ELECTRICAL ENGINEERING

USING MODEL-BASED TECHNIQUES FOR IMPROVING PERFORMANCE AND RELIABILITY IN HIGH PERFORMANCE SCIENTIFIC COMPUTING

ABHISHEK DUBEY

Dissertation under the direction of Professor Gabor Karsai

Data processing in scientific and workflow-oriented computing is carried out as analysis campaigns, which consist of an input dataset and a set of interdependent jobs. Traditionally, these massively parallel computations required the services of supercomputers. However, recent trends show that the share of scientific computing carried out on clusters of commodity computers is on the rise. Commodity computers yield the highest performance per dollar but exhibit intermittent faults, which can result in systemic failures when operated over long continuous periods for executing analysis campaigns. Diagnosing job problems and failures in this complex environment is difficult, especially when the success of a campaign can be affected by even a single job failure. Manual administration, though essential, is slow to respond to the intermittent faults. Therefore, an autonomic approach is required that can ensure that the resources of the cluster are used to the best possible extent and improve the reliability of jobs, even in the presence of hardware/software failures.

Model-based design is a formal system design methodology that has gained momentum in recent years as a sound methodology of applying computer-based modeling and synthesis methods to a variety of problem domains, including distributed systems. A benefit of using formal models is that they can be queried or transformed to produce a variety of domain specific artifacts, which are critical to deployment and execution of the system, but are tedious and error-prone to produce manually.

This dissertation presents the design and discusses applicability of a model-based cluster management framework called Scientific Computing Autonomic Reliability Framework (SCARF). Basic components of this framework are distributed monitoring units, fault-mitigation units and a workflow-management system for dealing with workflow-specific concerns in case of failures.

Model-based techniques are used to capture workflow specifications, along with pre, post conditions and invariants for checking the validity of system state during execution. Formal data models are used to provide provenance and execution tracking of workflow jobs. Health monitoring is provided by synchronized, light-weight, distributed sensors that are augmented with a real-time fault-mitigation framework. This framework consists of hierarchical fault management entities called reflex engines, which use a timed automaton based abstraction for capturing failure management strategies. These engines track the state of components under their management zone and initiate reflexive mitigation actions upon occurrence of certain events or timeouts. This mitigation framework is verified against properties written in timed computation tree logic (TCTL).

Approved ______________________________ Date __________________

USING MODEL-BASED TECHNIQUES FOR IMPROVING PERFORMANCE AND RELIABILITY IN HIGH PERFORMANCE SCIENTIFIC COMPUTING

By

Abhishek Dubey

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Electrical Engineering

May, 2009

Nashville, Tennessee

Approved:

Professor Gabor Karsai
Department of Electrical Engineering and Computer Science
Vanderbilt University

Research Associate Professor Theodore Bapty
Department of Electrical Engineering and Computer Science
Vanderbilt University

Research Assistant Professor Sandeep Neema
Department of Electrical Engineering and Computer Science
Vanderbilt University

Assistant Professor Sherif Abdelwahed
Department of Electrical and Computer Engineering
Mississippi State University

Professor Paul Sheldon
Department of Physics and Astronomy
Vanderbilt University

Dedicated to my father, Dr. Anil Kumar Dubey.

ACKNOWLEDGMENTS

This research was supported in part by the Department of Energy grant number DE-FC02-06ER41447 and NSF under the ITR grant ACI-0121658. I am indebted to the help and guidance of my advisor Prof. Gabor Karsai. To my project advisors, Dr. Ted Bapty, Dr. Sandeep Neema, and Prof. Sherif Abdelwahed I offer my sincerest gratitude for your close collaboration and tutelage. I also thank Dr. Paul Sheldon, who provided great insights and support during the course of this work.

I have had the privilege of collaborating with a number of people at Fermi National Accelerator Laboratory including James Kowalkowski, James Simone, Don Holmgren, Luciano Piccoli, Amitoj Singh and Nirmal Seenu. They all have been an integral part of this project and their guidance and efforts are greatly appreciated.

During my years at Vanderbilt University, I have had the opportunity of working closely with a number of friends including Steve Nordstrom, Turker Keskinpala, Manish Kushwaha, and Daniel Balasubramanian. I am thankful to them for a number of great discussions and suggestions. I would especially like to thank Joe Porter for his help and suggestions with the presentation of material.

I thank my mother, Mamta Dubey and my sister Arpita Dubey for their encouragement. To my wife, Di Yao who helped me improve this manuscript and has been a constant source of support, I give my deepest devotion and gratitude. Finally, to my father-in-law, Songyi Yao and my mother-in-law, Hong Zhai, thank you for making Nashville a home away from home.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS iii

LIST OF TABLES viii

LIST OF FIGURES ix

Chapter

I. INTRODUCTION 1
- Scope and Organization of Dissertation 2

II. BACKGROUND AND CHALLENGES 5
- Lattice QCD Workflows 6
- Computing Infrastructure 8
  - Failure Propagation 9
- Summary of Challenges 11

III. MANAGING COMPUTING SYSTEMS 13
- Management from Dependability Aspect 13
- Fault-Tolerance via Design Diversity 14
  - Recovery Blocks 15
  - N-Version Programming 15
  - N self-checking Programming 16
- Recovery Oriented Computing 17
- Autonomic Computing 20
- An Architectural Outlook on Autonomic Computing 22
- Autonomic Computing and Artificial Intelligence 23
  - Prediction/Optimization 23
  - Planning 26
  - Reinforcement Learning 28
- Autonomic Computing and Feedback Control Theory 32
  - Controlling Resources 33
- Autonomic Computing and Formal Techniques 35
  - Self-Management Using Architectural Models 36
  - Model-Based Approach to Reactive Fault-Aware Systems 41
- Software Engineering Concepts For Autonomic Computing 44
- Engineering for Autonomic Computing Systems 46
  - Grid computing 48
  - Application of Autonomic Computing in Clusters 52
- Summary and Discussion 53

IV. TOWARDS A VERIFIABLE REAL-TIME, AUTONOMIC, FAULT MITIGATION FRAMEWORK FOR LARGE SCALE REAL-TIME SYSTEMS 55
Overview 55
Abstract 56
Introduction and Problem Motivation 56
Fault Diagnosis and Mitigation 59
Large Scale Real Time Systems 60
Abstracting Real-Time Systems as Discrete Event Systems 61
The Reflex and Healing Architecture Using a Discrete Event Model . . 62
Abstracting Real-Time Systems As Timed Automatons 64
The Real-Time Reflex and Healing Framework 69
Real-time Fault Mitigation Strategies and Timer Tasks 70
Buffer 73
Scheduler 74
Real-Time Reflex Engine 76
Sequence of Operations inside a Real-Time Reflex Engine 77
Analysis of Real-Time Reflex and Healing Framework 78
Case Study 79
Experiment Setup 80
Simulation 91
Analysis and Verification 91
Evaluation 96
Conclusion and Future Works 96

V. DESIGNING A LIGHT-WEIGHT SYNCHRONIZED DISTRIBUTED MONITORING FRAMEWORK FOR COMPUTING CLUSTERS 98
Overview 98
Abstract 99
Introduction 99
An Introduction to SCARF 103
Sensors 105
Sensor Scheduler 106
Problems caused due to jitter 108
Impact of sensor synchronization on application performance 109
Solution Approach 111
Controller Design 112
Plant and Controller Model 112
Feedback Controller 112
Transfer Function of the Control Loop 114
Steady State Error Analysis 116
Modified Sensor Scheduler with Synchronization and Feedback Controller . . 117
Results 119
Performance of feedback controller in controlling jitter. 119
Jitter with and without controller with changing CPU Load 119

Effect of Current Priority on Total Jitter 121
Controlling Jitter of Multiple Periodic Processes 121
Experiment4: Controlling Jitter of a Periodic Application in Windows XP 123
Performance of MILC Job- synchronization problem 123
Related Research 125
Feedback Control in Computing Systems 127
Conclusion and Future works 128
Acknowledgments 128

VI. ON THE DESIGN OF A FRAMEWORK TO ENABLE RELIABLE SCIENTIFIC WORKFLOWS 129

Overview 129
Abstract 130
Introduction and Motivation 131
Preliminaries 133
Model of Time 134
Resources 134
Computation Nodes N 135
Participants p 135
Workflows W 136
Workflow execution engine 137
Task Allocation Map 137
Sensors and Filters 138
Events and Conditions 140
Runtime Checkable Properties 141
Distributed Monitoring and Mitigation Framework 143
Reflex Engine architecture on a computation Node 143
Monitors: Checking for Satisfaction of a Property 145
Hierarchy of Managers 147
Workflow Framework 148
Provenance Model 148
Generation of Concrete Workflows 150
Implementation of Workflow Engine and integrating it with reflex engines . . . 151
Execution engine 153
Integration with Reflex Engines 154
Implementation of a Prototype 155
Results and Case study 156
Dcache failure mitigation scenario 158
Disk failure mitigation scenario 158
Computation Node Failure 159
Discussion on Model Predictive Recovery of Workflows 159
Conclusions and Future Work 160
Acknowledgments 162

VII. CONCLUSIONS AND FUTURE WORK 163
Future Research Goals 165

BIBLIOGRAPHY 166

LIST OF TABLES

Table Page

1. FNAL LQCD clusters . . . 8

2. A brief list of issues affecting reliability in Clusters. . . . 10

3. Selection of prediction algorithms as presented in [1]. . . . 24

4. An example Client/Server Architectural Style Definition by David Garlan. . . . 38

5. Control Program written in Reactive Model Based Programming language . . . 43

6. A list of Other Autonomic Research Projects . . . 49

7. List of all the events being used in the case study. . . . 81

8. FNAL LQCD clusters . . . 100

9. Sensors in the framework . . . 106

10. Mean, Variance and Root Mean Square values of Total Jitter in seconds for 8 processes referred in figure 53. . . . 122

11. Cluster Evaluation Metrics. . . . 132

12. Sensors present in our framework . . . 138

LIST OF FIGURES

Figure Page

1. Analysis campaign for a configuration file. Dataflow is shown by arrows. 7

2. Man-made and physical faults 14

3. ROC Brick 19

4. Levels of autonomic computing 21

5. Adaptation Cycle 22

6. Network attached storage used in Clockwork 25

7. Data center architecture used in [2] 30

8. Results of using an offline RL policy compared to the initial online policies. . . . 31

9. Model-based self management as proposed by Rohr et al. in [3] 37

10. Rainbow Framework 39

11. A Model Based Executive 42

12. An example state transition model of a plant: a simplified space craft 42

13. The seven stages of soft systems methodology. 46

14. Automate Architecture Diagram. 50

15. A typical Component in Automate. 50

16. Usual approach to fault diagnosis and mitigation. 59

17. Hierarchical fault management scheme in reflex and healing architecture. The reflex actions are user defined FSM. Communication between managers themselves and to/from the plant is via predefined events. 62

18. A timed automaton model of a behavior of traffic light. 66

19. A network of two timed automatons. 67

20. The previous reflex engine has been augmented to allow analysis within the semantic domain of networks of timed automatons. 68

21. A computing node in the real time reflex and healing framework. 69

22. Timer timed automaton. 71

23. An example strategy and its corresponding timed automaton. 72

24. Buffer timed automaton listening to two different events. More events can be added to the model by creating new self-transitions. The event, *dequeue*, is used to pop the event from the front of the queue. 74

25. A scheduler timed automaton, which has only one thread. The select state is the one in which the scheduling decision is made. 75

26. An overview of a real-time reflex engine 76

27. Overview of the chain of operations inside a computing node, which has a real time reflex engine. 77

28. The architectural setup for the case study. There are two local managers, one data source, two filter applications and one regional manager. 81

29. The data source generates data every four time periods. The number of data events generated varies over periods. 83

30. The filter buffer stores the generated data source events. 83

31. There are two filters in this setup. Both filters consume the data from the filter buffer and detect whether it is a good physics event or a bad physics event. . . . 84

32. The buffer of one of the local managers. 85

33. The local scheduler dequeues the local buffer and starts the appropriate strategy. This scheduler only has one thread. Therefore, only one strategy can be executed at a time. 86

34. The local strategy. The local scheduler starts this strategy. 86

35. The regional buffer stores the events communicated by its local managers. 87

36. The regional scheduler also works like the local scheduler. It consumes events from the regional buffer and starts the appropriate regional strategy. 88

37. This regional strategy issues a mitigating action if contiguous bad data events arrive before the timer event arrives. This mitigating action changes the parameters of the data source effectively reducing the number of physics events generated per cycle. 88

38. This timer is used by the regional strategy as a stopwatch set for a period of 2s to measure time elapsed between two bad data. It is necessary to use the timer because the regional strategy relinquishes the thread between the two contiguous arrivals of bad data. 89

39. A simulation of a temporal fault and its mitigation is produced by UPPAAL. . . 90

40. The regional strategy ignores the second false alarm by transitioning to sleep state. 95

41. Overview of SCARF 101

42. A plot of 99 samples of timing jitter obtained for a task with 1 second period on Red Hat Linux 2.6.9-67. 102

43. Implementing the reliability framework using hierarchical reflex engines. 104

44. Dataflow of messages exchanged between managers 105

45. Periodic sensing will accumulate delay when used in a standard OS. 109

46. Loss in the performance of a MILC job when run in the presence of unsynchronized sensor schedulers across the cluster. 110

47. Relationship between a hyperperiod and ticks. 111

48. The feedback control loop for the sensor scheduler 113

49. Total jitter accumulated in last 9 hours. 119

50. Improved performance of a MILC job with synchronized sensor scheduler. . . . 120

51. Plot of total Jitter with varying CPU Load. The controller was switched on at 155th second (dotted line). 120

52. Total jitter plotted over time for three different priorities. Note priority 0 is the highest priority among the three. 121

53. Plot of Total Jitter for 8 different periodic processes with the same time period of 5 seconds. Each process is being controlled by its own controller 122

54. Plot of Total Jitter with 100% and regular CPU load on a windows machine. The controller was switched on at 95th second (dotted line). 123

55. Time take during each iteration. 124

56. Box and Whisker plot showing the spread of synchronization error across 19 nodes across all iterations. The mean synchronization error on each node was approximately 10 msec. 124

57. Sample workflow definition 137

58. Combination of a heartbeat generator and heartbeat filter 140

59. A computing node in the real time reflex and healing framework. 144

60. A reflex engine monitoring the heartbeats for a pool node. The node diagnoser is a reflex engine that attempts to revive the node and marks it offline upon failure. 146

61. Hierarchical reflex engines 147

62. Transformation from parameterized to concrete workflow. kappa== $[1 \cdot \cdot 4]$, wsrc= $[1 \cdot \cdot 3]$, mass= $[1 \cdot \cdot 2]$,and d1= $[1]$. 152

63. A different concrete workflow derived from the abstract workflow of Figure 62 but different parameters. kappa== $[1]$, wsrc= $[1 \cdot \cdot 3]$, mass= $[1 \cdot \cdot 2]$,and d1= $[1]$. 152

64. Execution engines maintains a ready queue. 153

65. Complete runtime framework with integrated workflow, monitoring and mitigation integration 154

66. Two-point analysis workflow for one configuration file 156

67. Sequence diagram depicting a dcache violation 158

68. Sequence of events starting from scheduling of participant to marking the node offline. Heartbeat timeout was 300 seconds 160

CHAPTER I

INTRODUCTION

The last three decades have seen rapid advancements in distributed computing. While the last decade of twentieth century primarily saw the rise of personal desktop computing, we are now seeing information technology moving towards utility-based computing supported by large clusters of commodity computers organized as a cloud or loosely coupled together in a grid [4]. This model of infrastructure has not only enabled several end user services such as email, and e-commerce etc. but has also made cheap high performance computing possible. Coupled with the developments in storage technology, scientific research in areas such as biology, disaster simulation, and physics among others that traditionally required more expensive supercomputers is becoming financially more accessible [1] to researchers. Organization and reliable processing of the massive information produced is critical for its effective use in new discoveries.

The advantage of using a large number of commodity computers for performing research lies in the ubiquity of their components - commodity computers interconnected using high-speed networks such as Myrinet [5] and Infiniband [6]. However, the problem is that these computers exhibit intermittent faults, which can result in systemic failures when operated over long periods [7]. We need an infrastructure used in high performance computing to be dependable.

Laprie defines computing system dependability as the quality of delivered service such that reliance can be justifiably placed on that service [8]. In his definition, he has considered service to be the behavior of the computing system as perceived by another client system interacting with the considered system. Actually, any factor (not necessarily a failure) that results in a decrease in delivered quality of service is an impediment to dependability. These factors can range from environmental catastrophes to hardware problems to software problems.

[1] This is measured in FLOPS per USD

Traditionally, researchers have focused on building dependable hardware [9], but as hardware reliability improves, now more effort is being placed on software and network security [4]. According to Oppenheimer and Patterson [10] as well as Jim Gray [7], most of the faults in internet services systems are operator faults or man-made faults. However, when investigating software failure, one must not forget the interaction between hardware, human-operators and software: disk failure might cause a software stack to crash. Unaccepted sequence of human inputs can put an operating system in an unstable state.

Scope and Organization of Dissertation

Given the complexity of maintaining dependability in a large computing infrastructure, we require systematic, model-based techniques to monitor system health and take self-corrective mitigation actions as close to the source of problem as possible. Moreover, we require this infrastructure to be aware of the software's intent i.e. the workflow of tasks. We will review the challenges faced in managing large computing cluster with a focus on the computing infrastructure that has motivated this research in chapter II. Chapter III presents an overview of recent and seminal research in the areas of *autonomic computing* and *dependable computing* as candidates to provide new ideas and methods for managing computing clusters.

This dissertation describes the design of Scientific Computing Autonomic Reliability Framework (SCARF) with a focus on (a) fault mitigation framework, (b) health monitoring components, and (c) integration of reliability with workflow management.

Chapter IV describes the design of a distributed fault-mitigation framework called real-time reflex and healing. The design of appropriate mitigation actions depend upon the goals and state of the application and environment. Strict time deadlines in real-time systems exacerbate this problem. Any mitigation behavior in such systems must not only be functionally correct but should also conform to properties of liveness, safety and bounded time responsiveness. The work presented in this chapter details a real-time fault-tolerant framework, which uses a reflex and healing architecture to provide fault mitigation capabilities for large-scale real-time systems. At the heart

of this architecture is a real-time reflex engine that has a state-based failure management logic that can respond to both event- and time-based triggers. This chapter also presents a methodology for verifying properties of systems, which use this framework of real-time reflex engines. Lastly, a case study is presented, which examines the details of such an approach. This manuscript has been published in the journal, Innovations in System and Software Engineering [11]. This work is an extension of the reflex framework presented by Nordstrom et al. in [12]. Portions of this work were also presented in third IEEE International Conference and Workshop on the Engineering of Autonomic Systems (EASe 2006) [13].

Chapter V describes the monitoring framework used in SCARF. It also discusses the challenges faced by autonomic reliability frameworks in cluster environments such as non-determinism in task scheduling in general-purpose operating systems like Linux and the need for synchronized execution of monitoring sensors across the cluster. Additionally, it presents a solution to these problems in the context of our framework, which utilizes a feedback controller based approach to compensate for the scheduling jitter in non real-time operating systems. Finally, we present experimental data that illustrates the effectiveness of our approach. This manuscript is currently under review with Letters of IEEE Task Force on Autonomic and Autonomous Systems [14]. Portions of this paper have been presented at fifth IEEE International Conference and Workshop on the Engineering of Autonomic Systems (EASe 2008) [15] and twelfth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2009 [16].

Chapter VI describes the design of a scientific workflow execution framework that integrates run-time verification to monitor its execution and check it against a set of formal specifications. Lattice Quantum Chromodynamics (LQCD) workflows have been used as motivating examples. These workflows produce large amounts of data resulting from numerical simulations of QCD quantum field theory. Configuration files generated through Monte Carlo simulations are later used for computing certain physics quantities such as decay rates and particle masses. Existing workflow systems can be used to help standardize workflow descriptions, but they lack tracking of domain specific information. Additionally, queries on provenance metadata require domain

scientists to learn specific languages. This manuscript is currently under review with Journal of Cluster Computing [17]. It is an extension of the work presented at fourth IEEE International Conference on e-Science, 2008 [18] and sixth IEEE International Conference and Workshop on the Engineering of Autonomic Systems (EASe 2009) [19]. Finally, conclusions and future works are presented in chapter VII.

CHAPTER II

BACKGROUND AND CHALLENGES

Reduced cost of commodity computers and the advent of high capacity networks have made cluster computing economical. Clusters are used for solving complex problems in areas such as biology, disaster simulation, and physics among others that traditionally required the use of a supercomputer [20, 21]. Their advantage lies in the ubiquity of their components - commodity computers interconnected using high-speed networks such as Myrinet [5] and Infiniband [6]. This in turn delivers high performance computing at a fraction of price associated with supercomputers. However, only applications that can be parallelized by splitting into a number of smaller job-units can truly reap their benefits.

This chapter will introduce the complexities involved in supporting 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. Its calculations allow us to understand the results of particle and nuclear physics experiments in terms of QCD. LQCD is both computation and data intensive, and it is representative of large scale scientific computing.

Applications, software libraries, input data and workflow[2] recipes are shared among collaborators worldwide[3]. Typical routine of a scientist running a series of LQCD computations can be roughly described by the following steps: copy previous workflow script; change parameters in the file (e.g. quark masses); run the modified script; check periodically whether jobs are running; in case of an error look for the cause in the several produced log files; fix problem and repeat until all the processing succeeds.

[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/

As new problems are detected, the script is modified to perform additional checking and run recovery procedures. This cycle repeats until a stable version has been achieved (based on a predefined convergence criteria), which is later constantly copied and modified with new application parameters. Despite the ability to produce science, this methodology lacks fundamental aspects that allow scientists to devote most of their time to make science instead of constantly fixing and monitoring workflow scripts.

Groups of scientist within the LQCD collaboration use different tools and methods to coordinate the workflows. Perl, Python or shell scripts are used for invoking the sequence of MPI-based applications; input parameters are kept within the scripts or even hardcoded in the applications themselves; and generated files are stored in a shared area and saved to tape as needed. Therefore, one of the challenges facing a reliable cluster management system is to **standardize the scripts and techniques used for deploying workflows**. Next section provides an overview of LQCD workflows.

Lattice QCD Workflows

Typically two types of workflows are used in Lattice QCD studies: (i) configuration generations, and (ii) analysis campaigns. The first is used for creating a group of ordered configuration of gluons for a given set of physics parameters such as lattice spacing and masses via Monte Carlo simulations. These configurations are collectively called an ensemble. During configuration generation, the gluon configurations are iteratively evolved until a convergence criterion is met. Snapshots of configurations are saved during this entire process. An ensemble can have fork if more than one sequence of Monte Carlo evolution was started.

Configuration generation is done in two phases: tuning and production. Tuning is completed when input parameters and output values reach predefined convergence criteria. Production is repeated for a number of user defined steps. During configuration generation, meta data is stored along with other output files. These workflows account for forty percent of total time used for LQCD.

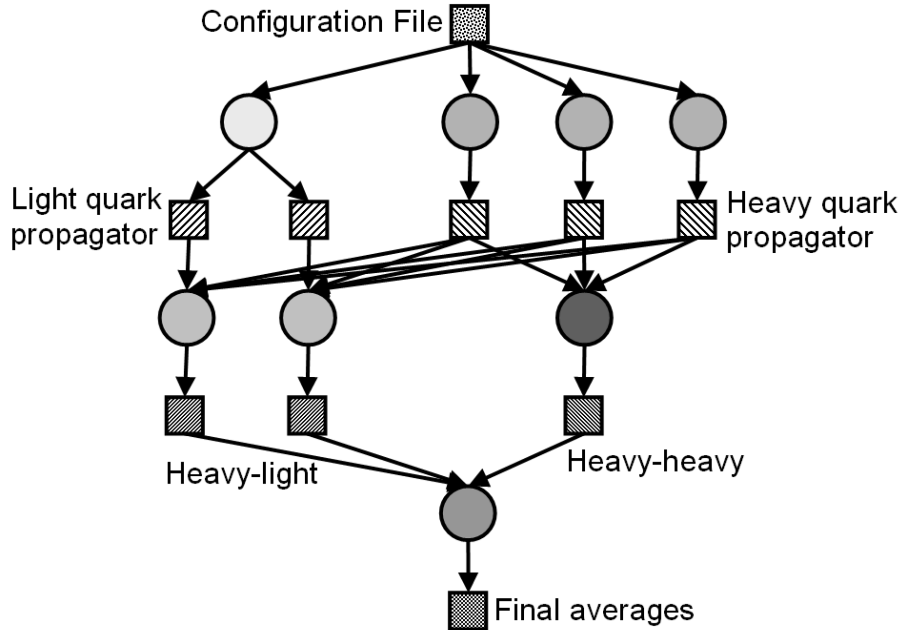


Figure 1: Analysis campaign for a configuration file. Dataflow is shown by arrows.

Analysis campaigns use the ensemble of gauge configurations generated by configuration generation. Their aim is to solve a set of calculations over a predefined lattice for finding a set of physics quantities as output. For example, a 2-pt analysis campaign predicts the mass of decay constant of a specific particle. A typical campaign generates intermediate data products (output of intermediate steps) such as quark propagators and compute meson n-point functions for all configurations. Computation of one configuration is independent of the other.

Figure 1 shows an analysis campaign for a single configuration file of an ensemble. The complete workflow consists of a number of independent instances, say n, of the example in the figure. The n outputs from these instances form the final campaign result. One of the properties of analysis campaigns is that the number of participants and output change depending on the number of input parameters. We will see an example of this in chapter VI.

Calculations involved in these workflows are tightly coupled parallel codes, requiring a high-speed low-latency network interconnection. Therefore configuration generation and analysis campaigns require dedicated hardware. National laboratories such as Fermi Lab maintain commodity clusters for processing LQCD workflows. Most of these systems use PBS batch system with Maui

scheduler [22]. LQCD application code based on USQCD software stack is currently being optimized as part of the SCIDAC -2 initiatives [23]. The codes, MILC and CHROMA are available at http://www.usqcd.org/usqcd-software.

Table 1: FNAL LQCD clusters

Cluster	CPU	Nodes	Performance
QCD	2.8 GHz Pentium 4	127	1400 MFlops/node
PION	3.2 GHz Xeon	518	1729 MFlops/node
KAON	2.0 GHz Dual Opteron	600	3832 MFlops/node
JPSI	2.1 GHz Dual CPU Quad Core Opteron	586	10061 MFlops/node

Computing Infrastructure

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, a national physics laboratory located in Batavia IL, majority of the computers that are used for LQCD computations are dedicated clusters (see table 1). Use of dedicated clusters allows 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.

A typical workflow can spawn hundreds of jobs while executing. 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, isolating faults and mitigation becomes critical, specifically when the success of whole workflow might be affected by even one job failure.

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

Failure Propagation

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. A list of examples of problems that can affect availability of nodes for executing jobs and their performance is provided in table 2.

These failures are separated into different layers: hardware, operating system, MPI middleware, workflow management system and the end user. Not all failures are primary. Some of them such as communication timeout in MPI layer can be either due to MPI stack problem (primary failure) or due to a faulty network card (secondary failure).

Typically, grid workflows have limited application monitoring; however, they provide fault-tolerance by using redundancy. Several sites replicate the same piece of work. This is especially evident in the protein folding at home[5] and SETI at home[6] workflows. 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 understand the complexity of failure propagation, consider large systems that are system of systems. They are composed of components, giving them a recursive structure that has potentially several levels of depth and several interfaces. Each component in this recursive structure has its own set of specified behaviors, which are correct. A component suffers a failure, when it loses its ability to perform the required function. The cause of a failure is a fault, which is defined as a defect or flaw in the hardware or the software. A fault may sometimes manifest itself as a discrepancy which can potentially lead to a failure. At other times the fault may be latent and unobservable and may still lead to a failure. In connected components a fault in a primary component may lead to a failure, which in turn might cause secondary failures in the other component. The goal of a fault-tolerant system is to stop the propagation of faults into failure and if failures do happen then recover from those failures.

[5]http://folding.stanford.edu/
[6]http://setiathome.ssl.berkeley.edu/

Rise in complexity due the numerous hardware and software interfaces makes faults and failures inevitable. It has been argued by Brown and Patterson in [24] that heterogeneity and complexity involved in most large-scale systems lead to unforeseen failures and it is more effective to try and reduce the mean time to recover compared to mean time to failure.

Table 2: A brief list of issues affecting reliability in Clusters.

Hardware	**OS**	**MPI Middleware**	**Jobs**	**Workflow Orchestrator**	**User**
Machine Unavailable	Memory Capacity Exceeded	Communication TimeOut	Memory Leak	Authentication Failure	User defined Exceptions
Network Link Unavailable	Out of Disk Space		Deadlock	Job Submission Queue Capacity Exceeded	Time taken in problem diagnosis
Corrupt Memory	File Not Found		LiveLock	Job Hanging in the Ready Queue	Time taken in problem mitigation
Rise in CPU Temperature	Network Timeout		Incorrect Input Data	Job Execution Time Limit Exceeded	
Power Outage	CPU Time Limit Exceeded		Incorrect Output Data	Job Scheduled on a fault node	
			Missing Shared Libraries	Blocked Workflow	
			Communication TimeOut	Data Movement Failed	
			Missing Shared Libraries		
			Data Stagein Failure		
			Data Stageout Failure		
			Job Crash		
			Input Data Not Available		

Summary of Challenges

Scientific workflows such as LQCD are data intensive and long running applications. Manual administration of large computing clusters, 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. The expected properties of an autonomic fault-tolerant framework are:

1. This framework should provide a standardized workflow specification language. A workflow management system that enables reliable execution with tracking of data items generated during a workflow execution is also required.

2. It should closely monitor performance and the status of job, and work together with the workflow subsystem, ensuring good progress for corresponding analysis campaign.

3. In order to identify faults, this system should contain a light-weight sensor framework that can monitor the health (performance, utilization, state) of all processors and networks in the system without overtly affecting the percentage of available system resources for physics computations.

4. It should support formal specification of properties for each component. Violation of a property will constitute a fault.

5. It should support persistent storage of monitored data so that it is available for diagnosis.

6. It should support automated mitigation policies that can be triggered when a fault is identified. We should be able to verify these policies to ensure consistency and correctness.

The challenges facing such a framework can be summarized into following categories:

Conceptual and Standardization Challenges There is a large body of work on design of autonomic behavior. Some of them are reviewed in the next chapter. In order to develop a clear

and formal structure for autonomic computing to be used in large clusters, appropriate abstractions and models of both resources and applications i.e. workflows, which can help in realizing the possible adaptations from the conceptual design model of the architecture are required.

Scalability Challenges A cluster management framework must be light enough (in terms of memory and CPU utilization) so that it does not become an impediment to the performance of scientific computations by consuming a lot of resources. In other words, the performance of application before and after the framework should be comparable. Furthermore, the framework should be dynamic so that it can be extended online.

Validation Challenges Mitigation policies used in this framework must be verifiable or extensively tested. This is necessary to ensure that the mitigation policy itself does not become a point of failure.

Middleware Challenges A middleware must be available to support transfer of monitoring data and actuation commands in a scalable manner, in spite of the application dynamism. This middleware must also be robust so that it can tolerate node failures. Lastly, it should facilitate discovery of capabilities provided by different autonomic elements that are part of the cluster.

Application Challenges In order to support application fault-tolerance, a cluster management framework should support the workflows to retain the knowledge of the design level models and later use it to monitor themselves and adapt. Alternatively, it should support a workflow execution engine that monitors and change workflows due to failures. We use the latter approach of using a workflow execution engine.

CHAPTER III

MANAGING COMPUTING SYSTEMS

Management from Dependability Aspect

According to Laprie, computer system dependability is the quality of delivered service such that reliance can be justifiably placed on that service [8]. In his definition, he has considered service to be the behavior of the computing system as perceived by another client system interacting with the considered system. In his 1996 article, Johnson described dependability as a set of concepts [25]:

1. Reliability: It is the probability that the system will work correctly in a given time interval, assuming that the system was working correctly at the start of that interval.
2. Availability: Probability that the system is working correctly at a given instant of time.
3. Safety: It is the probability that the system will not lead to a hazardous state, as defined by the specification of the computing system.
4. Performability: It is the probability that the system performance will be greater that some specified level at a given instant of time.
5. Maintainability: It is the probability that the system can be recovered from the failure in a specified time interval.
6. Testability: It is the measure of the ability to characterize the system via testing.

The major concern for the design of dependable systems is to design a fault-tolerant system (both hardware and software) that can maintain some of the above-defined properties in presence of failures. The call for formalization of fault-tolerance as perceived for digital systems is much older than the advent of autonomic computing in 2001. In a 1978 paper [26], Avizienis described

fault-tolerance as the attribute of a digital system that keeps the logic machine doing its set of specified tasks when its hosts, suffers various kinds of failures of its components. He further divided the faults in to two categories (a) Faults due to adverse physical Phenomena, and (b) Man-made faults. He further divided the man-made faults into two categories, (a) Design-time faults, caused by imperfections during phases of implementation of the original specification, or (b) Interaction faults, caused by inputs that are introduced into system via inappropriate inputs of the human operator. He claims that it is the man made faults that has had limited success with fault-avoidance techniques. Figure 2, describes a causal relationship between the man made faults and the physical faults. The arrowhead in the figure directs from cause to effect.

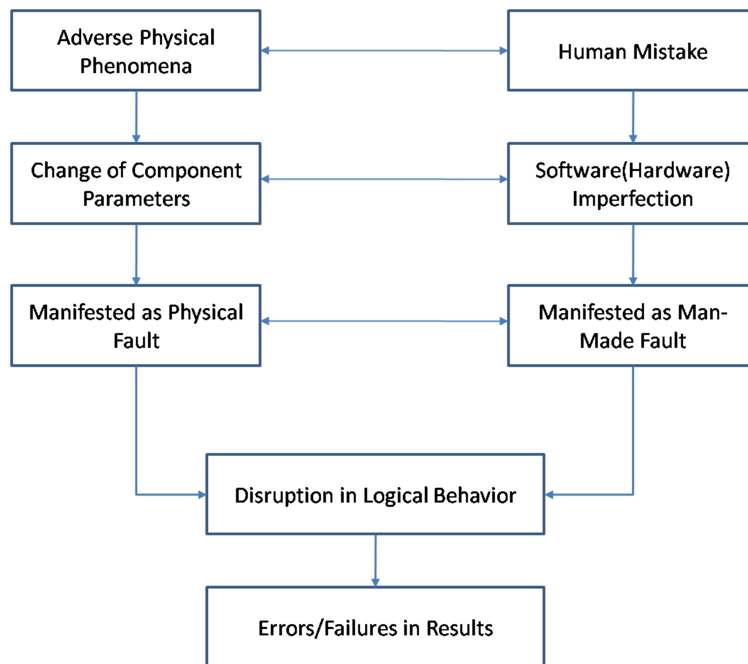


Figure 2: A cause and effect analogy between physical and manmade faults as described by Avizienis in [26].

Fault-Tolerance via Design Diversity

Most of the earlier work on software and hardware fault-tolerance focused strongly on using redundant components and design diversity to tackle design-time faults [9, 27]. In this approach,

the computer system was partitioned in to modules, with each module dealing with a specific sub function. This federated partitioning provided for fault-containment with a set of identical computing processors supporting software for several sub-functions. This approach improved the dependability of the computation system with more efficient use of redundant hardware. However, this still left system vulnerable against the software faults.

In a diversified design, several variants of the same software are used with an acceptance test employed at the end to compare the output from each of the variant. The rationale behind this approach is the expectation that component built differently will fail differently [28]. A diversified design has at least two variants plus a decider, which monitors the outputs of the variants. Three such strategies for software fault-tolerance are, Recovery Block approach [29, 30], N-version programming approach [31], and the N self-checking programming approach [9, 8].

Recovery Blocks

The recovery block technique [29, 30] uses the checkpoint and restart technique to recover from a fault. It uses multiple versions of the same software module as alternates. Upon failure of an acceptance test, the checkpoint state is used to restart the computation by using the next alternate version. In this approach, at least two versions of software module are required. If all alternates fail, the module issues an exception to the rest of the system.

N-Version Programming

In the N-version programming approach, multiple versions of same programs is executed in parallel, with a voter selecting the output most likely to be correct. This approach is different from the recovery-block approach in that it does not require an application dependent voter and that it needs at least three versions of the same module to work. However, the parallel execution necessitates the need to ensure input consistency.

In 1990, Brilliant, Knight and Leveson published results from a large-scale experiment conducted in N-version programming [32]. In the experiments, they prepared twenty seven versions

of a program at two different universities and then executed them one million times. The results of the experiment were intriguing. They noticed that different versions were found to be reliable when used individually with only small number of failure. However, when they correlated these failures across different versions they discovered that the number of input cases when more than one version failed was significantly large. Upon post failure analysis, they further discovered that correlated failures arise from logically-unrelated faults in different parts of the algorithms. They hypothesized that most of the faults resulted from the fundamental flaws in the algorithms that the programmers designed. Therefore, changing the development tools or methods to create new versions did not reduce the number of correlated failures in N-version software.

N self-checking Programming

N Self-checking approach [9, 8] is a combination of Recovery block approach and N-version approach. This approach has two variations. The first variation uses acceptance test for each version as in the recovery block approach, however, the acceptance test for each version can be different. By executing these versions in parallel, this approach enables switching of output in case of errors instead of restarting from previous checkpoint.

The other variant uses a comparison technique. It groups the variants into set of two, with a comparison unit forwarding the result to a selection unit only if the two versions in a set produce identical results. The selection logic then selects the outputs from different version similar to the N-version technique. The drawback of using this technique is the possibility of running into situation where both versions in a set produce identical wrong output.

Three primary issues with these fault-tolerance techniques are:

- Guaranteeing the independence of faults in multiple versions of same software
- Cost associated with executing multiple versions of the same software redundantly
- Developing appropriate output selection algorithms.

Recovery Oriented Computing

David Patterson and Armando Fox and co-workers at Berkeley and Stanford University have advocated a different outlook to the problem of dependable systems. They take the perspective that hardware/software faults, both design and man-made faults are facts to be coped with, not problems to be solved. According to Oppenheimer and Patterson most of the faults in internet services systems are operator faults or man-made faults [10]. They say that in general failures arose when operators made changes to the system - e.g. adding/replacing hardware. They point out that most of the dependable system designers overlook these errors. Their research project, aptly titled Recovery-Oriented Computing (ROC)[1] [33, 34, 24] concentrates on reducing time to recover from faults and thus offer higher availability and aims to reduce total cost of ownership. Primarily, they state that their vision is to investigate novel techniques for building highly dependable internet services. Kephart has stated that ROC is a promising start in the direction of graceful recovery from failures [35] and hence forms a synergistic relationship with autonomic computing.

In [33], Brown and Patterson state that the heterogeneity and complexity involved in most large-scale service systems inherently leads to unforeseen failures. Therefore, in ROC they emphasize the need for testing recovery systems and help make recovery procedures a holistic part of the architecture rather than a patch/add-on that leads to extra complexity. Moreover, they focus on reducing mean time to recover (MTTR) rather than on mean time to failure (MTTF). Based on their hypothesis they have proposed six techniques [24] for recovery oriented computing. These techniques are:

1. Redundancy: Introduce redundancy to reduce the probability of single point of failure. However, this generally adds to the initial setup cost of the system. However, they argue that by increasing the number of available resources as a safety margin, the mean time to recover is reduced that increases availability and hence reduce total cost of ownership.

[1] http://roc.cs.berkeley.edu/

2. Partitioning: They advocate the use of partitioning the system such that to ensure fault-containment and ease of fault-isolation.

3. Testing of recovery mechanisms: They advocate that live system should possess capability of fault insertion to test recovery process. This they argue helps in running availability benchmarks, which allows for reduction in time taken to deduct error.

4. Aid in Diagnosis of the cause of error: They advocate that a ROC system should contain sensors that help an operator determine the problem

5. Logging of operator inputs to enable undo: In [36], Brown and Patterson have stated that a system should have the capability to rewind and replay a set of inputs given by a user to assist in repair. It is a system-wide undo. They have used this technique to build and undo email store [37]. However, since system behavior is not always deterministic, this undo technique can only work for a class of system and for that system, the cost of storing the undo-trace would be enormous.

6. Orthogonal Mechanisms: Fox and Brewer state that one can reduce the likelihood of a complete failure and increase availability by using multiple versions for the same function that are independent from each other.

One of the key examples of ROC implementation is a hardware platform called ROC-1 [38]. It is a 64-node prototype cluster platform for providing reliable internet services. Each node, called brick, in this cluster possesses redundant network interfaces, and a dedicated Motorola MC68376-based diagnostic processor connected to its own private diagnostic network. Figure 3 shows a block-diagram of this prototype brick board.

