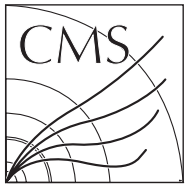
Available on CMS information server

CMS CR -2009/095



The Compact Muon Solenoid Experiment

Conference Report

Mailing address: CMS CERN, CH-1211 GENEVA 23, Switzerland

12 May 2009 (v2, 14 May 2009)

Job Life Cycle Management Libraries for CMS Workflow Management Projects

Frank van Lingen, Dave Evans, Simon Metson, Stuart Wakefield, Rick Wilkinson, James Jackson, Daniele Spiga, Stephen Foulkes, Anzar Afaq, Valentin Kuznetsov, Eric Vaandering, Seangchan Ryu, Fabio Farina, Giuseppe Codispoti, Mattia Cinquilli.

Abstract

Scientific analysis and simulation requires the processing and generation of millions of data samples. These processing and generation tasks are often comprised of multiple smaller tasks divided over multiple (computing) sites. This paper discusses the Compact Muon Solenoid (CMS) workflow infrastructure, and specifically the Python based workflow library which is used for so called task life-cycle management. The CMS workflow infrastructure consists of three layers: high level specification of the various tasks based on input/output datasets, life cycle management of task instances derived from the high level specification and execution management. The workflow library is the result of a convergence of three CMS subprojects that respectively deal with scientific analysis, simulation and real time data aggregation from the experiment.

Presented at *CHEP09: International Conference On Computing In High Energy Physics And Nuclear Physics, 21-27 Mar 2009, Prague (Czech Republic),21-27 Mar 2009,Prague,Czech Republic,15/05/2009*

Job Life Cycle Management Libraries for CMS Workflow Management Projects

Frank van Lingen[1], Dave Evans[2], Simon Metson[3], Stuart Wakefield[4], Rick Wilkinson[1], James Jackson[9], Daniele Spiga[10], Stephen Foulkes[2], Anzar Afaq[2], Valentin Kuznetsov[6], Eric Vaandering[2], Seangchan Ryu[2], Fabio Farina[7], Giuseppe Codispoti[8], Mattia Cinquilli[5].

[1]California Institute of Technology

[2]Fermi National Laboratory

[3]Bristol University

[4]Imperial College London

[5]University of Perugia and INFN Perugia

[6]Cornell University

[7]INFN Milan

[8]University of Bologna and INFN

[9]Bristol University and Rutherford Appleton Laboratory

[10]CERN

E-mail: fvlingen@caltech.edu, evansde@fnal.gov, s.metson@bristol.ac.uk, stuart.wakefield@imperial.ac.uk, rickw@clatech.edu, james.jackson@cern.ch, daniele.spiga@cern.ch, sfoulkes@fnal.gov, anzar@fnal.gov vk@mail.lns.cornell.edu, ewv@fnal.gov, sryu@fnal.gov, fabio.farina@cern.ch, giuseppe.codispoti@bo.infn.it, mattia.cinquilli@pg.infn.it

Abstract. Scientific analysis and simulation requires the processing and generation of millions of data samples. These processing and generation tasks are often comprised of multiple smaller tasks divided over multiple (computing) sites. This paper discusses the Compact Muon Solenoid (CMS) workflow infrastructure, and specifically the Python based workflow library which is used for so called task lifecycle management. The CMS workflow infrastructure consists of three layers: high level specification of the various tasks based on input/output datasets, life cycle management of task instances derived from the high level specification and execution management. The workflow library is the result of a convergence of three CMS subprojects that respectively deal with scientific analysis, simulation and real time data aggregation from the experiment.

1. Introduction

Scientific analysis and simulation requires the processing and generation of millions of data samples (also called events). These processing tasks (also called jobs) are often comprised of multiple smaller tasks and are divided over multiple (computing) sites. The Compact Muon Solenoid (CMS) experiment [1] is one of the two general purpose physics experiments at the European Laboratory for Particle Physics (CERN) [2] which is due to start operation in 2009. The scale of the collaboration and the expected data size (Petabytes per year) present several challenges with respect to data processing, storage and transfer. One such challenge is the management of a large number (more than several 1000s on a daily basis) of so called processing jobs, which represent a typical single (user) analysis or simulation request. At the time of writing the current simulation system generates approximately 100 million events per month from a limited number of requests.

