TOWARDS DEVELOPING TOOLS AND TECHNOLOGIES FOR MODELING FAULTS IN LARGE SCALE, REAL TIME, REACTIVE EMBEDDED SYSTEMS

By

Shweta Shetty

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

May, 2004

Nashville, Tennessee

Approved: Date:

_____ _____

_____ _____

Dedicated to,

My parents

ACKNOWLEDGEMENTS

This research was sponsored by the National Science Foundation in conjunction with Fermi National Accelerator Laboratories, under the BTeV Project, and in association with RTES, the Real-time Embedded Systems Group. This work has been performed under NSF grant #ACI-0121658.

To start with I would like to thank Dr. Theodore Bapty my graduate advisor for his guidance and time, whenever I needed either. His advice has been most valuable throughout my academic period here in ISIS. I am greatly indebted to Dr .Sandeep Neema for giving me a chance to work in the BTeV project and his constant support and technical guidance were instrumental in keeping me motivated without loosing sight of the goal.

I cannot overemphasize my family's contribution in getting me where I am today. I am grateful for every little word of advice, encouragement and concern they have spoken for years.

Finally I would like to thank all my team members and also the folks who are actively involved in this project for being there for me whenever I needed their help.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

LIST OF ABBREVIATIONS

BTeV -B physics at the TeVatron.

FNAL -Fermi National Accelerator Laboratories .

RTES- Real-Time Embedded Systems

DSP –Digital Signal Processing

FSM - Finite State Machine.

CFSM - Codesign Finite State Machine.

HPN - Hierarchical Petri Nets.

SDF - Synchronous Data Flow.

ADF - Asynchronous Data Flow.

COSYMA - COSYnthesis of eMbedded Architectures.

DFG - Data Flow Graph.

MIC - Model Integrated Computing.

GME - Generic Modeling Environment.

OCL - Object Constraint Language.

COM - Component Object Model.

API - Application Programming Interface.

BON - Builder Object Network.

XML - eXtensible Markup Language.

UML –Unified Modeling Language

TMR – Triple Modular Redundant

CHAPTER I

INTRODUCTION

Research in fault-tolerant distributed computing aims at making distributed systems more reliable by handling faults in complex computing environments. Moreover the increasing dependence on well designed and well-functioning computer systems has led to an increasing demand for tools for building dependable systems, systems with quantifiable reliability properties.

Arguably dependability is one of the most crucial design considerations for a majority of embedded systems. Often in this class of systems the term "the system crashed" is not just a metaphor, it may have real and physical consequences. Even with best practices and careful design decisions, the growing scale and complexity of embedded systems makes it impossible to perform an exhaustive coverage that assuredly scrubs out each potential failure source. Potential failures could arise from hardware, or software, or the physical processes with which the embedded system has to interact, or a combination of any of these. Furthermore, extended operational life-cycle of the embedded system introduces the fatigue factor also as a possible source of failure. Given the myriad potential fault sources – some of which may not even be identifiable, and the inability to prevent those failures from occurring, the only possible mechanism of introducing dependability in a system is by means of ***fault tolerance*** or ***fault mitigation***.

Traditional Fault Tolerance Strategies

Fault Tolerance is the ability of a system to perform its function correctly even in the presence of internal faults. The benefits of fault tolerance are usually advertised as improving dependability –the amount of trust that can justifiably be put in a system. Normally, dependability is defined in statistical terminology, stating the probability that the system is functional. The metric provides the expected service at a specific point in time. There is considerable ambiguity in the literature on the meaning of some central terms like fault and failure. Cristian [1991] [1] remarks that "what one person calls failure, a second person might call fault, and the third person might call error". The term fault is usually used to name a defect at the lowest level of abstraction, e.g., a memory cell that always returns a value 0 [Jalote 1994]. A fault may cause an error, which is a category of the system state. An error, in effect, may lead to a failure, meaning that the system deviates from its correctness specification. In other words, a fault is the root cause of a failure. Thus, an error is merely the symptom of a fault. A fault may not necessarily result in an error, or the same fault may result in multiple errors. Similarly, a single error may lead to multiple failures. These basic concepts are illustrated using Unified Modeling Language (UML) class diagram in Figure 1.

Traditionally faults were handled by describing the resulting behavior of the system and grouped into a hierarchic structure of fault classes or fault models [1]. Well known models are the *crash* failure models (in which processors simply stop executing at a specific point in time), *fail-stop* (in which a processor crashes, but this may be easily detected by its neighbors), or *Byzantine* (in which processors may behave in arbitrary ways).

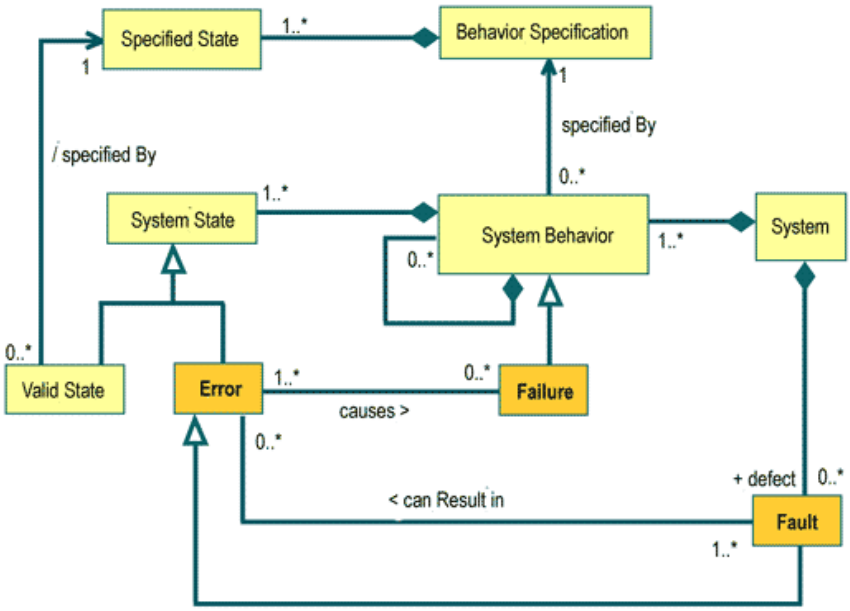


Figure 1 Classification of Error, Faults and Failure

The fault tolerant research community has been able to produce some useful techniques – such as warm / cold / hot replication, active or passive replication, check-pointing, heartbeat, n-way redundancy with or without voting to name a few. At the heart of all the fault tolerance techniques is some form of masking redundancy. This means that components that are prone to defects are replicated in such a way that if a component fails, one or more of the non-failed replicas will continue to provide service with no appreciable disruption.

Replication Checks: In this case, multiple replicas of a component perform the same service simultaneously. The outputs of the replicas are compared and any discrepancy is an indication of an error in one or more components. A particular form of this that is used in hardware is called triple-modular redundancy (TMR), in which the output of three independent components is compared, and output of the majority of the components is

actually passed on. In software, this can be achieved by providing multiple independently developed realizations of the same component. This is called N-version programming. This method of course works well for small systems where hardware is cheap and memory is plentiful. In high performance parallel systems, fault tolerance using conventional redundancy methods can be extremely costly, as system size and complexity increase.

Problems With The Traditional Approach

The traditional approach has been around for a while and is still in practice. However, this approach is not suitable for many cost-sensitive or large scale systems. It is well understood that the software development for real time embedded systems is a difficult undertaking. In addition to the functional and temporal correctness, reliability is a key design factor. Certain classes of reliable RT systems, such as a high energy physics trigger systems employ very large numbers of processors that must operate consistently over several months. N-mode redundancy is not an option due to scale of the system and cost constraints. A "reasonable behavior" is expected from these systems when the hardware or the software components fail. There is not a single answer to how the system should react when fault occur, rather behavior is dependent on the application or mode of operation. In the class of systems of interest, namely large-scale real-time embedded systems, often different levels of failures and thereby reduced level of functionality are acceptable under certain circumstances. The overarching requirement is that the failures should not have a cascading effect that some level of functionality of the system should remain. Within this umbrella of non-cascading failures, the specific response to failures is often user-defined and situation dependent. This response could range from a simple

notification on an operator interface, to active load shedding (with due logging of system state), to processor resets/test and hot-spare failovers. Thus, in the domain of very large parallel systems, designers of embedded systems rarely have the luxury to employ such hardware-intensive tactics. In such cases, software plays an important role in achieving fault tolerance, but this can severely complicate implementation. Maintaining the high quality in software-adaptive, fault tolerant embedded systems is only possible with the assistance of advanced software tools. These tools can shift the burden of the design away from human. The tools should take the form of a highly customizable fault-mitigation framework that includes high level design tools and runtime support infrastructure.

Fault Mitigation Design Methodologies

As we mention above, the class of systems of interest for this research is very large scale real time embedded systems which have a need for a customizable fault mitigation methodology. There are two minimum requirements of such a system by the users

1) The system should not fail abruptly

2) Faults should not have a cascading effect.

However, between these two absolutes, there is a diverse range of specifications for responses to different kinds of faults. For example, if for some reason an algorithm residing on a processor terminates without finishing the computation due to either hardware, or software fault, the designer may require the algorithm to be computed in an n-mode redundancy mode or check-pointed for a restart, or even a notification that is duly logged may suffice. Thus, a simple notification mechanism combined with an

algorithm restart is sufficient to meet the dependability specifications in this case. On the other hand, a repeated recurrence of algorithm failure on the same processor might require a fault response in the form of a processor reset followed by diagnostic session. In this case some amount of hardware redundancy is required. This diversity of fault responses demonstrates the need for a highly flexible and customizable fault tolerance framework. This framework should allow the users of the system, to specify fault responses, in an abstract (high-level) yet precise form (i.e. fault response specifications).These specifications could then be used to automatically customize the fault-tolerance infrastructure to suit their application and system requirements. With this motivation we have been developing high-level design tools, and a highly flexible fault-tolerance runtime infrastructure to construct systems – specifically, Fermi Lab's BTeV trigger system –that falls into the class of large scale real time embedded system.

Problem Statement

The aim of this thesis is to develop a model based tool set for software based fault mitigation strategies and to apply these tools within the framework of BTeV. The first step comprises developing a domain-specific modeling environment for fault-adaptive applications implemented on large-scale parallel systems. The next step is to integrate the modeling environment with the existing software infrastructure to create a heterogeneous environment for the design of embedded application. The integrated environment should allow the users to specify modular system design with alternative implementations.

The tools should have the capability to interpret the system models and generate code for various fault mitigation strategies and also it should have the ability to synthesize low-level programming artifacts from the higher-level abstractions, alleviating

the system developer from the burden of constructing low-level artifacts, ensuring consistency with the higher-level abstractions.

This thesis presents the tool rationale, implementation strategy, and a case study of the approach. Chapter II presents a survey of different relevant models of computations (MOCs) and other real time embedded system design tool. Chapter III describes the BTeV trigger system and the modeling paradigm for hardware/software. Chapter IV presents a case study to validate the tools and design philosophies and in Chapter V a detailed explanation of the fault mitigation language and formalisms and the conclusions are drawn in Chapter VI.

CHAPTER II

BACKGROUND AND LITERATURE SURVEY

Models of Computations

The specification process is a fundamental aspect of system design. We advocate using an unambiguous formalism to represent design specification and implementation choices. These formalism are often called the *Model of Computation (MOC)*. There are several models of computation that have been specified and used in embedded systems. A MOC is composed of a description mechanism (syntax) and rules for computation of the behavior given the syntax (semantics). Models of computation can serve different domains or aspects of a specific system. For example, some MOC's are suitable for describing complicated data transfer functions and completely unsuitable for complex control, while others are designed with complex control in mind. A MOC is typically realized (implemented in practice) by a particular language and its semantics. Multiple Models of Computation can be used in any particular system, each describing different aspects of the computation. Here, we survey several MOC's that are appropriate for applications like the BTeV Trigger.

Finite State Machines

State Machine based specification methodologies are an efficient and popular way to describe and analyze event-driven systems. System behavior is represented by a directed graph, consisting of a finite set of conditions, called **states** and paths between the states, called **transitions**. Finite State Machine (FSM) is a well-established

representation. It can be defined as a model of computation consisting of a set of **states**, a **start state**, an **input alphabet**, and a **transition function**, which maps input symbols and current states to a next state. This representation is useful in describing applications that are tightly coupled with the environment. It is also suited for control-dominated and reactive applications. With basic FSM's however, concurrency is not easily captured. Representing concurrent states results in the exponential growth in the number of states with linear increase in degree of concurrency. This problem is also known as the state space explosion problem. Furthermore, small changes to the requirements may result in large changes to the corresponding FSM. In order to overcome various weaknesses of the classical FSM a number of extensions such as hierarchy and concurrency have been developed. A few such variants are discussed [2].

A classic state machine is basically a quintuple of the form:

$$FSM\ M: = (Q, \sum, \Delta, \partial, q_0) \quad (1)$$

where

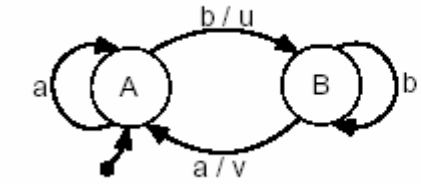
- Q is a finite set of states
- $\sum$ is a set of input events
- Δ is a set of output events
- $\delta: Q \times \sum \rightarrow Q \times \Delta$ is the state transition function
- $q_0 \in Q$ is the initial state

An example of simple state machine is:

$$M = (\{A, B\}, \{a, b\}, \{\varepsilon, u, v\}, \delta_{M, A}) \quad (2)$$

$$\delta_M = \{(A , a) \rightarrow)(A ,\varepsilon), (A ,b) \rightarrow(B ,u),(B ,a) \rightarrow(A ,v),(B ,b) \rightarrow(B ,\varepsilon)\} \quad (3)$$

There are chiefly two options for the representation of a state transition function δ. The graphical depiction on the basis of a state transition diagram as shown in Figure 2 is the preferred way, because the behavior is readily apparent from the diagram. All modern state machine supporting software tools provide a graphical design environment for FSMs. Alternatively, a state transition table (table in Figure 2) can be used to represent the transition function of a particular state machine.



$Q\backslash\Sigma$	a	b
A	(A,ε)	(B,u)
B	(A,v)	(B,ε)

Figure 2 State Transition diagram and State Transition Table

Mealy and Moore Automata are just two of the different ways in which state machines can be interpreted. The major difference in the two methods is in the execution time of the output events. Both architectures split a state transition function into a pure transition function and an additional output function. Therefore, a Mealy as well as a Moore automaton is defined as a sextuple:

$$FSM\ M: = (Q, \sum, \Delta, \delta, \mu \setminus \lambda, q_0) \quad (4)$$

where

- Q is a finite set of states
- $\sum$ is a set of input events
- Δ is a set of output events

- $\delta: Q \times \sum \rightarrow Q$ is the state transition function
- $\mu: Q \rightarrow \Delta$ is the Moore output function
- $\lambda,: Q \times \sum \rightarrow \Delta$ is the Mealy output function
- $q_0 \in Q$ is the initial state

In case of a Mealy automaton, output events are associated with transitions and depend on the current state and current input event. Based on these semantic, the Mealy approach assumes that output actions take no time to execute, because a system is not in a well defined state while outputs are performed. The Moore architecture avoids the so called zero time assumption. Here, output events are associated with states. The system state is always well defined during action execution. Both automata are mathematically equivalent and one always can be transformed into the other. As a rule, a Mealy state machine interpretation requires fewer states, because a Moore automaton must use different states to represent conditions in which different actions are performed

Codesign Finite State Machine (CFSM) is another model based on the FSM. It is intended to describe embedded applications with low algorithmic complexity. Both hardware and software can be depicted using this model of computation. It can be used to partition and implement applications. The basic communication primitive is an event. The behavior of the system is defined as a sequence of events. The events are broadcast and have zero propagation time. This model of computation is used as an intermediate representation that high-level languages can map to [2][3].

Statecharts by Harel [4] is another extension of FSMs that provides three major facilities, namely: hierarchy, concurrency and communication. Statecharts are high-level Finite State Machines having AND and OR states. The AND states primarily achieve

concurrency while the OR states are for representing hierarchy. Communication is based on events that are broadcast instantaneously. This representation is well suited for large and complex reactive systems.

