Preliminary study on major design issues of the integrated beamline software platform

1. Introduction .......................................................................................................................... 3
   1.1 Status of software on beamlines at ESRF ..................................................................... 3
   1.2 Ways of improvement: the need for an integrated beamline software platform ........... 5

2. Specifications for an integrated beamline software platform ............................................ 5
   2.1 Communicating with beamline instruments and devices ............................................. 5
      2.1.1 Heterogeneous hardware brought together ......................................................... 5
      2.1.2 Middleware ......................................................................................................... 6
      2.1.3 Unification layer: abstraction, adapter and emulation roles .................................. 7
   2.2 Data acquisition & experiment control ......................................................................... 8
      2.2.1 Role of a sequencer ............................................................................................. 8
      2.2.2 Writing sequences ............................................................................................. 8
         2.2.2.1 Graphical approach ...................................................................................... 9
         2.2.2.2 DSL approach ............................................................................................. 9
         2.2.2.3 Computer scripting language approach ....................................................... 9
      2.2.3 Multi-tasking abilities ............................................................................................ 9
      2.2.4 Architecture of the sequencer ............................................................................. 11
         2.2.4.1 Approach A: Sequencer and Unification Layer as servers ................................ 12
         2.2.4.2 Approach B: Sequencer embedding the Unification Layer ................................ 13
         As for approach A, here multiple clients can connect to the sequencer, the sequencer being a server ............................................... 14
         2.2.4.3 Approach C: Sequencer and Unification Layer as libraries ............................ 14
         2.2.4.4 Locking issues ............................................................................................. 15
      2.2.5 Interacting with users ......................................................................................... 16
      2.2.6 Data handling: the data manager ....................................................................... 18
         2.2.6.1 Role of the data manager ......................................................................... 18
         2.2.6.2 Data storage ............................................................................................... 19
         2.2.6.3 Data display ............................................................................................... 19
   2.3 Configuration .................................................................................................................. 20
   2.4 Graphical interfaces in an integrated beamline environment ........................................ 21
      2.4.1 Monitoring ........................................................................................................ 21
      2.4.2 Running experiments ......................................................................................... 21
      2.4.3 Visualizing data ............................................................................................... 22
      2.4.4 Current model for graphical interfaces ............................................................... 22
      2.4.5 Evolution of the model .................................................................................... 23

3. Conclusion ........................................................................................................................ 24

Fig. 1: Base components of the Beamline Integrated Software Platform .......................... 4
Fig. 2: Overview of the sequencer’s architecture ............................................................... 11
Fig. 3: Communication between the unification layer, the sequencer and its clients........ 13
Fig. 4: Sequencer embedding the unification layer .......................................................... 14
Fig. 5: Sequencer and Unification Layer as libraries ..................................................... 15
Fig. 5: The Conductor manages access to devices between the different processes using the
Unification Layer library (example from approach C) .................................................. 16
Fig. 6: Different kind of users are interacting with the sequencer to control the beamline .... 17
Fig. 7: Example sequencer user interface ..................................................................... 18
Fig. 8: Current model for graphical interfaces applied to the Integrated Beamline Software
Platform .......................................................................................................................... 23
1. Introduction

1.1 Status of software on beamlines at ESRF

Over the years the Bliss group at ESRF, acting both as a support team for instruments control and as a software development force, introduced standard tools and common practices on ESRF beamlines. That was a big challenge: each beamline has its own issues and way of doing experiments; however, since the early days effort has been put in trying to find generic solutions to common beamline data acquisition and experiment control problematics.

This reasoning does not imply that predefined solutions are hopelessly tried to be applied to specific problems: it is quite the opposite. Specific problems on beamlines are opportunities to improve the generic software and solutions, whenever it is possible. The generic approach helps a lot on several aspects: all Bliss members share the same know-how, and it becomes easier to spread state-of-the-art techniques when people share a common background. Then, every beamline can easily benefit of some software development triggered by another one. And even more importantly, the overall quality of service is improved compared to a situation where everybody develops and maintains his own piece of software: bugs have a higher probability to occur and thus to be solved, a wider range of cases are taken into account, and software usually has already been tested. When better solutions are found to some problems the standard software is updated accordingly, enforcing a kind of “natural selection” within the software ecosystem.

