OFFENBURG UNIVERSITY OF APPLIED SCIENCES

DEPARTMENT OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

PRACTICAL REPORT

K-Modulation Software Development and Automation

Author:
Martin Lukas
SPITZNAGEL

Supervisor:
Dr. Rogelio Tomás
GARCIA

CERN-THESIS-2019-013
11/03/2019

European Organization for Nuclear Research (CERN)
Optics Measurements and Corrections(OMC), Beams Department (BE)

February 22, 2019

Declaration of Authorship

I, Martin Lukas SPITZNAGEL, declare that this thesis titled, K-Modulation Software Development and Automation and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the report is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

OFFENBURG UNIVERSITY OF APPLIED SCIENCES

Abstract

Department of Electrical Engineering and Information Technology

Optics Measurements and Corrections(OMC), Beams Department (BE)

K-Modulation Software Development and Automation

by Martin Lukas SPITZNAGEL

The *k*-modulation turned out to be the fastest and most precise tool to get a precise determination of the beam size directly at interaction points (IP) of the Large Hadron Collider (LHC). These measurements can be used to increase the collision rate for each experiment directly at its detectors around the LHC ring. The *k*-modulation functionality was already implemented by the Optics Measurement and Correction (OMC) team. The tool with its corresponding Graphical User Interface (GUI) was used with success during the past years of operation of the LHC. This report will introduce a new extension to the existing *k*-software making it feasible to start a variable amount of *k*-modulation processes at once and optimizes the usability significantly with new developed GUI components. This report functions as a documentation for the new *k*-modulation module which will be used for measurements in the CERN Control Center (CCC) during Run III of the LHC.

Contents

List of Figures

List of Abbreviations

OMC	**O**ptics, **M**easurement and **C**orrection
LHC	**L**arge **H**adron **C**ollider
CCC	**C**ern **C**ontrol **C**enter
GUI	**G**raphical **U**ser **I**nterface
IP	**I**nteraction **P**oint
CALS	**C**ERN **A**ccelerator **L**ogging **S**ervice
LSA	**L**HC **S**oftware **A**rchitecture
API	**A**pplication **P**rogramming **I**nterface
CMS	**C**ompact **M**uon **S**olenoid
ALICE	**A** **L**arge **I**on **E**xperiment
ATLAS	**A** **T**oroidal **L**HC **A**pparatu**S**
LHCb	**L**arge **H**adron **C**ollider **b**eauty

Chapter 1

Introduction

The Large Hadron Collider (LHC) is CERN's flagship accelerator, colliding hadrons with an energy of 7 TeV per proton beam. The particle beams collide at four specific interaction points of experiments positioned around the LHC ring. At these points the experiments ATLAS [1], CMS [2], ALICE [3] and LHCb [4] are located. Each experiment has to be provided with a maximum amount of particle collisions between both circulating beams. To increase this value the optics have to be adjusted to be able to control the beam size and its delivered luminosity at interaction points of the experiments. After optimal settings are found, they need to be validated with a decisive method. In the past years, the *k*-modulation turned out to be the fastest and most precise method to get a precise determination of the β-function and its waist shift directly at interaction points [5].

The Optics Measurement and Correction (OMC) software tools are providing the functionality of the *k*-modulation in its corresponding Graphical User Interface (GUI). The k-modulation GUI provides functionality for the following tasks: sending instructions and receiving data from the LHC, calling python scripts for data analysis and presenting the results of the analysis graphically. Since the GUI is mostly used in the CERN Control Center (CCC) it has to use the same environment for graphical user interfaces, which is based on Java. In the past the *k*-modulation software was improved several times in order to enable the direct measurement of β^* [6]. This leads to a split into several input masks and options which decreases its usability. In addition, the module was designed for only one modulation and has to be restarted for each analysis, making it laborious to use in cases where multiple data-sets need to be evaluated, for example when re-analyzing past measurements.

The functionality of the GUI as now been improved by implementing a new module coordinating the execution of multiple modulations at once. Besides

that, the module has to handle the execution, error handling and optimization for each modulation and the corresponding pipeline in order to increase the usability and hence significantly decrease the time needed for the analysis. This can be achieved by removing unnecessary user interactions and gathering input and environment data directly. In order to increase its usability even further, the user is now able to see the status online as well as read the results directly from the GUI which was not possible in the old versions. Documentation and code quality was a big target as well, since the existing *k*-modulation software has not been documented and code was hard to understand for developers. This was prone to lead to avoidable faults during tests or measurements in the CCC.

This report is divided into 7 chapters: Chapter 2 gives a very general overview of the CERN accelerator complex and the basics of accelerator physics in order to understand the importance and the way of functioning of the *k*-modulation in modern circular colliders like the LHC. Chapter 3 summarizes the existing software tools from the OMC team and explains the usage of some CERN-provided interfaces. Chapter 4 and 5 are outlining the work carried out in the project. This is split into the design phase in Chapter 4 and its implementation in Chapter 5. After that, additionally implemented features are listed in Chapter 6, being an addition to the main module and improving the other existing tools in the *k*-modulation software as well. Chapter 7 will summarize the work done and gives an outlook for the coming challenges in the field of software engineering for the OMC team. Other projects which are not part of the new module are mentioned in Appendix A.

Chapter 2

Basic Accelerator Physics

The following chapter will give a short overview about the CERN accelerator complex and the basics of accelerator physic in order to understand the way of functioning and importance of the k-modulation for modern and powerful accelerators like CERN's Large Hadron Collider.

2.1 Large Hadron Collider

The Large Hadron Collider is CERN's biggest accelerator with a circumference of 27 km and a collision energy of proton beams up to 14TeV [7]. During operation, both circulating beams are injected in the LHC by using the accelerator chain shown in Figure 2.1. Each beam is pre-accelerated by several of the accelerators and sends step by step towards the LHC with increasing energy. Since the main goal is to provide a maximum number of collisions in a given amount of time it is important that the beam size is minimized at the interaction points of ATLAS [1], CMS [2], LHCb [4], and ALICE [3]. The LHC is able to provide up to 1 billion collisions per second for its experiments [8].

The mentioned experiments are located around the LHC ring as indicated in Figure 2.1. Each experiment has its own field of scientific research. The main focus of ATLAS and CMS is to test the standard model of particle physics and look for physics beyond the standard model. Based on collisions in these two experiments, a great milestone in particular physics was achieved in 2012 proving the existence of the Higgs Boson [10]. The LHCb studies investigate the properties of anti-matter and ALICE is specialized in heavy-ion collisions, to detect the creation of quark-gluon plasma [8].

Since the beam settings can differ between some modes of operation, the LHC optics have to be optimized for each beam configuration. This is one

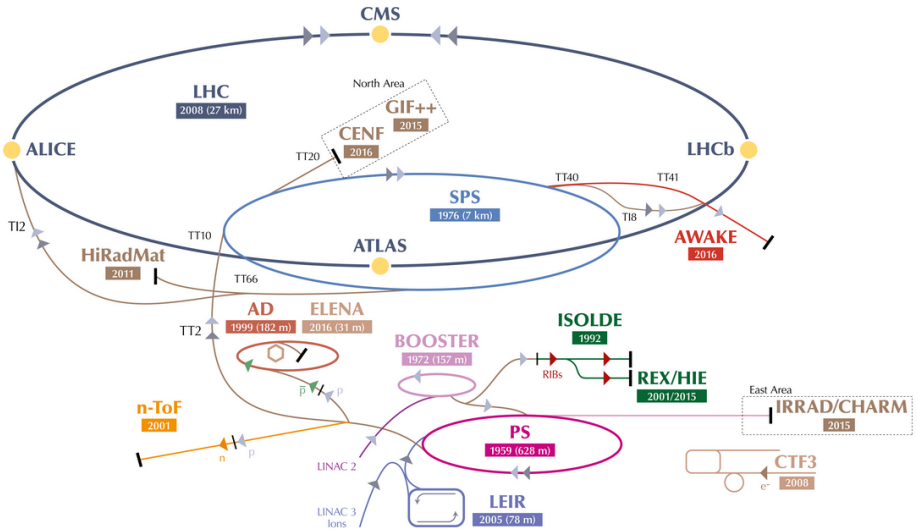

FIGURE 2.1: CERN accelerator complex [9].

major task of the Optics Measurement and Correction team. On one hand, having multiple modes extends the physical range that can be studied by the several experiments of the LHC but it also increases the complexity of the whole accelerator infrastructure which has to be handled.

2.2 Accelerator Parameter

2.2.1 Luminosity

In order to provide the experiments with a maximum amount of data the collision rate has to be as high as possible. This rate between two colliding beams is determined by the luminosity and can be calculated by using the following formula:

$$\mathcal{L} = \frac{N_1 N_2 f n_b}{4\pi\sigma_x\sigma_y} \tag{2.1}$$

During operation the luminosity $\mathcal{L}$ can be only adjusted by the beam sizes $\sigma_{x,y}$ since N_1, N_2 and n_b are set during the injection and f is fixed by the circumference of the LHC [11]. The beam sizes σ_x and σ_y have to be minimized in order to maximize the delivered luminosity.

2.2.2 β-Function

The transverse beam size in a circular collider is determined by two quantities according to:

$$\sigma_{x,y} = \sqrt{\beta(s)_{x,y}\epsilon_{x,y}} \tag{2.2}$$

The emittance, ϵ, stays mostly the same in LHC after injection, since radiation effects for hadrons at the given energies are negligible [11]. The machine optics and their variations are represented with the β-function and can be used in order to adjust the beam size at any point of the LHC.

The main parameter of interest is β^*, representing the beams β-function at the experiments interaction point where the collision of the LHC bunches are detected and analyzed. Decreasing beta* will therefore lead to an increased luminosity, which in turn leads to a higher rate of collisions, providing more collision data for the experiments.

2.2.3 Tunes

Because of spread in location and momentum, particles in the beam are not exactly following the design orbit. Instead, each particle is oscillating around it. The number of these so-called betatron oscillations for one turn in the accelerator in units of 2π is the tune Q [12]. Since the motion of the beam can be different in the horizontal and vertical plane the tunes are represented as Q_x and Q_y.

$$Q_{x,y} = \frac{1}{2\pi} \oint \frac{ds}{\beta_{x,y}(s)} \tag{2.3}$$

Imperfections of the magnets can lead to beam instabilities, since the particle passes through the same magnet each turn. In this case, the oscillating amplitude could get bigger and bigger each turn increasing the possibility of particle losses. In order the avoid losing too many particles the resonance conditions, which depends on the tune, have to be avoided by choosing a working point far away from resonance lines [12].

2.3 k-Modulation

In order to maximize luminosity and the corresponding amount of collisions per second, it is necessary to control the β-function close to an interaction point. Furthermore, it is important to avoid a significant imbalance of luminosity between experiments like ATLAS and CMS. The LHC requires the full commissioning of optics and collimation before any luminosity may be delivered to the experiments [13]. With the other existing methods, this has to be done in many steps exceeding the anticipated commissioning time window. In the past, k-modulation was successfully used to measure the β-function at interaction point and turned out to be a faster and more accurate alternative method for modern circular collider.

The k-modulation is the method of changing the strength of individually powered quadrupoles in order to calculate the β-function. If the modulation is small enough the beam orbit will stay the same and only the tune is changed [14]. Measuring this tune change based on modulating the strength of an individually powered quadrupole can be used to calculate the average beta function in the quadrupole with the following formula

$$\beta_{AV} = \frac{2}{l\Delta k}\left[cot(2\pi Q_{x,y}) - \frac{cos(2\pi(Q_{x,y} + \Delta Q_{x,y}))}{sin(2\pi Q_{x,y}}\right] \tag{2.4}$$

where Δk is the quadrupole strength change, l is the length of the quadrupole, ΔQ the tune change and Q the nominal tune [10]. This can be simplified for small tune changes far away from resonances to

$$\beta_{AV} \approx \frac{4\pi}{l}\frac{\Delta Q_{x,y}}{\Delta k}. \tag{2.5}$$

The ratio between two β_{AV} values can now be used to calculate the β^* value at a specific interaction point as explained in [13].

Figure 2.2 is a sketch of an interaction region, showing the left and right course of the beta function next to the interaction point. The precision of this measurement depends mostly on the accuracy of the tune measurement method.

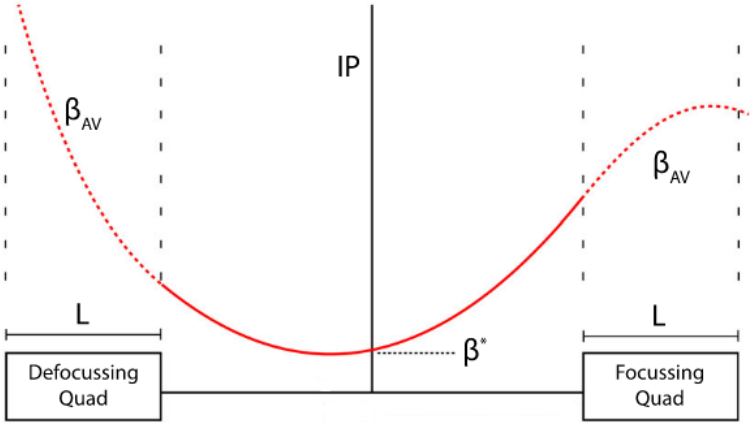

FIGURE 2.2: Schematic representation of interaction region configuration with a focusing quadrupole on the right and a defocusing quadrupole on the left [13].

Chapter 3

OMC Software Overview

3.1 Introduction

The Optics Measurement and Corrections Team has a large package of software with many different analysis tools and GUIs written in Java, Python and C/FORTRAN77 [15]. The analysis data is mostly gathered during dedicated measurement sessions at the CCC and analyzed using several applications. Goal of these measurements is to optimize the LHC optics to maximize the performance and help to understand physical accelerator conditions of the LHC. This section gives an overview of the existing OMC Software tools with the main focus on the k-modulation application.

3.2 BetaBeat GUI

The most important and powerful GUI of the OMC team is the BetaBeat GUI. This GUI combines the functionality to call external python scripts with a graphical user interface for results, charts and several input parameters [15]. It is mostly used to analyze beam data gathered during measurements in the CCC and to apply the optimized optics settings to the LHC. In order to apply corrections to optics, the provided LHC Software Architecture (LSA) API is used [15]. Therefore the BetaBeat GUI provides the functionality to generate correction knobs based on measured data for different optics. This can be used for changing the β-function at interaction points and can be verified using the *k*-modulation software tools.