Discrete-Event Systems

Systems having discrete states and driven by events over a period of time are referred to as Discrete-Event Systems [5]. A discrete event system (DES) is a dynamic, asynchronous system, where the state transitions are initiated by events that occur at discrete instants of time. Typical examples of DES are: flexible manufacturing systems, telecommunication networks, traffic control systems, multiprocessor operating systems and logistic systems. These systems are asynchronous in nature and react to discrete events over time. An event is considered instantaneous, that is, the transition and action due to the event occur with no execution time. Although DES lead to a nonlinear description in linear algebra, there exists a subclass of DES for which this model becomes "linear" when we formulate it in the max-plus algebra.

Signals form the primary method of communication between tasks. They consist of a set of events over time. The events are time stamped and are sorted and processed in chronological order. Discrete-Event Systems are backed with formal mathematical description [6] that allows modelers to do formal verification and build deterministic systems. Though these systems are good for real time applications the primary disadvantage is the computational cost of sorting the events globally to maintain the chronology.

Petri Nets

Petri-Nets [7] methods are graphical notations with a solid mathematical foundation. They can be used in a number of different ways to represent computer based systems, especially those concerning concurrency, distribution and non-determinism. The concept was invented by Carl Adam Petri in 1962 and has been proposed for a wide range of applications including performance evaluation, communication protocols, and multiprocessor design. A Petri-Net is described as a 5-tuple,

PN= {P, T, F, W, Mo} where: (5)

P = {p1, p2, p3, …. ,pm }	is a finite set of places
T = {t1, t2, t3, ……..,tm }	is a finite set of transitions
F is a subset of (P x T) U (T x P)	is a set of arcs giving flow relations
W: F $\rightarrow$ {1, 2, 3, …}	is the weight function
Mo: P $\rightarrow$ {0, 1, 2, ….}	is the initial marking

The places hold tokens and a transition can occur if the number of tokens required for the transition is present in the place. A transition removes a specific number of tokens from its source and adds tokens to the destination. A snapshot of places with the number of tokens describes the state of the system.

Petri nets are a promising tool for describing and studying systems that are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic. As a graphical tool, Petri nets can be used as a visual-communication aid similar to flow charts, block diagrams, and networks. In addition, tokens are used in these nets to simulate the dynamic and concurrent activities of systems. As a mathematical tool, it is possible to set up state equations, algebraic equations, and other

mathematical models governing the behavior of systems. The primary feature of Petri Nets is its concurrent and asynchronous nature. Along with concurrency and asynchronicity, there are a number of mathematical analyses that can be performed on Petri Nets. The lack of hierarchy makes Petri Nets difficult to be use for large systems.

Hierarchical Petri Nets (HPNs) [8] have been developed to mitigate the complexity of a flat representation. HPNs are modeled by bipartite directed graphs with inscription on the nodes and edges. There are two types of nodes, transitional nodes that represent activity and places that represent data or the state of the system [2]. This approach extends the Petri Net semantics with hierarchy making it suitable for large and complex systems.

Data Flow Graphs

The classical programming structure of computer-based systems is control dominated. An alternative approach is data-dominated where the control flow is determined by availability of data. These systems have nodes describing computation and edges between nodes denoting a data path. Here the control is based on the availability of data. In other words the scheduling of the computation is tied to the availability of data. If a node has sufficient data available on its incoming edges then it is ready to fire and will use the input data to generate output data. Transfer of data between computational modules is typically done via buffers. This allows the tasks to run independently.

Various flavors of data flow are seen in literature. The two popular and distinct ones are Synchronous Data Flow (SDF) and Asynchronous Data Flow (ADF). In SDF the number of token produced and consumed is fixed and must be known at the system design stage. This requirement allows the SDF to be statically scheduled [9] and

optimized for minimum buffers. The ADF is defined as a data flow graph with unbounded buffers where the computations can produce and consume variable number of tokens. Since the consumption and production of tokens can change at runtime, the ADF cannot be scheduled statically and hence, it has a greater run-time cost. However, it is more flexible than the SDF and can represent the majority of systems. Similarly, in hardware, structural description is widely used. The structural layout of the target DSP hardware is data driven and on a closer look it is very similar to an unbuffered asynchronous data flow. It is defined in a variant of ADF with the buffer length being zero. In other words there is no buffering of data.

Results Of The Survey

Most of the models of computation discussed are suitable for either data-dominated or control-dominated systems. SDF and ADF methods are good for the signal-processing domain. Similarly FSM is useful in representing control-dominated systems. There is no single model well suited for both control and data requirements. Thus, depending upon the target application area, the model of computation needs to be chosen. Composing different models of computation is also worth considering [22][23].

Table 1 Strengths and Weakness of different models.

Model	Strengths	Weaknesses
Finite State Machine	• Good for sequential control • Can be made deterministic • Maps well to hardware and software	No code reuse, application specific.
Discrete Events	• Good for digital hardware • Global time • Can be deterministic	Expensive to implement in software
Synchronous/Reactive Model	• Good for control-intensive systems • Tightly synchronized • Deterministic • Maps well to hardware and software	Computation-intensive systems are over specified.
Dataflow	• Good for signal processing • Loosely synchronized • Deterministic • Maps well to hardware and software	Control-intensive systems are hard to specify.
Communication Sequential Processes	• Models resource sharing well • Partial-order synchronization • Supports naturally non-deterministic interactions	Some systems are over synchronized and difficult to be made deterministic.

Real Time Embedded System Design Tools

Many Computer Aided Software Engineering tools (CASE) are available to assist the developers with the software development process. Since the early days of writing software, there has been an awareness of the need for automated tools to help the software developer. Initially the concentration was on program support tools such as translators, compilers, assemblers, macro processors, and linkers and loaders. However, as computers became more powerful and the software that ran on them grew larger and more complex, the range of support tools began to expand. In particular, the use of interactive time-sharing systems for software development encouraged the development of program editors, debuggers, code analyzers, and program-pretty printers. The first generation of CASE tool developers concentrated to a large extent on the automation of isolated tasks such as document production, version control of source code, and design method support. While successes have been achieved in supporting such specific tasks, the need for these `islands of automation' to be connected has been clearly recognized by many first generation CASE tool users. For example, a typical development scenario requires that designs be closely related to their resultant source code, that they be consistently described in a set of documentation, and that all of these artifacts be under centralized version control. The tools that support the individual tasks of design, coding, documentation, and version control must be integrated if they are to support this kind of scenario effectively.

In fact, such tools are more often used as components in a much more elaborate software development support infrastructure that is available to software engineers. A typical CASE environment consists of a number of CASE tools operating on a common

hardware and software platform. Also note that there are a number of different classes of users of a CASE environment. Some users, such as software developers and managers, wish to make use of CASE tools to support them in developing application systems and monitoring the progress of a project. On the other hand, tool integrators are responsible for ensuring that the tools operate on the software and hardware platform available, and the system administrator's role is to maintain and update the hardware and software platform itself.

Software developers, tool integrators, and system administrators interact with multiple CASE tools and environment components that form the software and hardware platform of the CASE environment. It is these interactions among the different CASE environment components and between users and those components which are the key elements of a CASE environment. In many respects the approach toward the management, control, and support of these interactions distinguishes one CASE environment from another. Hence, we can define a CASE environment by emphasizing the importance of these interactions:

"A CASE environment is a collection of CASE tools and other components together with an integration approach that supports most or all of the interactions that occur among the environment components, and between the users of the environment and the environment itself. "

This section reviews the existing research and tools for co-design for real time embedded systems.

MetaH

MetaH is a language and toolset for developing reliable, real-time multiprocessor avionics system architectures. It is also used for system analysis and integration as illustrated in Figure 3. It is a product of Honeywell and is a research prototype tool.

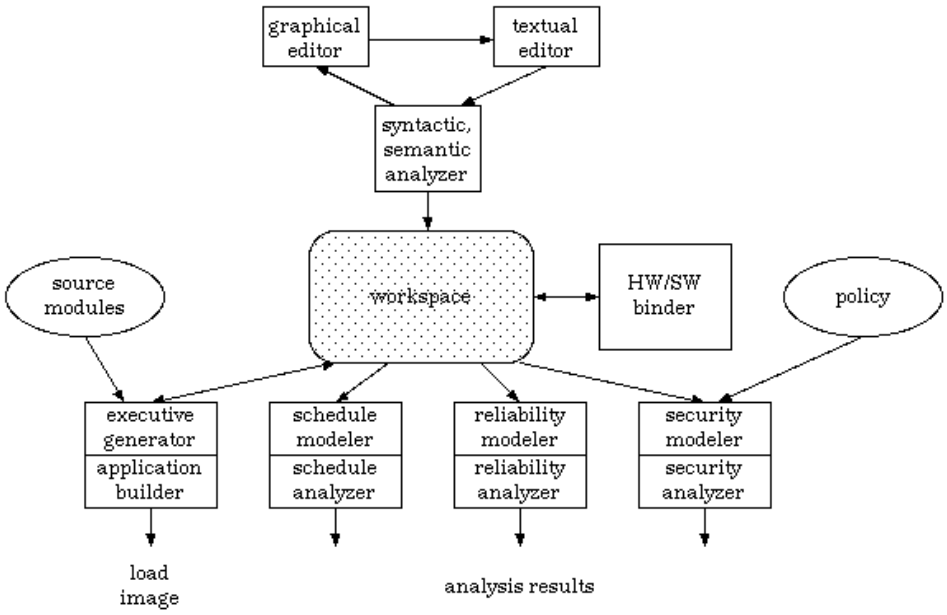


Figure 3 MetaH Architecture

MetaH specifies how software modules developed in a variety of styles are composed together with hardware objects to form complete system architecture. MetaH exhibits elements of real time process and concurrent state machine styles. MetaH specifications allow developers to compose software objects such as subprograms, packages and processes and hardware objects such as memories and processors. Hierarchical specification is supported, where macros and modes hierarchically combine software objects, and systems hierarchically combine hardware objects. MetaH allows

computer system engineers to integrate the source modules for all the various functional subsystems to form the final real-time, fault-tolerant, securely partitioned multi-processor system.

Three major classes of features distinguish MetaH from existing approaches to designing and integrating software using CASE tools and real-time operating systems.

1. MetaH specifications can be used to drive automatic software and system integration. Users do not need to design and hand-code configuration-specific sequences of calls to the various application modules and real-time operating system services. This reduces development effort and reduces defects in the code that integrates the application modules. Because the integration code is automatically generated, it is possible to rapidly reconfigure systems. Finally, static analysis by the tools allows many operations to be preplanned at development time rather than dynamically computed at run-time, which can significantly reduce on-board code size and overhead.
2. MetaH specifications can be subjected to formal analysis. Partial MetaH specifications can be partially analyzed, which supports design trade-off studies beginning very early in the development process. The co-generation of code and analytic models from a common specification provides high assurance that the analysis results accurately predict and bound final implementation behavior. The accuracy of analysis results can be relied upon during design trade-offs, and analysis results can be used in the V&V process.
3. The MetaH architecture specification language includes as part of its definition a discussion of the coding guidelines used for source modules. These guidelines

together with the MetaH language features define a set of common software/software and software/hardware interface mechanisms. Source modules can thus be more independent of the application, hardware and software context in which they are used. MetaH supports increased reuse of source modules. It also allows system architectures to be rapidly reconfigured to adapt to changing hardware and functional requirements without making changes to application source modules.

MetaH allows a specification of system components and connections, and attributes of those components and connections that are relevant to real-time, fault-tolerant, secure partitioning, and multi-processor aspects of an application. The kinds of objects in a MetaH specification can be divided into lower-level objects that describe source code modules (e.g. subprograms, packages) or hardware elements (e.g. memories, processors); and higher-level objects that specify how previously declared objects may be combined to form new objects (modes, macros, systems, applications).The language has both a textual and a graphical syntax, and the tools allow a specification to be viewed and edited interchangeably in either format.

The release toolset includes targets to a portable Ada 95 configuration (the MakeH build scripts use the GNAT toolset), and to single processor 960EXV and single and dual-processor 960CVME configurations (i80960MC processors using the Tartan Ada toolset and protected memory run-time).

PTOLEMY

Ptolemy is a project dedicated to modeling, simulation and design of real-time, embedded applications started in 1990 at University Of California at Berkeley with focus

on component-based design. The philosophy of this project is centered on using different models of computation and developing an environment that allows the mixing of these models of computation to create a heterogeneous application [30].

The primary aim of the Ptolemy project is to build a framework for modeling, designing and development of embedded applications. Figure 4 shows the design management strategy proposed by the Ptolemy project. Design starts with application specification using different models of computation and constraints. Different tasks of the system are evaluated and estimates are drawn. These estimates decide the hardware and software partition of the application. This is followed by hardware and software synthesis and verification. The final stage is the integration and system wide simulation [31].

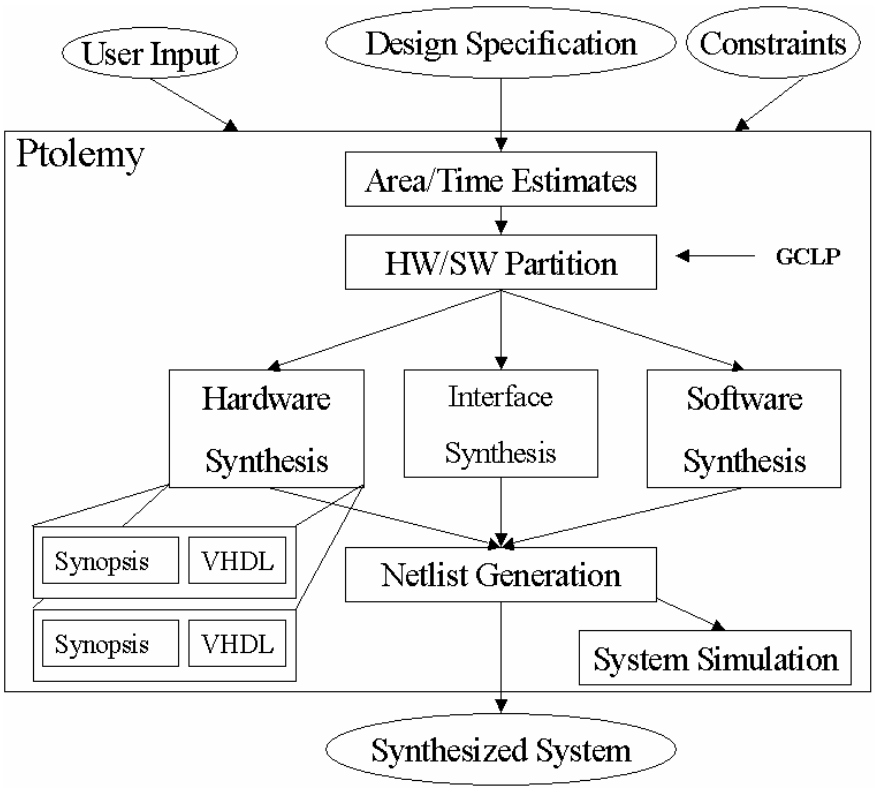


Figure 4 Design Methodology Management using Ptolemy [31]

A Java-based framework called Ptolemy II has been developed that implements the project ideas. The framework has an environment for the simulation and prototyping of heterogeneous systems. It is an object-oriented system allowing interaction between diverse models of computation. The Ptolemy software is extensible and publicly available. It allows experimentation with various models of computation, heterogeneous designs and co-simulation. The primary feature of Ptolemy is the facility to compose various models of computation. Some of the models of computation supported by Ptolemy are hierarchical finite state machine, data flow graphs, discrete-event and synchronous/reactive. After specifying the application using heterogeneous models, the next step is to partition the application. This is done using different partitioning algorithms like GCLP [29]. Ptolemy facilitates mixed mode system simulation and synthesis. Software synthesis is supported for various models of computation along with support for composing these models. Hardware portions of the application are synthesized to VHDL. A register transfer level simulator (THOR) has also been added for simulating hardware applications [30].

Other key features of the project are the representation of modern theories in a block diagram specification, modular approach, a mathematical framework for comparison between models of computation and simulation and scheduling of complex heterogeneous systems [30].

