LQCD Workflow Execution Framework: Models, Provenance and Fault-Tolerance

Luciano Piccoli[†‡], Abhishek Dubey[*], James N. Simone[†] and James B. Kowalkowlski[†]

[†]Fermi National Accelerator Laboratory, Batavia, IL, USA 60510
[*]Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN, USA 37235
[‡]Illinois Institute of Technology Chicago, IL, USA 60616

Abstract. Large computing clusters used for scientific processing suffer from systemic failures when operated over long continuous periods for executing workflows. Diagnosing job problems and faults leading to eventual failures in this complex environment is difficult, specifically when the success of an entire workflow might be affected by a single job failure. In this paper, we introduce a model-based, hierarchical, reliable execution framework that encompass workflow specification, data provenance, execution tracking and online monitoring of each workflow task, also referred to as participants. The sequence of participants is described in an abstract parameterized view, which is translated into a concrete data dependency based sequence of participants with defined arguments. As participants belonging to a workflow are mapped onto machines and executed, periodic and on-demand monitoring of vital health parameters on allocated nodes is enabled according to pre-specified rules. These rules specify conditions that must be true pre-execution, during execution and post-execution. Monitoring information for each participant is propagated upwards through the reflex and healing architecture, which consists of a hierarchical network of decentralized fault management entities, called reflex engines. They are instantiated as state machines or timed automatons that change state and initiate reflexive mitigation action(s) upon occurrence of certain faults. We describe how this cluster reliability framework is combined with the workflow execution framework using formal rules and actions specified within a structure of first order predicate logic that enables a dynamic management design that reduces manual administrative workload, and increases cluster-productivity.

1. Introduction

Unprecedented amounts of data are currently produced and analyzed by e-science experiments. The organization of this massive information is critical for its effective use in new discoveries. Lattice Quantum Chromodynamics (LQCD), the numerical study of QCD quantum field theory on a four-dimensional discrete lattice, generates considerable data that are typically processed at several facilities. Applications, software libraries, input data and workflow recipes are shared among collaborators worldwide.

Unlike many e-science experiments, which use Grid resources for harvesting capacity processing power, LQCD computations employ tightly-coupled parallel processing which requires computers with high-speed low-latency networks. Binary codes are fine tuned to exploit capabilities of underlying architectures. LQCD workflows effectively exploit the capacity of one or more parallel computers by running many independent computations at once.

LQCD workflows are categorized into two typical types: configuration generation and analysis campaign (see figure 1). The former is used for creating an *ensemble* through a sequence of Monte Carlo simulations. An ensemble is an ordered collection of gluon configurations sharing the same physics parameters (e.g. lattice spacing and masses). An analysis campaign iterates over an ensemble to compute physics quantities such as decay rates and particle masses. The processing for each configuration is independent of the other configurations. Many such analysis campaigns are conducted on each ensemble.

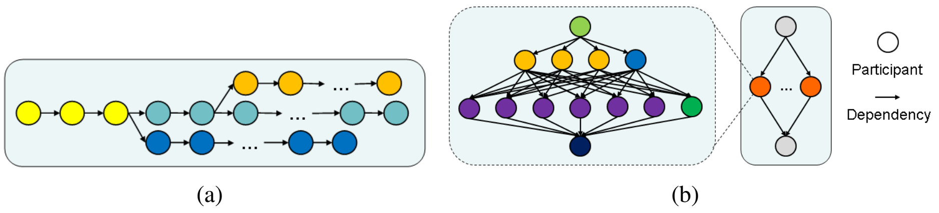

Figure 1. Example of typical LQCD workflows: (a) configuration generation workflow, (b) two-point analysis workflow. Circles represent workflow tasks (participants) and arrows indicate the dependencies between tasks.

Fault tolerance and provenance are two of the critical requirements for LQCD workflows [1]. Fault tolerance is critical for improving productivity and utilization of the dedicated hardware, while data provenance is necessary for recording and retrieving details about produced data, including the metadata and information about its origin.

In this paper, we propose a model-based, hierarchical, reliable execution framework for executing scientific workflows. During the workflow specification, each task (participant) has its execution conditions (invariant sets) and associated actions identified. At execution time, the workflow system interacts with the monitoring system in order to have conditions periodically verified and receive notifications if conditions are violated. While executing, information regarding provenance of produced products is recorded in a database that also contains information about the status of running workflow.