ROC-1 allowed the diagnostic processor to isolate subsets of nodes to contain faults. Due to the number of distinct Ethernet interfaces, it actually has two distinct routes from each node to other node, providing redundancy in case of failure. The diagnostic processor provided redundancy and isolation of nodes by turning off the power of the connected node. Problem diagnosis was

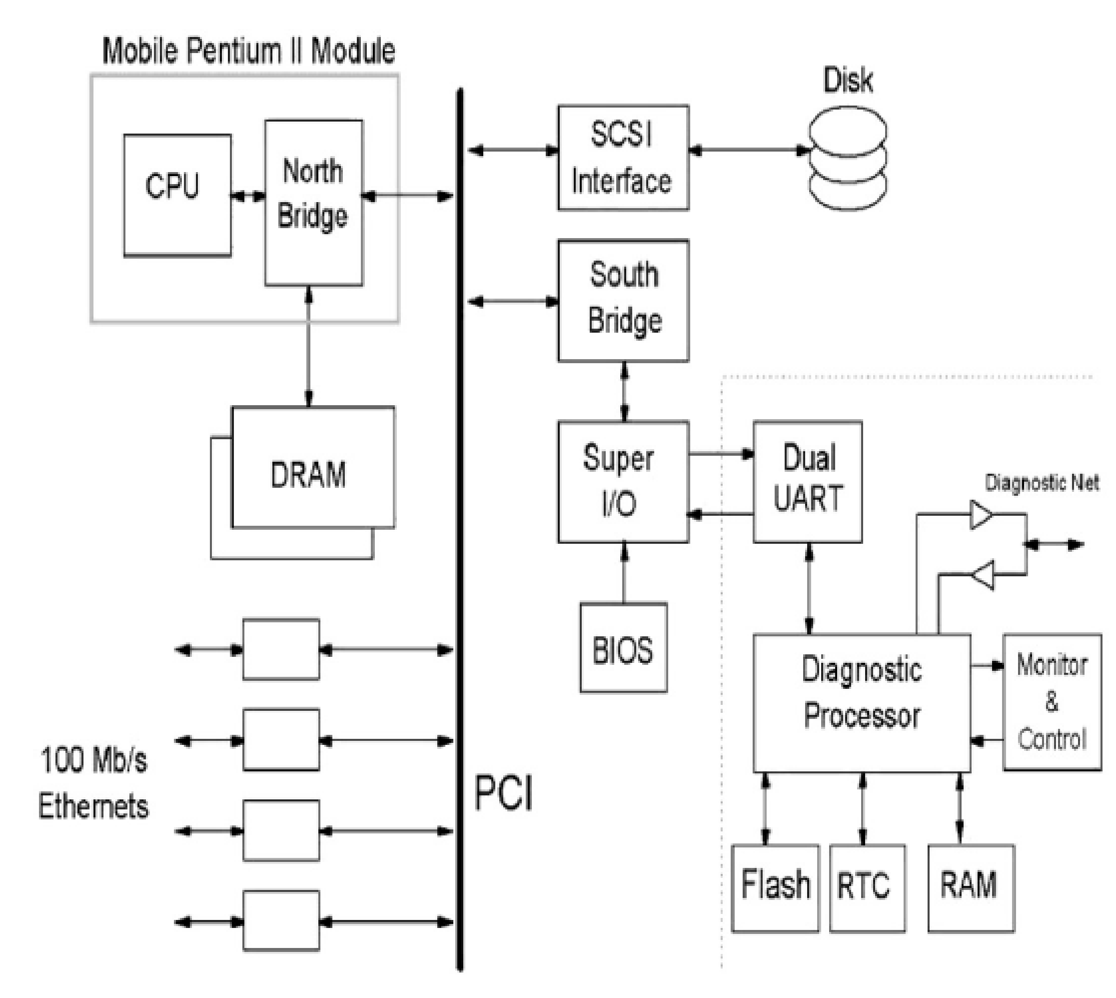


Figure 3: Each brick consisted of four network interface, 266 MHz Pentium II processor, 18 GB SCSI disk, 26 MG RAM and a diagnostic processor. Eight of these bricks were organized as trays, with eight trays forming the cluster.

handled at the hardware level by the diagnostic processor by utilizing data from the temperature and humidity sensors and RAM based loggers for recording system events.

One of the drawbacks faced in the design was the excessive area on the main brick board. The main lessons from this prototype were the possibility of hardware partitioning with standard components. Recently, similar board management controllers (BMC) have been introduced for intelligent Platform Management Interface (IPMI) architecture by Intel[2].

[2] http://www.intel.com/design/servers/ipmi/index.htm

Next section provides a review of several research projects that deal with enabling the autonomic elements to make the self-management decisions (autonomic computing) with a focus on their application towards cluster management.

Autonomic Computing

The vision of *Autonomic Computing* was introduced in 2001 during a presentation made by Paul Horn, a senior vice president of IBM research at that time [39]. Later, Kephart and Chess formalized the notion of autonomic computing in [35]. According to the autonomic computing vision, computing systems should manage themselves based on high-level objectives set by administrators. IBM believes that this will be the only option to handle the complexity and increasing total cost of ownership of large pervasive computing systems, with several interconnected components, in the near future [40].

Autonomic computing is inspired from the idea of self-management in autonomous nervous system of biological systems that governs heart rate and body temperature involuntarily, freeing the brain to make other high-level decisions. Hewlett-Packards's *Adaptive Enterprise Initiative* [41] and Microsoft's *Dynamic Systems* [42] initiative are other industry efforts proposing similar self-managed computing systems vision. Together, these industry initiatives outline the importance of self-management in computing systems.

According to a white paper published by IBM in 2002[43], the road to autonomic computing consists of five distinct levels of system management. Figure 4 concisely presents these levels, which are manual, managed, predictive, adapted and autonomic. They relate to the attainment of an "ideal autonomic state" based on advanced, delivered and planned technologies that would enable the system to have four specific features: Self-configuring, Self-healing, Self-optimizing, and Self-protecting [43, 35, 40, 44]:

Self-configuring: This feature deals with installation, configuration and integration of IT systems. Such systems must possess the capability to dynamically add/remove/configure hardware/software without disrupting existing services.

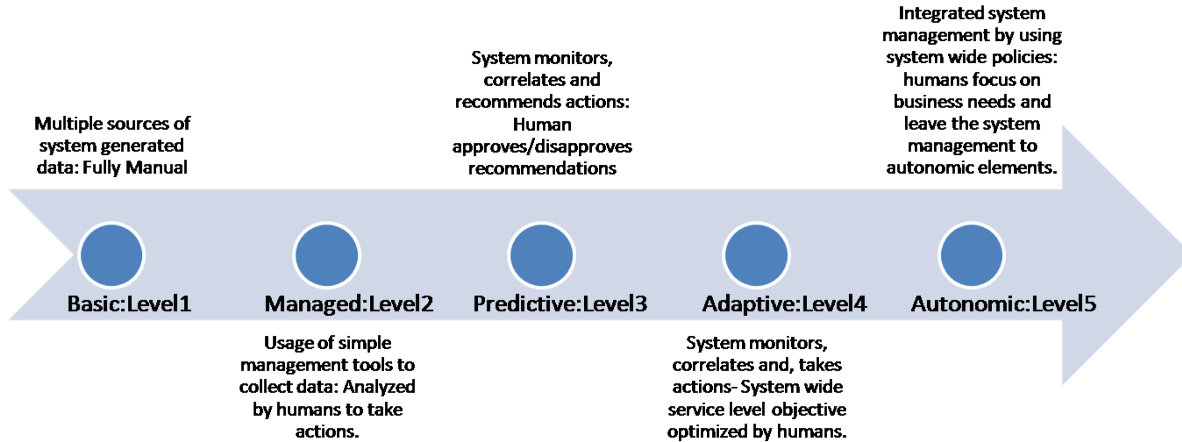


Figure 4: Levels of autonomic computing

Self-healing: This deals with the fault-tolerance and fault-recovery aspect of managing systems. A self-healing system must possess the capability to determine failures and should be able to perform adequate mitigation actions in order to maintain acceptable level of quality of service (QoS), which can be defined as higher -level system specification by the administrators. Self-healing requires the system to reason about the activities that can be performed, how diagnostic information is produced and how new changes can be affected. Specification of possible mitigation actions for faults is required at design time.

Self-optimizing: This feature implies that the systems monitor and tune resources automatically to meet end-user demands.

Self-protecting: Self-protecting systems can anticipate, detect and protect themselves from malware attacks, and unauthorized resource access. They must have the ability to define and manage user access and provide backup and restore procedures.

Since its inception, the self-* list of features has grown substantially [45]. This list now includes:

- Self-governing, Self-adapting, self-diagnosis, self-managing, self-recovery, self-reflecting (Sterritt et. al. [46]).
- Self-planning, self-learning, self-simulation, self-scheduling, self-organized, self-evolution (Tianfeld [45]).
- Goal-driven self-assembly, real-time self-optimizing (Tesauro [47]).

An Architectural Outlook on Autonomic Computing

Kephart et al. outlined the scheme of autonomic computing architecture in [35, 48]. Autonomic Elements (AE) are the basic building block of this architecture. Their job is to provide the capability of following the adaptation cycle as illustrated in Figure 5 to a computation resource. They have defined AE as a component that is responsible for managing its own behavior according to set policies, and for interacting with other AE to provide computational services. Each AE is associated with a managed element - the computation resource being managed. While the description of the architecture in these papers is superfluous and not detailed, their intention is to outline the architectural guidelines for researchers in the area to follow.

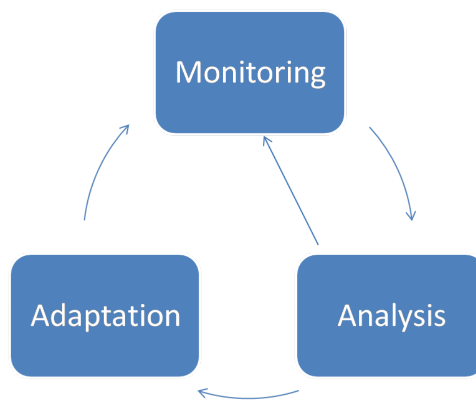


Figure 5: Monitoring and analysis forms an inner loop that is essential for learning. Then the monitoring, analysis and adaptation cycle is the one that provides self-management.

In their approach, every component of an autonomic system is an autonomic element, also referred as software agents in [47, 49]. Each autonomic element possesses the capability to monitor its own managed component using sensors, and then use the historical knowledge to analyze the current and future course of actions, which can involve analysis and planning stages. Finally, it can give commands to the managed component to change its configuration to confirm with system policies.

Autonomic elements draw upon a number of common technologies, including monitoring, modeling, rule execution, optimization, prediction, planning, feedback control, and machine learning. Overall research efforts in autonomic computing can be "loosely classified" into one of the three areas of artificial intelligence, feedback control theory or system and software engineering (including model-based design attempts). By loose classification we imply that there are no strict boundaries and many autonomic computing projects encompass more than one research area. Therefore, any classification attempt of research projects is bound to be imperfect.

Autonomic Computing and Artificial Intelligence

In [50], Roy Sterritt described that there are two ways of approaching autonomic computing. One is the idea of *Making autonomic computing* and the other is *Achieving autonomic computing*. While the former has an implied systems and software engineering viewpoint, achieving autonomic computing has an implied artificial intelligence and adaptive learning viewpoint. Next, we will review some of the concepts from artificial intelligence community such as prediction/optimization, machine learning, and planning that some researchers have used to achieve autonomic computing for systems.

Prediction/Optimization

An important characteristic of an intelligent agent is its ability to learn from previous experience in order to predict future events. Vilalta et al. presented a study of different prediction

Table 3: Selection of prediction algorithms as presented in [1].

	Short-Term	**Long-Term**
Numerical Data	Stationary Models for time series	Trend and seasonal analysis
Boolean Data	Data Mining	Periodicity analysis using failure models

algorithms related to the scenarios of, (1) long-term prediction of performance variables, (2) short-term prediction of abnormal behavior, and (3) short-term prediction of system events in [1].

Table 3 presents their suggestions on the use of prediction algorithms, short-term and long-term, which depends on the kind of data, numerical (measured value) and Boolean type (failures yes/no . While long-term predictions depend on the trend and long-time fluctuations, short-term predictions require that the data obtained is stationary and its statistical properties do not vary over time. In their work, the authors provided several examples of using k-step extrapolation to predict performance parameters, such as response time or disk utilization for long-term capacity planning. Even though this work was theoretical in nature, it provides insight into application of predictive algorithms in computing systems.

In 2003, Russell et al. proposed a framework called Clockwork for predictive autonomic computing [51]. Clockwork uses an autoregressive integrated moving average filter for predicting future loads. It then reassigns the load from the heaviest loaded servers in the descending order of load. The authors of this paper argue that statistical reasoning and prediction is important as it alleviates the pitfall of using short-term forecast employed in feedback control that can lead to instability. They also described a prototype network attached storage system for demonstrating the feasibility of the method in the same paper [51].

Figure 6 describes the setup used for the prediction experiments in [51]. In their setup, the four stores were four computers that served the user requests made by two clients. These requests involved the use of files residing on a shared disk. A router, which was the autonomic element in this setup, was responsible for allocating a user request to a store. The router was equipped with sensors to measure the response time as served by a particular store and the response time

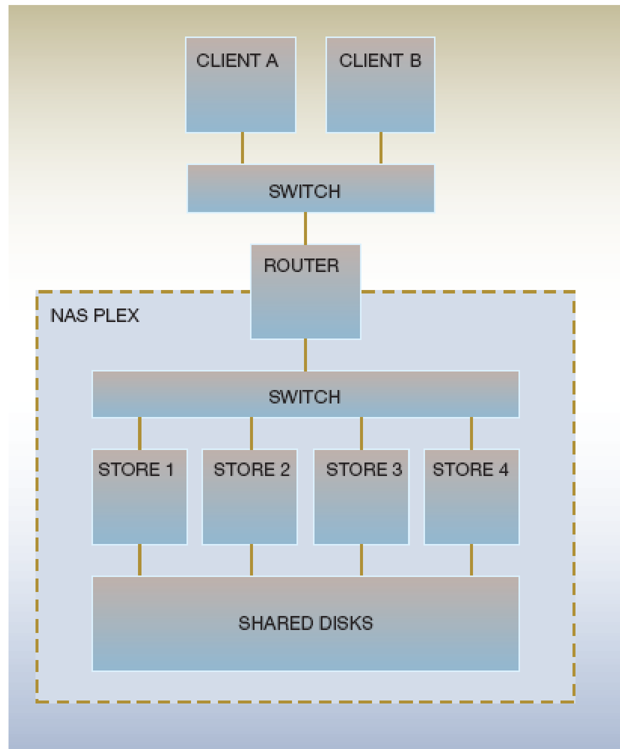


Figure 6: Network attached storage used in Clockwork

per individual file request. Upon receipt of a client request, the router forwarded the request to a store based on a rule setup by the predictive algorithm. Periodically, the statistical data gathered by the router was used by Clockwork to forecast store response time at different loads and track per file access time at that load using an autoregressive integrated moving average filter. The forecast determined the stores that would be overload in the next period. It then iteratively proposed a reassignment of files in the descending order of response time. The iteration stopped as soon as the forecast of average response time across all stores was below a set value. In [52], the authors extended their work to include the facility of changing and selecting a different forecast model online.

This work showed the feasibility of using a predictive technique for load balancing. However, the authors did not discuss the specifics of the forecasting model that they had used. Moreover, they conceded that often the forecast from Clockwork was different from the actual demand, which led to a response time that was over the set goal. They suggested that using a feedback controller to update the forecast model parameter might improve performance.

Use of other probabilistic techniques, such as Bayesian Networks is also central to the research in autonomic computing. A Bayesian network is a probabilistic graph representing events as nodes and the conditional probability of one event occurring after another as edges. A temporal version of Bayesian network for dynamic systems is the Dynamic Bayesian Network. It represents the probability of transition of value of system state variables over discrete time steps. Such networks have been used previously in diagnosis of dynamic systems in [53].

Bayesian networks have also been used in selecting an optimal algorithm for a given computation problem (self-optimization). Haipeng Guo presented his approach towards automatic algorithm selection in [54]. The core of his system was a Bayesian network that modeled the relationship between use of an algorithm and the corresponding value of the performance parameters such as response time. He trained his network by using randomly generated input data based on the extracted real distributions of actual input data. Once the training was performed, the system was used to find the most probably algorithm for using a certain set of inputs and performance requirement. The key issue with this approach is to obtain a random training set that can appropriately represent the real world situation. Bayesian networks have also been used for autonomic, on demand optimization of an IT business infrastructure according to high-level business objectives [55].

Planning

Automated Planning is a branch of artificial intelligence that deals with realization of action sequences, which must be executed to drive an automated agent from its initial state to a final goal state, while following a set of defined constraints. Planning is a wide discipline characterized by the representation of environment, the agent's goal and its model of the world. Planning algorithms are a refinement search over the space of possible situations (situation space) or possible plans (plan space) to find a single or a set of feasible plans[56].

Over the last few years, it has been recognized that despite the obvious connections between planning and search for a suitable path to recovery/optimization in autonomic computing, there are

still issues that must be addressed for planning to be truly practical for autonomic computing [57]. While making a case of planning in field of autonomic computing Srivasatva et al. points that autonomic elements have to conduct planning in open environments, which are not completely understood i.e. planning techniques are required that could handle conditions where complete pre/post conditions of an action are unknown. Moreover, it is necessary to reevaluate the plans in autonomic computing based on the changes monitored in the course of a plan execution periodically. Finally, they suggested that the need to maintain metadata for plans, as it would prove helpful in use of plans that have been successful in the past.

Two planning based autonomic computing systems in existence are CHAMPS [58] (Change Management with Planning and Scheduling) and an extension of ABLE [59]. CHAMPS automates the construction of change management plans for an IT infrastructure. It aims to provide support to administrators by automating the large portions of configuration process. It consists of a task graph builder, a planner and a scheduler. The planner and scheduler generate a change plan that are specific to the domain and satisfies the constraints annotated in the task graph that includes deadlines and maximizes the parallelization in task executions. Once the plan is generated, a deployment system automatically checks out required software packages and installs them onto the appropriate subsystems. The advantage of a domain-dependent planner like CHAMPS for self-configuration is that it can find efficient plans for the specific scenario. However, it will not work for a generic situation.

One implementation of domain-independent planning is the work of Srivastava et al. [60], which is based on an extension of the ABLE toolkit, a toolkit for building autonomic system with multiple autonomic elements [59]. Firstly, they added the option of specifying a plan as a rule by using the planning community's Planning Domain Description Language (PDDL) in ABLE. Then, they added a general planning framework that comprises of a set of common interfaces that agree with PDDL. In the same paper, they presented a usage scenario for planning with ABLE. The objective of the scenario was to modify the configuration of an online bookstore application at runtime to keep the website available. The problem was a two-machine setup with two servlets

that could run on any one of the machines, provided an application server was running on that machine. The applications needed a database and a directory server that could run on any of the two machines. The goal was to keep the system in a running state and move the applications to another machine upon failure so that the servlets continued to run.

They modeled the problem as a planning problem with the initial state of hardware and software configuration and the final goal conditions that corresponded to a running system. They also defined several actions such as start/stop database, install database with their preconditions and post conditions. The actions defined in planner required a corresponding user defined Java function to implement the action. Their planner found the plan that led from initial condition to the goal condition. One quick remark is to mention that the search space of this example was small as there were only 2 machines and there were only 16 possible correct configurations:- 2 machines * 2 servlets * 2 possible location of working for database * 2 possible location of working for directory server.

ABLE only provides use of standard planners in autonomic computing application and assumes that the domain specification and all preconditions and post conditions are known. This is not always the case in scientific computing applications.

Reinforcement Learning

Reinforcement learning is a technique, often used in artificial intelligence where a trial and learn technique coupled with rewards for certain actions, is used, to learn the best policy (here policy stands for a mapping from a state to an action i.e. which possible action to choose when in a particular state) [61]. Tesauro et al. proposed the use of reinforcement learning (RL) for automatically learning management policies [62, 63].

Formally, the basic reinforcement-learning model is comprised of:

1. A set of states S,

2. A set of action A, s.t. $S \times A \rightarrow S$ i.e. an action in a state leads to one of the possible states,

3. A set of scalar reward functions R, s.t. $S \times A \rightarrow R$, i.e. a reward value is associated with taking an action in a state.

Then the problem of RL is to develop a policy $\pi : S \rightarrow A$, which maximizes the sum of rewards $\sum_t \gamma^t R_t$, where t denotes the depth in future, $\gamma \in (0, 1)$ is a discounting factor to ensure that the series converges, R_t denotes the reward obtained at the tth action. The classical RL problem is solved by using the dynamic programming approach also known as the Value-Iteration method [56]. In this method, a value $V(s)$, $s \in S$ associated with each state is kept in a lookup table. Iteratively, this value is updated for all states by sweeping through the state space, updating the value of each state according to equation 3.1, until a sweep through state space is performed, in which there are no changes to state values. In this equation 'a' is the action that leads from state s_t to state s_{t+1} and gets a reward $r(s_t, a)$.

$$\delta V(s_t) = max(r(s_t, a) + \gamma V(s_{t+1})) - V(s_t) \tag{3.1}$$

There are many variation of the basic RL model. A particular variation mentioned by Tesauro is the SARSA (State-Action-Reward-State-Action) algorithm. A problem with the value-iteration method is that it requires the knowledge of the exact future state that will result because of an action. In real-life, systems are non-deterministic and hence only a probabilistic distribution over the possible future states is available. For such systems, SARSA proposes the use of a value function called $Q_\pi(s, a)$, which is maintained for each state and action pair (unlike the value-iteration function that depends only on the state). It estimates the long-range expected value starting in the state 's', taking initial action 'a' and then using policy 'π' to choose subsequent actions [61, 2]. The typical policy is to choose the action in a state with highest $Q - value$.

Tesauro states that RL has advantages in autonomic computing because it does not require an explicit model of the system and it accounts for future consequences of an action by using the discounting factor. However, he cautions that RL suffers from the problem of state explosion, because in simple form RL uses a lookup table to store the $Q - value$ for each state and action

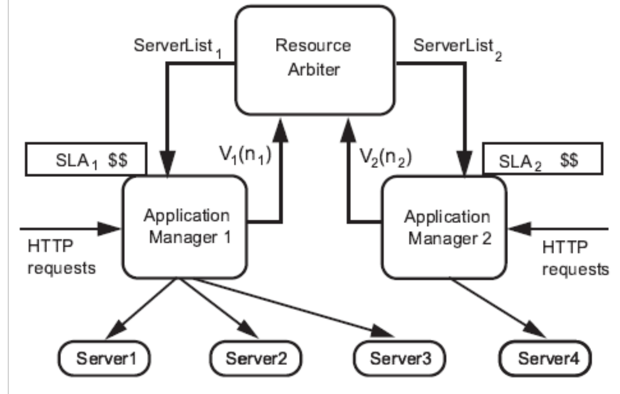


Figure 7: Data center architecture used in [2]

pair. Moreover, the convergence of an online training RL algorithm depends on the arbitrarily chosen initial state, which requires domain knowledge. This was the case in his work presented in [62, 63]. In that, they had to make severe simplifying approximations to reduce the state-space of an online RL algorithm for a trading application. In the concerned example, the action was chosen to be the number of servers online and the state was chosen to be the arrival rate of page requests, averaged over the number of servers. Tesauro suggested that his model was too simple for a real system, however he stated that they were able to successfully train the system by using a greedy exploration rule in which the actions were chosen randomly instead of trying for maximization of the $Q - value$.

In [2], he augmented the method with the use of offline training. He called this method Hybrid-RL. In this method they used a fixed policy to obtain online readings for $Q - value$ functions associated with states and actions. Then they used that data to obtain an offline policy that used the online data as the initial state and tried to maximize the $Q - value$ function. They also changed the lookup table to a nonlinear function approximator to represent the $Q - value$ function. They chose a neural network (multi-layer perceptron) for this approximator.

They tested the hybrid RL approach on a prototype of Data Center shown in figure 7. This figure describes a set of identical server that can be allocated to different application managers. Application managers were responsible for their performance optimization. It communicated with the resource arbiter for resource needs. The optimization goal was given by a local SLA to each

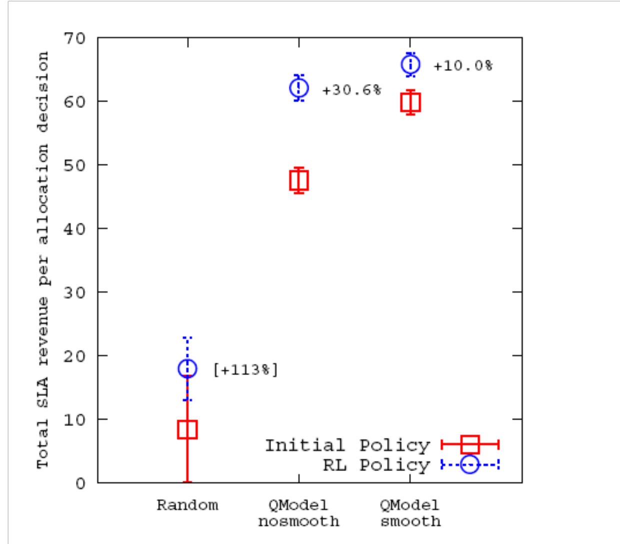


Figure 8: Results of using an offline RL policy compared to the initial online policies.

application manager. The resource arbiter's goal was to allocate servers to maximize the sum of expected business value. These allocation decisions were made every five seconds. The reward function in this case was based on a commonly scaled utility curve reported by each application manager to the resource arbiter, estimating the expected business value, in the current application state, as a function of the number of allocated servers.

For the experiments, they chose three initial polices based on the queuing theory, a random policy, and the third based on a smooth estimate of the response times from each application depending on the number of servers allocated. Figure 8 illustrates that there was gain in all the three initial policies when a new policy was learned using RL offline. The response times are inversely proportional related to the utility curve of the application.

From the above discussion, we can conclude that RL can be used with dynamic systems where the system's actions for each state can be quantized and given an associated value. However, these techniques also require approximation to limit the exploratory state space to ensure a convergent solution can be reached within a bounded time interval. Usually, these techniques will map well to self-optimizing systems that need to change allocated resources to service client requests.

Autonomic Computing and Feedback Control Theory

Road map given by European Network of Excellence ARTIST2 on control of computing systems [64] specifically identifies the need of active research in six areas of computing systems, which can benefit from control theory. These are control of server systems, control of CPU resources, feedback scheduling of control systems, control of communication networks, error control of software systems, and control middleware, which are all essential to the vision of self-management.

Autonomic computing, by definition of self-* properties, provides a blueprint for feedback loops (see figure. 5). This observation suggests that control theory must provide guidance for development of autonomic computing elements. However, in reality this is not straightforward in all autonomic computing problems. The adaptation of the typical feedback control science to software systems has only been formalized in the last decade[65]. Only recently, textbooks such as "Feedback Control of Computing Systems" by Hellerstein et al.[66] that deal specifically with control in computing science have been published.

The foremost problem in applying traditional control theory is the problem of capturing the behavior of a software system using tools like Laplace domain transfer functions (for analog systems)[67] and Z-domain transfer functions (for discrete systems)[68]. In physical world, the laws of nature, to a large degree, lend some pattern to the plant's dynamic behavior, which can be predictably captured by using a mathematical model. However, a computing system (and mainly the software) does not necessarily follow any physical rule on a macroscopic level. Therefore, the modeling formalisms for computing systems have to rely on abstraction. Moreover, the choice of formalism depends on the problem in consideration.

Queuing theory based models are popular for flow control problems in situations such as data centers [69]. For such problems, it is possible to treat the computing system as a black box and concentrate on its interaction with the physical world in terms of rate of incoming requests and outgoing responses using the theoretical foundation of queuing theory. Keshav has described the use of queuing theory for developing a rate allocation server that regulates the flow of TCP/IP

packets in [70]. Such models leads to stochastic control as used in [71] by Li et al. to control short-term rate variations in TCP/IP.

Automata based formalisms are well suited for expressing and analyzing safety properties. This approach is usually followed in tools like SMV [72]. These models are also useful for discrete supervisory controllers [73]. Other approaches such as in Alloy [74] use abstract design specification to check system properties. Abstract specification can also be checked with tools like PVS [75] that utilizes a theorem proving approach to ratify an assertion about the system properties.

Mostly, application of feedback control theory in computing science is related to dealing with issues related to performance and control of computation resources. For these problems, a modeling formalism that allows for expression of timed behaviors is required. However, these models in computing systems, usually require a system-identification approach, in which inputs and outputs are used to formulate a discrete-time difference question [76]. In the next section, we will provide a brief overview about several works in this area.

Controlling Resources

One of the primary uses of control theory in computing systems is in scheduling algorithms. Stankovic et al. pioneered the case for this approach in [77]. The control in feedback scheduling is achieved by using the allocation of CPU resource as a control variable, e.g. a set of tasks with desired resource consumption, such as CPU-time/memory or I/O bandwidth. For example, Stankovic et al. used CPU utilization as a control variable to regulate deadline-miss ratio for a set of soft real-time tasks in [78]. In the same, they also provided general guidelines for designing feedback loop for different quality of service (QoS) parameters.

Steere et al. introduced a new scheduling scheme based on periodicity of tasks in [79]. Their scheme was to allocate each thread a percentage of CPU cycles over a period, and then use a feedback loop to control both proportion and period. Such schemes are typically suitable for soft real-time system because feedback control cannot guarantee absolute conformance with deadlines.

Control based ideas have also been used for server performance control. In [80], Sha et al. presented a queuing model based feedback controller to keep the performance of a network server to a desired level. In the same spirit, the authors of [81] provided various models for a web server and designed feedback controller for QoS adaptation.

Many scheduling algorithms that allow for QoS adaptation have also been developed. Lu and Koutsoukos, described a multi-input multi-output controller for distributed real-time systems in [82]. The goal of their system was to achieve a utilization set point for CPU on each node to guarantee the processing of incoming periodic tasks. They solved the problem as a constraint optimization problem by minimizing the difference between the set points and actual utilization, subject to the constraints: (1) ensure that no processor exceeds its utilization set point, (2) at the same time, avoid underutilization by making each CPU work as close to its set point as possible. They used a Model predictive controller for solving this control optimization problem.

Abdelwahed et al. have described a similar model predictive approach in their generalized online control framework for self-managing systems, presented in [83]. The main point of their approach is that they only considered switched hybrid systems, which only have a finite number of possible inputs. Moreover, the continuous dynamics of such systems can be established as a discrete-time state space equation (usually a difference equation) such as $x((k+1)T) = f(x(kT), u(kT), d(kT))$, where $x(kT)$ is the system state at time step kT, where T is the sampling time and k is the index of the current time step. $u(kT \in U \subset \mathbb{R}^m)$ is the set of input values, while U is a finite subset of $\mathbb{R}^m$, where m is the number of state variables. $d(kT)$ is the set of environmental parameters at that time.

The finite input sets make an exhaustible search over all inputs a possibility. Thus, they are able to employ a limited look-ahead controller, which explores the state space (as a tree) up to a prediction horizon, measured by the depth of the tree. Any path from the initial state ($x(kT)$) (root of the explored tree) to a future state (the leaf of the tree at the set prediction horizon) that, (a) minimizes the cumulative costs associated with the controlled optimization problem, and (b)Satisfies constraints, is chosen as the next input by the controller. This process is repeated at every time sample.

They have extended this approach to a hierarchical controller in [84]. In their papers, they have shown this controller working for power management of a computer processing a time-varying workload comprising of HTTP and e-commerce related requests in [85].

Recently control theoretic techniques have also been applied for controlling the power used by a distributed computing system such as a data center as in [86]. Though their approach used conventional feedback theory, their main contribution was the use of precision power measurement, instead of using average power. They also showed that a P-controller was sufficient to provide the accuracy and stability guarantees required in an actual data center.

A related and interesting area of research is the generalization of the techniques to apply control-theory based methods for QoS adaptation. Generalizing the application of control techniques to computing systems requires generic services. Zhang et al. presented one such middleware service called ControlWare in [87]. This service enables specification of QoS set points, provides a mapping of these set points into appropriate feedback loops, and connects the necessary sensors and actuators [88]. However, their middleware only provides performance control (not reliability) and does not scale well with the increase in number of computer nodes.

Autonomic Computing and Formal Techniques

Formal methods are a collection of notation and techniques for describing and analyzing systems [89]. These methods are formal in the sense that they are based on sound mathematical theories, such as logic, automata or graph theory. Formal specification techniques introduce a precise and unambiguous description of the properties of system. One of the benefits of using a formal specification technique is the ability to perform validation of the design using theorem proving or model checking methods. Moreover, the use of formal technique also lends itself to the possibility of generating policies of guiding the state of the system to a final state by solving the reachability problem to generate counter examples, which can then be used to guide the transition of the real system.

For example, in supervisory control of DES[3], one seeks to restrict the behavior of a plant in such a way that the supervised system (closed loop of the supervisor and plant) meets the required specifications. These might include "safety" specifications (e.g. prohibit the behavior that can lead to a catastrophe) and "liveness" specifications (e.g. guarantee the eventuality of a specified goal). The seminal work on supervisory control in DES was pioneered by Ramadge and Wonham [90, 91]. In their work, depending upon history of executed events, the control is achieved by disabling the events that leads to undesirable behavior.

Several researchers have indicated that one of the problems in self-management of complex systems is to develop an efficient design method [92, 93, 94, 3]. A key aspect to handling complexity is to use models to represent the different components of the system that reflect the various aspects of self-management. The diversity in this research extends from focus on only architectural/structural for configuration/adaption purposes to modeling of behavior of plant and controller.

Self-Management Using Architectural Models

Rohr et al. advocates the use of architectural models for self-management [3]. They suggest the use of a runtime model, which can be generated from a development model, to reflect the system state and provide reconfiguration functionality by using causal graph over various possible states of its architectural entities. At the core of their approach, they use meta-level specs based on UML to specify constraints, monitoring and reconfiguration operations at the time of development (see Figure 9). They suggest the use of OCL as the constraint language. The monitoring model is used to generate sensors to track the value of important parameters. The reconfiguration model is used to determine the changes (adding or removing components) that should be made to the run-time model in case of failures. It also provides for mapping of reconfiguration operations supported by the middleware. The structure of the runtime model is realized by parts of the development model that are instrumented by monitoring, and on parts of development model that are involved

[3] A Discrete event system (DES) is a discrete state, event-driven system, in which, the state evolution depends entirely on occurrence of discrete events over time

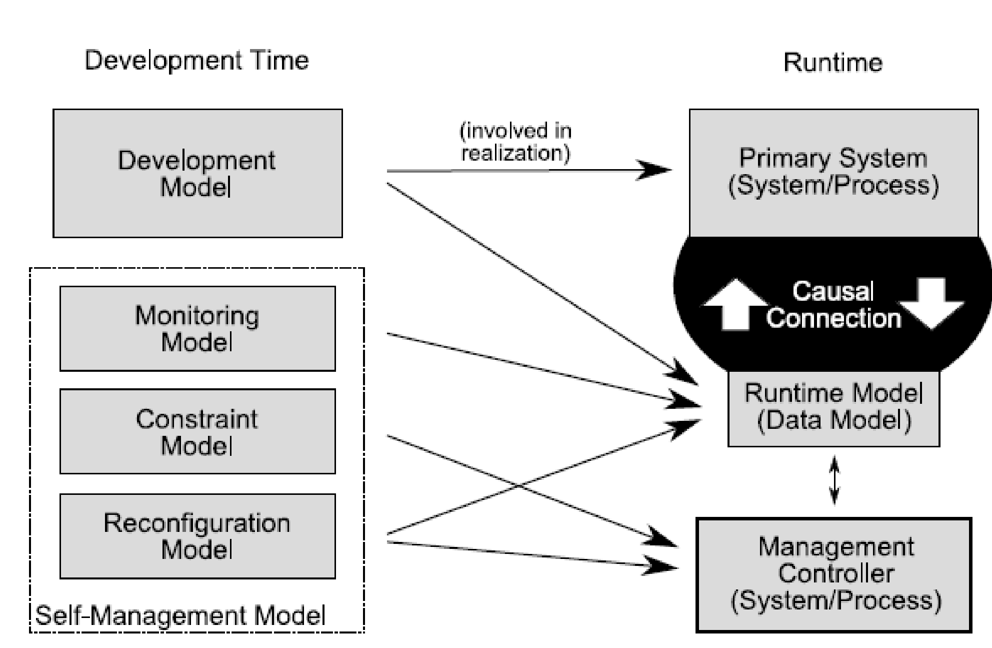


Figure 9: Model-based self management as proposed by Rohr et al. in [3]

in reconfiguration process. Their work, however, leaves several details out about the possible metamodels and the language for specifying constraints. Moreover, they have not provided any concrete example for their approach. A similar but concrete implementation of model-based self-management is found in the work of Garlan et al.

Garlan et.al. [95, 96] and Dashofy et al. [97] have proposed the use of architectural models representing the system as a composition of several components, their interconnection and properties of interest to be used as the formalism upon which system adaptation can be based. Their work follows the theme of Rohr et. al, architectural models are used as runtime models to track system state and take reconfiguration decisions. For reconfiguration, Garlan et al. uses rule based strategies. Their idea is powerful in that it uses models as abstraction that allows for reuse of system design. Primarily, the basis of their adaptation relies upon five points, which are:

1. Ability to model the current architecture of the system using a formal modeling language. This model is represented as an annotated hierarchical graph in a manner similar to the representation scheme followed by a number of architectural description languages. The formal definition of the language (or the metamodel) is called an *architectural style* with

set constraints on the interactions of the components of the system. For example, table 4 provides the definition of a client-server architectural style commonly used by Garlan in his examples. The architectural styles used will be different for different systems.

2. Ability to capture adaptation operators that can be used for creating rules for changes to the model For example, Garlan refers to the use of add and remove component operators in his client server model.

3. Ability to capture constraints so that a model can be judged syntactically correct before and after adaptation. These constraints are also called invariants by Garlan. He used Object Constraint Language [98] for capturing these constraints.

4. Ability to translate the concrete system concepts to the abstract architectural model and vice-versa.

5. Ability to actuate the changes made in the architectural model to the actual system without restarting.

```
Family ClientServerFam = {
        Component Type Clientl = {...};
        Component Type ServerT = {...};
        Component Type ServerGroupT = {...};
        Role Type ClientRoleT = (...};
        Role Type ServerRoleT = {...};
        Connector Type LinkT = {
                invariant slze(select r: role in Self.Roles |
                        declaresType(r, ServerRoleT)) == 1;
                invariant size(select r: role in Self.Roles |
                        declaresType(r, ClientRoleT)) >= 1;
                Role ClientRolel : ClientRoleT;
                Role ServerRole: ServerRoleT;
};
}
```

Table 4: An example Client/Server Architectural Style Definition by David Garlan.

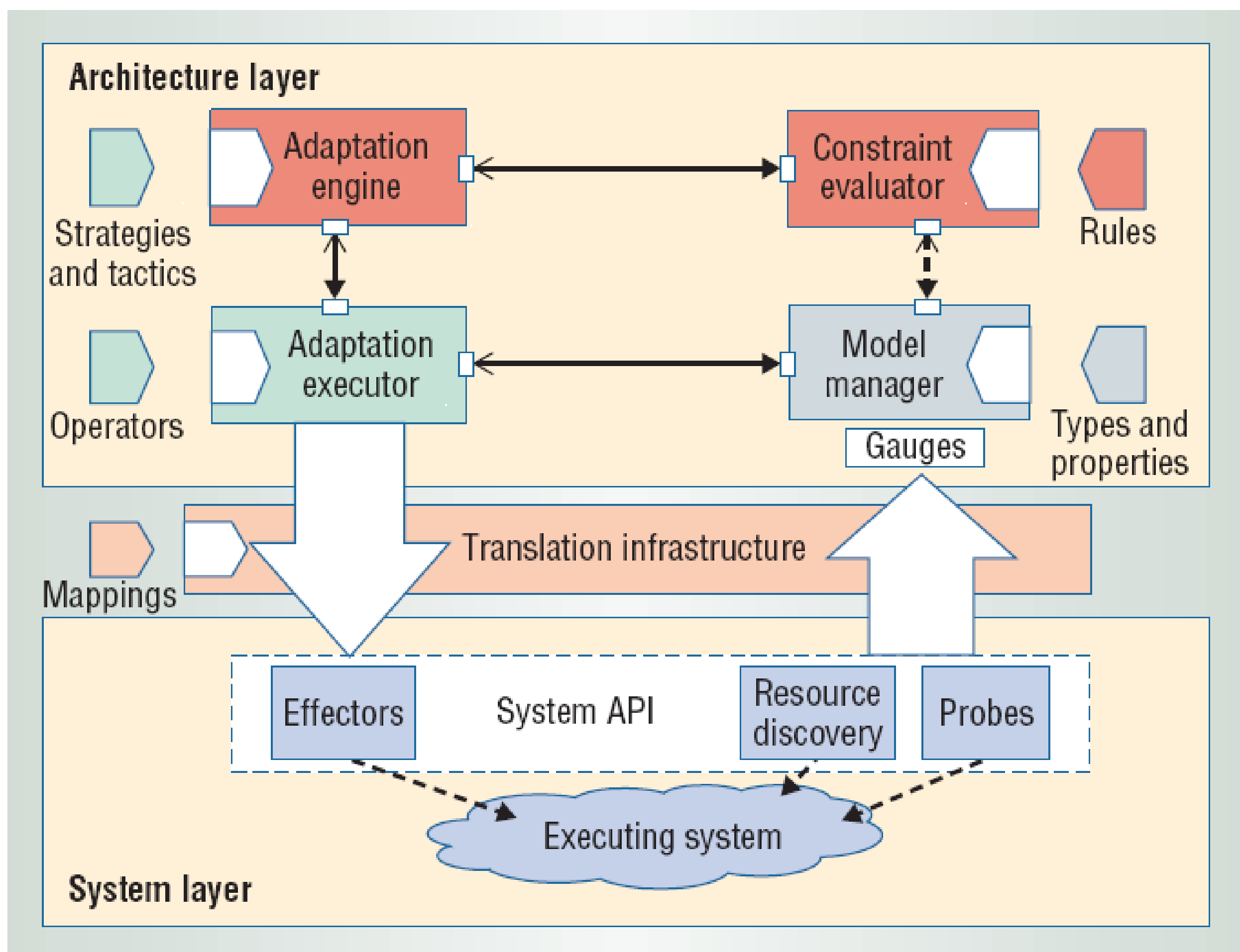


Figure 10: Garlan et al.'s Rainbow Framework [96]. It uses an abstract model of the system layer to monitor, evaluate possible violations (faults) by using a constraint manager, and perform adaptation on an abstract model by using predefined operators. Finally, the adaptations in the model are mapped down to the system layer. Notice that this is a centralized architecture with one central model for the whole system.