3.3 *k*-Modulation Software

The *k*-modulation software is an automated k-modulation tool that is able to modulate the currents of a given quadrupole with an adequate frequency and amplitude [10]. It is a from the BetaBeat-GUI independent application written in Java, which can be run in the CCC. The user can either choose to modulate the two magnets next to an interaction point or select each magnet on its own. The first option is the most used, since measuring the β^* at the IP of experiments is most important in order to adjust the optics to deliver as much luminosity as possible. This standard module is called "Full IP Trim" and starts the modulation with displaying the LHC beam and magnet data. The k-modulation software also includes some more functionality which is not of interest for this report since the new module only extends and optimizes the "Full IP Trim" and the following analysis.

3.3.1 Full IP Trim

The Full IP Trim module executes the k-modulation on a selected interaction point with its predefined quadrupoles. Each bigger experiment like ATLAS (IP1), ALICE (IP2), CMS (IP5) and LHCb (IP8) is numbered by its corresponding number in the LHC complex and can be selected from the user as seen in Figure 3.1. By using this module the quadrupole identifier and several other parameters are predefined to the corresponding interaction points. After the modulation is done the trim data is present on the database and can be visualized and analyzed using the measurement view dialog. In order to start the *k*-modulation on a magnet the power converter has to be accessed using the CERN provided LSA Java API.

FIGURE 3.1: Interaction point selection dialog in order to start the *k*-modulation trim.

3.3.2 Measurement View Dialog

The user can extract trim data from the database and analyze it in the measurement view dialog with several analysis tools. This data extraction can be done by using the LHC logging database mentioned in 3.5. In order to do that, the user can choose between several options to extract trim data for each magnet as shown in Figure 3.2. The user can either extract the trim with start and end timestamps of the left and right magnet or with only an start timestamp that can be selected in an external selection panel. The second method was developed more recently and is most commonly used since it simplifies the extraction of trim data for two quadrupoles and increases the usability for accessing the database.

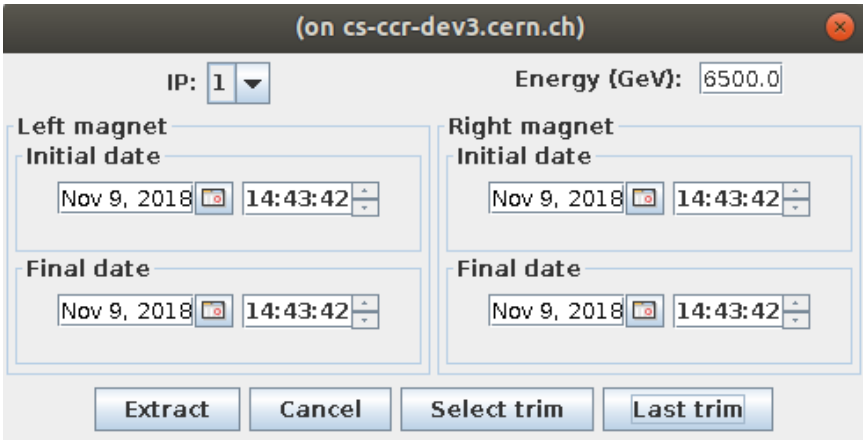

FIGURE 3.2: Main dialog to extract a trim from the logging database. The dialog also includes the functionality to extract the data for specific magnets for the left and right side which is not used at the moment.

Each step of the analysis has to be done manually with several user interactions. As seen in Figure 3.3 the trim data is displayed as a chart separated into 4 pieces displaying the current for the left and right quadrupole and tune data for beam 1 and 2 of the LHC. After an analysis is finished the results have to be manually extracted from a saved file in the selected working directory.

Since most of the time the main workflow stays the same, the existing GUI has many unnecessary user interactions. Each step has to be executed separately, making it laborious to use and decreasing its usability. This has to be optimized by implementing a fully automated (k-modulation to β^* module). In the current state, the whole software is mostly undocumented without any

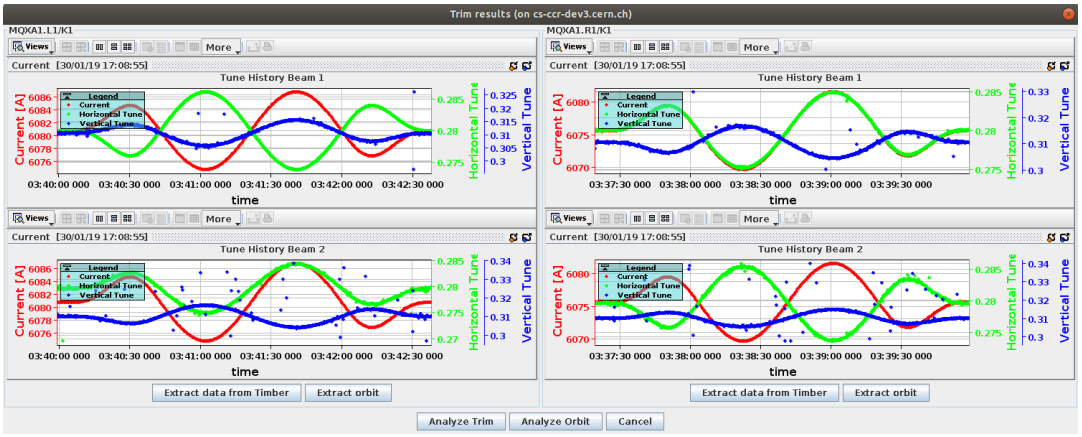

FIGURE 3.3: The standard measurement view dialog showing a the trim data for interaction point 2. The current is displayed in red, horizontal tune green and vertical tune blue. There are several buttons to start the data analysis.

comments or Javadoc attached, making it hard to understand the functionality of the many CERN API's that are used. The new module has to be fully documented with an attached "how to use"-wiki.

3.4 External Programs

The external programs are the core of the data analysis. Each external python program, which can be executed from the GUIs, is available from the BetaBeat.src GitHub repository [16]. The GUI can locate the executables by accessing a dictionary located in the repository including each program name with its corresponding path. This dictionary is implemented as a *.properties-file, making it easy to access.

All the classes for calling an external python script are located in the same package. Each class in the program's package represents a python script. The class has to implement the ExternalProgram interface which provides the basic methods to run a script. Since these scripts are executed from the command line the corresponding command has to be assembled as a string with its needed arguments.

In order to add a new python script as external program to a Java GUI the following methods have to be implemented:

```
/* arguments as list */
public List<String> getArguments();

/* application name as set in *.properties */
public String getApplicationName();

/* To be called before the execution. Should return state if
    everything is ok or not */
public boolean runBeforeExecute();

/* To be called after the execution. Should return state if everything
    is ok or not */
public boolean runAfterExecute();
```

The script can then be executed by instantiating this new class and using the provided ExternalPrograms class. This handler will extract the path to the corresponding python script by accessing the ProgramVersions.properties file in the BetaBeat.src directory with the returned application name of the implemented getApplicationName() method as a key. The whole command line command is then assembled with adding the path to the python executable and the script arguments, set in the getArguments() method. The handler waits until the script is fully finished and checks before and after execution if everything went as expected.

This can be done for any executable python script making it feasible to add new scripts in a short amount of time with enough precautions to avoid critical errors during runtime.

3.5 CERN Accelerator Logging Service

Before the start of the first LHC run, a new long-term data logging service was developed [17]. Its main intention is to provide easy time-series data access for comparison of data between years. The CERN Accelerator Logging Service (CALS) stores more than 1 million signals coming from several sources including a wide range of data like currents, beam positions, losses, etc. The logging service provides access to its data for many custom applications from around CERN and is used in all OMC software tools in order to provide data for analysis.

Currently, the data is persisted in Oracle RAC databases and can be accessed using a Java API or an optional GUI called TIMBER, in which the data can be also visualized. The entire Java infrastructure is based on the Spring framework and pure JDBC for database interactions. In order to filter and transfer data from the short-term to long-term database, PL/SQL is used [17].

The interaction between GUI and database was already implemented by the OMC team. The database can be accessed by using the provided extraction tools. In the *k*-modulation software this is used to extract the needed data for its analysis of β^*. The data extraction can only be done for a specified time window with a fixed start and end timestamp. In order to gather the trim data for the right and left magnet, each start and end timestamp of the corresponding modulation is used as well as an unique identifier specified for this database entry set.

Chapter 4

Module Design

The new *k*-modulation module extension provides the functionality to execute multiple independent modulations after each other with optimizing the needed time and improving the usability for the physicists during difficult time-limited measurements. It combines several separated tasks in the old software to one executable unit. A pipeline provides the functionality to start the execution of multiple units at once. By adding optional tasks or changing input parameters the user has to be able to configure each executable unit individually. Since in the field of scientific research, the requirements change in a small period of time the design has to be as extendable as possible to be able to easily extend and adjust its functionality in case of future development. The new module will just be an extension of the old software until its fully tested during Run3 in 2021. The old functionality mentioned in Chapter 3 has to be available if needed and can be replaced if the tests are successful.

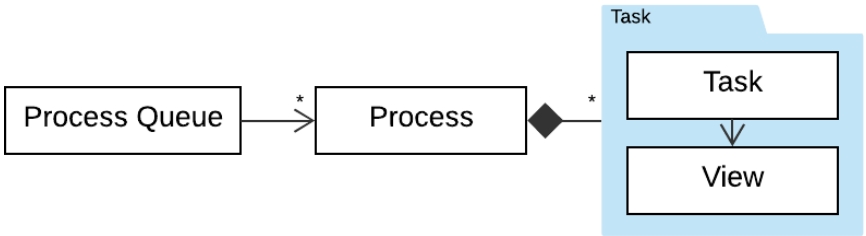

FIGURE 4.1: Simplified schematic module design

Figure 4.1 shows a schematic overview of the new module design. As mentioned before several tasks are combined to an executable unit which is represented by one *k*-modulation process. Each process consists of a variable amount of tasks holding the executable functionality and sending its results and status to the associated view. In order to provide a fixed execution

scheme all *k*-modulations processes are held in a process queue which reduces the execution time by optimizing the process order.

The following sections will give an in-depth view of how the module is designed starting with the process and task functionality.

4.1 *k*-Modulation Process

Starting with the key component, the *k*-modulation process is representing one whole measurement with its several tasks performed by physicists at the CCC. Most of the times this includes a *k*-modulation trim which is executed live on two quadrupoles next to an interaction point and the following analysis to receive the β^* and waist shift. These results need to be provided online since they can change the proceeding of the measurement. Even though this order of tasks is mostly used, the process object has to be designed as extendable and variable as possible to provide an open interface for new functionality.

The first step will always be the *k*-Modulation and the procurement of the LHC data followed by a variable amount of analysis tasks. Since this modulating task is affecting the magnets of the LHC, it is not possible to put all processes into one sorted list and execute them in a random order. The online modulation needs more precautions.

From a more code-based point of view, the *k*-modulation process is represented by an object holding public methods to run its tasks and only provides the basic execution structure to extract the data and start the followed analysis.

4.1.1 Tasks

Each operational step during a measurement is represented as a task. Depending on the type of task it has a fixed execution routine with variable input parameters and the functionality to send status messages and results to the user view.

To provide traceability for the user, a task has to display its current status as well as some general information like start and end time to the representing GUI component. These methods are provided in a general interface and have

to be implemented by the executed tasks. For the current requirements, it was necessary to implement an abstract class to represent a task with and without results to display. Both task types are getting executed from the parent *k*-modulation process using the scope of the main interface.

Currently, there are four main steps which have to be implemented as tasks:

- Executing a *k*-modulating trim on LHC
- Loading the data of an existing trim from the CERN logging database
- Analyzing the data to receive β^*-results
- Extracting and analyzing beam orbit during the *k*-modulation trim

It is then necessary to differentiate between an actual *k*-modulation task which provides the trim data and an analyzing task which needs the data to calculate the results for the user.

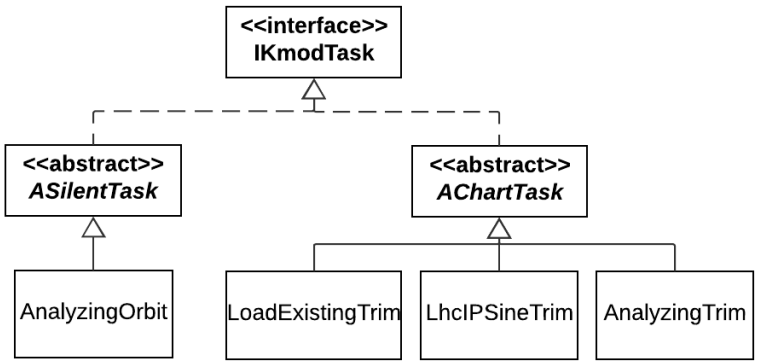

FIGURE 4.2: Class diagram for each task giving an overview of the class hierarchy.

As seen in Figure 4.2 each task has to implement the main interface with the basic methods like executing its functionality and the connection between task and view. These basic methods are mostly implemented in the inherited abstract classes providing a partially implemented interface for new tasks. The current design differs between tasks showing results or not.

Each *k*-modulation object can hold an unlimited amount of tasks, starting with one modulation task which provides that data for the following analysis. With this extension, it is possible to execute many k-modulation processes in sequence. Since there are some race conditions between functionalities, it is necessary to execute a list of processes in an optimal order. This has to be handled from a separated class holding a list of *k*-modulation processes and organize their execution.

4.1.2 Process Queue

The process queue provides the functionality to execute multiple *k*-modulations and their analyzing tasks in a pipeline. This pipeline will optimize the needed time for multiple modulations by sorting the processes in the pipeline and using concurrency.

During a session, there is only one queue with a variable amount of processes. This queue will execute each process by a fixed order. Since only modulating tasks cannot happen in parallel it is possible to execute the analyzing tasks of one process in another thread while the queue proceeds with the trimming task of the next process. This optimization can only be accomplished if the analyzing tasks are synchronized.

The order of execution also takes a significant role when it comes to runtime optimization. The time needed for a process is defined by the type and amount of analyzing tasks it is holding. Because the analyzing tasks are executed in a separated thread it is worth to change the order of processes in the pipeline depending on their runtime.

Figure 4.3 shows the functionality of the whole process. After all input panels are created by the user the main dialog iterates through each panel and creates one process per panel depending on its input and chosen options. Each process is added to the process queue. After all k-modulation processes are created the queue will sort them and execute the trim task of the first process. This task is executed using the scope of the main task interface. After the trim is finished the results are shown in the associated result panel. Now a new thread is getting started by the queue (symbolized with dashed lines). The whole operation is repeated for every modulation process. The new thread, called analyzing thread, is now executing each analyzing task sequentially. If there are results to display the task will send them to the corresponding result panel in the GUI. This is done until every trimming task is fully executed and all analysis threads are finished. After that, the pipeline can be started again by the user.