To allow such an organization within the Bliss group, software tools need to fulfill some criteria. Flexibility comes in mind, but also stability, durability and efficiency. This last quality may be difficult to quantify. Of course, an efficient software can be the one that simply “does the job”. Often, tough, more subjective criteria apply like ease of use, ease of configuration, required time to do a task, user-friendliness, etc.

Software on beamlines can be classified in 4 groups:
- Drivers and other low-level software (e.g. hardware test software)
- Device or instruments control
- Data acquisition & experiments control
- Graphical interfaces for experiments

An integrated beamline software platform covers the 3 last items from device or instruments control to graphical interfaces. There are also other aspects that need to be taken into account into an integrated beamline software platform:
- Configuration
- Software deployment

The Bliss group currently uses different technologies and different kind of software depending on each task group:
- MIDDLEWARE: Taco / Tango control systems are mainly used for device or instruments control
- SEQUENCER: CSS Spec is mainly used for executing data acquisition sequences
- GRAPHICAL INTERFACES: the Bliss Framework (based on Python / Trolltech Qt) is the main tool to build graphical interfaces on top of the two previous systems
- SOFTWARE DEPLOYMENT: it is achieved using a couple of in-house developed
tools, Bliss Builder and Bliss Installer. The builder is creating RPM files, that can be deployed on beamlines via the installer.

![Diagram of Beamline Integrated Software Platform components]

**Fig. 1 : Base components of the Beamline Integrated Software Platform**

The drawing above shows the different base components of a Beamline Integrated Software Platform. Next paragraphs will focus on each component, detailing required specifications and major design issues.
1.2 Ways of improvement: the need for an integrated beamline software platform

In our current beamline control software some components are missing and some need to be improved or upgraded:

- there is no centralized configuration
- there is an overlap between the different software components; there is no overall architecture thought about since the beginning on how the different components should interact. Almost none of the tools are limited to a particular domain: it is possible to communicate with a device on a beamline using Spec directly, for example, instead of relying on a Tango server. It is also possible to write a data acquisition sequence in Python within a graphical application built from our graphical user interface framework. As there is no clear path, multiple solutions are easily found to a problem which can lead to tware odds and ends
- interoperability between the systems requires “glue code” to be maintained, since every system has its own preferred communication protocol
- the sequencer doesn’t offer multitasking abilities
- the sequencer language is too limited in term of data structures, flow control and programming language features
- the data format used to store acquired data doesn’t fit our needs anymore
- our standard data visualization tools (while data acquisition is going on) need to be improved in their functionalities and usability. They should be used in the graphical applications built on top of the control system as well as in our general data visualization applications.

2. Specifications for an integrated beamline software platform

2.1 Communicating with beamline instruments and devices

2.1.1 Heterogeneous hardware brought together

A beamline consists of dozens of heterogeneous devices (motors, encoders, pneumatic actuators, switches, cameras, etc.) and scientific instruments brought together. In the case of scientific devices sold by some company, there is no control on how the particular device will be connected to the beamline; a wide range of communication buses and protocols can be used: serial line (RS232, RS432, etc.) with some ASCII-based or binary protocol, GPIB (IEEE 488) with some specific binary protocol, TCP/IP sockets again with some protocol, CameraLink, USB...

Sometimes it is possible to decide on a common way to address some particular devices, for example at ESRF the new ICEPAP motor controllers can connect to an Ethernet network and
it is possible to “talk” with the controller using TCP/IP sockets. Then, all motors that can be plugged on an ICEPAP controllers and can be controlled using this protocol, which is very convenient. But this is not the most usual case.

An integrated beamline software platform has to provide a solution for communicating with these different instruments and devices, using different protocols but in an uniform manner. Indeed, all the job of such a platform is to communicate with these components, to execute experiment control sequences, to acquire data and to display results.

2.1.2 Middleware

In order to provide such communication layer from the hardware to the integrated beamline software platform, middleware is to be used.

What is “middleware”? Middleware is computer software that connects software components or applications. The software consists of a set of enabling services that allow multiple processes running on one or more machines to interact across a network.

