Toward self-reconfiguring, fault-adaptive, high-performance distributed real-time systems*

Steven G. Nordstrom, Shweta Shetty, Di Yao, Shikha Ahuja, Sandeep Neema, Ted Bapty, Gabor Karsai

Institute for Software Integrated Systems, Vanderbilt University, Box 1829, Station B Nashville, TN USA
e-mail: {steve-o, shweta, dyao, shikha, sandeep, bapty, gabor}@isis.vanderbilt.edu

Submitted for review 30 April, 2005

Abstract. Designing embedded systems that guarantee some form of reasonable behavior in the presence of failure is a difficult task, and the addition of scalability and real-time constraints only serve to exacerbate the problem. In large-scale systems such as those used in high-energy physics experiments, the necessary conditions to ensure complete and reliable operation of the system can not be guaranteed. Moreover, typical methods of fault tolerance can not be applied to these systems given their size and budgetary constraints. As the performance, size, and inherent complexity of these systems increase, it becomes necessary to develop increasingly advanced tools and techniques to help manage their design, implementation, and configuration. This paper outlines an approach toward designing large-scale embedded systems by leveraging concepts of Model Integrated Computing to configure a scalable Reflex and Healing architecture to implement model-based user-defined fault-mitigation strategies within the system.

Keywords: Fault-mitigating -- Fault-tolerant -- Reflex -- Healing -- Model -- Integrated

1 Introduction and Problem Motivation

As high performance computing grows more prominent and accessible, system designers leverage the commoditization of components by designing increasingly complex embedded computers not from scratch but by composing collections of existing components to form larger, more elaborate systems. This activity places considerable strain on both the design process and the systems engineering process, and serves to add complexity to the design; both of which can adversely affect reliability.

Reliable operation of a high performance distributed computer system is dependant upon 1.) the proper operation of each system component, and 2.) a proper interaction between components in the composition. In systems with many thousands of components, neither of these conditions can be guaranteed. Proper consideration must be given to the occurrence of failure in both the physical components as well as in software; one must anticipate that failures will occur sometime within the operational lifetime of the system.

Acceptance of possible component failures should be evident in every aspect of system design. Complex embedded systems should behave such that some degree of component or subsystem malfunction does not result in catastrophic system failure. Ensuring reliable and predictable system behavior in the presence of failure is a difficult and costly task, and the addition of real-time constraints only serves to exacerbate the problem. Internal knowledge of the system structure (past and present) may be required to reason not only about possible failure mitigation actions, but also about the effects of undergoing such adaptations. This becomes especially true in systems where real-time constraints affect the time allowed for such reasoning processes. Determination of when to perform adaptation, in what ways may the system be reconfigured, the amount of time used to perform the adaptation, and what benefit or increased utility these actions will yield are all important design decisions that must be evaluated when building a system under such constraints.

1.1 Problem Motivation

Of particular interest to us are the embedded systems used to perform online phenomena capture, signal processing, and data collection for high energy physics (HEP) experiments. Physicists and engineers collaborate to build massive real-time embedded systems which perform the necessary data acquisition and signal processing required for capturing, processing, and recording the various physical phenomena that occur inside a particle accelerator. HEP experiments have a physical process interaction periodicity as high as ~25 nanoseconds yielding aggregate front-end data rates on the order of several Terabytes/second with ~1 Petabyte/year in permanent storage needs [1] [2]. Typical hardware architectures for HEP computation are composed of several thousands of processors. Given that there can exist only a minimal amount of redundant resources, designers of these systems face considerable difficulty when applying traditional techniques for fault tolerance [3].

Typical baseline criteria for these systems are the following:

- Data must not be lost
- Maximum latency between phenomena detection and storage must be guaranteed
- The system must be scalable (on the order of several thousand processors)
- The system must exhibit *reasonable* behavior in the presence of failures
- Redundancy must be kept to a minimum (<10% overhead)
- Operator intervention should be minimal for all but the most critical tasks
- Development and maintenance costs should be minimized

A *reasonable* behavior is one which is predictable, stable, and meets the criteria set forth at design time regarding minimum system performance and graceful degradation during conditions of failure. Clearly, these concerns can be conflicting; a reduction in development costs may prohibit the implementation of customized fault mitigating tasks and behaviors; Likewise, if redundancy is minimized the ability to exhibit reasonable fault tolerant behavior clearly suffers.

The challenges then become the following:

- How does one design and manage a system such that these goals can be met with some degree of certainty?
- What architectures are suited for these classes of systems and do they scale to thousands nodes?
- Do existing tools and techniques exist which can be used to reduce the complexity of the design process?
- What levels of time, expertise, and risk are associated with customizing the system to suit ones specific needs?

This paper outlines our proposed and tested design tools, techniques and best principles that should be used when building complex, large-scale, embedded systems when real-time and fault-management constraints are applied.

2 Managing complexity through modeling

A critical aspect of engineering large computer systems is the management of inherent complexities of the final product. Proper design, improved processes, and better supporting tools are all potent stimuli of more robust system construction. This chapter will cover some specific tools and techniques regarding the management of design and integration of large scale computer based systems.

2.1 Model Integrated Computing

The software engineering community has seen a marked increase over the last decade in the use of model-centric tools which aid in system design and complexity management [4] [5]. Early computer based modeling tools were confined to highly specialized domains whose budgets could afford the added costs of computer based modeling. For example, early and expensive CAD-CAM tools were being used in the 1970's for circuit design and analysis, and CADD systems for modeling aircraft and automotive designs were becoming more prevalent [6]. The success of these tools and of the model-based design approach in general is well known, and over time the costs of using such tools has made it increasingly feasible to apply model-based design approaches to other domains, including software design.

Model Integrated Computing (MIC) [7] has proven itself in recent years as a sound method of applying computer based modeling approaches to a variety of problem domains. MIC incorporates the creation of domain-specific, model-based abstractions which serve to capture relevant aspects of a target system. These models can then be programmatically traversed and transformed to produce a variety of domain specific artifacts. These models are often transformed into alternate but equivalent representations which can be used by external analysis and simulation tools to verify

certain properties of the system [8] [9]. Examples of MIC uses are the generation of real-time schedules from a software model, the creation of configuration files to integrate distributed systems, or the generation of source code that can be integrated into an existing framework [10].

MIC relies heavily on the use of domain-specific modeling languages to capture relevant characteristics of an object or system of objects. A domain-specific modeling language (DSML) allows a designer to describe objects in terms of the domain rather than in terms of traditional computer languages. The Generic Modeling Environment (GME) [11] [12] is a freely available tool which provides a platform upon which to perform MIC design and development. Specifically, the GME is a configurable and domain-independent modeling environment that supports the creation and instantiation of multiple user defined (domain-specific) modeling languages.

Recent and State-of-the-art technologies for model driven design include research into graphical methods of model transformations [13], and leveraging the OMG's Model Driven Architecture (MDA) 14 for meta-level language specification [15]. Microsoft is bringing their .NET Software Factories framework to fruition for high level modeling of software [16].

2.2 Using models effectively

Without proper analysis and translation tools, models are nothing but design documents; they may have well defined semantics, but their only role is that of documentation. The difference between traditional models (PowerPoint or Visio diagrams, blueprints, or CAD drawings) and MIC models is that in addition to capturing and documenting properties of an object, computer-based MIC models can be programmatically leveraged to *produce something* for the designer. The application of semantics to a model is often referred to as model interpretation [17], translation [18], or transformation [19].

