A SEQUENCER FOR THE LHC ERA

V. Baggiolini, R. Alemany-Fernandez, R. Gorbonosov,
D. Khasbulatov, M. Lamont, CERN, Geneva, Switzerland

Abstract

The Sequencer is a high level software application that helps operators and physicists to commission and control the LHC. It is an important operational tool for the LHC and a core part of the control system that interacts with all LHC sub-systems. This paper describes the architecture and design of the sequencer and illustrates some innovative parts of the implementation, based on modern Java technology.

ARCHITECTURE AND DESIGN

The sequencer tool is conceptually divided into two parts: the sequencer, i.e. the software system capable of running sequences, and the sequences themselves.

Architecture and Design of the Sequencer

The sequencer follows the same architecture and technology as most of the accelerator controls software: a 3-tier architecture implemented in Java using the Spring Framework [1]. This architecture is shown in Figure 1.

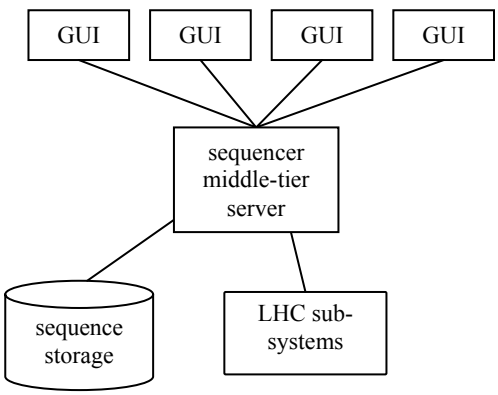

Figure 1: Sequencer 3-Tier Architecture.

The sequencer middle-tier server contains the core functionality, such as sequence execution. It runs on a Linux server in the computer centre. The resource tier is composed of sequence storage and the controls for the different LHC sub-systems. The client tier consists of Graphical User Interfaces (GUIs), which run on Linux or Windows consoles in the CERN Control Centre (CCC). Many GUIs may connect to the same middle-tier server.

Figure 2 gives an insight into the middle-tier sequencer server. Boxes with rounded corners represent software modules and rectangular boxes with a folded corner represent sequences. Items inside the dotted line are running as part of the sequencer process itself.

The sequencer server works as follows. In the beginning all sequences are stored in long-term storage (1) which can be a database or an SVN source repository, and typically has some versioning functionality. The storage manager module retrieves the operational sequences and passes the contents to the sequence composer module. The composer module creates a standard representation of the sequence (explained below) and stores it in file format on the local hard disk of the sequencer server (2). To execute a sequence, the sequence loader reads it from the hard disk, transforms it to the in-memory representation appropriate for the sequence executor (3). The sequence executor runs the sequence by executing the tasks inside. The tasks are simply Java methods implemented in a set of task libraries, which are deployed together with the core modules of the sequencer server. The task libraries interact with various accelerator controls services to ultimately control and supervise the LHC sub-systems.

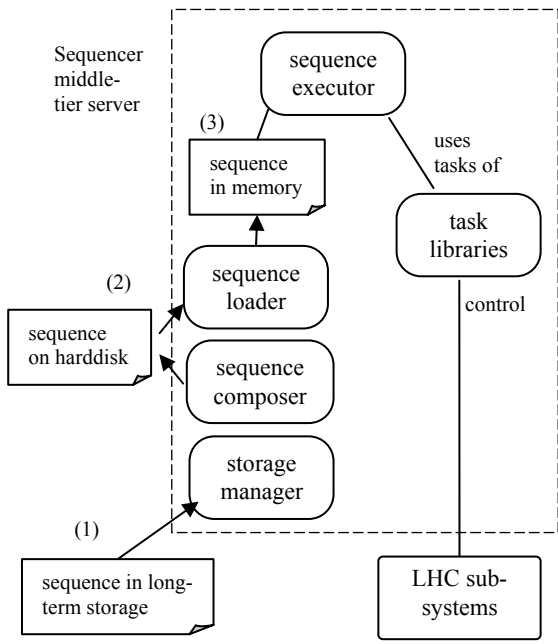

Figure 2: Sequencer middle-tier server.

Sequences and Tasks

In the simplest case, a sequence consists of a list of tasks which are executed one after the other. A more advanced sequence may contain both tasks and sub-sequences; there can be blocks of tasks to be executed in parallel; and there can be flow control statements such as if/else, loops and try/catch. In some cases variables are used to store data and state.

