STATE MACHINE FRAMEWORK AND ITS USE FOR DRIVING LHC OPERATIONAL STATES

M. Misiowiec, V. Baggiolini, M. Solfaroli Camillocci, CERN, Geneva, Switzerland

Abstract

The LHC follows a complex operational cycle with 12 major phases that include equipment tests, preparation, beam injection, ramping and squeezing, finally followed by the physics phase. This cycle is modelled and enforced with a state machine, whereby each operational phase is represented by a state. On each transition, before entering the next state, a series of conditions is verified to make sure the LHC is ready to move on. The State Machine framework was developed to cater for building independent or embedded state machines. They safely drive between the states executing tasks bound to transitions and broadcast related information to interested parties. The framework encourages users to program their own actions. Simple configuration management allows the operators to define and maintain complex models themselves. An emphasis was also put on easy interaction with the remote state machine instances through standard communication protocols. On top of its core functionality, the framework offers a transparent integration with other crucial tools used to operate LHC, such as the LHC Sequencer. LHC Operational States has been in production for a year and was seamlessly adopted by the operators. Further extensions to the framework and its application in operations are under way.

RUNNING LHC

Introduction

A ***state machine*** is a model used in computing. It is composed of states connected by transitions and associated with actions. State machine holds a status where an active state is its most essential attribute.

LHC

During the first period of LHC run it was decided to introduce a state machine with the scope of increasing the operational performance and ensuring control of the LHC operations.

The *LHC Operational States* state machine has a threefold aim of:

- minimizing the risk of errors and mistakes,
- lowering human risk factor,
- increasing machine efficiency by reducing time losses and creating more rigid structure for the LHC nominal cycle.

Meeting such goals has a natural consequence in form of diminished flexibility for the operations, the trade-off all have agreed upon. Moreover, although *Operational States* helps maintaining a high level of equipment safety, it is not a safety but an operational tool.

In order to have a reliable convenient tool, able to cope with the large number of different situations faced by the LHC operations, the state machine had to be carefully studied and prepared. Twelve states were identified, as shown in Table 1. Each LHC machine state is a condition in which the LHC can be found during its lifecycle. In any state, the execution of certain tasks has been restricted (i.e. while the state machine is in "preparation" and there is no beam it is not possible to declare "stable beams" mode) to ensure that no wrong action is performed in critical operational phases.

Table 1: Description of LHC Operational States

state	description
PREPARATION	LHC is prepared to inject beam. All devices are prepared, magnets set to injection current.
INJECTION PROBE	The probe beam is injected and the beam parameters (tune, chromaticity, etc) are checked before injecting nominal beam.
FILLING	Nominal bunches are injected filling the machine.
PREPARE RAMP	Orbit correction and device preparation for ramp is performed.
RAMP	Energy ramp from injection (450 GeV) to nominal.
SQUEEZE	Squeeze beam to nominal value.
ADJUST	Beams are set in collision.
STABLE BEAMS	Experiments can move in their reading devices.
BEAM DUMP	A programmable or unplanned beam dump has happened.
TEST	Used during Stop to test software changes. No limitations applied.
STOP	LHC is not operational (cryo stop, device repair, technical stop).
INJECT DUMP	Special test with operational configuration: beams are injected then dumped immediately after.

STATE MACHINES

Basic Terms

State machine diagram, a ***layout***, is composed of ***states*** and ***transitions***. Each transition is directed, linking precisely two states, hence the whole diagram constitutes a directed cyclic graph. States and transitions are uniquely named, as well as the state machine itself, the ***instance***. Exactly one state is qualified ***initial***, while the others can be assumed ***final***, from which nothing else can be

reached. The current state of the instance is marked ***active***. In principle, there may be only one initial and one active state, being the same at the start of the instance lifetime.