4.2 Graphical User Interface

One of the main tasks of this extension is to improve the existing graphical user interface and its usability. The old *k*-modulation software is laborious to

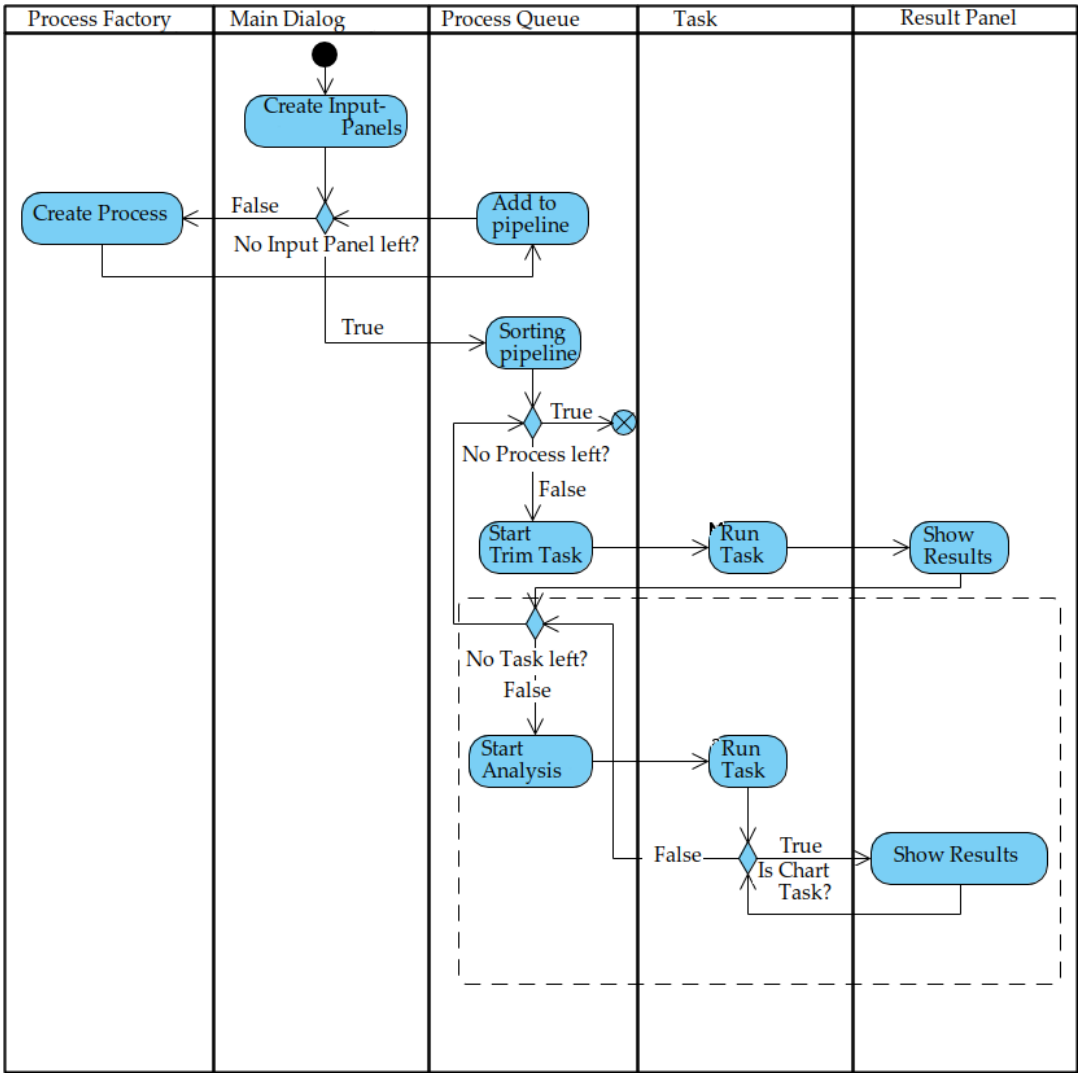

FIGURE 4.3: Execution scheme activity diagram

use and has too many user interactions which are slowing down the whole process.

The used GUI widget toolkit is Java Swing since the existing software is programmed in Java and the interface style should look similar to the other GUI's that are currently used in the OMC team. This toolkit provides all the needed widgets and panels to provide a clear and responsive design.

The GUI is divided into several views. Each view has its own purpose either gathering input parameters from the user or keeping the user up to the date of the current progress.

4.2.1 Input View

Even though the GUI has to handle a large amount of input parameter the interface should be easy to understand. It provides options for adding new *k*-modulation processes for each existing interaction point with options to load an existing trim from the CERN database. Since it is unnecessary to call an analyzing task twice the user can add them via tick boxes. Each task has its own input panel with the corresponding input fields and options. The user is able to switch between the existing *k*-modulation processes and delete them if needed. In addition to that, it is possible to start the whole process from the input view. To improve usability and save time during measurements all input values are preset with a standard or extracted value that most of the inputs are setup automatically.

The GUI provides an option for selecting multiple trim entries and load them from the database. This is done in a separated dialog with the possibility to set some general options like default input values for the selected trims.

4.2.2 Result View

Since this new module executes many processes and tasks at once it is very important to keep the user up to date to the ongoing process. Each process has its own panel for displaying the current status and its results for each task. The GUI includes an overview panel to see the currently executed task of each process in one view.

Status Panel

The status panel provides the user with all the information about the k-modulation process and its tasks. It should be possible to switch between the tasks and see its start/end time, input parameter, a logging console, and the current status. The main label of the status panel should always display the name of the currently running task.

Result Panel

Each task displaying results is adding a panel to the result panel of the corresponding *k*-modulation process. Since the results of the tasks can be very

different it is necessary to implement a result panel class for each task to provide a correct visualization of each resulting data.

4.3 Synchronization

A further problem to note is the synchronization between the parallel execution of several *k*-modulation processes. It is important to guarantee that only one modulating task is executed at once to ensure the quality of the measurement. This can be archived by using a synchronization variable like a semaphore or a fixed execution scheme. In this case, there are several negative aspects of using a semaphore for synchronization. By using a synchronization variable every process is started as an own thread making it harder to provide an optimal execution order. Since the execution scheme is always the same and only the modulating operation has to be blocked, this is accomplished much easier by executing each modulating task in a pipeline.

The view also needs some kind of synchronization since multiple tasks are running at the same time, displaying their results and status to the associated view. Since all analyzing tasks are executed in a separated thread it is important to care that only one task can change the value of a widget at a time. This could lead to wrong results or in the worst case to a crash of the whole application.

In addition to the internal synchronization, it is important to care about the external delay in the CERN logging database. The data is provided several seconds after the modulation is fully done. Loading the data too early would change the resulting values since some data would be missing. The new module has to ensure that the data is fully loaded and inform the user in case the application has to wait until the data is ready.

4.4 Error Handling / Machine Protection

With the new module, it is possible to execute many *k*-modulation processes so it is necessary to care about keeping the user up to date if something went wrong. Crashes could occur during each task step making it more important to inform about where the crash happened and why. The error dialog displays in which task of the corresponding process the crash happened to

provide information about which task was fully executed, crashed the process or is only scheduled in the pipeline. This avoids restarting an already fully executed task.

In the past, user oversight lead to some avoidable mistakes which caused losing precious time. Most of the time the reason for the faults was a wrong state of some LHC parameter like the orbit feedback. These are automatically checked and visualized for the user to avoid the loss of time in the future.

Chapter 5

Implementation

This chapter gives an overview of the implementation of the design from the previous chapter. It presents how each working unit and the graphical interface is implemented with some short code listing to give an insight view of its key functionality.

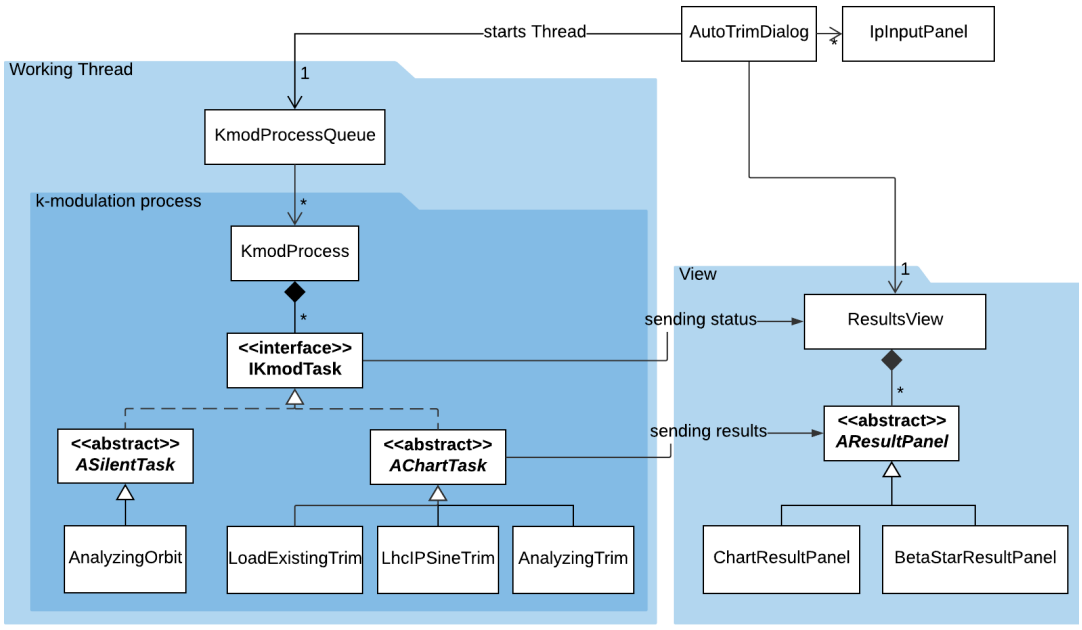

FIGURE 5.1: Schematic class infrastructure of the new automated k-modulation pipeline module

The class dependency diagram is shown in Figure 5.1. The following chapters explains the purpose of each class and how the components interfere with each other. Since the GUI elements are depending on the functionality of the working processes this chapter starts with working process implementation. After that, the GUI elements and the communication between graphical interface and working processes are explained.

The hierarchical order is the same as in Chapter 4. This time the main focus is to give a more code based view of the concrete implementation of each segment of the new module.

5.1 *k*-Modulation Process

Since the *k*-modulation process only has to provide the infrastructure to hold and execute one trimming task and multiple analyzing tasks the implementation is quite simple. It separates a single reference and a list of references to the provided task interface, being set with the corresponding setter methods. The process provides two methods to either start the modulating task or to run the analysis. The analysis contains a variable amount of analysis tasks which are getting executed one by one. Since all tasks are held in the scope of the basic task interface each task can be started with the same method, implemented in the inherited task classes. In order to inform the user what task is currently running in this process, the process thread bar in the main GUI is updated each time a task is started.

A main responsibility of the *k*-modulation process is to inform the user in case of an error during execution. The error handling implementation is explained in 5.9.

5.2 Modulating Tasks

A modulating task provides the process with trim data for the following analysis. Each process object has only one modulating task which can be executed by calling the startTrimProcess-method.

5.2.1 LHC Interaction Point Trim

The LHC interaction point trim class implements the abstract AChartTask as seen in Figure 5.1 since this task wants to display the trim result in a chart in the associated view. The new task has to implement the startTask-method which is getting called from the parent k-modulation process object as well as the printResult-method to print the chart to the result panel.

The k-modulation trim on the magnets next to the interaction point was already implemented by the OMC team. The task takes this existing functionality and adjusts it to the new environment.

k-Modulation

In order to modulate the current of a magnet near an interaction point, it is mandatory to arm the magnets power converter with the modulation settings provided as input in the k-modulation software. Since for β^* calculation it is necessary to modulate two magnets this is done in two separated steps starting with the right magnet modulation. This process is the same for both sides with the quadrupole name as the only difference.

Each quadrupole and its associated power converter can be in three different states as visualized in Figure 5.2. At the start of a k-modulation, the magnet should always be in the idle state representing the availability for instructions. After arming the power converter it remains in the armed state. During the armed state, the power converter is charged with a modulation action which can be started using the provided interfaces. Once this action is started the power converter is in the running state until the action is fully finished. After the modulation is done the power converter returns to the idle state.

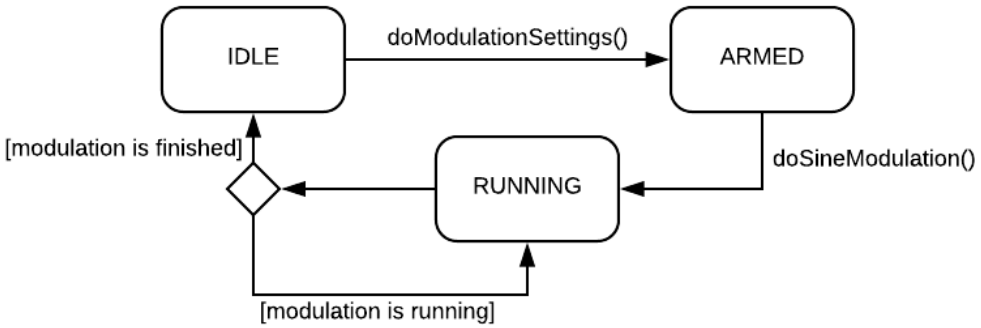

FIGURE 5.2: Simplified state machine for quadrupole with its associated power converter

The input parameters, as well as the power converter name and hardware handler, are represented as an own class, making it easier to access all the needed information for the modulation. If the magnet is in idle state the power converter can be armed by passing the modulation parameter with

the associated unique identifier. In the following listing, this is implemented for the amplitude using the existing API to send instructions to the power converter which is associated with the quadrupole.

```
1 TransactionalParameter paramAmpl =
      ParameterFactory.newInstance().newParameter(device, AMPLITUDE);
2 SimpleParameterValue valueAmpl =
      SimpleParameterValueFactory.newValue(trimAmplitude);
3 paramAmpl.setValue(null, valueAmpl);
```

This is done the same way for every modulation parameter. If everything went right the power converter is now armed and ready for executing its action. This can be checked by querying the state of the current communicator, using the hardware handler. In order to provide a strict procedure each step is protected with a state check:

```
1 if (("IDLE").equals(hardwareHandler.getCurrentCommunicatorState()) {
2     doModulationSettings(); //applying modulation settings
3     if
          (("ARMED").equals(hardwareHandler.getCurrentCommunicatorState())
          {
4         doSineModulation(); //starting modulation
5         ...
```

If the state is not as expected the modulation is canceled. In case the k-modulation software crashes while a power converter is armed the armed action has to be cleared manually in the CCC. The modulation can be started if the power converter is an armed state. This is done by firing an event using the already provided timing services of CERN. The following statement shows how this event is fired:

```
1 EventType eventType =
      TIMINGSERVICE.findEventTypeByName("HX.SRMP-POW-CT");
2 Event event = eventType.getEvent(PAYLOAD);
3 TIMINGSERVICE.sendEvent(event, 0);
```

The GUI will wait until the power converter is in the idle state again while the modulation data gets visualized. After the modulation is fully done the data can be loaded from the database in order to start the analysis.