At the ESRF, we first used TACO as middleware. Starting from 2001, an evolution of TACO has seen the light of day at ESRF: TANGO. New developments are using TANGO but in order to keep our investment, TACO is still used for devices with very low update rates.

TANGO uses CORBA for doing network communication. CORBA is a language independent standard for implementing distributed objects on the network. CORBA is an evolving standard and is continuously adding new features which are of interest for controls e.g. real-time, embedded, component model. TANGO is available for several platforms including major ones like Windows and Linux. TANGO application programmer’s interface (API) is provided in C++ and Java. TANGO uses the omniORB implementation of CORBA in C++ and JacORB for Java.

Bindings exist for Python, C and some various proprietary development platform like NI LabView. The API and bindings allow to control many devices but also from the client side to write applications to access the TANGO devices.

In TANGO, devices are distributed in the beamline environment following a client/server model. For example, a motor controller device server can run on a dedicated computer, whereas the beamline user applications are running on a different host. This architecture has a lot of advantages:

- CPU/processing load is shared among the different computers of the beamline
- the whole software environment becomes more fault-tolerant, in case of hardware breakdown only some servers becomes unavailable
- it facilitates integration of any kind of equipment or device, without platform considerations: if some devices require a Windows computer, they can run on a dedicated Windows machine, even if the rest of the beamline computers are mainly running Linux
- clients can be informed of a change, or can receive notifications from servers in the form of asynchronous events
At the moment, TANGO fulfill our needs in terms of middleware. However, an integrated beamline software platform cannot be viewed only as separate distributed entities: the “glue” between the servers is essential.

### 2.1.3 Unification layer: abstraction, adapter and emulation roles

In addition to accessing hardware to make devices running, more than often it is mandatory to add some extra higher-level behaviours. Beamline components cannot only be viewed as independent distributed devices, as they consist of hardware components built together to achieve a particular functionality. For example, a slit system is made of 2 motors moving 2 blades but gap and offset, calculated from the blade positions, are the useful values. At one point, the upper layer of an integrated beamline software platform need to deal with components having complex behaviour and not with a collection of simples ones. The unification layer should provide an abstraction role.

Experience has proven that using a common middleware will never guarantee that devices of same kind are accessed the same way. Moreover, a middleware change or evolution is possible, as it already happened in the past. Then, it is important to consider having an adapter layer that would flatten the differences on how hardware is accessed. This way, whatever middleware system(s) are used, the upper layers of the integrated beamline software platform will always be able to access devices without changes. All changes would be centralized in the unification layer.

In an ideal world, distributed servers from the middleware for the same kind of hardware devices would always share the same interface, allowing for example clients to seamlessly access 2D detectors (CCD cameras) using the same commands and attributes. However, this is almost always impossible because hardware is inherently different: some hardware capabilities cannot always be found in another one. Taking the example of the camera, the ability to retrieve only a partial region of interest (RoI) from a full frame can exist in one manufacturer API and not in another one. However, the upper layers would rely on this feature. An important role of the unification layer would be to emulate missing functionalities.
2.2 Data acquisition & experiment control

Data acquisition and experiment control require operations to be executed one after the other: that's what we call sequences. Sequences are run by a sequencer. What is the role of a sequencer? How sequences can be written? User needs to have full control over what the sequencer is doing, and to be able to interact with it: how interaction with user can be done? And what can be the architecture of a sequencer?

2.2.1 Role of a sequencer

On a beamline, the sequencer is a key part: from the middleware unification layer (see previous section), it allows to get access to the hardware and helps implementing the logic behind any experiment. If motor movements are synchronized by hardware, or if data acquisition is made very close to the hardware for performance reasons, the role of the sequencer can be reduced. However, a sequencer is always needed in one form or another.

The sequencer executes sequences; sequences are usually written in a computer programming language. As sequences need to be changed often, it is highly desirable to be able to write these sequences in an interpreted language.

The interpreted language for the sequencer should:
- be extensible (allows to interface existing libraries in C or C++)
- provide complex data types (tables, dictionaries, lists, sets...)
- have name spaces (to avoid polluting the global name space)
- offer debugging possibilities
- have the essential modern features of scripting languages nowadays: first-class functions, iterators, closures, Object Oriented capabilities, etc.