While domain models can be interpreted and transformed to produce a variety of artifacts, the models themselves need only be defined once. A good example of this is the generation of either C++ or Java classes from the same UML class diagram [20] or in the creation of platform-specific executable code from a StateCharts model [21]. Similarly, a single set of system models can be used to drive simulations, generate source code and configuration files, or produce system documentation.

3 Modeling languages for large-scale embedded systems

A major portion of our research is motivated toward providing model-based design and fault management tools for large scale HEP systems [22]. There are several different aspects of these systems ranging from hardware, communication protocols, fault management, data and message types, run control, event logging, and deployment. While each individual aspect of these systems can be modeled, building a fully unified model which captures the system from all of these viewpoints would be an arduous and error-prone process.

In order to extend the best practices of large scale system development to MIC without sacrificing the benefits of designing systems from multiple aspects, one can create a set of modeling languages, each corresponding to a particular aspect of the system. For example, a hardware designer's aspect of an embedded control system may include processors, busses, and sensor and actuator interfaces, while the control systems engineer's aspect of the same system may include signal integrators, delays, and transfer functions.

Merely modeling each aspect of the system, however, is not enough; these aspects need to be meaningfully incorporated through a higher level modeling language. As systems become more interconnected and diverse, the number of aspects from which the system is modeled increases. Clearly, some form of automated tool support is required to assist designers in managing the complexity of such large system models.

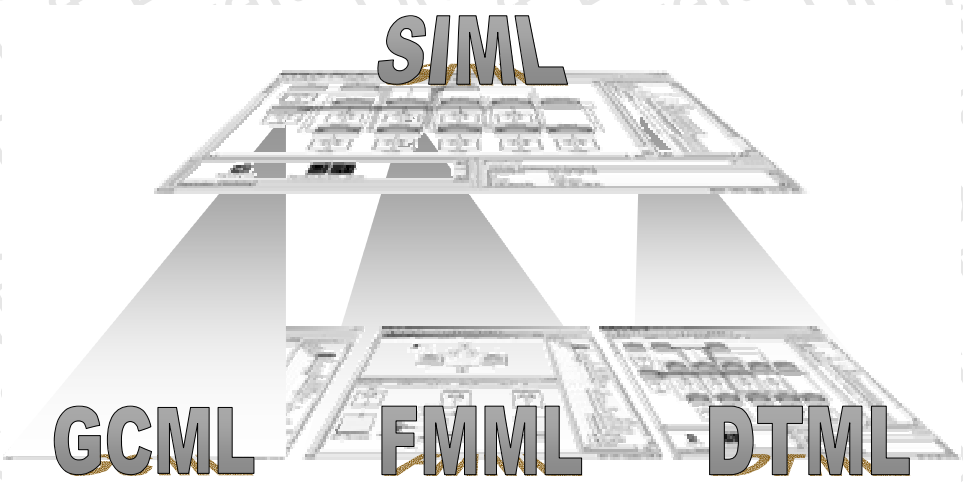

Fig. 1. A high level modeling approach supplemented with multiple narrowly focused modeling languages provides a system-level picture as well as access to refining models expressed in other languages

A set of narrowly focused domain specific modeling languages integrated through a higher level language provides a modeling tool suite capable of specifying numerous relevant aspects of large-scale embedded systems. This concept is illustrated in Fig. 1. The following languages are present in the tool suite:

- System Integration Modeling Language (SIML) – a language used for high level specification of the system
- Fault Mitigation Modeling Language (FMML) – a language for specifying the behavior of fault management components in the target architecture
- Data Type Modeling Language (DTML) – a language for data type modeling and creation of architecture specific message models
- GUI Configuration Modeling Language (GCML) – a language used for rapid layout and design of control, monitoring, diagnostics, and fault injection user interfaces
- Run-Control Modeling Language (RCML) – a language used to describe the behavior of the HEP experiment control components for loading the software, starting and stopping the experiment, etc.

What follows is a brief discussion of the SIML and FMML languages present in the tool suite. Details of the remaining modeling languages in the suite are omitted for brevity, but can be found in [23] and [24].

3.1 Systems Integration Modeling Language

The System Integration Modeling Language (SIML) is a high level systems modeling language that utilizes a loosely specified model of computation for capturing components, component hierarchy, and interactions within the system. SIML allows designers to model from a global view the information and components relevant for a given system configuration. An example SIML model is shown in Fig. 2.

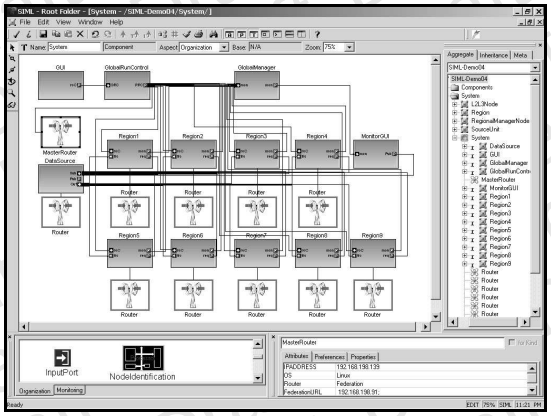

Fig. 2. This SIML model defines a 64-node HEP application configuration with Links to models refinements defined using other modeling languages.

SIML is used to model the general dataflow, as well as the structural and communication layout of the system. Typical objects which are represented in SIML are regions or partitions of the system, software components, dataflow connections, and message routers. SIML also serves as the highest level language through which models of other languages of the systems are accessed.

Figure. 2 shows the model of a prototype HEP application from a global view as expressed in SIML. This model defines a GlobalManager,

several Regions each of which contain a RegionalManager and several LocalNodes. Each LocalNode contains a LocalManager and a HEP application. This view of the model defines the dataflow between the components at global level. Data paths represent point-to-point communication, and Routers represent access points to publish-subscribe middleware services. Models in SIML can be decomposed to show more detail. The decomposition of a SIML component is expressed in one of the many other languages in the tool suite.

To address challenges of representing models in multiple graphical languages we introduce the concept of a Link type which forms a bridge between two graphical modeling languages. Links can be thought of as a *conduit* between layers of models. A Link has attributes to identify the modeling language of the target layer and properties of the linked object, and provide a concise interaction between different modeling languages. Specialized tools facilitate the process of link creation and link navigation, as the creation of a Link is non-trivial and may require an intricate mapping between concepts in different modeling languages. These tools are a benefit to the designer in that they place the onus of Link creation on language and tool developers, thereby reducing the possibility of error.

3.2 Fault Mitigation Modeling Language

There have been efforts towards decentralized algorithms and techniques for managing large scale distributed systems [25] [26]. Centralized or 'expert' methods for controlling complex systems can become unwieldy and impractical as the number of components increase to those required by HEP systems.

Alternative techniques for managing large scale distributed systems usually consist of a hierarchy of managing entities placed strategically in the system whereby control decisions are made using only partial knowledge of the system [27]. Corrective behavior embedded within these entities is automatically activated when fault conditions are observed, and certain actions will be taken in an attempt to mitigate the failure. The benefits of hierarchical behavior in large scale distributed systems include improved robustness and scalability, and the lack of centralized points of failure or attack.