An overview of the proposed framework for LQCD workflows is described in section 2. Section 3 discusses the integration of the framework subsystems, followed by case studies using the current prototype in section 4. Finally, section 5 summarizes the proposed framework and discusses future work and extensions.

2. Framework overview

The proposed reliable workflow execution framework results from the integration of the workflow and reliability subsystems. The workflow subsystem provides workflow specification and execution along with the recording of data provenance, while the reliability subsystem performs execution tracking and online monitoring of individual workflow tasks.

Workflow tasks are plugins to this framework that we call participants. Participants are, in general, legacy applications wrapped by scripts that allow communication with the reliability and workflow subsystems. The framework supports parameterized abstract workflows instantiated with specific input parameters. The relationships among participants are described using types and parameters. The dependencies in concrete workflow are translated based on the types and parameter values.

2.1. Workflow specification

LQCD workflows are specified by scientists using parameterized abstract templates, which define abstract workflows. The templates applied to input parameters yield concrete workflows. In the typical configuration generation workflow (figure 1(a)) only a single participant is used. The same wrapped binary code represented by a participant is invoked with different parameters within a simple loop, defining a chain of configuration files. Figure 2(a) depicts an abstract configuration generation workflow, where the node in the middle represents the main participant. The extra nodes are defined for handling workflow inputs and outputs.

The notion of abstract workflow is better illustrated by two-point analysis campaigns (figure 2(b)). Analysis campaign workflows are a coordinated set of calculations aimed at determining a set of specific physics quantities. For example, predicting the mass and decay constant of a specific particle determined by computing ensemble averaged two-point functions. A typical campaign consists of taking an ensemble of vacuum gauge configurations and using them to create intermediate data products (e.g. quark propagators) and computing meson n-point functions for every configuration in the ensemble. An important feature of such a campaign is that the intermediate calculations done for each configuration are independent of those done for other configurations.

The final number of participant instances on the concrete two-point analysis campaign workflow depends on user specified parameters. The concrete workflow for a combination of three heavy quarks and six particle masses input parameters is shown in figure 1(b). The number of sub-workflows on the left side of figure 1(b) is equivalent to the number of configuration files within the ensemble used for the analysis.

The current prototype uses the Ruote BPM engine [2], but we plan to use the Pegasus workflow management system [3] to specify the abstract workflows and perform the mapping to concrete workflows in the future. The latter are defined as Condor DAGMan workflows. Although Ruote integrates well with the prototype due to the common use of the Ruby language, the project will benefit from the use of Pegasus because it is well integrated with Grid and cluster computing packages. Additionally Pegasus does fit better the LQCD workflow requirements [1] and it is also successfully used in other scientific areas such as earthquake simulation [4].

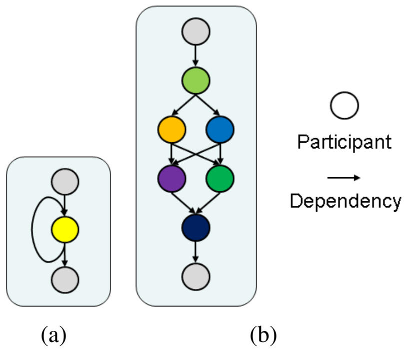

Figure 2. Example of typical LQCD workflows: (a) abstract configuration generation workflow, (b) abstract two-point analysis workflow. Both figures expand into workflows shown in figures 1(a) and 1(b) respectively

2.2. Data model and provenance

This section describes the data model defined for the framework. We use an object-oriented model for describing the entities and their relationships. Groups of classes are implicitly divided into six spaces: parameter, data provenance, secondary data, process history, monitoring, and mitigation spaces. The spaces do not define a hard boundary between entities, but rather a logical and functional aggregation. The first four spaces deal with workflow related data, while the last two are used for the system reliability.

The *parameter space* contains all parameters used as input for workflows, including physics parameters (e.g. quark masses), algorithmic parameters (e.g. convergence criteria) and execution parameters (e.g. number of nodes used). Parameters are name-value pairs that can be grouped in sets. Parameter sets are used to describe the physics properties of ensembles or hold analysis campaign attributes.