Since processing requests are larger than a single site can handle, the requests are split across multiple distributed sites. With the large number of jobs submitted, and the nature of the Grid fabric, errors and failures are inevitable and care is taken to handle as many of these in application logic as possible to minimise the impact of failures on the end user.

As individual jobs each produce part of the data products, data aggregation becomes important too. The scale of requests, data aggregation, failure and interaction with third party components led to the development of an infrastructure to manage the life cycle of the various tasks associated to a (physics) request.

This paper discusses the CMS workflow infrastructure, and specifically the Python based workflow library for task and job lifecycle management.

The library is based on the experience from three CMS subprojects that each deal with task life cycle management. This library consolidates the strength of these projects into a single generic library which is used by these three projects. The motivation for the work discussed in this paper is pragmatic: reduce the code base of current projects and increase maintainability (do more with less).

Section 2 gives an overview of the workflow infrastructure in the CMS experiment and introduces the terminology. Section 3 discusses the task lifecycle layer. Section 4 discusses the core of task lifecycle management and optimization strategies, while section 5 and 6 discuss related work and conclusions respectively.

Within the remainder of this paper the words task and job are used interchangeably as they have a one to one correspondence. Task usually refers to various administrative steps needed to process or generate data, while job refers to the particular step in a task where the actual generation takes place (execution on a compute node).

2. Overview

Within the CMS workflow infrastructure three layers can be identified: Request, task life cycle and execution

The request layer (also called the physics workflow) is a user driven specification of steps that need to be executed in order to create the desired data product. Depending on the request it typically contains a set of input datasets or a number of data samples that need to be generated and a sequence in which the steps are executed as well as some configuration data for the physics framework that is used. The specification is encoded in XML (eXtendable Markup Language) and represents a high level specification of what needs to be created.

Within the task life cycle layer the physics workflow is ‘sliced’ into pieces of work that are submitted to various sites. These pieces (called jobs or tasks) use the high level specification (physics workflow) as a template and apply it to non overlapping ranges of requested events or input datasets. For example a user can create a physics workflow which requests 1 million events to be generated. Within the lifecycle layer this request can be ‘sliced’ into 1000 jobs of 1000 events each. The processing parts of these jobs or tasks are executed independently from each other.

Each of these jobs produce relatively small data products which together represent the requested data set. As these data products are relatively small (= many small files) the final step is to merge these

small files into larger ones. Data is merged to simplify book keeping (meta data management) and perform more efficient transfers at a later stage (larger files generally give better transfer performance). File merging can also occur in intermediate steps if intermediate data products are generated. Within the task life cycle layer the physics workflow is transformed into the appropriate job specifications. This specification together with a job wrapper is submitted for execution.

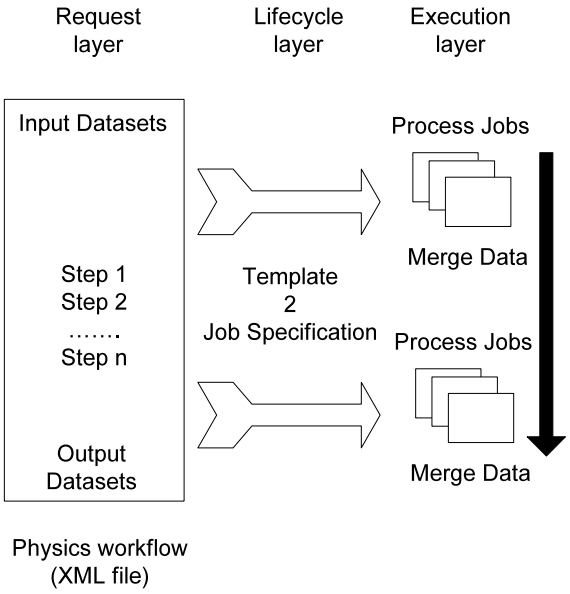


Figure 1. Workflow layers

Two of the three subprojects use ShREEK [3] in the execution layer. Shreek represents the execution layer and is a Python based runtime management extension to allow workflow control at runtime on a *single node*. ShREEK executes tasks and adds control points to the template specification. ShREEK is invoked at runtime and loads the job specification. The tasks found in the configuration are executed and ShREEK monitors and manipulates the tasks according to the configuration. Figure 1 shows the three CMS workflow layers and their relations. The observant reader will realize that the steps in the execution layer can be easily described in the MapReduce model [7].