2.2.2 Writing sequences

In their day to day work, scientists use various computer tools, each one with their own logic and way of working, sometimes with their own programming language or macro facility. We can think of gnuplot, Matlab, LabView, Igor... In the synchrotron world, we can add Spec. Most of the time beamline scientists and users will only have a “working knowledge” of the basic tools they need to run their experiments. For these users writing some simple sequences, like repeating several times a scan changing some beamline parameters between them will be needed. But in any case, complex data acquisition and/or automation sequences will have to be written.

It appears that the choosen language to write sequences will have to be used by staff having very different level of expertise in computing programmation. Thus, the choice of the
language for writing sequences is essential. What are the different approaches to the sequence writing problem? Which language is best adapted for a sequencer?

2.2.2.1 Graphical approach

The graphical approach consists of creating a sequence by assembling small code parts using a graphical tool. LabView is using this approach; the Soleil synchrotron is also using the same approach with the Passerelle tool, on top of the Ptolemy project. The code is written by some software experts, then users only arrange new sequences around some predefined functional bits. The algorithm for the sequences is described in terms of I/O flow between small components.

2.2.2.2 DSL approach

The domain specific language (DSL) approach is the one chosen in some specific tools like Spec. The application embeds a domain specific language; it is designed to make common operations very easy to write since it includes whole concepts around basic control structures and syntax.

2.2.2.3 Computer scripting language approach

This approach consists of using a full-featured computer scripting language as the sequencer language. It opens a lot of opportunities and brings maximum flexibility, at the cost of a non-adapted syntax and some inherent concerns: since all scripting languages are designed to be very general since the ground up.

Depending on the targeted audience, the choice of one approach can appear to be better than another one. The graphical approach still looks like sci-fi: it is probably great for simple sequences, but it certainly becomes exponentially complicated for non-trivial sequences. However, it may be a good idea to experiment. The real scripting language approach “as-is” is probably not a good choice, because it will force users to turn themselves into computer engineers and it will essentially become counter-efficient. Finally an appropriate approach could be to follow the DSL one (language + concepts), using an advanced underlying scripting language (not reinventing the wheel on this topic!).

2.2.3 Multi-tasking abilities

Today, Spec can be considered as the sequencer used at ESRF. One of the ways of improvement is to add the ability to execute sequences simultaneously. Although it is very common nowadays to do multi-tasking with computers (everyone can browse the web while listening to music for example), it is not so obvious for a beamline data acquisition sequencer. Indeed, there are only little cases where having sequences executed in a time-sharing manner is really useful:

- in case of multiple clients connected to a sequencer server, it is natural to not block other clients when a sequence is executed
- in case of relying on events from the hardware (through the middleware and the unification layer, see 2.1), events being asynchronous, “something” has to be listening for notifications. Within the sequencer, this might well be a background task, awaking other sequences in case of changes. For example it is desirable to pause a running scan when the synchrotron machine is refilling. We can imagine having a monitoring task
running in the background, at the same time as a main sequence, and when the machine is not delivering current the main sequence would be paused;

- in some cases, sequences are completely independent – for example, a temperature ramping sequence and a slitbox alignment scan. In some cases, it would gain time if such sequences would have the possibility to be executed in parallel. It could even be mandatory to execute several sequences in the same period of time.

Having a multi-task sequencer makes much more complex the sequencer implementation. Appropriate design choices can help having a more simple and robust implementation. There are two ways of doing multi-tasking (time-sharing):

- preemptive multi-tasking: a scheduler takes care of a group of tasks. Each task is given a certain time “slice”. The scheduler can be viewed as a privileged task – it will run a task for its dedicated time, then interrupt it without its cooperation and give control to another one (“context switch”), following task priorities and/or task loads;

- cooperative multi-tasking: in this model the scheduler only passes control to another task when the previous task “yields”, i.e when it interrupts itself. To ensure proper functioning, tasks have to yield regularly. Of course, if a task hangs or never yields, the whole system is blocked.

At different levels, operating systems offer the possibility to create scheduled tasks using processes (independent running programs, communicating through Inter-Process Communication -IPC- techniques) or threads (independent threads of execution inside a same parent process, sharing the same memory). Nowadays almost all operating systems rely on pre-emptive multi-tasking; Microsoft Windows was using a cooperative multi-tasking model until Windows 95. Today only RISC OS systems still use cooperative multi-tasking.