The relationship between input and output files is kept within the *data provenance space*. Output files are modeled as products, which have optional properties. Properties are metadata associated with

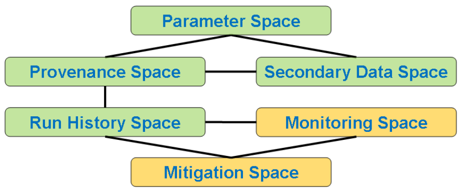

Figure 3. Data model divided into six spaces

the output files. For complex analysis campaign workflows, products may have multiple parent-children relationships. Products also contain a reference to the workflow participant instance and parameters used to generate it. This allows the product to be reproduced by invoking the original participant with the original input parameters.

In addition to data files and their properties, LQCD workflows also produce domain specific *secondary data*. Secondary data classes have meaning only within the LQCD context. The application of this model to another scientific problem requires the addition or removal of classes within the secondary data space.

The *process history space* holds information regarding workflow descriptions and participant definitions. Abstractions in this space keep information about the binary code, including command line format, version and pre and post run scripts for manipulating input and output parameters and command line formatting to invoke the actual binary code. Additionally each participant may have a list of conditions that must be evaluated pre-execution, during execution and post-execution.

In the *monitoring space*, information regarding sensor configuration and historical values are kept. Common values tracked by sensors are CPU usage, memory available and free disk space.

The *mitigation space* contains failure recovery strategy and properties used in conjunction with the monitored data.

2.3. Execution tracking

During workflow execution participants that satisfy data and control dependencies are mapped onto worker machines and executed. Periodic and on-demand monitoring of vital health parameters on allocated nodes is enabled according to pre-specified rules. These rules define conditions that must be true pre-execution, during execution and post-execution. Figure 4 depicts the mapping of a participant onto worker machines and related sensors.

Participant conditions are used to define preconditions, postconditions and invariants. Pre and postconditions are evaluated before a participant is started and after its completion, while invariants are periodically monitored during the participant life time.

Conditions refer to sensor values, which are periodically checked. Values are verified according to the scope defined by the condition, insuring the monitored values fall within the expected range. We define the following notation to describe conditions: $\mathcal{H}(x)\ op\ value$, where $\mathcal{H}(x)$ defines the sensor being monitored, *op* defines the comparison operation and $value$ is the expected sensor condition.

An example of a precondition for participants that must retrieve data from dCache[1] [5] is the following: $\mathcal{H}(pm)(t) > 0$, where pm is the dCache pool manager, $\mathcal{H}(pm)$ is the sensor value at instant t. The verification of participant output products can be performed by a postcondition $\mathcal{H}(file) = 1$, where $file$ is the output file name and the value 1 indicates it is present. Finally, during the execution the computing node (cn) assigned to the participant must be available: $\mathcal{H}(cn) = 1$.

Monitored sensor values ($\mathcal{H}$) for running participant are propagated upwards through the reflex and healing architecture, which consist of hierarchical network of decentralized fault management entities -

[1] LQCD uses the dCache system (http://www.dcache.org) for data storage. dCache is a system for storing and retrieving large amounts of data, distributed among a large number of heterogenous server nodes, under a single virtual file system.

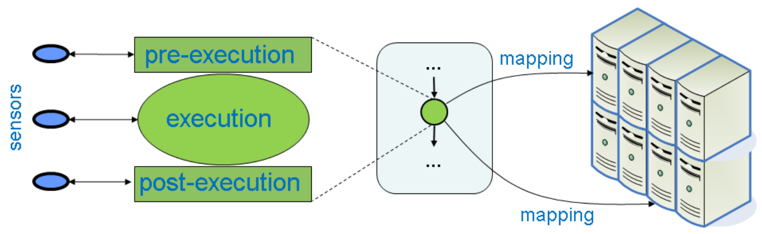

Figure 4. Sensor values, represented on the left, are evaluated by preconditions, postconditions and invariants associated with a participant. The actual machines containing the monitored sensors are defined when a participant is mapped to run on a set of computing nodes.

the reflex engines [6].