3. Task Lifecycle

The task lifecycle describes the various stages of jobs or tasks from its creation to its destruction. Given the scale of the physics workflows it deals also with all kinds of exceptions or failures that can occur within a task lifecycle. For example: A third party service such as a metadata catalog might be offline. The applications representing task lifecycle will need to retry periodically to see if the catalog is back online. It is vital that data or data products are not lost as this can result in datasets that are biased in nature as certain data is missing. Reducing data loss is therefore a very important requirement. Data loss can occur if for example metadata is not properly registered in a meta data catalog.

To reduce system or application level failures in the task lifecycle, components responsible for the various lifecycle stages are represented as autonomous applications. The components communicate with each other through messages. Messages typically represent a state change in one or multiple tasks, but the system also supports control messages like ‘StopComponent’.

Components are not aware which other components subscribe to their messages and it is up to the developers of the components to define the message flow between groups of components. The

message paradigm enables deployment of components on different machines if necessary. Message types are similar to the channel concept used in other publish/subscribe infrastructures.

When a component is shutdown (failure, maintenance, etc..) other parts of the system keep operating. Besides minimizing system level failures the autonomous application approach enables developers to work on separate components provided they agree on the messages being sent. The approach also opens the possibility to easily integrate components developed in languages other than Python, or support different implementations of the same component if necessary.

The task lifecycle workflow library can be divided into three parts: a transient data management system that tracks relations between the high level physics workflows, their input and output data, a task management system tracking the breakdown of high level workflows into jobs executing on the various systems and a more functional part providing building blocks for developing components that represent the task management system.

The data management and task management systems are closely related. Workflows are associated to input data (via a "subscription") and, when needed the workflow is divided into appropriately sized tasks (so called job splitting). The algorithms used can depend on many parameters, such as size of the input data set, number of desired events per job or CMS internal parameters. The tasks are then submitted to the system and tracked.

The building blocks (Python modules) are an aggregation of functionality from the three subprojects with the aim of reducing development (and maintenance) overhead. The core modules are the component harness (base class for components), message service (provides communication between components), trigger (synchronizes components that work in parallel on a similar task), a persistent thread pool library, and a database abstraction layer (based on SQLAlchemy [4]) as the various projects either support Oracle, MySQL or both.

To facilitate migration of the current subprojects towards integrating this new library, a simple (Python) based scripting language has been developed. This scripting language enables the developers of the different projects to define the task lifecycle flow through the definition of handlers and synchronizers and generate the proper (stub) classes based on the task lifecycle workflow library.

```
synchronizer = {'ID' : 'JobPostProcess',\
                    'action' : 'PA.Core.Trigger.PrepareCleanup'}
handler      = {'messageIn'   : 'SubmitJob',\
                    'messageOut'  : 'TrackJoblJobSubmitFailed',\
                    'component'   : 'JobSubmitter',\
                    'threading'   : 'yes',\
                    'createSynchronizer' : 'JobPostProcess'}
handler      = {'messageIn'   : 'JobSucess',\
                    'component'   : 'MergeSensor',\
                    'synchronize'  : 'JobPostProcess',\
                    'threading'    : 'yes'}
```

Figure 2. Lifecycle workflow specification snippet.

A handler is a piece of code that takes as input a message (type) and its payload and performs certain actions on a particular task. For example: a submit job handler takes as input messages of type ‘SubmitJob’ with payload a job specification, and submits the job of a task to a particular site.

Handlers are grouped into components. For example a ‘job submit failure’ handler and ‘job process failure’ handler can be grouped into an autonomous ‘ErrorHandler’ component.

Synchronizers synchronize work that is done within different components on the same job/task in parallel. For example: when a task is finished with its processing step, processing results (data products) need to be registered in the meta data catalog, the transfer system and merge sensor which

