2009 Sixth IEEE Conference and Workshops on Engineering of Autonomic and Autonomous Systems

Using Runtime Verification to Design a Reliable Execution Framework for Scientific Workflows

Abhishek Dubey[*] Luciano Piccoli[†‡] James B. Kowalkowski[†] James N. Simone[†]
Xian-He Sun[‡] Gabor Karsai[*] Sandeep Neema[*]
[†]Fermi National Accelerator Laboratory, Batavia, IL, USA 60510
[*]Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN
[‡]Illinois Institute of Technology Chicago, IL, USA 60616

Abstract

In this paper, we describe the design of a scientific workflow execution framework that integrates run-time verification to monitor its execution and checking it against the formal specifications. For controlling workflow execution, this framework provides for 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 used to generate 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 prespecified invariant conditions with actions to be taken upon violation of invariants.

1. Introduction and Problem Motivation

Current computing power and storage capabilities allied to distributed computing models allow the production of e-science in areas such as biology, disaster simulation, and physics among others. The organization and reliable processing of the massive information produced is critical for its effective use in new discoveries. The long range objective of our group is to support the computation infrastructure needed for studying the physics of Lattice Quantum Chromodynamics (LQCD) by using computer simulations[1]. LQCD, numerical study of QCD quantum field theory on a four-dimensional discrete lattice, generates considerable data that are processed at several institutions. Applications, software libraries, input data and workflow[2] recipes are shared among collaborators worldwide[3].

Table 1. Cluster Evaluation Metrics.

$$Availability_{T1}^{T2} = \frac{\int_{T1}^{T2} OnlineNodes.dt}{\int_{T1}^{T2} NodesinCluster.dt} \quad (1)$$

$$Utilization_{T1}^{T2} = \frac{\int_{T1}^{T2} BusyNodes.dt}{\int_{T1}^{T2} OnlineNodes.dt} \quad (2)$$

$$Productivity_{T1}^{T2} = \frac{\int_{T1}^{T2} SuccessfulJobs.dt}{\int_{T1}^{T2} BusyNodes.dt} \quad (3)$$

Unlike many e-science experiments that make use of Grid resources[4] for harvesting capacity processing power, LQCD computations employ tightly-coupled parallel processing, which require computers with high-speed, low-latency networks. At Fermi lab in Batavia IL, majority of the computers that are used for LQCD computations are dedicated clusters. Use of dedicated clusters allow fine tuning of binary codes to exploit capabilities of underlying architecture. LQCD workflows can effectively exploit the capacity of one or more parallel computers by running many independent computations at once.

Clusters built out of commodity computers, used for scientific computing, exhibit intermittent faults, which can result in systemic failures when operated over a long continuous period for executing workflows. While executing, a typical workflow can spawn hundreds of jobs. Many of these jobs are computation intensive and use MPI across dozens to hundreds of processing nodes. Typically, several users analyze different workflows on the clusters concurrently. Given the scale of numbers, diagnosing job failures,

[1]http://www.usqcd.org/fnal/

[2]A workflow is a planned set of computation jobs.

[3]Information about LQCD project can be obtained from http://www.usqcd.org/

[4]See DOE Scientific Computing on Grid initiative at http://www.doesciencegrid.org/

978-0-7695-3623-1/09 $25.00 © 2009 IEEE
DOI 10.1109/EASe.2009.13

fault isolation and fault mitigation becomes critical, specifically when the success of whole workflow might be affected by even one job failure.

Manual administration, though essential, is slow to respond to the intermittent faults. Therefore, we need to supplement it by an autonomic management subsystem that can provide effective management support to the administrators.

We measure the effective use of our machines by using three metrics, which are availability, utilization and productivity (see table 1). The primary accounting measurement is node-hours. We use node-hours instead of computation cycles as we have homogeneous clusters. A node hour is the number of hours that can be used for computation on a computing node. In the table, $T1$ and $T2$ are the global timestamps specifying the duration of metric evaluation. Availability is the ratio of number of nodes online versus all the nodes available in the cluster (including offline machines). Utilization measures the number of nodes that are busy. Productivity is the fraction of busy time that was spent in doing successful workflows i.e. ones that did not fails. Here, we are discounting cases where a workflow might succeed but due to various algorithmic reasons might lead to unproductive data.

In typical e-science workflows, the Grid is used as the main source for job processing and close remote application monitoring is limited [14]. On the other hand, it has the advantage of replicating the same job on different sites for fault tolerance purposes. This flexibility does not exist in the dedicated LQCD processing environment. We need to increase productivity by enabling reliable operation over available hardware without use of redundancy.