CHAPTER III

BTeV TRIGGER SYSTEM OVERVIEW

At Fermi National Accelerator Laboratory in Chicago, high energy physicists conduct experiments using massive facilities to delve into the basic composition of matter. The objective of the BTeV experiment is to search for charm charge-particle (CP) violations, mixing, and rare decays, as well as take measurements of beauty CP violation and rare decays [9]. The physicists expect to explain the large discrepancy between matter and anti-matter in the universe with such experiments. In these experiments, a particle accelerator called the Tevatron, applies enough energy to accelerate protons and anti-protons up to relativistic speeds. Packets of these particles rotate around a 1 mile diameter ring with particles and anti-particles moving in opposite directions. At specific locations within the ring, called the detector stations, the packets collide where collisions breaking the particles into the basic components of matter, or quarks [10]. Figure 5 shows a view of the equipment used to detect these basic particles.

The accelerator is set up so that the collisions occurs every 135 nanoseconds. The sub-particles, product of the collisions, are measured by a set of planes of detectors, giving a 3-dimensional dataset. Each dataset consists of 200 KB of data. The aggregate raw data rate exceeds 1.5 TB per second. Most of these collisions result in well-known components, and are uninteresting to the physicists. Interesting products of collisions occur remarkably infrequently. In order to get sufficient data (on the order of 100 events), the experiments must run days or even months at a time.

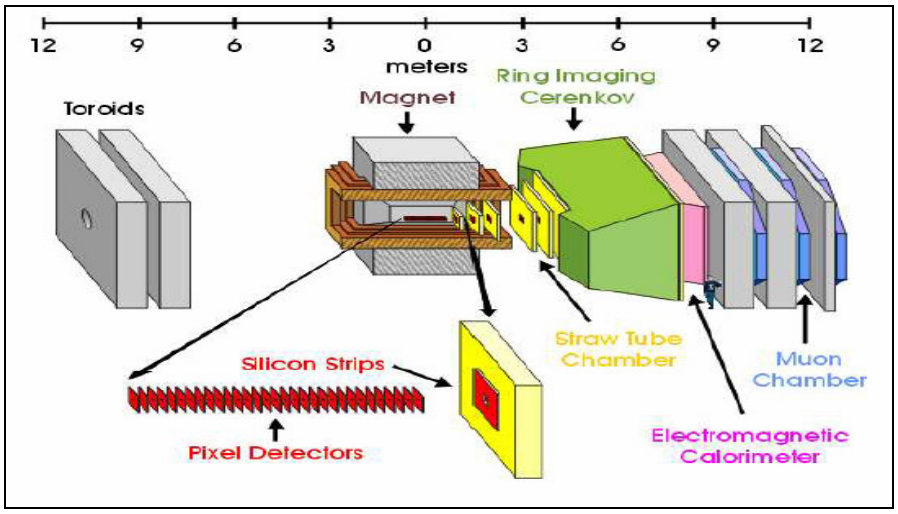


Figure 5 BTeV Detector

Clearly, the aggregate data rate is too high to blindly record all data. Instead, algorithms, called Triggers, must be executed online to dynamically compute a keep/discard decision. These algorithms necessarily must be computed in real-time, although significant queuing is typically available. Distributed and concurrent computing solutions are necessary in order to perform the trigger algorithm in real-time. It is estimated that ~ 2500 front-end current-generation DSP processors and ~ 2500 general-purpose processors will be required. The primary focus of our research collaboration, which is known as the Real-Time Embedded Systems (RTES) group within the BTeV experimenters' community, is with the Trigger algorithm and its fault-tolerant execution. Next we provide an overview of the planned architecture for the trigger implementation.

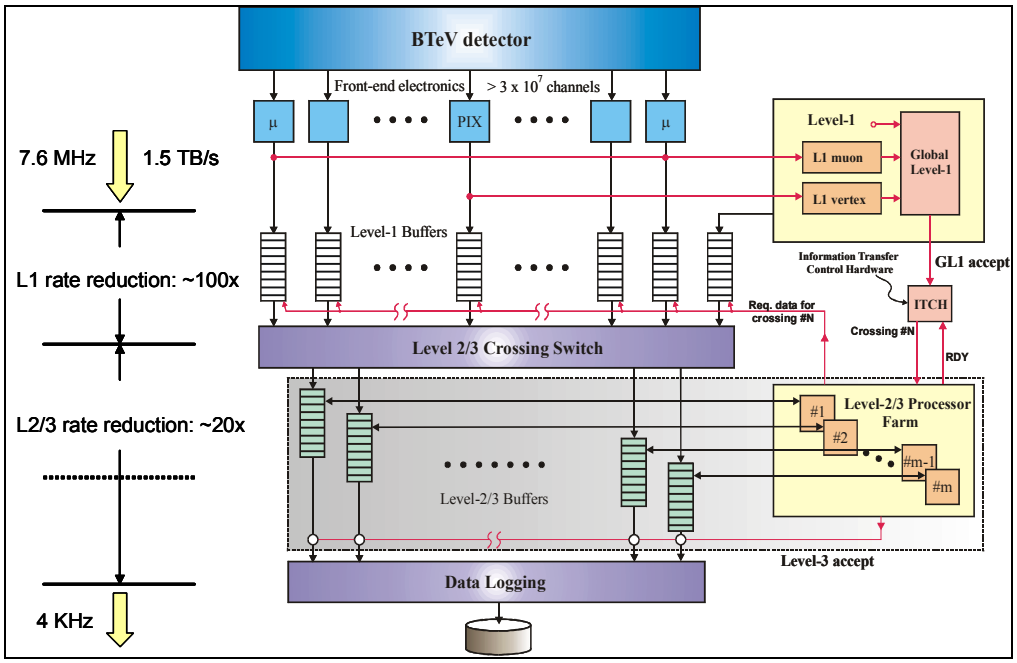


Figure 6 The Trigger Architecture [1]

BTeV Trigger Architecture

Functionally, the trigger spans three different levels. L1 is the first-level of trigger system responsible for detecting secondary vertices in a 3-dimensional data set, and rejecting non-interesting data-sets [15], with a reduction order averaging ~100:1.

L2 is the second level of the trigger algorithm. It does a refined tracking and vertex cut [34], with a reduction order of ~5-10:1. L3 is the third and final level of the trigger algorithm responsible for a complete event. L3 has a reduction order of ~5-10:1. Figure 2 shows a view of the trigger architecture.

This functional organization is mapped over three-categories of processor architectural organization. The L1 is implemented by a network of Field Programmable Gate Arrays (FPGA-s), and a network of Digital Signal Processors (DSP-s). The L2 and L3 is implemented over a network of commodity (general-purpose) workstations. The

FPGA-s are programmable, but are typically not dynamically loaded and adapted during runtime. Some limited fault-tolerant concepts can be applied in the FPGA hardware designs, but since deployment is relatively static, FPGA-s are currently considered outside the scope of the fault mitigation target.

DSP Network

A network of Digital Signal Processors (DSP) is the primary computational engine for the L1 trigger. Chosen for their simple architecture, low-power and efficiency at low-level mathematical operations, the DSP-s must sustain high input data and processing rates, and operate with stringent real-time requirements. It has been estimated that ~2500 DSP-s are required. A DSP has minimal resources – 1 MB SRAM, and 64 MB DRAM – with no memory management facilities. To keep overhead to a minimum, a small microkernel provides a small set of facilities to the programmer. Communication facilities are also minimal. A point-to-point network allows transfer of data packets between processors. DMA co-processor assists the transfer over the communication ports and permit direct access of data from specialized I/O interfaces.

The DSP processor network is configured in application-specific topologies that optimize the aggregate system bandwidth. Since multiple communication ports are available per processor, the network can be tuned to increase bandwidth, or to support redundant pathways between processors.

A network of general purpose computers with roughly the same node count as the DSP subsystem is the primary computational engine for the L2/L3 trigger. The planned architecture leverages commodity processors implementing a large-scale Linux-based network. These nodes are connected via high-speed Ethernet interfaces, with multiple

interfaces in each permitting some fault resilience. Each node has more resources than a DSP-s, with large memory (~1 GB), storage, and runs a full-scale commodity operating system. In addition, the real-time requirements are much more relaxed than with the DSP-s. The nodes still have soft real-time requirements, but the deadlines are at least an order of magnitude greater

Buffers and Switches

A third component of the BTeV trigger architecture which is present across multiple levels are buffers and switches. The pixel data from the detectors is collated into packets and time-stamped, and are queued in a large buffer (L1 buffer). A de-multiplexing switch delivers elements, which represents a single crossing's (collision) worth of data, from the queue to DSP-s in a round-robin fashion. The data is deleted from the L1 buffer if the trigger algorithm results in a reject decision; otherwise it is retained for further processing in the L2/3 trigger.

Given the scale of the BTeV Trigger system, a wide range of failures can be expected. Failures ranging from mechanical failures such as fans, cables; to electrical failures such as power supplies, voltage regulators; to processor hardware failures such as memory errors, communication link errors, processor and board failures; to software failures, such as algorithm bugs, are expected to occur. Moreover, the constraint of operating in proximity of a high radiation environment, with budget and technology limitations (radiation hardened processors are typically a generation behind their commonly available counterparts, and an order of magnitude greater in cost) precluding the use of radiation hardened devices, renders the system more vulnerable.

When failures do occur, they must be corrected very quickly. Otherwise the system will rapidly saturate the available queuing and start dropping potentially valuable experiment data. The loss of data in the event of a failure must be minimized and controlled. A highly coordinated fault mitigation mechanism is required that responds rapidly to failures, restoring the functionality in a short period of time thereby limiting the loss of experiment data. In the following section we describe the self-adaptive approach that rapidly adapts the system when failures occur retaining the system functionality, and in a longer time-scale re-optimizes the system to minimize the performance degradation.

Proposed Solution

The system to support BTeV experiment is extremely large and expensive. Budget limitations preclude redundancy approaches to add robustness to the system such as the triple mode redundancy, or more than that. The physics community would not permit the "waste" of these resources, so any redundancy will be consumed with elective processing. The solution needs to be robust and cost effective. In this case a, fault mitigation approach is more desirable than traditional fault tolerance. Fault mitigation allows the system to adapt, giving a 'best-effort' behavior. The definition of 'best-effort' is application and mode specific.

A proposed solution provided by the BTeV group at Fermi lab is shown in Figure 7 which shows the overall approach where the modeling tools provide the ability to define expected faults and corresponding fault mitigation actions. These actions can be analyzed and simulate the fault mitigation behaviors to assess the systems ability to respond to different failure scenarios. Analysis and simulation allows assessment without

having to create expensive test-beds and running exhaustive tests. Moreover, the tools have the ability to synthesize low-level programming artifacts from the higher-level abstractions, alleviating the system developer from the burden of constructing low-level artifacts and ensuring consistency with the higher-level abstractions.

With this motivation we have been developing high level design tools, and a highly matching flexible fault-tolerance runtime infrastructure. With this approach, the physicists – domain experts and users of the system – model the system using a domain-specific graphical language (DSL). Here they specify the hardware configuration, the software application and the fault-mitigation behaviors.

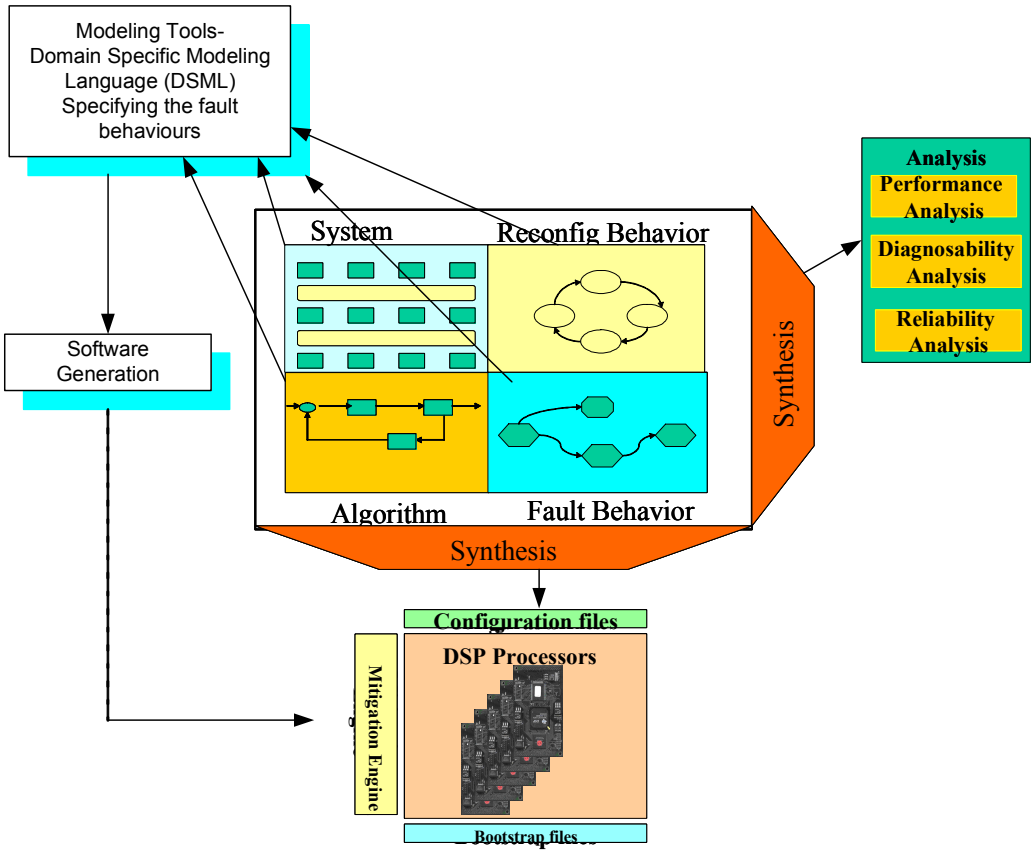


Figure 7 Model Based Approach to Fault- Mitigation

The DSL is implemented within the Generic Modeling Environment (GME) tool, which is a meta-programmable modeling environment, developed at the Institute of Software Integrated Systems ISIS, Vanderbilt University.

Model Integrated Computing (MIC) is a design philosophy that advocates the use of domain specific concepts to represent system design. The models capturing the design are then used to synthesize executable systems, perform analysis or drive simulations. The advantages of this methodology are that it speeds up the design process, facilitates evolution, helps in system maintenance and reduces the cost of the development cycle [14].