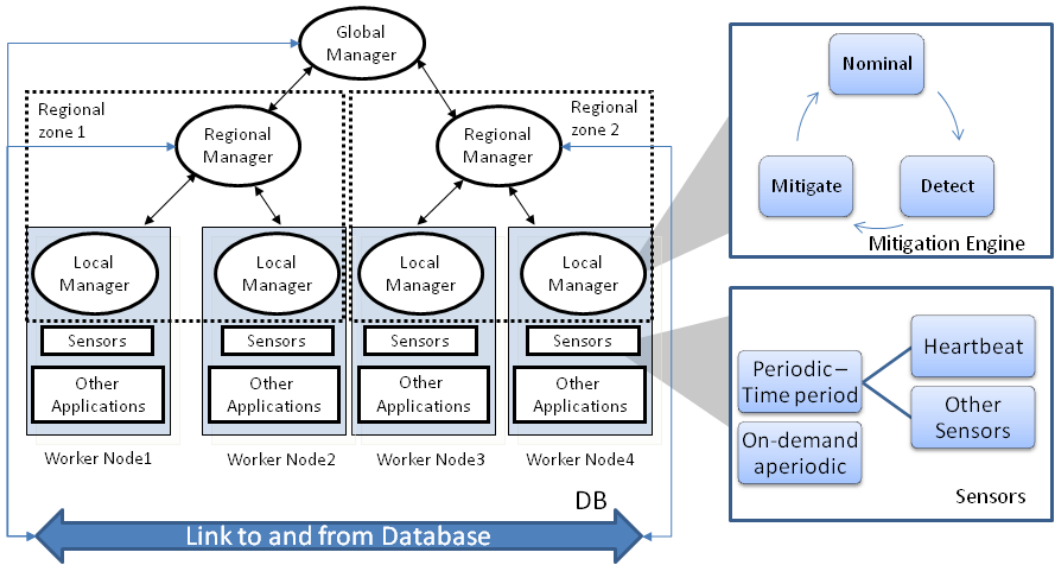

Figure 5. Hierarchical organization of reflex engines within computing cluster

Reflex engines are distributed across the cluster on all computation nodes. To aid in fault isolation and quick recovery, they are divided into regions based on their computer racks (figure 5). Each region is managed by a head node that is identified as the *regional manager*. This manager relays the sensor information for the computing nodes under its supervision to the database. *Local Managers* run on all computation nodes that are used in execution of participants. They are used to monitor and mitigate the behaviors internal to that node.

We follow the principles of autonomic computing [7] and try to incorporate the mitigation and monitor state machines as close to the source as possible. In other words, most of them are located on the concerned computation node. However, some commands such as IPMI (Intelligent Platform Management Interface) reset and the heartbeat monitors need to be outside concerned machine and are placed on the regional node. Monitoring information from all nodes is channeled into a database for future forensic analysis, if required.

3. Integration

The integrated workflow, monitoring and mitigation system is shown in figure 6. The left side contains the workflow execution while components on the right side are part of the cluster reliability system.

The workflow execution engine provides an interface for submitting concrete workflows. Multiple concrete workflows can be handled by multiple execution engine threads. As participants are declared ready to run based on the dependencies, the workflow engine contacts the global manager. Events are exchanged asynchronously between the global manager and the workflow engine. The centralized controller processes the events (associated actions are specified in the configuration database) and then sends required commands using events to the local managers and regional managers. Information regarding the participant conditions along with a unique participant and workflow instance identifier are used to start participant specific monitors in the worker nodes.

At participant start up, the preconditions are checked at the global manager level. Any violation of preconditions generates a message back to the workflow engine informing about the violation. This message sent back to workflow engine is a mitigation strategy and is specified in the mitigation reflex engine module at the global controller level.