Moving from state A to state B is feasible only if A is *active* and there exists a *transition* from A to B. Moreover, a set of ***actions*** can be assigned to the layout and required to be successfully performed while moving between states. Performing an *action* may relate to any additional operation executed by an *instance* and yield a success or failure. If an action is identified ***fail-forward***, failure is considered neutral to the *move*, otherwise the new *active* state must be decided.

An *action* can be placed in one of three ***locations*** with regard to a single transition or state:

Table 2: Location of Actions

(1)	***on exit***	while *leaving* state A; all transitions from A
(2)	***on transition***	while *moving* from A to B; transition A->B
(3)	***on entry***	while *entering* state B; all transition towards B

Assigning action locations helps to organize the layout and order the actions on the transitions. The actions are performed in natural order of sets: (1), (2) and (3). Those sets can be further sorted if necessary, so that it is known prior to the move in which order actions are called.

Actions are logically divided into two types: ***tasks*** and ***conditions***. *Task* is an operation which holds a significant impact on the environment, changing the state of external entities. An example would be an archiver, logging information to the database, or a publisher which broadcasts messages upon calling. Performing the *task* may end up in failure, although it should not be considered harmful to the *move*. Conversely, *condition* has little impact on the environment, yet it checks its particular quality. Calling the condition results in a boolean value, but should bring no side effects. Failing condition check should inhibit the *move*.

Concepts Omitted

Due to the complexity of the notion, *State Machine Framework* does not support any action rollback mechanism. Each action as a separate entity, unrelated to others, should clean itself up in case of its failure.

Some other reoccurring state machine concepts have proven superfluous in our environment. Hence lack of support for sub-states, nested states, parallel transitions or synchronization blocks that can be found on the state machine library market [2].

LHC OPERATIONAL STATES

State Machine Instance

LHC Operational States is a production instance of *State Machine Framework* based on the specific LHC configuration and dedicated action implementations. The instance follows classic 3-tier architecture with a clear separation of concerns. The middle-tier server runs on a Linux server in CERN computing centre. Configured with twelve states, it manages walking through over twenty transitions. Each transition is equipped with a series of checks that need to be satisfied in order to successfully move forward. They are triggered by a dedicated state machine action, but performed on a remote machine as Sequencer tasks. The results of the checks are communicated back to the core of *Operational States* server where the decisions about the new active state are taken and published further to interested parties. Those being all the listening clients, the database storage and local logging files. The primary clients are Sequencer GUIs, State Machine GUIs and any application using SM client API, although only the first two are authorized to perform state changes.

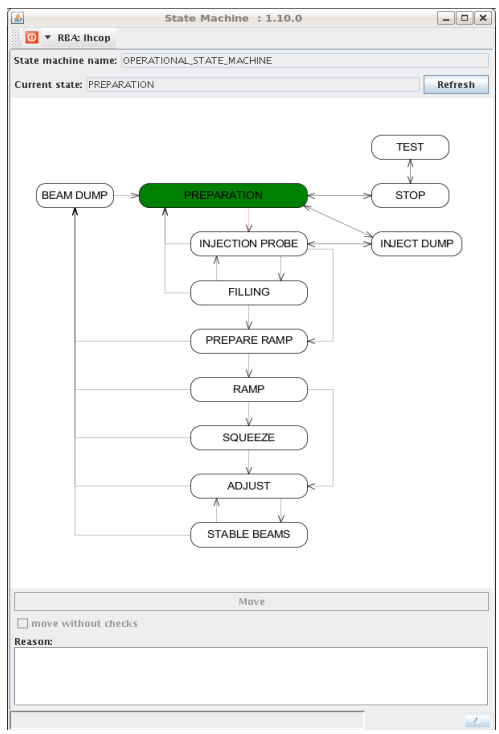

Figure 1: LHC Operational States diagram in SM GUI.

