A RUNTIME ENVIRONMENT TO SUPPORT FAULT MITIGATIVE LARGE-SCALE REAL-TIME EMBEDDED SYSTEMS RESEARCH

By

Steven Gregory Nordstrom

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, 2003

Nashville, Tennessee

Approved: Date:

_____ _____

_____ _____

To my source of Inspiration,

My Family

ACKNOWLEDGEMENTS

The research conducted 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.

First, I would like to thank Dr. Theodore Bapty, for the opportunity to work on this project, which has increased my knowledge of embedded systems tremendously. Also deserving of my thanks is Dr. Sandeep Neema, whose body of work forms much of the foundation on which this project sits.

To my father, Dr. Greg Nordstrom, who from the very beginning has fostered my endeavor for knowledge, I give thanks. This accomplishment could not have been achieved without the love, help, and knowledge I have received from him.

My deepest love and gratitude goes to my beautiful wife, Hope, who continually encourages me to march toward my goals and never give up. None of this would be possible without her love and support.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

LIST OF ABBREVIATIONS

ISIS – Institute for Software Integrated Systems

BTeV - B physics at the TeVatron.

FNAL - Fermi National Accelerator Laboratories

RTES - Real-Time Embedded Systems

DSP - Digital Signal Processor

FPGA - Field Programmable Gate Array

FSM - Finite State Machine

SDF - Synchronous Data Flow

DFG - Data Flow Graph

MIC - Model Integrated Computing.

GME - Generic Modeling Environment

API - Application Programming Interface

BON - Builder Object Network

ATR - Automatic Target Recognition

CBS - Computer-Based System

ACS - Adaptive Computing System

RTE – Real-Time Embedded

TMR – Triple Modular Redundant

CHAPTER I

INTRODUCTION

High-energy physics experiments use massive facilities to delve into the basic composition of matter. These experiments may run for several months at a time and produce data at rates that are on the order of several gigabytes per second. Systems to acquire and analyze data at this rate require thousands of processors and must be highly reliable. Historically, physicists have developed custom hardware and software solutions to perform the acquisition and analysis of their test data. In this scenario, however, the target system's complexity and budget constraints preclude a traditional development strategy. Tools to assist in the specification and design of such systems are necessary. The focus of this work is the development of a design environment for specifying and modeling the properties of fault mitigative, large-scale, real-time embedded systems from multiple aspects including application data flow, hardware resources, and failure mitigation strategies.

Review of the BTeV Experiment

One such system that will employ methods of fault mitigation is the BTeV experiment. The following is a summary of the motivations and challenges of the BTeV experiment currently under development at Fermi National Accelerator Laboratory. Information in this summary was gleaned from [23] and [24]. Only elements of the experiment relevant to the focus of this thesis are discussed, as there exists a vast amount of particle physics theory and research involved in the creation of the BTeV experiment.

BTeV is a physics experiment to be conducted at the Fermi National Accelerator Lab Tevatron whose goal is to study charge-particle violation, mixing, and rare decays of particles known as beauty and charm hadrons [24]. The BTeV experiment will exist inside a particle accelerator where the collision of protons with anti-protons can be recorded and examined for detached secondary vertices from charm and beauty hadron decays. The proposed BTeV detector layout is shown in figure 1. The ultimate goal of the experiment is to learn about the anomalous decays in an attempt to explain the matter-antimatter discrepancy that exists in the universe today.