Inside a sequencer, the same models can be applied.

Pre-emptive multi-tasking can be achieved using OS-level threads, i.e tasks inside the sequencer will be scheduled by the operating system.

Cooperative multi-tasking can be achieved using coroutines. In this case, a specific scheduler or dispatcher inside the sequencer executes coroutines sequentially.

Models can even be mixed, i.e having coroutines running in an OS-level thread.

The big advantage of using coroutines is that it removes a lot of problems related to classic multi-threading: threads are typically preemptively scheduled while coroutines are not. Because threads can be rescheduled at any instant and can execute concurrently, programs using threads must be careful about locking. In contrast, because coroutines can only be rescheduled at specific points in the program and do not execute concurrently, programs using coroutines can often avoid locking entirely. (This property is also cited as a benefit of event-driven or asynchronous programming.)

Coroutines are a way to express multiple cooperating threads of control in a convenient and natural way, but do not execute in parallel, and thus gain no performance benefit from multiple CPU's. However, since coroutines switch much faster than operating system threads and do not typically require complex and sometimes expensive locking mechanisms, using
Coroutines is typically faster than the equivalent program using full OS threads. The only showstopper is blocking I/O: in this case a coroutine will block, thus blocking the main task as well whereas a thread would just block itself. In the context of the integrated beamline software platform, since the sequencer is relying on the middleware and the unification layer, blocking I/O should be handled in those lower level layers.

2.2.4 Architecture of the sequencer

Fig. 2: Overview of the sequencer’s architecture
The sequencer embeds a scripting engine in order to execute sequences (see 2.2.2); most of the time, sequences which are to be executed by the sequencer are written to files, so the sequencer is able to read those sequence files (scripts containing code for the sequencer) and to load them. The sequencer is just on top of the unification layer that gives access to the beamline hardware devices; all high-level entities provided by the unification layer (see 2.1.3, “abstraction role”) are mirrored by the sequencer to make them available as objects within the scripting engine environment. For example, the unification layer can provide measurement groups containing counters – these objects would be proxied inside the sequencer scripting engine, in order to be used in sequences.

The sequencer also embeds a scheduler for multi-tasking (see 2.2.3).

The sequencer writes log files; it helps a lot to follow what the sequencer is doing, or in case of crash to analyse what went wrong. Ideally, all sequencer actions should be kept in rotating log files.

The architecture scheme presented below (see Fig. 2) makes no assumptions on how the different parts of the sequencer and the rest of the system are connected.

Let’s study different approaches for the possible underlying design coming from the architecture.

2.2.4.1 Approach A: Sequencer and Unification Layer as servers
A first approach consists of having the unification layer and the sequencer as servers. Indeed, sequences often need to be started/monitored/started/etc. from other programs like graphical interfaces for example, and other applications can also need to have access to the unification layer in order to access beamline devices. Servers guarantee the different layers are independent, and enforces the minimum number of connections to middleware devices since objects created by the unification layer would be unique. There can be a performance issue in case of big amount of data being transferred from the unification layer to the sequencer, because there will be 2 transfers: one from the middleware to the unification layer server, and one from the unification layer server to the sequencer.

**2.2.4.2 Approach B: Sequencer embedding the Unification Layer**

A second approach that can be considered when putting in place the architecture for the integrated beamline software platform is to have the unification layer running within the same process as the sequencer.

The unification layer becomes a library exporting high-level objects to the scripting engine, i.e. merging unification layer and objects proxied inside the scripting engine. Then no
performance issue has to be considered accessing the devices of the unification layer from the sequencer. Other programs would import the same library to access hardware, if direct access cannot be avoided.

As for approach A, here multiple clients can connect to the sequencer, the sequencer being a server.

### 2.2.4.3 Approach C: Sequencer and Unification Layer as libraries

A third approach is to consider both the sequencer and the Unification Layer as libraries. Any process can have access to all the sequences installed and can use any device by importing the libraries.