The sequence itself (in its version stored on hard disk, point (2) in Figure 2) is simply a Java source file. This Java-centric representation was chosen for several reasons. Java is an obvious way to express method calls

(= task invocations) and all the constructs mentioned above. The coherence of a sequence can be ensured to a great extent by simply compiling the file. Java is the language used in most accelerator controls applications, which facilitates sequence development and does not force developers to learn new representations and tools. Finally, Java is an executable format. Any sequence in the Java format can be executed directly from within the development tools, without need to deploy it on the sequencer server. This makes preliminary debugging of sequences very efficient.

Even though task libraries are deployed together with the core modules of the sequencer server, the two are well separated. They are developed by different people and contained in different libraries (jar files). A small team of software engineers is responsible for the sequencer core, whereas several domain experts from hardware commissioning, from operations and from equipment groups write sequences. Task libraries are developed partly by the sequencer core team and partly by the domain experts.

SEQUENCE EXECUTION

Running a sequence is a bit like running a program in an enhanced debugger. The tasks of the sequence are displayed in the GUI, and the user can set breakpoints to stop execution, and "skip points" to skip certain tasks. Execution is either automatic or manual. In automatic mode, the sequence normally just runs to the end. It stops if it hits a breakpoint, or if an error occurs in a task. In manual mode, the user can step through a sequence task by task; s/he can also execute tasks out of order, and jump back and forth inside the sequence. In either mode, the GUI is updated to show the execution flow, and to display the results of the tasks as they are executed.

LHC Hardware Commissioning (HWC) and LHC beam commissioning and operations have different requirements. This lead to two different implementations of the execution core as explained in the next sections.

Requirements for Execution

The purpose of HWC is to test all the 1600 magnetic circuits and the associated protection systems. Between 3 and 30 tests are executed for each circuit, which sums up to a total of almost 10000 tests for the whole LHC. Each test is implemented as a sequence. The HWC sequencer automates execution of these test campaigns: several HWC teams can work in parallel, and each team can execute up to a hundred sequences (= tests) simultaneously. In terms of sequencer functionality, HWC sequences run in automatic mode. Breakpoints and skipping are used only in special circumstances, and jumping is not needed. HWC sequences contain loops, if/else statements, and try/catch blocks, and they have variables to store input data and results. Typically, many sequences are executed in parallel, but there is limited parallelism inside one sequence – tasks are mostly executed sequentially.

For beam operations, the sequencer helps the operator drive the LHC. Safety and reliability comes first, and flexibility (e.g. skipping tasks) is needed only in special cases, and must be carefully controlled. Sequences are executed in automatic mode, with breakpoints but without jumping and skipping. On the other hand, Beam commissioning and machine development (MD) needs more flexibility to deal with experimental situations, such as systems that are not fully operational or new functionality just being tested. In general, sequences are executed in manual mode before they are run automatically.

LHC sequences are more linear than HWC sequences, they do not contain any loops, if/else statements and try/catch blocks, and do not use variables. Parallelism requirements are opposite to HWC: only one sequence is typically executed at any time, but several of its tasks must be executed in parallel, to act on different LHC sub-systems simultaneously. Error behaviour must be configurable: if a task fails, the sequencer can be configured to either ignore the error, to stop sequence execution or to continue with a so-called "recovery sequence". In LHC beam commissioning and MD, the sequences are similar to those of beam operations. The additional flexibility is achieved by supporting skipping and jumping in addition to breakpoints.

The execution requirements are summarized in Table 1.

Table 1: Execution Requirements

	HWC	Beam comm. and OP
Execution	Run, stop, break, skip	Run, stop, break, skip, jump
Error handling	Fail and stop on error	ignore, stop, run recovery sequence
State	In variables	No variables
Control statements	Loops, if/else, try/catch	N/A
Typical parallelism	Sequences in parallel	Tasks running in parallel
Typical mode	Run-through automatically	Both "debug" and run-through

HWC Executor Implementation

There are two implementations of the executor module, a script-based one for HWC, and a reflection-based one for Beam operations and commissioning. The script-based executor is implemented using the Pnuts scripting language and its interpreter [2]. Pnuts is similar to Java, and Java methods can be called directly from Pnuts scripts. It has the features needed by the HWC sequencer such as variables and flow control statements. The interpreter supports debugging, with breakpoints and stepping functionality. However, it does not support skipping of tasks. We implemented skipping with aspect-oriented programming (AOP) using AspectJ [3]. We declared an advice around the central method invocation code in the Pnuts interpreter. To skip a task, the advice simply continues without invoking the advised method. With AspectJ it is possible to integrate ("weave") this

AOP code into the Pnuts jar file without any need to modify or even re-compile the Pnuts sources.