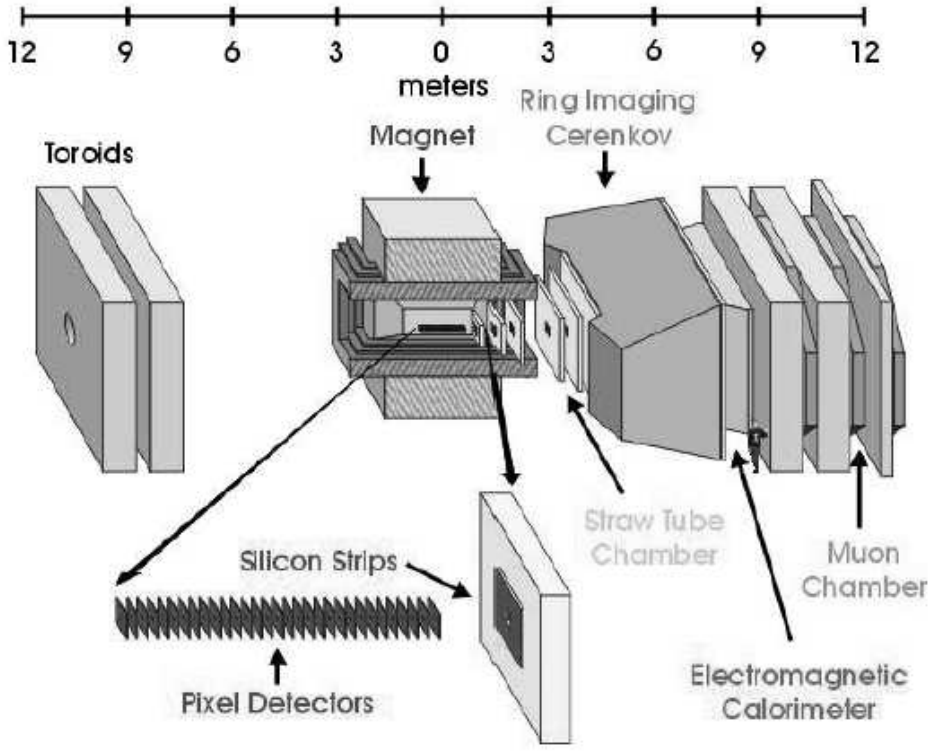

Figure 1. BTeV detector layout

The BTeV experiment employs the use of 30 planar silicon pixel detectors to record interactions between colliding protons and antiprotons in the presence of a large magnetic field. Geometry of the proposed pixel detector is depicted in figure 2. Protons and antiprotons arrive at detector from opposing directions, and the pixel detectors record the collision and subsequent particle interaction. These proton and antiproton streams are delivered to the interaction chamber at a periodic rate of 7.6 million collision/interactions per second (1 event every 132ns). The results from this reaction are carried via custom circuitry hardware to localized processors that reconstruct the 3-dimensional crossing data from the 30 silicon pixel detectors to examine the trajectories for detached secondary vertices.

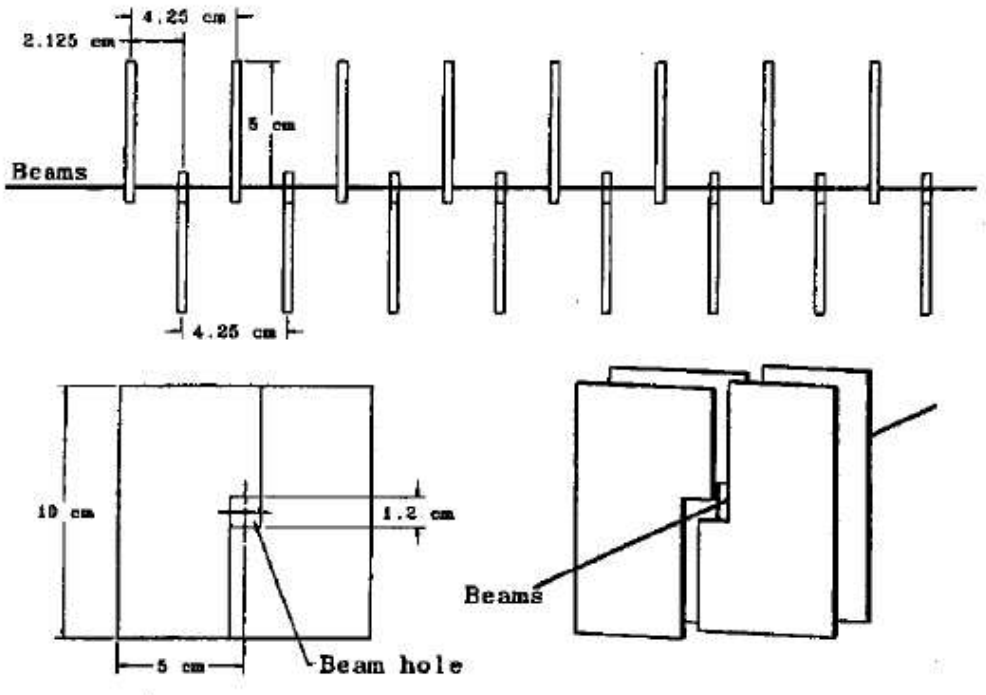

Figure 2. Proposed geometry of pixel detectors

The data sizes from interaction are on the order of 2Kbytes per event. Since these events occur at a rate of 7.6MHz, the aggregate data rate is clearly too high to be blindly recorded (it would take 12,800 disks of 100GB each to meet these demands for a single day). Instead, decision algorithms, called Triggers, must be executed online to dynamically compute an accept/reject decision. These algorithms necessarily must be computed in real-time, although significant queuing is typically available.

Detached vertices detected by the L1 trigger form the indication that beauty or charm decays are present in the event. Researchers at Fermilab estimate that 99% of the interactions recorded in the L1 chamber will be rejected by the L1 trigger, with close to 50% of accepted events actually containing beauty or charm decay. Data that passes the L1 trigger is then sent to a processing farm of commodity processors for offline processing using more stringent algorithms to confirm the presence of the decay [23][24].