To increase productivity, we must provide

- Workflow Management Framework: Current processing model relies on simple Perl-based scripting workflow languages for driving LQCD workflows. We require a framework that provides specification and execution of workflows with data provenance using a formal dataflow based language.

- Fault Isolation: Identify and explicitly specify conditions for all jobs belonging to a workflow, which when failed will constitute a fault. Also, we need to construct proper observers and formal logic that will be required to monitor these conditions.

- Sensor Framework: We require a framework of sensors that provide the basic monitoring information [6].

- Fault Prediction: If we can identify explicit failure conditions, we can do a search over all specified workflows to predict the failure of executions in near future.

- Fault Mitigation: Currently, recovery from failures is manual and is the responsibility of the user running the experiment, i.e. understand failures from log files and restart processing from a known working state. We need to automate the execution of mitigation actions.

Sole addition of a monitoring and reliability framework as an add-on does not suffice to properly add failure feedback and recovery action to the current workflow execution model. A proper solution is to have both systems integrated from the design specification.

In this paper, we propose a framework for executing scientific workflows with integrated reliability features. During the workflow specification, each job (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.

2. Formal Foundations

2.1. Preliminaries: Model of Time

To reduce the complexity of maintaining multiple clocks, we assume the notion of a global time, $\mathcal{T}$, that is monotonically increasing and dense in the set of rational numbers $\mathbb{Q}$. We use rationals rather than real numbers because they correspond to practically measurable time intervals i.e. that can be measured by clocks with finite precision. All model values and all measurements are defined with respect to this global time. It is implemented on all cluster computers using a synchronization protocol such as NTP.

To define invariant conditions that must be true for a workflow we require a timed automaton model to observe the behavior of the system while it is executing. Timed automaton is a classical model used for abstracting time-based behaviors of systems, which are influenced by time [3, 9]. It consists of a finite set of states called *locations* and a finite set of real-valued clocks. Time passes at a uniform rate for each clock in the automaton. Instantaneous transitions between locations are triggered by the satisfaction of associated clock constraints known as *guards*. During a transition, a clock is allowed to be reset to zero. Clock constraints called *invariants* are added to all locations to make them urgent. These invariant constraints must be satisfied for the system to validly stay at a location. A timed automaton that cannot transition out of a location with violated invariant is said to be *blocked*. For a detailed formal definition, readers are encouraged to refer to [3].

2.2. Resources

Generic workflow management problem is defined over a set of "Deployment Resources". Deployment resources

can be further divided into **computation** and **communication** resources. In this discussion, we will ignore communication resources and focus on computation resources.

Computation attributes correspond to hardware facilities required to execute computation tasks, including processing speed (number of instructions per second), memory size (amount of random access memory required). Formally, deployment resources are defined as a structure $R = (A_C)$, referred to as resource domain, consisting of a set A_C of computation resource attributes. A resource $r \in R$ is defined as a tuple $r = (n, d, u)$, where n is the name, d is the valid range of values, u is the measurement unit. For example, ('cpu', [0,4], GHz) $\in A_C$.

2.3. Computation Nodes N

Scientific workflows are executed and managed over a set of computation nodes that provide the resources R mentioned in previous section. This resource configuration RC^R, defined with respect to the computation domain R is a structure $RC^R = (N, E, \phi)$, where

- N is the set of available physical computing machines.
- $E \subseteq P \times P$ is a set of communication links connecting computation machines.
- $\phi : N \times R \rightarrow \Re$ is a resource capacity level map, where $\phi(n, r)$ is the level of resource r in the node n.

2.4. Participants P_t

Each computation task is defined as a participant P_t in the system. A scientific computation task takes in a number of parameters from the set of input parameters I, and generates a number of output products from the set of output products O: $P_t : 2^I \rightarrow 2^O$. Here 2^I represents a power set of parameters.

Participants are classified into categories called participant types, denoted by P_T. A participant type is a class of algorithms used for generating similar products but with varying quality parameters. For example, numerical integration is a "participant type", while Gauss algorithm or the Simpson's rule algorithm are its implementations and hence its participants. Choosing one participant belonging to a type over another is an optimization problem that considers all the quality properties of all participants. However, discussion about quality parameters is out of scope of this paper.

For every participant, we denote by a map, $\psi : P_t \times R \rightarrow \Re$, the minimum required resources to run the participant on one node. For example, $\psi(P_t, '\text{RAM}') = 512$ means that the algorithm implemented by participant P_t cannot run if the available RAM is less than 512 MB.

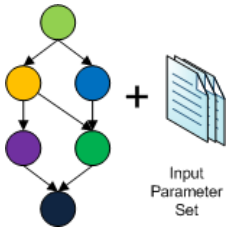

Figure 1. Sample workflow definition

A participant is generally a legacy application invoked from within a scripting language program. Typically, they are parallelized over multiple computation nodes. The relationship between participants is defined by a workflow W.