Often the behaviors of fault managing entities are intimately tied to a specific system, and designers will need to create custom fault-mitigation behaviors to suit their own needs. The tool suite provides a general framework by which a designer can model the behavior of fault mitigating entities in a large scale embedded system using concepts specific to failure management within his or her own system architecture.

The Fault-Mitigation Modeling Language (FMML) allows system designers to model the behavioral specification components by providing a general notation refined with additional domain specific features to support fault management activities. This refinement is necessary to facilitate complete generation of source code (object classes, middleware API calls, etc.) to implement the behavior within a software component. The language is general enough, however, that changes to the underlying middleware and execution platform do not necessitate a change to the behavioral model.

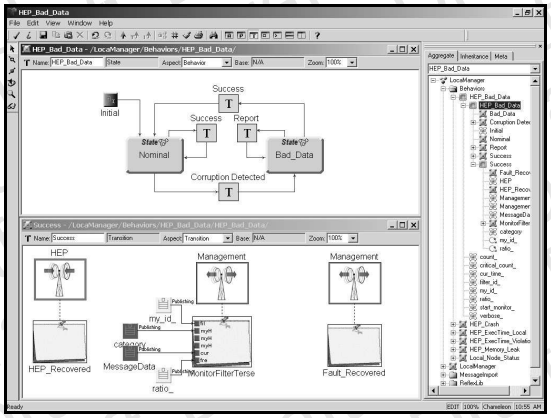

Fig. 3. The Fault-Mitigation Modeling Language (FMML) allows system designers to manage specification of fault-mitigation responses within the system

Features of the FMML can be summarized as follows:

- Blocks represent system states
- Transition models capture the necessary conditions to progress between states
- Transitions are annotated with a Trigger, Guard, and Action expression
- Transitions may contain Message Channels to receive and dispatch Messages
- Transitions may specify an incoming Message events from Message Channels as additional triggering conditions for the transition
- Messages may be produced and consumed from within an Action

FMML utilizes a basic formalism similar to StateCharts [28] for capturing the logical progression of states which define component behavior. As HEP systems place a large emphasis

on messaging and Message-Oriented Middleware (MOM), it is necessary that any fault managing entities have use of these facilities. These messaging systems often have complex interfaces and interactions which are not fully understood by designers of HEP systems. Therefore, state transitions (which in traditional formalism are specified using textual trigger, guard, and action expressions) have been refined to model graphically the concepts and operations common to message passing middleware. Such refinements include the modeling of event channels, message structures, and message creation and distribution operations. The benefit of a modeling language which can represent these facilities allows the full implementation of the behavior (message passing inclusive) to be automatically generated from the model.

Figure 3 shows a custom fault behavior expressed in FMML. The specific behavior being modeled is used for reporting input data stream errors in a HEP application. Nominal and Bad_Data states are defined, and a model defining the transition from Nominal to Fault is shown. The transition model refinement shows the particular messages which are produced and consumed during the transition.

4 Model-integrated frameworks for large-scale embedded systems

The need for dependability in large scale embedded systems is certainly not unique to those in the high energy physics community. Furthermore, much work has been done in the areas of fault recovery, fault tolerance, and autonomicity of a wide range of distributed and embedded systems [29] [30] [31] [32]. Some existing approaches are geared toward design-time techniques; others focus primarily on runtime aspects of the system. Our contribution in the area of large scale embedded systems is the development of fault tolerant software architectures in which models are an integral part, playing an active role in the engineering process and during runtime.

4.1 The Reflex and Healing Architecture

In order to ensure proper operation of an embedded system 1) the computational hardware must be in physical working order, 2) any utilized communication channels must be intact, and 3) the software components must be exhibiting desired behavior. A Reflex and Healing architecture [33] employs a hierarchical network of fault management entities called *reflex engines* whose primary purpose is to implement a fast reflex action when they detect failures within their immediate area of observation. User applications (such as HEP applications) are managed locally, and coordination between managers is limited to adjacent levels of the hierarchy.

4.2 Reflex Engines

Customized software processes are used to perform monitoring, diagnostics, and implementation of failure mitigating actions in the Reflex and Healing Architecture. These processes have mechanisms by which they sense and affect their environment and can produce and consume event notifications to and from other processes to exhibit coordinated behavior. These customizable software processes are called *reflex engines.*

Definition 1. *A reflex engine is a software-based failure management process whose behavior can be configured by an external source. A reflex engine* e_r *can be defined as a quad-tuple*

$$e_r = \langle Q, Z_i, Z_o, R_c \rangle$$

where

Q: *The set of all possible states of* e_r

Z_i: *A set of inputs accepted by* e_r

Zo: *A set of all outputs produced by* e_r

R_c: *The current set of reflex actions performed by* e_r, *where* $R_c \subset R \subseteq Q \times Z$

given

R: *The complete set of possible reflex actions which can be performed by* e_r.

Reflex engines can be organized hierarchically to form a Reflex and Healing network. This network has tree structure to define hierarchical level of a reflex engine. Each reflex engine is responsible for mitigating failures which occur in descendant portions of the tree. This structure defines the various management levels within the hierarchy. A *global* manager resides at the top node of the tree structure, *local* managers reside at the leaf nodes of the tree structure, and *mid-tier* or *regional* managers reside at all other nodes. The specific structural specification of this hierarchy (number of management levels, number of branches of each node, etc.) is defined within the SIML model, and the behavioral specification of each reflex engine is defined using the FMML modeling language.

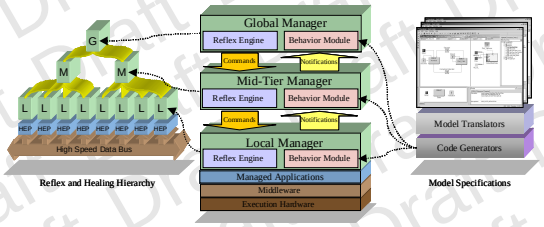

Fig. 4. A set of reflex engines whose behavior are synthesized from FMML models help form the Reflex and Healing architecture

Each reflex engine has a modular behavioral component which is loaded at runtime. Source code to implement each reflex engine's behavioral module is automatically generated during the FMML model interpretation process. Software to instantiate the Reflex and Healing network and load each reflex engine's behavior is generated during the SIML model interpretation process (see fig. 4).

4.3 Scalability

A hierarchy of reflex engines can be used to monitor a vast number of nodes given that peer-level communication between reflex engines is strictly forbidden. That is, any reflex engine will never communicate with reflex engines other than its governor or direct subordinates.

For example, consider a cluster of commodity computers used to implement a Reflex and Healing hierarchy consisting of one global manager, two levels of mid-tier management, and one level of local management. Also consider that each global or mid-tier manager is directly responsible for 20 subordinates, and that all managers reside on unique machines. This architecture would support up to 8000 instances of a managed application with ~5% redundancy overhead (8421 total nodes with 8000 nodes utilized for target applications and 421 nodes utilized for fault management).

Designing increasingly larger Reflex and Healing networks is simply a matter of creating a more elaborate SIML model of the system, and modeling the behavior of additional reflex engines to implement the management hierarchy. Simple copy and paste operations are largely used to perform this task. For example, one can double the size of a Reflex and Healing network model by simply making a duplicate copy of the existing network and creating a new global node to manage the two sub-trees.