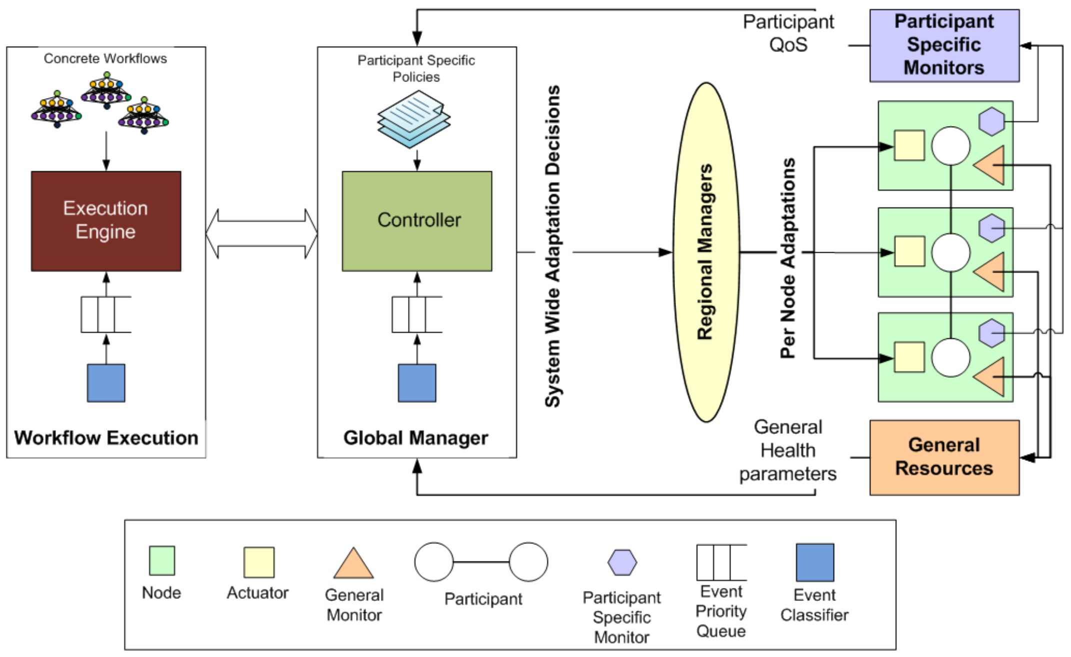

Figure 6. Complete runtime framework with integrated workflow, monitoring and mitigation integration

When participants are submitted for execution, all invariant conditions result in the activation of participant specific monitors, if required. An example of invariant condition is the availability of the dCache pool manager: $\mathcal{H}(pm)(t) > 0$. When a condition is violated, an event is sent to the workflow execution engine. Action to avoid fault propagation is then taken, for example, by restarting the same participant on a different set of nodes.

Similarly when a participant completes, any postconditions are evaluated. Usually postconditions are workflow related, for example to make sure expected output files have been created. Specific examples of these conditions are discussed in the case study section.

4. Case study

In this case study we considered the example of a two-point analysis workflow. It is a coordinated set of calculations aimed at determining a set of specific physics quantities. For example, predicting the mass and decay constant of a specific particle determined by computing ensemble averaged two-point functions. A typical campaign consists of taking an ensemble of vacuum gauge configurations and using them to create intermediate data products (e.g. quark propagators) and computing meson n-point functions for every configuration in the ensemble. An important feature of such a campaign is that the intermediate calculations done for each configuration are independent of those done for other configurations.

The sample workflow shown in figure 1(b) is the representation of an analysis campaign for a single configuration file of an ensemble. The complete workflow consists of N independent instances of the concrete workflow on left side of the figure. The N outputs from the campaign output are later combined and analyzed. An implicit behavior of analysis campaign is that the number of participants and outputs depend on the input parameters. For example, the number of participants generating heavy quark propagators is derived from the number of quark masses and source types specified by the input parameters.

For this case study, we only used a single computation node, even though the participants are MPI (Message Passing Interface) jobs capable of running on multiple machines. Jobs for this study were submitted using a simulated PBS (Portable Batch System) queue that transferred the job to the concerned computation node and started it. The other node was used to run reflex engines and the workflow execution engine. The communication between reflex engines and workflow execution engine was achieved using UNIX pipes as they were running on the same machine. A third node was used to simulate the dCache pool manager.

We used the following failure scenarios to test our prototype framework:

Failure of a dCache node: The first participant on the two-point concrete workflow is called getFile. This participant is responsible for fetching the gauge configuration used as input for the LQ (Light Quark) and HQ (Heavy Quark) participants. A precondition of getFile is that the dCache pool manager (pm) must be available, therefore $\mathcal{H}(pm)(t) > 0$ should be true.