2.5. Workflows W

All participants belonging to a workflow are related to each other by control and data dependencies. The workflow is defined as $W(P_{ts}, I_{ps}, D_{pt})$, where P_{ts} is the set of participants, I_{ps} is the set of user input parameters and $D_{pt} \subseteq P_{ts} \times P_{ts}$ defines a set of dependencies between the participants. A Workflow, therefore, is a plan dictating how various participants will be executed.

Scientific workflows require the coordination of data processing activities, resulting in executions driven by data dependencies; whereas in business workflows, the control dependencies describe the processing steps, which usually are coordinating activities between individuals and systems [14]. A pictorial example of a workflow is shown in figure 1, where P_t is represented by a circle of different shades for distinct participants, arrows define the data dependencies D_{ps}, which added to input parameter sets define a workflow W.

The task of running a workflow W is carried out by an "execution engine", W_e. The input to W_e is a workflow and input parameters sets (fig. 1) described in a workflow language format. Currently, there is no agreed standard for scientific workflow languages. Several groups have developed languages, such as SwiftScript [15] and AGWL [8], that are specific for their respective execution engine.

Ideally, the execution engine W_e is responsible for tracking the run time dependencies D_{ps} and enable participants as dependencies are met and input data are available. However, this is not true in all workflow engines that are currently available.

2.6. Task Allocation Map

Task allocation map is used by a workflow engine to assign computing nodes that will be used to execute a participant. Formally, given a participant P_t and the set of com-

putation nodes N, the task allocation map $\theta : P_t \rightarrow 2^N$ allocates a number of nodes to a participant.

The task allocation map is computed with respect to a scheduling policy. This is an active area of research. In this paper, we assume the existence of such a policy. We use PBS/Torque for imposing the scheduling policy[5].

Note that a valid task allocation map θ has to satisfy the following:

Resources Nodes allocated to a participant P_t must have the minimum amount of resources required. Notice that the unit of scheduling that we consider is a computation node. Alternative scheduling policies may allocate a participant a specific CPU core on a node.

$$(\forall r \in R)(\forall n \in \theta(P_t))(\phi(n,r) \geq \psi(P_t,r))$$

2.7. Sensors and Filters

A sensor program is used to monitor the current utilization of computation resources associated with a node. It is a map $S : N \times R \times \mathcal{T} \rightarrow \Re$. It provides a measurement of current value with the current timestamp for the resource. For all resources, sensor scripts produce normalized values. Therefore, the actual value in terms of a measurement unit of the resource can be attained by $S(n,r) * \phi(n,r)$. The timing of a sensor program can be periodic or aperiodic. Jitter and synchronization are two challenges posed by sensors executing in a computing cluster. A detailed discussion about these can be found in [6].

In order to sieve unimportant sensor information, they are used with a filter. As the name suggests, a filter evaluates the measurement by the sensor against the filter criteria. Outputs of a filter are events. Events are a tuple, $E = (\text{Type}, \text{Timestamp}, \text{Value})$. A type is a unique combination of the computation node identifier and resource. For example, the event generated for sensor $S(n,r)$ will have a type of $n.r$. Timestamp is the time of measurement and Value is the actual measure.

These events on a global time scale can be correlated to get interesting statistical insight into general cluster health. For example, a sustained rate of increase in temperature from a region in cluster signifies a cooling system failure. We currently employ and use the temperature based correlation.

Heartbeat: For computing nodes, a specialized sensor called heartbeat [6] acts as watchdog and informs whether a computing node is online or not. A Heartbeat Sensor is in fact a combination of a periodic sensor on the concerned computing node and a filter on a monitoring node. Overall this combination generates events with two possible values $\{0,1\}$, where 0 means the node is offline and 1 means the node is online. We denote heartbeat sensor as $\mathcal{H} : P \times \mathcal{T} \rightarrow \{0,1\}$. Liveness condition for a node named pion1 implies that $\mathcal{H}(\text{pion1}, t) = 1$, where t is the current time.

[5]http://www.clusterresources.com/pages/products/torque-resource-manager.php

2.8. Events and Conditions

Timed Traces of Events: Output of a sensor and filter combination are timed traces of events, which are sequences of the form $< a_1, t_1 >, < a_2, t_2 >, \cdots, < a_n, t_n >$, where $a_1, \cdots, a_n$ are events and $t_1, \cdots t_n \in \mathbb{Q}$ are the times at which those events have occurred. Every event is typed by the name of the sensor and computation node that it is being generated from. We maintain these entity relationships by using a configuration database.