Sequencer

LHC Sequencer [3] is a crucial operators tool. It allows for executing preconfigured, ordered series of operations, i.e. *sequences*. They exemplify the steps taken in order to move the machine along its operational lifecycle. Thus, Sequencer naturally employs the state machine concepts, becoming both its client and executor. Each transition of *Operational States* is requested from a certain Sequencer task. *Operational States* server reacts to the request, among others, by executing its action which in turn launches another Sequencer task.

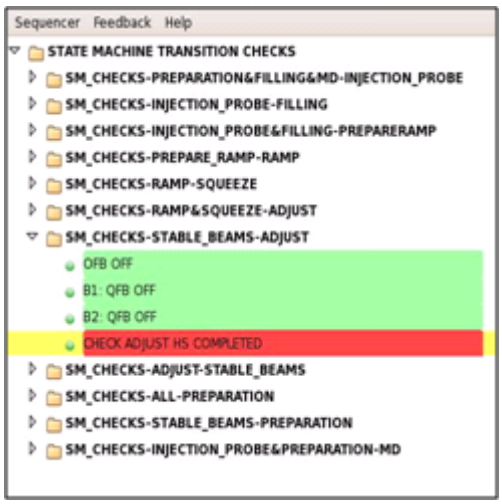

Figure 2: Sequencer checks for transition in SM Checklist.

Those tasks (checks) examine the actual context of the LHC machine, e.g. the external subsystems must meet certain conditions to safely allow to progress to the next stage. In case the check is successful, new active state is decided and returned back to the Sequencer. Otherwise, the operators must take necessary measures so that the conditions are met while the request is repeated.

SM GUI and SM Checklist

To help the operators in visualizing the states diagram two graphical applications were developed. *State Machine GUI* presents the layout indicating the active state, the undergoing transition and the messages issued from the Operational States server. It lets test the transitions outside the Sequencer tasks to ease up their preparation. It also permits to skip certain checks. Such tool proves indispensable in the early development or debugging stages. *State Machine Checklist* is an easy means of presenting the Sequencer checks associated with the transitions. Both tools are extensions to the LHC Sequencer, used on a daily basis in CERN Control Centre.

STATE MACHINE FRAMEWORK

Library

State Machine Framework (SMF) as most of CERN accelerator controls software is created in Java using the Spring framework [4]. Communication between client and server layers of the classic 3-tier architecture is realized with RMI and JMS protocols, employing high level concepts and infrastructure of JAPC [5]. Applications connect to the server through a dedicated client API that allows for performing state changes in both, synchronous and asynchronous ways. In addition to sending the response to the transition requests, server publishes the results through asynchronous channels of JMS. Any client application interested in following state changes, including failed attempts, can become a listener.

SM instance is identified by a unique name that is announced along with the addressing details to the Directory Service [6]. Client library looks up the Directory Service to discover that information. Using an intermediate publishing service greatly improves the flexibility of the SM instance deployment.

By default, SM instance stores its lifecycle data, including state changes, in several resources. Database archive and local log files are the major ones. The current status of the state machine is always preserved between the restarts of the server which provides for smooth upgrades.

Configuration

Configuration of a SM instance can be supplied either through an XML file or a database. In each case, template schema is provided to verify the syntax. Seemingly easier way is to define the layout as an XML document, more readable and simpler to edit. That was also the choice for LHC Operational States, letting operators manage the configuration fully on their own. It has proven an excellent opportunity for stimulating the ultimate users to actively participate in creation and maintenance of the software.

Deployment

SMF source code is split into several packages. Client and server libraries are clearly divided, while actions are shipped in another package. It lets action developers, often outside the state machine team, to contribute their implementations without interfering with main course of *SMF* development. It also alleviates the testing process.

Actions package is combined at the level of deployment of the state machine instance. Lacking an action class results in calling a default one, hence does not break a running server. SM instances are always bundled into a separate product, loosely coupled with the framework packages.