Database Check

The logging database provides all data needed for the analysis but the application has to care about the synchronization and the database delay. Since the CERN logging database is timestamp based it is not possible to easily check if the data is ready or not. If the trim extraction would be executed directly after the modulation the database will just return the data in the given timestamps even though the last few seconds of data are missing. The interaction point trim task has to care about this critical fact and ensure that the whole data is present in order to avoid wrong results.

This is done by implementing a new jDialog class checking the trim data and keeping the user up to date on the current status and delay. The dialog is initiated by passing the quadrupole objects holding the hardware identifier and the end timestamp of both modulations of right and left side.

```
1 if(SimulationSwitch.equals(SimulationSwitch.MEASUREMENT))
2 {
3     TrimDataCheckDialog trimDataCheckDialog = new
          TrimDataCheckDialog(leftQuadrupole, rightQuadrupole, new
          Timestamp(endLeft.getTime()), new
          Timestamp(endRight.getTime()));
4
5     trimDataCheckDialog.checkTrimData();
6 }
```

To check if the database is already up to date the dialog compares the end timestamp of each magnet with the last entry on the logging database. If the difference is higher than MAX_OFFSET of one second it will notify the user and show him the offset in the corresponding GUI widget. This is done by using the existing database functionality of CALS and the easy comparison of two timestamps.

```
1 public static boolean trimDataReady(Quadrupole quadrupole, Timestamp
      endTimestamp) {
2     String currentIdentifier = quadrupole.getPowerConverterName() +
          CURRENT_SUFFIX; // Suffix for current value of magnets
3     TimeseriesData data =
          extractLastValueOfVariable(currentIdentifier);
4     Timestamp t = data.getStamp();
```

```
5     return (endTimestamp.getTime() - t.getTime()) <= MAX_OFFSET; //
          1000
6 }
```

The method will return true if the trim data is ready or false if the offset is not in acceptable range. If the first check returned false the message dialog will be created and visualized for the user. This can be done by calling the setVisible-method provided by the abstract jDialog class. It is necessary to update the offset values continually to keep the user up to date. This is implemented as a separated thread checking the offset and updating the displayed data on the corresponding panel. To avoid killing or interrupting itself the thread is checking if the canceled-variable is set to true on each turn. The dialog is waiting until the thread is finished and then returns the result.

```
1 new Thread(() -> {
2     while(!isTrimDataReady() && !canceled)
3     {
4         updateTimberData();
5         Thread.sleep(1000);
6     }
7     updateTimberData();
8 });
9 this.setVisible(true);
```

This implementation is still work in progress and designed to give the user the option to proceed even if the calculated offset is not in the accepted range. The proceed button and the cancel button will both set the canceled-variable to false to end the checking thread as well as setting the result-variable to the corresponding value. When this functionality is tested in the CCC this check can be either done fully in the background without any displayed dialog or with a continue button that is only clickable if the offset is in the accepted range.

After the check is accepted the trim data is loaded from the logging database according to the given time stamps.

```
1 // for left and right magnet with associated time stamps
2 TimberExtractor.getTrimResult(
3     leftQuadrupole,
4     hardwareHandler.getLsaSelection().getIrMap().get(ip),
5     startTimestampLeft,
```

```
6     endTimestampLeft,
7     energy);
```

The data is then saved to the specified directory and sent to the results panel by calling the printResult-method. The results are holding three maps with the timestamp as a key and the current, tunex and tuney as their values.

This method will call the showResults-method of the linked result panel for the current task. In this case, we have to cast the result panel to the corresponding TrimChartResultPanel object to be able to call the showResults-method which accepts the trim results as parameters. The trim data is then plotted in the result panel to display the results to the user and saved in the k-modulation process object for following analyzing tasks.

5.2.2 Load an existing trim from database

Since loading a trim also provides data for the following analysis this task is also implemented as a modulating task. This task holds the trim data represented with the given start and end time stamps for each magnet and the identification number of one specific interaction point. During execution, the trim data is loaded with the given timestamps from the logging database using the given extraction interfaces.

5.3 Analyzing Tasks

After the modulating task is finished the trim data is provided in the k-modulation process object. The analyzing tasks can now use these values for further analysis. Each process has a sorted list of analyzing tasks which can be executed by calling the startAnalyzingProcess-method. At the current state, there are two implemented analyzing tasks.

5.3.1 Trim Analysis

This task runs the script for calculating the β^* and waist shift values of a given trim. This is done by executing the corresponding python script as

used in [13]. Since this task has to provide some results to the user it is necessary to extend the abstract AChartTask class and implement its printing methods.

The functionality to call a python script from Java was already used in other OMC software and just had to be ported to the k-modulation GUI. In order to find the path to the corresponding python script, it is necessary to set a specific identifier in the programversions.properties file as well in the new script class as mentioned in 3.4.

Reading the resulting TFS file

After the script is fully executed the results are saved in two separated TFS files for each beam in LHC. This file is formatted as a table with column names and unique string labels. The following listing visualizes the resulting TFS file and its keys and values:

```
1                LABEL            BETASTAR        BETASTAR_ERR     ...
2 MQXA.1L1-MQXA.1R1.B1.X   0.5060233258616  0.001094636472054     ...
3 MQXA.1L1-MQXA.1R1.B1.Y   0.4983394143296  0.001152421781037     ...
```

A row is representing either the horizontal or vertical plane. The label contains the magnet name and interaction point number, as well as x or y for the plane. Each parameter can be accessed with the specified column name by using a TFS reader which was already implemented by the OMC team. The following code snippet shows how the values are read from the TFS file for beam 1:

```
1 TfsFile tsfB1 = TfsReader.readFile(resultFileB1);
2
3 Float[] betaStarB1XY = tsfB1.getFloatData("BETASTAR");
4 Float[] betaStarB1XYErr = tsfB1.getFloatData("BETASTAR_ERR");
```

Since these values are pretty precise it is necessary to trim them in order to provide a simple visualization in the GUI. They are passed in a map to the view class for display. This is done for each parameter the same way.

5.3.2 Beam Orbit Extraction

The orbit extraction was already implemented and its functionality was now ported to the new task infrastructure. In order to execute the orbit analysis script, the orbit has to be extracted for the two measurements using the online model extractor. This is done in two separated threads for each side.

After the orbit is fully extracted and both threads have finished the task will execute the external program located in the BetaBeat.src directory for the analysis. Since the orbit extraction is a task which runs in the background it will not display any data to the GUI and therefore has no result panel.

5.4 Process Pipeline

Each task needs a different amount of time for its execution making it necessary to implement a handling pipeline for a variable amount of individual assembled processes. In order to provide the optimal order of execution, the process queue has to implement two checks. Both checks are applied using the Java comparator class to sort the list of k-modulation processes. These comparators are used to provide an ordering for collections of objects that do not have a natural ordering like in this case.

5.4.1 Sorting Algorithm

Since the trimming can only be done at one interaction point at the same time, it is important to sort the order to optimize the runtime. The two following checks will provide the optimal order of execution for the new module.

The first check compares two tasks and sort them depending on their priority level. The priority is specified as of how long a task will take in order to provide the results to the user. This priority for each type of task is currently set by the developer since these values are not changing during runtime. The task with the higher priority is executed first. The total value of priority for one process is calculated during runtime by adding up all priority values of its tasks. Since the analyzing tasks are executed in a separated thread and the modulation trim takes always the same amount of time it is worth to execute the process with the highest priority value first.

The second check compares the trimming task of two k-modulation processes. If the trimming task is just loads trim data instead of executing a full interaction point trim, it is much better to load the trim first in order that this process can analyze its result in the background. Loading a trim from the database takes a negligible amount of time.

The following example will explain the algorithm in a specific scenario:

- **Process1 (P1)**: priority = 3, trimming;
- **Process2 (P2)**: priority = 3, loading trim from database;
- **Process3 (P3)**: priority = 5, trimming;

The k-modulation processes are added to the queue with the order: P1, P2, P3 as visualized in Figure 5.3.

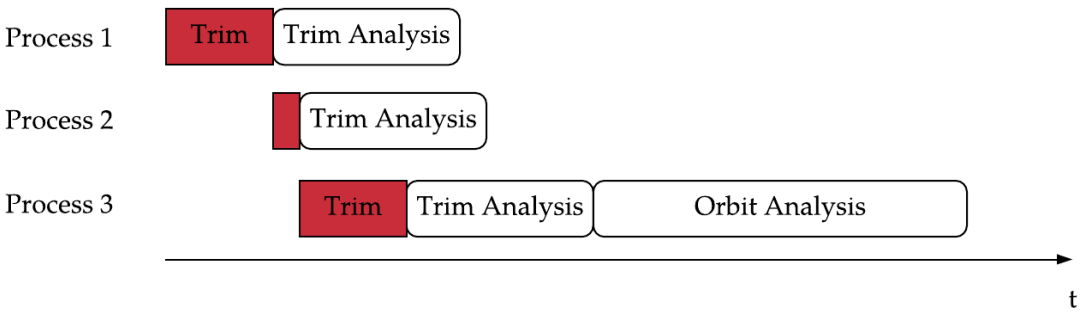

FIGURE 5.3: Simplified scheduling diagram for three specific processes in the queue. The time critical trim is visualized as a red rectangle. Process 2 is extracting a trim from the database. The following analysis, attached to each process, does not have to be synchronized and is started in a separated thread. In this case, the list is unsorted without optimization in runtime.

If we would execute in the unsorted order it would take longer since the longest and heaviest process is executed at the end of our queue. In addition, the second process will wait until process1 is finished even though it could already start analyzing the trim since loading the data from the database will only take a negligible amount of time.

The first check will sort the list depending on their priority value resulting in the order: P3, P2, P1. Now the modulating task of the longest process P3 is executed first but the second process is still blocked by the modulation of Process1 which will be fixed in the second check.

The second check will sort the list depending on their trimming task resulting in the optimal execution order: P2, P3, P1 which is visualized in Figure 5.4.

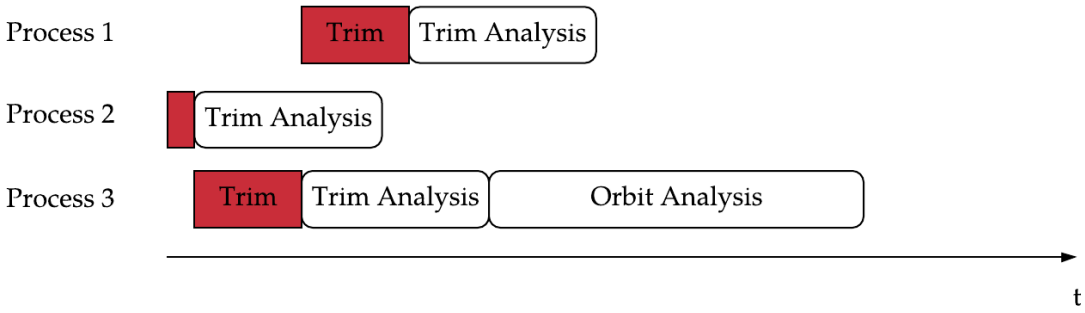

FIGURE 5.4: Simplified scheduling diagram for three specific processes in the queue. The time critical trim is visualized as a red rectangle. Process 2 is extracting a trim from the database. The following analysis, attached to each process, does not have to be synchronized and is started in a separated thread. In this case, the list is sorted and provides therefor a significantly shorter runtime.

Since this is implemented by overriding the compare-method of the java comparator class the list can be sorted really easily by using the following statement:

```
1 Collections.sort(kmodProcesses, new ThreadComparator());
```

5.4.2 Execution Scheme

Executing a trim at one interaction point can only be done once at a time so it is important to synchronize all the k-modulation processes. Since executing the trim is always the first step of a process we do not have to use semaphores or other synchronization variables. The process queue executes the trimming task of the first k-modulation process and waits until it is finished. After that, it will start the analyzing tasks of this process in a new thread. This is done for each process in the queue. The reference for each thread is stored in a check out list to be able to wait for each started thread.

```
1 // starting every task
2 for(KmodProcess process : kmodProcesses) // for each KmodProcess in
      the queue
3 {
4     if(process.startTrimProcess()) // if trim was successful start the
          analyzing process in a new thread
5     {
```

```
6           Thread analyzingThread = new
                Thread(process::startAnalyzingProcess);
7           analyzingThread.start();
8           threadList.add(analyzingThread); // we add the threads to a
                list to wait until each is finish
9       }
10  }
```

5.5 Process Factory

Since each k-modulation process is built depending on the content of the input panel object it is the best way to provide a factory for this kind of instantiation. The process factory takes a reference to the input panel, extracts its input values and creates the ensuing k-modulation process object.

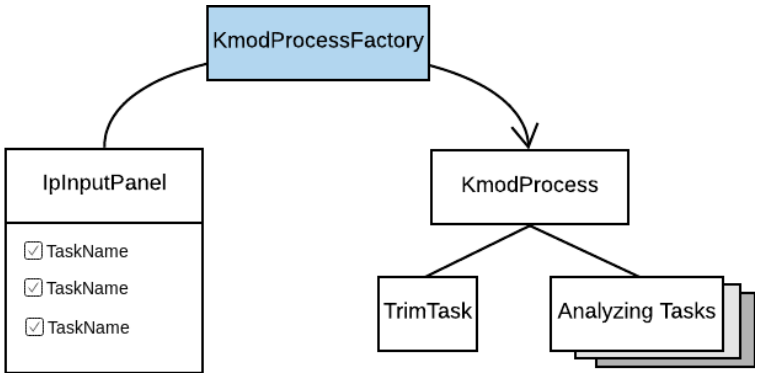

FIGURE 5.5: Schematic overview of the *k*-modulation process factory functionality.

The input panel provides several checkboxes for each task and the corresponding input fields. By using this design pattern it is made easy to add new tasks if needed in future development. To make sure the reference for each process to its view class is set properly, this matching is also done in the k-modulation factory.

Setting References

Each task needs the reference to its corresponding view classes to be able to inform the user about its status and new results. One results view holds a specific amount of result panels depending on the number of tasks in this *k*-modulation process.

The factory always checks first if the checkbox for the task type is set. After that, a new result panel is created and added to the process result view. The panels are stored in a map with the depending task name as the key. This is necessary to be able to identify each panel, while providing a variable amount of result panels.