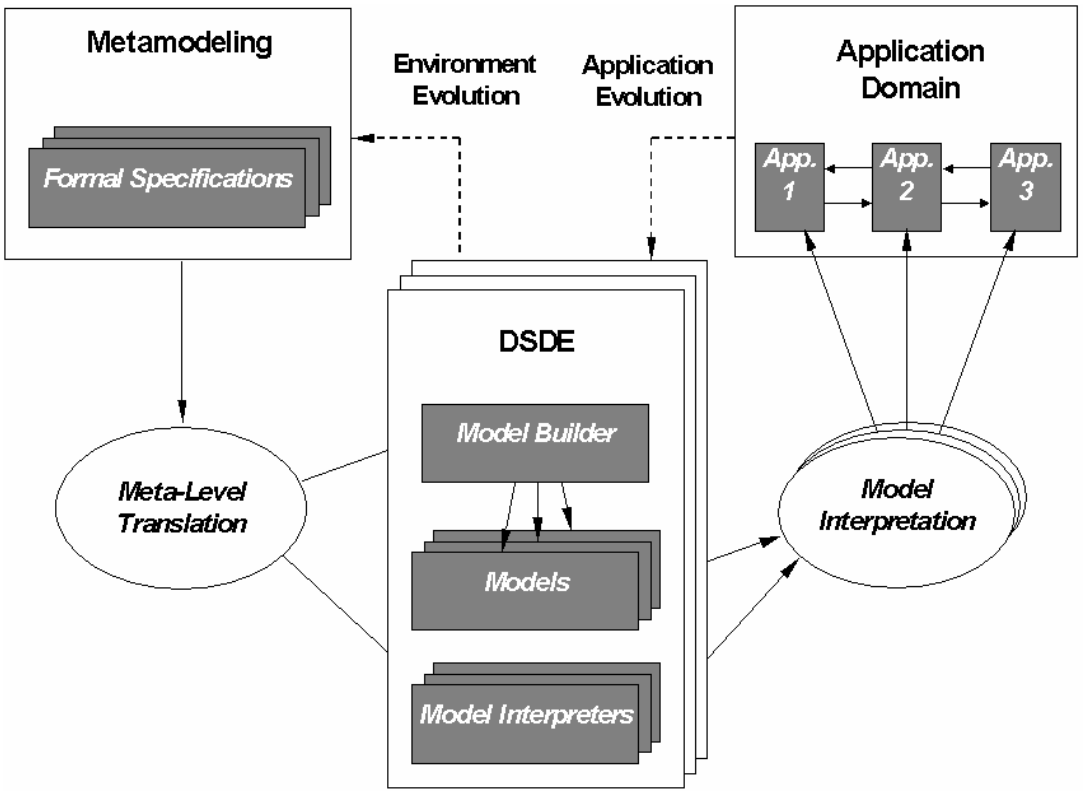


Figure 8 Design flow from metamodeling to application synthesis using GME

A metaprogrammable toolkit within Generic Modeling Environment (GME) implements the MIC methodology. It provides an environment for creating domain-specific modeling environments [11][14]. The GME metamodeling enviroment provides a graphical interface similar to UML [21] class diagrams (tutorial in appendix A), in which the user can specify the modeling environment to be developed for the specific domain. The graphical description is referred to as a metamodel. It captures the syntax, semantics and visualization rules of the target environment. In the metamodels the user can specify the set of entities or objects that can be created in the target environment, their organization and interactions with other entities. It also specifies associations, grouping and ordering of these objects. The target environment implements a modeling language, which allows the user to create any possible models consistent with the metamodel. A tool called the meta-interpreter interprets the metamodels and generates a configuration file for GME. The file is used to automatically configure GME so that it behaves like the target environment. Thus GME is used as both the metamodeling environment and the target environment.

GME models are entity relationship diagrams that are graphical and hierarchical with multiple aspects and attributes. The semantics of the models are enforced in two stages. The first stage is in the form of constraints that are applied to the models to enforce the static semantics. These constraints are specified in the metamodel using an Object Constraint Language (OCL) type specification and are applied on the models using a built-in constraint manager. In the second stage dynamic semantics are enforced by the model interpreters. Model interpreters parse the application models and generate source code, configuration files and analysis as output.

The key elements of an MIC-based approach in our application are:

1. A **Domain-specific modeling language (DSML),** the syntax and static semantics of which are precisely defined using UML-based notations and OCL-based constraints in a meta-programming environment.
2. System developers build **Integrated multiple-view models** in the DSME capturing information relevant to the target system from several aspects. The information captured includes adaptive behaviors, information processing architecture, and physical architecture of the target system.
3. **Model translators**, generate inputs to various analysis tools, as well as synthesize various low-level artifacts for instantiating/deploying the system.

Summary

Given the scale of the BTeV Trigger system, a wide range of failures ranging from mechanical failures such as fans, cables; to electrical failures such as power supplies, voltage regulators; to processing hardware failures such as memory errors, communication link errors, processor and board failures; are expected to occur. Moreover, the constraint of operating in proximity of a high radiation environment, with budget and technology limitations (radiation hardened processors are typically a generation behind their commonly available counterparts) precluding the use of radiation hardened devices, renders the system more vulnerable. The following summarizes the motivation behind handling such kind of systems.

1. When failures do occur, they must be corrected very quickly, or the system may rapidly saturate the available queuing and start dropping valuable experiment data.
2. The loss of data in the event of a failure must be minimized.
3. A highly coordinated fault mitigation mechanism is required that responds rapidly to failures, restoring the functionality in a short period of time thereby limiting the loss of experiment data.
4. Self-adaptive approach that rapidly adapts the system when failures occur to retain the system functionality, and in a longer time-scale re-optimizes the system to minimize the performance degradation.

CHAPTER IV

MODELING ENVIRONMENT

Prototype Overview

In the previous chapter we presented an overview of the BTeV system. In this chapter we will discuss how we design, specify and implement the system using model based tools.

Prior to discussing the modeling tools we discuss a prototype software application that was implemented as a case study to simulate the L1 Trigger of the BTeV system. The L1 trigger prototype contains several different software entities which are as follows:

1. ***Generator***: simulates the behavior of the detector subsystem which is responsible for collating the pixel data from the detectors into time-stamped packets that are queued in a large buffer (L1Buffer).
2. ***Buffer Manager***: simulates a large buffer of 120kb on each farmlet for queueing up events prior to processing.
3. ***Switch***: mimics the behavior of a round robin network switch that distributes the incoming events from the L1 Buffer to the Buffer Managers It is a de-multiplexing switch which delivers events, which is crossings worth of data, from the queue to Buffer Manager DSP's in a round robin fashion. The data is removed from the L1 buffer if the trigger algorithm results in a reject decision; otherwise it is retained for further processing in the L2/3 trigger.
4. ***Workers***: are composed of processes which include the **Physics Application** (Trigger Application), **Detector** and the **Local Manager**. The *Physics*

Application (PA) grabs the events from the Buffer Manager and executes the trigger algorithm on the given event. The *Detector* is a fault detection process which monitors the occurrence of different types of faults including those in the Physics Application (ex: PA is stuck in a loop, event data errors, etc.). The Detector sends a message to the nearest Manager (Local Manager) to notify of the failures. The *Local Manager* acts a lowest level of Fault Manager, handling the faults in a user-defined manner.

The **Fault Managers** are synthesized from the modeling tools, using the user-specified fault mitigation behavior captured in the models. A highly coordinated fault mitigation mechanism is required that responds rapidly to failures, restoring the functionality in a short period of time thereby limiting the loss of experiment data.

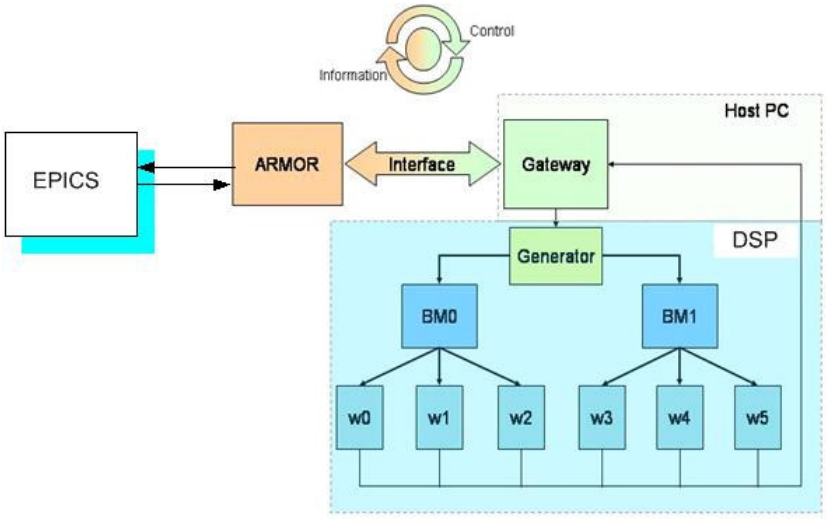


Figure 9 Prototype Software Application

As shown in Figure 9 EPICS is the user interface and ARMOR is fault manager software – a product of UIUC [20]. In the BTeV application, we have defined three different hierarchical levels of control: local, regional, and global fault-managers. Hierarchy is a simple, yet powerful concept [32] that has been applied and proven in many types of complex and large-scale organization. Clearly, in a system of this size, sending all fault-information to a centralized fault-manager for a mitigation decision is not a scalable approach. Reaction time would suffer in small systems, and be increasingly large as systems are scaled up. We propose a hierarchical organization of fault-managers (FM) operating independently. Each FM has its individual control domains. FM coordinates with peers and parents/children, above/below outside their control domain. This improves the reaction time, and enhances scalability. Furthermore with distributed and coordinated decision making, a single-point failure of a centralized fault manager will not cause the entire system to fail.

The *Local* managers are the leaf nodes, responsible for sensing faults and implementing control actions and making low-level mitigation decisions. *Regional* managers handle successively larger regions of hardware and higher levels of mitigation behavior. There can be multiple levels or layers of regional manager. *Global* managers are the top-level fault mitigation agents, interfacing with external systems and/or users. These levels roughly correspond to the inherent hierarchical organization of the BTeV Trigger architecture.

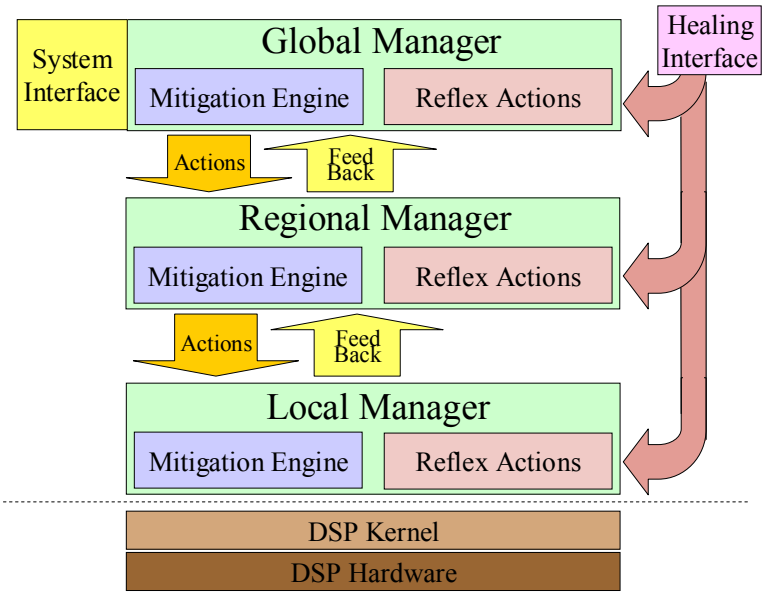


Figure 10 Hierarchical Fault Mitigation Runtime Environment

Faults are handled at the lowest layer possible. At any specific level, if mitigation is not possible due to resource availability, or lack of sufficient contextual information, the fault is promoted to the next level of fault manager. This has the advantage of scaling, since the number of fault managers at the lower levels increases with the architecture size. In addition, response to a failure can be more rapid, since decisions are made closer to the source of error.

BTeV Paradigm

Mapping of Prototype into Modeling Environment

We now define a language to specify the Software /Hardware and the Fault Mitigation behavior of the target application. The meta-programmable tool GME implements a modeling environment with this graphical language. The goals of the

domain-specific graphical abstractions are to be are particularly natural to the physicists, amenable to analysis, and suitable for system synthesis. The BTeV modeling paradigm attempts to meet these goals by allowing for the specification of system from several different aspects. The significant aspects are:

1. Application **Data Flow**: the component-based specification of information processing to be performed by the system,
2. Hardware **Resources**: the physical computer hardware, consisting of processors and interconnections, used in system implementation, and
3. **Failure Mitigation Strategies**: the specification of how the system should detect and react to component and system failures.

These aspects will be described in detail below.

Application Dataflow

This Application data flow aspect allows a system developer to define the key software components and the flow of data between them. The semantics are captured as a Dataflow Model, a modeling formalism, particularly suitable for modeling image and signal processing computations [37]. The basic dataflow model doesn't support hierarchical representation. However many extensions have been proposed that introduces hierarchy in the dataflow model [38]. Based on these extensions a hierarchical dataflow notation is used, where nodes (boxes) capture the software components (algorithms) and lines show the flow of data between nodes. Each of the software components are connected to each other by means of *Ports*. These models can represent synchronous or asynchronous behavior, and a variety of scheduling policies. For the BTeV trigger, these are primarily *asynchronous* in operation, with data-triggered

scheduling. The primitive software components represent a single computational block or algorithm to be executed in the runtime system. Each software primitive in the dataflow modeling are associated with a script that provides the implementation of the software component. Software Compounds are hierarchical composition of nodes. A Compound can contain other Primitives or Compounds. These data flow models are derived from those used in ACS-MIDE [19][24] and are made closely resemble a directed data flow graph [33].

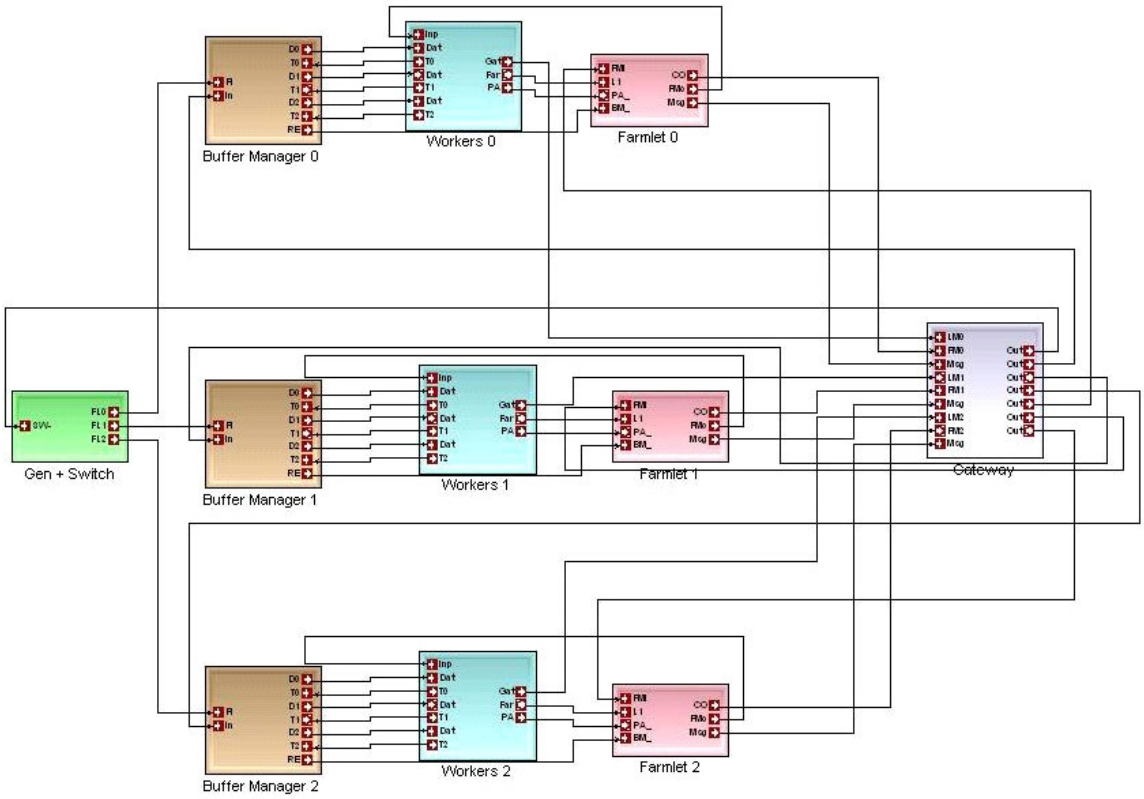


Figure 11 Software Component

Figure 11 shows a screen-shot of an example dataflow modeling aspect of the trigger prototype with several processes and dataflow links between them. The network of heterogeneous processing elements constitutes execution resource of the system. The set of computational components, the communication topology between the components, and the resource allocation together define the computational structure of the system. System configuration refers to computational structure of the system. Formally as mentioned in [24] the execution resources may be expressed as a set R of resources (processing elements) available for system execution.

Formally the computational structure of the system may be expressed as 3 tuple

$$\{P, F, A\} \tag{6}$$

where,

P is the set of computational processes (components)

$F \subseteq P \times P$ is the set of dataflow between processes and

A: $P \rightarrow R$ is the resource allocation. Each process is assigned to a processing element.

The semantics of the computational structure can be described with an attributed directed graph know as process graph [37]. The nodes of the graph are computational processes. The edges of this graph represent communication (dataflow) between processes. Conceptually the processes operate continuously and concurrently transforming infinite sequence of input data to infinite sequence of output data. The processes communicate via exchange of data tokens. The communication is asynchronous and the tokens are buffered in FIFO queues. The processes in the process graph are distributed and executed over the set of resources R.