These timed traces of events are converted to an interval timed trace defined over measured value of resources. It is done by using a zero-order hold digital to analog filter.

Interval Timed Trace: From a given timed trace of event, which are discrete in nature, one can create an interval timed trace, which is defined over continuous time ranges. It is a timed trace of the form $\tau_1 a_1.value, \tau_2 a_2.value, \cdots, \tau_n a_n.value, \tau_{n+1}$ where τ_i is an interval $[t_{min}, t_{max}] \subseteq \mathbb{Q}$, $a_i.value$ is the measurement value contained in the event a_i. There is one interval timed trace for each type of an event.

Query Operator: For an interval timed trace, we specify a query operator $query : \mathbb{Q} \times Type \rightarrow \Re$ that provides the measurement value at that time. Here $Type$ is a universal set of all types of events. This query operator can be used from an observing computing node to evaluate condition over resource variables on another node using predicates $\leq, <, >, \geq$. Since this query operator is implicitly defined by association to a sensor type, we will omit it for brevity and in this paper write conditions directly on the interval timed trace of a sensor type. More details are discussed in next section.

2.9. Runtime Checkable Properties

The underlying foundation of specifying safe operation conditions for workflows is a system of logic based on events and conditions. Events occur instantaneously during system execution. Conditions represent state of systems over duration of global time. Primitive conditions are defined with respect to a measure provided by a sensor. Primitive events are defined with respect to start and end of primitive conditions and specialized actions such as start of a workflow participant.

Properties that can be tested for a participant are specified over timed traces of events and interval timed traces of conditions. The set of all properties is denoted as $\mathbb{P}$. We

divide the set of properties that can be specified into two groups:

Untimed Untimed properties are instantaneous properties defined as a predicate over current value of resource type as specified by its interval timed trace. Formally, given an interval timed trace, the untimed properties are defined as $P ::= p|!p|P \wedge P|P \vee P$, where p is a primitive property defined using the query operator described in previous section, a real number and the predicates $\leq, <, >, \geq$.

Timed Timed properties are defined using timed computation tree logic (TCTL) [10] and references therein.

- *Reachability*: These sets of properties deal with the possible satisfaction of a given untimed property a in a possible future state of the system. For example, the TCTL formula $E\Diamond\phi$ is true if the predicate logic formula $\phi \in P$ is eventually satisfied on any execution path of the system.
- *Invariance*: These sets of properties are also termed as safety properties. As the name suggests, invariance properties are supposed to be either true or false throughout the execution lifetime of the system. For example, the TCTL formula $A\Box\phi$ is true if the system always satisfies the predicate logic formula $\phi \in P$. We use the invariance property $A\Box\phi$ to describe the safe set for a participant.
- *Liveness*: Liveness of a system means that it will never deadlock, i.e. in all the states of the system either there will be an enabled transition and/or time will be allowed to pass without violating any location invariants. Liveness is also related to the system responsiveness. For example, the TCTL formula $A\Box(\psi \rightarrow A\Diamond\phi)$ is true if a state of the system satisfying $\psi \in P$ always eventually leads to a state satisfying $\phi \in P$.

In the current state of the framework we only support untimed properties and invariance properties. They are specified for all participants as **preconditions, invariant conditions**, and **postconditions**. Preconditions (postconditions) are untimed and evaluated before (after) participant starts (finishes). Invariant conditions must be true during the execution of a participant.

2.10. Monitors: Checking for Satisfaction of a Property

Given a property $\phi \in \mathbb{P}$, the set of all valid properties, we construct a timed automaton representation. We call this a model of the property. For all such models, a universal state *fault* is used to represent the state that the property has been violated. The satisfaction condition is then the dual of acceptance of a given timed/interval trace by this model. The module in the framework that performs this check is called a **Reflex Engine**[5]. We call the class of reflex engines that are used to identify the faults, which are induced due to violation of properties **Monitors**.

Example: dCache [4] is well known and respected as a powerful distributed storage resource manager. It provides a single rooted view of the file system to any actor that wants to access or store a file. In the basic configuration, an actor such as a participant sends the request to access a file to the manager of the pools. The manager sends the participant the information about the actual node where the file is stored on and the absolute path to the file. In our clusters, we have seen that sometimes the pool manager dies i.e. it does not respond to a request. In such a case, a participant that is scheduled to execute cannot run. To improve productivity, we specify a precondition that the pool manager is alive for all participants.

Let $pm \in N$ be the pool manager. Then we specify the property $\mathcal{H}(pm, t) > 0$, $t \in \mathbb{Q}$ is the time at which the property is evaluated.