In this scheme the sequencer doesn't exist as a computer program itself; it is designed to be embedded into host applications. The scripting engine would become part of those host programs, importing the sequencer library. The sequencer library would provide the basic facilities like access to sequence functions loaded from files, etc.

Different applications would have to import the libraries in order to be able to execute sequences or to access hardware.
2.2.4.4 Locking issues

Locking is a common problem of the three approaches previously described. Concurrent access to the same hardware at the same time has to be prevented. It is not always possible to have a device level lock, because the middleware might not support this feature (for example, Taco does not offer device server locking). Thus locking has to be done at the unification layer level. As a benefit it would protect whole abstractions instead of single components, e.g. protecting access to the “slit” instead of protecting access to the 2 blades motors.

For the first approach, a classic locking mechanism is enough. For the second and third approaches, as multiple instances of the objects provided by the unification layer library are living at the same time within different processes (eventually running on different computers), the locking mechanism needs to introduce an additional component. This component would be a server. This server would have a very defined role, and could be viewed as a conductor known by all processes importing the unification layer library: for each of them, it would manage access to devices from the middleware.

For example, if a motor is moved through the graphical interface, first the motor would be locked by the graphical interface thanks to the Conductor, then the motor would be moved through the Unification Layer, then released with a call to the Conductor and finally information would be emitted to other other clients sharing the same hardware resource, e.g. the CLI application, that the hardware device has changed its state (“dirty flag”).
2.2.5 Interacting with users

At ESRF Spec, the sequencer, is also the first access to beamline experiments offered to the user through its CLI (Command Line Interface), which allows commands and functions to be executed by taking inputs from the computer keyboard.

A more than 10 years experience has proved that this kind of direct-access interface, together with an interpreted language for sequences, is a really nice way to bring flexibility, to allow a fast and easy implementation and to offer a high-level workbench when testing new hardware on beamlines. Moreover, after a reasonably small learning period, any user can drive her own experiment by typing or executing herself dedicated commands and functions from the sequencer CLI. The command line interface has to be enough to drive any experiment.

Several kind of users are using a beamline and need to have access to the sequencer:
Fig. 7: Different kind of users are interacting with the sequencer to control the beamline

Each kind of user brings its own culture; the sequencer, through its command line interface, should allow each kind of user to be efficient with the tool very quickly. This implies a number of sequencer functionalities to be available through the user interface:

- to have an interactive prompt to display sequencer output and to communicate with the sequencer (CLI);
- to have a global overview of the sequences currently under execution, to get status information and feedback;
- to display logging messages in a quickly accessible area;
- instant access to important system information like where data are saved, configuration tools, fast access to the most used sequences, etc.

Nowadays everyone operating on a beamline is used to computer graphical interfaces; coupling a simple graphical interface together with the command line prompt would allow to offer a full-featured fancy interface to users. Ergonomics should be studied carefully in order to produce a nice user-friendly tool, that would really help users of the sequencer. It has to be fast and to provide obvious access to all basic tasks, while allowing immediate handling for more experienced users:
A full-featured integrated beamline software platform has to provide a sequencer, and through a CLI it should also play the role of the first interface for the beamline user (scientist, technician, software engineer...).

At this point, the choice of the language for the command line interface has to take the user factor into account. It can be envisaged to have a dedicated language for the CLI, in order to facilitates the use of the sequencer through the command line interface by non-computer expert users. There is already a language choice to make for the sequencer language (see ). Ideally, it would be the same both for the sequencer and the CLI.

### 2.2.6 Data handling : the data manager

Data handling is one of the more important topics : at the end, to have data produced is the final goal of experiments as far as beamline control is concerned. At the sequencer level, sequences can trigger measurements or scans, that will produce data. Some high-level paradigms have to be introduced in order to be able to deal with data.

#### 2.2.6.1 Role of the data manager

Within the sequencer, data is always accessed through the data manager. The data manager
should offer mechanisms to retrieve data, and to do basic operations on data. For example in case of a scan, the data manager should provide a way to get back data or subset of data to the sequencer for doing further processing (e.g. doing some interpolation, calculating some values out of the data), but also to get back useful values from the scan (peak, inflexion point if relevant, etc.).