Since the actual accepted events occur so infrequently, and the cost of operation of the experiment (facility, personnel, energy, etc.) is so high and demand for the facility is so large, the data system must be extremely reliable. When malfunctions occur, they must be able to be corrected very quickly. At the same time, the system cost must be minimized. The system to support the BTeV experiment will be extremely large and expensive. Budget limitations preclude approaches to add robustness to the system such as triple-mode redundancy, where components are replicated. Total system cost would go up significantly, more than 3x. The application would not permit the 'waste' of these resources. The solution needs to be both robust and cost effective. It is in these situations where the notion of fault mitigation over traditional fault tolerance plays an important role. A proposed solution provided by the BTeV group at FermiLab is shown in figure 3.

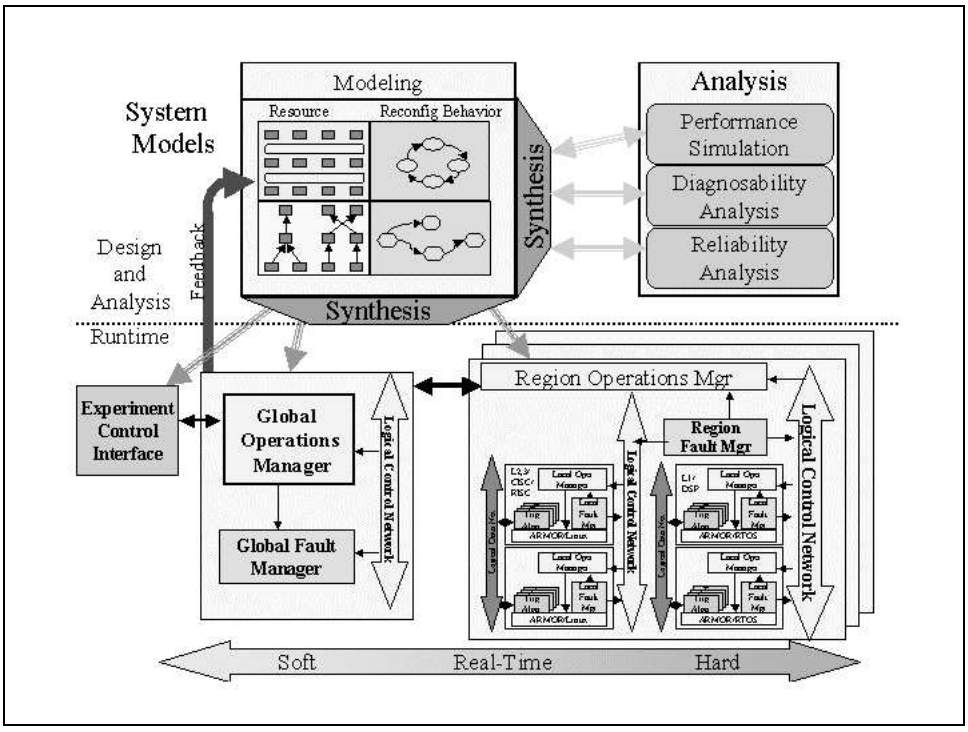

Figure 3. FermiLab's proposed BTeV solution

These constraints are what drive the research and development of toolsets to solve these complicated design problems. This is the basis for the work done at ISIS, Vanderbilt University toward achieving this goal. In order to test tools for the use in this application, prototype hardware and software must be created and tested. This thesis describes the work done to improve the ACS kernel (a real-time embedded kernel developed at ISIS) to support the creation and deployment of such fault mitigating systems.

Traditional Fault Tolerance

When designing such large-scale real-time embedded computer systems, one can be certain that at some point in time hardware will break, malfunction, or become generally unusable. Similarly, there is a chance that somewhere in the system the

software will become corrupted or hang unexpectedly. These occurrences are generally referred to as system faults. Large scale distributed systems need to be able detect failures and act accordingly without greatly affecting the reliability or the real-time performance of the system. Systems that are designed to handle such faults are referred to as fault tolerant systems.

Every fault tolerant system has some resilience against failure, but some are more capable than others at handling these occurrences. The degree of fault tolerance can vary greatly depending on the specifications or constraints of the system.

Traditional fault tolerance usually involves some type of redundancy. Extra hardware or software is present in the system, ready to dynamically replace malfunctioning parts during runtime. At the hardware level, techniques for fault tolerance have traditionally included redundancy.

Traditional fault tolerance is mainly used where cost considerations are typically offset by the need for extreme resilience to failure, where system failure may have disastrous effects on resources or human life. Examples of such systems could include flight dynamics and avionics systems, missile guidance systems, and other systems where absolute guarantees about system reliability must be made.