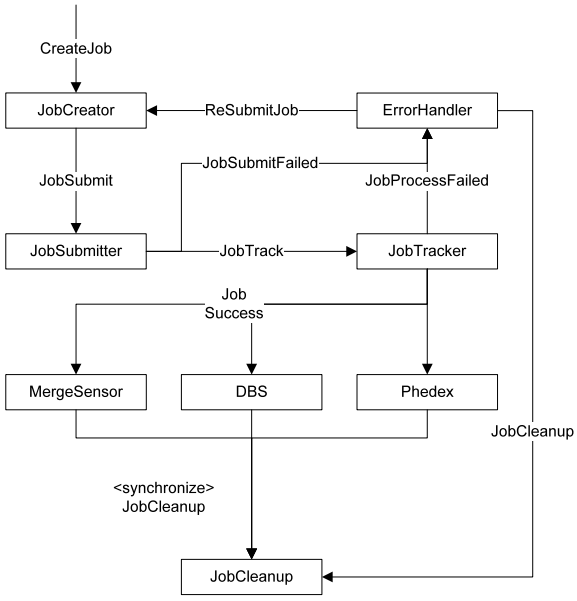


Figure 3. Example of a lifecycle workflow

are represented as separate components. These components perform these tasks in parallel. Only after all three components are finished processing the information, the task can be destroyed (or archived) by sending a message to a job cleanup component.

Figure 2 shows a snippet of a task lifecycle flow. The first handler is associated to the 'JobSubmitter' component and acts on messages of type 'SubmitJob'. Depending on the outcome, this handler either publishes messages of type 'TrackJob' or 'JobSubmitFailed'. The 'createSynchronizer' means that the component will instantiate the 'JobPostProcess' synchronizer for the particular task it is handling. The 'threading' field indicates that messages received by this handler are processed in a threaded fashion and not sequential one by one. The second handler is part of the 'MergeSensor' component and acts on messages of type 'JobSuccess'. When finished it will set the appropriate flag for the particular task that is associated to the incoming message in the 'JobPostProcess' synchronizer. Once all flags in this synchronizer are set for a particular task, the associated action to the synchronizer is executed for this task (for example publishing a 'JobCleanup' event).

Figure 3 shows part of the task lifecycle flow for the CMS simulation application. Components are represented as boxes and an arrow represents an outgoing message (arrow tail) from one component to an incoming message (arrow head) in another component. The DBS box [5] represents the CMS metadata catalog, while the Phedex box [6] represents the CMS transfer system. At the time of writing, the CMS simulation application (partly depicted in Figure 3) consisted of 20 components and approximately 80 message types. Figure 3 represents the control flow perspective of the application [13] while the tasks which contain the job specifications and execution reports (called framework job reports) represent the data perspective.

Once the specification is finalized the stub classes are generated based on the workflow lifecycle library and developers of the components can focus on the actual logic within the component handlers.

The message based autonomous component paradigm supported by a workflow scripting language is sufficiently flexible for specifying workflows needed by the CMS projects. The basic scripting language supports several well established workflow patterns [8]: sequence, parallel split, exclusive choice, synchronized merge and arbitrary cycles.

The functional part of the lifecycle library that supports workflow creation is completely domain (in our case physics) agnostic. An advantage of the high level specification language is that it naturally enables to compare life cycle workflows across the three sub projects and identify similarities. For example the analysis and simulation sub project share the same job submission component.

Two subprojects use the library to submit jobs to so called 'Grid' sites. The sites do not always use the same submission software and to be able to support multiple sites, components use the concept of plug-in. A plug-in typically represents an adapter for a third party application which are conceptually similar. For example: the job submission component has multiple plug-ins supporting submission to a variety of systems such as BossLite [22], Condor [16], Condor-G [18], LSF or GLite [17]. Plug-ins are currently defined on the component development level although the workflow language can be extended to support this.

Each component has its own configuration parameters (generated as a separate file). The project configuration file consists of an aggregation of these configuration options. It enables users of the system to select what components to activate and how to configure them.

4. Queuing, Caching, Bulk and Buffer

This section discusses some of the optimization techniques used within the lifecycle workflow systems to increase performance and reduce load of the system.

The core of the lifecycle workflow library is the message service which provides reliable delivery of messages. This reliability needs to be guaranteed even when components unexpectedly stop. Components that unexpectedly stop need to keep receiving messages. These messages need to be stored until the component restarts.

During peak operation of some of the components there are 100s of messages per second being sent. Components do not always have the resources to handle all incoming messages at once. To prevent loss of messages the message service provides a persistent buffer enabling components to retrieve and handle messages when they can. The persistent backend of the message service is currently based on MySQL. During early prototypes, scalability (database performance) became quickly an area of concern. Sometimes components crashed and this created a huge backlog of non processed messages or too many messages where being sent in overall.