Figure 10 illustrates the architecture of the generalized framework called Rainbow described by David Garlan in [96]. Rainbow employs common off-the shelf monitoring sensors to instrument the value of interesting parameters from the executing system. However, sometimes the direct measurements might have to be converted into meaningful observations such as ”system response latency” by using translators called “Gauge”. In the architectural layer, these parameter values are evaluated against several pre-defined constraints. If these constraints are violated then

different rules/strategies can be employed to make changes to the architectural model using various adaptation monitors.

Garlan's approach has several benefits. The primary benefit is the ability to check if an adaptation will lead the system to an inconsistent state. Furthermore, it enables users to create various strategies for adaptation using the basic adaptation operations. A typical strategy looks like an if-then-else clause present in several high-level languages. However, there are several issues:

1. There is no guarantee that the adaptation strategy will only result in one possible change candidate. Garlan does not explain how one can choose between the possible candidate models.

2. The Rainbow architecture proposed by Garlan uses a central controller and a monolithic model for the whole system. This approach when applied to large scale-systems will have scalability issues. The biggest example given by Garlan in his paper consisted of 10 to 12 nodes.

3. There is no mention of the latency caused by converting the information from system layer to the architectural layer and then back to system layer. This latency can increase as the size of the system model increases.

4. There is no characterization of robustness of the changed architectural model to future failures. Some consideration must be given to the future resilience of the change candidate.

5. There is no central repository to maintain a historical database for the faults and affected adaptations. Such a database can prove very useful in building future strategies.

Upon careful observation, familiar concepts can be found between the method of using models as proposed by Rohr et al. and Garlan et al. and the principle of Model-Integrated Computing (MIC) [99]. MIC advocates use of formal models to capture the behavior and structural semantics of a system. A key capability supported by MIC is the definition and implementation of *domain-specific modeling languages* (DSMLs). Crucial to the success of DSMLs is *metamodeling* and

model-translation. A *metamodel* defines the elements of a DSML, which is tailored to a particular domain, is similar to the architectural style used by Garlan.

While some research work have tended to the structural part of the autonomic computing components, other such as Brian Williams et al. and Zhang et al. emphasizes on the need for behavioral modeling of the components.

Zhang et al. described an approach to specify the behavior of adaptable programs in [93]. Their approach is based on separating the adaptation behavior specification and non-adaptive behavior specification in autonomic computing software. They model the source and target models for the program using state charts and then specify an adaptation model, i.e., the model for the adaptation set connecting the source model to the target model. This adaptation model is specified using an extension of the Linear Temporal Logic [100] called Adapt-LTL [101]. In their models, they identify one or many quiescent states in the source model such that if the source system is in one of the quiescent states it will lead to target model upon application of the adaptation operation. This is still a work in progress. Their goal is to be able to verify the target model based on the source model and the adaptation specification. However, they concede that this problem grows exponentially with the size of the source and target model and will be infeasible for specification of adaptation of large systems.

Model-Based Approach to Reactive Fault-Aware Systems

Brian William's research concentrates on model-based autonomy [102, 103]. In his work, he has suggested that failures are a reality of software implementation. They will occur. The emphasis, he believes, should be on developing techniques to enable the software to be able to recognize that it has failed and to recover from the failure [104].

Their technique lies in the use of a Reactive Model-based programming language (RMPL)[105] for specifying correct and faulty behavior of the software components. They also use high-level control programs [106] for guiding the system to the desirable behaviors. Their model based programming language provides executable specification like similar language such as Statecharts

and Esterel. However, there is a difference - their control program interacts in runtime with a plant model and in that sense, it is both control and state aware.

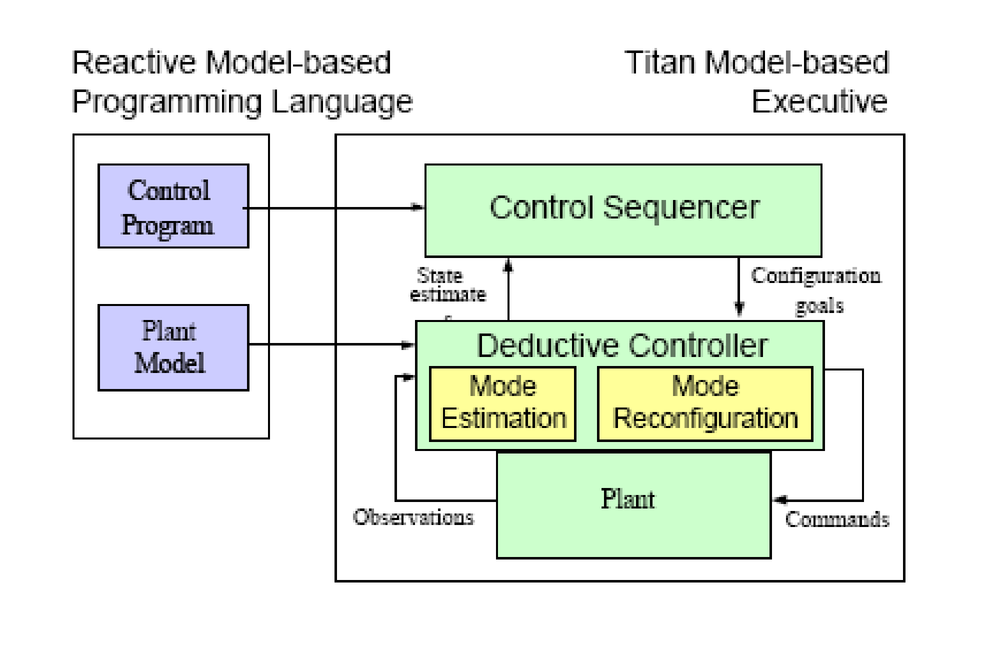


Figure 11: A Model Based Executive

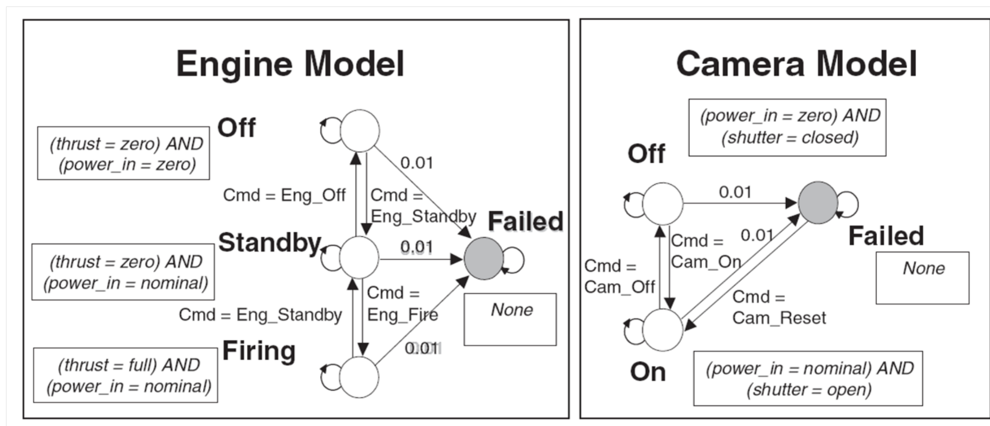


Figure 12: An example state transition model of a plant: a simplified space craft

A model-based program is comprised of two components, control program, which uses standard programming constructs to specify desired system behavior, and a plant model that includes

```
OrbitInsert ():: {
         do
        {
                EngineA = Standby,
                 EngineB = Standby,
                 Camera = Off,
                do
                        when EngineA = Standby AND Camera = Off
                         donext EngineA = Firing
                 watching EngineA = Failed,
                 when EngineA = Failed AND EngineB = Standby
                                                        AND Camera = Off
                        donext EngineB = Firing
                }
        watching EngineA = Firing OR EngineB = Firing
        }
```

Table 5: Control Program written in Reactive Model Based Programming language

plant's nominal behavior and failure modes. Plant model uses a probabilistic modeling formalism that unifies constraints, state charts, Markov processes and provides for concurrency.

Figure 12 describes the model of a spacecraft as a three-component system, two engines and a camera in [106]. An engine can be in one of the three nominal states: off, standby or firing. The behavior within each of these modes is described by a set of constraints on plant variables, namely thrust and power input. Any faulty behavior is captured by the failed state.

Table 5 shows the control specification for inserting this spacecraft into an orbit. The configuration goals specify a final state of the plant, which depends upon the current plant state. The control program generates a control sequence that moves the physical plant to the states specified in the configuration goal as shown in Figure 11.

In his setup, as shown in Figure 11, both the plant model and the control model are executed in a model executive. A model-based executive comprises of a control sequencer and a deductive controller. Control sequencer generates a set of configuration goals based on current plant state estimates. Each configuration goal specifies an abstract state that has to be achieved by the plant. The deductive controller is responsible for observing the plant's state (mode estimation)

and issuing commands to move the plant through a sequence of state that achieve the goal. This approach inherently provides for fault recovery by using the control program to set an appropriate configuration goal that negates the problems caused by the faults.

Both Mode estimation and Mode reconfiguration involves a search over the discrete state space of the plant model to obtain the best solution that satisfies a set of finite constraints, set forth by the configuration goal. Brian Williams advocates the use of Conflict-directed A* technique [103] for performing this search. Conflict directed-A* search reduces the number of states to be explored by pruning all the states that are in conflict with the set of finite domain constraints.

His plant and controller paradigm is critical in functioning of embedded systems. Moreover, the use of formal models for plant allows the use of formal reasoning techniques along with constraints to check if the plant can reach a safe working state and recover from a failure. For example, [107] describes translation from Livingstone [102] (a model executive used to execute plant model and controller program) models in to specification that can be checked by the SMV model checker [100]. He states that the presence of a plant model also tends to validate a sensor reading as probable or completely incorrect. For this, he provides the example of the faulty altimeter reading in the Mars lander project that had a faulty sensor reading, which, he cites registered the height incorrectly and shutdown the engines earlier than it did [105]. He believes that this problem could have been avoided by a deductive controller that would have used the physical model of the plant, the current velocity, the height of entry and figured out that the distance measured by the sensor is incorrect in the sense that the measured distance cannot change from 40 meters to zero in a few seconds. Such reasoning will definitely be useful in autonomic computing systems. However, the problem is to obtain models of plant behaviors that capture all the states realistically and still keep the state space scalable.

Software Engineering Concepts For Autonomic Computing

Systems and software engineering research in autonomic computing attempts to bridge the gap between theoretical developments and real-world deployment of autonomic computing. The

focus in the early years of autonomic computing was on developing implementations, architectures and tools as a proof of concept. Lately, the focus has started to shift to formalize the software engineering concepts for self-management. In [108], Lightstone suggested that systems should be made "just sufficiently" self-managing and should not have any unnecessary complicated function, unless it is absolutely needed. For that, he suggests integrating the autonomic concepts into the design process at the time of requirement itself. There is also a need to develop techniques that will enable augmentation of legacy systems with the autonomic computing elements.

Mary Shaw proposes a practical process control approach for autonomic systems in [109]. She maintains that several dependability models commonly used in autonomic computing science are impractical as they require precise specifications that are hard to obtain. She believes that practical systems should use development models that include the variability and unpredictability of the environment. She further says that the development methods should not pursue absolute correctness (regarding adaption) but should rather focus on the fitness for the intended task, or sufficient correctness. Further, the design should emphasize on adaptations that produce resilience to environment change. She calls this concept as software homeostasis.

Several authors have also considered the application of traditional requirements engineering to the development of autonomic computing systems [110, 111]. In [110], Bustard and Sterritt explain that traditional requirement analysis can be approached from two different dimensions:

1. First dimension: It can either be top down, starting with an environment analysis, or can be bottom up starting with the computing system analysis.

2. Second dimension: It can either start the analysis from the problem statement, focusing on current shortcomings and determining how to address them. Or, it can be goal oriented, developing a final vision for the computing system.

Based on the combination of the two dimensions, there are four possible choices for the starting point of analysis: Top down/goal oriented, bottom up/goal oriented, top down/problem oriented, and bottom up/problem oriented. Bustard and Sterritt believes that autonomic computing systems

require the knowledge of the environment and hence the most suitable approaches are either Top down/goal oriented or top down/problem oriented. They further state the choice between the two suitable techniques will depend on the circumstances. The problem oriented approach is adequate where the desire is to simply address a current situation. The goal oriented approach will be prudent where long-term improvements are cherished and the system is being built from scratch.

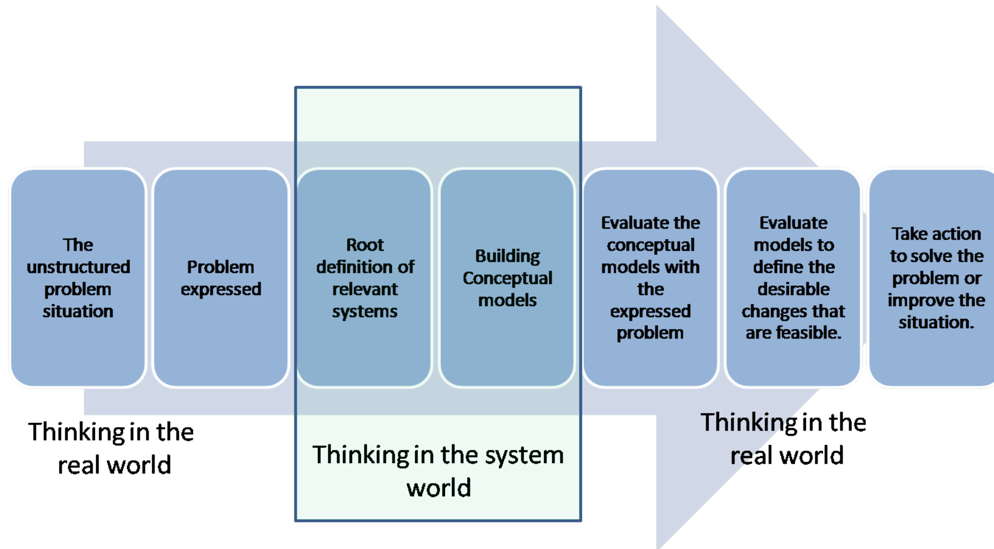


Figure 13: The seven stages of soft systems methodology.

Bustard et al. [110] and Taleb et al. [111] further suggest the use of Checkland's Soft Systems Methodology (SSM) [112] for a top down/goal oriented requirement analysis of autonomic computing systems. SSM was developed in United Kingdom at the University of Lancaster Systems Department and is a general systems improvement technique that identifies beneficial change in the system by using a seven stage process, as illustrated in figure 13.

Engineering for Autonomic Computing Systems

The earlier sections in this chapter focused on the theoretical aspects of various research efforts in autonomic computing. This section reviews the research efforts in autonomic computing that deal with practical aspects of building autonomic systems and do not necessarily fall in any one of

the earlier categories introduced in this chapter. Most of these works can be classified as “enablers”, in the sense that they provide the facilities required to bring autonomicity to legacy applications.

For example, KX (Kinesthetics eXtreme) [113] is a sensor and monitoring middleware for legacy systems. It provides adjustments to the system using actuators. KX is a decentralized collection of middleware components: probe, gauge, controller and effector that communicate using a publish-subscribe event system. A software developer is allowed to write implementations of probe component interface that can monitor a particular application. It also provides for writing controller and effectors for deploying the full system. However, the onus of developing the rules for the system lies on the system administrator.

A problem with a large autonomic computing system is the centralized storage of all the monitoring data, which becomes a bottleneck for querying and data mining procedures. Astrolabe [114] is an infrastructure that tackles this data handling problem by acting as pseudo relational database built using peer to peer protocol running between the applications for which Astrolabe is installed. Astrolabe enables users to create data mining algorithms using SQL and then write corresponding controllers for affecting adaptation. The problem, however, with using astrolabe is the secondary affect on performance of computers, which are part of the autonomic system running Astrolabe.

Chameleon is an adaptive infrastructure, which allows different levels of availability requirements to be simultaneously supported in a networked environment [115]. It is based on reconfigurable processes known as Adaptive Reconfigurable Mobile Objects of Reliability (ARMOR). Each armor process can contain specific user-defined code that can take mitigation actions. Collectively, these processes and services form a reconfigurable fault-tolerant framework. This framework was used in the development of fault-tolerant framework for BTeV trigger subsystem by the Real Time and Embedded System group [116]. The main problem, however, in this framework was a flat hierarchical structure that usually led to overwhelming of the network with communication events. This happened because for every communication each armor object had to broadcast its message to every other armor object.

The Oceano project from IBM[4], facilitates management of computing resources for a "computing utility power plant". This computing utility plant is essentially a virtualized collection of hardware resources. This is still a work in progress. They aim to develop middleware and infrastructure, which provide composition of hosting services, including monitoring of Service Level Agreements, Dynamic Resource Allocation, and High Availability.

Table 6 provides a listing of some other ongoing research projects in autonomic frameworks.

Grid computing

Grid is a large scale distributed computing platform, usually spanning over large geographical areas, enabling the users to solve computation problems at a scale traditionally reserved only for super computers. The idea of the grid was put forward in 1996 by Ian Foster in [123]. It can be visualized as several heterogeneous clusters working together. The key difference between a cluster and a grid is that clusters are usually homogeneous and are owned by a single entity in a small geographical area. However, grid is a collaborative dynamic environment, usually spread across the world. The stark difference is that there is no single owner of the grid and hence a centralized management of all the computing nodes, which are part of the grid, is almost impossible. Therefore, the focus in grid computing is on making applications that are self-aware and manage themselves.

Automate [124, 125] and Autonomia [126] are two similar projects that provides dynamically programmable control and management services to support the development and deployment of smart (intelligent) applications. Since these projects are similar, it is sufficient to review only one of them. Automate [124, 125], is a research project that concerns itself with developing grid applications that are context-aware and can perform self-management tasks. It enables this by incorporating ACCORD, an autonomic component framework, which enables development of grid applications as dynamic composition of components that have been augmented with sensors, actuators, and rules for adaptation. Dynamic application workflows are defined using programming

[4]http://www.research.ibm.com/oceanoproject/index.html

Table 6: A list of Other Autonomic Research Projects

Project Name	Research Institute	Description	Autonomic Issues Addressed
OceanStore [117]	Berkeley	A global persistent data storage. Any computer can join the infrastructure, contributing storage or providing local user access in exchange for economic compensation.	Self-healing,self-optimization, policy based caching
Storage Tank [118]	IBM Research	It is another universally accessible storage management system. It uses virtualization to redirect a user to a particular server.	Supports optimization of resources, self-healing and policy based storage. It also has the provision of checkpoint-based recovery by using logs.
Q-Fabric [119]	Georgia Institute of Technology	It is provides system support for continuous online management of resources and the applications using them. It enables QoS specification to be maintained for individual applications. It uses OS-kernel level components for achieving this control. Q-Fabric focuses mainly on multimedia applications.	Self-organization and self-optimization
Auto Admin [120]	Microsoft Research	Self tuning large databases to reduce total cost of ownership.	self-optimization
Smart DB2 [121]	IBM Research	Reduction of human intervention in the IBM DB2 database.	self-configuration,self-optimization and automatic index generation based on history of queries. It also supports recovery in case of disaster.
Sabio [122]	IBM Research	It Classifies large number of documents and creates taxonomy automatically that is useful for searches	self-organization.

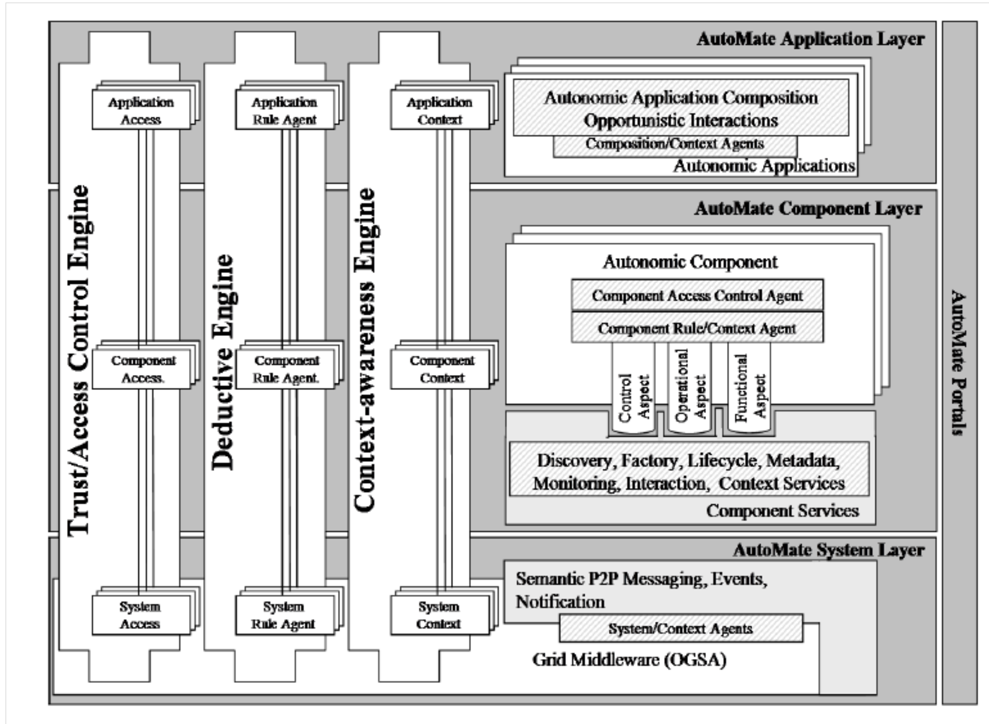


Figure 14: Automate Architecture Diagram.

abstractions built over these components. At runtime, the execution engine selects the most appropriate component based on user/application constraints.

Figure 15, provides an overview of typical Automate component. Each component has a functional, an operational and a control aspect. The functional profile is used at runtime by the execution engine to specify the services provides. The operational aspect abstracts the computational resource requirements and performance (scalability). Finally, the control aspect defines the policies for management, sensor/actuators and the adaptability of the component overall.

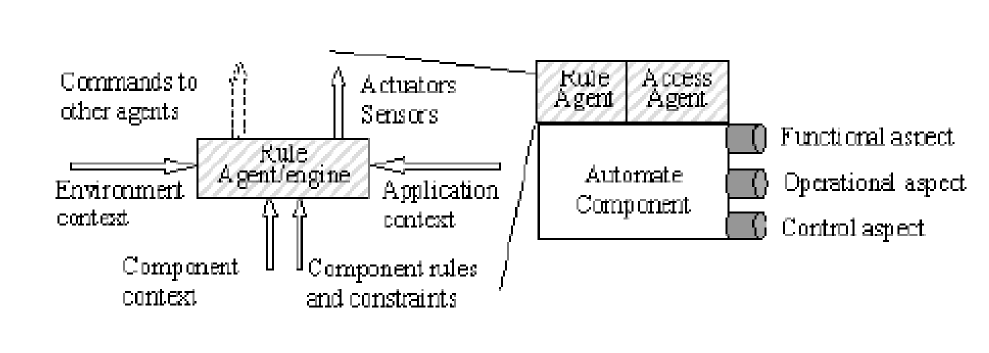


Figure 15: A typical Component in Automate.

A schematic representation of the overall architecture is given in Figure 14. Primarily, it consists of three layers:

- **System Layer:** The system layer is built upon core grid services and provides for peer-peer messaging and notification. They often refer this layer as the Automate middleware layer.
- **Component Layer:** This layer addresses the programming abstractions that are needed to define the components. It also addresses the execution and runtime management of components.
- **Application Layer:** This layer uses the programming abstractions defined in the middleware and component layer to construct applications.

In addition to these layers, Automate also provides for several agents or Automate engines that are decentralized and provides for execution and evaluation of various rules that have been defined for adaptation across the three layers.

Automate, has been shown to be of use in several grid application including oil well localization studies [127] and network security analysis [128]. By pushing the autonomic services transparently to the user (as part of the execution environment), Automate's approach proves beneficial for users who cannot be concerned with developing autonomic applications from scratch. However, Automate is geared towards Grid Services and for applications, which are similar in nature and can be built from a set of reusable components. It will be inherently difficult to adapt its concept of autonomic components to be used in scientific computing where most programs are one-time computation campaigns that have to be written from scratch. Moreover, in scientific computing it is not the application that has to adapt, the execution environment has to adapt to fit the needs of the application.

Application of Autonomic Computing in Clusters

Generalized frameworks are being increasingly used to monitor the health and status of cluster resources. Early tools such as ClusterProbe [129] concentrated on per node monitoring and visualization without considering the health of the cluster as a whole.

Many monitoring tools have been developed at centers for supercomputing at the national laboratories. NetLogger [130] was developed at the Lawrence Berkeley National Laboratory and provides high performance event logging channels for capturing status messages across clusters into a centralized location. The Monitoring and Managing Multiple Clusters (M3C) [131] was developed at Oak Ridge National Laboratory, which provides a web-based GUI administration tool that allows a human administrator to monitor parameters across a federation of clusters on demand. Further developments of cluster management systems include the Java Agents for Monitoring and Management (JAMM) [132] system that was developed at Lawrence Berkeley National Laboratory to facilitate monitoring of more dynamic configurations of cluster environments using a publisher/subscriber methodology. These frameworks were tailored for centralized, human-in-the-loop management of clusters but little investigation was done to provide autonomic monitoring and control of cluster computing environments.

OVIS [133] uses a more complex statistical method to deduce models for a cluster's baseline status and provides a mechanism for automatic detection of early failures based on a node's conformity to those models. However, the computational costs of this approach are intensive, and can have a detrimental effect on the available cluster resources. Other tools such as the RVision [134] monitoring system have investigated the effects of the monitoring framework on the performance of the cluster. Metrics for measuring how the RVision monitoring system interferes with the performance of hosted cluster applications is given by Ferreto in [134]. While the results were promising, no special consideration was given to mechanism for ensuring that the monitors do not interfere with the applications. Moreover, no consideration is given to the necessary failure mitigation portions of a truly autonomic monitoring and control framework.

In recent years, many modern frameworks such as Ganglia [135], Nagios [136] have been undergoing development. These frameworks are well suited toward cluster monitoring and even simple control, and their open source licenses promote their use within the scientific community. While many of these frameworks are easily extended, they provide no mechanisms for bounding either the invocation time or execution time of components within the system beyond mechanisms given by the operating system. Nevertheless, there is a need for modern monitoring frameworks that considers the effect of non-determinism of commercial operating systems and compensates for the same.

Summary and Discussion

In this chapter, we have visited several aspects of research efforts related to autonomic computing. Understandably, autonomic computing, by the virtue of automating the decision procedure, requires learning and depends on historical information. For this purpose, it builds up upon research in prediction algorithms, machine learning and feedback control. There has also been a push for using models as an abstraction for handling the design complexities. Even though the field has seen a lot of progress, there are still key research issues and challenges such as management of system complexity, effective handling of multi objective QoS and efficient power management remain.

There is work to be done to develop abstractions and standard models of different architectures in order to bridge the gap between design of autonomic behaviors and the design of system architecture. This will enable researchers to develop adaptations that are synthesized from the model of system architecture like the work done by Garlan et. al . in Rainbow framework, discussed earlier in section on self-Management using architectural models. Such abstractions will also help in consolidating the analysis methods used today in autonomic computing such as Tesauro (Hybrid RL), Abdelwahed et al. (feedback control).

Another challenge in Autonomic Computing is to develop the infrastructure or the middleware i.e. support core autonomic services as part of the infrastructure to realize autonomic behaviors in

a scalable manner, in spite of the application dynamism. Automate [137] and ABLE [59]) have taken a step in the right direction. Nevertheless, there are still issues such as latency and scalability that have to be addressed.

Finally, we still require research in self-aware applications. One possibility is to investigate linking the design models into applications and then use these design models for generating adaptations.

CHAPTER IV

TOWARDS A VERIFIABLE REAL-TIME, AUTONOMIC, FAULT MITIGATION FRAMEWORK FOR LARGE SCALE REAL-TIME SYSTEMS

Overview

This paper was published in the journal Innovations in System and Software Engineering [11]. It details the fault-mitigation aspect of Scientific Computing Autonomic Reliability Framework. Arguably, one method of fault mitigation is to let humans in the loop to make decisions when failures occur. Alternatively, a component of the computer-based system can itself take fault mitigation decisions. The former technique is useful in systems which do not necessitate quick reactions. However, in critical systems where fault mitigations require time-bounded responses, the latency in decision due to human involvement in the control loop is unacceptable.

In this paper, a real-time reflex and healing framework used for providing fault mitigation capabilities for large-scale real-time systems is described. This approach was inspired by biological systems that employ a reflex and healing strategy to mitigate local faults quickly and allow a hierarchy of "managers" to be responsible for healing of widespread faults. To mimic this organization, this framework uses a hierarchical setup of reflex engines that implements reflexes when certain event like "Heartbeat miss" for a computing node is detected. One of the key challenges with implementing mitigation policies is to guarantee that do not contradict each other. Moreover, if there is a real-time aspect to the quality of service, mitigation policies must either guarantee a bounded time resolution or send a failure signal to the supervising manager.

This paper also provides formal semantics for the components of this framework. This semantics enable formal verification of specified failure-management logic. Finally, a case study is presented that describes a scenario in which an existing system [138] is augmented with user-defined behaviors for recovering from temporal faults due to data stream corruption. These behaviors are analyzed using UPPAAL [139], a tool for verifying behavioral models expressed as timed automata.

Abstract

Designing autonomic fault responses is difficult, particularly in large-scale systems, as there is no single 'perfect' fault mitigation response to a given failure. The design of appropriate mitigation actions depend upon the goals and state of the application and environment. Strict time deadlines in real-time systems further exacerbate this problem. Any autonomic behavior in such systems must not only be functionally correct but should also conform to properties of liveness, safety and bounded time responsiveness. This paper details a real-time fault-tolerant framework, which uses a reflex and healing architecture to provide fault mitigation capabilities for large-scale real-time systems. At the heart of this architecture is a real-time reflex engine, which has a state-based failure management logic that can respond to both event- and time-based triggers. We also present a semantic domain for verifying properties of systems, which use this framework of real-time reflex engines. Lastly, a case study, which examines the details of such an approach, is presented.

Introduction and Problem Motivation

The increased influence of smaller, more powerful and more ubiquitous computers in human lives demand a higher degree of reliability from these systems. They are expected to consistently produce correct outputs with very small tolerance for errors. However, as computing technology grows more prominent, the complexity of system design is on the rise. In order to increase reuse and reduce cost, designers are leveraging composition of smaller systems to construct a larger system, which not only increases complexity but also stresses the design process, which makes it difficult to achieve high reliability. Therefore, it is imperative to accept that failures can and will occur, even in meticulously designed systems, and take steps to study the effect of these failures and design proper measures to counter those failures.

Detection and mitigation of faults in large-scale systems with hundreds of components is a crucial and challenging task. Even though the correct operation depends on individual components as well as their interactions, it is necessary that the systems do not suffer a catastrophe every time a subcomponent misbehaves. Most practical systems have to employ some kind of fault detection,

the simplest of which can be a threshold sensor and alarm for every critical aspect of the system. In past decades, the problem of fault diagnosis has received considerable attention in literature and a number of schemes based on fault trees [140], quantitative analytical model-based methods [141, 142], expert systems [143, 142], and model-based reasoning systems [144, 53] have been proposed for both continuous systems [145], as well as discrete event systems [146, 147, 148, 149].

While fault diagnosis is certainly a key aspect of the problem, fault mitigation is another crucial problem that deserves added attention. Arguably, one method of fault mitigation is to let humans in the loop to take decision. Alternatively, a component of the computer-based system can itself take fault mitigation decisions. The former technique is useful in systems, which do not necessitate quick reactions. However, in most critical systems, where fault mitigation tends to require time-bounded responses, the latency in decision due to human involvement in the control loop is unacceptable. For example, any human mitigation decision in interplanetary space exploration systems has to travel the vast distances of space between the earth-based station and the spacecraft, which may lead to a situation where it is impossible to correct for a fault before it becomes threatening to the system [150]. In such and similar critical systems, the only option is to let the computer-based system work autonomously and take the fault mitigation decisions itself.

This drives the need for autonomic computing to address the needs of system reliability. Such self-managing systems should be also self-aware, self-configuring, self-healing, and self-protecting [151, 152]. A motivation for this paradigm comes from biological systems which employ a reflex and healing strategy to mitigate local faults quickly yet still allow a hierarchy of decentralized "managers" to be responsible for healing of widespread faults.

Our previous work in large scale autonomic computing yielded a *Reflex and Healing (RH)* framework presented in [153, 12, 154], which employs a hierarchical network of decentralized fault management entities, called reflex engines. These reflex engines are instantiated as state machines, which change state and initiate reflexive mitigation actions upon occurrence of certain fault events. Other researchers have also proposed to equip systems with decentralized self-management

algorithms [155], and an alternative technique has also been developed that consists of a hierarchy of managing entities placed in the system with control decisions being made based on partial knowledge of the system [95].

In order to apply the aforementioned techniques to systems, which are real-time in nature, additional considerations are required. The addition of time deadlines exacerbate the problems associated with self-management. Under such conditions, one has to not only be concerned about the correctness of a given self-management task; one must also ensure that a task finishes before certain deadlines expire [156]. Fault mitigation tasks, which lead to missed deadlines, are, therefore, considered faulty in hard real-time systems.

Even in systems that are not real-time (and hence do not have specific task deadlines), conditions might require that a fault mitigation is completed within a certain boundary from the time of the fault occurrence. Such requirements of time-bounded responses are true for almost all systems that require a fault-tolerant framework.

In such a context, any autonomic computing technique used must respect time constraints. Moreover, in order to ensure correct operation, the autonomic system must conform to the following properties:

1. *Liveness:* If the system was deadlock free, the addition of autonomic behaviors shall not lead to a deadlock in the system.

2. *Safety:* The system shall meet all of its deadlines and never operate in unsafe modes.

3. *Bounded Time Responsiveness:* In the case of faults, the designed fault mitigation action shall be executed within a specified time bound.

It is necessary to check that the system and its corresponding fault tolerance framework conform to the properties mentioned above. However, an initial investigation revealed that the previous RH framework was unsuitable for verifying these properties. This is because the framework lacks the formalism required for modeling the passage of time. In addition, the previous RH framework also lacks formalism that can allow arbitration between several eligible fault mitigation strategies.

In this paper, we extend the results presented in [13] and detail work toward a verifiable fault tolerant framework for real-time systems. This framework uses a reflex and healing architecture in order to provide fault mitigation capabilities. At the heart of this architecture is a new real-time reflex engine, which implements a state-based failure management logic that can respond to both event- and time-based triggers. Furthermore, this work elaborates our existing model with concepts of schedulers that can arbitrate between several eligible mitigation strategies, which can be shown to respond to failure events within a given time limit. With the additional help of a case study, we show how one can now use the semantics of timed automata [157, 158] to analyze the real-time properties of the framework using UPPAAL [159], a model checking tool for networks of timed automatons.

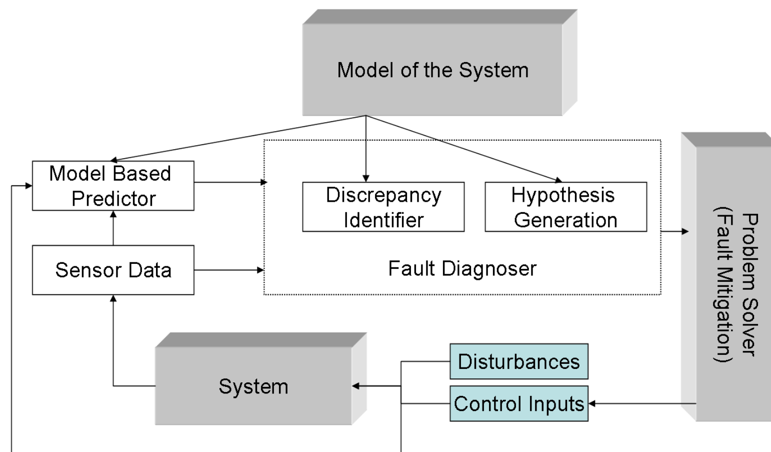


Figure 16: Usual approach to fault diagnosis and mitigation.

Fault Diagnosis and Mitigation

Figure 16 depicts a general approach of model-based fault diagnosis and mitigation. The system model is an abstraction of the plant and is used to estimate the current system state and output using prediction. If the predicted data and measured sensor data differ, the discrepancies are identified to form a symptom set that can be used to generate hypotheses about the faults. The problem

solver uses these hypotheses to generate fault mitigation actions, which provide correctional input to the plant.

As we look more in detail at the problem of fault diagnosis and mitigation, a number of questions arise. What type of model should we use for the system? What are the mechanics associated with the communication between various components? What is the architecture of this problem solver? Answers to these questions are open ended, as there is no "one fits all" solution.

Large Scale Real Time Systems

The real-time embedded systems group (RTES) [138] is a research collaboration of computer scientists, electrical engineers and experimental physicists whose aim is to provide new research in the areas of large-scale, fault-tolerant, high-performance embedded systems. Research efforts of the RTES group are focused on finding new tools and techniques which strengthen the design and development of large scale computation systems used for online phenomena capture, signal processing and data acquisition for high energy physics experiments.

High energy physics (HEP) experiment setups such as the Compact Muon Solenoid (CMS) [160] at the Large Hadron Collider (LHC) at CERN are multi-year, multi-million dollar projects. The estimated particle interactions in these experiments have a periodicity as high as 25 ns, which results in aggregate data rates of several terabytes/second with a petabyte/year in permanent storage needs [160, 161]. It is estimated that an overwhelmingly large percentage of the observed particle interactions will not lead to new science in the areas of high energy physics. Given the size and rate of the data generated during an experiment run, it is infeasible to record complete particle interaction data for a later examination. Therefore, massive real-time embedded systems must be used to execute data-reductive algorithms called "filters", which perform the necessary examination of the collected data to isolate and retain only the observations of interesting phenomena.

Data acquisition and processing systems of high energy physics (HEP) experiments require high throughput and low latency. These systems are typically composed of thousands of data

processing nodes, which incorporate various computational resources such as FPGAs, DSPs, commodity PCs, high-bandwidth fiber and copper interconnects. Given the long durations of typical experiments, failures of these systems are expected to occur.

The high costs involved and the number of computing nodes required makes it infeasible to use traditional redundancy as the primary fault tolerance approach. Furthermore, it is required that the implemented fault tolerance approach should be able to localize and isolate faults and yet maintain some form of acceptable system operation. The system must also possess capabilities to run online recovery procedures, compensate by changing system parameters such as thresholds, and prune out bad computing nodes with minimum impact on system performance. It should be noted that these problems are not unique to the field of HEP but are also present in other fields such as space systems (see [150]).

Like any model-based fault diagnosis and mitigation approach, the first step is to choose a suitable model. The choice of model has profound effect on simplicity of implementation as well as guarantees that can be made about the properties, which the system is required to satisfy.

Abstracting Real-Time Systems as Discrete Event Systems

The act of *modeling* has been described in [162] as, "representing a system formally in order to describe and analyze the working of some relevant portions of the concerned system". A widely used abstraction that is valid for most embedded systems is a discrete event system (DES) model.

A DES is a discrete state, event-driven system in which the state evolution depends entirely on occurrence of discrete events over time [163, 91]. The event driven property implies that the state of the system changes at discrete points in time which correspond to *instantaneous* occurrences of events.