3. Distributed Monitoring and Mitigation Framework

Before describing the overview of cluster-wide framework let us explore the architecture of monitoring, diagnosing and mitigation framework running on a computation node.

3.1. Reflex Engine architecture on a computation Node

Recall that the key monitoring timed traces and event traces are generated by sensor and filter programs working together. In our previous work, we had presented a distributed monitoring framework used in our clusters [6]. In the same paper, we had also described a scheduling algorithm used to execute the sensors on all computing nodes.

Along with the sensors all computation nodes also contain two real-time reflex engine modules. A real-time reflex engine contains one or more timed state machines that can accept a timed event trace and/or an interval timed trace. We divide all possible reflex engines into two sets, **monitors** and **mitigators**. Note that this is only a logical classification. Both of them have the same semantics. Refer to [5] for a detailed discussion.

Fig. 2 describes the basic structure inside a managed node. The lowest layer is the hardware. Above it we have the operating system, which schedules all the user space and

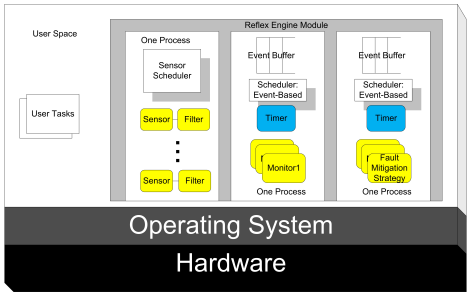

Figure 2. A computing node in the real time reflex and healing framework.

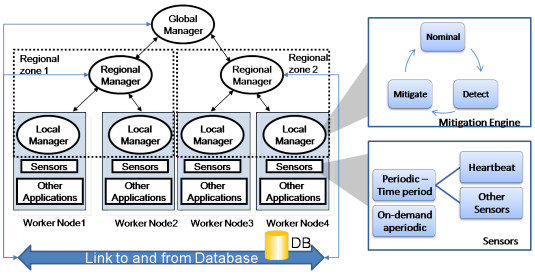

Figure 3. Hierarchical reflex engines

kernel space tasks. In the user space, we have the reflex engine module.

A reflex engine module on a computing node is called manager. It is composed of three distinct user level processes. First process is used to execute all the sensor and filters to generate events. Second process is a reflex engine classified as monitor. It has several monitors that can consume some of the events received in its buffer. The event-based scheduler maintains a lookup table for the event-type and the monitor that can use the event. Once an event comes in it is forwarded by the scheduler to the appropriate monitor. The timer is used by the monitor to measure the interval of time to check if a given interval timed trace violates the monitored property. All monitors execute on their own thread and maintain their state during execution. Upon reaching a faulty state, they generate an event which is again a tuple with a type, timestamp and, if applicable, the last measured value that caused the violation.

Monitors are divided into groups, general purpose i.e. they are always on, and participant specific i.e. they are turned on/off on-demand.

The third process, the mitigation reflex engine, works similarly but the state machines are mitigation strategies, which cause various set of actions upon occurrence of a certain type of fault event. Semantics of mitigation strategies are discussed in [5].

3.2. Distribution: Hierarchy of Managers

Reflex engine managers 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 racks (fig 3). 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 [13] 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 reset[6] 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.

Centralized Control: The centralized controller oversees the reported events generated across the cluster. It integrates with the workflow framework and report violations of conditions/events that might trigger a workflow failure. We will present the steps in this integration in section 5. It also contains an http rendering engine to visually report the system health as a whole.

4. Workflow Framework

To coordinate the computation of a large number of applications (participants P_t), physicists describe the work to be performed in a generic recipe called parameterized workflow. Typically, the parameterized workflows are of two classes: creation of configuration gauge ensembles and process of ensembles in analysis campaigns. The parameterized workflow in conjunction with a set of input parameters yields a concrete workflow. Example of input parameters are values for particle masses.

Management of input parameters is one of the requirements outlined for LQCD workflows [12]. In order to provide an integrated workflow and reliability environment we have developed an object oriented provenance model. This model is used for describing and managing input products, maintain participant types, participants and also the respective runtime and resource information. It is also used to generate products and related properties. This framework enables us to restart a workflow from last point-of-failure. Provenance model is described in next section.

[6]IPMI is a standardized management protocol that can be used to reset the power of a machine out of band

4.1. Provenance Model

We use an object-oriented model for describing the provenance classes and their relationships. Groups of classes are implicitly divided into three spaces: parameter, data provenance, and process execution spaces. The spaces do not define a hard boundary between sets of classes, but rather a logical and functional aggregation.