Hardware Resource Library

This Hardware resource aspect defines the physical structure of the target architecture. Block diagrams capture the processing nodes (e.g. CPU-s, DSP-s, FPGA-s). Connections capture the networks and busses over which data can flow. These hardware resources provide the processing capabilities for executing software component currently modeled in the system. These hardware models are decomposed hierarchically to enable system scaling and support the use of types and instances to preserve the correctness of lower level hardware models.

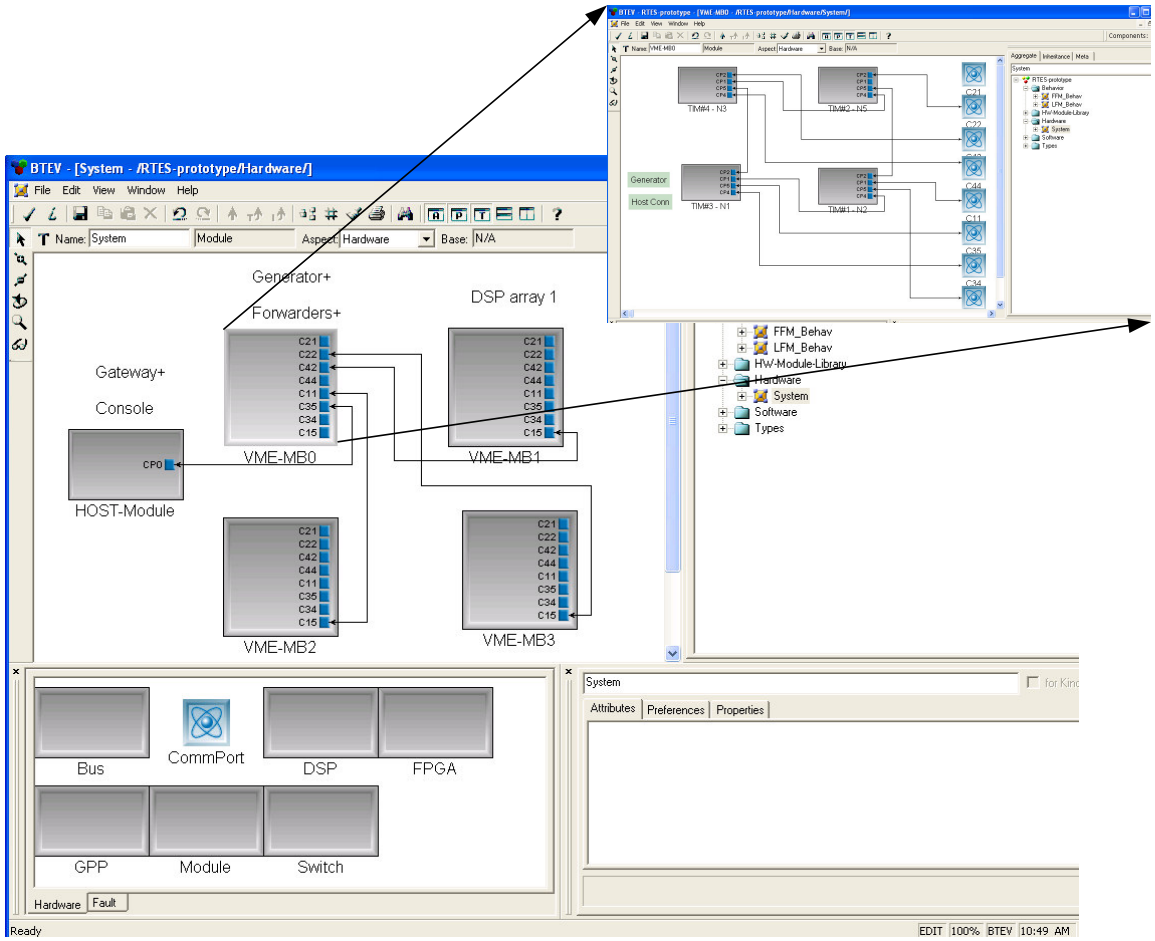


Figure 12 Hardware Component

Figure 12 shows the segment of the hardware resource model for the prototype – 4 VME boards are wired to the host processor .Each VME board has maximum 4 DSP processors (TI TMSC6xxx).The hardware picture is arranged out the same way as the actual physical prototype hardware. The language for modeling the hardware component is again defined in UML.

CHAPTER V

FAULT MITIGATION LANGUAGE

As discussed previously about our approach towards mitigating the fault uses three level of hierarchy of fault managers operating independently with their own control domains and coordinating with peers outside their control domain. In this present chapter we will see the detailed explanation of the generation of these Fault- Manager behaviors from the models.

Requirements

Historically physicists have developed custom software and hardware for experiments such as BTeV [15]. Any fault-mitigation would be manually programmed into the system. The motivation behind the fault mitigation strategy language is to provide a flexible method for specifying arbitrary mitigation behaviors. This tool provides for integration of prior knowledge into the fault detection system and the ability to use a recursive narrowing of fault probabilities to aid in the identification of symptoms. The goals of the fault mitigation language are as follows:

1. *User friendly* , keeping in mind that the target users are physicists
2. The language should be able to help the physicists introduce custom *self-adaptive behaviors* without any difficulty as they are the best people to define how the system should behave in fault conditions.
3. The language should integrate the application specification and design, since application is closely linked with the fault mitigation behavior.

4. It should support direct generation of software and system configuration artifacts from the specifications.

User Defined Mitigation Strategies

State machine [4] based formal specification methodologies have found widespread acceptance in system design and development. This is especially true in the field of hardware /software Co-Design, since concepts of finite automata are often used to describe reactive system behavior. In addition to defining the normal system behavior, as in typical HW/SW codesign, we describe the *desired* behavior under fault conditions

One of the aspects of the BTeV modeling paradigm is the establishment of fault-mitigation strategy. A Statecharts-like [4] notation is provided that allows a developer to define various failure states. Conditions necessary to enter or leave these states, along with actions to be performed when state transitions occur are also defined in transitions. The language can be summarized at a high level as follows:

1. The nodes in the state diagram are system states, corresponding to a particular phase of system operation or a mitigation step.
2. Lines are transitions between states, capturing the logical progression of system modes. Transitions occur in reaction to specific events (i.e. hardware faults, OS faults, user-defined errors, fault-mitigation commands from higher level of fault-managers etc).
3. Transitions are annotated with triggers, guards and actions. Triggers determine the specific combination of events present when state transition should occur.
4. Actions define the operations to be performed as a transition occurs. These actions can include moving tasks, rerouting communications, resetting and

validating hardware and changing the application algorithms. An action language definition is currently being developed, and is left as future work.

The hierarchical mitigation scheme is specified by: 1) defining fault-managers to be executed at specific levels in the dataflow modeling hierarchy and 2) associating fault-mitigation strategies with each fault manager. In effect the fault-mitigation managers can be thought of as a system of concurrent and coordinating state-machines interacting via system state detectors data and event propagation.

The first step in the development of a new language is to specify the syntax and the visualization in the GME [14] -modeling environment. The meta-model for the state machine language uses a notation based on UML [21] class diagrams describing the key concepts of the modeling language and their associations. Figure 13 shows the meta-model of the fault mitigation language. It can be seen that *Behaviour* Component contains *Machine* which in turn contains the *State*. The *Machine* is the *Fault Manager* whose behavior can be described using StateChart concepts. Based upon whether they receive or send data the *Ports* can be specialized as *Input* or *Output*. The Behavior state machines perform actions based on triggering conditions. The *Triggers* is defined as a Connection which has *attributes*. These attributes specify the triggering condition and action to be performed. The action code which is the *fault handling* code is mainly written in C and based on the type of the guard/trigger. The action is user defined e.g. to send the message (action, error, statistical) upstream or downstream. These triggering conditions are logical equations using messages as inputs. Messages are utilized to propagate notification of failures up through the hierarchy, interact with fault detection processes, or communicate with other system components (e.g. user interfaces or system control).

-

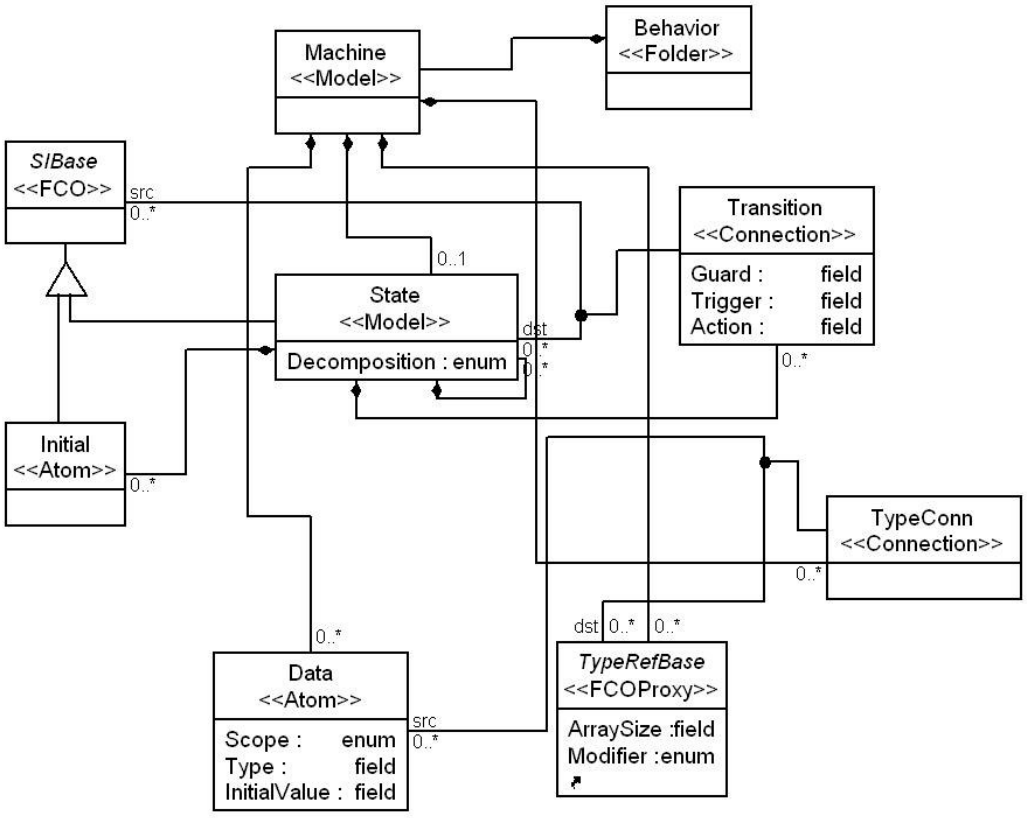


Figure 13 State Machine paradigm

In order to minimize the bandwidth while providing the maximum flexibility, the messages are specified to have a variable length, based on the originator of the message.

Message Structure

Considering the reactive and distributed nature of the system there is a need to send messages containing commands, status, and monitoring data back and forth between various software components in BTeV trigger systems. The messages defined for this prototype are:

1. *Faults/Error Message* for reporting errors in hardware, or application

2. *Controls/Command* serves as both decision requests and commands that force parameters to change in the running system
3. *Statistical/Info* is periodic in nature and they contain the data of averages over n time.
4. *Response Message* is feedback message which are sent when any kind of command or error messages are either executed or failed to execute.

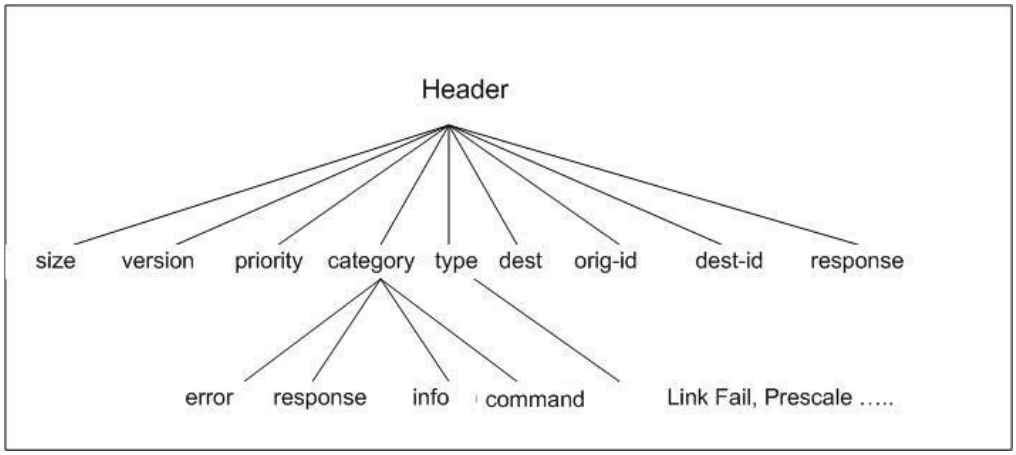


Figure 14 Message Structure

The above Figure 14 shows the message structure and the hierarchy of the message schema.

Input and Output of the Fault Manager Machine

The state machine is an automaton whose state transitions may involve multiple input and output simultaneously on any number of ports. The key concepts here are the states (and in particular the transitions between them) and the interactions. By interactions we mean the explicit buffered communication via named *ports.* On each port

one receiver listens to potentially many senders. The input buffers are a set of message FIFOs. Message exchange is triggered by a state machine output within an Action specification or by other parts of computational environment. Inputs can be queued, i.e. they may occur at any time, appending the received value to the corresponding FIFO. The values in the input buffer will be processed based on the scheduling of the fault manager and its internal behavior. Processing is done by the user specified transitions, which is written in C language checks the type of message and uses it in a transition condition or guard. Based on the type of message the physicist can specify an action to be taken. This action would range from simply forwarding the message up the hierarchy to a parent fault manager, notification to another entity on the Fault Manger processor, updating an, activity log or internal state, restarting processes or hardware, or rerouting a communication link (if there is a link failure). Figure 15 shows the input and the output of the state machine.

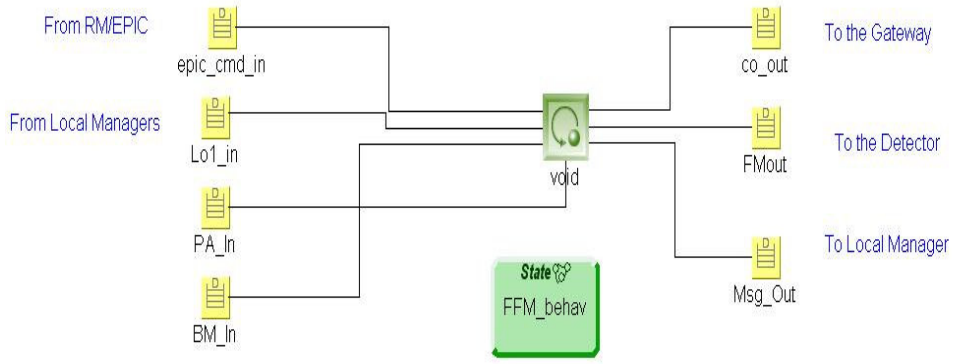


Figure 15 Input and Output of State Machine

In order to use the system model described above, the messages and FSM definition is formally defined as follows:

Formally the messages can be expressed as:

Let ***M*** be the type of all messages potentially exchanged by the state machine (extended) and ***PN*** be the type of port names. Then the message families which are used to denote both input buffers and the output patterns have type

$$MSGs = PN \rightarrow M^* \quad (7)$$

where,

M*is any finite sequence of elements of ***M.***

States and Transitions: The type of a state is

$$STATE(\sum) = MSGs \times \sum \quad (8)$$

where, the parameter $\sum$ stands for the type of the local state. The set of transitions has type

$$\mathbf{TRANS}(\sum) = (STATE(\sum) \times MSGs \times STATE(\sum)) \quad (9)$$

where, each of its elements has the form **((i , τ) , o ,(i ‘, τ’))** and means that the state machine can perform a step from local state τ to τ’, taking the current input buffer contents from **i** to **i’** (thus consuming as much as input as required) and producing output **o**. Here **i**, **i’** and **o** each denote the whole families of FIFO’s.