In a DES framework, a description of an application's (plant) behavior is given in the form of a finite automaton. Any behavior of the plant can be explained as an execution of this automaton (i.e., a sequence of events). These events can be either *observable* or *unobservable*. The objective is to find a diagnoser that can detect the occurrence of a fault event within a limited number of steps

of its occurrence using the observable events. Note that our objective is not to solve this diagnosis problem, rather we are concerned with the mitigation of faults. For the diagnosis problem, readers can refer to [147].

Based on this discrete event model, we presented a hierarchical reflex and healing (RH) framework in our earlier papers [153, 12, 154] for providing fault mitigation capability in large-scale systems. In large-scale reflex and healing systems, software applications use an event based communication scheme. Each application generates a regular heartbeat, which informs an observer that the application is alive. Moreover, each application has a self-monitoring process and generates failure events in case of any problem. Given the failure events, the task of the framework is to contain faults, correlate diagnosis from different regions, and decide on an appropriate mitigation strategy.

The Reflex and Healing Architecture Using a Discrete Event Model

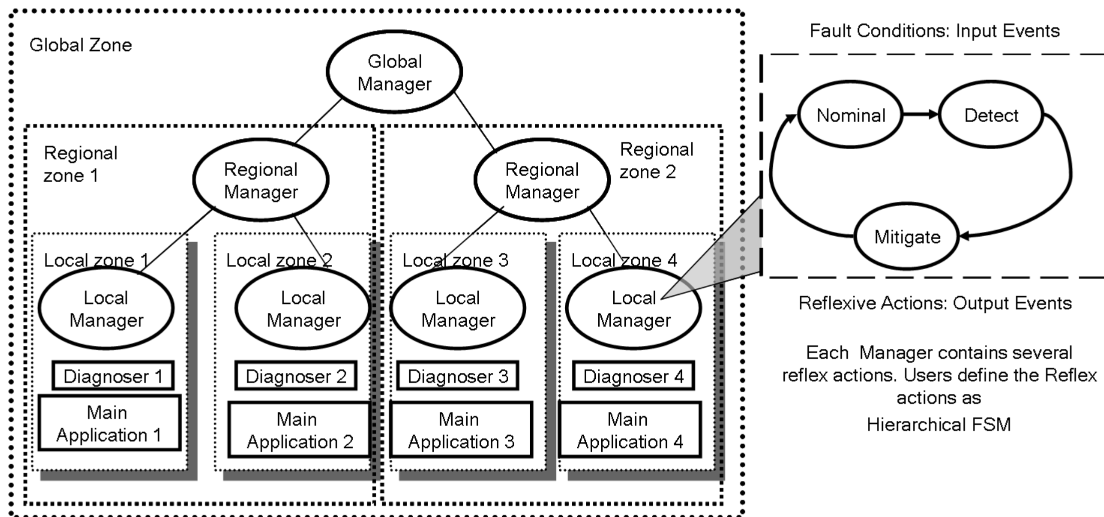


Figure 17: Hierarchical fault management scheme in reflex and healing architecture. The reflex actions are user defined FSM. Communication between managers themselves and to/from the plant is via predefined events.

As shown in Figure 17, the RH framework employs a hierarchical network of fault management entities, called reflex engines, whose sole purpose is to implement fast reflexive fault-mitigating actions.

Reflex engines are customized software processes that perform monitoring and execute fault mitigation actions. It is possible to configure a reflex engine using an external source by programming their state machine. As seen in Figure 17, a reflex engine consumes input event (e.g., fault events) and produces output events (e.g., mitigation actions). Upon occurrence of a certain class of fault events, a reflex engine, which is specifically listening for that event, performs fault mitigation by changing its state and executing a set of actions upon the transition.

In order to manage scalability, all reflex engines are divided into logical hierarchical levels called zones. Each of these zones contains a manager whose area of observation, is limited to that particular zone. There are three main categories of managers, which are:

1. *Local Manager:* These managers are nearest to the main applications. They are usually responsible for managing one user process, as shown in Figure 17.

2. *Regional Manager:* These managers supervise and communicate with a number of subordinate managers which are in their area of observation. A regional manager has a wider area of observation and can correlate diagnosis to ascertain if a problem is common to a number of user applications and take coordinated mitigation action. Note that there can be a number of regional levels.

3. *Global Manager:* This manager lies at the top of hierarchy. It coordinates regions and performs any required optimization of the system. This process is usually referred to as *healing*. For more detail on the complicated process of healing readers can refer to [164, 165].

These reflex engines operate concurrently with user applications. The state machine based failure management logic in each reflex engine responds to faults as they are observed. To minimize fault propagation, intra-level communication between reflex engines is forbidden.

There are definite advantages in following a rigid hierarchical structure with strict communication protocols in place. First, fault reaction time improves because the mitigation decisions are made closer to the fault source. Secondly, scalability improves because new fault managers can be added easily to a zone without disrupting other zones. Lastly, chances for the propagation of faults from one user application to another application are greatly reduced.

The development of the framework to date lacks the details necessary to make real-time guarantees. This is because the discrete event-based model used does not capture time. Hence, one cannot analyze any real-time property such as bounded time response. Moreover, this model does not specify how the operating system under the reflex engine affects its execution sequence. In addition, the state machines can be non-deterministic due to the runtime and the state machine definition. Specifically, if a reflex engine has more than one fault strategy enabled, which one is executed? For these scenarios, we need to incorporate the notion of a scheduler, which is necessary when a reflex engine has to choose between various triggered events in order to process them in a consistent fashion.

First, we need to extend the model for real-time systems to capture notion of time. Then we need to augment the existing RH framework with components that can describe the execution behavior of a reflex engine in detail. Finally, we need to provide exact semantics to this new model using a real-time model of computation, which allows for formal analysis of time based properties outlined.

Abstracting Real-Time Systems As Timed Automatons

A fault-tolerant framework that is capable of providing time guarantees and detecting time anomalies requires a model, which can capture the notion of continuous time. Classically, the timed automaton (TA) model [157, 158] has been used for abstracting time-based behaviors of systems which are influenced by time. This approach has been used to solve scheduling problems [166, 167] by modeling real-time tasks and scheduling algorithms as variants of timed automatons and performing reachability analysis on the equivalent region graph automaton [157].

A timed automaton consists of a finite set of states called *locations* and a finite set of real-valued clocks. It is assumed that time passes at a uniform rate for each clock in the automaton. 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. These transitions are assumed instantaneous. At any time, the value of each clock is equal to the time passed since the last reset of that clock. In order to make the timed automaton urgent, locations are also associated with clock constraints called *invariants*, which must be satisfied for a timed automaton to remain inside a location. If there is no enabled transition out of a location whose invariant has been violated, the timed automaton is said to be *blocked*. Formally, a timed automaton can be defined as follows:

Definition 1 (Timed Automaton) *A timed automaton is a 6-tuple TA=*$< \Sigma, S, S_0, X, I, T >$ *such that*

- Σ *is a finite set of alphabets, which the TA can accept.*
- S *is a finite set of locations.*
- $S_0 \subseteq S$ *is a set of initial locations.*
- X *is a finite set of clocks.*
- $I : S \rightarrow \mathscr{C}(X)$ *is a mapping called location invariant.* $\mathscr{C}(X)$ *is the set of clock constraints over* X *defined in BNF grammar by* $\alpha ::= x \prec c | \neg\alpha | \alpha \wedge \alpha$*, where* $x \in X$ *is a clock,* $\alpha \in \mathscr{C}(X)$*,* $\prec \in \{<, \leq\}$*, and* c *is a rational number.*
- $T \subseteq S \times \Sigma \times \mathscr{C}(X) \times 2^X \times S$ *is a set of transitions. The 5-tuple* $< s, \sigma, \psi, \lambda, s' >$ *corresponds to a transition from location* s *to* s' *via an alphabet* σ*, a clock constraint* ψ *specifies when the transition will be enabled and* $\lambda \subseteq X$ *is the set of clocks whose value will be reset to* 0*.*

It is customary to draw a timed automaton model as a directed graph with nodes, drawn as circles or ellipses, which represent locations and edges, which represent transitions. Initial locations are marked using concentric ellipses or circles. Figure 18 shows a timed automaton model of a

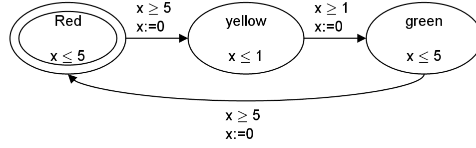


Figure 18: A timed automaton model of a behavior of traffic light.

traffic light. This automaton has three locations and one clock variable. The model periodically circulates between red, yellow, and green location at a time gap of 5, 1, and 5 units, respectively.

The state of a timed automaton is a pair of the current evaluation of clock variables and the current location, $Q = (s, v)$, where $s \in S$ and $v : X \rightarrow \mathbb{R}^+$ is the clock value map, assigning each clock a positive real value. At any time, all clocks increase with a uniform unit rate, which, along with events, enable transitions from one state of the timed automaton to another. Since there are an infinite number of possible clock evaluations, the state space of a timed automaton is infinite. The transition graph over this state space, $A =< \Sigma, Q, Q_0, R >$, is used to describe the semantics associated with a timed automaton model. The initial state of A, Q_0 is given by $\{(q, v) | q \in S_0 \wedge \forall x \in X(v(x) = 0)\}$.

The transition relation R is composed of two types of transitions: delay transitions caused by the passage of time, and action transitions, which lead to a change in location of a timed automaton. Before proceeding further with transitions, it is necessary to first define some notation.

Let us define $v + d$ to be a clock assignment map, which increases the value of each clock $x \in X$ to $v(x) + d$. For $\lambda \subseteq X$ we introduce $v[\lambda := 0]$ to be the clock assignment that maps each clock $y \in \lambda$ to 0, but keeps the value of all clocks $x \in X - \lambda$ same. Using these notations, we can define delay and action transitions as follows:

- *Delay Transitions* refer to passage of time while staying in the same location. They are written as $(s, v) \xrightarrow{d} (s, v + d)$. The necessary condition is $v \in I(s)$ and $v + d \in I(s)$

- *Action Transitions* refer to occurrences of a transition from the set T. Therefore for any transition $< s, \sigma, \psi, \lambda, s' >$, we can write $(s, v) \xrightarrow{\sigma} (s', v[\lambda := 0])$, given that $v[\lambda := 0] \in I(s')$ and $v \in \psi$.

Usually, a system is composed of several sub-systems, each of which can be modeled as a timed automaton. Therefore, for modeling of the complete system, we will have to consider the parallel composition of a network of timed automatons [159, 100, 168].

A network of timed automatons is a parallel composition of several timed automatons [100]. Each timed automaton can synchronize with any other timed automaton by using input events and output actions. For this purpose, we assume the alphabet set Σ to consist of symbols for input events denoted by $\sigma?$ and output actions $\sigma!$ and internal events τ.

Apart from the synchronizing events, tools like UPPAAL [159] and KRONOS [168] have introduced concepts of integer variables and arrays of integers, which can be shared between different timed automatons. Moreover, UPPAAL has introduced some additional notions in order to make modeling as timed automaton simpler [139]. Some of the notable additions that we will later use are:

- *Committed Locations:* The semantics of a committed location is that if any timed automaton of the network is in a committed location it is assumed that time cannot pass. In such a case, the only possible transition for the whole network is the one that goes out of the committed locations.

- *Urgent Channels:* A usual synchronized action transition can be ignored if any other transition is possible. However, if the synchronized event is urgent, then the transition becomes compulsory.

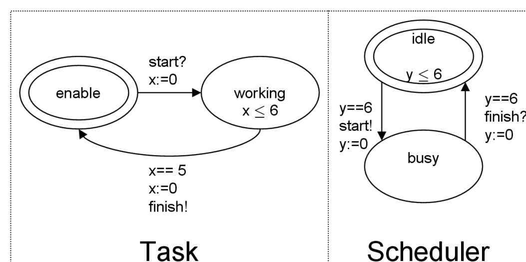


Figure 19: A network of two timed automatons.

The semantics of network-timed automatons are also given in terms of transition graphs. A state of a network is defined as a pair $(\vec{s}, v)$, where $\vec{s}$ denotes a vector of all the current locations of the network, one for each timed automaton, and v is the clock assignment map for all the clocks of the network. The rules for delay transitions are the same as those for a single timed automaton. However, the action transitions are composed of internal actions and external actions.

An internal action transition is a transition, which happens in any one timed automaton of the network, independent of other timed automatons. On the other hand, an external action transition is a synchronous transition between two timed automatons. For such a transition, one timed automaton produces an output event on its transition leading to a change in its location (denoted as $a!$), while the other timed automaton consumes that event (denoted as $a?$) and takes the transitions leading to a change in its location. An external action transition cannot happen if any of the timed automatons cannot synchronize. Figure 19 shows model of a scheduler executing in parallel to a task. The scheduler enables the task 6 time units after it enters the idle location. The two automatons synchronize using start and finish event/action pair.

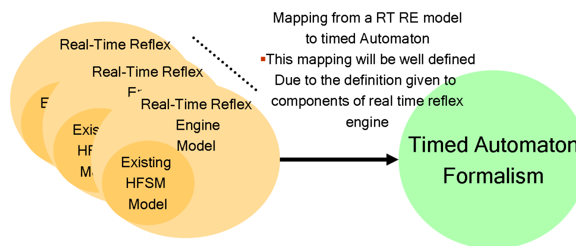


Figure 20: The previous reflex engine has been augmented to allow analysis within the semantic domain of networks of timed automatons.

The next section details the real-time reflex and healing framework. Figure 20 illustrates the approach at a high level. The first major addition in the new framework was the concept of timers, which generate timeout events after a set period. This timer, when used in parallel with an event-based reflex engine, enables the reflex engine to mitigate time-based faults. Then, by using the network of timed automatons as a semantic domain for the real-time reflex and healing framework, we are able to use timed automaton model checkers for verifying real-time properties.

The Real-Time Reflex and Healing Framework

A real-time reflex and healing framework has a hierarchical structure identical the discrete reflex and healing framework discussed earlier. However, each manager model is now more capable than a simple event-based state machine.

In order to provide an overview of the workings of the real-time reflex engine we need to explore the notion of execution threads. For the moment, let us assume that each manager runs on a dedicated node, which has some kind of an operating system. The reflex engine will use the threads leased from the operating system. It is ideal if this operating system is a real-time operating system; this abstraction, however, will work even with non real-time operating systems, though in such cases any real-time guarantee will depend upon the assumption that no RH task will be delayed for an arbitrarily long time. It should be noted that this thread abstraction would still hold true if more than one manager is installed on a node.

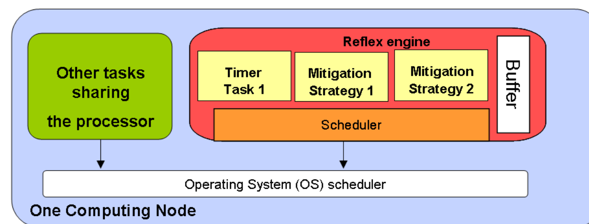


Figure 21: A computing node in the real time reflex and healing framework.

Figure 21 describes the basic structure inside a managing node. The lowest layer is the operating system scheduler. There can be multiple fault mitigation strategies (state machines) running on that node. In an ideal situation, the operating system is able to provide an execution thread to each of the strategies. If there are only a limited number of threads, then the reflex-engine scheduler is needed. A reflex-engine scheduler has a fixed number of threads that it can lease from the operating system to then schedule the different mitigation strategies. A buffer provides temporary

storage of input fault events that the reflex engine scheduler schedules with the corresponding mitigation strategy. Finally, a number of high-priority timers measure the time lapsed for the mitigation strategy.

We can now detail these components and describe their semantics using a model of networked timed automatons.

Real-time Fault Mitigation Strategies and Timer Tasks

A real-time reflex engine uses fault mitigation strategies as reflexes. Formally, we can define a strategy as:

Definition 2 (Fault mitigation strategy) *A fault mitigation strategy used in a real time reflex engine is state machine based failure management logic, $\mathscr{S} =< Q, q, Enable, Z_i, Z_{\tau^+}, \mathscr{T}, R, Z_o >$, where*

- *Q is the set of all possible states.*
- *$q \in Q$ is the initial state.*
- *$Enable \in \{True, False\}$ is a Boolean flag used to enable or disable a strategy.*
- *Z_i is the set of all possible external events (input) to which the strategy is subscribed. Every strategy has two special events $start \in Z_i$ and $finish \in Z_o$, which are used to communicate with the scheduler.*
- *Z_{τ^+} is the set of all events generated due to passage of time.*
- *$\mathscr{T} \in Q \times (Z_i \cup Z_{\tau^+}) \times Q$ is the set of all possible transitions that can change the state of the strategy due to passage of time or arrival of an input event.*
- *Z_o is the set of all possible output events and mitigation actions generated by the strategy. For sake of brevity, one can abstract the mitigation action as an event.*

- $R : \mathscr{T} \rightarrow Z_o$ *is the reflex action map which generates an output event every time a transition is taken.*

The time-based events associated with any strategy can be divided into two groups, internal and external. Internal time-based events are generated while the strategy is executing on a thread. If the strategy needs to measure time across two instances of execution, it can start a timer task, which shall generate the time-based event after the required passage of time. Thus, fault mitigation strategies use timer tasks to measure time for them and generate an event that wakes them up.

Definition 3 (Timer task) *A timer is a task that executes non-preemptively until completion. This task generates a timeout event that is subscribed to by the corresponding fault mitigation strategy. We call the execution time duration of the timer as the lifetime of the timer.*

These strategies and timer tasks have a one-to-one mapping with a corresponding timed automaton model. For the purposes of analysis, we restrict this lifetime of the timer to the set of dense but countable positive rational numbers. This is necessary because we wish to use the timed automaton model to govern the semantics of a timer task.

Timer timed automaton

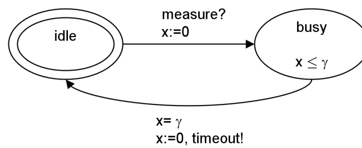


Figure 22: Timer timed automaton.

A timer task is mapped to a timed automaton, which has two locations, $idle, busy$ and a single clock variable x. We call this timed automaton a timer timed automaton. The transition from idle to busy is guarded by a synchronized event $measure?$, which allows the timer to start working. The invariant associated with $busy$ is given by $x \leq \gamma$, where $\gamma \in \mathbb{Q}^+$ is the time interval which

has to be measured for the requesting strategy. The transition from $busy$ to $idle$ is guarded by constraint $x = \gamma$. Upon this transition from $busy$ to $idle$, the timer generates an output action $timeout!$, which is an input event for the requesting strategy. Figure 22 shows this model.

Strategy Timed Automaton Model

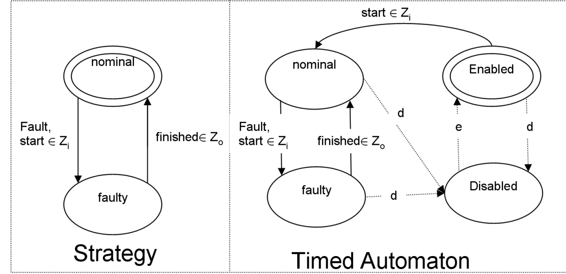


Figure 23: An example strategy and its corresponding timed automaton.

Consider a strategy of a reflex engine. We can formulate an equivalent timed automaton model TA=$< \Sigma, S, S_0, X, I, T >$ in the following way:

- States of the strategy Q are mapped to locations of the timed automaton model. Furthermore, two new locations $\{Enabled, Disabled\}$ are added to model the cases when a strategy is enabled or disabled. Only when the strategy is enabled can it be triggered by the presence of events in the buffer and dispatched to a thread by the scheduler. Thus $S = Q \cup \{Enabled, Disabled\}$.

- Initial location of the timed automaton, S_0 is derived from the initial states of the strategy plus the new locations $\{Enabled, Disabled\}$. Thus $S_0 \in q \cup \{Enabled, Disabled\}$.

- In order to measure the time spent by the strategy in each of its locations we use a clock variable $x \in X$. This clock is used to generate internal time-based events while the strategy is executing.

- All invariants and guards to TA are specified by a time specification used for the generation of internal time-based events that can trigger a transition.

- Output actions of the timed automaton are the same as the set Z_o and are all written with a suffix of '!'. The input events are same as the set $Z_i \cup Z_{\tau^+}$ and are written with a suffix of '?'. Therefore, $\Sigma = Z_i \cup Z_{\tau^+} \cup Z_o$.
- Transitions T of the timed automaton are a union of transitions of the strategy i.e., $\mathscr{T}$ and transitions due to $Disabled$ and $Enabled$ locations. To model the ability of disabling the strategy in any of its states, we specify transitions from all locations of the timed automaton to the $Disabled$ location. Moreover, one transition from $Disabled$ to $Enabled$ location is specified to model the enabling of the strategy. Lastly, a transition is added from $Enabled$ location to the initial state q of the strategy.
- The reflex action map R of the strategy is created by mapping the output events Z_o to the corresponding transition of the timed automaton as output actions.

Figure 23 shows an example of a strategy which has two states *nominal* and *faulty* and its corresponding timed automaton model. Note the extra states and transitions added in the timed automaton model. The events d, e are used by the supervisory reflex engine to disable or enable the strategy.

Buffer

A buffer is a FIFO, which provides an interface for events that are coming into a reflex engine. Since a reflex engine has a number of strategies that might be more than the number of available threads, the scheduler has to arbitrate as to which strategy should execute. For this purpose, the scheduler can sort the buffer and allow execution of only those strategies that have a subscribed event at the front of the buffer.

There are several issues associated with a buffer that affect the correct operation of the RH framework. If the buffer is not of adequate size and the incoming rate of events is greater than the rate at which they are being processed, the buffer might drop some events. Such mishaps will directly affect the safety property of the real-time framework.

Buffer Timed Automaton Model

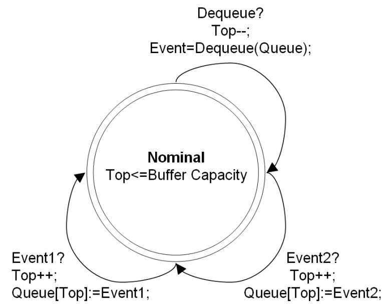


Figure 24: Buffer timed automaton listening to two different events. More events can be added to the model by creating new self-transitions. The event, $dequeue$, is used to pop the event from the front of the queue.

Buffer timed automatons have an associated shared array of integers, called a *queue*. In order to identify events in the model, we map them to the domain of positive integers. The Buffer TA (see Figure 24) has only one location and no clocks. It uses an internal integer variable, Top, to keep track of the queue size. An event is added to the queue by using a self-transition that also increments Top. A maximum queue size is set to drop events if the queue becomes full, however, such a situation is undesirable.

Scheduler

Scheduler is an important component of a real-time reflex engine. Based on the number of threads available from the main operating system, the scheduler governs and arbitrates the execution of timer tasks and strategies. It picks up the input events from the buffer and then executes a number of strategies, which in turn will perform the required mitigation actions and generate some output events. We say that a strategy is triggered when the event to which it is subscribed is present in the buffer. It can be defined as follows:

Definition 4 (Scheduler) *A scheduler, S, maps the input events in the buffer to the corresponding strategies. It uses an arbitration scheme to sort the buffer based on the priority of the notification events. It then executes a maximum m (number of threads leased from the operating system)*

strategies based on m number of events from the head of the buffer. A number of priority schemes can be used with the notification event. The simples being equal priorities. In such a case, the scheduler will never sort the buffer and always pick m events from the head of the buffer.

Scheduler timed automaton model

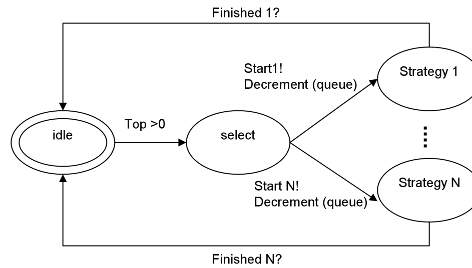


Figure 25: A scheduler timed automaton, which has only one thread. The select state is the one in which the scheduling decision is made.

The semantics of a real-time reflex engine scheduler is a network of m timed automatons, m is the number of available threads. Each of these individual timed automatons is identical to the one illustrated in Figure 25. Working in parallel, one of these timed automatons keeps control of execution on one of the threads available to the scheduler. The number of threads that are available for the strategies is equal to the number of threads leased from the operating system minus the number of threads for the timers (timers are non-preemptive and must have a thread for their executions).

These timed automatons start from the idle location. When the buffer is non-empty, the location changes to select, which implements the scheduling policy by sorting the queue with a given priority associated with the incoming events. It then pops the event from the top of queue, transitions to the corresponding strategy location and passes the thread of execution to the strategy by using the start event. When the strategy finishes execution, it generates a finished event that returns the timed automaton to the idle state.

With all its subcomponents defined, we can now review the operation of a real-time reflex engine.

Real-Time Reflex Engine

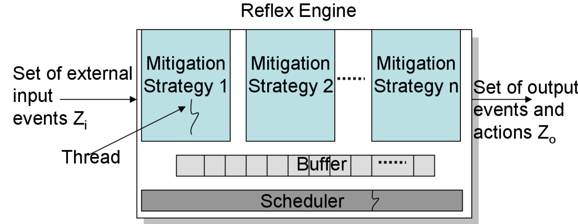


Figure 26: An overview of a real-time reflex engine

A Real-time Reflex Engine (RTRE), as shown in Figure 26, is comprised of multiple fault-mitigation strategies that take action in the presence of certain input events, which signify a fault condition. New strategies can be added to a real time reflex engine by an upper level external reflex engine or by direct human intervention. Moreover, a supervisory higher-level reflex engine can enable or disable a strategy. Formally, a RTRE can be defined as:

Definition 5 (Real-time reflex engine) *A real time reflex engine is a tuple* $E_r = < S, B, \mathscr{S}, \mathscr{S}', \mathscr{Z}_i, \mathscr{Z}_o >$*, where* S *is the scheduler ,* B *is a buffer,* $\mathscr{S}$ *is a parallel composition of all the enabled strategies.* $\mathscr{S}'$ *is the set of disabled strategies,* $\mathscr{Z}_i$ *is the set of all the possible inputs to a reflex engine and* $\mathscr{Z}_o$ *is the set of all the possible outputs generated by the reflex engine.*

All possible communication in and out of a RTRE is carried out through event sets $\mathscr{Z}_i, \mathscr{Z}_o$. These event sets are a union of corresponding input and output event sets of all strategies implemented by that engine. In a multi-level hierarchy as the one motivated in [12], some events are reserved for communication between reflex engines that have a manager/subordinate relationship.

Since a reflex engine is a parallel composition of its subcomponents, its timed automaton is also a parallel composition of the component's timed automatons. Hence, we can consider a real-time

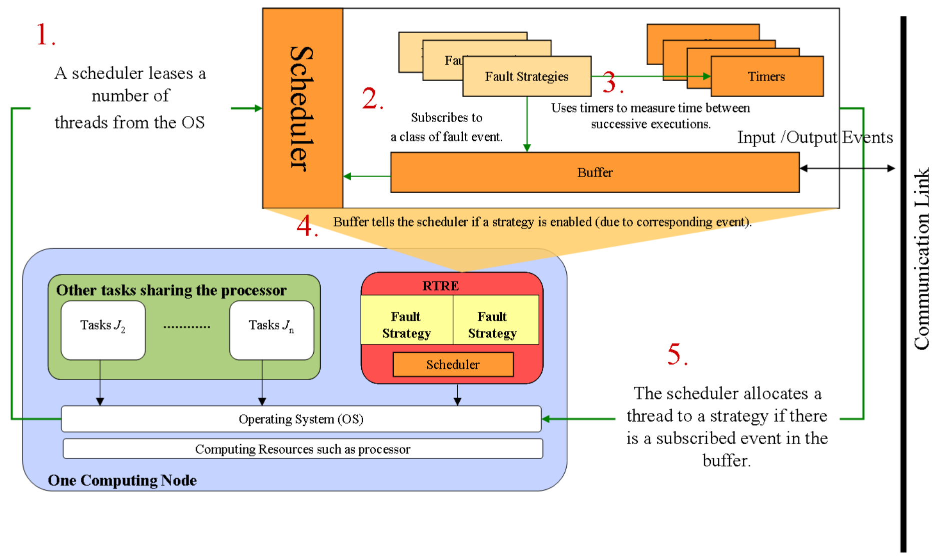


Figure 27: Overview of the chain of operations inside a computing node, which has a real time reflex engine.

reflex engine to be a network of timed automatons. In the same spirit, we can consider a number of reflex engines in a framework to be a larger network of timed automatons.

Sequence of Operations inside a Real-Time Reflex Engine

Figure 27 illustrates the operation inside a computing node, which houses a RTRE. Once it is operational, the scheduler leases a number of threads from the operating system. The fault strategies then inform the scheduler about their subscribed events through the publish subscribe mechanism.

When a new event arrives in a buffer, it signals the scheduler. If the buffer is full, then the event is dropped. This is an undesirable condition, which necessitates prior verification to check that such a situation will never arise. Once there is an event in the buffer, the scheduler waits for a thread to be available and then triggers the corresponding strategy and transfers the thread to that strategy. A strategy then takes the corresponding event and transfers the output event onto the communication link. If the strategy needs to measure time, it starts a timer task, which is non-preemptive and runs until completion or its termination, whichever happens first.

Analysis of Real-Time Reflex and Healing Framework

In the last section, we detailed the mapping between a real-time reflex and healing framework and a corresponding network of timed automatons. Since we wish to analyze the network's real-time properties, we can use timed computation tree logic (TCTL) [169, 100, 139] to specify the system properties. Then we can use model-checking tools to check the veracity of these properties against the system model. In general, these model-checking tools check the following properties:

- *Reachability:* These sets of properties deal with the possible satisfaction of a given state-based predicate logic formula in a possible future state of the system. For example, the TCTL formula $E\Diamond\phi$ is true if the predicate logic formula ϕ 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 ϕ. A restrictive form of invariance property is sometimes used to check if some logical formula is always true on execution paths of the system. An example of such a TCTL property is $E\Box\phi$.

- *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 action 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 ψ always eventually leads to a state satisfying ϕ.

It is possible to use these general classes of TCTL properties to map the real-time properties of a RH framework to corresponding TCTL properties for the network of timed automatons.

- *Liveness* of the RH architecture amounts to checking if the system has any deadlocks. In UPPAAL $A\Box not\ deadlock$ is the specification for this property.

- *Safety* of the RH architecture requires a check for any violation that might lead to not finishing a task on time. Safety violations also include checking for queue overflow conditions. One way for checking a safety property is to introduce an error location in all time automatons and force a transition to this error location if the queue overflows or if a time deadline expires. Then the checking of the safety property will amount to checking the reachability property $not\ E\Diamond\ error$.

- *Bounded Response Properties* can be formalized using the reachability property and liveness property. Suppose we have to check if $state = state1$ happens then $state = state2$ will happen within 5 time units. In order to formulate this property, we augment the time automaton with an additional clock called a formula clock, say z, whereby we reset its value to 0 on all the transitions leading to $state1$ and check if the liveness property $A\Box(state = state1 \rightarrow A\Diamond state = state2 \wedge z <= 5)$ is true or not.

In the next section, we will present a case study that will help in clarifying the ideas introduced in this paper so far. In this case study, we will use UPPAAL for analysis purposes.

Case Study

Typical HEP data processing systems experience a variety of both temporal and persistent faults. Persistent faults are those in which software or hardware components undergo terminal failure and need to be replaced. Temporal faults, however, are sporadic in nature and are not necessarily indicative of direct failure of the software or hardware. One example of temporal faults is data stream corruption.

Our approach toward designing tools and techniques for advancing the capabilities of HEP data processing systems has included the means for non-experts (physicists and plant engineers) to easily design and deploy fault mitigation behaviors, which are specific to their domain of experience. Systems that provide this capability have been shown to be beneficial for HEP computing [170].

However, with this great benefit comes the risk that non-expert users might create behaviors, which may, at best, cause unexpected behavior or, at worst, be harmful to the system.

The following case study presents a scenario in which an existing system [138] is augmented with user-defined behaviors for recovering from temporal faults due to data stream corruption. These behaviors are analyzed using UPPAAL [139], a tool for verifying behavioral models expressed as timed automata.

Experiment Setup

The data rate at which physics events are generated in an actual HEP experiment are on the order of several Tera Byte per second [161]. Since only a few of these physics events are of value to physicists, a number of *filters* are used to examine the events to determine which should be kept. Due to the nature of the environment, a number of these physics events may become corrupted as they flow through the system. This type of data corruption is classified as a temporal fault; the system should detect these corrupted physics events and ensure that the number of subsequently corrupted events is reduced. This is done by reducing the intensity (also referred to as the *luminosity*) at which the accelerator operates, thereby reducing the number of physics events produced by the experiment.

The architecture used in this case study is illustrated in Figure 28. The sub-components of the reflex and healing architecture have been arranged in a hierarchical fashion. The communication between the local managers, filters, data source and the regional manager is strictly event-based. For the sake of brevity, we have summarized all these events in Table 7. We will now detail the models of each of these components.

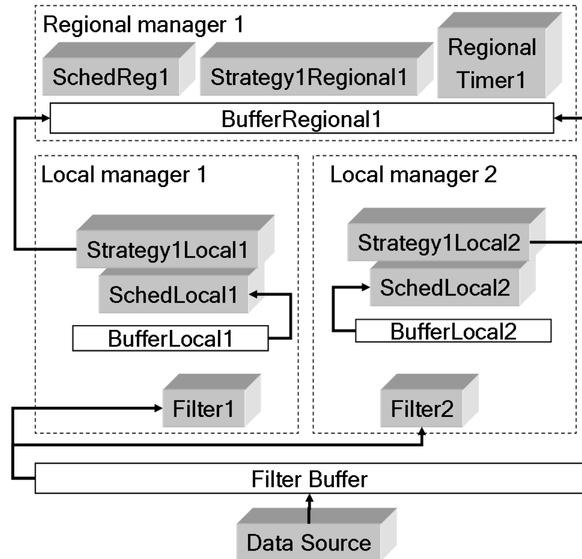


Figure 28: The architectural setup for the case study. There are two local managers, one data source, two filter applications and one regional manager.

Table 7: List of all the events being used in the case study.

Event Name)	**Relevance**
GoodData	The Data source produced a good physics event.
BadData	The Data source produced a bad/ corrupt physics event.
GetData	One of the filters is dequeuing the event from filter buffer.
F1BadData	Filter1 has detected a bad physics event.
F2BadData	Filter2 has detected a bad physics event.
BufSchedLoc1	Scheduler of local manager 1 is dequeuing the local buffer.
StaStra1Loc1	Instruction from local scheduler 1 to start the first local strategy.
StopStra1Loc1	The local strategy 1 is signaling its completion to its scheduler.
L1R1BadData	Signal from local manager 1 to its regional about a detection of bad data by its Filter.
BufSchedLoc2	Scheduler of local manager 2 is dequeuing the local buffer

Continued on next page

Table 7 – continued from previous page

Event Name	Relevance
StaStra1Loc2	Instruction from local scheduler 2 to start the first local strategy.
StopStra1Loc2	The local strategy 2 is signaling its completion to its scheduler.
L2R1BadData	Signal from local manager 2 to its regional about a detection of bad data by its Filter
BufSchedReg1	Regional Scheduler is dequeuing the regional buffer.
StaStra1Reg1	Instruction from regional scheduler to start the first regional strategy
StopStra1Reg1	Signal from regional strategy to the scheduler that it is relinquishing the thread.
StaReg1Timer1	Signal from regional strategy to the timer to start measuring time.
StopReg1Timer1	Signal to stop the timer, issued either by the timer itself after the set time has elapsed or by the regional strategy if it no longer needs to measure time.
Reg1DataSource	A mitigation instruction, in response of the fault condition, from Regional manager to the data source to change its behavior.

DataSource

A data source is used to mimic the physical data production mechanism of a high-energy physics experiment by generating simulated data for all the particle collisions that would normally happen in the particle accelerator. These data are referred to as *physics events*. The data source is true to the behavior of the actual system in that sets of physics events are generated at a fixed periodic rate, with a variable number of events per set. A control parameter of the data source is exposed which allows the number of physics events generated per unit time to be varied.

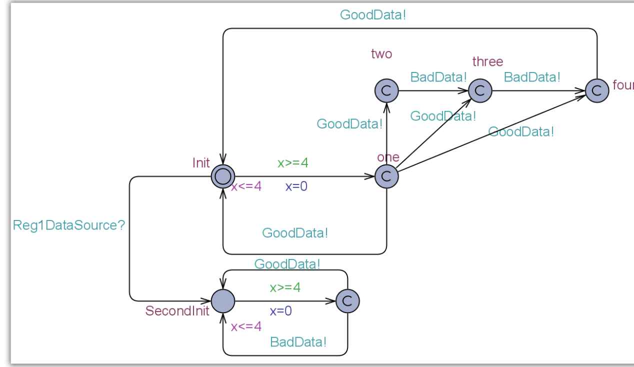


Figure 29: The data source generates data every four time periods. The number of data events generated varies over periods.

Figure 29 is the timed automaton model constructed in UPPAAL for the data source. Its initial state is $Init$. In its normal model of operation, after every four time units - measured by using the clock x - a variable number of physics events are generated. There are two categories of these physics events, good data and bad data (see Table 7).

In case a temporal fault is detected such that the number of bad data events generated has to be reduced, the data source can be instructed by using the event $Reg1DataSource$ to change its luminosity and move to the $SecondInit$ location.

$FilterBuffer$

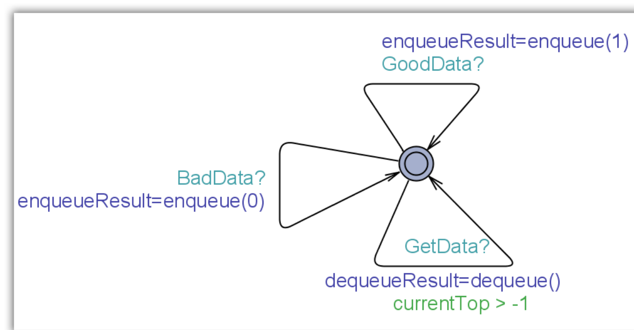


Figure 30: The filter buffer stores the generated data source events.

A buffer is required to store the generated physics event before they can be processed by a filter application. As shown in Figure 30 buffers use two methods called enqueue and dequeue to push

and pop physics events into the queue. The maximum size of the queue is modeled by a fix integer constant (not shown in the figure). The filter applications uses the get data event to access the top of the queue.

It is evident that the overflowing of the queue is undesirable as it might lead to a loss of important physics events. Therefore, one of the key properties that we will analyze in this case study is to check if the buffer queue overflows. This will be done by checking if the Boolean variable $enqueueResult$ becomes false.

Filter Applications: $Filter1$

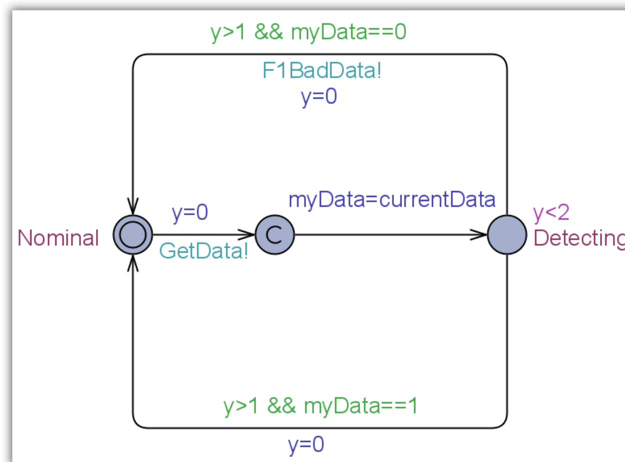


Figure 31: There are two filters in this setup. Both filters consume the data from the filter buffer and detect whether it is a good physics event or a bad physics event.

The filter applications, filter1 and filter2, work concurrently to process the physics events present in the filter buffer. Figure 31 illustrates the timed automaton model for the first filter. After obtaining the data from the filter buffer, a filter stores the physics event into a temporary local variable and then executes the detection algorithm. It has been assumed in the model that the filter takes between $[1, 2]$ time units in order to detect if the physics event is good or bad. If the physics event is bad, the filter generates an event, $F1BadData$, to the buffer of its local manager, and resumes work with the next physics event.

Filter2 is identical in its model to filter1 except that it generates ,$F2BadData$, and sends it to its local manager, which is different from the first filter's local manager. Together, in the worst case, the filters can process 2 data events every 4 time units.

Local Managers: $Local\ Manager\ 1$

Recall that a local manager has a scheduler, a buffer, and one or more strategies. This case study considers two local managers. In this section, we will describe the first local manager's components. The second local manager is identical to the first with the exception that it has its own corresponding events as described in the Table 7.

Local Manager's Buffer: $BufferLocal1$

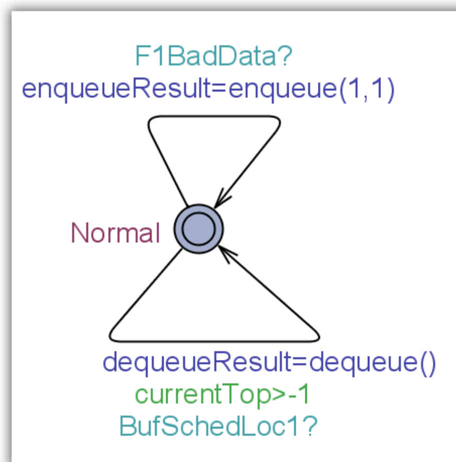


Figure 32: The buffer of one of the local managers.