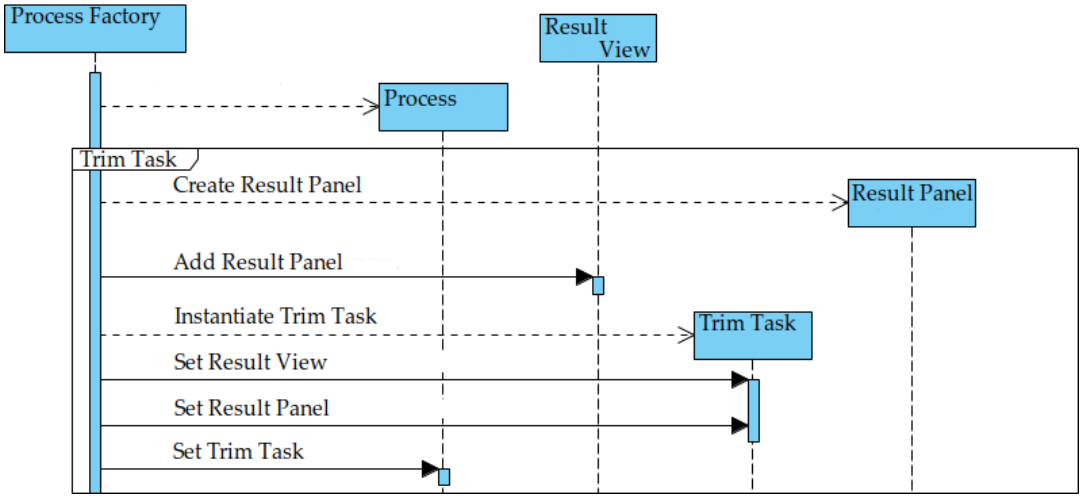

FIGURE 5.6: Sequence diagram showing how the factory sets the reference between view and task

After the panel is created and added to the result view the task can be created using a private factory method which extracts all the significant input variables and returns the created task object. Before adding the new analyzing task to the process it is important to set the references to the result view and panel. This is done for every task currently present in the input panel until the process is fully arranged.

```
1  if(ipPanel.getAnalyzeTrimCheckBox()) //checking value of check box
2  {
3      AnalyzingTrimResultPanel analyzingResultPanel = new
           AnalyzingTrimResultPanel();
4      resultsView.addResultPanel(ETask.ANALYZING_TRIM,
           analyzingResultPanel);
5
6      AChartTask analyzingTrimTask = CreateAnalyzingTrimTask(ipPanel,
           ipSaveDirectory);
7      analyzingTrimTask.setResultsView(resultsView);
8      analyzingTrimTask.setResultPanel(analyzingResultPanel);
9      process.addNewAnalyzingTask(analyzingTrimTask); //add the new
           analyzing task to the KmodProcess
10 }
```

5.6 Input View

The main dialog holds a variable amount of input panels which can be added by the user. This can either be done by clicking on the IP buttons located on the left panel or by using a wizard which loads trim from the database. Each input panel stands for one k-modulation process and has its own input parameter and options. These panels are displayed as tabs in the main dialog.

The user has now several input fields and options for the k-modulation and its later analysis. Each optional analyzing task is represented with a checkbox in the option bar. By checking these boxes the input panel will enable/disable the associated input tabs located in the center area of an input panel. These tabs provide input fields for all required parameters that have to be set for the corresponding operation.

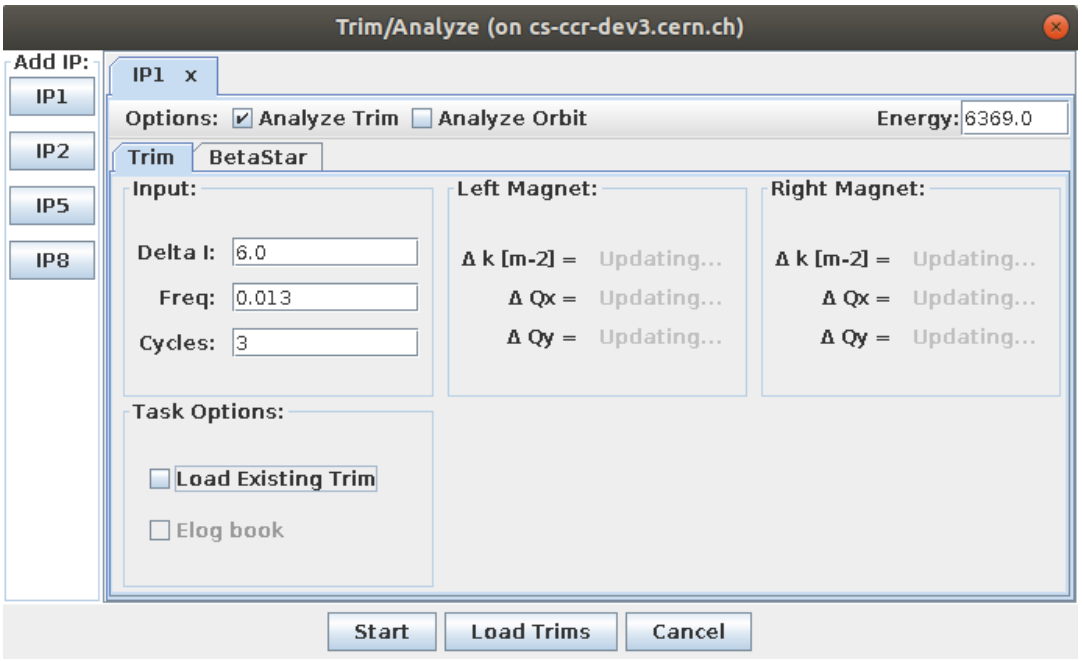

FIGURE 5.7: Input panel with one process at interaction point 1

The input panel is also used as input for the k-modulation process factory so it has to provide some functionality to check the input parameter from outside. This is simply implemented by getter-methods returning the input field value or booleans (in case of checkboxes).

Before the whole process is started each input field and parameter is checked in order to avoid any crashes during runtime. Each invalid input is then displayed to the user in order to be able to fix the fault immediately.

The beam energy value during the k-modulation can be edited with an input field in the top bar. Since the energy value is also logged in the CERN database it can be extracted live by timestamps to provide the exact energy value during measurements. In most cases, a wrong energy value invalidates the results and should be avoided at all costs. Extracting the last value for a specific database variable was already provided as an API. Since the energy value will not change that often the database delay can be ignored.

Modulation Parameter

This modulation input tab represents the trimming task and is always enabled. The user can either load an existing trim or input the parameter for the modulation. If the user wants to load a trim another dialog will appear as seen in Figure 5.8 providing some options to choose an existing trim for the current interaction point of this process tab.

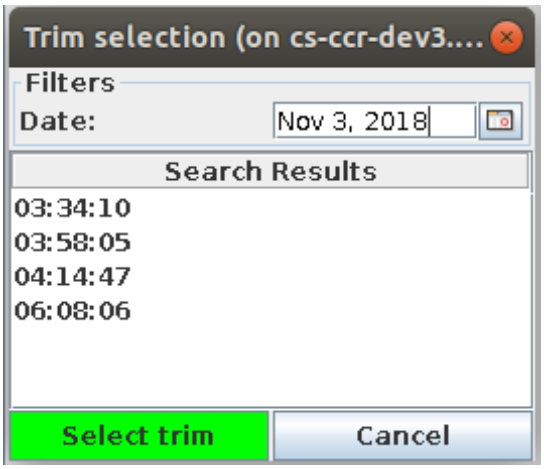

FIGURE 5.8: Trim selection dialog to import an existing trim from the database to an existing process tab.

Since one process tab represents a k-modulation at one interaction point the results are filtered by the corresponding date in the drop-down menu. The user can select one entry and proceed with pressing the green "select trim" button. The input fields are disabled if a trim is loaded and indicated as seen in Figure 5.9. This can be undone easily by unchecking the corresponding checkbox.

The sinusoidal modulation parameters that can be chosen by the user are the trim amplitude (Delta I), trim frequency (Freq) and the number of modulation periods (Cycles).

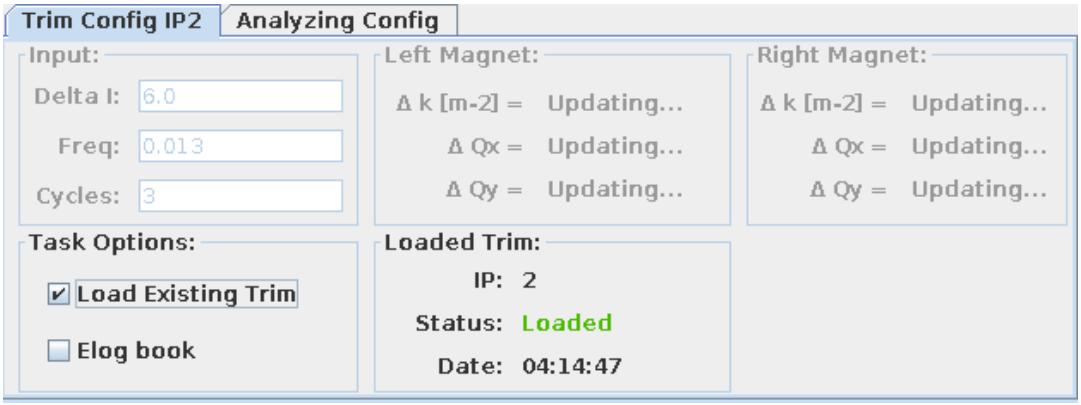

FIGURE 5.9: Trim input tab indicating a trim is already loaded

Identifier	Description
Delta I	Change of magnet strength in ampere, default: 6.0A
Freq	How fast the magnet strength is changed, default: 0.013
Cycles	How often this modulation is done, default: 3

These values are preset with some standard values that are most often used. Some features are still disabled per default since they need some further testing during measurements in the CCC like the magnet change preview.

Analysis Parameter

This input tab is shown in Figure 5.10 is separated into two sides each for one existing beam in the LHC. One side holds the input fields for all needed parameter in order to run the β^* analysis script. It is also possible to provide only one estimated β^* value as input by using the round optics option. This checkbox will disable the input for the β^* in the vertical plane and uses the horizontal input for both.

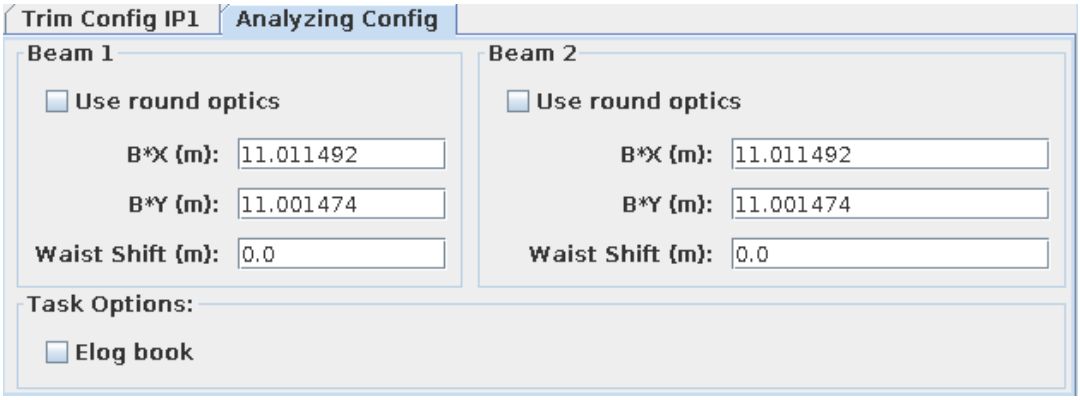

FIGURE 5.10: Analyzing trim input tab

Identifier	Description
β^* X [m]	Estimated β^* in x plane as double value.
β^* Y [m]	Estimated β^* in y plane as double value.
Waist Shift [m]	Estimated waist shift as double value.

All values are preset if the corresponding model is loaded as mentioned in Section6.1 for beam 1 or 2. If no model is loaded the input fields remain empty to throw an error during the input check before execution.

5.7 Trim Selection Wizard

Selecting multiple existing trims is implemented in a separated dialog shown in Figure 5.11. In this dialog, it is possible to select already existing trims from different dates and interaction points to simplify the re-analysis of data, for example with different input parameters or optimized python scripts. The search results are displayed on the left side while the already selected trims are listed on the right side with several basic options underneath. After confirming the selection each selected trim is added as a new input panel tab to the main dialog. The options are then preset as selected in the input panel.

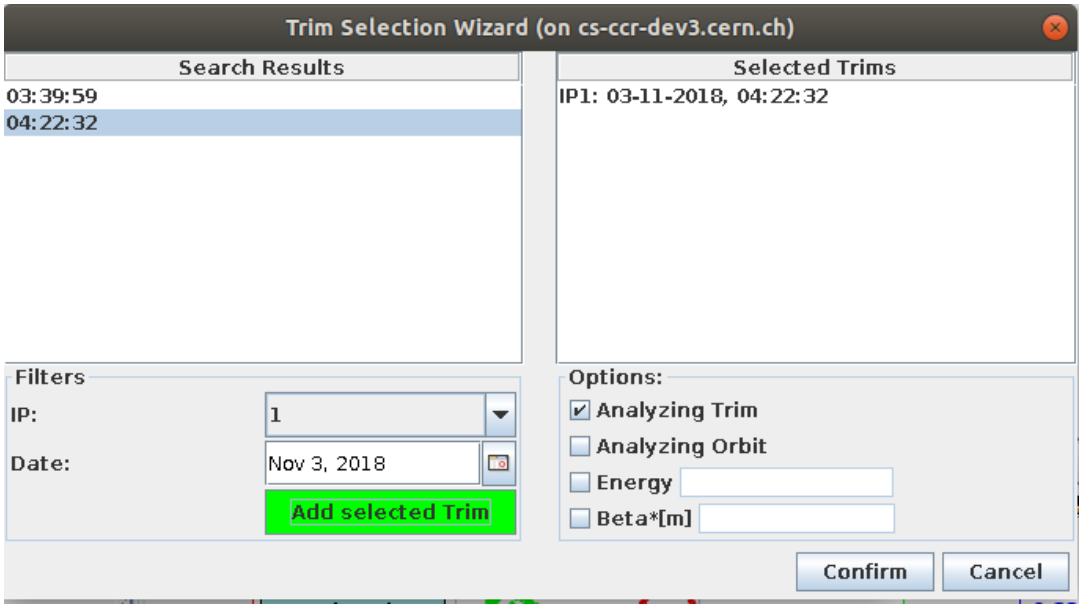

FIGURE 5.11: Trim selection wizard

Searching existing trims

The simplest but also the most computing-intensive way to show all trim data in the database would be to extract all trims and filter all the data with the given date and interaction point value of the GUI elements. Since this would take too much time, each trim is stored in a separated object, holding the start and end times for each magnet as well as the interaction point number. These objects are saved with the start and end timestamp in a file located in the CERN data storage. Since this file is in the well-known TFS format it is easy to extract the data for one object based on the start time stamp.

Each object holds all parameters to display the needed information for the user selection. After the dialog is created all existing trims are represented in the mentioned trim class. The search results on the left side are then filtered depending on the selected date and interaction point. The user can add existing trims to be extracted by double-clicking the entry or by selecting one or multiple trims and pressing the "Add selected Trim" button. This is implemented using a jList already provided by java swing utilities.