Formally, the Fault Managers behavior can be given as a 5 tuple:

$$(Q, In, Out, \tau_0, Trans(a)) \quad (10)$$

where ,

-*Q* is the set of finite states

-*In* is the set of input ports names

-*Out* is the set of output port names

- τ_0 is the initial local state

-*Trans (a)* is the transition relation

Various Features Of State – Machine Language

Since the classic state machine doesn't handle hierarchy, extensions have been proposed to the FSM representation to introduce hierarchy and concurrency by Harel [4].The main entities of the state machine are:

State: a particular condition in which a fault manager may consume signal /messages and evaluates conditions for transitions. The states are extended with two main concepts:

-***State Hierarchy:*** The semantic of hierarchical state decomposition are designed to allow sharing of behavior

-***Orthogonal Regions****:* Hierarchical decomposition can be viewed as the classical exclusive-or applied to states. State hierarchy can be viewed as *or-decomposition* and the nested states are called or-states. UML statecharts also introduce the complementary *and-decomposition*. This decomposition means that a state can contain two or more orthogonal regions which means independent state, and being in such a composite state entails being in all of its orthogonal regions simultaneously [4].

Trigger: A trigger uses a sequence of activities triggered by the (consumption of the signal/message/events) and evaluates the information with a logical equation. The

transition specifies a initial state and a destination state. In most general terms, an event is an occurrence in time and space that has significance to the system.

Guard: a particular condition which needs to be satisfied before taking the transition. It can be seen as a Boolean expression which is evaluated dynamically based on the value of extended state variables. Guard conditions affect the behavior of a state machine by enabling or disabling certain operations.

Action: An action is an activity in reaction to a transition. Every state in the model can have optional entry actions and exit actions.

-***Entry Action****:* Actions which are performed upon entering into the state

-***Exit Action****:* Actions which are performed upon exit of a state

Entry and Exit actions are associated with the states, not transitions. Such actions allow the designer to implement a common desired behavior without replicating for all transitions touching a state. This results in more ease of use and reduction in modeling errors.

Data Type*:* Data types can be defined which are translated to a state variable. The state variables maintain local state to be sued in transition, guard, and action logic. These data types can be of type int, float, volatile etc as defined by the user. The user can also specify the initial value of the data variables

***Timer**:* In order to express real time constraints, the state machine engine provides access to the actual time. A timer can be created with an expiration time. The expiration of the timer creates a signal to the state machine, which can be processed as any other input signal. This timer is dependent on the underlying OS and it is implementation specific.

***Action Language**:* Actions associated with states, transitions and top level default entrance are defined by statements of the FSM language. This language is an evolving offset of built-in functions and macros. Currently there are built in functions for sending message on user defined port numbers. Extensions will include user specific action to a particular fault which may include setting a variable for reset processor, reroute the link etc.

Standard State Machine Implementation

Implementing state machines efficiently is challenging. Even with the classical non hierarchical state machines, we must make large number of design decisions and various tradeoffs. Typical implementations of state machine in the high level programming language such as C, C++ include

- Nested switch statement,
- The state table
- The object oriented state design pattern and
- Other techniques that are mostly combinations of the previous three.

The majority of published state machine implementation techniques use state machines that are intimately intertwined with a specific concurrency model and a

particular event dispatching policy. For example, embedded systems engineers often implement state machine inside polling loops or interrupt service routines (ISRs) that extract events directly from hardware or global variables. GUI programmers are typically base code on a runtime model that handles event queuing and dispatching for the programmer. For a scalable real time system, it is better to separate the state machine code from a particular concurrency model and to provide a flexible way of passing signals and event parameters. Implementations in this chapter provide a simple and generally applicable interface to a state machine.

Nested Switch Statement

Perhaps the most popular technique of implementing state machines is the nested switch statement. States and Signals are typically represented as an enumeration. Basically this technique uses two levels of the switch form of multi way decision. Thereby, a scalar variable discriminates the first level and an event signal is used in the second level.

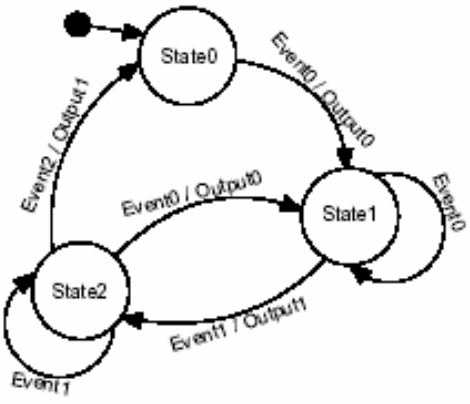


Figure 16 Example FSM for Standard Implementation Techniques

Considering as an example the state machine from Figure 16 the corresponding nested switch implementation is shown by the listing from Figure 17

```
enum State {State0, State1, State2};
enum Signal {Event0, Event1, Event2};
State myCurrentState;
Signal myCurrentSignal;
switch (myCurrentState) {
case State0:
  switch (myCurrentSignal) {
  case Event0:
  myCurrentState = State1;
  send(Output0);
  break;
 }
 break;
case State1:
  switch (myCurrentSignal) {
  case Event0:
  myCurrentState = State1;
  break;
  case Event1:
  myCurrentState = State2;
  send(Output1);
  break;
 }
 break;
case State2:
  switch (myCurrentSignal) {
  case Event0:
  myCurrentState = State1;
  send(Output0);
  break;
  case Event1:
  myCurrentState = State2;
  break;
  case Event2:
  myCurrentState = State0;
  send(Output1);
 break;
 }
break;
}
```

Figure 17 Switch Case Implementation

The nested switch statement implementation has the following advantages:

- It is simple.

- It requires enumeration of states and triggers.
- It has a small memory footprint, since only one scalar state variable is necessary to represent a state machine.
- It does not promote code reuse since all elements of a state machine must be coded specifically for the problem at hand.
- Event dispatching time is not constant but depends on the performance of the two levels of switch statements. This degrades with increasing number of cases typically as O (*log n*), where n is the number of cases [39].
- Nested switch statements can be used to implement hierarchical state machine.

State Table

Another popular approach is to use the state tables (typically sparse) arrays of transitions for each state.

Signals---------→ (Triggers)

States----------→

	Signal1	Signal 2	Signal3	Signal 4
State1	Action11 () nextstate	Action21 () nextstate	Action31 () nextstate	Action41 () nextstate
State2	Action12 () ……	Action22 () ……..	Action32 () …….	Action42 () ……
State3	……	……	……	…..

Figure 18 State Table Representation

This technique uses a two dimensional *m* x *n* array to implement a state transition diagram. Thereby m is the number of states and n determines the number of possible

signal events. Transitions are represented by the contents of array cells, consisting of transition target state and associated actions. This table lists signals (triggers) along the top and states along the left edge. The contents of the cells are transitions represented as {action, next-state} pairs. As opposed to switch case implementation which depends on the number of states and possible signal events, the state table approach requires a large two dimensional array, which is typically sparse and wasteful. In conjunction with event dispatching time, the state table technique provides directly access to a transition with a complexity of O (*const*).

Summary

The standard implementation techniques and their variations discussed in this chapter can be freely mixed and matched to explore different trade-offs. For our purposes we will be using switch case implementation technique.

CHAPTER VI

CASE STUDY

The tools described in the previous chapter were tested by implementing a physical prototype. A specific subset of the architecture and a small set of sample errors were defined, along with the actions to handle these problems. The prototype system structure and fault handling procedures are described below.

The BTeV system can have many failure scenarios. Some of these failures were implemented and the behavior was studied. Recall from Chapter IV that the user interface is implemented in EPICS. EPICS has some of the user control modes and it can send messages directly from the user interface to the DSP's. EPICS act as a Fault Injector, which can inject the faults and the user can see the effects of those faults on the system. It has the following control buttons:

1. Start /Stop the System
2. Change the values of the Interaction per crossing
3. Change the values of Interaction Size
4. Set the Authority Vector
5. Send a message to hang a PA
6. Send a message to restart a PA
7. Send a message to run the systems well (robustness to faults) or poorly (crash on data errors).
8. Send the message to change the pre-scale factor.

The Authority vector is an array of permissions set to authorize the fault managers to perform certain mitigation actions autonomously. The motivation behind these authority vectors is to give physicist the ability to enable/disable certain automated behavior giving them an overall control over the system operation, which is necessary for system acceptance. These permissions are encoded as messages that affect state variables in the network. The three levels of Manager: Local, Regional (Farmlet), Global (ARMOR) need to set their authority bit vector in their local state in order to take a decision and act on the fault per their defined behavior.

Failure Scenarios

Case 1 – Supercomputing 2003 Implementations

Capturing fault scenarios is a useful way to understand and identify what a system is required to do. We will consider 3 main fault scenarios:

PA Application Hang

This scenario was considered to analyze the behavior of the system if the Physics Application gets hung. The reason for hanging may be possible due to various reasons of which one could be improper error handling leading to the program to get in an infinite loop. The figure below shows the data flow interaction inside the Worker node. The worker node has Local Manager, PA, and Very Light Weight Agents (VLA -Detector).

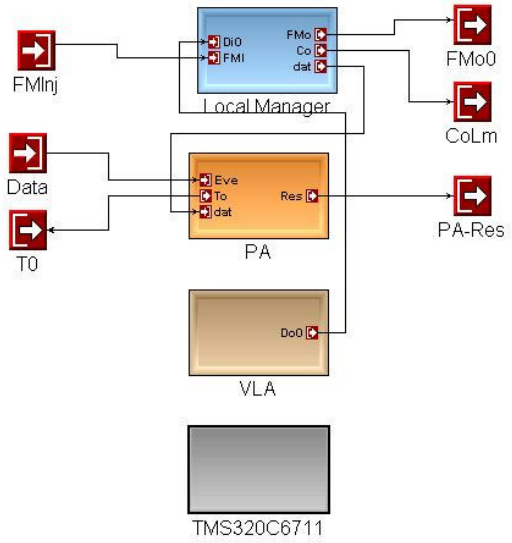


Figure 19 Worker Node

Local Manager resides on every DSP worker node in the network. It can receive messages from detector (VLA) and the Fault Injector (EPIC). Hence the local manager listens on 2 input ports: ***det_fault*** and ***fm_inj_in***. PA Application Hang (message type is PA_APPHANG) message comes down the hierarchy through the port ***fm_inj_in***. The message path is the following:

EPIC→ARMOR→GATEWAY (Console) →FARMLETMANAGER→LOCAL MANAGER

Once this message arrives on the port the local manager changes its state from NOMINAL_LM to PA_APP_HANG if the guard condition is true. In PA_APP_HANG state, it executes the exit action which sends a message to the worker (PA) of type **PA_APPHANG.** Now the Local manager can propagate message through its output

ports: ***fm_msg_out*** (to farmlet), ***data_rate*** (to PA), ***co_out*** (to console).When PA receives this message from the Local Manager it sets an **error_bit_value** in order to hang

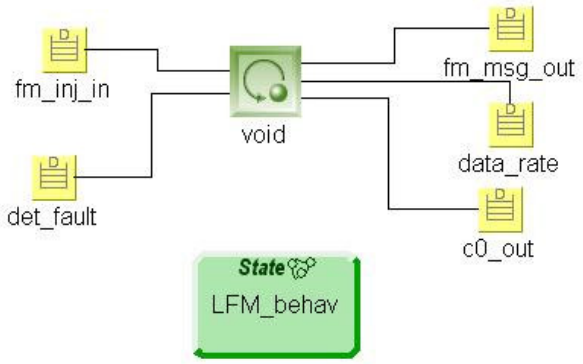


Figure 20 Local Manager data ports

the PA application by entering a while(1) loop till the bit is reset.

Results

When the message is sent for PA to hang, the Buffer Manager queue size starts to increase- since there will be only two alive worker nodes- and it will reach a red alert bar. As shown in the Figure 21, as soon as the red alert bar is reached the 1) system starts dropping events 2) system efficiency drops 3) system utilization increases.

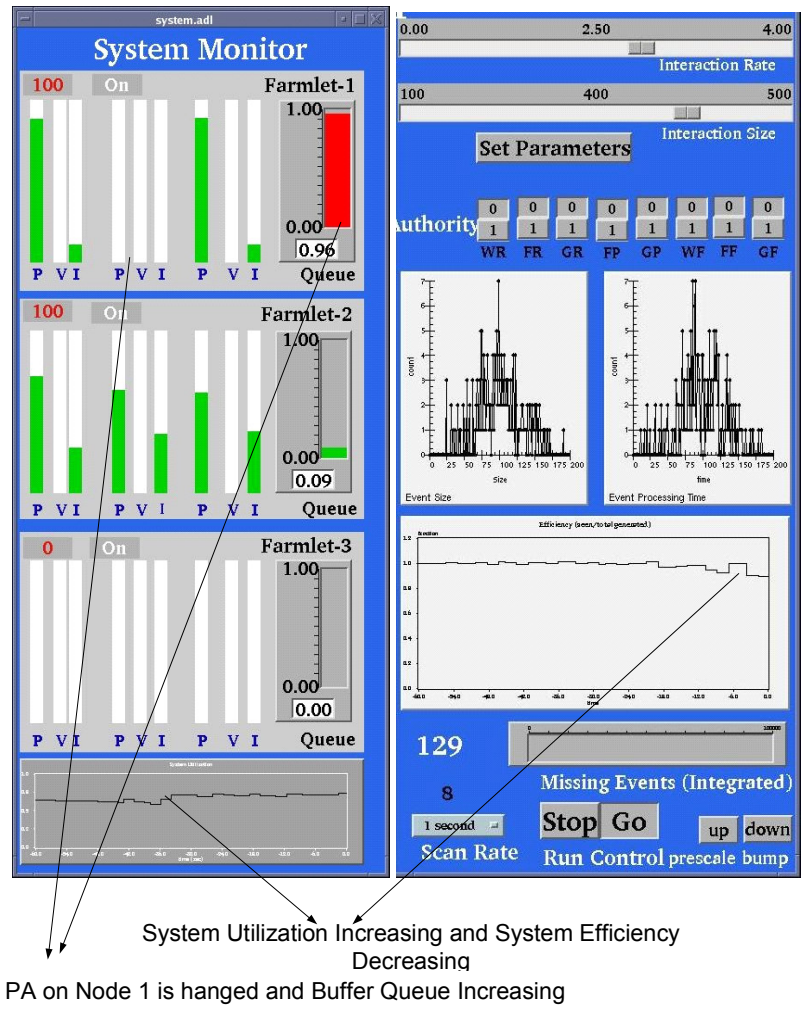


Figure 21 User Interface Showing the System Information

Prescale

Pre-scaling is a common technique that is applied in physics trigger application when the processing capacity is not adequate to process in real-time all the events generated by the detectors. Pre-scaling causes the trigger system to process only n out of m events where m is fixed (say 100) and n is the pre-scale factor which changes depending on the performance of the system. The remaining m-n events are simply marked as 'unprocessed' and queued for later offline processing. In this prototype pre-

scaling can be controlled by the system operator and/or by the fault-managers. The pre-scale factor changes when:

- System operator controls
- efficiency is too low
- average queue occupancy too high or too low
- average processing time is too high
- low utilization of resources (CPU)

Based on the authority either the Farmlet Manager or the ARMOR will start pre-scaling if it senses any of the above mentioned reasons.

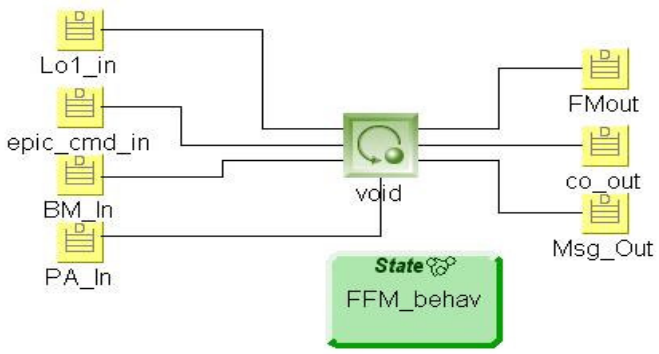


Figure 22 Farmlet Manager Data ports

The Farmlet Manager listens from four input ports simultaneously: Lo1_in (from Local Manager), epic_cmd_in (from EPICS), BM_In (from BufferManager), PA_In (from PA). Again the behavior of the Farmlet Manager is laid out as a state machine.

