *The Run Control system of the NA62 experiment at CERN SPS*. URL: http://inspirehep.net/record/1596792/files/PoS(ICHEP2016)911.pdf (visited on 24/02/2018).

[68] Inc Object Management Group®. *Corba*. URL: http://www.corba.org/ (visited on 24/02/2018).

[69] Oracle. *HK2 - Dependency Injection Kernel*. URL: https://javaee.github.io/hk2/ (visited on 09/02/2018).

[70] Luciano Orsini. *JobControl*. URL: https://twiki.cern.ch/twiki/bin/view/XdaqWiki/JobControl (visited on 30/11/2017).

[71] Visual Paradigm. *How to Write Effective Use Cases?* URL: https://www.visual-paradigm.com/tutorials/writingeffectiveusecase.jsp (visited on 29/01/2018).

[72] Pivotal. *RabbitMQ*. URL: https://www.rabbitmq.com/ (visited on 08/12/2017).

[73] Checkstyle project. *Checkstyle*. URL: http://checkstyle.sourceforge.net/ (visited on 20/12/2017).

[74] Hannes Sakulin. *Dynamic configuration of the CMS Data Acquisition cluster*. URL: http://cmsdoc.cern.ch/cms/TRIDAS/RCMS/Docs/RCMSPapers/proceedings/2009_CHEP8-2_submission_revised.pdf (visited on 16/01/2018).

[75] Hannes Sakulin. *First Operational Experience With a High-Energy Physics Run Control System Based on Web Technologies*. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6218207 (visited on 28/11/2017).

[76] Hannes Sakulin. *RCMS Software evolution*. URL: https://indico.cern.ch/event/678570/contributions/2779363/attachments/1555578/2446163/2017_11_09_TopicalSWRoadMap.pdf (visited on 28/11/2017).

[77] Hannes Sakulin. *RCMS Status and Plans*. URL: https://indico.cern.ch/event/325954/contribution/12/material/slides/0.pdf (visited on 28/11/2017).

[78] Christoph Schwick. *DuckCAD documentation*. URL: http://cmsdoc.cern.ch/cms/TRIDAS/DuckCAD/documentation/html/index.html (visited on 29/11/2017).

[79] Siemens. *SIMATIC WinCC Open Architecture*. URL: http://w3.siemens.com/mcms/human-machine-interface/en/visualization-software/simatic-wincc-open-architecture/pages/default.aspx (visited on 24/02/2018).

[80] Siemens. *SIMATIC WinCC Open Architecture - Options for Web functions*. URL: http://w3.siemens.com/mcms/human-machine-interface/en/visualization-software/simatic-wincc-open-architecture/wincc-oa-options/internet-functions/Pages/Default.aspx (visited on 24/02/2018).

[81] Pivotal Software. *Spring Boot*. URL: https://projects.spring.io/spring-boot/ (visited on 26/02/2018).

[82] Pivotal Software. *Spring Framework*. URL: https://spring.io/ (visited on 24/02/2018).

[83] Pivotal Software. *Spring Web*. URL: https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#spring-web (visited on 24/02/2018).

[84] Pivotal Software. *Spring Web Services*. URL: http://projects.spring.io/spring-ws/ (visited on 08/02/2018).

[85] SmartBear Software. *Swagger*. URL: https://swagger.io/ (visited on 08/02/2018).

[86] STOMP Specification. *STOMP - Simple Text Oriented Messaging Protocol*. URL: https://stomp.github.io/ (visited on 16/01/2018).

[87] Inc. Sun Microsystems. *Design Note on page 60 of The Java (TM) Architecture for XML Binding (JAXB) 2.0*. URL: http://download.oracle.com/otn-pub/jcp/jaxb-2.0-fr-eval-oth-JSpec/jaxb-2_0-fr-spec.pdf?AuthParam=1517844283_65844a232a9fc476b75a82f54df48aa2 (visited on 06/02/2018).

[88] CMS Sysadmins. *[Restricted Access] Cluster Users Guide*. URL: https://twiki.cern.ch/twiki/bin/viewauth/CMS/ClusterUsersGuide (visited on 12/12/2017).

[89] Lucas Taylor. *Triggering and Data Acquisition*. URL: http://cms.web.cern.ch/news/triggering-and-data-acquisition (visited on 27/11/2017).