The local manager's buffer is used to store the events, which are subscribed to by the manager's strategy. The model of a local manager's buffer, illustrated in Figure 32, is similar to that of the filter buffer, with an enqueue function call for each event that is being added. The arguments passed to this function are the ID of the event and number of strategy, which has subscribed to the event. This buffer's communicates with the scheduler, which pops the queue and executes the corresponding strategy.

Local Scheduler: $SchedLocal1$

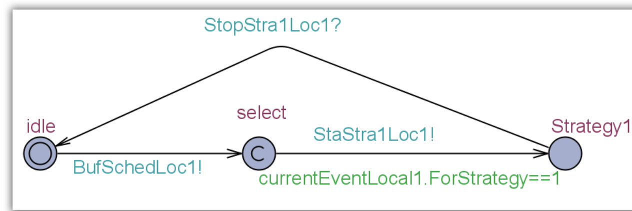


Figure 33: The local scheduler dequeues the local buffer and starts the appropriate strategy. This scheduler only has one thread. Therefore, only one strategy can be executed at a time.

In this case study, the scheduler only has one thread on which it can execute a strategy. Figure 33 illustrates the scheduler model. It uses the $BufSchedLoc1$ event to obtain the current event from the queue and then starts the strategy - in this case there is only one possible strategy. It returns to its idle state in which the strategy generates the $StopStra1Loc1$ event, signaling that the strategy has finished its execution and has relinquished the thread.

Local Strategy: $Strategy1Local1$

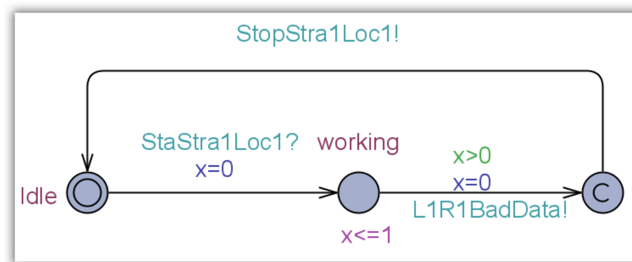


Figure 34: The local strategy. The local scheduler starts this strategy.

Temporal data corruption faults can arguably be mitigated at the local level as well as the regional level. However, we will show in the analysis that if one attempts to detect and mitigate a temporal fault at a local level there is a chance of missing a subsequent temporal fault. It is due to this reason that we chose to implement the mitigation strategy at the regional level. Therefore, the local strategy has been reduced to simply notify its regional manager whenever its filter reports a

bad physics event. Figure 34 illustrates this local strategy for the first local manager. The model captures the possibility of a worst-case delay of one time unit at the local strategy level before it can generate the $L1R1BadData$ event to notify the regional manager.

Additional local strategies can be added by including an additional start strategy state to the local scheduler.

Regional Manager: $Regional\ Manager\ 1$

The components of a regional manager are identical to a local manger, the only difference being its hierarchical position. We will now describe these components.

Regional Buffer: $BufferRegional1$

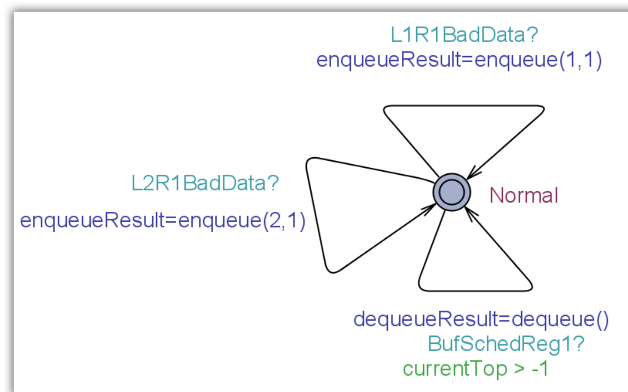


Figure 35: The regional buffer stores the events communicated by its local managers.

Figure 35 illustrates the model for the regional buffer. It stores the events communicated to it by the local managers. In this case these events are $L1R1BadData$ and $L2R2BadData$. The former event signifies the detection of a bad data by the first local manager, while the latter is the signal from the second local manager.

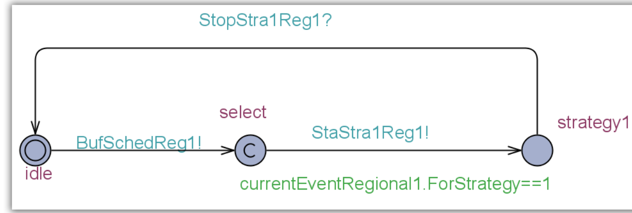


Figure 36: The regional scheduler also works like the local scheduler. It consumes events from the regional buffer and starts the appropriate regional strategy.

Regional Scheduler: $SchedRegional1$

The regional scheduler, shown in Figure36, is identical in its operation to the local schedulers. It has one thread at its disposal, which it uses to execute a strategy depending upon the event popped from the regional buffer.

Regional Strategy :$Strategy1Regional1$

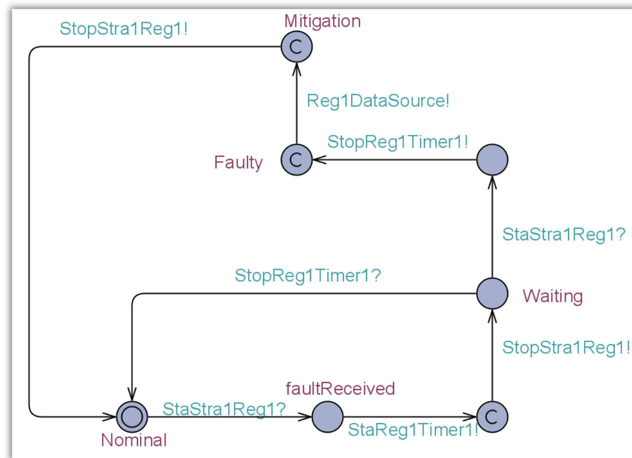


Figure 37: This regional strategy issues a mitigating action if contiguous bad data events arrive before the timer event arrives. This mitigating action changes the parameters of the data source effectively reducing the number of physics events generated per cycle.

The regional strategy, shown in Figure 37, detects a region wide fault and then issues the mitigating action as an event, $Reg1DataSource$. The fault is assumed to have occurred if the regional manager receives two bad data events in any particular order from its local managers within a period of two time units.

As shown in the Figure 37, this strategy starts its operation in the nominal state. When the scheduler wakes the strategy, it moves to the fault received state and starts a timer to measure two time units. After starting the timer, the strategy moves to a waiting state and relinquishes its thread back to the scheduler. It the timer sends a $StopReg1Timer1$ event before the next start event from the scheduler, the strategy resets and moves back to the nominal mode of operation. However, if a second start event from scheduler is received before the timer's event, the strategy moves to a faulty state. This is because two bad physics events have been detected within two time units of each other. Then the strategy issues a mitigation action using the $Reg1DataSource$ event to tell the data source to reduce its luminosity and hence reducing the number of bad data being generated.

Regional Timer: $RegionalTimer1$

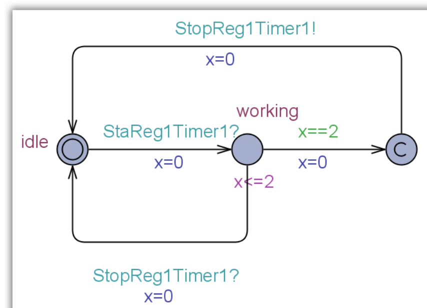


Figure 38: This timer is used by the regional strategy as a stopwatch set for a period of 2s to measure time elapsed between two bad data. It is necessary to use the timer because the regional strategy relinquishes the thread between the two contiguous arrivals of bad data.

Figure 38 describes the model of the non-preemptive timer used in this case study. Once started by the regional strategy, this timer generates a $StopReg1Timer1$ after two time units, unless it is stopped.

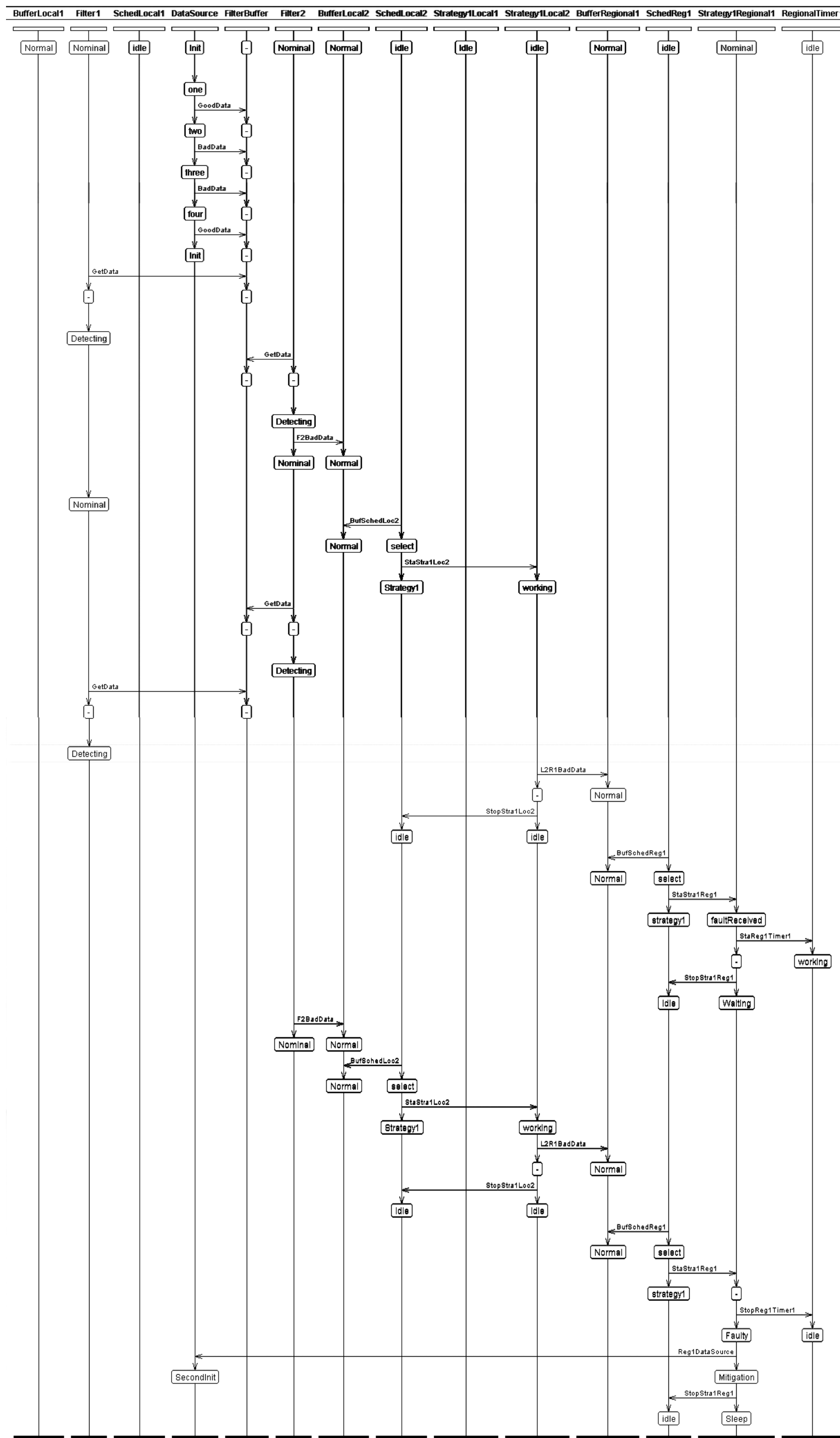


Figure 39: A simulation of a temporal fault and its mitigation is produced by UPPAAL.

Simulation

Figure 39 details a simulation trace as obtained from UPPAAL for the temporal data stream corruption fault and the corresponding mitigation. In the figure, the time line flows from the top to bottom. Trace starts with the production of four physics events in the following sequence: $GoodData$, $BadData$, $BadData$, and $GoodData$. These events are first enqueued into the filter buffer. Then, filter1 synchronizes with the filter buffer by using the $GetData$ event - in this case $GoodData$ - and finally transitions to the $Detecting$ location. After a brief delay, filter2 also pops an event from the filter buffer - $BadData$ and moves to the $Detecting$ location. Once it completes the detection process, filter2 informs its local manager about the bad physics event by using an $F2BadData$ event and then it gets another physics event. Since this next physics event is also a corrupt, it again detects it and raises another $F2BadData$ event. Simultaneously, filter1 detects that its physics event was good and transitions to $Nominal$ location and gets another physics event.

Once the second local buffer becomes nonempty, its scheduler wakes up the local strategy, which raises the $L2R1BadData$ event and enqueues it into the regional buffer. At this time, the regional scheduler dequeues its buffer. It then starts the regional strategy. The regional strategy moves to the $faultReceived$ location and starts the regional timer and proceeds to the $waiting$ location. When the second $F2BadData$ event reaches the local buffer, the local manager again raises an $L2R1BadData$ event, which causes the regional scheduler to start the regional strategy again. At this time the regional strategy is in the $Waiting$ location and the regional timer has not sent the $StopReg1Timer1$ event, yet. Thus, the regional strategy moves to the $Faulty$ location. Then the strategy moves to the $Mitigation$ location, issues a $Reg1DataSource$ event and causes the data source to change its luminosity and hence reduces the number of contiguous bad data events.

Analysis and Verification

In the subsequent subsections, we will present the properties, which we verified for this case study. For each property, we will provide the following details:

- *Specification:* The formal specification of the property in TCTL.

- *Description:* A textual explanation of the property.

- *Result:* The result of the verification process as obtained with UPPAAL on a $3GHz$ computer with $1024MB$ RAM running Linux.

- *Computation Time:* Time required to compute the results.

- *Analysis:* A textual explanation if the property is not satisfied.

- *Correction of Design:* If applicable, we provide a corrected design that will satisfy the concerned property.

Feasibility Analysis: Why not a Local Temporal Fault Strategy?

One might argue why we need to implement a regional strategy to mitigate the data stream corruption. Why not use two local strategies, one for each local manager instead?

As an experiment, we demoted the current regional strategy and replaced the two local strategies with this regional strategy. We then checked the setup against the following property:

Specification:

A□(DataSource.two→A◇(Strategy1Local1.Faulty || Strategy1Local2.Faulty))

Description:

Signature of contiguous bad data fault is the occurrence of two consecutive $BadData$ events in the filter buffer. This can only happen if the Data Source is in its location “two” (see Figure 29). If we consider that the current local strategies are replaced by the current regional strategy (see Sects. IV and IV), then we can say that the fault will be detected when either of the local manager is in its fault state. The formal specification mentioned above checks for this fact.

Result:

The result of verification stated that that this property is not universally true.

Computation Time: 2 seconds.

Analysis:

The reason for this is because there can be a case when two contiguous faults from the filter buffer can be picked by different filters and hence can remain undetected by their local manager.

Therefore, we decided to promote this strategy as a regional strategy.

Regional Strategy Will Always Catch Data Fault

This analysis furthers the property check mentioned in the previous section. The architectural setup detailed previously describes the regional strategy. In order to check if this strategy will always work, we checked the following property:

Specification:

$A\Box$DataSource.two$\rightarrow A\Diamond$ Strategy1Regional1.Faulty

Description:

This specification is similar to the one mentioned in Sect. IV. It should be noted that the verification of this property would not have been possible without a timed automaton model. This is because the definition of the fault requires a measure of time.

Result:

This property was universally true.

Computation Time: 2 seconds.

Analysis:

All contiguous bad physics events will be mitigated.

Liveness: Deadlock check

It is important to check that the designed system will not end in deadlock. For this purpose, we verified the model against the following property.

Specification:

A□ not deadlock

Description:

The system will never deadlock.

Result:

The result of this check was false.

Computation Time: $3 - 4$ seconds.

Analysis:

The UPPAAL verification engine generates a counter example in the case where a property is deemed false. Upon reviewing the counter example, we found the problem. Currently, the data source can generate two pairs of contiguous bad physics events before the first one is detected by the regional strategy, which then issues a $Reg1DataSource$ event to lower the luminosity of the data source. The deadlock happens when the regional manager detects the second pair of contiguous bad data events and it again tries to synchronize with the data source by using the event $Reg1DataSource$. However, the data source at this time is in the $SeondInit$ location and cannot synchronize the event from regional manager.

Correction of Design:

To correct the deadlock, we modified the regional strategy to ignore the second bad data. The new strategy is shown in Figure 40. Although this is modification is not complex, it still underscores the benefit that the verification provides in correcting bad mitigation strategies.

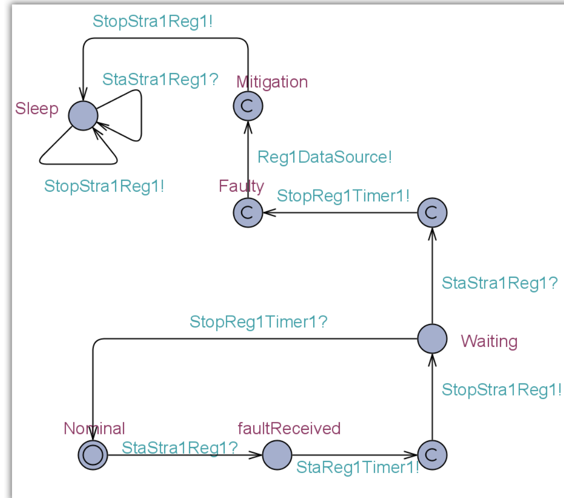


Figure 40: The regional strategy ignores the second false alarm by transitioning to sleep state.

Safety: Buffer Queue Will Not Overflow

We had mentioned earlier that it is imperative that the filter buffer never overflows. This is necessary to ensure that generated physics events are processed and not lost. To ensure that this never happens we have to verify the following property:

Specification:

A□ $FilterBuffer.enqueueResult$

Description:

The enqueue method call is used in the filter buffer to push the data event into the queue. This method returns a Boolean value stored in the variable $enqueueResult$, which is set to true if the action was successful. In the event of a buffer overflow, the return value is set to false. Hence, it is sufficient in the formal specification to ensure that $enqueueResult$ is always true.

Result:

This property is satisfied.

Computation Time: 4 seconds.

Analysis:

The buffer will never overflow.

Evaluation

In this case study, we verified a real-time reflex and healing setup with two local managers, and one regional manager. Overall, there were 14 timed automaton models in the network (see Figure 28), with five clock variables, one for the data source, two for the two filters, two for the two local strategies and one for the regional timer. With this model, the property verification took 3s, on an average.

It has been known that the model-checking tools for timed automaton suffer from the state-explosion problem. Specifically, it has been experienced that the performance of the verification algorithm degrades as the size of the timed automaton network increases. With the increasing number of clocks, the memory required to be able to explore all possible regions of the region automaton becomes extremely large. Therefore, a concern is how we will be able to scale our verification technique to systems with hundreds or even thousands of processors.

It is evident that we cannot verify the whole system model with several processors together. However, due to our architecture, which forbids any communication between peer managers and only allows communication between a parent and child node we can abstract the system at any particular node by only using its parent node and children node. It is our contention that by using a bottom to top approach and performing iterative verification checks for each level of hierarchy we can guarantee that the full system model will also satisfy the property. This is the basis for our future work.

Conclusion and Future Works

In this paper, we have shown that the real-time reflex and healing framework is an autonomic, fault mitigation framework, which with the semantics of networked timed automation can verify

real-time properties of the system. In particular, we have shown this to be useful in determining the liveliness and safety of the system, as well as the time-boundedness of the system's mitigation responses. One can further conclude that in the area of large scale computing systems, model-based analyses such as these are necessary to ensure that non-expert users do not introduce into system behaviors, which violate these properties.

As future work, we are investigating the possibility of using a discrete time model, which will help in reducing the computational complexity of verifying increasingly larger systems. This approach may lead to automated synthesis of new mitigation behaviors by considering the fault mitigation as a discreet timed supervisory control problem. This would further serve to provide self-managing systems, which are more robust, yet more easily augmented with new behavior.

CHAPTER V

DESIGNING A LIGHT-WEIGHT SYNCHRONIZED DISTRIBUTED MONITORING FRAMEWORK FOR COMPUTING CLUSTERS

Overview

This paper details the monitoring capabilities of the Scientific Computing Autonomic Reliability Framework (SCARF). This manuscript is currently under review with Letters of IEEE Task Force on Autonomic and Autonomous Systems [14]. Portions of this paper have been presented at fifth IEEE International Conference and Workshop on the Engineering of Autonomic Systems (EASe 2008) [15] and Twelfth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2009 [16].

Scientific applications are tuned to run at high performance. Any application that "disturbs" the operating environment can alter their performance negatively. This paper will describe the change in performance of an application when the sensors were enabled on cluster. To reduce the chances of intruding on application performance, sensor programs need to be small i.e. low CPU utilization and low memory footprint. Moreover, even though most sensor programs are periodic it should allow aperiodic sensors to be deployed on demand. Finally, one must allow new sensors to be created and old sensors to be deleted dynamically.

This paper also details a jitter-control algorithm that was used in monitoring framework. Periodic task invocations suffer from jitter, defined as the maximum deviation between start time and the release time among all instances of a periodic task [156]. It ranges on the order of microseconds even on hard real-time systems. However, on general-purpose operating systems such as Linux that are designed for good average performance, the typical jitter can range from a few milliseconds when the system is not busy to a few seconds when the system is busy.

Jitter in periodic sensor tasks is a cause of concern for a reliable subsystem. The global controller expects information for all sensors to arrive periodically according to the pre-arranged schedule. However, deviations from that schedule due to jitter can create false positives. For

example, a sensor such as heartbeat periodically sends a message to a recipient to indicate that a computing node is alive. Delay of heartbeat due to jitter can cause false positives.

Abstract

It is essential to ensure that the scientific computing clusters maintain a reliable working state most of the time to justify economics of operation. By construction, these clusters use general purpose operating systems such as Linux that are built for best average case performance and do not provide deterministic scheduling guarantees. Consequently, periodic applications show jitter in execution times relative to the expected execution time. Obtaining a deterministic schedule for periodic tasks in general purpose operating systems is difficult without using kernel-level modifications such as RTAI and RTLinux. However, due to performance and administrative issues kernel modification cannot be used in all scenarios.

This paper introduces a model-based hierarchical reliability framework that enables periodic monitoring of vital health parameters across the cluster and provides for autonomic fault mitigation. We discuss two challenges faced by autonomic reliability frameworks in cluster environments, (a) non-determinism in task scheduling in standard operating systems like Linux, and (b) need for synchronized execution of monitoring sensors across the cluster. As our contribution towards a solution, (a) we present a feedback controller based approach that runs in the user space and actively compensates periodic schedule based on past jitter; and (b) we show through analysis and experiments that this approach is platform-agnostic i.e. it can be used in different operating systems without modification and also that it maintains a stable system with bounded total jitter. Finally, we present experimental data that illustrates the effectiveness of our approach.

Introduction

Reduced cost of commodity computers and the advent of high capacity networks have made cluster computing economical. Clusters are used for solving complex problems that traditionally required the use of a supercomputer [20, 21]. Their advantage lies in the ubiquity of their components - commodity computers interconnected using high-speed networks such as Myrinet [5]

and Infiniband [6]. This in turn delivers high performance computing at a fraction of price associated with supercomputers. However, only applications that can be parallelized by splitting into a number of smaller job-units can truly reap their benefits.

Several such large computing clusters are maintained at Fermi National Accelerator Laboratory (FNAL), some of which are primarily reserved for solving lattice quantum chromodynamics (LQCD) computations[1] (see table 8). LQCD is a challenging field of study that employs large-scale numerical calculations in order to extract fundamental parameters of the standard model of nuclear physics from experiments [171].

Table 8: FNAL LQCD clusters

Cluster	CPU	Nodes	OS
QCD	2.8 GHz Pentium 4	127	Linux
PION	3.2 GHz Xeon	518	Linux
KAON	2.0 GHz Dual Opteron	600	Linux

One of the primary problems with clusters is to ensure that they maintain a working state most of the time to justify economics of operation. Unfortunately, reliability is not a prime design consideration for the hardware, operating systems, and middleware in commodity computers that are often built for higher performance per dollar. However, when these computers are used together in a cluster, reliable operation is expected.

Large analysis campaigns like the ones involved in LQCD are composed of a number of interdependent tasks that have to be successfully performed to complete the full workflow[2]. In such cases, failure of even a single node can halt progress on all nodes assigned to the job, resulting in loss of both time and money. These failures can be (but not limited to) power outages, hardware failures, non-responsive job-units, or even ambient cooling failures.

While applications can be written to be fault-tolerant, the development cost will be significantly greater and the performance lower. Instead, these applications are constructed for absolute

[1]See `http://lqcd.fnal.gov/`

[2]A workflow is a specification of a set of tasks or jobs to be performed, their execution order and their input/output dependencies

performance. When hardware failures occur, an application-specific set of tasks have to be done. For some LQCD jobs, for example, the distributed job is killed and restarted from a checkpoint. If done manually, this approach can be expensive, slow to respond, and limit scalability. Instead, an autonomic approach is required that can ensure that the resources of the cluster are used to best possible extent and achieve the best possible start to completion ratio of jobs, even in the presence of hardware/software failures. Such a system will also improve response time in problem solving.

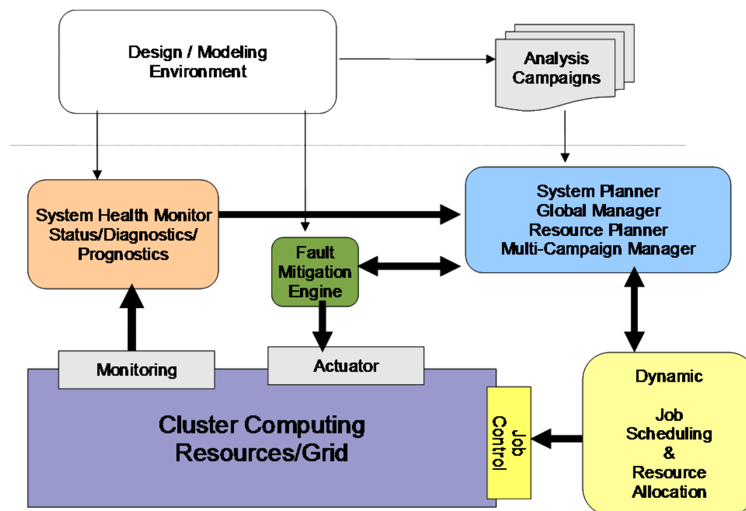


Figure 41: Overview of SCARF

The *Scientific Computing Autonomic Reliability Framework (SCARF)*, being developed by our research group for the LQCD computing clusters, is one such framework. Figure 41 illustrates its conceptual architecture. The monitoring and mitigation portions of SCARF are based on the hierarchical Reflex and Healing (RH) framework, presented in [11, 164, 12]. The basic components of this framework are distributed monitoring units, fault-mitigation units and a system wide planner for dealing with resource reallocation in case of severe failures. The primary monitoring units are *sensor programs* that execute periodically and on-demand across all nodes in the cluster. On the top is a design environment for deploying analysis campaigns and setting up monitoring and mitigation policies. The algorithm and components for the resource reallocation for a workflow have been presented in [172].

SCARF is a model-based reliability framework. It relies upon the use of a model-based specification of workflows, jobs, cluster resources and important monitoring parameters. It also provides for specification of models of mitigation policies based on a well-defined model of computations, along the same lines as those presented in [11]. Furthermore, it allows the realization of the monitoring and mitigation constituents of the framework using model transformations [173].

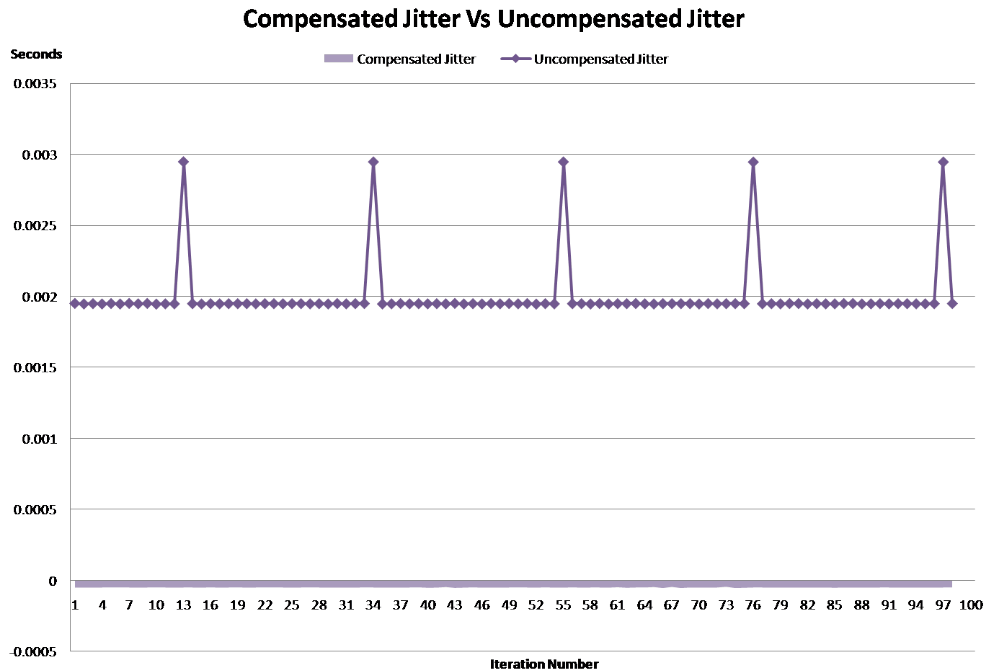


Figure 42: A plot of 99 samples of timing jitter obtained for a task with 1 second period on Red Hat Linux 2.6.9-67.

In this paper, we will present the monitoring portion of this framework. Primarily, the focus of this paper is concentrated on two main problems that affect SCARF's performance. We can summarize them as follows:

1. Operating systems such as Linux, widely used in cluster environments, are designed for good average performance rather than worst-case performance. This leads to a non-deterministic task scheduling, which in turn causes any periodic job running on that node to accrue *jitter between the expected start time and the actual start time*. Figure 42 shows a plot of 99

samples of timing jitter obtained for a task with 1 second period on Red Hat Linux 2.6.9-67. The regular pattern in uncompensated jitter suggests a common origin. Figure 42 also shows the compensated jitter profile for the same task when the periodicity of 1 second was implemented using the approach outlined in this paper with a smaller sleep cycle of 20 milliseconds. Average CPU utilization during the whole test was less than 1 percent. This jitter has detrimental effect on cluster management systems as they depend on periodic sampling and correlation of health status data from every node in the cluster. We will discuss this problem further in a later section.

2. *Synchronization between the various periodic sensor programs* running across the cluster is necessary to ensure execution of all instances happen at the same time on all nodes. This is required to prevent the worst-case scenario where a parallel application, waiting for synchronization across multiple nodes of the cluster, can be blocked forever. This can happen if at any time at least one of the nodes required for synchronization is not ready as it is executing the sensors of the reliability framework.

It should be mentioned that these problems are generic in nature and will affect any reliability framework designed for large clusters. In the latter half of this paper, we will present solutions to these problems by (a) using a discrete event based sensor scheduler, and (b) employing a feedback controller based approach that compensates for the scheduling jitter in non-real-time operating systems. Additionally, we will also present experimental data that illustrates the effectiveness of our approach.

An Introduction to SCARF

Figure 41 shows the basic components of the framework: distributed monitoring units, fault-mitigation units and a system wide planner for dealing with workflow re-planning. The benefit

of using the model-based approach is the possibility of policy verification as presented earlier in [13, 11]. In this paper, we will be limiting the description to the monitoring and mitigation portions.

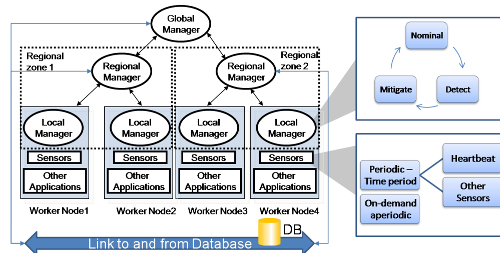


Figure 43: Implementing the reliability framework using hierarchical reflex engines.

This framework employs a hierarchical network of decentralized fault management entities called reflex engines. A reflex engine comprises of several state machines, which change states and perform actions associated with the transitions upon occurrence of certain predefined events. Figure 43 shows the deployment of reflex engines in the framework. To manage scalability, all reflex engines are divided into three logical hierarchical levels, which are global, regional and local.

Figure 43 describes the hierarchical aspect of SCARF. Local managers are the closest to a worker node in the cluster. In response to a mitigation command given by a superior level manager, they can deploy or remove various sensors on their node or even make changes in the configuration, such as a change in periodicity of a particular sensor. These sensors provide monitoring for different health parameters. In effect, they realize the “system health monitor” block from Figure 41. Local managers are also responsible for actuating the necessary mitigation action dictated by the regional manager.

Regional managers supervise and communicate with a number of subordinate managers that are in their area of observation. They also store all the sensory information received from their subordinates in a database to be used for historical correlation in future. A regional manager has a wider area of observation and can correlate diagnosis to ascertain if a problem is common to

a number of user applications and take coordinated mitigation action. Together, the local and regional managers realize the fault-mitigation engine block from Figure 41.

The head node of the cluster is usually designated as a single global manager and is used as a resource planner and the gateway to submit new jobs or plan the resources assigned to an existing job.

Sensors

The primary fault-detection entities are sensor programs, which periodically execute on the nodes in the cluster. Data is channeled from the sensors to the local manager, which sends it to the regional manager, where a built-in state machine is used for fault detection. This flow of data is achieved by using *Syslog-ng* - a next generation version of the popular protocol *Syslog* used for the transmission of event notification messages across networks [174]. The fault-mitigation commands are sent by the regional manager to a local manager by using UDP. In this case, UDP is preferred over TCP as it is stateless and does not require dedicated socket connections between the two managers, saving precious network bandwidth. Figure 44 illustrates this dataflow.

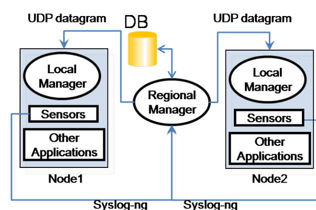


Figure 44: Dataflow of messages exchanged between managers

Table 9 contains the list of 14 sensors currently running on all nodes in the Pion and Kaon cluster (refer to table 8). One of the most critical sensors is the *heartbeat sensor*. It indicates the health of a node. If more than one contiguous heartbeat is missed then the regional manager checks the health of the node by sending pings to the local manager. If the ping is ignored, it marks the node in question as dead and reallocates its part of the job to another node. Readers are referred to [172] for the algorithm of workflow re-planning in case of faults.

A special sensor (number 13 in table 9) is employed to reload the executable code of other sensors if a configuration change is detected in their respective source code, for example, when the local manager changes the periodicity of sensors running on that node. Function of other sensors is obvious from their name. These sensors (except heartbeat) only report the value if it is over or below a certain threshold to reduce network traffic. This threshold is decided based on the historical correlation between faults and health parameter values at that time.

Table 9: Sensors in the framework

No.	Sensor	Period(sec)
1	CPU Fanspeed	10
2	Motherboard Fanspeed	10
3	CPU Temperature	10
4	Motherboard Temperature	10
5	Aggregate CPU Utilization	10
6	CPU Utilization per process	10
7	Hard Disk Utilization	10
8	RAM and Swap Utilization	10
9	Aggregate RAM Utilization	10
10	Aggregate Swap Utilization	10
11	CPU Voltage	10
12	Motherboard Voltage	10
13	Monitor configuration changes	60
14	Heartbeat	300

Since the sensors execute frequently, it is necessary that they are least intrusive i.e. the average CPU utilization due to sensor measurements should be minimal. In order to achieve this, the sensor scheduler is based on a technique presented in [163] used for simulating discrete event systems. This scheduler is implemented as a single process with one thread to ensure the small CPU load, which executes periodically.

Sensor Scheduler

Algorithm 1 describes a typical discrete-event based scheduling algorithm as seen from the perspective of a single node. All periodic sensor programs have two variables, a time period

(P) that is constant for a given sensor, and a current clock value (C) that is initialized to the respective time period (P) of that sensor. The prominent feature of this algorithm is the scheduling step in which the time for the next scheduled iteration is set. This time is set by making the sensor scheduler sleep for a time T, which is decided by evaluating the current clock values (C) of all sensors. Sleep can be realized in standard operating systems, such as Linux, either by a system call or by using an alarm signal and an appropriate signal handler [175]. On any given operating system, the resolution of sleep implementation used in the sensor scheduler determines the minimum periodicity that can be set for a sensor program.

Algorithm 1 Sensor scheduler

Input: S {Set of periodic sensors. Each sensor has two variables: P(Time period), C (current clock Value)}
Pre Condition: $(\forall s \in S)(s.C = P)$
1: **loop**
2: Set $T = min(s.C|s \in S)$
3: Set Alarm for T {Alarm can be implemented by sleep or using signals in both Linux and Windows}
4: $(\forall s \in S)(s.C \leftarrow s.C - T)$
5: $(\forall s \in S)(s.C = 0 \implies Execute(s))$ (Note: assertion: $s.C \geq 0$)
6: $(\forall s \in S)(s.C = 0 \implies (s.C \leftarrow s.P))$
7: Run any aperiodic sensors if present.
8: **end loop**

At the start of an iteration, clock values of all sensor programs are decremented by the current value of sleep time, T. Subsequently, the scheduler runs any sensor program that has a zero clock value and updates its clock value to the respective time period.

Here we wish to distinguish two distinct terms, *iteration of the sensor scheduler* and the *sensor runs*. The iteration is the periodic tick of the sensor scheduler, while a sensor run happens when the sensor scheduler executes the program of a sensor during iteration.

Note that the precision of the periodicity for sensors completely depends upon the accuracy of the sleep function provided by the operating system. However, in many cases the standard Linux kernel provides no upper bound on the difference between the actual time of `sleep` compared to the requested time. Next two sections discuss these problems in detail.

Problems caused due to jitter

One of the foremost problems faced in a monitoring framework is the timeliness of sensor readings. Ideally, the nodes in the framework should have a real-time operating system (RTOS) to guarantee the performance and timeliness of sensor readings. However, usually clusters are built with standard version of Linux kernel as the chosen operating system, which is geared towards best average-case performance than the worst-case performance. Consequently, the standard Linux kernel does not guarantee deterministic task scheduling.

Non-deterministic scheduling can lead to unbounded jitter between the expected and actual time of the sensor reading and make the task of analysis engines even more difficult. This problem has been studied earlier by P. Marti et. al in [176]. They showed that the jitter can not only lead to performance degradation but can also lead to instability. For example, the regional managers cannot wait forever to determine the absence of a heartbeat from worker nodes. They have bounds on the maximum time that they can wait for before a mitigation action is executed. Consequently, false alarms might be issued if the heartbeat does not arrive within the stipulated time.

Moreover, unbounded jitter in sensor readings will eventually lead to a large skew in timestamps of observation that were taken at the same time. This can cause problems in analysis, where a global snapshot of the state of cluster at any particular time might be required.

In order to formalize the notion of jitter, consider a generic periodic task τ with a time period T. The ideal release time of such a task would be a sequence $< kT >_{k=1}^{k=\infty}$ relative to some time, t_ϕ. Let $s(kT)$ be the start time of kth instance of this task such that the start is delayed by $tj(kT)$ with respect to the expected start time, $t_\phi + kT$. Then total jitter, $tj(kT) = s(kT) - kT - t_\phi$. Now, consider the $k + 1^{th}$ instance of the same task. If the schedule is set by using sleep for T time, like in line 3 of algorithm 1, then it can be seen that the relative time difference between the two start times is greater than the time period, $s((k+1)T) - s(kT) \geq T$. In such a case, the total jitter will keep accumulating and increase with time.

At any point, $tj(kT)$ will represent all the delays accumulated over time till that point and will be equal to the absolute relative jitter until that instance, which is defined as the maximum

deviation between start time and the expected start time among all the instances of a periodic task. Figure 45 illustrates the effect of accumulating jitter. From the figure we can see that if nothing is done the sensor measurements will always be off from the expected time by some value. Two problems arise when this phenomenon of delayed sensor readings is seen from the perspective of the cluster: (a) sensor measurements are not exactly periodic any more. This affects analysis routines that rely on uniform sampling rate of sensor readings; (b) sensors are run at different times on the nodes of the cluster. This affects the performance of jobs that needs to synchronize across the cluster. We explain this problem in detail in the next subsection.