This technique has been demonstrated to create the Reflex and Healing network described in Chapter 5 scaling a four node configuration to an 8-, 16-, 32-, then 64-node configuration. This process was completed within a matter of minutes. Due to the extensive use of model-based tools, all artifacts necessary to implement and deploy each of these system configurations were automatically generated from the models.

5 Case Study: The RTES 2004 capstone demonstration system

The Real-Time Embedded Systems (RTES) group is a collaboration of physicists, electrical engineers, and computer scientists from Fermi National Accelerator Lab, Vanderbilt University, University of Illinois Urbana-Champaign, University of Pittsburg, and Syracuse University. The RTES group was formed to address design, integration and fault tolerant issues associated with large-scale embedded systems for upcoming HEP experiments.

To demonstrate new tools and techniques, the RTES group formed an initiative in 2003 to build and test a capstone demonstration system that embodies all the areas of our research regarding the design of large-scale embedded systems. This case study is of our second generation of tools which were demonstrated at the Second Workshop on High Performance, Fault Adaptive, Large Scale Embedded Real-Time Systems (FALSE-II) co-located with the 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2005).

A baseline prototype HEP data processing system was created as a foundation to demonstrate the effectiveness of the tools, and a number of failure scenarios were created. The system was then tested with respect to its ability to correctly survive each fault scenario. Faults which may occur in the system can be classified as being in one of the following groups:

- Intermittent faults: Failures which are not lasting in time or sporadic
- Persistent faults: failures which will persist if no controlled intervention is performed (persistent periodic faults are classified as persistent faults)

Examples of intermittent faults include:

- Spurious data from the physical sensors, the nature of which can be classified in stochastic measures but is considered temporal in nature due to the uncertainty of the periodicity at which failures of this type occur.
- Communication channel corruption. Temporal errors caused by heat, radiation, or faulty cabling may corrupt a data channel.

- A data-driven process whose output depends on integrity of input data may exhibit temporal faulty behavior if corrupted data occurs at the input.
- Processor hard resets (the boot sequencer may recover and restore state to the processes, resulting in a temporal perturbation of system state)

Examples of persistent faults include:

- Processing hardware failure (corruption of memory block or disk sectors)
- Task or process failure due to programmatic error
- Communication link failure (bad I/O ports, broken cables, etc.)

This case study details some experiments toward applying a specific behavior to mitigate faults in a selection of predefined failure scenarios.

5.1 RTES Demonstration Test-bed

A test-bed was created in an effort to faithfully recreate a (small-scale) 64 node HEP Level 2/3 software and hardware architecture. Level 2/3 is a HEP designation specifying that this system implements the second and third stages of online data filtering. Timing deadlines are somewhat relaxed for Level 2/3 systems, and the filtering algorithms run much longer and are much more precise than of those in HEP Level 1 systems. Figure 5 shows an overall view of the system. The nodes run a distribution of Scientific Linux packaged and maintained at Fermilab.

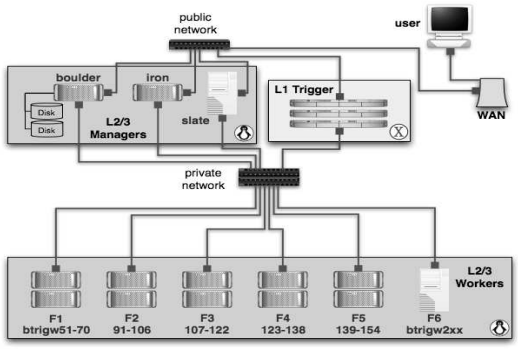

Fig. 5. A diagram of the prototype HEP data processing system for evaluating RTES tools and technologies (courtesy of H. Cheung, Fermilab)

This HEP application performs data distribution and filtering within a system consisting of a data source, 64 dual processor worker nodes, and a command and control user interface. Two instances of the HEP filtering application are present on each node. A Reflex and Healing hierarchy was placed in the system using a global manager, nine (9) regional mangers, and six (6) local managers per region. Faults are introduced into the system through a fault injection interface on the command and control GUI.

Seven (7) failure scenarios were constructed as being representative of typical fault conditions known to occur in HEP data processing systems. The failure scenarios are the following:

1) General application crash (HEP or other, with or without error)
2) Temporal corruption of the HEP application's input data stream
3) HEP application infinite loop (halt with no response)
4) HEP application exponential slowdown
5) HEP application processing time violation
6) HEP application memory usage violation
7) Aggregate processing time violations across a region

The following sections detail the process of creating and testing custom behaviors which attempt to mitigate faults in two (2) of the above scenarios. Each behavior is described in more detail below. It is important to note that contribution of this work is not necessarily in custom failure mitigation behaviors that were defined, but rather in the tools, techniques, and infrastructure that allow these custom behaviors to be easily designed and integrated into a large scale embedded system.

5.2 Experiment 1: Temporal data stream corruption

Often times a temporal corruption can occur which leads to a crash of the key physics data filtering application. These failures are common in HEP experiments, as the detection mechanisms are extremely sensitive to environmental conditions not under direct control of the user. However, it is extremely important to physicists running an experiment that the occurrence of such an error be detected and recorded, so that the portion of the data stream which went unfiltered may be marked as such. The filtering algorithms used in HEP experiments are generally provided with an ability to detect a corrupted data stream. However, the occurrence of such an error should be recorded at the local fault manager level as opposed to the filtering level. This allows a higher-level manager to detect trends which may be present regarding data stream corruption that are not detectable by a single instance of a filter algorithm.

The following scenario was created to test the system's ability to detect and record such an event.

A reflex engine in the Nominal state will transition to the Bad_Data state when it detects and event from the HEP application that the input stream data is corrupt. Using FMML, a model of the desired behavior was created and placed in a reflex engine at the local level of the Reflex and Healing network.

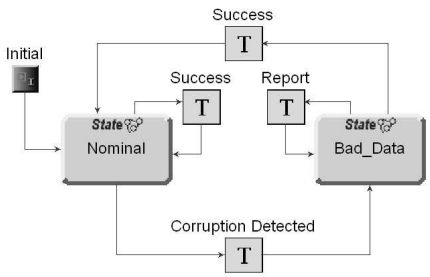

Fig. 6. An FMML model for detecting and reporting errors in the target physics application

To evaluate this behavior, the 64-node HEP system was instantiated and the Data Source was directed to produce corrupt data packets at random intervals. The reflex engines were instructed to log all of their events regarding fault management. Figure 6 shows a model of the behavior as created in FMML.

Table 1. Event legend for Experiment 1

LM1	Received processing time report from HEP application
LM2	Detected HEP application encountered corrupt input data
LM3	Notification to Regional Manager of fault detection

Table 1 details each class of events raised and Fig. 7 shows the event timeline during the handling of this fault scenario. Recall that two (2) instances of the HEP application are running on each node.

Clearly, model based techniques are effective for specifying a simple behavior to handle this fault scenario. One might argue, though, that this fault scenario could easily be handled with minimal risk by writing a small number of lines of code. However, scenarios such as the detection of localized failures across an entire region of the hierarchy are not as simple to implement, and require modeling of the interaction between management levels in the Reflex and Healing architecture. The next experiment illustrates how the tools facilitate the design and deployment of a more challenging hierarchical behavior.