The data manager should not automatically transfers data from its source (often a device from the middleware) to its storage area. As data producer devices generate more data faster, it will become crucial to partially manage data on the middleware level. In this case the role of the data manager will be to provide path to data and, only if expressively asked by a client, transfer them.

Another case is well illustrated by the use of video camera. This is generally at the highest level of the integrated beamline software platform (GUI) that images are needed (display purpose). In order to not transfer each image through all the system (not efficient but even not possible for live video), objects of the middleware layer should provide through their unification layer representation, a fast path to their data. For example, a file based shared memory should be used to display live video images, this is the role of the middleware server to indicates to the clients the name of file to be used to get the images.

### 2.2.6.2 Data storage

The data manager also have the responsibility to save data. Data should be saved in a standard format, offering high performance but also flexibility and the ability to store meta-data within the same file.

It appears that the HDF5 file format from the HDF group is the best candidate. The Nexus project is an effort to define a file format with a consensus on well-defined meta-data information in the neutrons and synchrotrons world, based on HDF5.

At ESRF, some enhancements for Nexus has already been proposed and it is already under evaluation. In case of acceptance of the propositions by the Nexus committee, Nexus will be the recommended file format for ESRF data storage.

### 2.2.6.3 Data display

The data manager should provide a fast access to the data for other applications to display data and associated meta-data. The sequencer, as a server, is not supposed to display data.
2.3 Configuration

Any software system needs configuration. In the case of the integrated beamline software platform, configuration is essential – ideally, it would encapsulate all configuration needs for the different software components in a centralized way.

For the middleware, configuration indicates how to address hardware devices, i.e. which kind of protocol, and where: serial line port number, IP address, GPIB address, internal PCI card address, etc. and contains all parameters that need to be set on the hardware to make it work as expected.

For the unification layer, configuration specifies links between middleware devices and higher level abstractions. For example, configuration for the unification layer would specify that slit S uses blades B1 and B2, or that measurement group M contains counters C1 and C2.

The Data Manager also needs configuration: where to store data? What are data sources? Finally, the sequencer configuration represents the whole set of sequences, and paths for storing log files, etc.

One of the required feature is to be able to save “configuration snapshots” of the whole or parts of the beamline and to be able to retrieve them later, to allow users to save whole experiments setup with associated data acquisition objects.

Configuration has to be stored in a “database”. The “database” is not necessarily a SQL-compliant relational database like Oracle or MySQL. Any structured way of storing parameters is a potential candidate.

Some important criteria are:

- flexibility: the database should handle new kind of hardware easily, naturally; copying configuration from one beamline to another must be straightforward
- reliability: the database should be protected from software crashes
- configuration parameters should be able to be changed very easily with a convenient editor
- browsing configuration should be easy to do
- searching capabilities

Experience has proved that a bunch of text files in a human-readable format is one of the best way to store configuration information.

Configuration tools are highly desirable to change the integrated beamline software platform configuration. Configuration tools should allow users to disable a device, to re-enable it again etc. in a transparent manner, without losing any configuration information. Users should also be able to add new devices newly installed by a technician for example.

This implies to have a complete chain of configuration tools, from the middleware configuration to the sequencer configuration. Ideally there should be a unique entry door to the configuration, although there might be several tools for each part of the system.

Another important feature to have is the history: to be able to follow what happened with the configuration, for example as a log file.
2.4 Graphical interfaces in an integrated beamline environment

The need for graphical interfaces can be characterized in 3 main domains on beamlines:

- to monitor beamline components, or physical quantities coming from sensors
- to run experiments
- to visualize data

The integrated beamline software platform should provide standard tools to build graphical interfaces and applications to fulfill these needs.

2.4.1 Monitoring

More than often, scientists want to have an overview of the state of beamline components at a glance. How much is the machine current? Is it refilling? Is the beamline safety shutter open? Are motors moving? Has the intensity read by a diode changed? What is the temperature of my sample? What is happening inside the experiment hutch?

Complementary to that, they want to be able to access beamline components: to open shutters, to move motors, to change gains for a diode, to change cryo stream flux, etc.