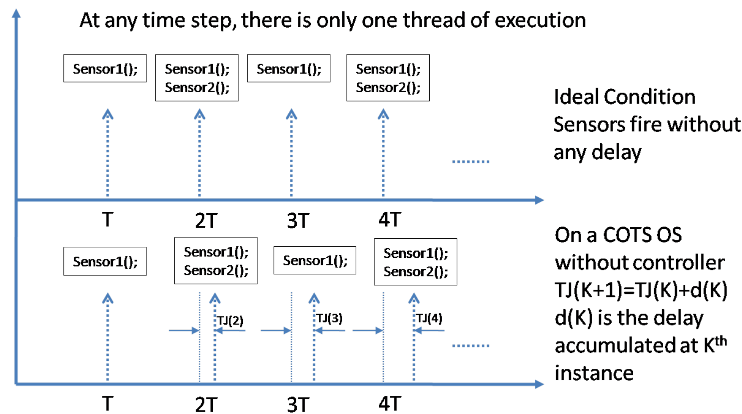


Figure 45: Periodic sensing will accumulate delay when used in a standard OS.

Impact of sensor synchronization on application performance

Typically, LQCD jobs use the Multiple Instruction Stream and Multiple Data Stream (MIMD) technique. In this model, different nodes execute in parallel on different data streams. However, the computation results from various nodes have to synchronize together in order to solve for boundary conditions. For example, in an application called MILC[3](MIMD Lattice computation collaboration) has to do a global synchronization at a rate of every 45 milliseconds. The requirement is that all nodes in the job must be ready at the time of synchronization i.e. the scheduler on each of those

[3]http://physics.indiana.edu/~sg/milc.html

nodes should not be running any other task at that time, otherwise the job is delayed till all the nodes are ready.

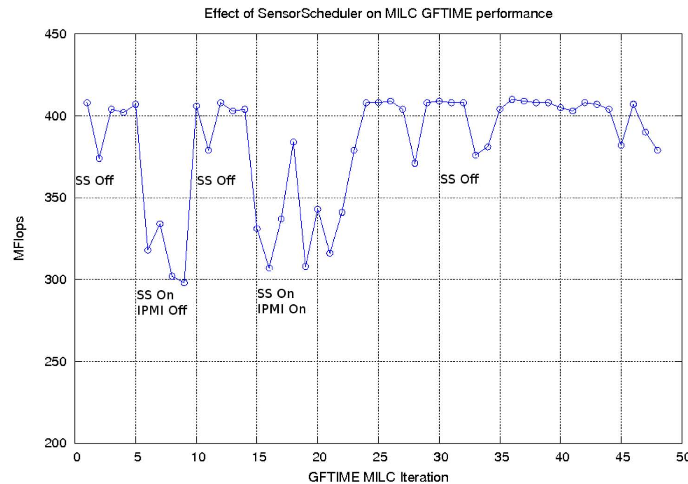


Figure 46: Loss in the performance of a MILC job when run in the presence of unsynchronized sensor schedulers across the cluster.

In the current setup, on an average a sensor takes 30 milliseconds for measurement. Moreover, to ensure fairness MILC jobs and the sensor scheduler runs with the same priority. Therefore, it is possible that a sensor run is scheduled such that it preempts the run of MILC task on the node and delays the global synchronization. In the worst case, it might be possible that at any time there is always at least one sensor running on a node, which will preempt the global synchronization of the MILC job across all nodes. In that scenario, the MILC job will be stalled forever. However, in average case we will see a performance drop because of the extra time spent by the MILC job in waiting for the synchronization. For example, Figure 46 illustrates the drop in MFLOPS achieved from an actual computing job executed across 256 nodes when the unsynchronized sensor framework was brought online on the cluster. In this figure, MILC GFTIME is the name of the application executing on the cluster. It can be seen that the performance of the job drops from around 400 MFLOPS to 300 MFLOPS when the sensor schedulers comes online across the whole cluster.

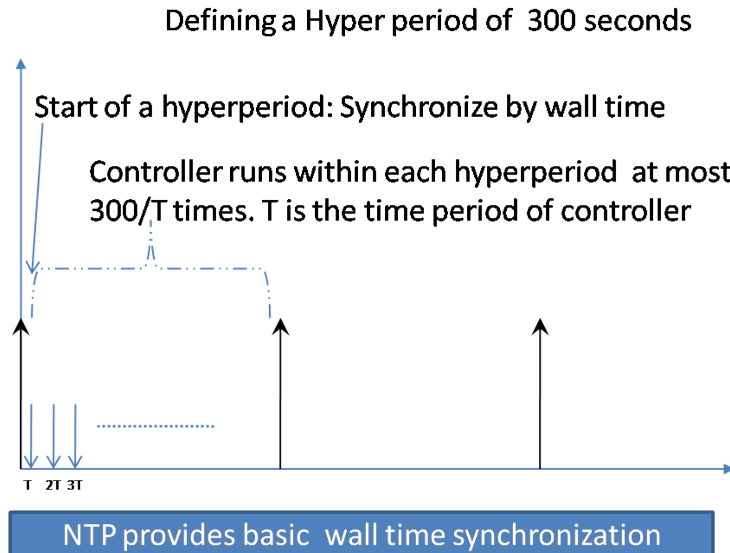


Figure 47: Relationship between a hyperperiod and ticks.

Solution Approach

Synchronization and jitter are two interdependent issues. Even if the sensor schedulers across the cluster are started at the same time, they will soon go out of sync because of the inherent jitter. Therefore, the solution for the two problems required an integrated approach. For this, we divide the time scale into hyperperiods and ticks between the hyperperiod as described in Figure 47. The ticks are instances in time when the sensor scheduler runs a sensor if its current clock value is zero (refer to algorithm 1). If more than one sensor is eligible for execution then the order of run must be deterministic. A lexical sort on their names is one way of achieving this determinism.

The start of the hyperperiod is the time when all sensor schedulers across the cluster reset their clocks and start a new cycle of sensor runs. In effect, this forces them to resynchronize at the start of each hyperperiod. The behavior of all sensor schedulers is such that they must wait for the hyperperiod signal when starting for the first time.

One of the existing facilities of Linux operating system is the Network Time Protocol (NTP). By using the global manager as the NTP server, we can ensure that wall clock times of all nodes in the cluster are within a few milliseconds of each other. This provides for an opportunity to use

a set time of day as the hyperperiod signal across the cluster. For experimental purposes, we chose every 300 second of the day as the hyperperiod. It should be noted that this decision requires a trade-off. If a very fine hyperperiod is used, say 20 seconds, the performance of the cluster will be reduced. However, if a very coarse hyperperiod is chosen, say 20 minutes, then there will be a large accumulation of jitter between hyperperiod synchronizations.

Between any two hyperperiods, the sensor scheduler uses the sleep to space out the ticks for sensor runs. However, even with this hyperperiod synchronization it is possible to accumulate jitter between the expected and actual time of the tick. In effect, this will lead to many unsynchronized sensor runs between the hyperperiod synchronizations. This is why we need a mechanism to control the uniformity of ticks.

Controller Design

Plant and Controller Model

The discrete-time plant model is given in equation 5.1. The state variable in this equation is the total jitter at any time sample, $tj(kT)$. $d(kT)$ is the finite jitter accrued during that sleep iteration. For brevity, we will drop the term kT and only use k.

$$tj(k+1) = tj(k) + d(k) \tag{5.1}$$

Empirically, we have observed that the disturbance during each sleep cycle is proportional to the average CPU utilization and the current priority of the process.

Feedback Controller

Feedback control has been successfully employed for a long time in analog systems. One of the basic feedback configurations is the proportional, integral and derivate (PID) scheme. Basic

principle of the PID control scheme is to act on the control variable through a combination of proportional action, integral action and the derivative action. The proportional action is proportional to the error signal, which is the difference between reference input and the feedback signal. Integral action is proportional to integral of error signal and the derivative action is proportional to the derivative of the error signal [68]. In discrete time systems, the integral and derivative components are approximated by trapezoidal summation and difference equation respectively. Discrete-time PID controller equation is given as:

$$
\begin{aligned}
c(k) =& K[e(k) + \frac{T}{T_i}\sum_{j=1}^{j=k}\frac{e(j-1)+e(j)}{2} + \\
& \frac{T_d}{T}[e(k) - e(k-1)]]
\end{aligned}
\tag{5.2}
$$

Where, T is the sampling time period, e is the error signal, c is the controller output, K is the proportional gain, T_d is the derivative time constant and T_i is the integral reset time constant.

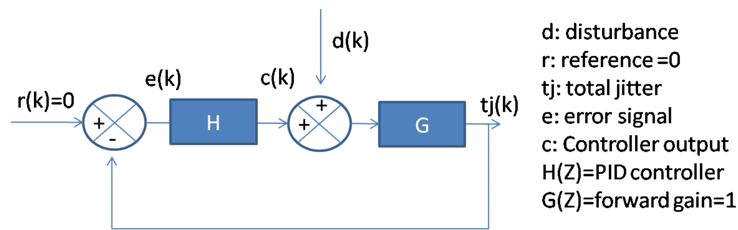


Figure 48: The feedback control loop for the sensor scheduler

Figure 48 shows the control loop for plant specified in equation 5.1. We implemented the controller using a first order system (PI controller) because we found that quick, sudden changes in current jitter values were causing instability in the derivative term.

The state space variable as mentioned in equation 5.1 is total jitter, $tj(k)$. The reference or the ideal value for total jitter is zero. Therefore, from figure 48 we can deduce that the error signal is given by $e(k) = -tj(k)$. The plant and controller equations are:

$$tj(k+1) = tj(k) + d(k) + c(k), \text{ where} \tag{5.3}$$

$$c(k) = -K[tj(k) + \frac{T}{T_i}\sum_{j=1}^{j=k}\frac{tj(j-1)+tj(j)}{2}] \tag{5.4}$$

In Z-domain the equations become:

$$C(z) = -[K_p + \frac{K_i}{(1-z^{-1})}]TJ(z), \text{ where} \tag{5.5}$$

$$K_p = K - \frac{K_i}{2} \text{ is the effective proportional gain} \tag{5.6}$$

$$K_i = \frac{KT}{T_i} \text{ is the integral gain} \tag{5.7}$$

Transfer Function of the Control Loop

Figure 48 shows that $C(z) = -H(z)TJ(z)$. From equation 5.5 we can write

$$H(z) = [K_p + \frac{K_i}{(1-z^{-1})}] \tag{5.8}$$

Further, we can write the transfer function, $TF(z) = \frac{1}{1+H(z)}$ as:

$$TF(z) = \frac{1}{1 + [K_p + \frac{K_i}{(1-z^{-1})}]} \tag{5.9}$$

For stability, poles of TF must lie within the unit circle in z-domain or $|z| < 1$. From equation 5.9, the characteristic equation for finding the poles is $1 + [K_p + \frac{K_i}{(1-z^{-1})}] = 0$. Therefore, the pole is at $z = \frac{1+K_p}{1+K_p+K_i}$. Since K_p and K_i are real, the stability criterion implies:

$$
\begin{aligned}
&-(1 + K_p + K_i) < (1 + K_p) < (1 + K_p + K_i) \\
&i.e. \ (1 + K_p + K_i) + (1 + K_p) > 0, \\
&and \ (1 + K_p + K_i) - (1 + K_p) > 0 \\
&i.e \ K_i > 0, \ and \ K_p \geq 0
\end{aligned}
\tag{5.10}
$$

Equation 5.10 implies that the effective proportional constant can be zero. But the integral constant should be chosen to be a positive real number. To rephrase, we need to maintain a history of previous values of total jitter to be able to compensate for current jitter in a stable fashion.

Algorithm 2 Compensated Sleep Implementation

Input: Expected Sleep period T. Maximum number of Iterations $Count$

1: Initialize $T1 \leftarrow CurrentTime()$,$Iterm \leftarrow 0$ {Integral Term}, $c(1) \leftarrow 0$,$k \leftarrow 1$,$tj(1) \leftarrow 0$
2: **repeat**
3: sleep($T + c(k)$)
4: $T2 \leftarrow CurrentTime()$
5: $d(k) \leftarrow T2 - T1 - T$
6: $tj(k+1) \leftarrow tj(k) + d(k)$
7: $Iterm \leftarrow Iterm + (tj(k+1) + tj(k))/2$
8: $c(k+1) \leftarrow -(min(T, K_i * Iterm + K * tj(k+1)))$
9: $k \leftarrow k + 1$
10: **until** $k = Count$

Algorithm 3 is the modified algorithm for implementing sleep with feedback controller. The maximum correction that can be applied is equal to the minimum time period T. There can be two approaches to using this algorithm:

A1 Pass a value of period Δ to the subroutine i.e. $T = \Delta$. Set $Count$ to the total number of periods. In this approach, a task scheduled to be executed will be released just after step 3 of the algorithm i.e. sleep step.

A2 Implement a period of Δ as a number of small periods $\delta < \Delta$. Pass δ to the algorithm. Set $Count = \Delta/\delta$. Task will be released after step 10 i.e. when the sleep algorithm returns in this approach.

Approach A2 is preferable when we want to use the control algorithm as just a single instance high resolution sleep timer. Approach A1 is preferable when there are a number of consecutive sleeps to be implemented. One can also implement approach A1 by using approach A2 in a loop and preserving the state variables $Iterm$ and tj between subsequent invocations of approach A2. We used approach A1 in experiments described in the results section.

In case of multiple schedulers, we can use independent controllers. Independence here implies the control loop independence. We will show later how we can control multiple periodic processes using different scheduler in the results section.

Steady State Error Analysis

For steady state error, we assume that the system is stable and then use the final value theorem. The final value theorem states that if the system error is given by $e(k)$, with $E(z) = \mathcal{Z}[e(k)]$, then the steady state value of the error, e_{ss}, is:

$$e_{ss} = \lim_{k\to\infty} e(k) = \lim_{z\to 1}[(1 - z^{-1})E(z)] \quad (5.11)$$

Since $E(z) = -TJ(z)$ (see figure 48), from equation 5.9 we can write $E(z) = -TF(z)D(z)$. Hence, from equation 5.11 we can conclude that the steady state error is given by

$$e_{ss} = -\lim_{z\to 1}[\frac{(1-z^{-1})D(z)}{1+[K_p+\frac{K_i}{(1-z^{-1})}]}], \text{ or}$$
$$e_{ss} = -\lim_{z\to 1}[\frac{(1-z^{-1})^2 D(z)}{(1+K_p)(1-z^{-1})+K_i}] \tag{5.12}$$

Note: Arguably, we can achieve jitter control by setting the next sleep duration as the difference between the next expected release time and the current time. However, this will be same as setting the sleep duration based on the current jitter i.e. the last expected release time and the current time. This scheme is equivalent to a proportional control. It is a well known fact that proportional control cannot achieve a non-zero steady state error. Therefore, if we do not use the integral controller we will never be able to achieve a steady-state of zero jitter.

For a unit-step disturbance, $d(kT) = 1$, we know $D(Z) = \mathcal{Z}[d(kT)] = \frac{1}{1-z^{-1}}$. Plugging this value into equation 5.12, we see that the steady state error for a unit-step disturbance will be 0 only if $K_i \neq 0$.

Modified Sensor Scheduler with Synchronization and Feedback Controller

Algorithm 3 shows the algorithm with the modification made for synchronization and feedback controller to sensor scheduler presented earlier. Note that the feedback controller is executed between two hyperperiods. In each hyperperiod, synchronization is achieved by sleeping till the next hyperperiod. The number of ticks between two hyperperiods is computed by dividing hyperperiod width by the length of a tick and taking its greatest lower bound. The variable tj stores the current value of the total jitter . $Iterm$ is used to calculate the integral component of the feedback compensation in accordance with equation 5.4. The maximum compensation that can be provided is

limited by the value of timeperiod (T) (see line 13 of the algorithm). This is done because sensor scheduler cannot sleep for negative times.

Algorithm 3 Modified sensor scheduler with synchronization and feedback controller to compensate for jitter

---

Input: S {Set of periodic sensors by name. Each sensor has two variables: P(Time period), C (current clock Value). T is the greatest common factor of sensor time periods.}
Input: $HYPER$ {the Hyperperiod value}
1: $S \leftarrow Sort(S)$
2: **loop**
3: Use nanosleep() to sleep till the next Hyperperiod signal
Pre Condition: $(\forall s \in S)(s.C = P)$
4: Initialize $TimeStamp1 \leftarrow CurrentTime()$ { CurrentTime() returns the current time in the system}

5: Initialize $Iterm \leftarrow 0$ {Integral Term}
6: $Count \leftarrow Floor(HYPER/T)$ {Floor provides the greatest lower bound integer.}
7: **for** $I = 0$ to $Count$ **do**
8: $TimeStamp2 \leftarrow CurrentTime()$
9: $d(k-1) \leftarrow TimeStamp2 - TimeStamp1 - T$\{$k-1$ because the disturbance is from the last step.\}
10: $tj(k) \leftarrow tj(k-1) + d(k-1)$
11: $Iterm \leftarrow Iterm + (tj(k-1) + tj(k))/2$
12: Set $T = minimum((\forall s \in S)(s.C)$
13: $c(k) \leftarrow -(min(T, K_i * Iterm + K * tj(k)))$
14: nanosleep($T + c(k)$) {nanosleep is a higher resolution sleep function available in Linux}
15: $(\forall s \in S)(s.C \leftarrow s.C - T)$
16: $(\forall s \in S)(s.C = 0 \implies Execute(s))$
17: $(\forall s \in S)(s.C = 0 \implies (s.C \leftarrow s.P))$
18: Run any aperiodic sensors if present.
19: **end for**
20: **end loop**

The controller and plant model developed here is not specific to this sensor scheduler framework. On the contrary, it can be applied to other periodic applications that control their own release time by maintaining their own clock. For periodic applications that do not control their own release, or are directly invoked by the kernel, we can incorporate a dummy scheduler with feedback controller between the kernel scheduler and the application.

Results

In this section, we will present results obtained by implementing the sensor scheduler with the feedback loop as discussed in algorithm 3. All these experiments were performed with the parameters $K = 1.0$ and $K_i = 1.2$ on Red hat Linux.

Performance of feedback controller in controlling jitter.

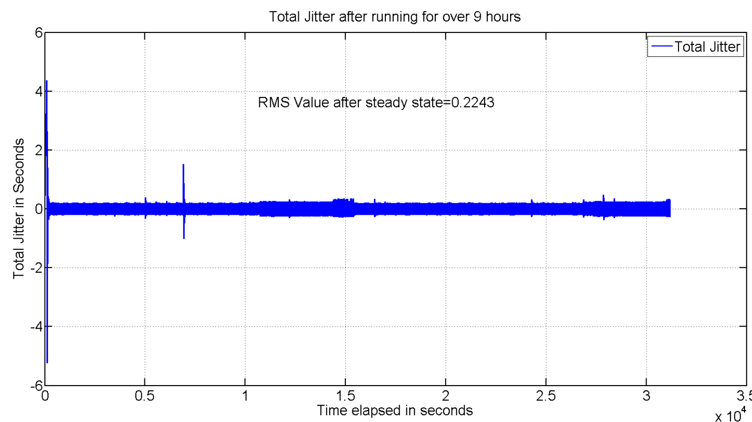
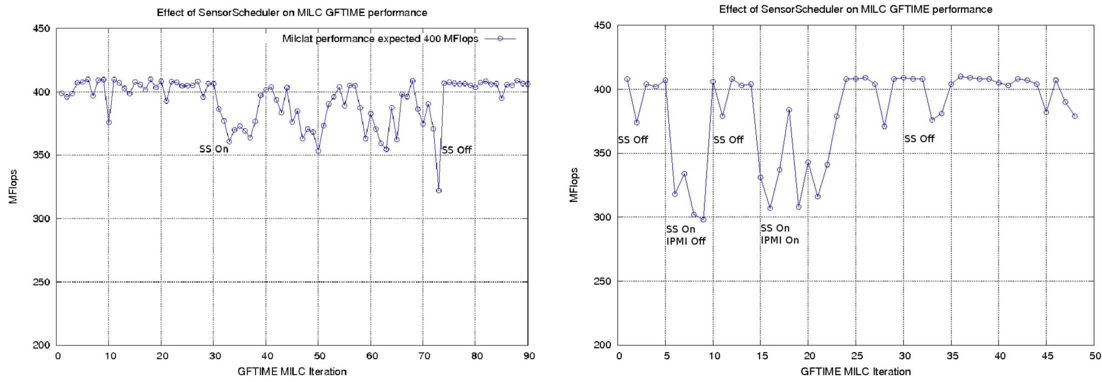


Figure 49: Total jitter accumulated in last 9 hours.

Figure 49 shows the total jitter accumulated by the sensor scheduler's main feedback loop running over a period of 9 hours. For this test, the sample time period was set as 5 seconds to make it more susceptible to disturbances. It was found that the root mean square (RMS) value of the jitter was 0.2243 after 9 hours, only 4.4% of the sampling time period, 5 seconds. This demonstrates the effectiveness of the feedback loop in controlling periodic jitter.

Jitter with and without controller with changing CPU Load

In order to emulate the step response of this controller, we decided to create a step disturbance by loading the CPU in steps to the 100% capacity manually. For this purpose, we used a number



(a) Performance with synchronized sensor scheduler

(b) Performance without synchronization. This is a restatement of Figure 46, reproduced here for ease in comparison.

Figure 50: Improved performance of a MILC job with synchronized sensor scheduler.

of prime number generators freely available over the internet to load the CPU. Furthermore, to emulate the step response, we initially started the framework without the controller, and then started the controller at a predefined time, in this case the 21st run of the sensor scheduler.

Figure 51 shows the result of this experiment. Notice that the total jitter rises initially when the controller is off. It settles down with a few oscillations when the controller is switched on. This figure also illustrates the contrast in jitter with and without the controller. The RMS value of total jitter in steady state was 0.209 seconds. This shows that the total jitter was bounded even when the system was heavily loaded.

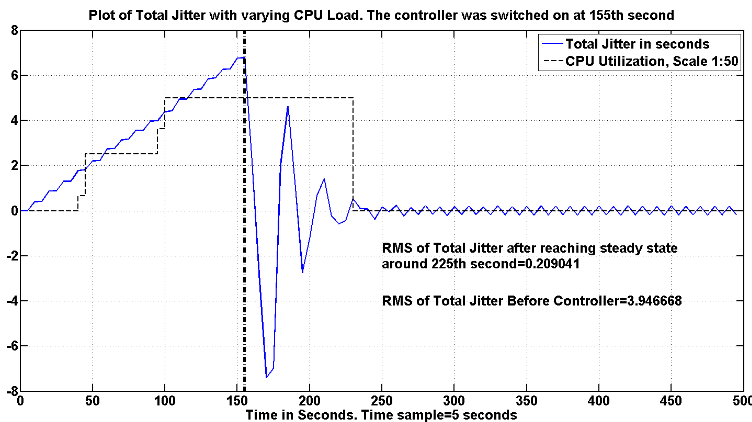


Figure 51: Plot of total Jitter with varying CPU Load. The controller was switched on at 155th second (dotted line).

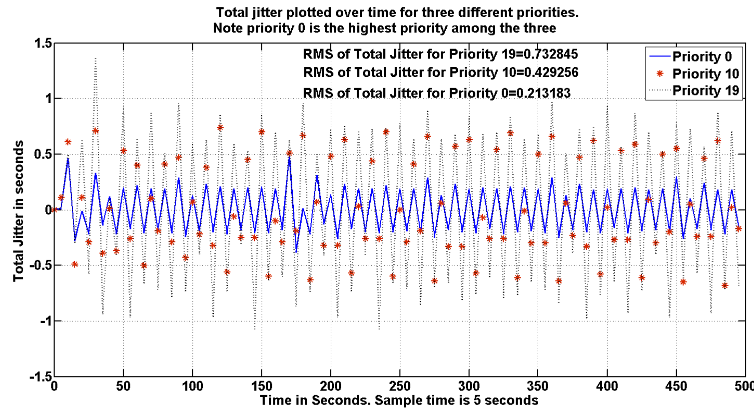


Figure 52: Total jitter plotted over time for three different priorities. Note priority 0 is the highest priority among the three.

Effect of Current Priority on Total Jitter

The performance of controller also depends on its operating system priority. All processes executing under Linux have a concept of niceness number. This number can be altered by nice and renice commands [175]. A niceness of -20 is the highest priority, while the priority of +19 is the lowest. By default, unless specifically altered, all processes start with a niceness of 0.

Figure 52 plots the total jitter accumulated when scheduler and controller were executed thrice, with three different priorities 0, 10, 19. As expected, even though the control loop stabilized in all three cases, the RMS value of total jitter increased as we lowered the priority. We can attribute this observation to the fact that a lower priority process will have to wait longer for the CPU time, which will increase the disturbance.

Controlling Jitter of Multiple Periodic Processes

In this experiment, we invoked eight different instances of the sensor scheduler to show that we can compensate the jitter of different applications with different controllers. Each scheduler had its own feedback controller built-in. All interactions between any pair of periodic applications can be attributed to their own random disturbance pattern. Thus, allowing the control loop design to work as it is.

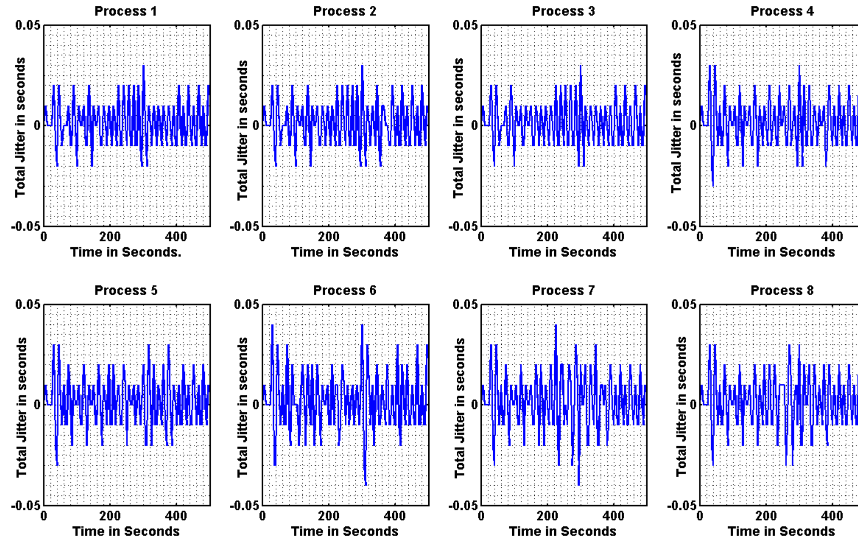


Figure 53: Plot of Total Jitter for 8 different periodic processes with the same time period of 5 seconds. Each process is being controlled by its own controller

Table 10: Mean, Variance and Root Mean Square values of Total Jitter in seconds for 8 processes referred in figure 53.

Process	Mean	Variance	RMS
1	0.0002	0.0001	0.0116
2	0.0002	0.0001	0.0118
3	0.0002	0.0001	0.0113
4	0.0002	0.0002	0.0127
5	0.0002	0.0002	0.0134
6	0.0003	0.0003	0.0159
7	0.0002	0.0002	0.0155
8	0.0002	0.0002	0.0143

Figure 53 shows the total jitter variation for each of the 8 processes. All processes have a period of 5 seconds. Table 10 provides the mean, variance and RMS value of all the processes. We can notice that all instances have similar mean and RMS value. The implication of this result is the conclusion that different controllers running together did not introduce any undue behavior in the control loop that would degrade the total jitter value.

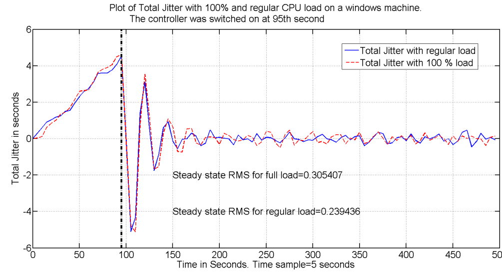


Figure 54: Plot of Total Jitter with 100% and regular CPU load on a windows machine. The controller was switched on at 95th second (dotted line).

Experiment4: Controlling Jitter of a Periodic Application in Windows XP

This experiment highlights the ability of the controller algorithm to work on an operating system other than Linux, in this case Windows XP. Since the sensors we had were specific to Linux, we created pseudo sensors that slept for a random period of time, uniformly distributed between $[0, 1]$ seconds. This helped us to emulate the part of disturbance due to time required for sensor execution. Figure 54 show the result under 100% CPU utilization and regular CPU load (around 10-20%). The 100% load was emulated by using a prime number generator. The system was initially started without the controller to check the transient response. The controller was switched on 95 seconds after the scheduler was started. It can be seen that the response is similar for both load conditions, however, the steady state RMS value of total jitter is more when the system was under a heavier load.

Performance of MILC Job- synchronization problem

This test was conducted on 128 nodes in the pion cluster. All instances of sensor schedulers were started from the head node in parallel . After the launch, sensor schedulers synchronized to the next plus one 5th minute of the hour. In this case, that time was ’Fri Oct 19 09:45:10 2007 (call it base time). Then onwards the sensor scheduler used the feedback controller for iterating every 10 seconds till the next hyperperiod. Sensors were run during an iteration based on their clock value. Lexical order was used to ensure that a deterministic order of sensor runs. Overall, 14 sensors ran on the nodes (see table 9). The test was stopped after 584 iterations with a sampling time of

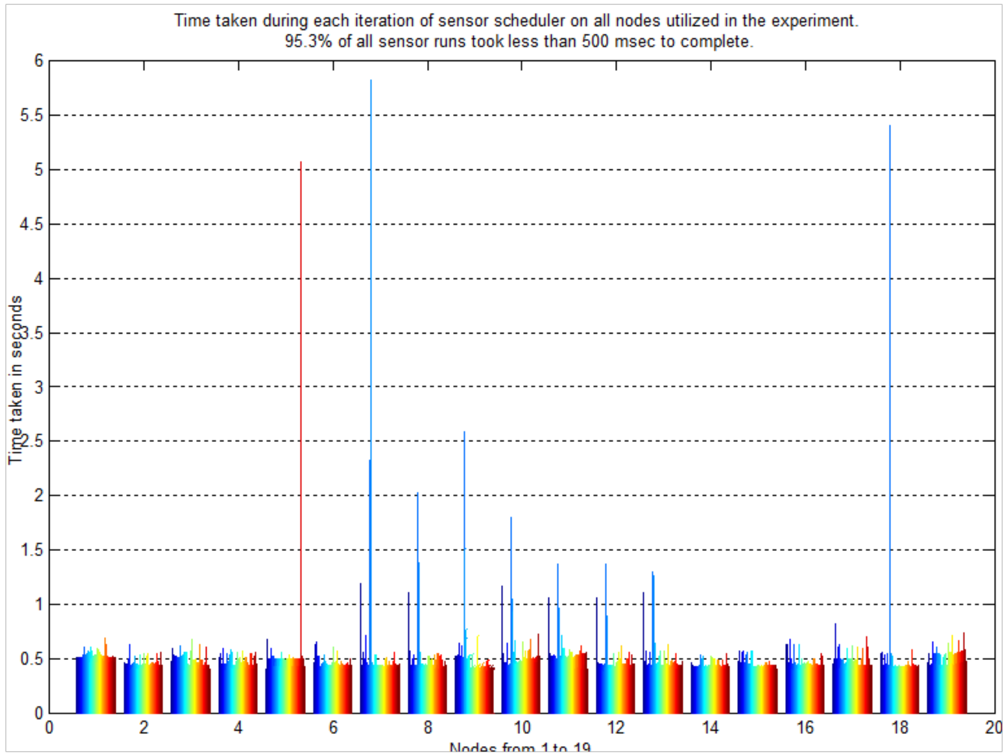


Figure 55: Time take during each iteration.

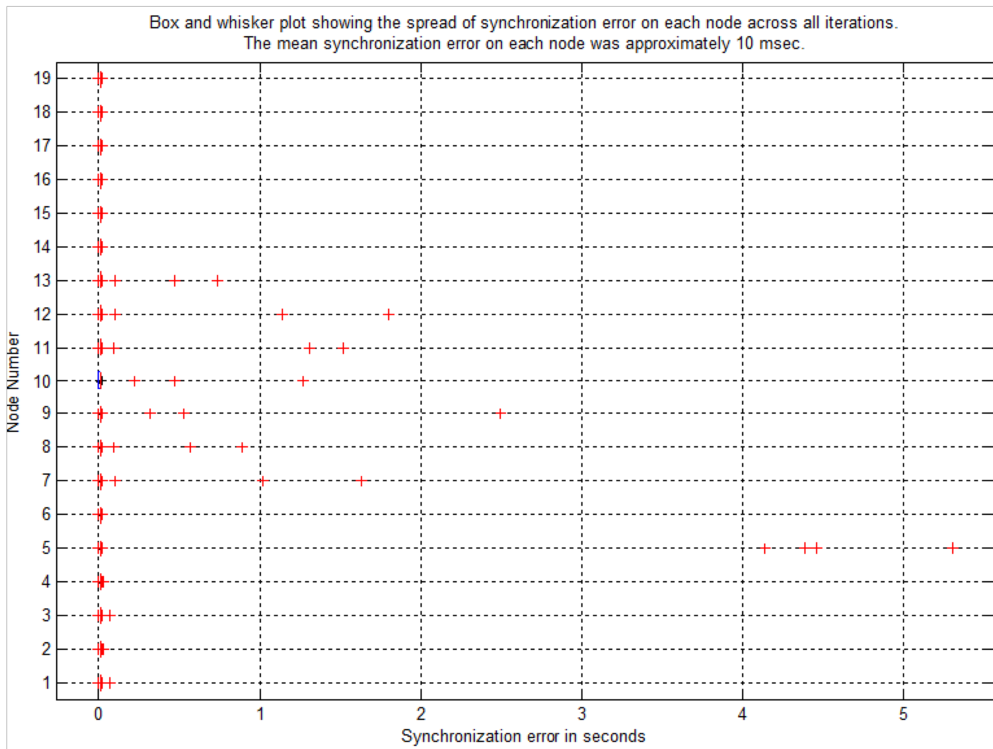


Figure 56: Box and Whisker plot showing the spread of synchronization error across 19 nodes across all iterations. The mean synchronization error on each node was approximately 10 msec.

10 seconds. On an average 12 sensors ran during all iterations. 13th sensor (monitoring changes in sensor codes) was executed in every 6th iteration, while 14th sensor (heartbeat) ran every 30th iteration.

Figure 56 shows a Box and Whisker plot showing the spread of synchronization error across 19 nodes across all iterations. The mean synchronization error on each node was approximately 10 msec. The synchronization error is the maximum deviation between sensor execution time on all nodes.

Figure 55 shows the time taken during all sensors. Notice that 95.3% of the sensor runs took less than 500 milliseconds for the measurement. Recall that the sensor run is the execution of a sensor program during an iteration of the sensor scheduler. The excess time taken during some sensor runs can be attributed to the non-deterministic scheduling provided by the operating system.

Figure 50 shows the performance drop in the MILC job when synchronized sensor schedulers were switched on in the cluster. Compare figure 50(a) to the earlier result without synchronized sensor scheduler in Figure 46,restated for comparison as figure 50(b). We can see that, on an average, there is a 50 MFLOPS gain in the performance compared to before. We can attribute the still existing performance drop of 50 MFLOPS to the 500 milliseconds taken on an average in a sensor run, which can still block the synchronization task. However, since the sensor schedulers are synchronized we do not see an accumulated delay in the MILC job as before. During this experiment, the average CPU utilization over a period of 24 hours on one of the nodes due to Sensor Scheduler was found to be 0.024 percent. The corresponding RAM utilization value was 0.318 percent.

Related Research

Generalized frameworks are being increasingly used to monitor the health and status of cluster resources. Early tools such as ClusterProbe [129] concentrate on per node monitoring and visualization without considering the health of the cluster as a whole.

Many monitoring tools have been developed at centers for supercomputing at the national laboratories. NetLogger [130] was developed at the Lawrence Berkeley National Laboratory and provides high performance event logging channels for capturing status messages across clusters into a centralized location. The Monitoring and Managing Multiple Clusters (M3C) [131] was developed at Oak Ridge National Laboratory, which provides a web-based GUI administration tool that allows a human administrator to monitor parameters across a federation of clusters on demand. Further developments of cluster management systems include the Java Agents for Monitoring and Management (JAMM) [132] system that was developed at Lawrence Berkeley National Laboratory to facilitate monitoring of more dynamic configurations of cluster environments using a publisher/subscriber methodology. These frameworks were tailored for centralized, human-in-the-loop management of clusters but little investigation was done to provide autonomic monitoring and control of cluster computing environments.

OVIS [133] uses a more complex statistical method to deduce models for a cluster's baseline status and provides a mechanism for automatic detection of early failures based on a node's conformity to those models. However, the computational costs of this approach are intensive, and can have a detrimental effect on the available cluster resources. Other tools such as the RVision [134] monitoring system have been investigating the effects of the monitoring framework on the performance of the cluster. Metrics for measuring how the RVision monitoring system interferes with the performance of hosted cluster applications is given by Ferreto in [134]. While the results were promising, no special consideration was given to mechanism for ensuring that the monitors do not interfere with the applications. Moreover, no consideration is given to the necessary failure mitigation portions of a truly autonomic monitoring and control framework.

In recent years, many modern frameworks such as Ganglia [135], Nagios [136] have been undergoing development. These frameworks are very well suited toward cluster monitoring and even simple control, and their open source licenses promote their use within the scientific community. While many of these frameworks are easily extended, they provide no mechanisms for bounding either the invocation time or execution time of components within the system beyond mechanisms

given by the operating system. Nevertheless, there is a need for modern monitoring frameworks that considers the effect of non-determinism of commercial operating systems and compensates for the same.

Feedback Control in Computing Systems

Feedback control based approaches are a common tool for solving a number of engineering problems. A recent roadmap provided by European Network of Excellence ARTIST2 on Embedded System Design focuses on achieving performance and adaptivity in real-time computing systems by using control theory, see [64] and references therein. The roadmap identifies six different research areas including CPU resource control and feedback-based scheduling. Our work in this paper relates closely to feedback scheduling. A state-of-the-art survey is given in [177].

One of the early results in the case for feedback control based scheduling algorithms was presented in [77] by Stankovic et al. In their approach, they used a PID controller to regulate deadline-miss ratio for a set of soft real-time tasks, by using CPU utilization as a control variable. They provided an extended version of their work in [78]. In that, they described their feedback control real-time scheduling framework for adaptive real-time systems and provided general guidelines for designing feedback loop for different quality of service (QoS) parameters. Implementation of this framework requires kernel level modification to an operating system. However, we require a user space solution to the problem to maintain flexibility of choosing any general purpose operating system.

In [80], Sha et al. presented a queuing model based feedback controller to keep the performance of a network server to desired levels. In the same spirit, the authors of [81] provided various models for a web server and designed feedback controller for QoS adaptation. Feedback control theory has also been applied in controlling CPU utilization in real-time systems [82]. Steere et al. introduced a new scheduling scheme based on periodicity of tasks in [79]. Their scheme was to allocate each thread a percentage of CPU cycles over a period, and then use a feedback loop to control both proportion and period.

These approaches depend on the ability to implement an algorithm at the kernel level of the operating system. The approach presented in this paper uses a generic user space approach that can be implemented on different operating systems readily.

Conclusion and Future works

In this paper, we presented a reliability framework for clusters called SCARF that implements this feedback controller based synchronization technique. Experiments presented in the previous sections show that the reduction of total jitter is possible with the use of the feedback controller presented in this paper. Moreover, the use of feedback controller only imposed a trivial amount of extra CPU load. This along with the hyperperiod synchronization allows all sensor runs across the cluster to execute simultaneously. Consequently, the performance drop in parallel jobs with the sensor framework present was reduced.

Future investigations will include the semantics of the modeling language to help deploy the analysis routines along with the required monitoring framework. We will also investigate the automatic synthesis of workflows to be deployed on the cluster from a given analysis specification. Our goal is to achieve a holistic framework that enables deployment of workflows along the cluster and manages the health of the cluster to maximize its efficiency.

Acknowledgments

This work is supported by the National Science Foundation under the ITR grant ACI-0121658. The authors will also like to acknowledge their colleagues at Fermi National Accelerator Laboratory.

CHAPTER VI

ON THE DESIGN OF A FRAMEWORK TO ENABLE RELIABLE SCIENTIFIC WORKFLOWS

Overview

The third contribution of this research is the integration of reliability framework with the workflow management system. Despite the similarities with business workflows, scientific workflows have fundamental differences that impose unique challenges for a management system. Some of these challenges are:

- Scientific workflows contain large number of activity instances that are dataflow-oriented. This means that a data provenance framework for scientific workflows is a key component of the management system. Data Provenance is a record of the history of the creation of any data object related to the workflow.

- The dataflow structure is only partially known before execution. It often changes at runtime depending upon the output results of a previous step or the input parameters.

- Participants or jobs belonging to a scientific workflow are computationally intensive with long and sometimes unpredictable execution times. A long execution time increases the probability of failure due to an external factor such as node shutdown.

- Typically, data items involved in scientific workflows range in few GB in size. For some experiments the data size is even larger. For example, it is estimated the data generated due to Large Hadron Collider at CERN will range in petabytes [160].