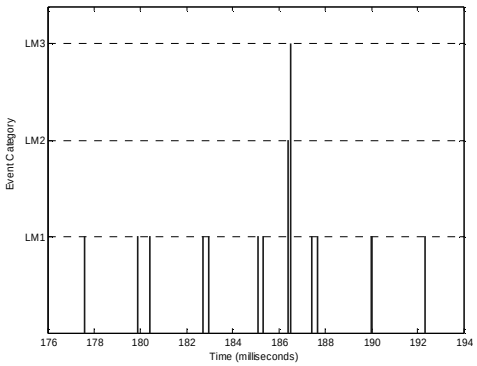

Fig. 7. Events raised during the successful mitigation of a fault scenario

5.3 Experiment 2: Hierarchical mitigation of regional HEP timing violations

Another typical occurrence in HEP Level 2/3 data systems is that the HEP application runs too long, causing a decrease in overall system throughput. This may be caused by a change in the characteristics of the incoming data. HEP applications employ algorithms whose execution times are data dependent. Therefore, as conditions in the particle accelerator change (e.g. a higher density of particle collisions is occurring) the behavior of the filtering applications will change. If this condition is detected, physicists may want to reconfigure certain parameters of the HEP application such that they have shorter execution times as not to overflow the system buffers. Recall that HEP experiments run at a constant physical periodicity, so slowing down the data acquisition is not considered a reasonable option.

Implementing the behavior to detect such a condition is not as trivial. Local Managers have only a small view of the entire system; they do not know the state of other components in the system. Regional managers have no knowledge of local HEP applications or other regions of the system, nor do they have the authority to initiate a global HEP application reconfiguration. Therefore, some degree of coordination must occur between local, regional, and global management levels in order to handle this type of scenario. Although this experiment details only the reflex action performed to handle this fault scenario, once an action is taken, the system state is re-evaluated and an iterative process of optimization and refinement can begin which serves to heal the system.

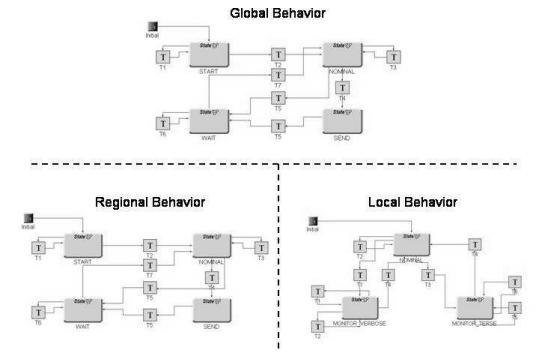

Fig. 8. A set of coordinated FMML models for detecting and reporting long physics application processing times using three levels of hierarchy

Local Managers send periodic notification regarding the processing time of the HEP data filtering application to their governing manager. When a regional manager detects that the average HEP processing time reported by each Local Managers exceeds a predetermined threshold, it requests a reconfiguration of the HEP application from the Global Manager. If the Global Manager determines a reconfiguration is in order, it will issue a command to each of its subordinate Regional Managers to perform the reconfiguration. Each Regional Manager then issues a command to each of its Local Manager to perform the actual HEP filter reconfiguration. Notification of the reconfiguration eventually propagates up the hierarchy and the scenario is considered complete.

Table 2. Event Legend for Experiment 2

LM1	Notification of average HEP application processing times
LM2	Detection of Regional command to reconfigure the HEP application
LM3	Notification that HEP application has been successfully reconfigured
RM1	Detection of slow regional processing times
RM2	Notification to request Global HEP reconfiguration
RM3	Detection of Global command to reconfigure the region
RM4	Command to initiate Local HEP reconfiguration
RM5	Detection of first local reconfiguration
RM6	Detection of last local reconfiguration
GM1	Detection of regional reconfiguration request
GM2	Command to initiate HEP reconfiguration

Table 2 provides an explanation of each event (for brevity, the final notification event propagations are omitted) and the timeline in Fig. 9 shows the sequence of events raised during the handling of this fault scenario.

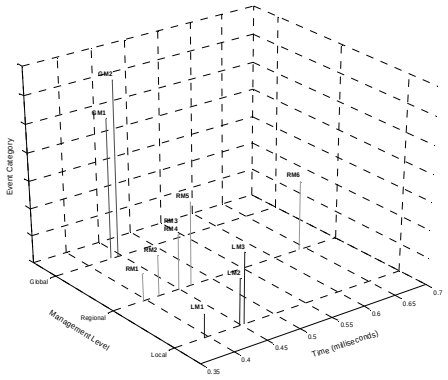

Fig. 9. Events raised during the successful mitigation of a fault scenario in Experiment 2

6 Conclusions and Future Work

This work shows that by combining scalable software architectures such as the Reflex and Healing Architecture with a model-integrated approach, one can more easily design large scale embedded systems which exhibit reasonable behavior in the presence of intermittent and persistent faults. By using a Reflex and Healing architecture, this behavior can be achieved with a minimum of redundancy and operator intervention.

A model-integrated approach towards designing high-performance fault-tolerant large-scale systems is certainly desirable for system integration, behavior modeling, and complexity management. This work has furthered the development of scalable modeling languages and fault tolerant architectures, and was demonstrated to show benefit. The tools allow a designer to model custom behaviors of the components in a fault management hierarchy, and provide automated integration those behaviors into an existing architecture. This reduces the complexity and risk associated with the design and evolution of large-scale systems. An added benefit of our work is that the supporting tools and techniques are general, such that they may be reused in other applications not specific to HEP systems.

Future work includes the application of these technologies to other HEP experiments and larger systems. An initiative has begun to develop a ~500 node HEP data processing system using the Vampire Cluster at Vanderbilt University's Advanced Computing Center for Research and Education [34]. Research is also continuing in the area of model replicators to provide modeling

scalability and further alleviate the designer from the complexities of creating such large models.

Many of the tools used in this research effort are freely available and may be downloaded from the following sites:

Generic Modeling Environment (GME) – http://www.escherinstitute.org/Tools/GME.asp
UDM Linux tool suite – http://www.escherinstitute.org/Downloads/Downloads.asp
Scientific Linux Fermi – http://www.oss.fnal.gov/projects/fermilinux/lts304/index.html