As a result of early prototype performance issues, the current message service library is modeled as a cached queuing system. The persistence layer (in this case MySQL) caches many single inserts and delete statements related to message delivery in relatively small tables (500-1000 entries). If the threshold is reached, it transfers these data to the potentially large message buffer tables (in bulk). Furthermore the internal logic of components has been modified to perform actions in bulk. Bulk means that tasks are grouped in small clusters and instead of sending one message per task either the messages for these tasks are grouped and send in bulk or a so called bulk messages is send representing a state transformation of all tasks in a particular cluster. For example early prototypes submitted single jobs to sites as not all sites submitted so called 'bulk job submission'. Bulk job submission is now supported within the components and multiple jobs are submitted at once to a site, which means it is possible to send a single 'BulkSubmitJob' message representing multiple jobs. An extension of the message system is the persistent thread pool library. Components use threading to handle messages in parallel. To prevent losing messages the thread pool is modeled as a persistent queue similar to the message queue and also includes the caching of message inserts and deletes to prevent single record inserts in large database tables.

Finally the environment in which the task lifecycle workflows operate was examined. The early lifecycle workflow prototypes performed operations on single tasks, while consecutive improvements focused more on bulk operations where possible. Bulk operations reduce the overall load of the system but can still lead to peak loads if a large number of resources (e.g. job slots) become suddenly available. In a short amount of time many new tasks/jobs need to be generated. To prevent (or minimize) this peak task/job generation, buffering is used. The site resources our systems can use, are monitored and if our queue (in case we have a dedicated queue on a site) or if the shared queue drops below a certain threshold the lifecycle workflow system will start creating and submitting new jobs. These jobs are usually submitted in bulk but the size of these so called bulk collections are limited and

the systems opt for submitting perhaps multiple smaller collections spread out over time than one big bulk collection.

5. Workflow and Task Monitoring

The applications using this task lifecycle library deal with a large number of tasks/jobs on a daily basis. Given the large number of tasks that flow through these systems, it is important for the operators of these applications to monitor the system and being able to track any potential anomalies.

The task lifecycle library enables components to subscribe to messages. Each component has two queues. One queue is for 'normal' messages. These are messages that are (in most cases) associated to a task (e.g. CreateJob, SubmitJob). The second queue is for control messages. Control messages are messages that perform a certain action on the component rather than the task. Control messages have a higher priority than the other messages. For example if a queue of a component contains 1000 messages it needs to handle, a control messages like 'Stop' will bypass all these messages and be handled first. Typical control messages are 'Start', 'Stop', 'Logging.Debug', 'Logging.Info', etc... Developers of components can add their own messages as they see fit. 'LogState' is a control message that writes the current state of the component to its log file. It consists of a list of configuration parameters with their current values and messages this component subscribes to. When anomalies occur within a component publishing a LogState message is usual the first step the application operator takes to diagnose the problem. The next step of the operator is usually to publish a message <component id>:Logging.Debug, turning debugging on.

Control messages are augmented with a monitoring component (called HTTPFrontend). The monitoring component typically does not subscribe to any task related messages but serves information via a HTTP server from the database backends to users/operators. The monitoring component provides a birds' eye view of task states and information related to the task lifecycle library components. The aim of the monitoring component is to provide two views: overview and detailed view. The detailed view 'drills down' from the overview. For example the overview can show aggregate information on how many tasks are failing. The detailed view can retrieve specifics on these tasks.

HTTPFrontend and control messages show information that is specific to the workflow system or the tasks that it keeps track of. It does not track any state information of the actual jobs. The jobs themselves publish (inject) information into the MonALISA publish/subscribe infrastructure [21]. The CMS Dashboard (a portal that aggregates information from all jobs within the CMS experiment) subscribes to these messages and publishes information on the actual jobs.

Figure 4 shows the total aggregated number of simulated events during one week displayed in the dashboard. The total is approximately 200 million events. It should be noted that most of these events are so called FastSim events which is a simulation application that runs approximately 8-10 times faster than a regular simulation. If we scale this to 'normal' simulation events it translates to roughly 80-100 million events per month.

6. Related Work

There are many (open source) products available that support scientific workflow management and workflow specification languages. We discuss several of them in relation to the work described in this paper.

Many (if not all) workflow systems are related to the (colored) Petri Nets concept [14]. The added value of workflow systems lies in part in a high level representation of concepts and relations for particular domains. Indeed newYAWL [12] provides an explicit relationship between workflows and Petri Nets with extensions. YAWL [13] is based on Petri nets but extended with additional features to facilitate the modeling of complex workflows. Our (simple) workflow specification language targets developers and the specific domain within the CMS experiment. The aim of the library discussed in this paper, is to aid convergence of the several workflow related projects rather than designing new generic workflow languages.