The parameter space archives all parameters used as input for workflows, including physics parameters (e.g. particle 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. Groups of parameters are used to describe the physics properties of ensembles or hold analysis campaign attributes. Parameter sets are optionally identified by names.

The relationship between input and output products is kept within the data provenance space. Data products are modeled as **Products**, which have optional **ProductProperties**. An example of a product generated from a configuration generation workflow is the ensemble configuration file named *l612f21b6600m0290m0484.6*. Its parent configuration file is the product named *l612f21b6600m0290m0484.3* and child *l612f21b6600m0290m0484.9*. These specific names are complicated. However, we reduce this accidental complexity by use of a database with specific tables implementing these relationships.

The products also have a reference to the workflow participant instance that generated it, allowing the file to be reproduced by reconstructing the processing steps.

The process execution space holds information regarding workflows and participants. Each generic class of participant is defined as a **ParticipantType** (e.g. numerical integration). An actual implementation of a ParticipantType is realized by a **Participant** (e.g. Gauss algorithm version 2). The Participant holds information about the binary code, including command line format, description of input and output parameters, and pre and post run scripts.

When a new participant is added the associated preconditions, invariant conditions and postconditions are specified. For example, the executing node must be available throughout the execution: $A\Box\mathcal{H}(n,t) > 0$. The default mitigation action is the restart of a participant. Execution of a Participant is recorded as a **ParticipantInstance** (e.g. Gauss algorithm version 2 ran successfully on node A producing file X).

Similarly, for managing workflows the **WorkflowType** defines a class of workflow (e.g. two point analysis). The **ParameterizedWorkflow** class contains the definition of parameterized workflows, which after being expanded into a concrete workflow is kept as **ConcreteWorkflow**. During execution, a **ConcreteWorkflowInstance** is generated with references to ParticipantInstances.

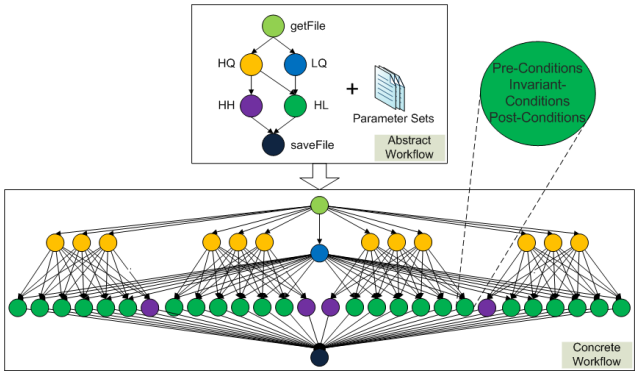

Figure 4. Transformation from parameterized to concrete workflow

The analysis of the process execution space in conjunction with the data provenance space allows the recreation of complete execution traces of workflows. The concrete workflows are executed by a simple workflow engine as described in the following sections.

4.2. Generation of Concrete Workflows

A parameterized workflow is a template that defines a range of values for each parameter to be processed in the workflow. For a given input, parameter set, a parameterized workflow generates multiple concrete workflows.

Each participant within the generated concrete workflows contains the set of pre, post and invariant conditions as specified in the data model. Conditions must be and remain true during the execution of the participant. These conditions induce a failure propagation graph, which is a tree with failure conditions as roots and the participant that fails as the leaf. They are causal models that capture the temporal aspects of failure propagation in dynamic systems [2].

Conditions in the abstract workflow are specified using templates that are replaced in the concrete workflows. For example, the any property that needs to be checked over heartbeat must be specified in the data model without knowing the exact nodes used to execute the participant. We use special function $ALL(\mathcal{H}(\theta(P_t)),t) > 0$ to symbolize the conjunction of heartbeat conditions for all nodes that are included in the task allocation map.

Generic Modeling Environment [1] is used to create a modeling language for specifying the parameterized workflow. A concrete workflow is generated by replicating a given participant and reevaluating the dependency condition for all valuations of input parameters.

Fig. 4 illustrates generation of a concrete workflow for a two-point analysis campaign. A simple configuration for a single gauge file and the following arrays of physics param-

eters: kappa== $[1 \cdot \cdot 4]$, wsrc= $[1 \cdot \cdot 3]$, mass= $[1 \cdot \cdot 2]$,and d1= $[1 \cdot \cdot 2]$.

In the example, the HQ participant is expanded into 12 instances according to the number of kappa and wsrc parameters (instances are grouped by kappa values). The concrete workflow contains a single LQ participant that generates files, which are then combined with HQ outputs by 24 instances of HL. Finally the HH participant combines HQ outputs for a given kappa value. Outputs of HH and HL are the final product. A production level concrete workflow for 1000 gauge configurations and default physics parameters yields approximately 40K participants.