References

1. J. Gutleber, et. al, "Clustered Data Acquisition for the CMS experiment", International Conference on Computing in High Energy and Nuclear Physics (CHEP 2001), Beijing, China, September 3-7, 2001.
2. Kwan S., "The BTeV Pixel Detector and Trigger System", FERMILAB-Conf-02/313-E, December 2002.
3. Buttler J.N., et. al, "Fault Tolerant Issues in the BTeV Trigger", FERMILAB-Conf-01/427, December 2002.
4. Robert France, Bernhard Rumpe, In search of effective design abstractions, Software and Systems Modeling, Volume 3, Issue 1, Mar 2004, Pages 1 - 3
5. Ivar Jacobson, Use cases – Yesterday, today, and tomorrow, Software and Systems Modeling, Volume 3, Issue 3, Aug 2004, Pages 210 - 220
6. Marian Bozdoc, "The History of CAD", MB Solutions web publication, Auckland, NZ 2000-2004, url: http://mbinfo.mbdesign.net/CAD-History.htm
7. Sztipanovits, J., Karsai, G.: "Model-Integrated Computing", IEEE Computer, pp. 110-112, April, 1997.
8. Abdelwahed S., W. M. Wonham, "Interacting DES: Modeling and Analysis", IEEE International Conference on Systems, Man & Cybernetics, Washington, D.C., October 2003.
9. Ledeczi A., Davis J., Neema S., Agrawal A.: Modeling Methodology for Integrated Simulation of Embedded Systems, ACM Transactions on Modeling and Computer Simulation, vol. 13, 1, pp. 82-103, January, 2003.
10. Gray J., Sztipanovits J., Schmidt D., Bapty T., Neema S., Gokhale A, "Two-Level Aspect Weaving To Support Evolution In Model-Driven Software", *Aspect-Oriented Software Development*, Addison Wesley, 2004, pp. 681-705, August, 2004.
11. Ledeczi A., Maroti M., Bakay A., Karsai G., Garrett J., Thomason IV C., Nordstrom G., Sprinkle J., Volgyesi P, "The Generic Modeling Environment", Workshop on Intelligent Signal Processing, accepted, Budapest, Hungary, May 2001.
12. The Generic Modeling Environment at the ESCHER Research Institute, url: http://www.escherinstitute.org/Tools/GME.asp
13. Karsai G., Agrawal A., Shi F, Sprinkle, J., "On the Use of Graph Transformations for the Formal Specification of Model Interpreters", *Journal of Universal Computer Science*, Volume 9, Issue 11, pp. 1296-1321, November 2003.
14. Richard Soley et. al, "Model Driven Architecture" url: http://www.omg.org/mda/
15. Emerson M., Sztipanovits J., Bapty T, "A MOF-Based Metamodeling Environment", Journal of Universal Computer Science, 10, 10, pp. 1357-1382, October 9, 2004.
16. Jack Greenfield, Keith Short, *Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools*, John Wiley and Sons, August 2004.
17. Nordstrom G., "Metamodeling – Rapid Design and Evolution of Domain-Specific Modeling Environments", Proceedings of the IEEE ECBS '99 Conference, 1999.
18. S. Nordstrom, S. Shetty, Kumar Guarav Chhokra, Jonathan Sprinkle, Brandon Eames, Akos Ledeczi, "ANEMIC: Automatic Interface Enabler for Model Integrated Computing", *Lecture Notes in Computer Science*, vol. 2830, pp. 138--150, November, 2003.
19. G. Karsai, A. Agrawal, "Graph Transformations in OMG's Model-Driven Architecture", Applications of Graph Transformations with Industrial Relevance, Charlottesville, VA, USA, Springer LNCS, 2003.
20. Jürjens J, "Formal Semantics for Interacting UML subsystems", Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002), Twente, March 20-22, 2002.
21. Harel D., Gery E, "Executable Object Modeling with Statecharts", *IEEE Computer*, vol. 30, no. 7, pp. 31-42, July 1997.
22. S.Neema, Ted Bapty, S.Shetty, S.Nordstrom, "Developing Autonomic Fault Mitigation Systems", Journal of Engineering Applications of Artificial Intelligence Special Issue on Autonomic Computing and Grids, Elsevier, 2004.
23. Shetty S., Nordstrom S., Ahuja S., Yao D., Bapty T., Neema S., "Systems Integration of Autonomic Large Scale Systems Using Multiple Domain Specific Modeling Languages", 12th IEEE International Conference on ECBS ,Engineering of Autonomic Systems, 0-7695-2306-0/05 pp481, Greenbelt, MD , USA, April 2005
24. Ahuja S., Yao D., Neema S., Bapty T., Shetty S., Nordstrom S, "Dynamically Reconfigurable Monitoring in Large Scale Real-Time Embedded Systems", IEEE SoutheastCon, pp. 327, CD-Rom, Fort Lauderdale, Florida, April 8, 2005.
25. D. Daly, D. D. Deavours, J. M. Doyle, P. G. Webster, and W. H. Sanders., "Möbius: An Extensible Tool for Performance and Dependability Modeling", *Lecture Notes in Computer Science* No. 1786, pp. 332-336., B. R. Haverkort, H. C.

Bohnenkamp, and C. U. Smith (Eds.), Berlin, Springer, 2000.
26. Mitchel Resnick, “Decentralized Modeling and Decentralized Thinking”, *Modeling and Simulation in Science and Mathematics Education* (pp. 114-137), edited by W. Feurzeig and N. Roberts, Springer, New York. 1999.
27. D. Garlan, S. Cheng, and B. Schmerl, “Increasing System Dependability through Architecture-based Self-repair”, Architecting Dependable Systems, R. de Lemos, C. Gacek, A. Romanovsky (Eds), Springer-Verlag, 2003.
28. D. Harel, "Statecharts: A visual Formalism for complex systems", The Science of Computer Programming, vol. 8, pp.231-274, 1987.
29. Broadwell, P., N. Sastry and J. Traupman., “FIG: A Prototype Tool for Online Verification of Recovery Mechanisms”, Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN), New York, NY, June 2002.
30. D. Garlan, B. Schmerl, “Model-Based Adaptation for Self-Healing Systems”, Workshop on Self-healing systems (WOSS), Proceedings of the first workshop on Self-healing systems, Charleston, South Carolina, 2002.
31. R. Sterritt, “Autonomic Computing”, *Innovations in Systems and Software Engineering, A NASA Journal*, vol. 1, No.1, ISSN-1614-5046, Springer, April 2005.
32. J. Kephart and D. Chess, The Vision of Autonomic Computing. *IEEE Computer*, 2003 0018-9162/03
33. S. Nordstrom, S. Shetty, S. Neema, and T. Bapty, "Modeling Reflex-Healing Autonomy for Large Scale Embedded Systems", *IEEE Transactions on Systems, Man, and Cybernetics*, Special Issue on Autonomic Computing (accepted for publication), 2004.
34. ACCRE: Advanced Computing Center for Research and Education, Vanderbilt University, USA, url: http://www.accre.vanderbilt.edu/accre/

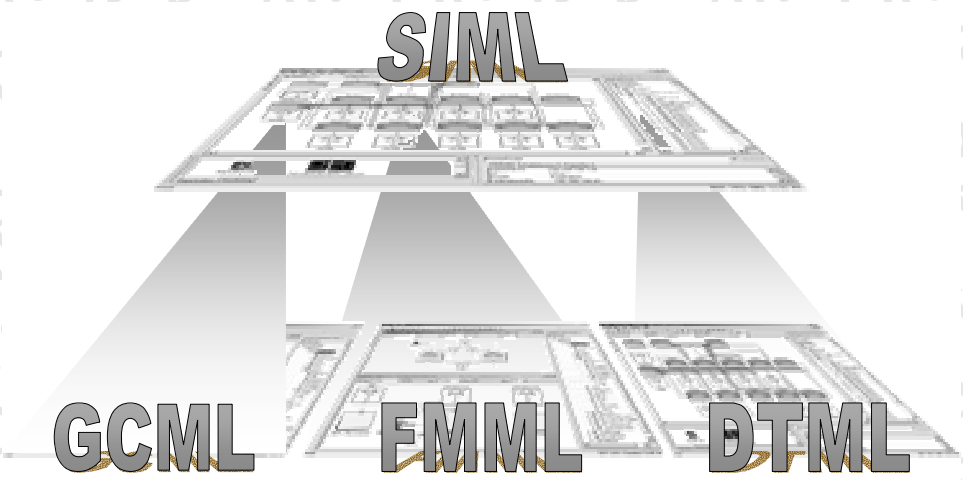

Fig. 10. A high level modeling approach supplemented with narrowly focused modeling languages provides a system-level picture as well as access to refining models expressed in other languages