Tools such as Kepler [11] and Taverna [15] provide functionality for (concurrent) modeling and specification of workflows. Both Kepler and Taverna provide a rich graphical user interface for workflow designers to specify and connect various workflow components. A difference between the task lifecycle library and Kepler and Taverna is that the number of workflows is limited in our case. The task lifecycle library does not cater to end users but developers and the workflows designed by developers are used 'as is' by users with perhaps minor configuration differences.

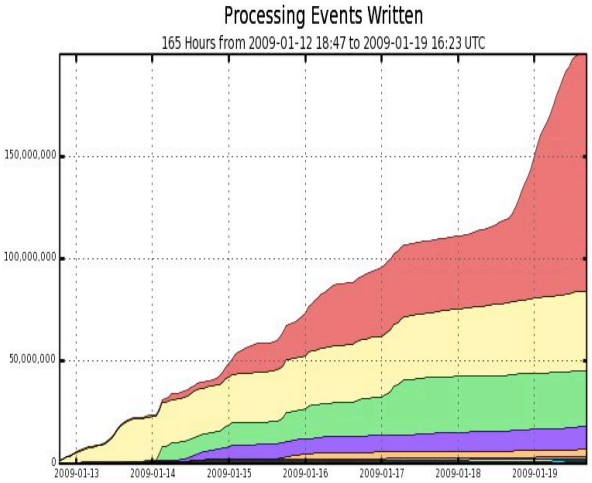


Figure 4. Events produced in a week

Most of the workflows within our projects are either, simple (sequences) and numerous (physics xml workflow) or workflows are complex, specific and limited (task lifecycle workflows). The task lifecycle components deal with CMS specific things such as physics job creation and interaction with third party (web services) such as the CMS meta data catalog (DBS). The main goal of the library discussed in this paper is to support lifecycle component developers in creating a coherent system, rather than supplying the end users with tools to create and manipulate workflows.

There is currently no need for a repository of components that we want to deploy for the task lifecycles, neither is there a need for graphical user interface for defining and specifying workflows.

The (physics) workflows processed as set of tasks in a lifecycle workflow follow a similar pattern: many jobs generate a collection of small data files from an input dataset after which an additional step merges the small datasets. This pattern can be described with the MapReduce model [7]. A difference is perhaps that the MapReduce model includes the master/worker model. Our current system is not master/slave oriented, in part because our tasks are running on shared resources where it is not always possible to 'reserve' nodes for long running tasks.

7. Conclusions

The work described in this paper is the result of convergence of three projects that deal with workflows in different yet also overlapping ways. As a consequence the convergence to a common layer needed to be non-intrusive and non-invasive so not to disrupt development and support of the current projects as the tools do significant work on a daily basis.

When the separate subprojects started a few years ago a generic workflow framework was not the highest priority. With the convergence of these projects (where possible) revisiting the workflow concept and requirements for future versions is becoming relevant.

The current system offers sufficient flexibility for developers to migrate subprojects into using the new framework and thus providing a coherent cross project code base. It enables developers of the different sub projects to work across project boundaries.

Future work includes replacing the communication layer of the message service to provide message service capability in a WAN environment with enhanced security. Candidate systems for this replacement include Twisted [9] and NaradaBrokering [10]. Such capability would enable users and operators to subscribe to messages within one or more of the lifecycle workflows regarding the status of their request or the state of the system and would add a more collaborative aspect to the system. Other work includes the introduction of pilot jobs (in the execution layer). Pilot jobs are long running jobs which acts as workers (following the master-worker model) and pull tasks in for execution [19], [20]. Augmenting the current system with the pilot job concept would make the process (from a data manipulation point of view) more similar to MapReduce.

While improving the system, the authors are keenly looking at available workflow solutions that support the requirements of the projects which the lifecycle library currently supports. Requirements include support for the Python programming language, providing reliable (high performance) messaging and providing a smooth transition of the current component logic into the new system. The library discussed in this paper is the first step of converging to a common (more maintainable) layer for workflow related functionality.

The workflow paradigm provided developers a much needed structure and enables them to focus on the core functionalities while 'default' features are provided by the lifecycle workflow library.