Since the Farmlet Manager gets the statistic message from BM_In port, it has the information about the queue size of the Buffer Manager; it keeps a track of the Buffer Manager queue size.

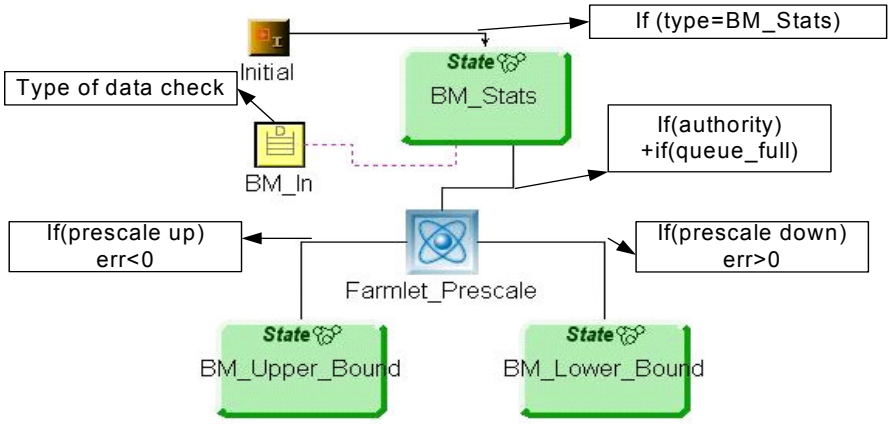


Figure 23 State Diagram for Prescale Behavior

The farmlet Manager applies a Proportional-Derivative (PD) controller algorithm to check for the constant increase in the queue size of the Buffer. This type of feedback controller produces a control output based on the error, between a set point and a measured process variable, plus a factor based on how fast the error is changing.

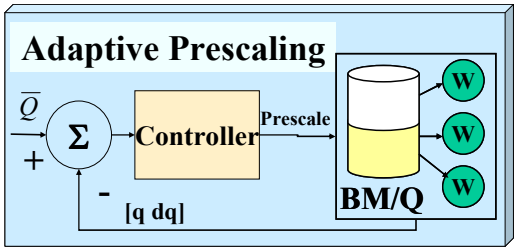


Figure 24 Adaptive Prescaling Using PD Algorithm

Each element of the PD controller refers to a particular action taken on the error.

- Proportional: Error multiplied by a gain Kp. This is adjustable and in many systems this is responsible for process stability.
- Derivative: The rate of change of error multiplied by a gain Kd. In many systems it is responsible for system response.

Looking into the Buffer Manager queue as the process (which is under observation), the values of Kp and Kd are:

```
Kp=1; Kd=1;
```

Figure 25 Values of Proportional and Derivative constants

The set point is configured to 40% of the queue size .Since the definition states that the process n out of every 100 events, we will consider the maximum queue size as 100, so the set point QSP_FFM 0.4.

After defining all the constant value we need to calculate the error value. The error value is calculated as follows:

```
err = Kp*(QSP_FFM -(bm_stats_monitor->avg_size/MAX_BM_Q_SIZE)) + Kd
*(0-avg_dq_size);
```

Figure 26 Equation for calculating error

Based on the value of error and also the authority the Farmlet Manager would change its states from *BM_Stats* to either *BM_Upper_Bound* or *BM_Lower_Bound* which is checked on the transition from *BM_Stats* to the two other states.

Results

To test this particular behavior we need to force the buffer queue size to fill up. We can do this either by 1) killing one or two of the worker node 2) Increasing the interaction rate or the interaction size, so that the PA is overloaded. We do the following few steps:

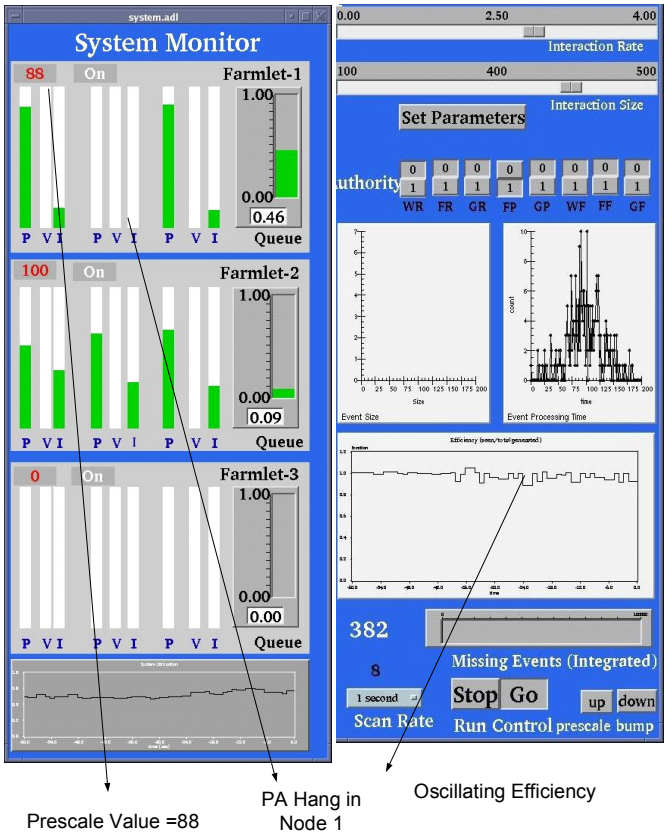


Figure 27 System Information showing the values of prescale and efficiency

- Set the Authority to Farmlet Manager.
- Kill the PA in one of the worker node.
- As the Buffer Queue size fills up, the pre-scaling comes into action, and we can see the queue size getting stabilizing after few iterations.

Buffer Manager Queue full

This error arises if the DSP's on the farmlet are not keeping up due to one or more processor deaths or algorithm is taking too long. When this error occurs the Farmlet Manager sends a activity log message to the ARMOR (up the hierarchy).

The complete behavior of the Farmlet Manager is shown below.

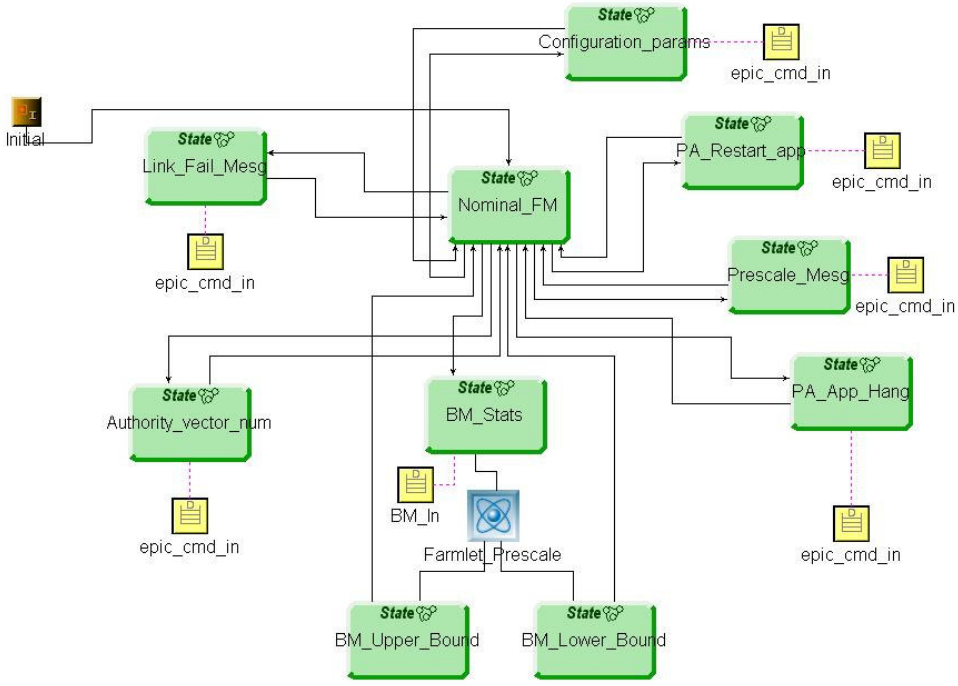


Figure 28 Complete Behavior

Semantics

This section gives the logical meaning of above mention state machine based language in detail.

Automata The state machine (extended) as given in Chapter 5, is 5 tuple:

a= (***Q, In, Out, τ 0, Trans (a)***)

where

-*Q* is the set of finite states

{*NOMINAL_FM, CONFIGURATION_PARAMS, LINK_FAIL_MESG, PA_APP_HANG, PA_RESTART_APP, AUTHORITY_VECTOR_NUM, BM_STATS, BM_UPPER_BOUND, BM_LOWER_BOUND, PRESCALE_MESG}*

-*In* is the set of input ports names

{*epic_cmd_in , Lo1_in, PA_In ,BM_In}*

-*Out* is the set of output port names

{*co_out , FM_out, Msg_out}*

- τ_0 is the initial local state

{*Nomial_FM}*

-*Trans (a)* is the transition relation

Case 2 – General Fault Scenario

The aim of this test case is to provide a methodology for the user to specify the behavior considering the real time distributed systems based on the formal tools. Providing a graphical environment for editing, prototyping and code generation have been successful in the system in which we are involved. Considering the real time systems where the timing constraint is an inevitable variable which needs to be

considered. We can picture the state machine to be consisting of local data environment and a behavior i.e., a timed state machine. Timing constraint can be involved in every state machine. A state transition is described by a guarded command with a timing constraint: G→C[τ]. The guard G is made up of state variables of the machine. The timing constraint [τ] is typically a timing interval [t_{min}, tmax], t_{min}<=t_{max}, which expresses the possible rendezvous times for an IO or the possible duration times for an internal command. Since the runtime environment provides a microsecond timer, the variable timer can be defined at any state utilizing the timer function.

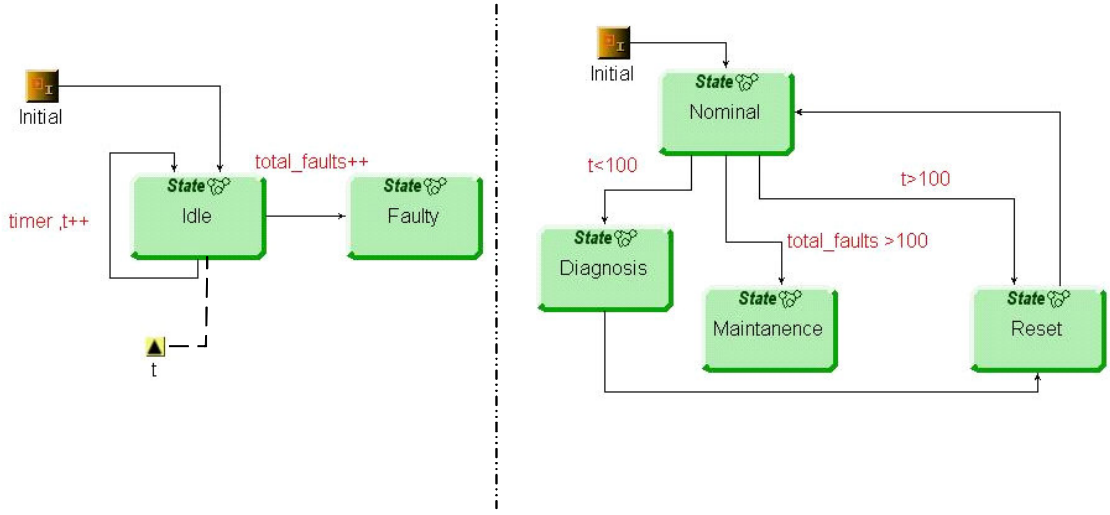


Figure 29 General Example showing the fault mitigation language

In the above example we consider a concurrent state machine. As seen from the example we have a timer which can be used to check if a particular fault state has been in

the same state for a long period of time (by specifying the upper bound). The state starts of in the *IDLE* mode and checks for any faults which occur in the system. If we keep a check on the *total_faults* occurring in the system and if we have an upper bound on the total faults (which is 100 in this case), we can change the state to *Maintenance*. In contrast, if we have the timer shooting just below 100 the process under observation can enter the diagnosis state, and if it shoots over 100 we can just reset the processor and start fresh. This example is just a high level idea of what other things can be done using the concepts of state machine.

Fault Manager Synthesis Algorithm

"Automation is key to agility" (Matt Stephens, [28])

Today, automated code synthesis of diverse input models is used in a wide range of application development. Especially in the field of system design, code generators fill the gap between high level design methodologies and low level application code. Therefore a complete design flow can be considered as seamless, if it's possible to generate portable code for a designed and verified model. Thus, in GME the gap between the models drawn by the designer and the code consumed by the fault managers is bridged using the model interpreters.

Input and Output

Interpreters parse the models to extract the required information. The foremost step is to describe the input and output of the interpreter. The input of the interpreter is the system design models built using GME. These models are stored in a database and can be accessed using COM (Component Object Model) API (Application Program

Interface). A high level C++ interface called the Builder Object Network (BON) also exists that enables the access of these models as C++ objects. Thus interpreter can access model information using BON. Applications described by the developers are stored as a network of objects. Inside GME these objects are instances of a set of generic classes in the BON. The class hierarchy of the generic BON is given in [14][11]. These generic classes can be extended for a specific paradigm.

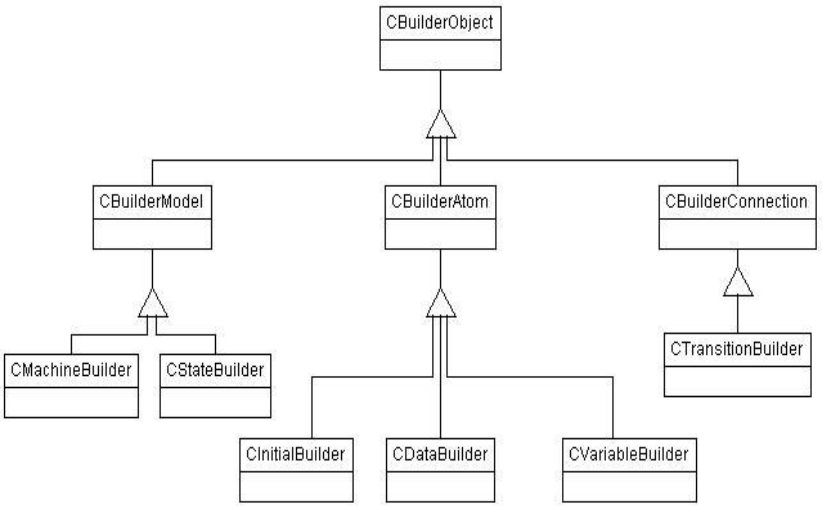


Figure 30 Class Diagram of BTeV specific classes

For the BTeV state machine paradigm these classes were extended according to theFigure 30. The CBuilderModel class is a generic class in the BON and all other classes in Figure 30 have been defined for BTeV. Each user defined class corresponds to a kind of model in the design environment.

The following steps are performed in the mapping:

1. For each fault manager (software) component model in the dataflow models, an associated state machine model that defines its mitigation strategy is located. This state machine model defines the behavior to be synthesized for each fault manager. In order to get the information of Machine we use the class *CMachineBuilder*
2. Given the behavior, the set of defined states is collected. The information of the states can be obtained by *CStateBuilder*. An **enum** construct is written into the source code.
3. Based on the trigger interface variables in the state model, a function prototype is defined for the state transition step function, with a parameter for current state and each of the trigger input and output variables. This function is used to compute next states and to read and write input and output messages.
4. Next, the body of the behavioral state transition step function is defined. For each state, a **case** segment is defined.
5. Within the **case**, the code is generated to implement the guard conditions, in the form of **if** clauses.
6. Within each of the **if** transition steps, the **action** code is inserted. This is based on the action attributes that the user specified when creating the behavioral model. The action shows creation of a message followed by a conditional transmission of the message to another behavioral process.
7. Steps 4-6 are repeated for each **state**, **guard,** and **action** specified in the model.
8. Steps 1-7 are repeated for each physical resource in the models.

This generated code is compiled and linked with the dataflow code generated to create a set of executable models for the system.