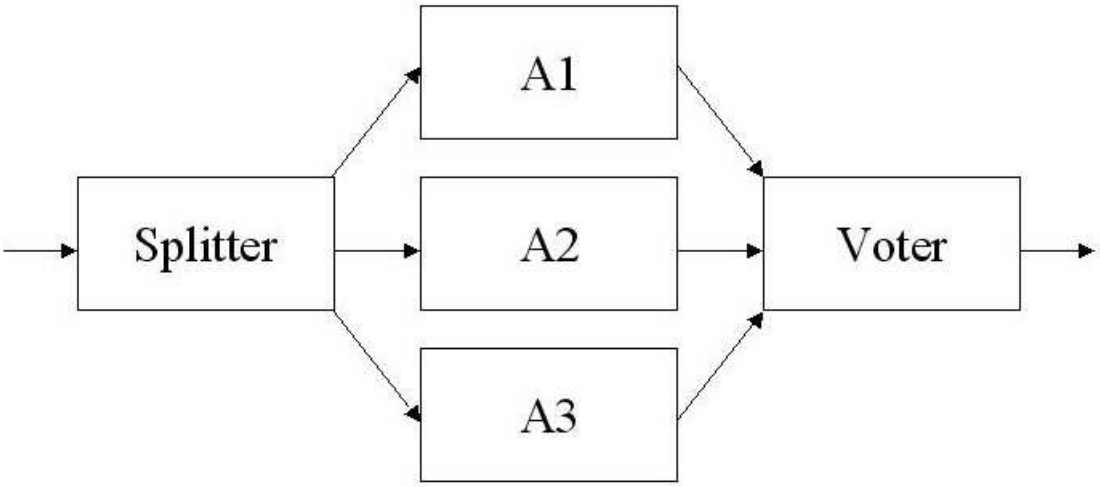

Figure 4. Traditional voter/splitter mechanism

In the splitter/voter mechanism shown in figure 4, data is carried to multiple processing destinations, where identical calculations are performed on the data in parallel. The data results are then demultiplexed in a voter, which examines the results. The algorithm of the voter is to simply examine the data results, and choose the results that appear in the majority, qualifying this answer to be considered correct. This method is very good for providing fault tolerant calculations, with the minimal latency expense of distributing and collecting the data. This algorithm is very costly in terms of hardware size and complexity [22].

To carry this technique to the fullest, redundant buses must also be deployed to carry the data safely and securely to the splitter, as well as from the voter to its final destination.

This method, of course, works well for small systems where hardware is cheap and memory is plentiful. There are, however, cases where redundant hardware is not the best solution. In high performance parallel systems, fault tolerance using conventional redundancy methods can be extremely costly, as system size and complexity increase. As systems get larger and more complex, a more cost effective way of providing some protection against hardware/software failures is needed.

In the domain of very large parallel systems, designers of embedded systems rarely have the luxury to employ such hardware-intensive tactics to make systems more robust to catastrophe. In these cases, Software must play the dominant role in achieving fault tolerance. Notably, the embedded software world is discovering that the development of high quality embedded systems is only possible through the use of advanced software toolsets that can shift the onus of intelligent design away from humans, who are prone to miscalculation and error. This is especially the case where many complex factors drive design decisions. Research is proving that dependable, robust embedded systems can be developed using intelligent, automated design tools [1].

Fault Mitigation as a Design Methodology

Strategies for mitigating faults reflect the engineering trade-offs that need to be made when deciding how a system should behave in the presence of failure. Given that limited redundant hardware is present, when a failure occurs in a fault-mitigating system, some change in the system behavior must occur. For example, the total output of the system could be diminished, or the quality of the output might be degraded. This is one of the fundamental differences between traditional hardware-based fault tolerant designs and more flexible software-based fault mitigative ones. Software fault mitigation is

preferred over hardware fault tolerance when total system costs or architecture complexity rule out the use of traditional redundant methods for fault resilience and the application can accept diminished system performance under fault conditions.

Problem Statement

The goal of the research described in this thesis is to implement a runtime environment for software-based fault mitigation. The environment is to be used as a research tool for investigating issues surrounding the development of fault mitigative, large scale, real-time embedded systems. The primary concerns surrounding the implementation of this system are what facilities are needed to support a model-based approach to large-scale fault mitigation systems, how to provide a working testbed on which to implement software synthesis tools, what kinds of issues arise from scaling up to large processor-count systems, and what issues arise in the deployment toward physics applications. (The tools should allow physics researchers to configure and implement fault mitigating, real-time embedded systems without needing the design expertise of those in the RTE systems field.)

The final system should be capable of specifying the system in terms of hardware, application data flow, and fault mitigation agent behavior. Also supported should be automated generation and deployment of the runtime environment. The system should allow fault diagnosis in the form of failure scenarios initiated by the user during runtime. The tool should provide the ability to rapidly model, synthesize, and deploy a variety of systems for physics research purposes.