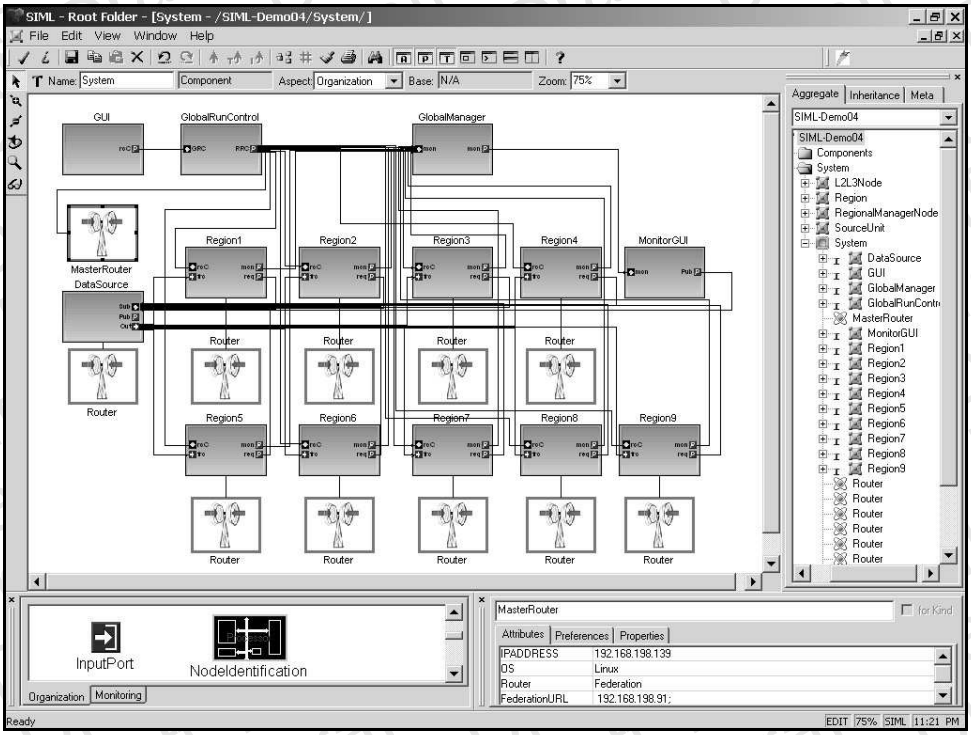

Fig. 11. This SIML model defines a 64-node HEP application configuration with Links to models refinements defined using other modeling languages.

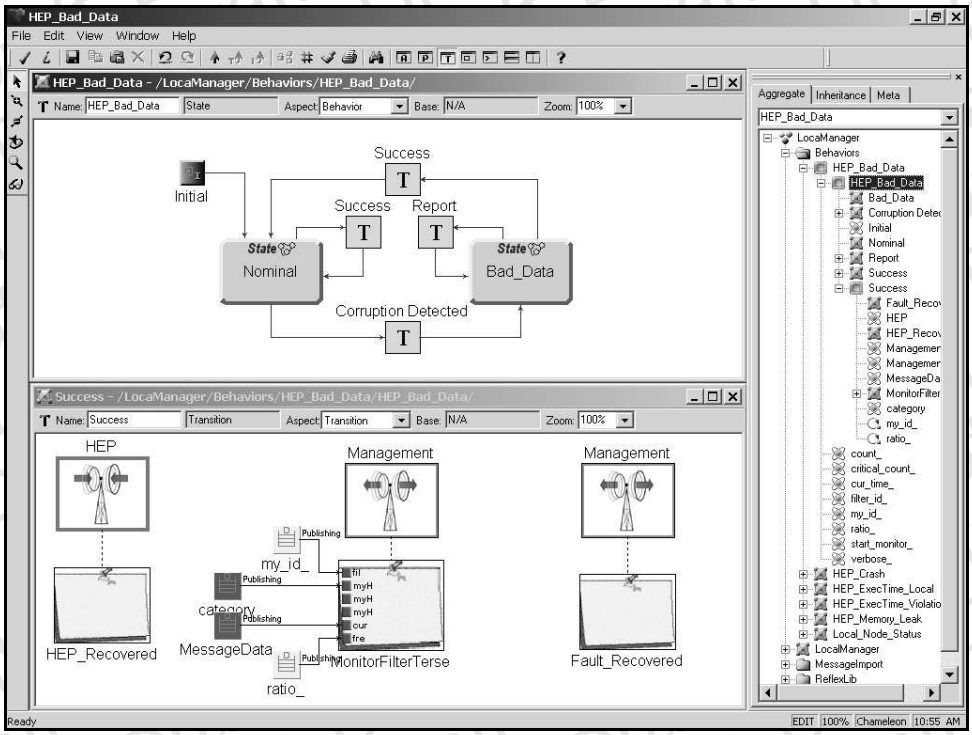

Fig. 12. The Fault-Mitigation Modeling Language (FMML) allows system designers to manage specification of fault-mitigation responses within the system

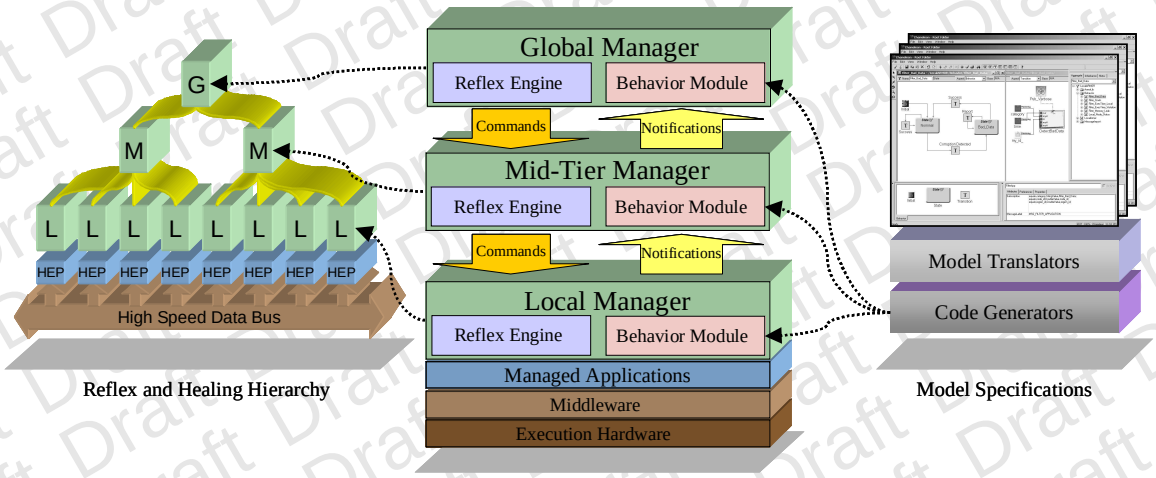

Fig. 13. A set of reflex engines whose behavior are synthesized from FMML models help form the Reflex and Healing architecture