Disk space precondition: The second level of participants on the two-point concrete workflow is composed by HQ and LQ instances. It is known that HQ produces a large output file whose size is in the order of a few GB. A precondition is to check if the disk space available meets the requirements: $\mathcal{H}(\mathrm{cn}, /\mathrm{project}) < 10000$. In this precondition $/project$ represents the disk partition.

Failure of a computation node: The allocated computation node must be online i.e. $\mathcal{H}(cn)(t) > 0$, where cn is the node allocated to a participant.

4.1. dCache failure mitigation scenario

We used the same setup as described in previous section. To inject failure, we deliberately sent a shutdown command to the dCache node making it unavailable at the time of execution of getFile. In the case of condition violation the workflow engine is informed about the unavailability of dCache and an action must be taken.

Currently, our strategy is to reschedule the getFile participant after $\Delta_{timeunits}$. The reschedule time is computed based on historical knowledge about recovery rate of pool manager. Other actions may be available for the same failure and could be taken by the mitigation framework, if a reflex engine is present on the pm node.

4.2. Disk failure mitigation scenario

We were able to simulate the effects of disk space precondition violation in a similar manner. Before the participant starts running, the local disk sensor is checked to make sure enough space is available. An event is created and sent back to the workflow execution engine if the condition has been violated. The mitigation action is provided by the local reflex engine, by deleting files from the temporary folder on project partition.

4.3. Computation node failure

Heartbeat monitors are used as invariant conditions on all nodes that execute a participant. If a node fails, the default action is to report the failure to workflow execution engine, which instructs other nodes involved in the job to perform clean up and terminate the job. This saves us time in scenarios where without any notification the other nodes running the job would have been blocked from executing any other participant for the specified wall time. Figure 7 shows the series of timed events related to a node failure detected by missing heatbeats.

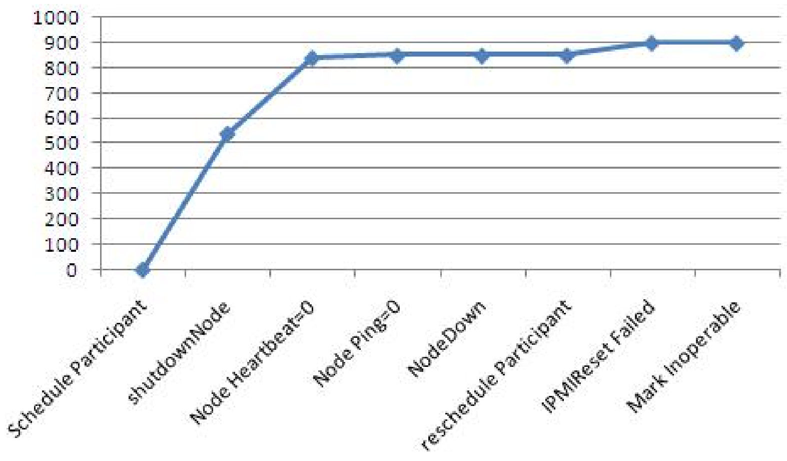

Figure 7. Timing in seconds

4.4. Data provenance
Information about data produced by workflows is saved in the provenance space data model. It allows users to quickly perform queries based on parameters such as participant type, input parameters and input configurations used. It is possible to reconstruct provenance through graphs for the visualization of the steps taken by the workflow.

The Ruby language, used in conjunction with Rails to implement the current prototype, can be used to perform specific queries based on the provenance model. The key feature of Ruby on Rails [8] is the implementation of the active record pattern [9]. Active record defines the connection between the classes defined in the provenance model and database tables. Ruby on Rails offers transparent connections to a variety of databases.

Results from the queries can be manipulated within the language or exposed to scientists through web interfaces. The queries using interactive Ruby shell shown in figure 8 provide answers for the following questions: (1) Where is the configuration file defined by the identifier *l612f21b6600m0290m0484.6*? (2) What where the input parameters used to generate the product defined by identifier *l612f21b6600m0290m0484.6*? (3) Which configuration files where generated using algorithm su3_rmd version 1.2? (4) What are the configuration files generated with residuals smaller than 10E-5?