8. Acknowledgements

This work is partly supported by US Department of Energy grant DOE DE-FG02-06ER86271 and US National Science Foundation grant NSF PHY-0533280.

Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and don't necessarily reflect the views of the Department of Energy or NSF.

9. References

[1] CMS, http://cms.cern.ch/

[2] CERN, http://cern.ch

[3] P. Love, I. Betram, D. Evans, G. Graham,"Cros Experiment Workflow Management: The Runjob Project", in proceedings of Computing for High Energy Physics, Interlaken, Switzerland, Sept. 2004.

[4] SQLAlchemy, http://www.sqlalchemy.org/

[5] A. Afaq, A. Dolgert, Y. Guo, C. Jones, S. Kosyakov, V. Kuznetsov, L. Lueking, D. Riley, V. Sekhri, "The CMS Dataset Bookkeeping System", Journal of Physics, Conferences Series (119) 072001, 2008

[6] R. Egeland, T. Wildish, S. Metson, "Data Transfer Infrastructure for CMS Data Taking", To appear in Proceedings of Science, XII Advanced Computing and Analysis Techniques in Physics Research, Nov. 3-7, 2008

[7] J. Dean, S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.

[8] W. Aalst, A. Hofsted, B. Keipuszewski, A. Barros, "Workflow Patterns", In journal of Distributed and Parallel Databases 14(3), pages 5-51, July 2003

[9] A. Fettig, "Twisted Network Programming Essentials", ISBN 10: 0-596-10032-9, O'Reilly

[10] S. Pallickara, H. Bulut, G. Fox, "Fault-Tolerant Reliable Delivery of Messages in Distributed Publish/Subscribe Systems" Proceedings of the 4th IEEE International Conference on Autonomic Computing. Jacksonville, Florida. 2007.

[11] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. Lee, J. Tao, Y. Zhao,"Scientific Workflow Management and the Kepler System", Concurrency and Computation: Practice & Experience, 18(10), pp. 1039-1065, 2006. (see also: http://kepler-project.org)

[12] R. Russell, A. ter Hofstede,, W. v/d Aalst, "newYAWL: Specifying a Workflow Reference Language using Coloured Petri Nets", Eighth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, Aarhus, Denmark, October 2007.

[13] W.v/d Aalst, A.ter Hofstede, "YAWL: Yet Another Workflow Language", Information Systems, 30(4):245-275, 2005.

[14] K. Jensen, L.M. Kristensen, L. Wells, "Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems", International Journal on Software Tools for Technology Transfer, 9 (2007), Springer Verlag, 213-254.

[15] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. Pocock, A. Wipat, and P. Li, "Taverna: a tool for the composition and enactment of bioinformatics workflows.," Bioinformatics, vol. 20, iss. 17, pp. 3045-3054, 2004

[16] D. Thain, T. Tannenbaum, M. Livny, "Distributed Computing in Practice: The Condor Experience" Concurrency and Computation: Practice and Experience, Vol. 17, No. 2-4, pages 323-356, February-April, 2005

[17] GLite, http://glite.web.cern.ch/glite/

[18] J. Frey, T. Tannenbaum, I. Foster, M. Livny, S. Tuecke, "Condor-G: A Computation Management Agent for Multi-Institutional Grids", Journal of Cluster Computing volume 5, pages 237-246, 2002

[19] S. Paterson, A. Tsaregorodtsev, "DIRAC Optimized Workload Management", in proceedings of Computing in High Energy and Nuclear Physics, Victoria, Canada 2007

[20] P. Saiz, "AliEn - ALICE environment on the GRID", Nucl. Instrum. Meth., A502 (2003) 437-440.

[21] H. Newman, I. Legrand, P.Galvez, R. Voicu, C. Cirstoiu,"MonALISA: A Distributed Monitoring Service Architecture", in proceedings of Computing for High Energy Physics, La Jola, California, March 2003.

[22] D. Spiga, S. Lacaprara, M.Cinquilli, G. Codispoti, M. Corvo, A. Fanfani, A. Fanzago, F. Farina, C. Kavka, V. Miccio, and E. Vaandering "CRAB: an Application for Distributed Scientific Analysis in Grid Projects", in proceedings of the 2008 international conference on Grid computing – WorldComp, GCA 2008, Las Vegas 14-17 July 2008, pp.187-193, ISBN: 1-60132-068-X