A graphical interface is usually the more convenient way to provide both monitoring and basic controlling facilities to users. On MX beamlines at ESRF, two applications serve these purposes: the synopsis and the control panel. The synopsis shows position and state for all beamline motors, the synchrotron machine current, the state of shutters and two graphs with a 1 hour history, displaying counts for some diodes. The control panel presents more or less the same components, allowing users to do actions like moving a motor, opening/closing a shutter etc.

2.4.2 Running experiments

A scientific experiment at a synchrotron beamline involves a lot of actuators, motors, sensors and detectors, to go from the sample(s) to data.

Some beamlines provide scientific users with a large collection of experiments setup with their associated sequences and these setup may change often. Some other beamlines have a more or less static configuration but provide either very sophisticated setup with specific sequences and/or have very high level of automation. To be able to build graphical interfaces to run each type of experiments the GUI framework needs to be very flexible, as applications will change as often as the experiment will change. The integrated beamline software platform should include a tool to be able to quickly generate graphical interfaces for experiments, and to easily modify existing ones.

Panels are needed for input parameters. Input parameters can be complex: for example, a user might want to click on a video display to specify coordinates for a scan.
2.4.3 Visualizing data

Scientists on beamlines need to have a convenient application to visualize data taken during an experiment. There are two ways of displaying data during an experiment: either by loading it from some file or fully “online”, i.e even before data is saved.

The simplest way to look at data while it is written is the first case, by loading/reloading the data file and displaying its contents using the appropriate software. At ESRF, Oxidis is one of the applications offered to the scientists to visualize data for example.

For online display, the Integrated Beamline Software Platform includes a data manager component within the unification layer; through this component, applications can access data (see 2.2.6) in order to display it.

However, depending on the architecture that will be decided for the Integrated Beamline Software Platform, a major design issue could be the data transfer between the data manager and the graphical online display application. If the data manager and the visualization applications run within the same process (e.g the data manager comes from a library imported by the visualization application), there is no data transfer problem in case of big amounts of data, typically a 2D image. If the processes are separated, but if they run on the same host, there is no problem neither because data transfer can occur through shared memory. Otherwise, if network has to be used, there is a risk of performance penalty because of the transfer. The simplest approach for transferring data in case of online display is to have data within the same process as the display code.

2.4.4 Current model for graphical interfaces

At ESRF, graphical interfaces on beamlines are currently being created using the Bliss Framework, an in-house developed tool that helps assembling dedicated beamline graphical components (“bricks”) to generate GUI applications.

The Bliss Framework relies on the MVC (Model-View-Controller) paradigm, meaning that GUI components are separated from the underlying model. In the case of beamlines, the model is constituted by the different beamline components available through the middleware and the sequencer.

If the actual architecture would be applied to the new integrated beamline software platform, it would be represented like the diagram below.

Basically, a Control Objects Server is reading XML files describing Control Objects, with the description of the commands and channels needed from the beamline devices and some properties, then the Control Objects Server is producing Control Objects from the descriptions. A GUI application is made of bricks, which can be viewed as independent graphical components; those components are connected to Control Objects to access the unification layer and the sequencer. Some applications require a very fast access to middleware resources, for example for video display: in this case, this connection bypasses the Control Object Server and is directly made to the unification layer.
2.4.5 Evolution of the model

With the new abstractions introduced by the Integrated Beamline Software Platform, an evolution of the current model for graphical interfaces is possible, simplifying the whole design: Control Objects could be moved to the sequencer, as it would contain high-level data acquisition and control objects, coming from the unification layer (see 2.2.4). A network link would be suppressed, which is always good. We can imagine to have other objects of the same kind that could completely cover what Control Objects are doing in our existing framework for graphical interfaces.
3. Conclusion

In this document, the major design issues for building an integrated beamline software platform have been studied.
Having a look on the existing solution managing beamlines on different synchrotron, none of them includes all the components described in this document. On another hand, we did not find components, in these beamline control systems, which had not enough interest to be described in this paper.
Then, we can consider that an implementation providing a middleware, a unification layer, a sequencer and its CLI, a data manager, a Graphical user interface and a tool to configure all of them, should provide the necessary elements to control any synchrotron beamline.
This has a development cost. The change from one control system to another has a bigger one because each hardware device and each sequence have to migrate.
ESRF has the opportunity to invest in this direction through its “upgrade program”. We hope this opportunity could help other institutes.