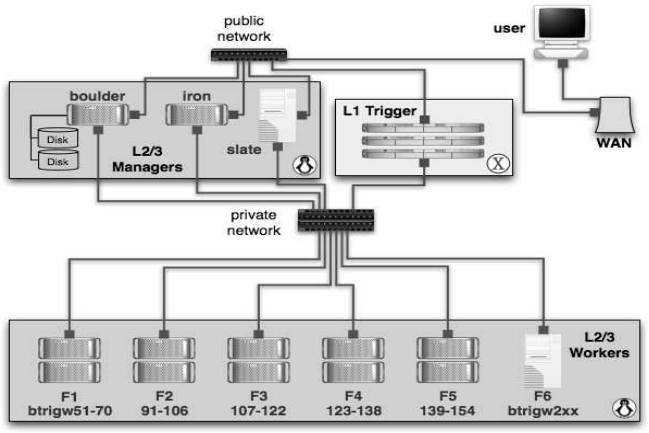

Fig. 14. A diagram of the prototype HEP data processing system for evaluating RTES tools and technologies (diagram courtesy of H. Cheung, Fermilab)

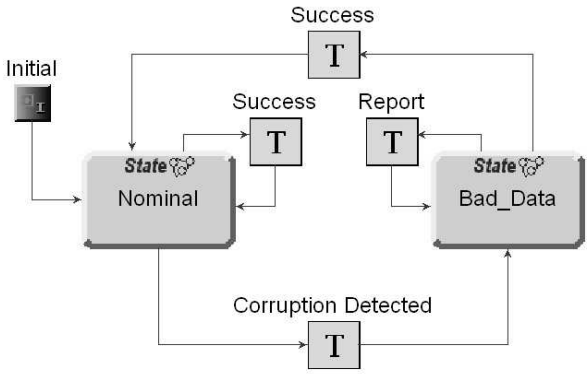

Fig. 15. An FMML model for detecting and reporting errors in the target physics application

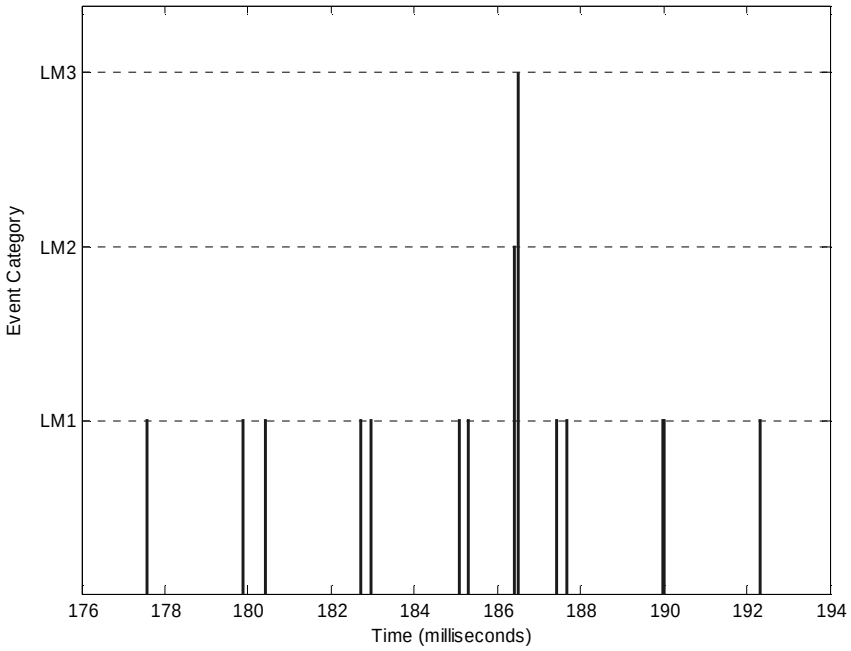

Fig. 16. Events raised during the successful mitigation of a fault scenario

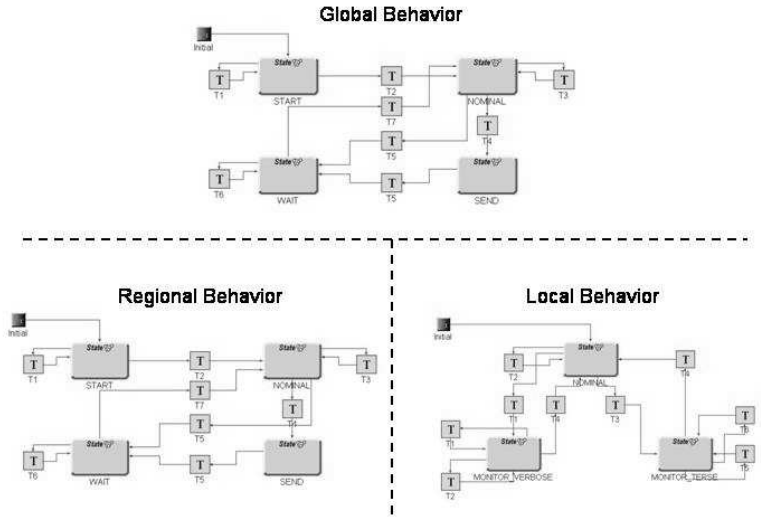

Fig. 17. A set of coordinated FMML models for detecting and reporting long physics application processing times using three levels of hierarchy

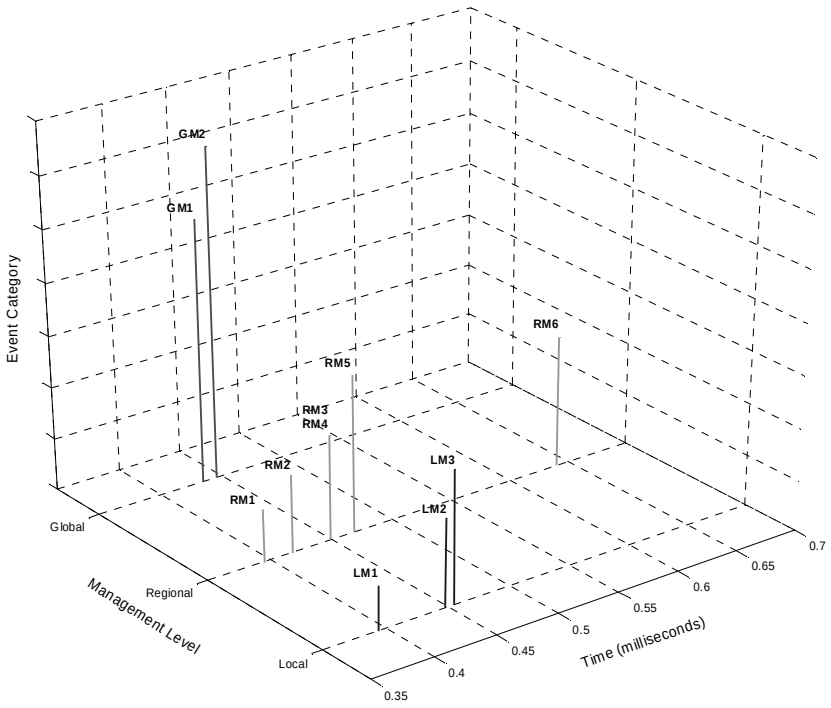

Fig. 18. Events raised during the successful mitigation of a fault scenario in Experiment 2