Each time the GUI elements are updated the whole list is filtered by the specific values and added to the model of the jList by adding a listener to the components. By using the lambda expression functionality provided in Java 8 this can be implemented in the following two lines:

```
1 dayChooser.addPropertyChangeListener((PropertyChangeEvent e) ->
      filterInList());
2 ipComboBox.addActionListener((ActionEvent e) -> filterInList());
```

The filterInList-method will apply the day and interaction point filters to the list containing all the trims which is loaded at the creation of the dialog. All objects that pass both filters are added to the model of the jList:

```
1 DefaultListModel<String> listModelToShow =
      (DefaultListModel<String>)jListTrimData.getModel();
2 listModelToShow.clear(); // clear the currently displayed entries
3 for(LhcTrimData trim : allTrimData) {
4     if(passIpFilter(trim) && passDayFilter(trim)) {
5         listModelToShow.addElement(
6             TimberTrimExtractor.getTrimTimeAsString(trim.getStartLeft()));
7     }
8 }
```

Applying selected options

After the selection is confirmed by the user, the data of each selected trim has to be converted to an input panel displayed in the main dialog. If the trim selection dialog is disposed by clicking on the confirm button it will return the newly created input panels to the main dialog. Creating these input panels is simply done by checking the state and values of the GUI elements inside the options panel and create a new input panel and set these options by using the provided setter methods:

```
1 for(LhcTrimData currentTrim : selectedTrimsData) {
2     IpPanelEntry panel = new IpPanelEntry(currentTrim.getIP(),
          currentTrim);
3     if(analyzingTrimBox.getSelectedObjects() != null) { //is selected?
4         panel.getIpInputPanel().setAnalyzingTrim(true);
5     //...
```

This is done for each option displayed in the trim selection wizard. The input panel class also provides an option for passing a trim object. By using this constructor the trim is directly loaded after instantiating the input panel. After each selected trim is represented as an input panel all the panels are returned in a list and then added to the tab panel on the main dialog.

5.8 Result View

After starting the k-modulation process, the result view is getting loaded to the main k-modulation panel. Figure 5.12 shows this panel during modulation. Each process is represented as one tab displaying its name. The tab holds the status panel on the left side and the result panel on the right side. During modulation, the result panel is used as a live view of the significant parameters like the magnets current and beam tune in the horizontal and vertical plane.

Each tab can be expanded in another frame by double-clicking the label to get a better view of multiple running processes as seen in Figure 5.14. By closing the frame the process panel gets added back as a tab to the main panel. The whole result view has an active thread bar displaying all the active processes

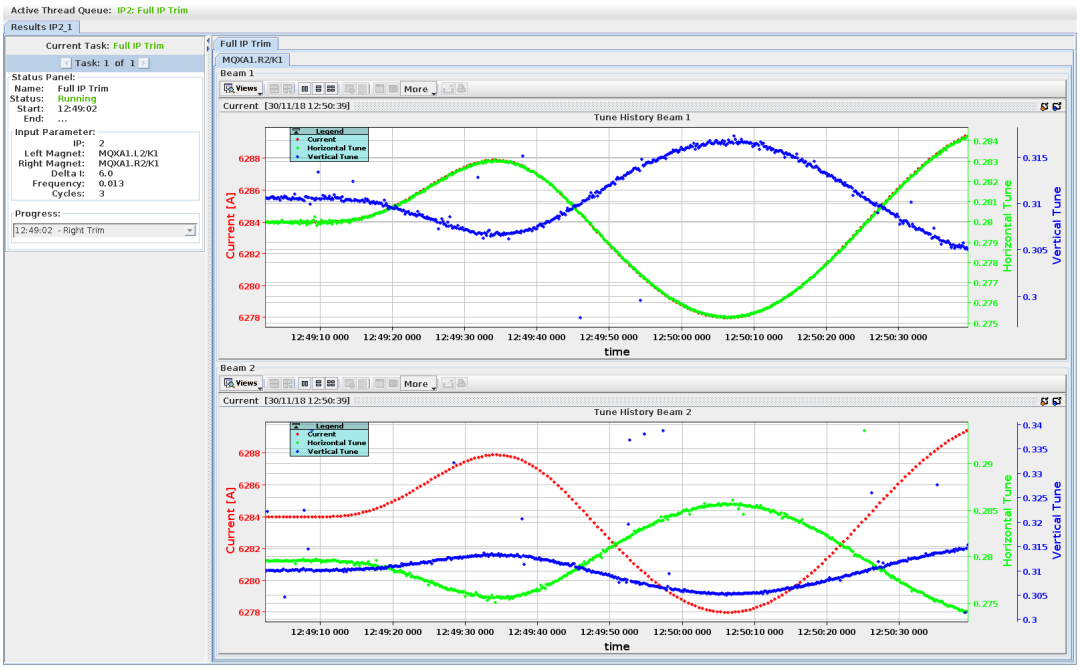

FIGURE 5.12: Result view during k-modulation trim

with their current executed task as seen in Figure 5.13. Each time the associated k-modulation process is starting a new task the label gets updated to the tasks name.

FIGURE 5.13: Thread queue displaying the status of each process

The connection between view and process is necessary to update GUI elements with useful information for the user. It is set as mentioned before in Section 5.5. The task of a k-modulation process is using this connection to send status, logging entries or results to the GUI components. These methods are mostly implemented in the abstract task classes which are calling the implemented setter methods of the associated component.

If the status of a task changes during runtime the user has to be informed about this event. In order to do that the task is calling the inherited printStatus-method which is already implemented in the abstract AChartTask class. This method is then changing the current status displayed in the status panel by accessing this value through the setter-method. Changes during runtime are then executed in the java swing thread to avoid multiple access to GUI components.

FIGURE 5.14: Example for extracting one running process from the main panel in another frame. This can be done for every task to provide an online comparison about multiple running tasks.

5.8.1 Status Panel

The status panel keeps the user up to date about all information of each task in the current displayed process. Figure 5.15 shows the status panel after the analysis of a trim was successfully executed. The main goal of this panel is to provide tractability for each task. Each task is represented as a new private class holding all the information to display. These task objects are created and added to the status panel as soon as the k-modulation process is fully created. The task information contains the name, status, start-/endtime, input parameter, the progress log and the position of the current execution order. These tasks can now be displayed in the status panel providing all the mentioned data to the user.

The user can switch between the different tasks by clicking on the left and right arrow buttons located on the top. The task information is stored in a list making it easy to just load the next or previous task information to the status panel.

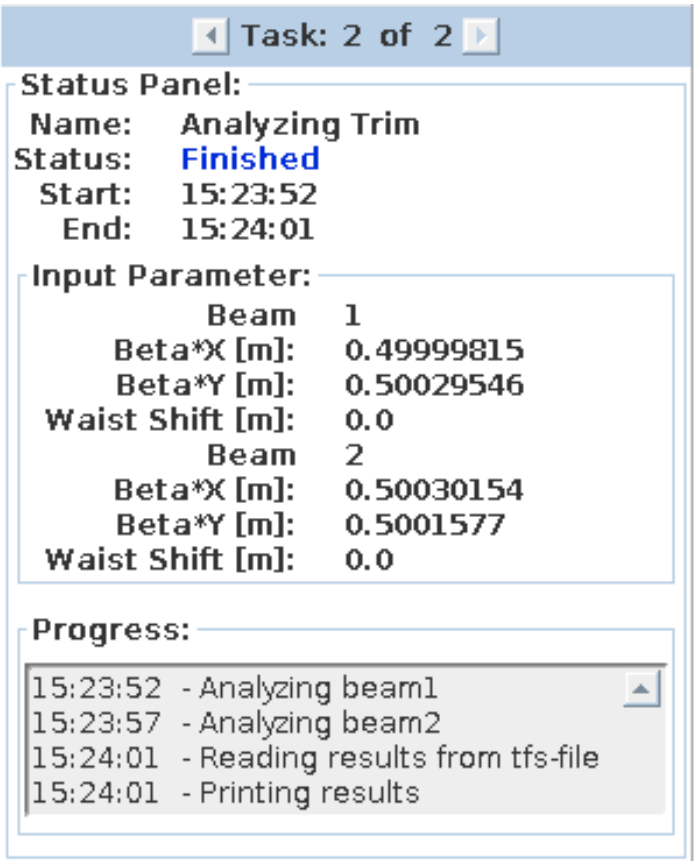

FIGURE 5.15: Status panel displaying the task information for an analyzing task.

Updating GUI Components

Each time a linked task sends new data to display the values in the task information object is updated. If a message is received from the currently displayed task in the status panel the GUI component is directly updated. In order to give one example for the connection between task and status panel, the following listing shows how the status is set for one task and updated in the GUI. The information for one task is held inside the task road map and can be identified by the corresponding task enumeration:

```
1 public void setStatus(ETask task, ETaskStatus status) {
2     this.taskRoadMap.get(task).setStatus(status); // change status of
          task
3     if(activeTask != null && task == activeTask.task) {
4         updateStatus(); // updating status GUI component
5     }
6 }
```

This is done the same way for each parameter or value which has to be updated during runtime by the associated process. Those changes in the GUI are executed in the java swing thread to avoid freezes or crashes by using the swing utilities invokeLater-method:

```
1 // inside updateStatus()
2 SwingUtilities.invokeLater(() ->
      taskStatusLabel.setText(activeTask.getStatus()));
```

Status Types

Different kinds of status are implemented as an enumeration holding the displayed string and color to make it easier to change the status label. Each time a task is sending its status to the status panel the label and color are updated according to the passed enumeration object. The color code is as following:

Status/Color	Description
Starting, Green	Task is instantiating all needed objects or waiting for some I/O operations
Running, Green	Main functionality of this task is currently executed
Finished, Blue	Task finished
Waiting, Blue	Task is waiting for some synchronization variable (not used at the moment)
Scheduled, Gray	Task is in the queue and ready to be executed
Crashed, Red	Task crashed because an exception is fired or a critical error occurred

The progress log provides an in-depth view of which steps the task already executed. This collection of logging entries are stored as a list in the task information object. Per default, the progress log only shows the last logging entry. By clicking on the arrow key located on the top right side the user can expand the log console to see all console outputs for this specific task.

5.8.2 Result Tabs

The result panel is displaying the tasks results directly after their execution. It is separated into several tabs. Each tab is representing the results for one associated task. This tab is added while creating the task for a process from

the k-modulation factory. After the task is finished with its calculations or operations it will send the results to the corresponding result panel tab.

Each new result panel extending the abstract class AResultPanel has to implement the showResults-method which is getting called by the associated task. The passed result data is then displayed according to the panels implementation. In the current state of the software, this is done for trim data and the β^* results. Plotting the trim data was already implemented and was adjusted to fit in the new environment.

Trim Results

Creating the trim chart was already implemented and provided by the OMC team. Since the code for this plotting is already creating a java swing panel for a given trim data object it was added as a tab for the given result panel.

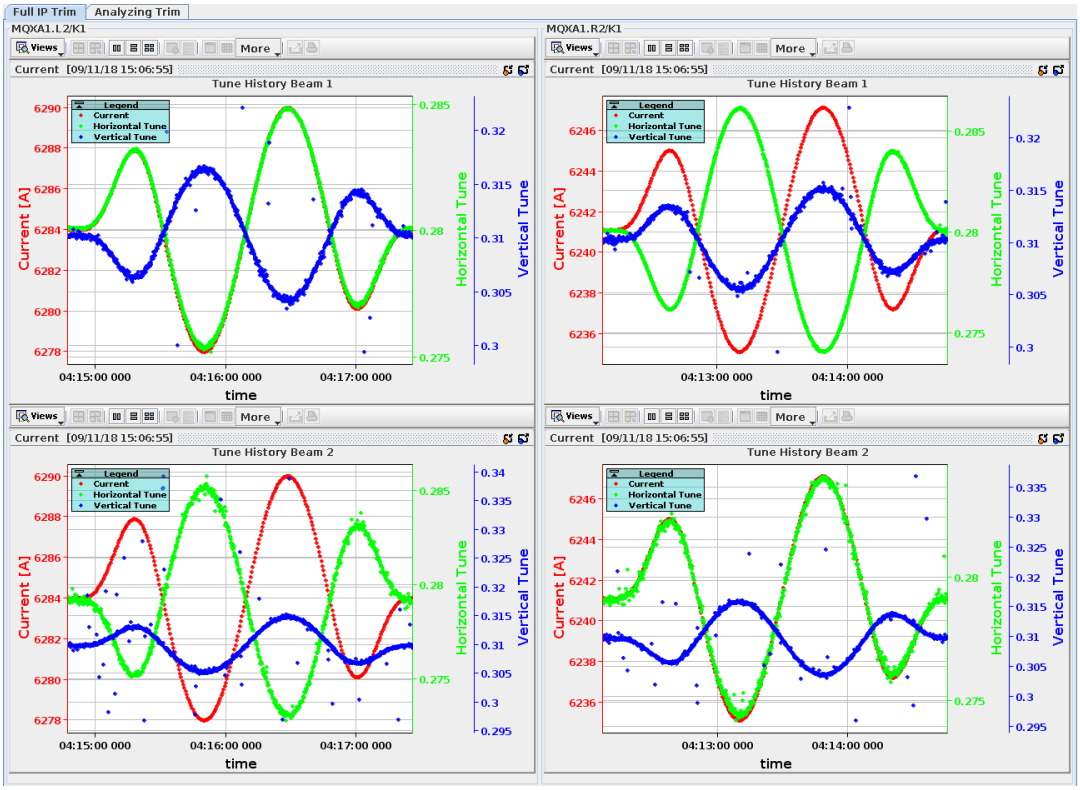

FIGURE 5.16: Result panel for trim results

β^* Results

After the analyzing script is finished and the associated task has read all the data from the resulting TFS file as mentioned in 5.3.1 the data is sent to the result panel by calling the showResults-method. The parameters include β^* in horizontal and vertical planes and the waist shift for both beams in the LHC and their error value. Each parameter is passed in a map with its associated unique identifier.

Since the result panel should also display the tune change depending on the current change on each magnet for both planes the task also has to provide some TFS file paths to the trim data containing this information. The chart is then created using the extracted current, tune and error values using the provided plotting tools from the CERN JDataViewer library.