This paper describes the design of a scientific workflow execution framework that integrates run-time verification to monitor the execution of each participant and check it against a set of formal specifications. 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 invariant conditions with actions to be taken upon violation of invariants. Lattice Quantum Chromodynamics (LQCD) workflows have been used as motivating examples in this paper. A Provenance Model and a prototype implementation are also presented in this paper. Finally, a few failure scenarios and overhead of workflow management system are explored.

This paper has been submitted to the Journal of Cluster Computing [17]. It is an extension of the work presented at fourth IEEE International Conference on e-Science, 2008 [18] and sixth IEEE International Conference and Workshop on the Engineering of Autonomic Systems (EASe 2009) [19].

Abstract

This paper describes the design of a scientific workflow execution framework that integrates run-time verification to monitor its execution and check it against a set of formal specifications. As a motivating example, we describe Lattice Quantum Chromodynamics (LQCD) workflows that produce large amounts of data resulting from numerical simulations of QCD quantum field theory. Configuration files generated through Monte Carlo simulations are later used for computing certain physics quantities such as decay rates and particle masses. Existing Workflow systems can be used to help standardize workflow descriptions, but they lack tracking of domain specific information.

Our framework provides 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. Periodic as well as on-demand monitoring of vital health parameters based on pre-specified invariant conditions is enabled on allocated nodes as participants belonging to a workflow are mapped onto machines and executed. Mitigation is specified as actions that must be taken upon violation of invariants.

Introduction and Motivation

Current computing power and storage capabilities allied to distributed computing models allow researchers to computationally solve problems of 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 amounts of data that are processed at several institutions. Applications, software libraries, input data and workflow[2] recipes are shared among collaborators worldwide[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 allows 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 [178] across dozens to hundreds of processing nodes. Typically, several users analyze different workflows on the clusters concurrently. Given the scale of numbers, identifying job failures, fault isolation and fault mitigation becomes critical, specifically when the success of whole workflow might be affected by even one job failure.

[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/

Table 11: Cluster Evaluation Metrics.

Let T1 = Start time, T2 = End Time, and ΔT = Sampling period.

Let N = $\frac{T2 - T1}{\Delta T}$.

Let $OnlineNode_i$ be the number of nodes online in sample i.

Let $TotalNodesInCluster_i$ be the total number of nodes in the cluster in sample i.

Let $BusyNodes_i$ be the number of nodes in the cluster in sample i that run a job.

Let $SuccessfulJobs_i$ be the total number of jobs that completed successfully in sample i.

Let $TotalJobs_i$ be the total number of jobs scheduled in sample i.

$$Availability_{T1}^{T2} = \frac{1}{N} \frac{\sum_{i=1}^{N} OnlineNode_i \Delta T}{\sum_{i=1}^{N} TotalNodesInCluster_i \Delta T} \quad (6.1)$$

$$Utilization_{T1}^{T2} = \frac{1}{N} \frac{\sum_{i=1}^{N} BusyNodes_i \Delta T}{\sum_{i=1}^{N} OnlineNode_i \Delta T} \quad (6.2)$$

$$Productivity_{T1}^{T2} = \frac{1}{N} \frac{\sum_{i=1}^{N} SuccessfulJobs_i \Delta T}{\sum_{i=1}^{N} TotalJobs_i \Delta T} \quad (6.3)$$

$$Effectiveness_{T1}^{T2} = Availability_{T1}^{T2} * Utilization_{T1}^{T2} * Productivity_{T1}^{T2} \quad (6.4)$$

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 11). The primary accounting measurement is node-hours (obtained by multiplying number of nodes and the sampling time). 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 hours available on the nodes that are online vs. 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 jobs i.e. ones that did not fails. Here, we are discounting cases where a job might succeed but due to various algorithmic reasons might lead to unproductive data. Finally, effectiveness measures the overall metric for our clusters.

In typical e-science workflows, the Grid is used as the main source for job processing, and close remote application monitoring is limited [179]. 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. Such a cluster management system 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: We need to 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 need a framework of sensors that provide the basic monitoring information [15].

- 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. However, this requires us to have a data provenance framework

Preliminaries

This section discusses the definitions and concepts that will be used subsequently.

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 i.e. $T \subseteq \mathbb{Q}$. This is required to be able to analyze time dependent tasks and algorithms as variants of timed automata and do reachability analysis on the equivalent region graph automaton as described in [157]. This global clock is synchronized with the local clocks of computing nodes by a jitter control algorithm and NTP as explained in chapter V.

A timed automaton is a classical model used for abstracting time-based behaviors of systems, which are influenced by time [157, 158]. 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 enabled 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. 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 [157].

Resources

We define the workflow management problem, including scheduling and monitoring over a set of "resources". These resources are composed of (a) communication attributes, and (b) computational attributes. Here, we will discuss the computation resources and state that communication resources can be defined in a similar manner.

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, these resources are defined as a universal set $R = (A_C)$, referred to as resource domain, consisting of a set A_C of computational 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) $\in A_C$.

Computation Nodes N

Scientific workflows are executed and managed over a set of computation nodes that provide the resources 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.
- $E \subseteq N \times N$ is a set of communication links connecting computation machines.
- $\phi : N \times R \to \Re$ is a resource capacity level map, where $\phi(n, r)$ is the level of resource r in the node n.

Participants p

Participants are classified into categories called participant types, $p \in P_T$. A participant type (P_T) groups together 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. Different versions of the same algorithm are used for recovery, if so identified by the user.

A participant is a computation task written as $p \in P_T$ i.e. it belongs to the type P_T. $\{p | p \in P_T\}$ denotes the set of all participants that belong to type P_T. Formally, a single participant can be defined as a computation task $p : 2^I \to 2^O$ (power of 2 represents a power set). It takes in a number of parameters from the set of input parameters I, and generates a number of output products from the set O.

A map, $\psi : p \times A_c \rightarrow \Re$ defines the minimum resources required to execute the participant on one node. For example, $\psi(HeavyLight, \text{`RAM'}) = 512$ implies that the algorithm implemented by participant $HeavyLight$ cannot run if the available RAM is less than 512 MB.

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.

Workflows W

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 [179]. All participants belonging to a workflow are related to each other by control and data dependencies.

Workflow is defined as tuple W=$(\mathbb{P}, I, O, D)$, where $\mathbb{P}$, is the set of participants, I is the set of all user inputs, O is the set of output products and $D \subseteq \mathbb{P} \times \mathbb{P}$ defines a set of directed dependencies between the participants. A workflow is called directed acyclic and executable if the graph induced by the set D is a directed acyclic (no cycles or self-loops). Notice that we allow the definition of cyclic workflows. However, we require existence of a translator that can generate directed acyclic and executable workflows from a cyclic workflow definition. There are two types of workflows, abstract and concrete.

Definition 6 (Abstract Workflow) *An abstract workflow is a specification that defines a directed relationship between several participants. They also describe the parameter type and the number of parameters (tokens) that are consumed by each participant. A pictorial example of a workflow is shown in figure 57, where participants $p \in \mathbb{P}$ are represented by circle of different shades. Arrows define the data dependencies D. The parameter set is used to generate a concrete workflow from an abstract workflow. This example is explained later in section on generation of concrete workflows.*

Definition 7 (Concrete Workflow) *Concrete workflows are directed acyclic workflows that can be executed by the workflow manager. They are generated from a given abstract workflow and the specified set of parameters. These workflows are directed acyclic graphs.*

We will discuss the generation of concrete workflow in a later section.

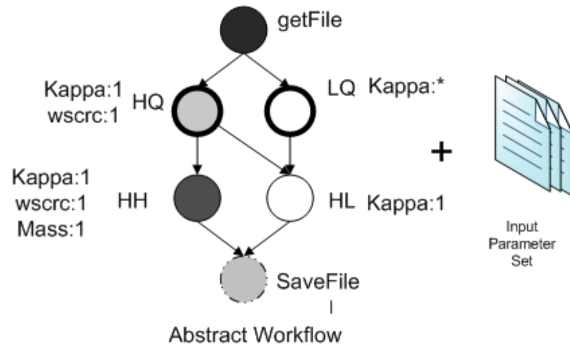


Figure 57: Sample workflow definition

Workflow execution engine

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 (Figure 57) described in a workflow language format. Currently, there is no standard for scientific workflow languages. Several groups have developed their own version of workflow languages, such as SwiftScript [180] and AGWL [181], that are specific to their execution engine. Ideally, the execution engine W_e is responsible for tracking the run time dependencies D and enabling participants as dependencies are met and input data are available. However, this is not true in all workflow engines that are currently available.

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 $p_T \in \mathbb{P}$ in the workflow and the set of

computation nodes N, the task allocation map $\theta : \mathbb{P} \rightarrow 2^N$ 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 [22, 182] with Maui Scheduler [183] for imposing an approximate First in First Out (FIFO) scheduling policy with some extra features such as reservations for queued jobs, the throttling rules, and the flexible priority system. See Maui documentation [183] for further details on available scheduling policies.

A valid task allocation map θ has to satisfy the constraint on resources. Specifically, nodes allocated to a participant P_t must have the minimum amount of resources required (see equation 6.5). 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.

$$(\forall r \in R)(\forall n \in \theta(P_t))(\phi(n, r) \geq \psi(P_t, r)) \tag{6.5}$$

Sensors and Filters

Table 12: Sensors present in our framework

No.	Sensor	Period(sec)
1	CPU Fanspeed	10
2	Motherboard Fanspeed	10
3	CPU Temperature	10
4	Motherboard Temperature	10
5	Aggregate CPU Utilization	10
6	CPU Utilization per process	10
7	Hard Disk Utilization	10
8	RAM and Swap Utilization	10
9	Aggregate RAM Utilization	10
10	Aggregate Swap Utilization	10
11	CPU Voltage	10
12	Motherboard Voltage	10
13	Monitor configuration changes	60
14	Heartbeat	300

A **Sensor** program can be considered as function $S_{n,r} : \mathcal{T} \rightarrow \Re$, where $n \in N$, is a computation node and $r \in R$ is a computational resource. Sensors are used to monitor the current utilization of computational resources associated with a node, where current is defined with respect to the global time $\mathcal{T}$. Every sensor is associated with a **Type** of measurement. It is a unique combination of the computation node identifier (node names are unique) and resource identifier (resource names are unique). Type of sensor $S_{n,r}$ is $n.r$. A configuration database contains the list of types and a script that can be called to provide the measurement. See Table 12 for a list of sensors available in our framework. 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 [15].

Filters evaluate the measurement produced by a sensor against specified criteria to generate an event. Filters are used to down sample, if required, and translate the information generated by a sensor. This is done to reduce the network traffic generated by a sensor by ensuring that a filter only forwards a new sensor measurement if it is over or below a deadband as compared to the last measurement value.

An **Event** is a tuple, $E = (\text{Type}, \text{Timestamp}, \text{Value})$. This event is subsequently published over the network. Timestamp is the time of measurement and Value is the actual measure. Events on a global time scale can be correlated to get significant statistical insight into general cluster health. For example, a sustained rate of increase in CPU temperature from a region in cluster signifies a cooling system failure. We currently employ and use the temperature based correlation.

The most commonly used sensor is the **heartbeat** described in [15]. It acts as a watchdog and informs whether a computing node is online or not. It is denoted as $S(n, H) : \mathcal{T} \rightarrow \{0, 1\}$. Here H is the heartbeat resource of node n. H is either present or absent.

A heartbeat sensor is in fact a combination of a periodic sensor on the concerned computing node and a filter on a monitoring node. Together, this combination generates events of type $n.H$ with two possible values $\{0, 1\}$, where 0 means the node is offline and 1 means the node is online. Since Heartbeat is one of the most commonly used sensor, we will use a special notation $\mathcal{H}(N)$ in

the text. Liveness condition for a node called 'pion1' implies that $\mathcal{H}(\text{pion1})(t) = 1$, where t is the current time. Figure 58 shows the timed automaton used to periodically generate heartbeats from all computing nodes. The timeout event is generated by a timer, which generates a timeout event after a pre-specified time. Semantics of this timer have been explained in chapter IV.

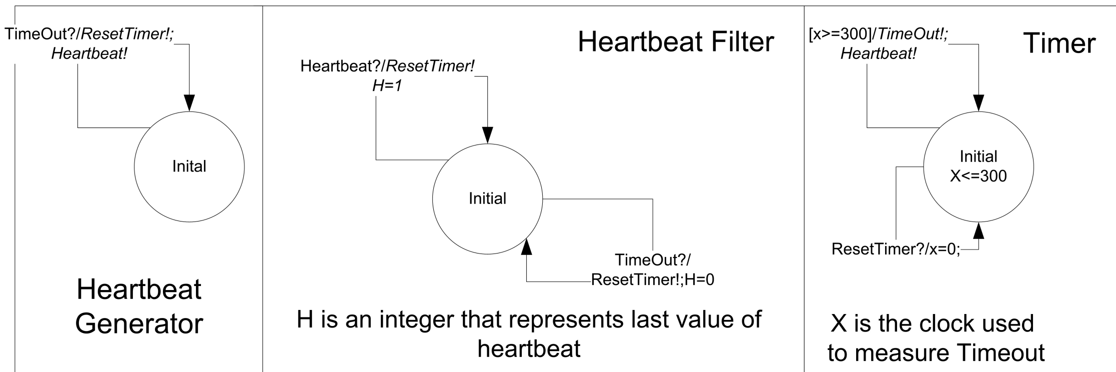


Figure 58: Combination of a hearbeat generator and heartbeat filter. The timeout event is generated by a timer, which generates the event after a pre-specified time. H is an integer that can be either 0 or 1. This figure uses the UPPAAL [139] notation. '!' is a generated event. '?' is a received event. Semantics of this timer have been explained in chapter IV

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 time points 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 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 $[t_{min}, t_{max}] \subseteq \mathcal{T}$, $a_i.value$ 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 $Query : Type \times \mathcal{T} \rightarrow \Re$ that provides the measurement value at that time. Here $Type$ is a universal set of all types of events. $\mathcal{T} \subseteq \mathbb{Q}$ is the global time value. This operator can be used from an observing computing node to evaluate condition over resource variables on another node using predicates $\leq, <, >, \geq$. For example, querying the disk sensor referred earlier will be written as $Query(pion1./project, t)$, where t is a point in time. If t is omitted, e.g. $Query(pion1./project)$, it implies that the query is referring to the current value of $/project$ utilization on a node called $pion1$, which due to the zero order hold semantics implies the last reported value.

A component can either poll the value of a sensor using the query operator (pull semantics) or rely on the middleware (we use syslog-ng [184]) to deliver the last value of sensor as an event (push semantics). We currently support both semantics. While pull is supported using a database that stores all the sensor value, push values are directly sent to the subscriber's syslog-ng server.

Runtime Checkable Properties

The underlying foundation of specifying safe operating 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.

Properties 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 These properties are instantaneous in nature. They are not evaluated over a history of states. These formulas, can be used to express either an entry condition or invariants of states. However, these properties are static in the sense that they do not express the dynamic evolution of states as software execute. They are 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 $p = Query\ op\ c|c \in \Re$, $op ::= < | > | \leq | \geq$. If time is omitted from the query specification it indicates the last reported event from that sensor. For example, property $Query(n./project) < 10000$ means that utilized space on the disk partition $/project$ should be less than 10000MB, as reported by the last event.

Timed First order and propositional logic (untimed) are capable of expressing properties of states. Temporal logic, extend these static logic by allowing expression of relations over a sequence of states. This is required in concurrent systems where we are not only interested in initial and final states but are also interested in all the possible interleaving sequence of concurrent states between the initial and final states. In temporal logics, time is not explicitly mentioned; but a formula specifies if eventually some specific state might be reached or a certain set of bad states will never be entered. Therefore, these logics are important to express safety, livelock and deadlock properties. However, sometimes correctness of system not only depends upon the correctness of computation but also on its timeliness. An effective way to specify formal properties for real-time systems is to use time bounds with the CTL temporal operators. Timed properties are defined using timed computation tree logic (TCTL) [169].

We are interested in the following timed properties for workflow systems:

- *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. For example we will use

$A\Box(Query(pion1./project) < 10000)$ to state that $/project$ utilization on pion1 is always less than $10000MB$.

- *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.

Distributed Monitoring and Mitigation Framework

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

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 (see chapter V). We had also described a scheduling algorithm used to execute the sensors on all computing nodes in that chapter.

All computation nodes also contain two real-time reflex engine modules along with the sensors. 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 [11] for a detailed discussion.

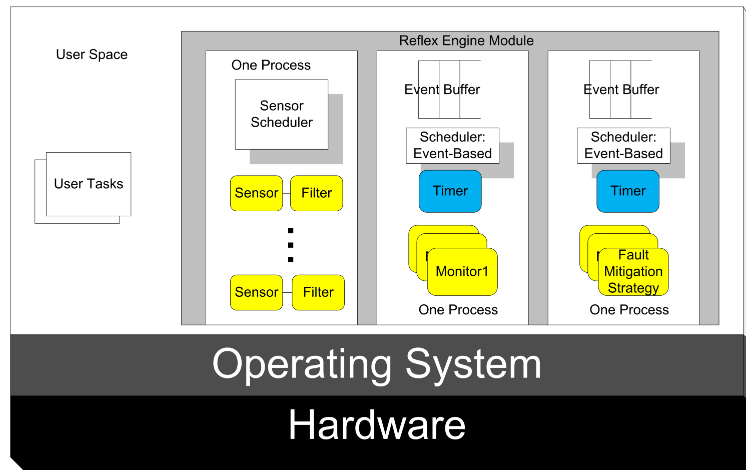


Figure 59: A computing node in the real time reflex and healing framework.

Figure 59 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 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,(i) general purpose i.e. they are always on, and (ii) 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 [11].

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 (This is done manually and not automatically). 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 same as testing for acceptance of a given interval trace by the property timed automaton. The module in the framework that performs this check is called a **Reflex Engine**[11]. We call the class of reflex engines that are used to identify the faults, which are induced due to violation of properties **Monitors**.

As an example, we will describe the heartbeat property for dcache nodes. dcache is a powerful distributed storage resource manager [185]. 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 $A\Box\mathcal{H}(pm)(t) > 0$, $t \in \mathcal{T}$ is the time at which the property is evaluated. Figure 60 shows the heartbeat generator, filter and a heartbeat monitor. This monitor receives the event from filter in a timed sequence i.e. an event generated in past cannot be received after an event in future (see chapter IV for discussion on buffers used in reflex engines).

Notice that we are using push semantics here. The monitor starts from the normal state. It transitions to "no heartbeat" state if a heartbeat event containing $pm.H = 0$ (in the figure pm is omitted for brevity) is received. To guard from overzealous mitigations, the engine uses a "ping engine" to check if a ping can be successfully sent to the pool node. Node is marked down and the diagnoser is called if the ping fails. Diagnoser is another reflex engine that goes through a series of steps upon receiving a node down event. This diagnoser is general in the sense that it is used

for the computation nodes as well. The decision block "PBS job running" checks if a job has been scheduled to run on the concerned node by PBS. This is done by querying in the job database. This decision block returns false for a dcache node as it does not run any computational job. It attempts to revive the node via Intelligent Platform Management Interface (IPMI) reset [186]. Upon failure the node is marked offline and an email is sent to the administrator.

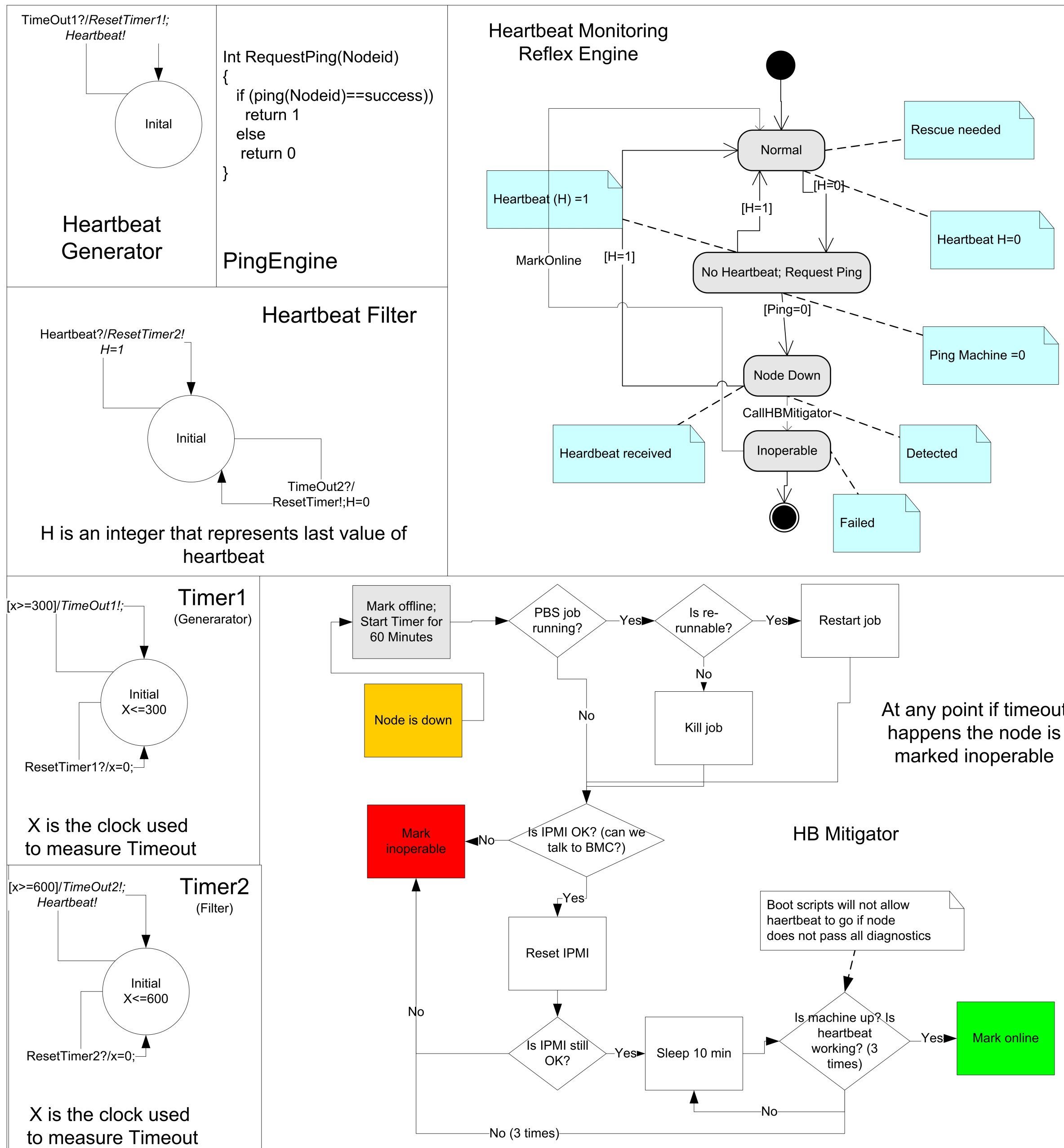


Figure 60: A reflex engine monitoring the heartbeats for a pool node. The node diagnoser is a reflex engine that attempts to revive the node and marks it offline upon failure.

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 61). 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 [46] 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 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.

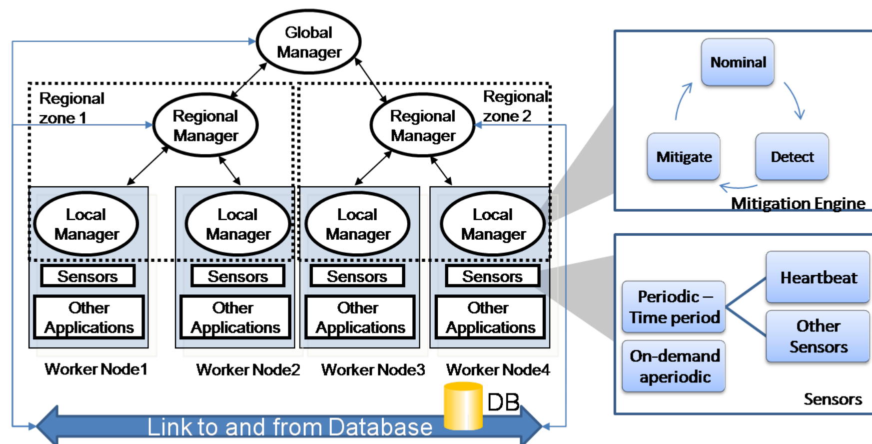


Figure 61: Hierarchical reflex engines

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. This integration will be discussed later in this paper. We have written scripts that enable us to obtain a snapshot of cluster status and report it to external world via http.

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 (also referred as abstract workflow). Typically, the parameterized workflows are of two classes: (i) creation of configuration gauge ensembles, and (ii) processing of ensembles in analysis campaigns. The parameterized workflow in conjunction with a set of input parameters yields a concrete workflow. Examples of input parameters are values for particle masses.

Management of input parameters is necessary to provide execution tracking and recovery. To do this 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. The Provenance model is described in next section.

Provenance Model

Data Provenance is a record of the history of the creation of any data object related to the workflow. It is one of the most important requirements of a reliable workflow management system. It is necessary for tracing files origins, including the workflow execution (instance) and which job (participant) generated the file along with physics parameters. Most workflow systems lack detailed provenance data and execution tracking. These features are critical for LQCD and similar large scale scientific workflows. Tracking of outputs in forms other than files (e.g. error messages, parameters and checksums) is needed for provenance to be complete.

Considering that generic provenance modeling is not trivial to adapt for specific problems, we developed our own model driven for LQCD workflow requirements. We are confident that this model is still generic and can be applied for other scientific applications as well. a specific problem, the model still allows modifications to support other scientific applications.

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.

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.

Generation of Concrete Workflows

To promote reusability of workflows, we support specification of abstract workflows that can be used to generate different concrete workflows based on the given parameter. The execution of a workflow causes the creation of a WorkflowInstance, which has references to ParticipantInstances.

An abstract workflow defines the relationship between participants and the input parameters they consume. A parameter set contains arrays of physics parameters. Generic Modeling Environment [187] was used to create a modeling language for specifying the abstract workflows [12]. For example, kappa$==[1\cdot\cdot4]$, wsrc$=[1\cdot\cdot3]$, mass$=[1\cdot\cdot2]$,and d1$=[1]$ is one parameter set. The abstract workflow also identifies the number of parameters of a type consumed by a participant.

Consider figure 62 for example. Participant HQ consumes 1 Kappa and 1 wsrc, while participant LQ consumes all Kappa. Conditions in the abstract workflow are specified using templates that are replaced in the concrete workflows. For example, heartbeat property must be specified in the data model without knowing the exact nodes used to execute the participant. We use special function $ALL(A\Box\mathcal{H}(\theta(P_t))(t))>0$ to symbolize the conjunction of heartbeat conditions for all nodes that are included in the task allocation map. For example, if the participant is allocated to run on 'pion1' and 'pion2', this condition will expand as $(A\Box\mathcal{H}(`pion2')>0)\wedge(A\Box\mathcal{H}(`pion1')>0)$.

This ALL, and P_t is automatically replaced by the node assigned to the participant by execution engine.

This information allows us to generate a concrete workflow for the given parameter set kappa= $[1 \cdot \cdot 4]$, wsrc= $[1 \cdot \cdot 3]$, mass= $[1 \cdot \cdot 2]$,and d1= $[1]$ as shown in figure 62. In this 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.

This generated graph also contains pre, post and invariant conditions as specified in the data model discussed in previous section. The pre and post conditions are executed as pre and post scripts for all participants as shown in figure 64. 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 [188].

Figure 63 shows the concrete workflow generated by using a different set of parameters: kappa== $[1]$, wsrc= $[1 \cdot \cdot 3]$, mass= $[1 \cdot \cdot 2]$,and d1= $[1]$. Notice that the number of participants (and parallel branches) in the second case are less than the first case where the parameter set was larger.

Implementation of Workflow Engine and integrating it with reflex engines

This section describes the execution engine that we are using and its integration with the reflex engines.

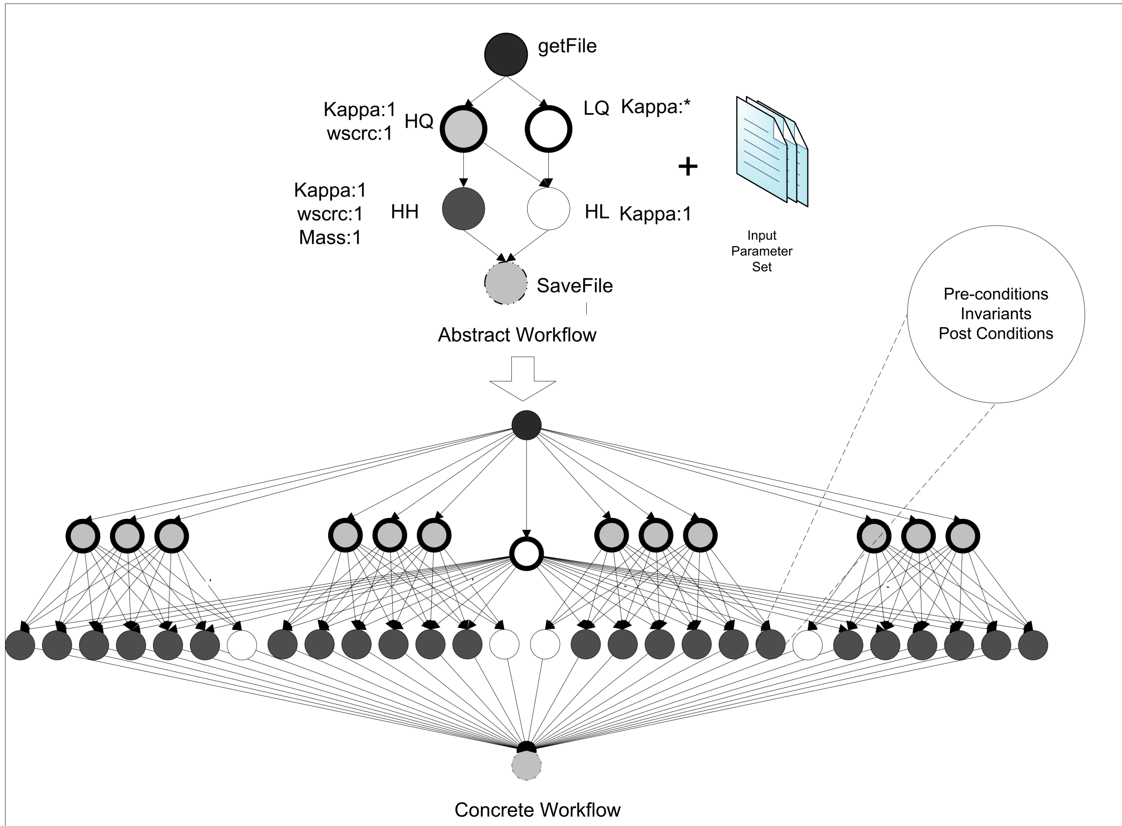


Figure 62: Transformation from parameterized to concrete workflow. kappa== $[1 \cdot \cdot 4]$, wsrc= $[1 \cdot \cdot 3]$, mass= $[1 \cdot \cdot 2]$,and d1= $[1]$.

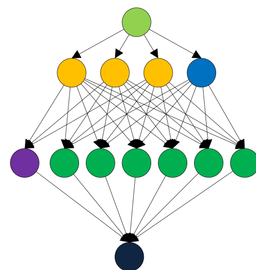


Figure 63: A different concrete workflow derived from the abstract workflow of Figure 62 but different parameters. kappa== $[1]$, wsrc= $[1 \cdot \cdot 3]$, mass= $[1 \cdot \cdot 2]$,and d1= $[1]$.

Execution engine

An execution engine is responsible for traversing the concrete workflow graph and submitting the available participants to a scheduling engine such as PBS. As a participant is picked from the ready queue, the engine checks the pre-condition, then submits the job and sends the invariant conditions to the reflex engines. The reflex engines take corrective actions as specified by the mitigation policies and also notify the execution engines when failures occur.

The workflow execution engine provides an interface for submitting concrete workflows. Multiple concrete workflows can be handled by multiple execution engine threads. However, all execution engine threads share the same ready queue. The advantage of using multiple execution engine threads is that it increases the service rate and reduces the average waiting time for a ready participant.

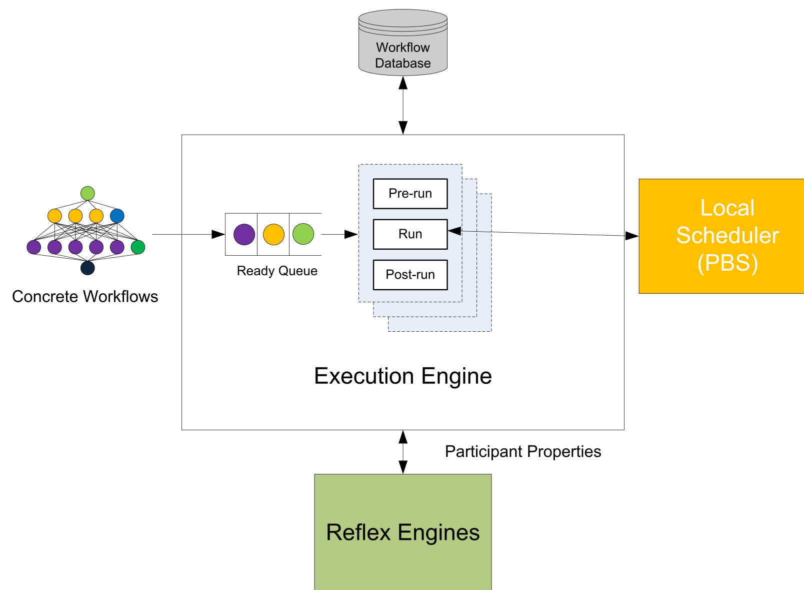


Figure 64: Execution engines maintains a ready queue.

Concrete workflows are 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 [189]. An execution engine such as DAGMan [190] or openWFEru BPM engine [191] 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.

Integration with Reflex Engines

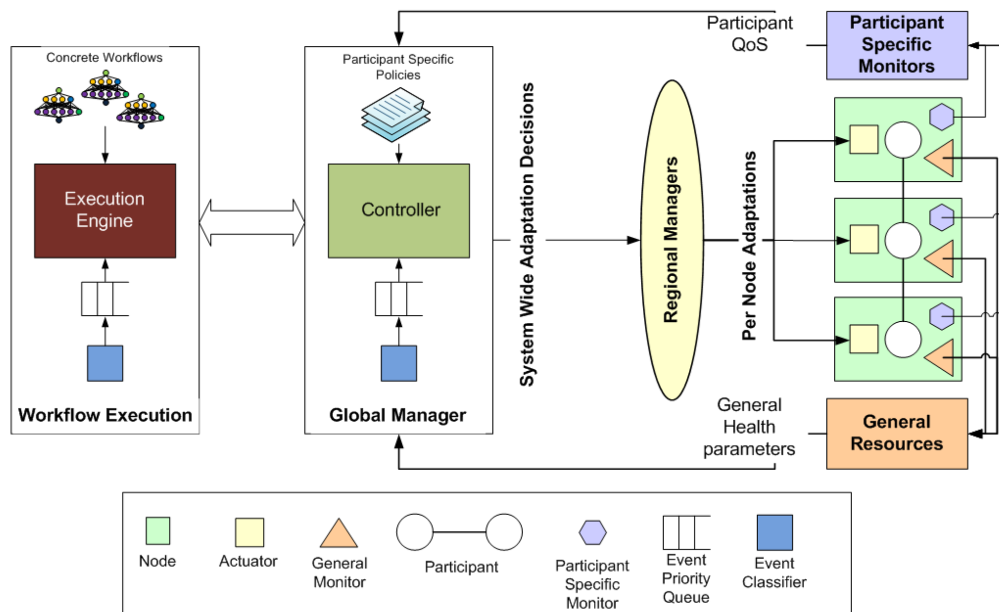


Figure 65: Complete runtime framework with integrated workflow, monitoring and mitigation integration

The run time framework, composed by the integration of the workflow, monitoring and mitigation frameworks is depicted in Figure 65. 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.

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 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 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 case study section.

Implementation of a Prototype

A prototype of this architecture was implemented with python and Ruby programming language and the Ruby on Rails (RoR) framework [192] with MySQL database as the backend.

After a workflow and parameters are selected, a workflow execution engine is launched. The openWFEru BPM engine [191] is used for the prototype. It is a business oriented workflow system

with good support for workflow patterns [193] implemented in Ruby, which integrates very well with the RoR framework.

As participants are invoked by the engine new ParticipantInstances are created in the database. Every participant optionally has a pre and post run script that are invoked before and after running the actual executable binary code of the participant. The pre-run scripts are used for translating parameters from the database parameter space into command line arguments suitable for the participant binary. After finishing the participant execution the post-run script saves provenance records. The prototype currently submits jobs to PBS/Maui running on clusters at Fermi lab.

Results and Case study

In this case study we will consider the example of a 2-pt 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 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.

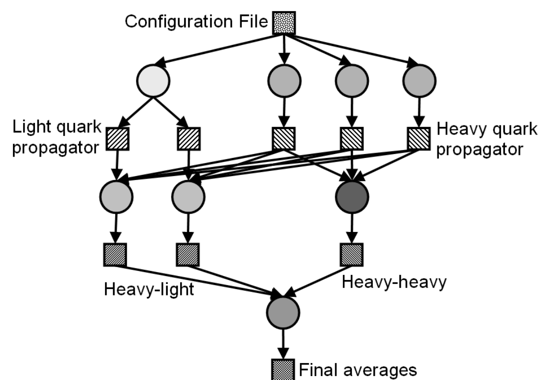


Figure 66: Two-point analysis workflow for one configuration file

The sample workflow shown in Figure 62 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 values of physics parameters (kappa and wsrc).

For this case study, we only used a single computation node, even though the participants are MPI jobs capable of running on multiple machines. Jobs for this study were submitted using a simulated PBS queue that transferred the job to the concerned computation node and started it. The other node was used to run reflex engines and the execution engines. The communication between reflex engines and 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. The getFile participant used secure copy command for obtaining a file from the simulated dcache node.

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

Considering the failure of a dache node: 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.

Specifying the available diskspace precondition: 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: $Query(n.'\text{/project}') < 10000$.

Considering the failure of a computation node: The allocated computation node must be online i.e. $A\Box\mathcal{H}(cn)(t) > 0$, where $cn \in \theta P_t$ is the node allocated to a participant, P_t.

The time stamps captured here are relative to the global manager that runs the workflow execution engine.

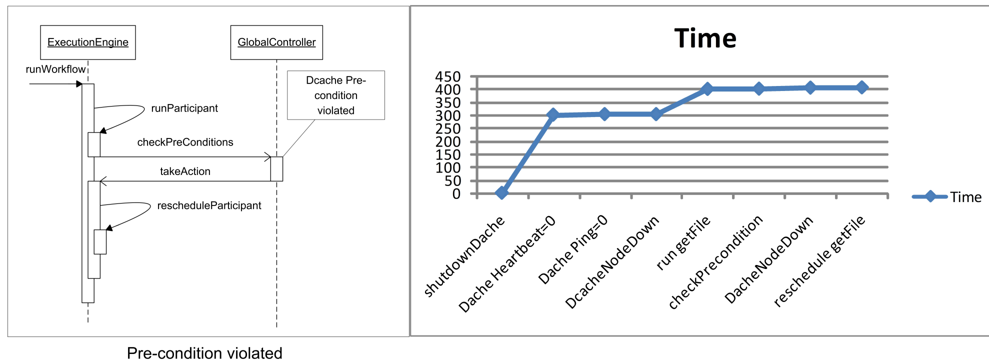


Figure 67: Sequence diagram depicting a dcache violation. The right hand figure shows the events as they are captured for the case study. Times shown on Y axis are in seconds relative to fault injection event in this figure. Notice that 300 second is the heartbeat timeout.

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 dache node making it unavailable at the time of execution of getFile. In the case of condition violation, as shown in Figure 67, the workflow engine is informed about the unavailability of dCache and an action must be taken. In the same figure, the chart on the right shows the timing of events as they were recorded. The heartbeat and node down events were generated by node diagnoser and heartbeat filters as discussed in figure 60.

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 failure mitigation scenario

We were able to simulate the effects of diskspace 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.

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. Cleaning a job that will eventually fail allows the portable batch scheduler to execute other jobs on the set of nodes that would have been otherwise reserved by the failed job. This improves the cluster productivity.

For this case study, we injected a node failure, 540 seconds after the light quark participant was scheduled on the node. The node was finally marked inoperable due to a communication problem with IPMI board. Later, during an offline diagnosis we found that there was a problem with the network interface card attached with the IPMI board. The IPMI failure was not intended but it was discovered during this case study. Figure 68 shows the sequence of events as generated during the case study.

Discussion on Model Predictive Recovery of Workflows

In previous sections, we have shown the ability to detect failures as a violation of certain pre-specified properties. To do this we observed the trace of events generated by sensors and filter using monitors. However, we only considered rescheduling the concerned participant as the possible course of action. We did not consider the impact of a participant failure on other participants belonging to the same workflow.