Chapter II begins by investigating the use of past design tools used to implement real-time embedded systems, followed by a literature survey regarding the use of

biological systems as models for fault mitigative systems. Chapter III details the modeling principles used to configure and specify the system. In Chapter IV, the implementation of the BTeV runtime environment is documented, which touches on aspects of the design relevant to the thesis topic. Chapter V details a prototype system built using the BTeV environment. Finally, Chapter VI discusses future work for the project, and draws some conclusions about the body of work detailed in this thesis.

CHAPTER II

LITERATURE SURVEY

Survey Of Embedded System Design Tools

This section reviews the existing research and tools for design of embedded systems and highlights their usefulness or deficiencies within the problem domain defined by this thesis.

MIC

Model-Integrated Computing (MIC) is a design philosophy used to build embedded systems in which domain specific modeling languages are used to fully specify a system from multiple aspects [4]. Hardware/software configurations, executable code, and system simulations can all be synthesized from the integrated set of domain-specific models [2]. The key element of this approach is the extension of the scope and usage of models such that they form the foundation of a model-integrated system development process. In Model-Integrated Computing models play the following central roles:

- Integrated, multiple-view models capture the information relevant to the system to be developed. Models can explicitly represent the designer's understanding of the entire system, including the information processing architecture, the physical architecture, and the environment it operates in.
- Integrated modeling allows the explicit representation of dependencies and constraints among the different modeling views.

- Tools analyze different, but interdependent characteristics of systems (such as performance, safety, reliability, etc.). Tool-specific model interpreters translate the information in the models to the input languages of analysis tools.
- The integrated models are used to automatically synthesize the software. The model-integrated program synthesis process utilizes model interpreters to translate the models into executable specifications.
- UML-based metaprogramming interface allows synthesis, evolution of domain-specific MIPS environments.

Using MIC technology one can capture the requirements, actual architecture, and the constraints of a system in the form of high-level models [9]. Models can be mapped to an implementation using a model interpreter, and other useful artifacts, such as configuration files or simulations can be generated from the system models. Evolution of the design environment can be achieved through the use of the meta-programming interface, which allows the domain specific modeling language to be refined as the needs of the domain change.

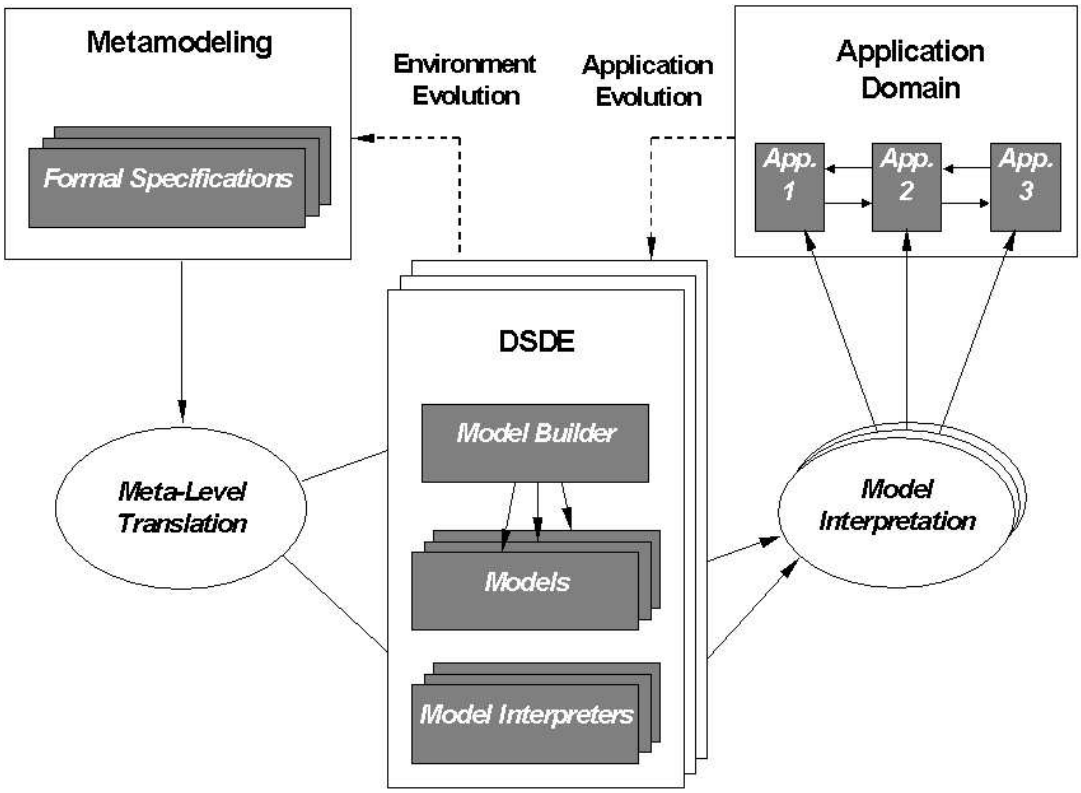

Figure 5. Design evolution using model integrated computing

A metaprogrammable application called the Generic Modeling Environment (GME) is a tool that uses the MIC methodology. It provides an environment for creating domain-specific modeling languages and environments [13]. The specific tool used to model and configure the BTeV runtime environment is GME 2000.