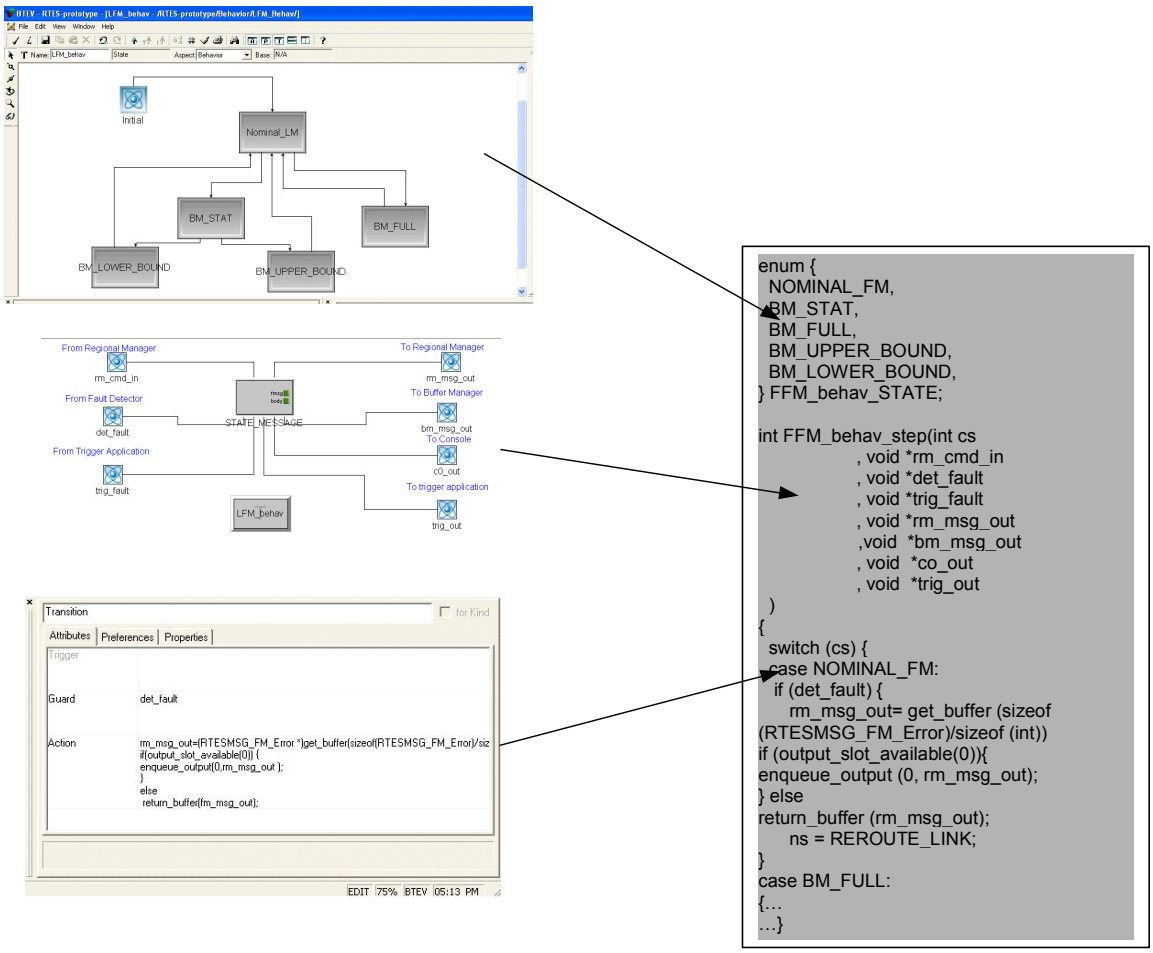


Figure 31 Mapping of Generated code and the models

Evaluation Of The Case Study

The tools for modeling analyzing and synthesizing large scale parallel fault adaptive real time systems have been developed and prototyped using the Model

Integrated Computing infrastructure. We see the following advantages of the tool set and design philosophies.

1. The concept of state machine being common and well known, its easy to model the fault mitigation strategies using this concept.
2. Software and hardware sub modules of the application could be designed in the same integrated framework.
3. The ease of setting up different fault manager's behavior and generation of the same by click of a button.

Finally, we see that a single framework is sufficient to design, implement, synthesize and verify a large scale embedded system application making the development cycle much shorter while improving the quality of the developed application.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

Conclusions

Software development for real time embedded systems can be difficult, as these systems are part of a physical environment, with complex dynamics and stringent timing requirements. Fault tolerance and reliability requirements further complicate these systems. This field is growing fast, with ever increasing design and development needs. The currently available support tools are not able to support the anticipated growth in this field. With the ever-increasing complexity of embedded systems, an integrated framework for design and development of these systems is needed in order to speed up the design cycle and to explore various alternative solutions.

BTeV is a prototype framework which provides an integrated environment to design reliable, large scale real-time embedded system applications using domain-specific languages and concepts. This environment will help the systems designers quickly build their applications without requiring deep expertise in the area of real time embedded systems. Using a MIC approach, we developed a robust environment that can provide a solid platform to support research in large-scale, fault mitigative systems. Information hiding and data abstraction is also achieved by the use of multiple aspects.

In addition to the domain-specific graphical language (modeling environment); model interpreters were developed to generate code for system operation and on-line fault mitigation. No restrictions govern strict homogeneity of individual levels of managers in

the network. Behaviors for different levels of fault managers are based on the physical proximity to faults, underlying architecture, system mode, or on specific application.

The tools for modeling, analyzing, and synthesizing large-scale, parallel, fault adaptive real-time systems have been developed and prototyped using the Model-integrated computing infrastructure at Vanderbilt University. These tools were used to model and synthesize a scaled down representative prototype of BTeV system. The prototype contained 16 embedded DSP processors. These processors were configured in an application-specific topology to reflect the dataflow of the BTeV trigger system. There were approximately 20 concurrently executing processes, with around 100 interconnections.

Several relatively simple fault mitigation behaviors were implemented. These behaviors ranged from a simple replication of a fault status message up the hierarchy, to analyzing a parameter and adjusting algorithm characteristics. The fault-mitigation behaviors took inputs from the kernel, the user interface, and the hardware monitoring devices. The actions taken by the behaviors ranged from message formation for user notification, simple algorithms, to resetting failed tasks etc.

These tools for modeling, analyzing, and synthesizing large-scale, parallel, fault adaptive real-time systems were used to model and synthesize a scaled down representative prototype of BTeV system. The prototype contained 16 embedded DSP processors, configured in an application-specific topology to reflect the dataflow of the BTeV trigger system. There were approximately 20 concurrently executing processes, with around 100 interconnections.

Several relatively simple fault mitigation behaviors were implemented. The fault-mitigation behaviors covered a wide range of system behaviors. Results and functionality of the prototype proved the efficacy of the tool-based approach.

The tools allowed full generation of all executable code. No by-hand modification was necessary. The time to modify a behavior and implement it across the entire array of processors was approximately 10 minute cycle. This represents a very large reduction in the effort and time required to adapt the system behavior. As a prototype, the framework is not complete and needs more effort to transform it from a research tool to a commercial-quality tool.

The modeling language was reviewed by practitioners in the high energy physics community, with a very detailed and extensive evaluation produced. Aside from several software engineering details, the concepts were deemed powerful and appropriate to the physics application domain. While some details will take training to become natural to the tool user, the basic concepts in the modeling language were natural to the domain. The prototype was demonstrated successfully at Supercomputing 2003.

Future Work

Several improvements and several new areas related to this research needs to be explored.

1. Real time behavior of the system needs to be supported to a greater depth by the tool. Adding a temporal behavior to the state machine for reconfiguration specification is a first step
2. Model checking and reachability analysis should be done in order to ensure model correctness.

3. Greater use of simulation will enable a more rapid design cycle, and enable design of systems for which the hardware is not yet available.
4. Modeling to date has focused on the behavior of the mitigation actions. In order to predict operation of the system, and possibly prove stability, the underlying 'plant' (i.e. computations, data throughput, etc) should be modeled. A hybrid model would be appropriate for this representation.

Currently the capability of the tool has not been fully utilized. The fault behaviors of the managers are relatively simple. Prototype sizes have been limited (16 vs. 2500). The tools should be used to model, generate, and analyze complicated behaviors, coupled with extensive performance measurements on real hardware.

REFERENCES

[1] Flavin Cristian,"Understanding Fault Tolerant distributed systems, "*Communications of ACM*, vol. 34, pp. 56-78, February, 1991.

[2] L. A. Cortes, P. Eles, and Z. Peng, "A Survey on Hardware/Software Codesign Representation Models", *SAVE Project Report*, Dept. of Computer and Information Science, Linköping University, Sweden, June 1999.

[3] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vicentelli, "A Formal Specification Model for Hardware/Software Codesign," *Technical Report UCB/ERL M93/48*, Dept. EECS, University of California, Berkeley, June 1993.

[4] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, pp. 231-274, June 1987.

[5] C. G. Cassandras, "Discrete Event Systems: Modeling and Performance Analysis", *Irwin Publications*, Boston, MA, 1993.

[6] E. A. Lee, "Modeling Concurrent Real-Time Processes using Discrete Events," *Technical Report UCB/ERL M98/7*, Dept. EECS, University of California, Berkeley, March 1998.

[7] J. Peterson, "Petri Net Theory and the Modeling of Systems", *Englewood Cliffs, NJ: Prentice-Hall*, 1981.

[8] G. Dittrich, "Modeling of Complex Systems Using Hierarchical Petri Nets," *Codesign: Computer-Aided Software/Hardware Engineering*, J. Rozenblit and K. Buchenrieder, Eds. Piscataway, NJ: IEEE Press, 1995, pp. 128-144.

[9] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing", *Transactions on Computers*, C36 (1): 24-35, January 1987.

[10] Sztipanovits J., "Engineering of Computer-Based Systems: An Emerging Discipline", Proceedings of the IEEE ECBS'98 Conference, 1998.

[11] Nordstrom G., "Metamodeling – Rapid Design and Evolution of Domain-Specific Modeling Environments", Proceedings of the IEEE ECBS '99 Conference, 1999.

[12] Bapty T., Neema S., Scott J., Sztipanovits J., Asaad S., "Model-Integrated Tools for the Design of Dynamically Reconfigurable Systems", *VLSI Design*, 10, 3, pp. 281-306, 2000.

[13] Agrawal A., Bakshi A., Davis J., Eames B., Ledeczi A., Mohanty S., Mathur V., Neema S., Nordstrom G., Prasanna V., Raghavendra, C., Singh M., "MILAN: A Model Based Integrated Simulation Framework for Design of Embedded Systems", Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES), Snowbird, UT, June, 2001.

[14] Ledeczi A., Maroti M., Bakay A., Nordstrom G., Garrett J., Thomason IV C., Sprinkle J., Volgyesi P., "GME 2000 Users Manual (v2.0)", Institute For Software Integrated Systems, Vanderbilt University, December 18, 2001.

[15] Buttler J.N., et. al, "Fault Tolerant Issues in the BTeV Trigger", FERMILAB-Conf-01/427, December 2002.

[16] Kwan S., "The BTeV Pixel Detector and Trigger System", FERMILAB-Conf-02/313-E, December 2002.

[17] Avizienis A., Avizienis R., "An immune system paradigm for the design of fault-tolerant systems", Presented at Workshop 3: Evaluating and Architecting Systems for Dependability (EASY), in conjunction with DSN 201 and ISCA 2001, 2001.

[18] Avizienis A., "Toward Systematic Design of Fault-Tolerant Systems", IEEE Computer, 30(4):51-58, April 1997.

[19] Bapty T., Scott J., Neema S., Sztipanovits J., "Uniform Execution Environment for Dynamic Reconfiguration", Proceedings of the IEEE Conference and Workshop on Engineering of Computer Based Systems, pp.181-187, Nashville, TN, March, 1999.

[20] Z.T.Kalbarczyk, R.K.Iyer, S.Bagchi, K.Whisnant, "Chameleon: A software infrastructure for adaptive fault tolerance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 6, pp.560-579, June 1999

[21] J.RumBaugh, I.Jacobson, and G.Booch, "The Unified Modeling Language Reference Manual", Addison-Wesley, 1998.

[22] Edward A. Lee, "What's Ahead for Embedded Software?", *IEEE Computer Magazine*, pp. 18-26, September 2000.

[23] W. T. Chang, S. Ha and E. A. Lee, "Heterogeneous Simulation - Mixing Discrete-Event Models with Dataflow", *Journal of VLSI Signal Processing*, Vol. 15, pp. 127-144, 1997.

[24] S. Neema, "System Level Synthesis of Adaptive Computing Systems", *Ph. D. Dissertation*, Department of Electrical and Computer Engineering, Vanderbilt University, May 2001.

[25] http://www.ilogix.com/products/magnum/index.cfm, Statemate MAGNUM, I-Logix Inc., Andover, Ma.01810.

[26] E. A. Lee, http://ptolemy.eecs.berkeley.edu/~eal/ee290n/glossary.html, EE290N: Advanced Topics in System Theory, Fall, 1996.

[27] A. Agrawal, et al. "MILAN: A Model Based Integrated Simulation Framework for Design of Embedded Systems", *Workshop on Languages, Compilers, and Tools for Embedded Systems* (LCTES 2001), Snowbird, Utah, June 2001.

[28] Matt Stephens, *Automated Code Generation*, 2002. http://www.softwarereality.com/programming/code_generation.jsp

[29] A. Kalavade, E. A. Lee, "A Global Criticality/Local Phase driven Algorithm for the Constrained Hardware/Software Partitioning Problem", *Proc. of Codes/CASHE'94, Third Intl. Workshop on Hardware/Software Codesign*, pp. 42-48, Sept. 22-24, 1994.

[30] Edward A. Lee, "Overview of the Ptolemy Project", *Technical Memorandum* UCB/ERL M01/11 March 6, 2001.

[31] A. Kalavade, Edward A. Lee, "Design Methodology Management For System-Level Design", Ptolemy Miniconference, March 10, 1995.

[32] Fussel D. and R. Isermann(2000). Hierarchical Motor Diagnosis Utilizing Structural Knowledge and a Self Learning Neuro-Fuzzy Scheme. *IEE Transactions on Industrial Electronics 47, no. 5:pp1070-1077*

[33] Scott J., Neema S., Bapty T., Abbott B., "Hardware/Software Runtime Environment for Dynamically Reconfigurable Systems", ISIS-2000-06, May, 2000.

[34] Kwan S., "The BTeV Pixel Detector and Trigger System", FERMILAB-Conf-02/313-E, December 2002.

[35] Nordstrom S.: A Runtime Environment to Support Fault Mitigative Large-Scale Real-Time Embedded Systems Research, *Master's Thesis, Vanderbilt University, Electrical and Computer Engineering, May, 2003*.

[36] Booch, Grady. 1994. *Object–Oriented Analysis and Design with Applications*. Addison-Wesley

[37] Dennis J.,"First Version data flow procedure language,"Massachusetts Institute of Technology Lab Computer Science Technical Memo MAC TM61, May 1975.

[38] Najjar W.,Lee E.,Gao Guang, "Advances in the dataflow computational model," Journal of Parallel Computing ,pp.1907-1929,vol.25,1999.

[39] Miro Samek. *Practical Statecharts in C /C++*. CMP Books, 2002. 2.2.2,2.2.4,5.3.1,5.3.2

ELECTRICAL ENGINEERING

TOWARDS DEVELOPING TOOLS AND TECHNOLOGIES FOR MODELING FAULTS IN LARGE SCALE REAL TIME EMBEDDED SYSTEMS

SHWETA SHETTY

Thesis under the direction of Dr. Theodore Bapty

The software development for real time embedded systems is widely acknowledged as a difficult undertaking. Certain classes of RT systems, such as high-energy physics trigger systems employ very large numbers of processors that must operate consistently over several months. A "reasonable behavior" is expected from these systems when the hardware or the software components fail or when faults occur. This class of large-scale real-time embedded systems has a need for a highly customizable fault-mitigation framework that includes high-level design tools.

This thesis presents a high-level tool for specifying the fault behavior which is model based using domain-specific graphical language (DSL). The DSL is implemented within the Generic Modeling Environment (GME) tool, which is a meta-programmable modeling environment, developed at ISIS, Vanderbilt University. The Fermi lab's proposed BTeV trigger system is being used as a target application driving the research and evaluation of the tools.

Approved_____ Date_____