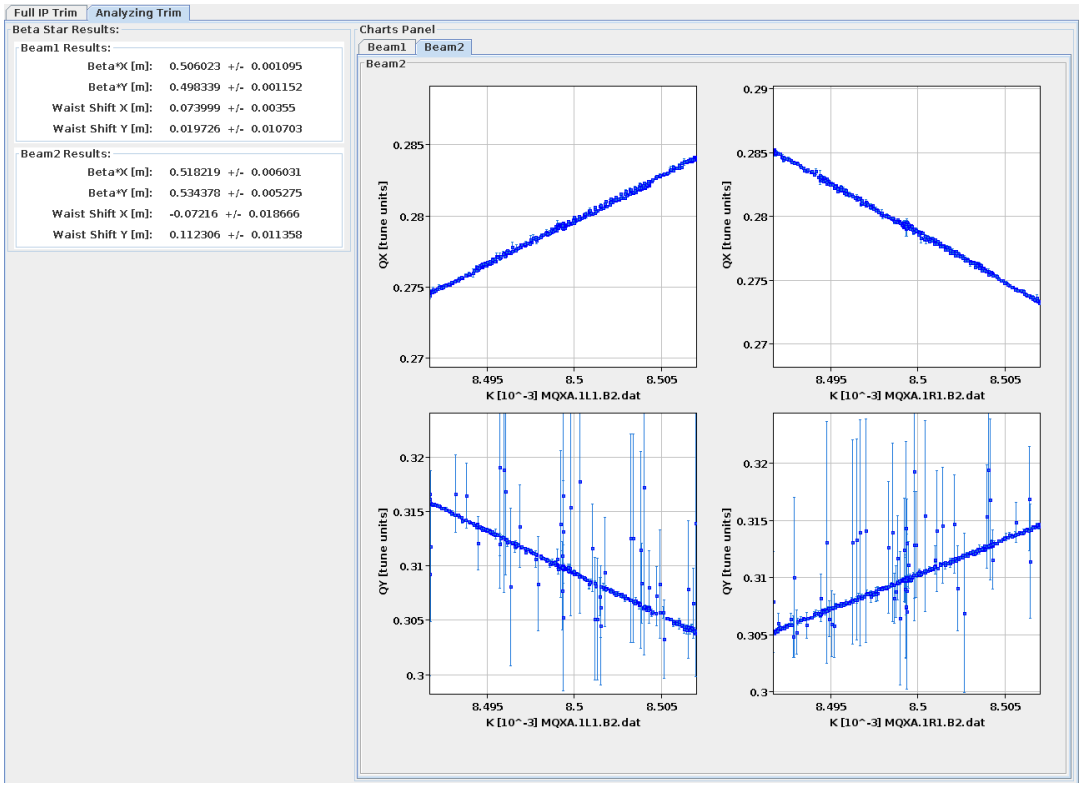

FIGURE 5.17: Result panel for beta star results displaying the values on the left side and the corresponding chart on the right side

5.9 Error Handling

In case of an error during execution, the *k*-modulation process will create an error dialog to display where the error occurred. If a task returns false or throws an exception, further execution should be stopped, since it may lead to more errors which could badly influence other running processes. Each task has to handle uncritical exceptions on its own sending the stack trace to the console and info GUI panel using the beta beating logger implemented by the OMC team.

The error dialog is shown in Figure 5.18 containing an error text with the information what went wrong and a task schedule displaying all tasks in the current process and their status. If a task crashes the main text is created with the name of the corresponding task. Since precise logging is already implemented in the *k*-modulation GUI the new message dialog will just refer to the console.

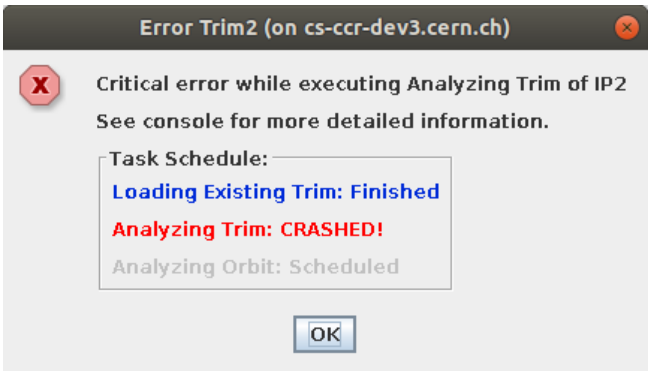

FIGURE 5.18: Error dialog with task schedule

In case of an error, the process adds a label for each task displaying the name and the corresponding status to the dialog providing an overview where the error occurred. A crash during one *k*-modulation process will not affect the other running processes.

5.10 LHC Feedback Check

Since the state of orbit and tune feedback during *k*-modulation changes the result significantly it is important to warn the user if these switches are in the wrong state. Before a *k*-modulation process is started the new module will check the state of both feedbacks and proceed to depend if their value is as

expected. During a normal measurement, the orbit feedback has to be on and the tune feedback has to be off.

If the state of both feedbacks is as desired the dialog will not show up and the *k*-modulation process is executed as expected. In the other case, the feedback check dialog will show up displaying the current state of both switches on the left side. Both feedbacks are in the correct state if both panels on the right side are displayed in green labeled with "OK" as seen in Figure 5.19.

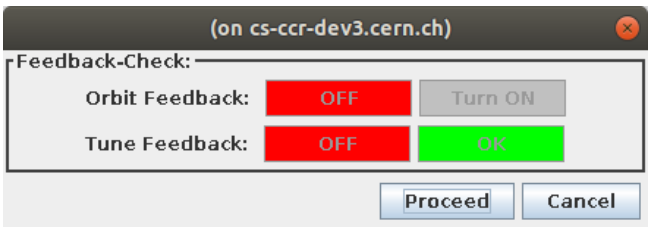

FIGURE 5.19: Feedback check dialog

The state of each feedback can also be obtained by using the CALS API. Each feedback is represented by one unique identifier and stored as a boolean value in the database. This dialog is implemented with a separated thread checking these values in the background while the main dialog is blocked until the user either clicks on proceed or cancels the whole operation. The following while loop checks both feedback values until both are in the correct state or the user canceled the whole check by clicking on the cancel button.

```
1 while(!isFeedbackCorrectState() && !canceled) {
2     setOrbitState(TimberExtractor.isOrbitFeedback());
3     setTuneState(TimberExtractor.isTuneFeedback());
4     Thread.sleep(2000);
5 }
```

Each time the feedback changes the new value is stored in the logging database, 1 for on or 0 for off. This value can be obtained by extracting the last entry for the specific identifier on the database depending on the current timestamp. Using CALS as a source for the state might be a wrong choice since the delay until the data is present on the database might be too long (approximately one minute). In case the delay is too long during Run III the data should be provided from some other database source, if available.

Chapter 6

Additional Features

This chapter provides information about new additional features which are improving the usability for the user or developer of the existing k-modulation software.

6.1 Loading Accelerator Optics Model

During a measurement, each beam in the LHC is represented with a model holding the accelerator state and its expected values like the tune, estimated β^*, etc. This functionality is used in every software of the OMC team except in the k-modulation software. Here, the user had to enter the beam parameters manually instead. In addition, the model is also needed for executing analysis on extracted orbit. Each model is stored in a directory which contains magnet configurations, and other useful information in TFS format for the k-modulation tasks.

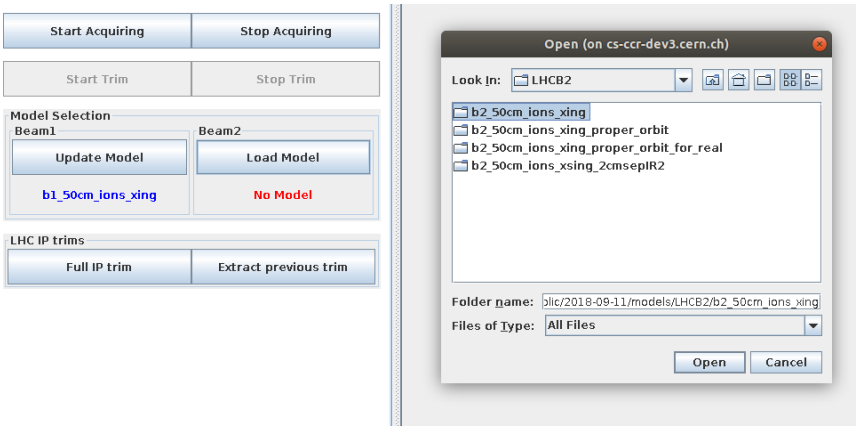

FIGURE 6.1: Model selection dialog.

The user is now able to load the model from the main view, displaying either the loaded model's name or info that no model is loaded. The model can be loaded for each beam of the LHC separately. The model directory is only accepted if it contains the needed files for the k-modulation software. This functionality is implemented in a model handler providing methods to access the needed parameter.

Once the model is selected each parameter is extracted from the corresponding TFS files and stored in the model handler. This is done for the tune values and estimated β^* on each interaction point by using the TFS reader class. The model handler can be obtained using the following code:

```
1  private static ModelHandler modelHandlerBeam1 = null;
2  ...
3  public static ModelHandler getModelHandler(Beam beam) {
4      if(beam.equals(LhcBeam.BEAM1)) {
5          if(modelHandlerBeam1 == null) {
6              modelHandlerBeam1 = new ModelHandler(beam);
7          }
8          return modelHandlerBeam1;
9      }
10     ...
```

This is done the same way for beam 2. By using the singleton pattern for both beams we can ensure that only one model handler exists for one beam at a time, which avoids parameter conflicts.

6.2 Simulation Mode

In the former state of the software, it was only possible to simulate the k-modulation trim by receiving random values as input. Since the new module is adding a lot of new functionality a simulation mode will be useful for developers and users. It is possible to simulate the whole module and the following analysis with realistic current and tune data to test it while not in the CCC.

The easiest way to provide realistic data for each interaction point is to use the CERN logging database as a source for trim. This data is then used as input while executing the standard LHC interaction point trim task and can

be used to test new or optimized analyzing tasks. The needed amount is the same as executing a normal k-modulation during measurement but it is also possible to adjust the displayed speed in the GUI.

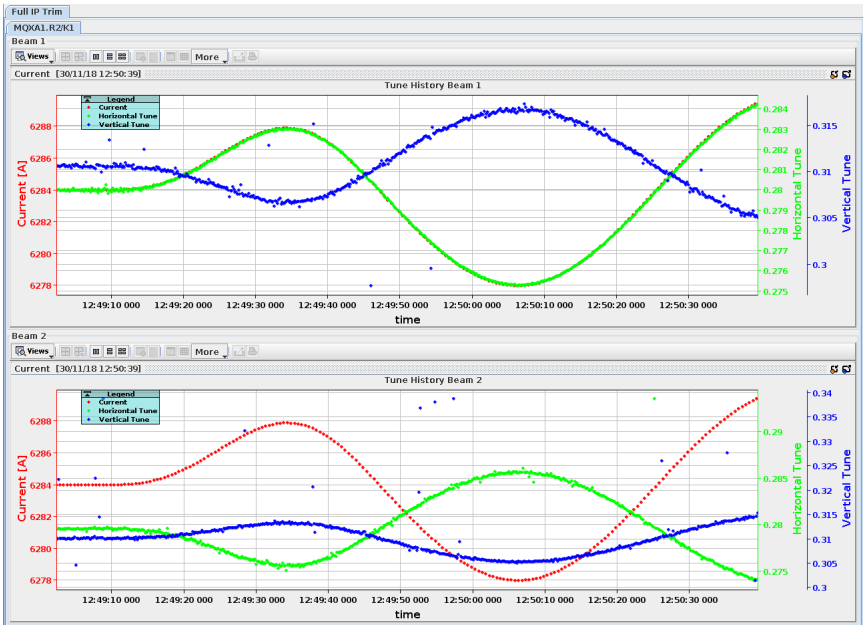

FIGURE 6.2: Simulation mode during runtime

Since this is just an update of the already existing simulation mode it only extends its functionality. In the old state of software, the data was randomly generated and displayed in the live modulation view. The simulation mode can be switched on in the main menu selecting the corresponding item in the mode combo box. In this view, the current and the tune in the horizontal and vertical plane is plotted according to the time. During a real measurement, these parameters are received directly by the associated tune and current hardware handler. The current is queried every 500 milliseconds and the tune every 160 milliseconds. In order to simulate this modulation, it is necessary to provide this data in the exact same time intervals to the displayed modulation panel.

This is archived by using scheduled tasks at a fixed rate which are already provided by the included CERN library. These tasks create and execute a periodic action that becomes enabled first after the given initial delay, and subsequently with the given period. IN this use case, there is no initial delay and the period is depending on querying the data as mentioned before. If any execution of the task encounters an exception, subsequent executions are suppressed. Otherwise, the task will only terminate via cancellation or termination of the executor. If any execution of this task takes longer than its

period, then subsequent executions may be delayed, but it will not concurrently execute, as it is not necessary in simulation mode. The scheduled task can be created with the following code:

```
future.set(executorService.scheduleAtFixedRate(
    () -> {/*things to do*/}, // executable action, lambda expression
    0, //intial delay
    PERIOD / SPEED_MULTIPLIER, // period
    TimeUnit.MILLISECONDS)); // specified time unit
```

Instead of subscribing directly to the magnets hardware handler the data is generated and added to the associated measurement class providing the data for the later analysis. On creation of the scheduled task, the data for each parameter is extracted from the CERN logging database and stored in a list. Each time this action is executed it will send the next entry in the list to the measurement object by using an iterator. This way it does not matter, if the data is send from the magnet handler in normal mode or from the scheduled task in simulation mode. After all the data is displayed the scheduled task is terminated.

Since a normal k-modulation takes about three minutes it is necessary to provide an option to reduce the needed time without changing or reducing the displayed data. This is done by dividing the above mentioned period by using a speed multiplier. The scheduled task is then given a shorter period and the data is added faster to the measurement object. If the new module is started while the simulation mode is active it will display some more options to specify this multiplier.

6.3 Developer Tools

During the implementation, it was necessary to add some functionality to improve the usability for developers. These additions simplify testing new changes made on the k-modulation software or the python code on the local workspace.

6.3.1 User Configuration

By writing a user configuration in a separated user.properties file in the workspace, the developer can define preset values to avoid setting them each time he restarts the software for debugging etc. This is already done in other software in the OMC team and was ported to the k-modulation software. This was easily archived by using the properties class provided by the included java utilities.

6.3.2 Change BetaBeat.Src Directory

In order to test or use some new python scripts, it is important to change the BetaBeat.Src directory during runtime. The directory can be changed in the menu tab of the k-modulation main panel.

Chapter 7

Conclusion & Outlook

The automated *k*-modulation module has been successfully implemented and will now significantly improve the usability for its users and decrease the needed time for data analysis. In case of multiple *k*-modulations on several IP, the module has to be started only once and will present its results online in the GUI components. Usability is improved by updating the user constantly about the current status and new results for each running process. Each modulation can be individually edited and added to the process pipeline. Additionally, the runtime for reanalyzing multiple trim data entries is significantly improved with better overview and visualization of the resulting data making it possible to validate new analysis scripts with several different trim data entries in a short amount of time. The whole module has been implemented as extendable as possible to provide an open interface for new functionality needed in the future.