4.3. Workflow Engine

Concrete workflows generated by the Generic Modeling Environment derived from parameterized workflows in conjunction with input parameter sets are submitted to the execution engine. The concrete workflows are represented as Direct Acyclic Graphs (DAGs) $G = (V, E)$, where the set of vertices V represents the set of participants generated based on the input physics parameters and the set of directed edges E represents the data dependencies between participants.

This simple graph representation allows an execution engine to extract full parallelism from LQCD analysis campaign workflows, which is a shortcoming encountered when modeling the same workflow using Askalon [7]. An execution engine such as DAGMan [11] with modifications suffice for running concrete workflows.

While traversing the graph G the workflow engine interacts with the workflow database (created from the data model previously described), to retrieve participant information, record execution participant and workflow execution information, and save data provenance.

5. Runtime Framework

The run time framework, composed by the integration of the workflow, monitoring and mitigation frameworks is depicted in Fig. 5. On the left side, the workflow execution engine schedules participants as dependencies are met. The remaining components on the right side are the monitoring and mitigation framework. The workflow execution engine and the global controller run on the cluster head node, regional managers are assigned to each rack head node, while local managers run on remaining worker nodes.

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 the concrete workflow 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 on the preconditions results into a message back to the workflow engine informing the violation. This message back to workflow engine is a mitigation strategy and is specified in the mitigation reflex engine module at the global controller level.

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 following section.

5.1. Example of a 2-point Analysis Workflow

One of the common LQCD workflows is the analysis campaign. These are 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 2-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 depicted in Fig. 4 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 in the figure. The N outputs form the final campaign output are later 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 HQ generating heavy quark propagators is derived from the val-

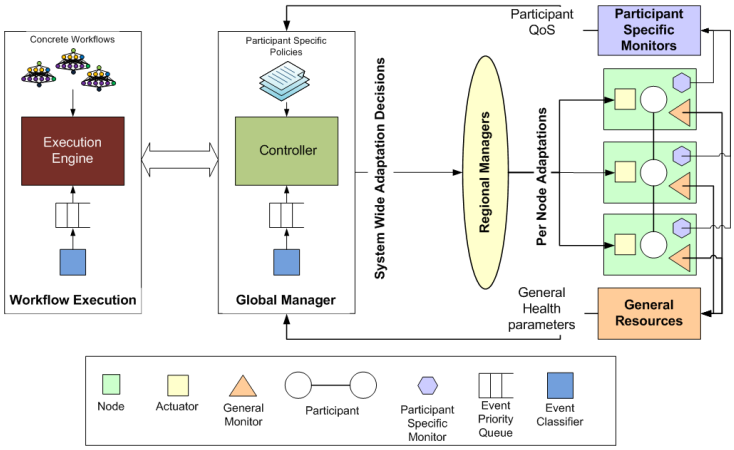

Figure 5. Complete runtime framework with integrated workflow, monitoring and mitigation integration

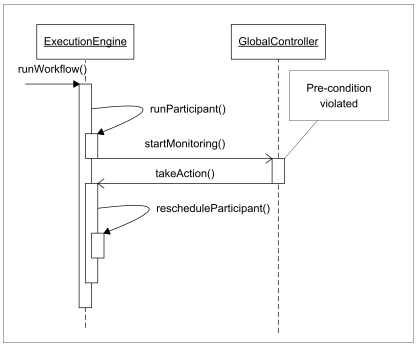

Figure 6. Sequence diagram depicting a pre-condition violation.

ues of physics parameters (kappa and wsrc).

dCache failure scenario: The first participant on the 2-point concrete workflow is getFile. This participant is responsible for fetching the gauge configuration used as input for the LQ and HQ participants. A precondition of getFile is that the dCache pool manager (pm) must be available, therefore $\mathcal{H}(pm, t) > 0$ should be true. In the case of condition violation, as shown in Fig. 6, 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 a Δ_{time}, which is an administrator decided parameter. It 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.

Disk space failure scenario: The second level of participants on the 2-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: $\mathcal{D}(n,'\text{DISK}') > 10000$.

Before the participant starts running, the local disk sensor is checked to make sure enough space is available, according to the first step in Fig. 7. If the condition is violated an event is created. In this case the mitigation action can be provided by the local reflex engine, for example by deleting files from the temporary disk area. In case the action does not succeed, an action by the workflow engine is requested.

Allocated node failure scenario: By definition, 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 stipulated contractual wall time[7].

Infiniband failure scenario: Most LQCD participants are parallel jobs that heavily use a dedicated infiniband network for exchanging data during iterations. It is essential that the network remains available throughout the participant execution time. This need can be expressed as an invariant condition of the type: $\mathcal{I}(n, t) > 0$, with possible values of $\{0, 1\}$.