The format of queries follows an object-oriented style since the provenance products are instances of classes defined by the data model. A common approach to minimize direct iterations with the provenance database is to define a set of common used queries. The Ruby on Rails environment can be used to quickly make these default queries accessible through the web.

5. Conclusions and future work
In this paper we presented a workflow execution framework that tracks data provenance and reliably runs scientific workflows. Essential run time information is constantly verified by the monitoring framework, while conditions pertinent to the running workflow are monitored only during execution time. Additionally, the conditions are specified at the participant granularity, avoiding overzealous

```
(1)$ product = Product.find(:all, :conditions => \
     "file_name like '%l612f21b6600m0290m0484.6%'")
...
(2)$ product.participant_instance.parameter_set
...
(3)$ su3 = ParticipantType.find(:first, \
              :conditions => "name == 'su3'")
(3)$ su3.participant.participant_instances.products
...
(4)$ properties = ProductProperty.find(:all, \
             :conditions => "name == 'error'" and \
                        "value > 10E-5")
(4)$ products = properties.products
```

Figure 8. Sample provenance queries using Ruby and active record

monitoring and consequent use of computing resources that would otherwise be available for the scientific applications. The data provenance provided by the framework allows users to quickly find and reconstruct products based on participants and input parameters. We have developed a prototype of the proposed architecture and used it to demonstrate the feasibility of fault-tolerant enabled LQCD workflows.

Many workflow systems currently lack fault tolerant features, which is one of the top priorities for scientific workflows that run for a long time. Hardware and software faults are common and need to be addressed at both workflow and node levels. This work fills the gap between monitoring and workflow systems, allowing proactive behavior on the presence of failures.

We plan to add missing features to the prototype, such as completing the set of conditions, distinction between reliability and workflow related conditions, replacement of the current workflow engine, and greater functionality for accessing and using data provenance information. Further tests are planned on a virtual cluster environment for further test the prototype framework. We are preparing a virtual environment for testing the system prototype based on Suns Virtual Box running Ubuntu Linux. This environment allows full control over the resources, including injection of failures and observation of system recovery behavior and avoids the use of actual cluster resources and competition with production runs.

References

[1] Piccoli L, Kowalkowski J B, Simone J N, Sun X H, Holmgren D J, Seenu N, Singh A G and Jin H 2008 *SWBES '08* (Indianapolis, IN, USA)

[2] Wohed P, Andersson B, ter Hofstede A H, Russell N and van der Aalst W M 2007 Patterns-based evaluation of open source bpm systems: The cases of jbpm, openwfe, and enhydra shark Tech. rep.

[3] Deelman E, Singh G, Su M H, Blythe J, Gil Y, Kesselman C, Mehta G, Vahi K, Berriman G B, Good J, Laity A, Jacob J C and Katz D S 2005 *Sci. Program.* **13** 219–237 ISSN 1058-9244

[4] Deelman E, Callaghan S, Field E, Francoeur H, Graves R, Gupta N, Gupta V, Jordan T H, Kesselman C, Maechling P, Mehringer J, Mehta G, Okaya D, Vahi K and Zhao L 2006 *Second IEEE International Conference on e-Science and Grid Computing*

[5] Bakken J, Berman E, Chih-Hao H, Moibenko A, Petravick D and Zalokar M 2003 *Proceedings. 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies* pp 101–104

[6] Dubey A, Nordstrom S, Keskinpala T, Neema S, Bapty T and Karsai G 2007 *Innovations in Systems and Software Engineering* **3**(1) 33–52

[7] Sterritt R, Parashar M, Tianfield H and Unland R 2005 *Advanced Engineering Informatics* **19** 181–187 URL `http://dx.doi.org/10.1016/j.aei.2005.05.012`

[8] Bachle M and Kirchberg P 2007 *IEEE Software* **24** 105–108 ISSN 0740-7459

[9] Fowler M, Rice D, Foemmel M, Hieatt E, Mee R and Stafford R 2002 *Patterns of Enterprise Application Architecture* (Addison-Wesley Professional)