ACS-MIDE

The Adaptive Computing System Model Integrated Development Environment (ACS-MIDE) is an integrated set of tools for design capture, analysis and synthesis of dynamically adaptive computing applications [3]. The representation methodology is captured in a domain-specific, model-integrated computing framework. Formal analysis tools are integrated into the design flow to analyze the design space to produce a

constrained set of solutions. From the models a set of hardware and software subsystems are synthesized to implement multi-modal, dynamically adaptive applications.

One such application developed using the ACS-MIDE is the Automatic Target Recognition (ATR) for missile systems [3]. The ATR application demonstrates the ability of the ACS-MIDE to design and generate dynamically adaptable embedded systems. Several aspects of the ACS-MIDE may eventually play an important role in the BTeV solution such as using Ordered Binary Decision Diagrams (OBDD) augmented with progressive scan state-space pruning techniques to solve design space exploration problems [12].

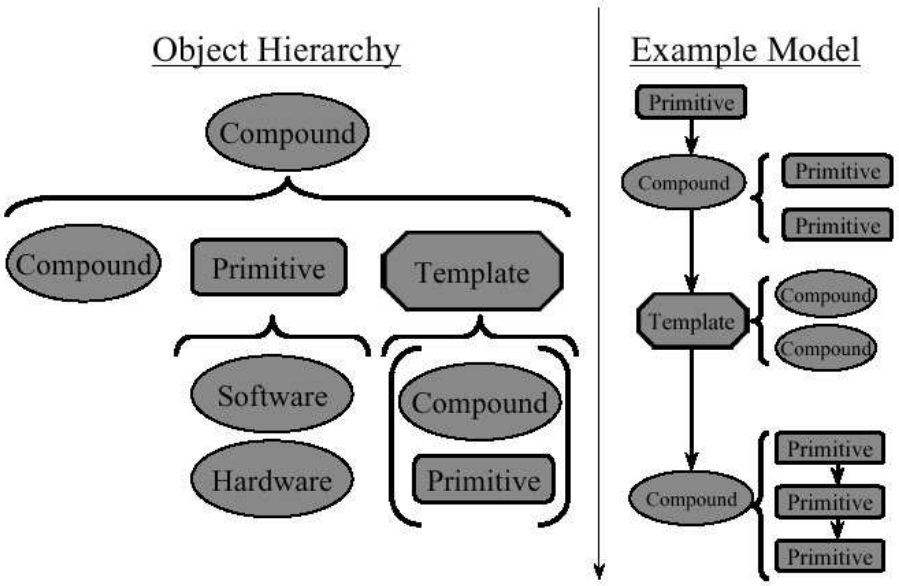

Figure 6. Object hierarchy in ACS paradigm

Several aspects of the ACS-MIDE directly relate to the BTeV runtime environment. One such relation is the use of data flow and structural modeling to configure an adaptive runtime environment. The hierarchy of such models is shown in

figure6. The use of common execution semantics [8] between software dataflow models (processing blocks, streams, queues) and structural system components (hardware processes, FIFO buffers) allow the unified set of models to appear familiar to both software designers as well as hardware engineers. Figure 7 illustrates the relationship between software execution semantics and hardware execution semantics in the ACS-MIDE.