Actions Interface

Actions are generally developed to the dedicated API outside the *SMF* core. They are placed in the instance layout simply through editing its XML/DB configuration. There it is decided if an action falls into a task or a condition logical category. In the action main call, *perform()* method, a variety of information is provided by the SM server, including remote client token and a full SM context from the time of the call. In case of a problem it can throw an exception indicating a non-recoverable error. Otherwise, the action is considered to have completed successfully.

Request Lifecycle

SM instance accepts client calls (RMI) and based on the supplied arguments decides if a state change is possible at the moment. The feasibility of the move is also verified against the SM layout before the call is executed, at the client level. Each request holds additional data, a *payload* that identifies the client application. It can also contain a user provided map of properties. Payload is made available to each action performed on the requested transition, thus client data can be processed at that stage.

Once verified, the request is translated into a series of conditions and tasks that are performed on the transition. The result is then returned to the client, published asynchronously and persisted. Exception or warning messages thrown by the actions are disseminated too. While a single request is handled, all the other incoming requests are put on hold by the server.

Concurrency

One of the key benefits of *SMF* is the soundness of the concurrent model employed. State machines open to a parallel use of many clients are prone to a variety of errors, typically hard to detect at the testing stage. Hence a diligent effort was taken to design and implement a flawless concurrency engine based on Java monitors. As clients are rightly handled also in case of heightened number of requests, the *SMF* proves an accurate solution for fast-changing environments. Over a year of intensive use of *SMF* has confirmed a bug-free environment in that respect too.

Safety

Unlimited access to the SM instance could pose a serious threat to the LHC environment, even if the Sequencer checks have no impact on the probed environment. Role based access model *(RBAC)* was employed to handle the issue, a common solution in CERN controls systems [7]. Each client request's payload is equipped with RBAC token that holds verified credentials of the user. Server decides upon its contents whether the request can be authorized. In case of *LHC Operational States* the access to the SM instance is narrowed solely to the LHC operators.

Embedded Mode

SMF supports not only 3-tier architecture with its separation of clients, server and resources, but also an embedded use within another application. The instance is then accommodated into the scope of controlling application using Spring context, whereby all the client calls are executed locally, between Java threads. Embedded mode is achieved by an appropriate configuration of the instance. No remote calls or auxiliary resources, e.g. Directory Service, are required.

Considering its simplicity and concurrency soundness, such a solution is advised in case a state machine is needed within a more complex application.

CONCLUSIONS

State Machine Framework and its first instance have been operationally used for almost a year now. Over the period they have become an integral constituent of LHC controls software architecture, running impeccably throughout. LHC Operational States instance has reliably intertwined with the existing infrastructure, whereby the LHC Sequencer plays the essential part. The collaboration with other software teams helped to improve the functionality *SMF* offers, while the feedback of the LHC operators have been continuously integrated. Open architecture promoted their active participation in the development of the library. It has also raised several subjects that are being considered for new requirements, thus more work is anticipated in the near future.

State Machine Framework is a general purpose library aimed at both standalone or embedded use wherever state machine concepts need putting in place. Minimal dependency on the accelerator controls infrastructure makes it a comfortable choice for any project seeking a similar tool.

REFERENCES

[1] R. Alemany-Fernandez and M. Lamont and S. Page, “LHC Modes”, CERN EDMS, LHC-OP-ES-0005, 2007
[2] W3C SCXML, http:///www.w3.org/TR/scxml
[3] V. Baggiolini and R. Alemany-Fernandez and R. Gorbonosov and D. Khasbulatov and M. Lamont, “A Sequencer for LHC Era”, Proceedings of ICALEPCS’2009, Kobe, Japan.
[4] Spring Framework, http://springframework.org
[5] V. Baggiolini et al, “JAPC - the Java API for Parameter Control”, Proceedings of ICALEPCS’2005, Geneva, Switzerland
[6] M. Sobczak, “Specification for the Middleware Directory/Name Server”, CERN, 2009
[7] S. Gysin, “Role-Based Access Control for the Accelerator Control System at CERN”, Proceedings of ICALEPCS’2007, Knoxville, USA.