This was the objective of one of our previous papers [172]. This paper presents algorithms for performing look-ahead analysis on workflows in order to finish the maximal part of workflow that can be executed in case of failures of certain participants. To determine the extent to which

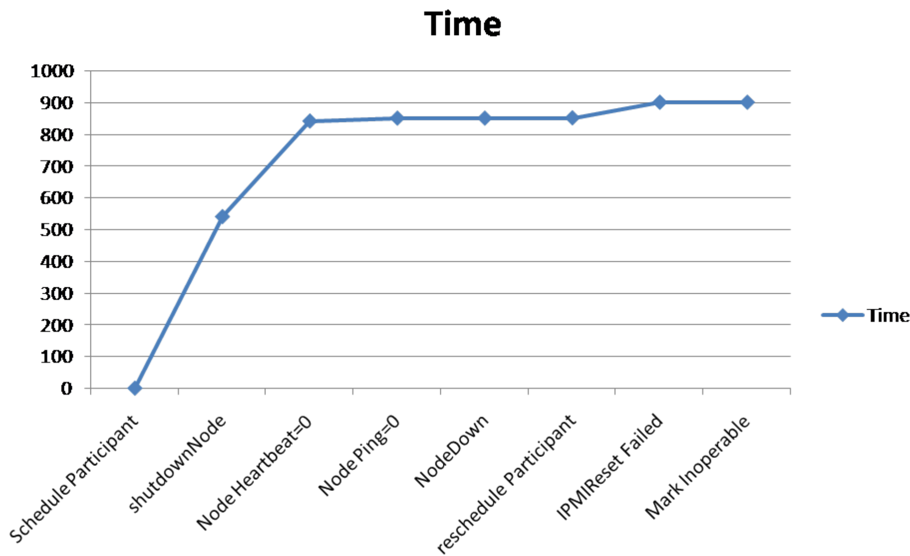


Figure 68: Sequence of events starting from scheduling of participant to marking the node offline. Heartbeat timeout was 300 seconds

a workflow can be executed, we construct a simple workflow execution engine simulator within the modeling environment that can be run in parallel with the actual system. When a fault occurs, all available autonomic reflexes are exhausted in an attempt to mitigate this fault at the local level before the workflow prediction is performed. If the fault is not mitigated by a reflex, the model is updated and a simple simulation algorithm begins in order to determine the future states of the workflow, given no external intervention and no future faults. Using the annotations of participants that described them as being critical or not, we determined the relative desirability of a partially completed workflow. If the workflow progressed to a state where no desirable jobs could be completed, the whole workflow was suspended. We are currently working on integrating this prediction engine with the existing execution engine.

Conclusions and Future Work

We presented a framework to execute scientific workflows reliably in this paper. 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 and used it to demonstrate feasibility of enabling fault-tolerant 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, and replacement of the current workflow engine. Further tests are planned on a virtual environment where failure scenarios can be simulated and repeated.

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.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

This dissertation has presented components of a model-based hierarchical management framework for scientific computing clusters. Portions of this framework are already being used on three large clusters at Fermi Lab. The components presented were:

1. A mitigation framework consisting of a hierarchical network of managers to monitor and repair faults in the system at local, regional, and global levels. The reflex portion of the architecture demonstrates how the basic concepts of human reflexes (fast, localized, pre-programmed responses) can provide tolerance toward a set of faults common to compute clusters.

2. A dynamic, light-weight, yet synchronized sensor framework that can be distributed on-demand across clusters.

3. A Workflow management system that allows specification of validity constraints, and enables data provenance, execution tracking and online monitoring of participants.

To measure the success of techniques developed during this research, we should compare it against the challenges outlined at the end of chapter II.

Conceptual and Standardization Challenges Scarf uses a timed automaton abstraction for designing mitigation behavior, which is an established model of computation for real-time systems. It uses a domain specific modeling language to represent workflow with the pre, post and invariant conditions. Monitoring sensors are defined using a standard template and a configuration database to provide a road map on how new sensors and components can be added to the framework.

Scalability Challenges The motivating policy behind the monitoring and mitigation framework is that they should be light-weight. Hence both monitors and reflex engines run on a single

thread using discrete event schedulers. Moreover, managers are arranged hierarchically to reduce the load on any one particular machine. In order to reduce the network stress on managing computers we use UDP, which is a stateless connection protocol, instead of TCP. Monitors across the whole cluster are synchronized so that they produce traffic in synced bursts, which reduces the average network contention with scientific applications.

Validation Challenges It has been shown that the mitigation policies used in SCARF can be verified using model checking tools such as UPPAAL.

Middleware Challenges We integrate the reliability components with the workflow management system to realize autonomic behaviors in a scalable manner, in spite of the application dynamism. Our framework uses syslog-ng and ssh protocol for communication. We are currently investigating the use of publish-subscribe middleware built upon OMG Data Distribution Service Standard [1].

Application Challenges Workflows in SCARF are managed by a workflow execution engine that monitors their state using sensor programs and then take corrective action that includes the specified reflex actions and a model predictive analysis for recovering workflows from failure [172].

The work presented here is a step towards advancement in design of processes and methods to create reliable and autonomic computing clusters. Novel research and discoveries that were part of this research have been published in a number of peer-reviewed venues. In long term, we are interested in developing the consolidated packages that are part of this framework and deploying it on clusters other than Fermi Lab to test its usefulness in external community.

[1] http://www.omg.org/technology/documents/dds_spec_catalog.htm

Future Research Goals

Future research efforts in this area include predicting the failure rate of workflow by analyzing the failure rates of individual participants. Towards that we have recently published a paper [194] that presents a technique for performing reachability analysis on probabilistic timed automaton models translated as Markov Decision process. By modeling the probability of success of a participant as a transition to the next participant we can determine the success of completing a workflow. This will prove useful in giving a feedback to users about their workflows based on historical failure rate of a class of algorithms.

Also of interest is the incorporation of proactive planning techniques in the workflow management framework that will use better heuristics for scheduling participants on machines that have been historically more reliable. We also wish to investigate research in better workflow modeling concepts that enable specification of transformations that a workflow model can undergo in case of failure. Such models will let us conduct a search through transformation space to find the optimum path for completing the workflow.

Lastly, we wish to continue our investigation in using Temporal Causal Graphs such as Timed Failure Propagation Graphs (TFPG) [188] for performing fault diagnosis. A TFPG is a directed graph in which nodes depict failures and discrepancies and edges represent dependencies of failure propagation over time. Currently, most of our effort is concentrated on identifying discrepancies. However, diagnosis will use the temporal sequence of these discrepancies to identify the most probable fault hypothesis that can explain them.

We see two issues with adapting TFPG to be used in the cluster framework (i) hardware components of a cluster are static. However, the software components are dynamic i.e. depending upon the workflow, the software dependencies will change. Consequently, the TFPG model will change based on the currently executing workflow. We will need models that are dynamic and yet scalable for analysis. (ii) Search space of a global TFPG model will be very large. We will need a component based approach where we generate local hypothesis in individual components and then consolidate them globally.

BIBLIOGRAPHY

[1] Ricardo Vilalta, Chidanand Apté, Joseph L. Hellerstein, Sheng Ma, and Sholom M. Weiss. Predictive algorithms in the management of computer systems. *IBM Systems Journal*, 41(3):461–474, 2002.

[2] Gerald Tesauro, Nicholas K. Jong, Rajarshi Das, and Mohamed N. Bennani. On the use of hybrid reinforcement learning for autonomic resource allocation. *Cluster Computing*, 10(3):287–299, 2007.

[3] Matthias Rohr, Marko Boskovic, Simon Giesecke, and Wilhelm Hasselbring. Model-driven development of self-managing software systems. In *Proceedings of the Workshop "Models@run.time" at the 9th International Conference on model Driven Engineering Languages and Systems (MoDELS/UML'06)*, 2006.

[4] Nicholas Carr. *The Big Switch: Rewiring the World, from Edison to Google*. W. W. Norton, January 2008.

[5] Ying Qian, A. Afsahi, and R. Zamani. Myrinet networks: a performance study. In *Network Computing and Applications, 2004. (NCA 2004). Proceedings. Third IEEE International Symposium on*, pages 323–328, 2004.

[6] Jiuxing Liu, A. Vishnu, and D.K. Panda. Building multirail infiniband clusters: Mpi-level design and performance evaluation. In *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, pages 33–33, 2004.

[7] Jim Gray, Jim Gray, and Jim Gray. A census of tandem system availability. In *IEEE Transactions on Reliability*, pages 40–9, 1990.

[8] J-C. Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *Proc. Twenty-Fifth International Symposium on Fault-Tolerant Computing, ' Highlights from Twenty-Five Years'*, page 2, June 27–30 1995.

[9] Jean-Claude Laprie, Christian Béounes, and Karama Kanoun. Definition and analysis of hardware- and software-fault-tolerant architectures. *Computer*, 23(7):39–51, 1990.

[10] D. Oppenheimer and D. Patterson. Architecture and dependability of large-scale internet services. *IEEE Internet Computing, special issue on Global Deployment of Data Centers*, September/October 2002.

[11] Abhishek Dubey, Steve Nordstrom, Turker Keskinpala, Sandeep Neema, Ted Bapty, and Gabor 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.

[12] Steve Nordstrom, Shweta Shetty, Sandeep K. Neema, and Theodore A. Bapty. Modeling reflex-healing autonomy for large scale embedded systems. *Systems, Man and Cybernetics, Part C, IEEE Transactions on*, 36(3):292–303, 2006.

[13] Abhishek Dubey, Steve Nordstrom, Turker Keskinpala, Sandeep Neema, and Ted Bapty. Verifying autonomic fault mitigation strategies in large scale real-time systems. *ease*, 0:129–140, 2006.

[14] Abhishek Dubey, Sandeep Neema, and Gabor Karsai. Designing a light-weight synchronized distributed monitoring framework for computing clusters. *IEEE Letters of AAS for EASe 2008*, 2009. Under Review.

[15] Abhishek Dubey, Steve Nordstrom, Turker Keskinpala, Sandeep Neema, Ted Bapty, and Gabor Karsai. Towards a model-based autonomic reliability framework for computing clusters. In *EASE '08*, pages 75–85, 2008.

[16] Abhishek Dubey, Gabor Karsai, and Sherif Abdelwahed. Compensating for timing jitter in computing systems with general-purpose operating systems. In *ISORC*, 2009. in press.

[17] Abhishek Dubey, Luciano Piccoli, James B. Kowalkowski, James N. Simone, Xian-He Sun, and Gabor Karsai. On the design of a framework to enable reliable scientific workflows. *Journal of Cluster Computing: Special Issue on Recent Research Advances in e-Science*, 2009. submitted.

[18] Abhishek Dubey, Sandeep Neema, Jim Kowalkowski, and Amitoj Singh. Scientific computing autonomic reliability framework. In *eScience*, 2008.

[19] Abhishek Dubey, Luciano Piccoli, James B. Kowalkowski, James N. Simone, Xian-He Sun, Gabor Karsai, and Sandeep Neema. Using runtime verification to design a reliable execution framework for scientific workflows. In *EASE '09*, 2009. inpress.

[20] Z. Fodor, S. D. Katz, and G. Papp. Better than $1/mflops sustained: a scalable pc-based parallel computer for lattice qcd. *Computer Physics Communications*, 152:121, 2003.

[21] Rajkumar Buyya. *High Performance Cluster Computing: Architectures and Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[22] B. Bode, D. Halstead, R. Kendall, and Z. Lei. The portable batch scheduler and the maui scheduler on linux clusters. *Proceedings of the 4th Annual Linux Showcase and Conference*, October 2000.

[23] Michael Strayer. Scidac 2008. *Journal of Physics: Conference Series*, 125, 2008.

[24] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery oriented computing (roc): Motivation, definition, techniques,. Technical report, University of California at Berkeley, Berkeley, CA, USA, 2002.

[25] Barry W. Johnson. An introduction to the design and analysis of fault-tolerant systems. *Fault-Tolerant Computer System Design*, pages 1–87, 1996.

[26] A. Avizienis. Fault-tolerance: The survival attribute of digital systems. *Proceeding of the IEEE*, 66(10):1109–1125, Oct. 1978.

[27] Michael R. Lyu. *Software Fault Tolerance*, volume New York, NY, USA. John Wiley & Sons, Inc, 1995.

[28] A. Avizienis and J.P.J. Kelly. Fault tolerance by design diversity: Concepts and experiments. *Computer*, 17(8):67–80, Aug. 1984.

[29] Brian Randell and Jie Xiu. The evolution of recovery block concept. *Software Fault Tolerance*, pages 1–21, 1995.

[30] B. Randell, P. Lee, and P. C. Treleaven. Reliability issues in computing system design. *ACM Comput. Surv.*, 10(2):123–165, 1978.

[31] A. Avizienis. The n - version approach to fault tolerant software. *IEEE Transactions on Software Engineering*, 11:1491–1501, December 1985.

[32] Susan S. Brilliant, John C. Knight, and Nancy G. Leveson. Analysis of faults in an n-version software experiment. *IEEE Transactions on Software Engineering*, 16(2):238–247, 1990.

[33] Aaron Brown and David Patterson. Embracing failure: A case for recovery-oriented computing (roc). *High Performance Transaction Processing Symposium*, 2001.

[34] David A. Patterson. Recovery oriented computing: A new research agenda for a new century. *hpca*, 00:0247, 2002.

[35] J. Kephart and D. Chess. The vision of autonomic computing. *IEEE Computer*, 2003.

[36] Aaron B. Brown and David A. Patterson. Rewind, repair, replay: three r's to dependability. In *EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop: beyond the PC*, pages 70–77, New York, NY, USA, 2002. ACM Press.

[37] Aaron B. Brown and David A. Patterson. Undo for operators: building an undoable e-mail store. In *ATEC'03: Proceedings of the USENIX Annual Technical Conference 2003 on USENIX Annual Technical Conference*, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[38] D. Oppenheimer, D. Oppenheimer, A. Brown, J. Beck, D. Hettena, J. Kuroda, N. Treuhaft, D.A. Patterson, and K. Yelick. Roc-1: hardware support for recovery-oriented computing. *IEEE Transactions on Computers,*, 51(2):100–107, 2002.

[39] Paul Horn. Autonomic computing: Ibm's perspective on the state of information technology. `http://research.ibm.com/autonomic/manifesto`, October 2001.

[40] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003.

[41] Hewlett-Packard. Adaptive enterprise: Business and IT synchronized to capitalize on change. Available at http://h71028.www7.hp.com/ERC/downloads/5982-3185EN.pdf.

[42] Microsoft. Dynamic systems initiative. White Paper, Online, March 2004.

[43] IBM Global Services. IBM global services and autonomic computing. Technical report, http://www-03.ibm.com/autonomic/pdfs/wp-igs-autonomic.pdf, 2002.

[44] A. Ganek. Autonomic computing: implementing the vision. In *Autonomic Computing Workshop, 2003*, pages 1–1, 25 June 2003.

[45] Huaglory Tianfield. Multi-agent autonomic architecture and its application in e- medicine. In *IAT '03: Proceedings of the IEEE/WIC International Conference on Intelligent Agent Technology*, page 601, Washington, DC, USA, 2003. IEEE Computer Society.

[46] Roy Sterritt, Manish Parashar, Huaglory Tianfield, and Rainer Unland. A concise introduction to autonomic computing. *Advanced Engineering Informatics*, 19(3):181–187, July 2005.

[47] Gerald Tesauro, David M. Chess, William E. Walsh, Rajarshi Das, Alla Segal, Ian Whalley, Jeffrey O. Kephart, and Steve R. White. A multi-agent systems approach to autonomic computing. *Autonomous Agents and Multi-Agent Systems*, 2004.

[48] S.R. White, J.E. Hanson, I. Whalley, D.M. Chess, and J.O. Kephart. An architectural approach to autonomic computing. In J.E. Hanson, editor, *Proc. International Conference on Autonomic Computing*, pages 2–9, 2004.

[49] Jeffery Kephart. Research challenges of autonomic computing. In *Proc. 27th International Conference on Software Engineering ICSE 2005*, pages 15–22, 2005.

[50] R. Sterritt. Towards autonomic computing: effective event management. In *Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE*, pages 40–47, 5-6 Dec. 2002.

[51] L. W. Russell, S. P. Morgan, and E. G. Chron. Clockwork: A new movement in autonomic systems. *IBM Syst. J.*, 42(1):77–84, 2003.

[52] L. W. Russell, S. P. Morgan, and E. G. Chron. On-line model selection procedures in clockwork. In *IJCAI workshop on AI and autonomic computing: developing a research agenda for self-managing computer systems*, Acapulco, Mexico, 2003.

[53] Uri Lerner, Ronald Parr, Daphne Koller, and Gautam Biswas. Bayesian fault detection and diagnosis in dynamic systems. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 531–537. The MIT Press, 2000.

[54] Haipeng Guo. A bayesian approach for automatic algorithm selection. In *IJCAI workshop on AI and autonomic computing: developing a research agenda for self-managing computer systems*, Acapulco, Mexico, 2003.

[55] S. Aiber, S. Aiber, D. Gilat, A. Landau, N. Razinkov, A. Sela, and S. Wasserkrug. Autonomic self-optimization according to business objectives. In D. Gilat, editor, *Proc. International Conference on Autonomic Computing*, pages 206–213, 2004.

[56] Stuart Russell and Peter Norvig. *Artifical Intelligence A Modern Approach*. Prenctice Hall, 1995.

[57] B. Srivastava and S. Kambhampati. The case for automated planning in autonomic computing. In S. Kambhampati, editor, *Proc. Second International Conference on Autonomic Computing ICAC 2005*, pages 331–332, 2005.

[58] A. Keller, J.L. Hellerstein, J.L. Wolf, K.-L. Wu, and V. Krishnan. The champs system: change management with planning and scheduling. In *Proc. IEEE/IFIP Network Operations and Management Symposium NOMS 2004*, volume 1, pages 395–408, 19–23 April 2004.

[59] J.P. Bigus, D.A. Schlosnagle, J.R. Pilgrim, W.N. Mills, and Y. Diao. Able: a toolkit for building multiagent autonomic systems - agent building and learning environment. *IBM Systems Journal*, 41:350, 2002.

[60] B. Srivastava, J.P. Bigus, and D.A. Schlosnagle. Bringing planning to autonomic applications with able. In J.P. Bigus, editor, *Proc. International Conference on Autonomic Computing*, pages 154–161, 2004.

[61] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[62] Gerald Tesauro. Online resource allocation using decompositional reinforcement learning. In *AAAI*, pages 886–891, 2005.

[63] Gerald Tesauro, William E. Walsh, and Jeffrey O. Kephart. Utility-function-driven resource allocation in autonomic systems. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pages 342–343, Washington, DC, USA, 2005. IEEE Computer Society.

[64] Karl-Erik Årzén, Anders Robertsson, Dan Henriksson, Mikael Johansson, Håkan Hjalmarsson, and Karl Henrik Johansson. Conclusions of the artist2 roadmap on control of computing systems. *SIGBED Rev.*, 3(3):11–20, 2006.

[65] Mieczyslaw M. Kokar, Kenneth Baclawski, and Yonet A. Eracar. Control theory-based foundations of self-controlling software. *IEEE Intelligent Systems*, 14(3):37–45, 1999.

[66] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[67] Katsuhiko Ogata. *Modern control engineering (3rd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[68] Katsuhiko Ogata. *Discrete-time control systems (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[69] Nathan Gnanasambandam, Seokcheon Lee, and Soundar R. T. Kumara. An autonomous performance control framework for distributed multi-agent systems: a queueing theory based

approach. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 1313–1314, New York, NY, USA, 2005. ACM.

[70] Srinivasan Keshav. A control-theoretic approach to flow control. *SIGCOMM Comput. Commun. Rev.*, 21(4):3–15, 1991.

[71] Kang Li, M.H. Shor, J. Walpole, C. Pu, and D.C. Steere. Modeling the effect of short-term rate variations on tcp-friendly congestion control behavior. In M.H. Shor, editor, *Proc. American Control Conference the 2001*, volume 4, pages 3006–3012 vol.4, 2001.

[72] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking, 2002.

[73] G. Tziallas and B. Theodoulidis. Building autonomic computing systems based on ontological component models and a controller synthesis algorithm. In *DEXA '03: Proceedings of the 14th International Workshop on Database and Expert Systems Applications*, page 674, Washington, DC, USA, 2003. IEEE Computer Society.

[74] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[75] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to pvs. In *Workshop on Industrial-Strength Formal Specification Techniques*, 1195.

[76] Yixin Diao, Joseph L. Hellerstein, Sujay Parekh, Rean Griffith, Gail Kaiser, and Dan Phung. Self-managing systems: A control theory foundation. *ecbs*, 00:441–448, 2005.

[77] John A. Stankovic, Chenyang Lu, and Sang H. Son. The case for feedback control real-time scheduling. Technical report, University of Virginia, Charlottesville, VA, USA, 1998.

[78] Chenyang Lu, John A. Stankovic, Sang H. Son, and Gang Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms*. *Real-Time Systems*, 23(1-2):85–126, 2002.

[79] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 145–158, Berkeley, CA, USA, 1999. USENIX Association.

[80] Lui Sha, Xue Liu, Ying Lu, and Tarek Abdelzaher. Queueing model based network server performance control. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, pages 81–90, Washington, DC, USA, 2002. IEEE Computer Society.

[81] Tarek Abdelzaher, Kang G. Shin, and Nina Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Trans. Parallel Distrib. Syst.*, 13(1):80–96, January 2002.

[82] Xiaorui Wang, Chenyang Lu, and Xenofon Koutsoukos. Feedback utilization control in distributed real-time systems with end-to-end tasks. *IEEE Trans. Parallel Distrib. Syst.*, 16(6):550–561, 2005.

[83] Sherif Abdelwahed, Nagarajan Kandasamy, and Sandeep Neema. A control-based framework for self-managing distributed computing systems. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 3–7, New York, NY, USA, 2004. ACM.

[84] Nagarajan Kandasamy, Sherif Abdelwahed, and Mohit Khandekar. A hierarchical optimization framework for autonomic performance management of distributed computing systems. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, page 9, Washington, DC, USA, 2006. IEEE Computer Society.

[85] Nagarajan Kandasamy, Sherif Abdelwahed, and John P. Hayes. Self-optimization in computer systems via on-line control: Application to power management. In *ICAC '04: Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*, pages 54–61, Washington, DC, USA, 2004. IEEE Computer Society.

[86] Charles Lefurgy, Xiaorui Wang, and Malcolm Ware. Server-level power control. In *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing*, page 4, Washington, DC, USA, 2007. IEEE Computer Society.

[87] Ronghua Zhang, Chenyang Lu, Tarek F. Abdelzaher, and John A. Stankovic. Controlware: A middleware architecture for feedback control of software performance. In *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 301, Washington, DC, USA, 2002. IEEE Computer Society.

[88] Tarek F. Abdelzaher, John A. Stankovic, Chenyang Lu, Ronghua Zhang, and Ying Lu. Feedback performance control in software services. *IEEE Control Systems Magazine*, 23:74–90, June 2003.

[89] Doron A. Peled. *Software reliability methods*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[90] P Ramadge and W. Wonham. Supervisory control of a class of discrete event processes. *Siam J. Control and Optimization*, 25(1), 1987.

[91] P. Ramadge and W. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[92] Markus Debusmann, Markus Schmid, and Reinhold Kroeger. Model-driven self-management of legacy applications. *Distributed Applications and Interoperable Systems*, 3543:56–67, 2005.

[93] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 371–380, New York, NY, USA, 2006. ACM.

[94] Kurt Geihs, Roland Reichle, Mohammad U. Khan, Arnor Solberg, and Svein Hallsteinsen. Model-driven development of self-adaptive applications for mobile devices: (research summary). In *SEAMS '06: Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*, pages 95–95, New York, NY, USA, 2006. ACM.

[95] David Garlan, Shang W. Cheng, and Bradley Schmerl. Increasing system dependability through architecture-based self-repair. *Architecting Dependable Systems*, 2003.

[96] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[97] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. Towards architecture-based self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 21–26, New York, NY, USA, 2002. ACM Press.

[98] Rational Software Corporation et al. *Object Constraint Language Specification ver 1.1*, Sept 1997.

[99] Janos Sztipanovits and Gabor Karsai. Model-integrated computing. *Computer*, 30(4):110–111, 1997.

[100] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge MA, USA, 2000.

[101] Ji Zhang and Betty H. C. Cheng. Specifying adaptation semantics. In *WADS '05: Proceedings of the 2005 workshop on Architecting dependable systems*, pages 1–7, New York, NY, USA, 2005. ACM.

[102] Brian C. Williams and P. Pandurang Nayak. A model-based approach to reactive self-configuring systems. In *AAAI/IAAI, Vol. 2*, pages 971–978, 1996.

[103] Brian C. Williams and Robert Ragno. Conflict-directed A* and its role in model-based embedded systems. *Journal of Discrete Applied Math, Special Issue on Theory and Applications of Satisfiability Testing*, January 2003.

[104] Paul Robertson and Brian Williams. Automatic recovery from software failure. *Commun. ACM*, 49(3):41–47, 2006.

[105] B.C. Williams, B.C. Williams, M.D. Ingham, S.H. Chung, and P.H. Elliott. Model-based programming of intelligent embedded systems and robotic space explorers. *Proceedings of the IEEE*, 91(1):212–237, 2003.

[106] Brian C. Williams, Michel Ingham, Seung Chung, Paul Elliott, Michael Hofbaur, and Gregory T. Sullivan. Model-based programming of fault-aware systems. *AI Magazine*, 24(4):61–75, 2004.

[107] Charles Pecheur and Reid G. Simmons. From livingstone to smv. In *FAABS '00: Proceedings of the First International Workshop on Formal Approaches to Agent-Based Systems-Revised Papers*, pages 103–113, London, UK, 2001. Springer-Verlag.

[108] Sam Lightstone. Seven software engineering principles for autonomic computing development. *ISSE*, 3(1):71–74, 2007.

[109] Mary Shaw. "self-healing": softening precision to avoid brittleness: position paper for woss '02: workshop on self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 111–114, New York, NY, USA, 2002. ACM Press.

[110] David W. Bustard and Roy Sterritt. A requirements engineering perspective on autonomic systems development. *Autonomic Computing: Concepts, Infrastructure, and Applications*, pages 19–33, 2006.

[111] A. Taleb-Bendiab, David W. Bustard, Roy Sterritt, A. G. Laws, and Frank Keenan. Model-based self-managing systems engineering. In *DEXA Workshops*, pages 155–159, 2005.

[112] Peter Checkland and Costas Tsouvalis. Reflecting on ssm: The link between root definitions and conceptual models. *Systems Research and Behavioral Science*, 14(3):153–268, 1997.

[113] J. Kaiser, G.and Parekh, P. Gross, and G. Valetto. Kinesthetics extreme: an external infrastructure for monitoring distributed legacy systems. In J. Parekh, editor, *Proc. Autonomic Computing Workshop*, pages 22–30, 2003.

[114] K.P. Birman, R. van Renesse, and W. Vogels. Navigating in the storm: using astrolabe for distributed self-configuration, monitoring and adaptation. In R. van Renesse, editor, *Proc. Autonomic Computing Workshop*, pages 4–13, 2003.

[115] Zbigniew T. Kalbarczyk, Ravishankar K. Iyer, Saurabh Bagchi, and Keith Whisnant. Chameleon: A software infrastructure for adaptive fault tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):560–579, June 1999.

[116] Shikha Ahuja, Ted Bapty, Harry Cheung, Michael Haney, Zbigniew Kalbarczyk, Akhilesh Khanna, Jim Kowalkowski, Derek Messie, Daniel Mosse, Sandeep Neema, Steve Nordstrom, Jae Oh, Paul Sheldon, Shweta Shetty, Long Wang, and Di Yao. Rtes demo system2004. *SIGBED Rev.*, 2(3):1–6, 2005.

[117] Chris Wells. The oceanstore archive: Goals, structures, and self-repair. Master's thesis, University of California, Berkeley, May 2001.

[118] J. Menon, D. A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg. Ibm storage tank– a heterogeneous scalable san file system. *IBM Syst. J.*, 42(2):250–267, 2003.

[119] Christian Poellabauer. *Q-fabric: system support for continuous online quality management*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 2004. Adviser-Karsten Schwan.

[120] Surajit Chaudhuri and Vivek Narasayya. Self-tuning database systems: a decade of progress. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 3–14. VLDB Endowment, 2007.

[121] Guy M. Lohman and Sam S. Lightstone. Smart: making db2 (more) autonomic. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 877–879. VLDB Endowment, 2002.

[122] Robert Pool. Natutal selection: A new computer program classifies documents automatically. `http://domino.watson.ibm.com/comm/wwwr_thinkresearch.nsf/pages/selection200.html`, 2002.

[123] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Multimethod communication for high-performance metacomputing applications. In *Supercomputing, 1996. Proceedings of the 1996 ACM/IEEE Conference on*, pages 41–41, 1996.

[124] M. Agarwal, V. Bhat, Z. Li, H. Liu, B. Khargharia, V. Matossian, V. Putty, C. Schmidt, G. Zhang, S. Hariri, and M. Parashar. Automate: Enabling autonomic applications on the grid. *Proceedings of the Autonomic Computing Workshop, 5th Annual International Active Middleware Services Workshop (AMS2003)*, pages 48–57, June 2003.

[125] M. Parashar, Z. Li, H. Liu, V. Matossian, and C. Schmidt. Enabling autonomic grid applications: Requirements, models and infrastructures. *Lecture Notes in Computer Science, Self-Star Properties in Complex Information Systems*, 2005.

[126] Xiangdong Dong, S. Hariri, Lizhi Xue, Huoping Chen, Ming Zhang, S. Pavuluri, and S. Rao. Autonomia: an autonomic computing environment. In S. Hariri, editor, *Proc. IEEE International Performance, Computing, and Communications Conference*, pages 61–68, 2003.

[127] Vincent Matossian, Viraj Bhat, Manish Parashar, Małgorzata Peszyńska, Mrinal Sen, Paul Stoffa, and Mary F. Wheeler. Autonomic oil reservoir optimization on the grid: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(1):1–26, 2005.

[128] Guangsen Zhang and Manish Parashar. Cooperative mechanism against ddos attacks. In *Security and Management*, pages 86–96, 2005.

[129] Zhengyu Liang, Yundong Sun, and Cho-Li Wang. Clusterprobe: an open, flexible and scalable cluster monitoring tool. In *Cluster Computing, 1999. Proceedings. 1st IEEE Computer Society International Workshop on*, pages 261–268, 2-3 Dec. 1999.

[130] D. Gunter, B. Tierney, B. Crowley, M. Holding, and J. Lee. Netlogger: a toolkit for distributed system performance analysis. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2000. Proceedings. 8th International Symposium on*, pages 267–273, 29 Aug.-1 Sept. 2000.

[131] M. Brim, A. Geist, B. Luethke, J. Schwidder, and S.L. Scott. M3c: managing and monitoring multiple clusters. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 386–393, 15-18 May 2001.

[132] Brian Tierney, Brian Crowley, Dan Gunter, Jason Lee, and Mary Thompson. A monitoring sensor management system for grid environments. *Cluster Computing*, 4(1):19–28, 2001.

[133] J.M. Brandt, A.C. Gentile, D.J. Hale, and P.P. Pebay. Ovis: a tool for intelligent, real-time monitoring of computational clusters. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 8pp., 25-29 April 2006.

[134] T.C. Ferreto, C.A.F. de Rose, and L. de Rose. Rvision: An open and high configurable tool for cluster monitoring. In *Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on*, pages 75–75, 21-24 May 2002.

[135] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(5-6):817–840, 2004.

[136] Richard C. Harlan. Network management with nagios. *Linux Journal*, 2003(111):3, 2003.

[137] M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang, and S. Hariri. Automate: Enabling autonomic grid applications. *Cluster Computing: The Journal of Networks, Software Tools, and Applications, Special Issue on Autonomic Computing*, 9(1), 2006.

[138] The Real Time Embedded Systems Group. The Real Time Embedded Systems Group. `http://www-btev.fnal.gov/public/hep/detector/rtes`.

[139] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A tutorial on uppaal. In *SFM*, pages 200–236, 2004.

[140] RC De. Vries. An automated methodology for generating a fault tree. *IEEE Transactions On Reliability*, 39:76–86, April 1990.

[141] Ron J. Patton, Paul M. Frank, and Robert N. Clarke, editors. *Fault diagnosis in dynamic systems: theory and application*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[142] Paul M. Frank. Fault diagnosis in dynamic systems using analytical and knowledge-based redundancya survey and some new results. *Automatica*, 26(3):459–474, 1990.

[143] Ahmed A. Rafea, Alyman El Desouki, and S. El-Moniem. Combined model expert system for electronics fault diagnosis. In *IEA/AIE '90: Proceedings of the 3rd international conference on Industrial and engineering applications of artificial intelligence and expert systems*, pages 23–31, New York, NY, USA, 1990. ACM Press.

[144] J de Kleer and B C Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[145] Janos Gertler. *Fault Detection and Diagnosis in Engineering Systems*. Marcel Dekker, May 1998.

[146] Gianfranco Lamperti and Marina Zanella. Diagnosis of discrete event systems from uncertain temporal observations. *Artificial Intelligence*, 137(1-2):91–163, 2002.

[147] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D.C. Teneketzis. Failure diagnosis using discrete-event models. *IEEE Transactions On Control System Technology*, 4(2):105–124, March 1996.

[148] Armen Aghasaryan, Eric Fabre, Albert Benveniste, Rene Boubour, and Claude Jard. Fault detection and diagnosis in distributed systems: An approach by partially stochastic petri nets. *Discrete Event Dynamic Systems, Volume 8, Issue 2*, pages 203–231, June 1998.

[149] Jan Lunze. Diagnosis of quantized systems based on a timed discrete-event model. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 30(3):322–335, 2000.

[150] W. F. Truszkowski, M. G. Hinchey, J. L. Rash, and C. A. Rouff. Autonomous and autonomic systems: a paradigm for future space exploration missions. *Systems, Man and Cybernetics, Part C, IEEE Transactions on*, 36(3):279–291, 2006.

[151] R. Sterritt and M. Hinchey. Autonomic computing - panacea or poppycock? In *Engineering of Computer-Based Systems, 2005. ECBS '05. 12th IEEE International Conference and Workshops on the*, pages 535–539, 4-7 April 2005.

[152] Manish Parashar and Salim Hariri. Autonomic computing: An overview. In *UPP*, pages 257–269, 2004.

[153] S. Shetty, S. Nordstrom, S. Ahuja, Di Yao, T. Bapty, and S. Neema. Systems integration of large scale autonomic systems using multiple domain specific modeling languages. In *Engineering of Computer-Based Systems, 2005. ECBS '05. 12th IEEE International Conference and Workshops on the*, pages 481–489, 4-7 April 2005.

[154] Di Yao, Sandeep Neema, Steve Nordstrom, Shweta Shetty, Shikha Ahuja, and Ted Bapty. Specification and implementation of autonomic large-scale system behaviors using domain specific modeling language tools. In *Proceding of International Conference on Software Engineering and Practice*, June 2005.

[155] Mitchel Resnick. Decentralized modeling and decentralized thinking. *Modeling and Simulation in Science and Mathematics Education*, pages 114–137, 1999.

[156] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 2005.

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

[158] Thomas Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real time systems. *Information and Computation*, 111(2):193–244, 1994.

[159] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaala tool suite for automatic verification of real-time systems. In *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control*, pages 232–243, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[160] Johannes Gutleber and et. al. Clustered data acquisition for the CMS experiment. In *International Conference on Computing In High Energy and Nuclear Physics*, September 2001.

[161] Simon Kwan. The btev pixel detector and trigger system. In *FERMILAB-Conf-02/313*, December 2002.

[162] Jeff Rothenberg. The nature of modeling. *Artificial intelligence, simulation & modeling*, pages 75–92, 1989.

[163] Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[164] Steve Nordstrom, Ted Bapty, Sandeep Neema, Abhishek Dubey, and Turker Keskinpala. A guided explorative approach for autonomic healing of model-based systems. In *Second IEEE conference on Space Mission Challenges for Information Technology, Mini workshop on Autonomous and Autonomic Systems in Space Exploration*, July 2006.

[165] Steve Nordstrom, Abhishek Dubey, Turker Keskinpala, Sandeep Neema, and Ted Bapty. Ghost: Guided healing and optimization search technique for healing large-scale embedded systems. In *EASE '06: Proceedings of the Third IEEE International Workshop on Engineering of Autonomic & Autonomous Systems (EASE'06)*, pages 54–60, Washington, DC, USA, 2006. IEEE Computer Society.

[166] Pavel Krcál and Wang Yi. Decidable and undecidable problems in schedulability analysis using timed automata. In *TACAS*, pages 236–250, 2004.

[167] Gabor Madl, Sherif Abdelwahed, and Gabor Karsai. Automatic verification of component-based real-time corba applications. In *RTSS*, pages 231–240, 2004.

[168] S Yovine. Kronos: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 126:110–122, 1997.

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

[170] M.J. Haney, S. Ahuja, G. Bapty, H. Cheung, Z. Kalbarczyk, A. Khanna, J. Kowalkowski, D. Messie, D. Mosse, S. Neema, S. Nordstrom, Jae Oh, P. Sheldon, S. Shetty, D. Volper, Long Wang, and Di Yao. The rtes project - btev, and beyond. In *Real Time Conference, 2005. 14th IEEE-NPSS*, page 4pp., 4-10 June 2005.

[171] D. J. Holmgren. PC clusters for lattice qcd, 2004.

[172] Steve Nordstrom, Abhishek Dubey, Turker Keskinpala, Rahul Datta, Sandeep Neema, and Ted Bapty. Model predictive analysis for autonomicworkflow management in large-scale scientific computing environments. In *EASE '07: Proceedings of the Fourth IEEE International Workshop on Engineering of Autonomic and Autonomous Systems*, pages 37–42, Washington, DC, USA, 2007. IEEE Computer Society.

[173] Gabor Karsai, Janos Sztipanovits, Ákos Lédeczi, and Ted Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, 2003.

[174] C. Lonvick. The BSD syslog protocol, 2001.

[175] Abraham Silberschatz and Peter Baer Galvin. *Operating System Concepts*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[176] P. Marti, P. Marti, J.M. Fuertes, G. Fohler, and K. Ramamritham. Jitter compensation for real-time control systems. In J.M. Fuertes, editor, *Proc. 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, pages 39–48, 2001.

[177] Karl-Erik Årzén, Bo Bernhardsson, Johan Eker, Anton Cervin, Klas Nilsson, Patrik Persson, and Lui Sha. Integrated control and scheduling. Technical Report ISRN LUTFD2/TFRT--7586--SE, Department of Automatic Control, Lund Institute of Technology, Sweden, aug 1999.

[178] Mpi-2: Extensions to the message-passing interface.

[179] Ian J. Taylor, Ewa Deelman, Dennis B. Gannon, and Matthew Shields. *Workflows for e-Science: Scientific Workflows for Grids*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[180] Yong Zhao, Mihael Hategan, Ben Clifford, Ian Foster, Gregor Von Laszewski, Ioan Raicu, Tiberiu Stef-praun, and Mike Wilde. Swift: Fast, reliable, loosely coupled parallel computation. *services*, 00:199–206, 2007.

[181] 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.

[182] Cluster Resources. *TORQUE Administrator Manual*.

[183] Cluster Resources. *Maui Scheduler Administrator's Guide*.

[184] The syslog-ng 3.0 administrator's guide.

[185] 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.

[186] Michael Herz. Remote system management principles. *Cluster Computing, IEEE International Conference on*, 0:137, 2000.

[187] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The generic modeling environment. In *Workshop on Intelligent Signal Processing, Budapest, Hungary*, volume 17, May 2001.

[188] Sherif Abdelwahed, Gabor Karsai, and Gautam Biswas. Notions of diagnosability for timed failure propagation graphs. In *Proceeding of IEEE Systems Readiness Technology Conference, AUTOTESTCON*, 2006.

[189] T. Fahringer, R. Prodan, Rubing Duan, F. Nerieri, S. Podlipnig, Jun Qin, M. Siddiqui, Hong-Linh 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.

[190] Grzegorz Malewicz, Ian Foster, Arnold L. Rosenberg, and Michael Wilde. A tool for prioritizing dagman jobs and its evaluation. *Journal of Grid Computing*, 5(2):197–212, 2007.

[191] Petia Wohed, Birger Andersson, Arthur H.M. ter Hofstede, Nick Russell, and Wil M.P. van der Aalst. Patterns-based evaluation of open source bpm systems: The cases of jbpm, openwfe, and enhydra shark. Technical report, BPM Center, 2007.

[192] Dave Thomas, David Hansson, Leon Breedt, Mike Clark, James Duncan Davidson, Justin Gehtland, and Andreas Schwarz. *Agile Web Development with Rails*. Pragmatic Bookshelf, 2006.

[193] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.

[194] Abhishek Dubey, Derek Riley, and Sherif Abdelwahed. Modeling and analysis of probabilistic timed systems. In *16th IEEE Conference on Engineering of Computer Based Systems (ECBS)*, 2009. in press.