HWC Loader Module

In our first implementation, the sequences were written in Pnuts, but this was tedious and error prone. People were not familiar with the Pnuts language and had no good editing tools for it. Even worse, as no compilation was done, many mistakes in the Pnuts sequences were detected only at run-time, when the sequence was executed. Therefore, we decided to use Java as central format of all sequences, as described above.

The HWC Loader module is responsible of transforming a Java sequence into a Pnuts script. The Java Compiler Compiler (JavaCC) [5] was used for this. The Java source is first parsed into an Abstract Syntax Tree (AST). A series of checks are done on the AST, to make sure the sequence conforms to some rules and restrictions, e.g. that it only uses for loops, if/else and try/catch and no other flow control constructs. Then a modified "pretty printer" component for Java source code is used to transform the AST into Pnuts source code.

This AST-based approach proved to be very powerful to fulfil also other requirements involving source code transformation. For instance, all HWC sequences begin with the same initialization code and end with the same bookkeeping code. Instead of replicating the same code in all sequences, we decided to use a better approach. It is based on the Template Method design pattern [5] plus source code transformation, as illustrated in the code example below. All HWC sequences inherit from the same abstract Java (`GenericHwcSequence`) class which contains the initialization and bookkeeping code, and calls an abstract method `specificPart()` in between. This abstract method is implemented in the derived sequence class (`HwcTest1`) and contains the HWC-test-specific tasks. The HWC loader is capable of "flattening" the hierarchy of the two classes (the abstract parent class with initialization code and the concrete implementation class with the sequence iself) into one flattened class with the full code of the sequence (`FlattenedHwcTest1`).

```
abstract class GenericHwcSequence {
   void exec() {
      initializeCircuits();
      specificPart();
      bookKeeping();
   }
}

class HwcTest1 extends HwcSequence {
    void specificPart() {
        task1();
        task2();
        ....
    }
}

class FlattenedHwcTest1 {
   void exec() {
      initializeCircuits();
      task1();
      task2();
      ....
      bookKeeping();
   }
}
```

Executor for Beam Commissioning and OP

To fulfil the requirements for beam commissioning, we implemented an executor from scratch in Java. It uses the Java reflection package to execute the tasks (Java methods), and Java's concurrency package to implement parallelism. Breakpoints, jumping and skipping were relatively easy to develop. The requirements did not include variables and control constructs, which according to preliminary studies would have been more complicated to implement. The on-error behaviour was realized as follows. For each task, the sequence developer can specify what should happen: if the execution should stop, if it should simply ignore the error and continue execution, or if a recovery sequence should be executed. By default, the execution stops.

A special feature of the beam commissioning sequencer is execution of a sequence when an external event (e.g. a beam dump) happens. From a conceptual point of view, this resembles interrupt handlers in assembly language. It was implemented keeping this concept in mind: there is a pre-defined set of external events (the events sent by the LHC timing system). A special task exists to associate (register) a sequence with an external event, and an opposite task to undo this association again.

CONCLUSIONS AND OUTLOOK

The sequencer has been used operationally for more than two years in HWC, and over 18 months for LHC beam commissioning. In addition, it is in operation also in the SPS accelerator and the Compact Linear Collider (CLIC). It has become a vital tool for operations: over 60 sequences and sub-sequences exist for HWC, 450 for LHC beam commissioning, 25 for SPS and a dozen for CLIC. Many improvements were made during this period, and a lot of feedback from operations was integrated. A new GUI was developed this year for the LHC operations, by one of the authors of this paper. Unlike the existing, general-purpose GUI, the new one is tailored to the needs of LHC beam commissioning and operations. A new area of work is to implement a good technical solution to conjugate the flexibility needed for beam commissioning with the rigor indispensable in LHC operations.

REFERENCES

[1] www.springframework.org
[2] www.pnuts.org
[3] en.wikipedia.org/wiki/AspectJ
[4] https://javacc.dev.java.net/
[5] http://en.wikipedia.org/wiki/Template_method_pattern