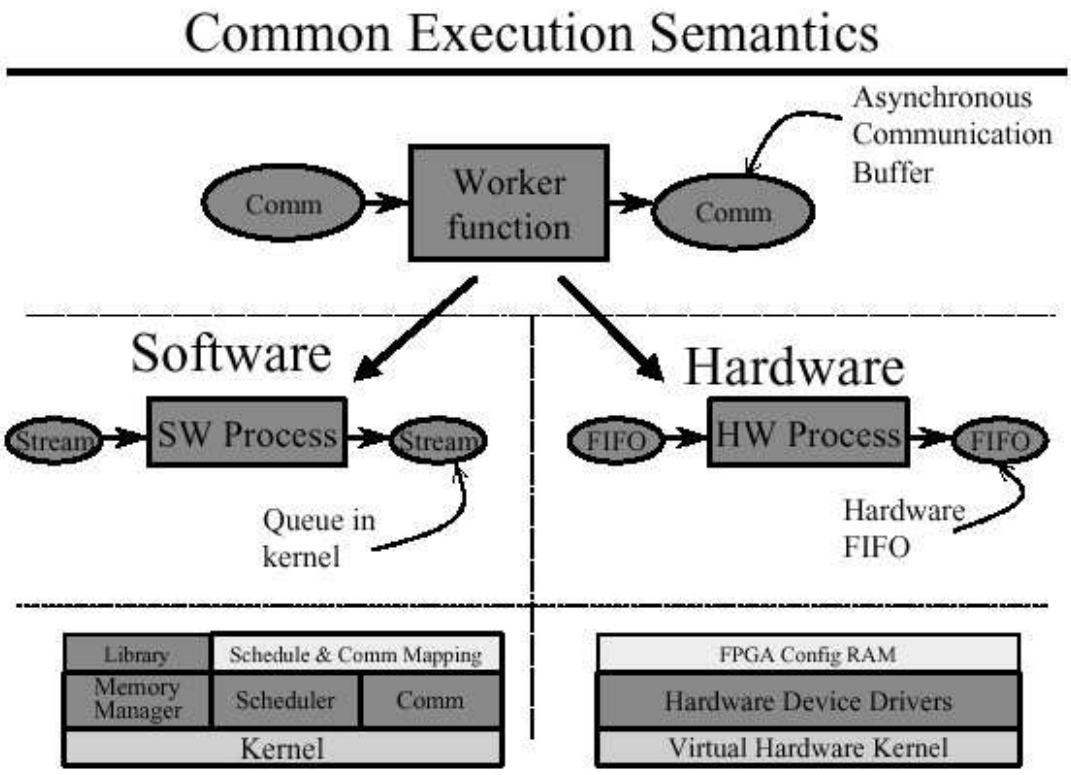

Figure 7. ACS-MIDE common execution semantics

The model-integrated approach used in the ACS-MIDE tools has been designed to support the many aspects and disciplines of embedded systems design. The flexible representation, along with synthesis and analysis of systems helps to reduce design effort

and increase system flexibility. The underlying runtime environment, through the abstraction of hardware and software details, presents a uniform architecture for system synthesis and application implementation [6].

MILAN

The Model-based Integrated simuLAtioN framework (MILAN) is a design and simulation framework to facilitate rapid development of efficient heterogeneous embedded system applications [10]. The framework is extensible and uses Model Integrated Computing (MIC) to drive a variety of common off-the-shelf (COTS) simulators, code generation engines, and design evaluation tools. The project is currently being developed at ISIS at Vanderbilt University and University and Southern California.

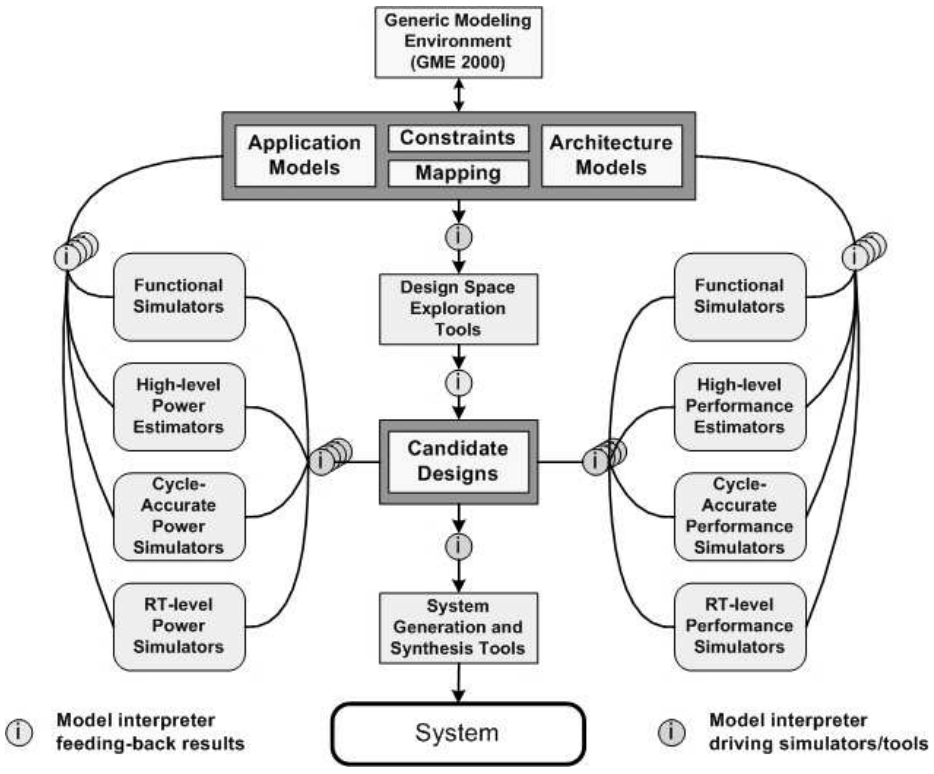

Figure 8. MILAN design environment

Design goals of MILAN are to integrate the design, development and simulation of embedded system applications into a unified environment, and are depicted in figure 8 [10]. Much of the MILAN project is applicable to the ultimate goals of the BTeV project. System simulations, while planned for the BTeV project, do not appear in the environment as of this date. However, more investigation into the ability to generate simulations from the models is necessary to determine how best to integrate this capability into the existing BTeV environment.