Since some parts of the new module interact with the LHC infrastructure only analytical and simulation features could be fully tested. Some functionalities like the feedback check in Chapter 5.10 or the database check in Chapter 5.2.1 can only be tested during operation and have to be adjusted during Run III of the LHC in 2021. Therefore, the new module is fully documented and provides an attached wiki with sections for both, developers or users. This will make it easier to adjust and extend the existing structure in the future.

The development turned out to be more difficult than expected since the existing *k*-modulation software was completely undocumented except some imported CERN libraries. This lead to some bugs during early tests in CCC. After these issues where determined the new module could be executed with success during the last machine development session before Long Shutdown 2.

In order to optimize the code quality in future development, the whole LHC infrastructure should be simulated enabling the possibility for automated tests for critical operations on the LHC hardware. Often bugs in the existing *k*-modulation software cannot be easily located since they only appear during measurements in the CCC. Automated testing should definitely be a hot topic in the coming years to improve the code quality significantly. Also, the *k*-modulation software should be refactored since some modules might be outdated and have not been revisited since creation. Besides that, the code quality should be improved by ongoing code documentation and naming conventions for parameters, classes, and methods. This can be achieved by defining a coding style guide for the used programming language and is necessary since the code is extended and edited by several different developers in a short amount of time. To find an optimal solution for the upcoming challenges in software engineering these changes should be discussed and optimized with other CERN IT departments if possible to benefit from their experiences and recommendations.

In addition to that, several basic modules like calling an external python script as mentioned in Chapter 5 are implemented in multiple software tools of the OMC team. This redundancy should be avoided by providing these functionalities as common libraries.

Appendix A

Other Projects

A.1 Resonance Line Plotting

As mentioned in 2.2.3 resonances have to be avoided to avoid beam losses due to magnet imperfections. Machine resonances can be seen in the spectrum of beam position data. For easy identification expected resonance lines in several orders are visualized in the frequency chart located in the BetaBeat GUI mentioned in Chapter 3.2. This gives a better overview of where the resonances are located in the frequency spectrum. In order to compare the measured and model tune resonance lines the user is able to select of them via dropbox. It is possible to add custom lines and export the chart as PNG.

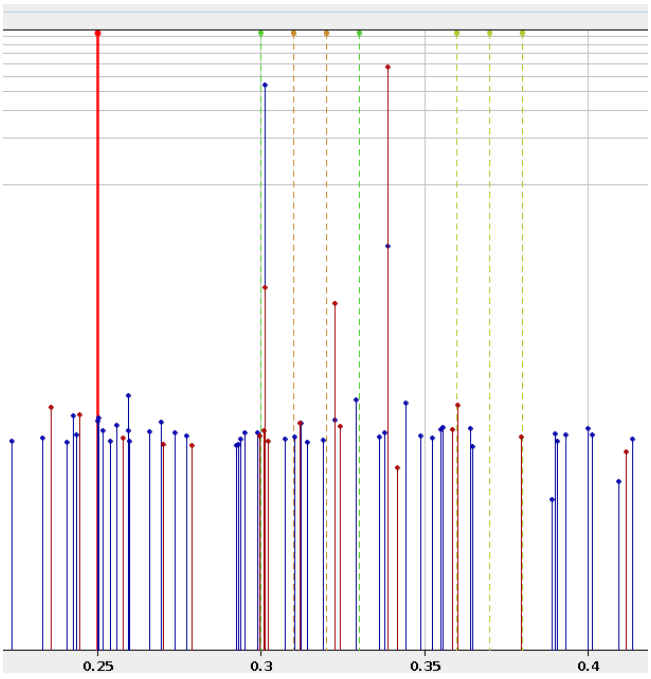

FIGURE A.1: Resonance chart displaying resonance lines with different orders and one custom line in red.

Each resonance line is listed in the legend and indicated with a specific color for each corresponding order. While hovering over the resonance line the label is displayed to provide an overview even in zoomed view of the chart.

Resonance Line Calculation

The resonance lines are linear combinations of the tune values of the LHC. These values can be gathered from the loaded model or directly from the selected measurement entry. Each tune value is multiplied by a given factor j and i, whose sum defines the order. For every sign combination of both factors, the frequency is calculated and mapped to the frequency range.

```
for (int i = 0; i <= rangeEnd; i++){
    for (int j = 0; j <= rangeEnd; j++){
        int order = i+j;
        if (order >= rangeStart && order <= rangeEnd){
            addAndMap(i, j, order);
            addAndMap(-i, j, order);
            addAndMap(i, -j, order);
            addAndMap(-i, -j, order);
}   }   }
```

The following listing shows how each resonance frequency is calculated with the tune values and its given factors:

```
float freq = (i * tuneX + j * tuneY) % 1f;
if(freq > 0.5) {
    freq = 1 - freq;
}
else if(freq < 0) {
    return;
}
```

The frequency chart displays the frequency's from 0 to 0.5. Each frequency above 0.5 has to be wrapped around. Frequencies below 0 can be ignored as the spectrum is symmetric. Each plotted frequency is represented as one object. The resulting objects are then added to the frequency chart data set using the provided plotting tools from the CERN JDataViewer library.

A.2 Smart Zoom Interactor Extension

Working with data is always paired with a corresponding visualization which is used in every software tool of the OMC team. In the current state of the software, these charts can be adjusted by zooming in and out. This is done by selecting a specific data frame with the mouse. The zoom interactor will then changes the chart depending on the selected window. This functionality is native to the developed Chart Viewer. In case the selection is not as expected the chart has to be resetted and the user has to do another attempt to select the right data frame. This has been improved, by being able to adjust the selected data frame after zooming, to increase the usability of the plotting tools of the OMC software.

A.2.1 Drag

Since the selection often has just a little offset to the desired data window a drag functionality was a needed extension to the existing zoom interactor. The new extension takes care about the different plotting styles including arithmetic log axis.

Each chart is represented by pixel and chart coordinates. The pixel coordinate system is the displayed size of the chart on the screen and the chart coordinate system is depending on the plotted data. Java already provides functionality to detect events for mouse buttons or movement which are used in this extension. These events are always detected in pixel coordinates and have to be translated into the chart coordinate system in order to change the displayed data borders of the chart. The drag action can be started by pressing the left and right mouse button at the same time. On release the chart movement is stopped.

The mouse events are caught by overwriting the corresponding processMouseEvent for the mouse buttons and processMouseMoutionEvent for its movement. In order to start a drag action, each fired mouse button event is compared with a specified event id and a button mask. For the drag action this is implemented as seen in the following listing:

```
1 public boolean isEndDragEvent(MouseEvent evt)
2 {
3     return // mouse event id if button is pressed
4           evt.getID() == MouseEvent.MOUSE_PRESSED
```

```
5
6           // mask for left and right mouse button.
7           && evt.getModifiersEx() == (InputEvent.BUTTON3_DOWN_MASK |
                InputEvent.BUTTON1_DOWN_MASK)
8
9           // boolean value to detect if dragging is already performed
10          && !dragging;
11  }
```

By using the dragging boolean variable it is possible to differ between a normal mouse movement event and an event fired while the the user is dragging. Each time this movement event is fired the event coordinates are compared to the start coordinates before the event occurred. This position is saved in a private global variable for the instance of this chart interactor. By detecting this mouse position offset in horizontal and vertical direction, new borders are calculated in the referenced frame. The displayed data range are then adjusted by translating the pixel coordinates to the chart coordinate system.

Since the pixel coordinate system of the chart also includes the legend and axis title, this chart offset has to be taken into account for the pixel coordinates of the borders. The new borders are saved as a rectangle represented by an upper left and a bottom right point. The following listing shows how the two points in pixel coordinates for the borders are calculated:

```
1  // upper left pixel point
2  int newMinX = xOffset + xChartOffset; // starting always with 0
3  int newMinY = yOffset + yChartOffset; // starting always with 0
4
5  // bottom right pixel point
6  int newMaxX = this.getPlotRect().width + xOffset + xChartOffset;
7  int newMaxY = this.getPlotRect().height + yOffset + yChartOffset;
```

These pixel points need now to be translated to the chart coordinate system by using the already implemented functionality of the coordinate system class. With this approach the drag and drop extension is independent of the axis style of the chart.

A.2.2 Mouse Wheel Zoom

The mouse wheel zoom extension is slightly different to the drag extension. There, it is necessary to detect the mouse wheel event which is not part of the mouse event or the mouse movement event. In order to detect this event, the chart interactor has to implement the MouseWheelListener and overwrite its mouseWheelMoved method.

Like in the drag extension the zoom is performed by finding new data borders in chart coordinates. The mouse wheel zoom functionality follows the same idea and calculates the new borders in pixel coordinates and translates them to the chart coordinate system of the corresponding chart. The zoom direction is determined by the wheel movement direction. This can either be a positive or negative integer value. Depending on this value the border values are increased or decreased. This direction value is used to provide an adaptable zoom scale for zooming in and out. The scaling value is calculated by the following formula:

$$1.5^{direction} \tag{A.1}$$

In case of a zoom in movement event the direction is +1 and will therefore provide a positive multiplier greater than 1 to the borders. If the direction is -1, the scale value is smaller than 1 leading to a decrease of the borders.

In order to be a useful zoom extension, the performed zoom focuses on the mouse position. As mentioned in Section A.2.1, the pixel coordinate system is slightly offset from to the chart coordinate system since it also includes the legend and axis titles. This offset has to be added to the detected pixel point of the mouse event to perform a comprehensible zoom as seen in the following listing:

```
1 Point correctedPoint = new Point(event.getX() - xChartOffset,
    event.getY() - yChartOffset);
```

The new borders represented by two pixel points can now be calculated using the corrected center point of the mouse event and the scale factor from Equation A.1.

```
1 // upper left pixel point
```

```
2 int newMinX = correctedPoint.x + (xChartOffset - correctedPoint.x)*
    scale;
3 int newMinY = correctedPoint.y + (yChartOffset - correctedPoint.y) *
    scale;
4
5 // bottom right pixel point
6 int newMaxX = correctedPoint.x + (getPlotRect().width -
    correctedPoint.x) * scale;
7 int newMaxY = correctedPoint.y + (getPlotRect().height -
    correctedPoint.y) * scale;
```

These pixel points are now translated to the chart coordinate system using the already provided functionality of the coordinate system class. The zoom is then performed depending on the new data borders.

Confirmation

The supervisor Dr. Rogelio Tomás Garcia confirms that the work mentioned in this report was carried out by the student Martin Spitznagel during the practical semester from 01.09.2018 until 28.02.2019.

Bibliography

[1] ATLAS collaboration homepage. [Online]. Available: https://atlas-collaboration.web.cern.ch/

[2] CMS collaboration homepage. [Online]. Available: http://cms.web.cern.ch/

[3] ALICE collaboration homepage. [Online]. Available: http://aliceinfo.cern.ch/Public/Welcome.html

[4] LHCb collaboration homepage. [Online]. Available: http://lhcb.web.cern.ch/lhcb/

[5] R. Tomás, M. Aiba, A. Franchi, and U. Iriso, "Review of linear optics measurement and correction for charged particle accelerators," *Phys. Rev. Accel. Beams*, vol. 20, no. 5, p. 054801. 16 p, 2017. [Online]. Available: http://cds.cern.ch/record/2270107

[6] T. Persson, F. Carlier, J. C. de Portugal, A. G.-T. Valdivieso, A. Langner, E. H. MacLean, L. Malina, P. Skowronski, B. Salvant, R. Tomás, and A. G. Bonilla, "LHC optics commissioning: A journey towards 1% optics control," *Phys. Rev. Accel. Beams*, vol. 20, no. 6, p. 061002. 9 p, 2017. [Online]. Available: http://cds.cern.ch/record/2272377

[7] W. Lukas, E. Kneringer, and G. Rudolph, "Measurement of Charged-Particle Distributions in ProtonProton Interactions at $\sqrt{s}$ = 8 TeV with the ATLAS Detector at the LHC," Aug 2016, presented on 2016-08-16. [Online]. Available: https://cds.cern.ch/record/2655802

[8] E. Fol, R. Tomas Garcia, and P. Henning, "Evaluation of Machine Learning Methods for LHC Optics Measurements and Corrections Software," Aug 2017, presented 23 Oct 2017. [Online]. Available: https://cds.cern.ch/record/2309558

[9] E. Mobs, "The CERN accelerator complex. Complexe des accélérateurs du CERN," Jul 2016, general Photo. [Online]. Available: https://cds.cern.ch/record/2197559

[10] M. Kuhn, "Transverse Emittance Measurement and Preservation at the LHC," Geneva, 2016, presented 20 Jun 2016. [Online]. Available: http://cds.cern.ch/record/2229712

[11] B. Muratori, "Luminosity and luminous region calculations for the LHC," CERN, Geneva, Tech. Rep. LHC-PROJECT-NOTE-301, Sep 2002. [Online]. Available: https://cds.cern.ch/record/691967

[12] M. Hofer, M. Benedikt, and R. Tomas, "Optics Design for the Low Luminosity Experiments in the FCC-hh," 2017, presented 28 Nov 2017. [Online]. Available: https://cds.cern.ch/record/2640686

[13] F. S. Carlier and R. Tomas Garcia, "Beam Optics Studies in the Large Hadron Collider: Observations on an Anomalous Octupolar Resonance Line in the LHC – and – Accuracy amp; Feasibility of the β^* Measurement for LHC and HL-LHC Using K-Modulation," Aug 2015, presented 28 Aug 2015. [Online]. Available: https://cds.cern.ch/record/2057179

[14] P. C. V. Thrane, R. Tomas, and J. A. Støvneng, "Measuring β^* in SuperKEKB with K Modulation," Dec 2018, presented 2018. [Online]. Available: https://cds.cern.ch/record/2652855

[15] T. Bach and R. Tomas, "Improvements for Optics Measurement and Corrections software," Aug 2013. [Online]. Available: https://cds.cern.ch/record/1595800

[16] Betebear.src, many scripts and files for optics simulations measurements in the lhc. [Online]. Available: https://github.com/pylhc/Beta-Beat.src.git

[17] C. Roderick, R. Billen, R. D. Gaspar Aparicio, E. Grancher, A. Khodabandeh, and N. Segura Chinchilla, "The LHC Logging Service : Handling terabytes of on-line data," CERN, Geneva, Tech. Rep. CERN-ATS-2009-099, Nov 2009. [Online]. Available: http://cds.cern.ch/record/1215574

[18] R. Billen and C. Roderick, "The LHC Logging Service: Capturing, storing and using time-series data for the world's largest scientific instrument," CERN, Tech. Rep. AB-Note-2006-046. CERN-AB-Note-2006-046, Nov 2006. [Online]. Available: http://cds.cern.ch/record/1000757