If the invariant condition is violated, the local fault mitigation is enabled, as in Fig. 7. After the mitigation, the preconditions are evaluated again. If the violation persists, an event is sent to the workflow execution engine. Similarly to Fig. 6, the action resulting from the infiniband failure is

[7]Every job specified a WCET. It is terminated, if it is still executing after passage of that time.

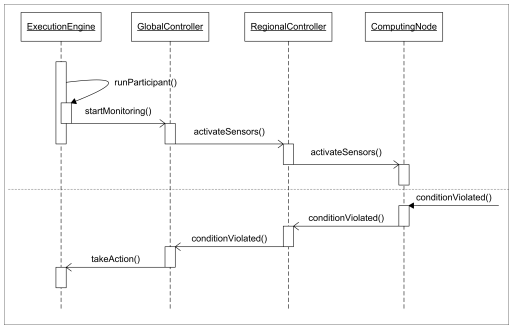

Figure 7. Sequence diagram depicting a precondition violation triggering local fault mitigation.

to have the whole participant rescheduled.

6. Conclusions and Future Work

In this paper we presented a framework to execute scientific workflows reliably. Essential run time information is constantly verified by the monitoring framework, while conditions pertinent to the running workflow are enabled only during execution time. Additionally, the conditions are specified at the participant granularity, avoiding overzealous monitoring and consequent use of computing resources that would otherwise be available for the scientific applications. We have developed a prototype of the proposed architecture used to demonstrate feasibility for fault-tolerant LQCD workflows. Many workflow systems currently lack fault tolerant features, which is one of the top priorities for large scale long time running workflows. 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, and replacement of the current workflow engine. Further tests are planned on a virtual environment where failure scenarios can be simulated and repeated.

7. Acknowledgments

This work was supported in part by Fermi National Accelerator Laboratory, operated by Fermi Research Alliance, LLC under contract No. DE-AC02-07CH11359 with the United States Department of Energy (DoE), and by DoE SciDAC program under the contract No. DOE DE-FC02-06 ER41442.

References

[1] A. B. e. a. A. Ledeczi, M. Maroti. Generic modeling envrionment. In *WISP*, volume IEEE International Workshop on Intelligent Signal Processing, Budapest, Hungary, May 2001.

[2] S. Abdelwahed, G. Karsai, and G. Biswas. Notions of diagnosability for timed failure propagation graphs. In *Proceeding of IEEE Systems Readiness Technology Conference, AUTOTESTCON*, 2006.

[3] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[4] G. Behrmann, P. Fuhrmann, M. Grønager, and J. Kleist. A distributed storage system with dcache. *Journal of Physics: Conference Series*, 119(6):062014 (10pp), 2008.

[5] A. Dubey, S. Nordstrom, T. Keskinpala, S. Neema, T. Bapty, and G. Karsai. Towards a verifiable real-time, autonomic, fault mitigation framework for large scale real-time systems. *Innovations in Systems and Software Engineering*, 3:33–52, March 2007.

[6] A. Dubey, S. Nordstrom, T. Keskinpala, S. Neema, T. Bapty, and G. Karsai. Towards a model-based autonomic reliability framework for computing clusters. In *EASE '08*, pages 75–85, 2008.

[7] T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, and M. Wieczorek. Askalon: A grid application development and computing environment. In *GRID '05: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 122–131, Washington, DC, USA, 2005. IEEE Computer Society.

[8] T. Fahringer, J. Qin, and S. Hainzer. Specification of grid workflow applications with agwl: an abstract grid workflow language. *Cluster Computing and the Grid, IEEE International Symposium on*, 2:676–685, 2005.

[9] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real time systems. *Information and Computation*, 111(2):193–244, 1994.

[10] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University press, 2 edition, 2000.

[11] G. Malewicz, I. Foster, A. L. Rosenberg, and M. Wilde. A tool for prioritizing dagman jobs and its evaluation. *Journal of Grid Computing*, 5(2):197–212, 2007.

[12] L. Piccoli, J. B. Kowalkowski, J. N. Simone, X.-H. Sun, D. J. Holmgren, N. Seenu, A. G. Singh, and H. Jin. Lattice qcd workflows: A case study. In *SWBES '08*, Indianapolis, IN, USA, 2008.

[13] R. Sterritt. Autonomic computing. *Innovations in Systems and Software Engineering*, 1(1):79–88, April 2005.

[14] I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields. *Workflows for e-Science: Scientific Workflows for Grids*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[15] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. V. Laszewski, I. Raicu, T. Stef-praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. *services*, 00:199–206, 2007.