Human Biological Systems as a Model for Fault Mitigative Systems

This section touches upon some of the existing research regarding the use of biological models as a basis for fault tolerant systems. Methods found in this area of research are used in the BTeV hardware runtime, specifically the Reflex Action table and the Healing Interface.

Only recently has the notion regarding biological systems as a model for distributed fault mitigation been investigated. Among the earliest references to biological systems as guides for fault tolerance in embedded systems was by A. Avizienis [18][19]. Since these references, he has explored the use of biological system models to aide in the development of embedded fault tolerant hardware components.

Even as early as the 1997, researchers realized the importance of degraded mode operation, as highly parallel systems were already pushing the limits of fault tolerant technology. This fact helps solidify the notion that fault mitigation is separate from traditional, strict, fault tolerant design [21].

More recent advances using biological systems as models for fault tolerance are strictly hardware-centric [15][16]. Bradley envisions the creation of an artificial hardware

immune system through evolvable hardware and embryonics, both of which are ideas inspired by biological systems. In [20] Thomas Becker recognizes the need for transparent fault tolerance that supports a variety of policies, but fails to address the needs of real-time embedded community, as his solution is implemented purely in software. His solution is also purely redundant one, not constrained by limited resources (traditional fault tolerance relies on redundant hardware, as opposed to fault mitigation which does not).

Clearly, the desired solution lies somewhere between a pure hardware or software implementation. With the boundary between hardware and software management of faults being blurred, the role of assigning responsibility for handling faults becomes increasingly difficult. It is therefore necessary for tools to assist (and automate where appropriate) the design and evolution (no pun intended, really) of such increasingly complex systems. The research done at ISIS for the BTeV runtime has investigated using the use of biological reflex and healing systems as a model for building fault mitigative systems. The correct solution must spread the responsibility of failure recovery between both hardware and software.

Main aspects of Human reflex and recovery systems are the following [19][17][15]:

- A brain to coordinate action
- Nerve paths to propagate messages of stimuli to the brain
- Tissues to be affected by stimuli
- Reflex reactions to stimuli (these are “pre-wired” and spontaneous e.g. not commanded by the brain)

- Degraded modes of operation when tissues are damaged
- Long-term healing and recovery actions dictated by the brain to ensure damaged tissue recovers and the body can get well again

These aspects would be translated to the following requirements for an embedded system:

- A global management unit to coordinate actions (High-level modeling system, model interpreters, etc.)
- Message paths to propagate messages to and from the global manager, and within the fault mitigation infrastructure
- Hardware resources to be affected by failures
- Reflex or pre-programmed reactions to certain hardware failures
- Degraded algorithms or discarded data due to loss of resources
- Long term schedules of actions to get hardware resources up and running again

From this review, one can find that biological systems act in a distributed manner and are capable of mitigating failure, and that their use as a model for fault tolerance is being explored in within the realms of hardware design and distributed software. However, little research is being done to apply biological system models toward the development of real-time embedded systems, where the line between hardware and software mitigation responsibility is less clear. Note that we are not attempting to imitate the internal functions of biological systems, rather make an analogy with the properties of a living body.

Summary

Several solutions exist for the modeling and creation of real-time embedded systems. The ACS-MIDE uses MIC to model the desired system using hardware (structural) and software (data flow) models. MILAN also employs the use of MIC, however it extends these methods to provide modeling of VHDL and SystemC components. MILAN also extends the traditional design environment to include simulation and design space exploration, both of which are relevant to the BTeV project [11]. Finally, while the use of biological systems as a model for fault tolerance is being investigated, the scope of the current research is limited to hardware-based systems, which neglects the needs of the distributed real-time embedded community.

Missing from each of these approaches is the ability to use models to describe not only hardware and software in the system, but also fault mitigation behaviors. Because of this, the ability to synthesize the fault mitigating systems from behavioral models is also missing. Of course, this capability cannot be realized without support in the runtime execution environment for the necessary fault mitigating actions required by the generated system, which is the topic of this thesis.

CHAPTER III

MODELING IN THE BTeV ENVIRONMENT

Structural Modeling

Available hardware resources are captured in the hardware aspect of the system model. These resources are available to any software component currently modeled in the system. The hardware models are hierarchically decomposed, and support the use of types and instances to preserve the correctness of lower level hardware models (instances of types cannot be modified by the user of the environment).

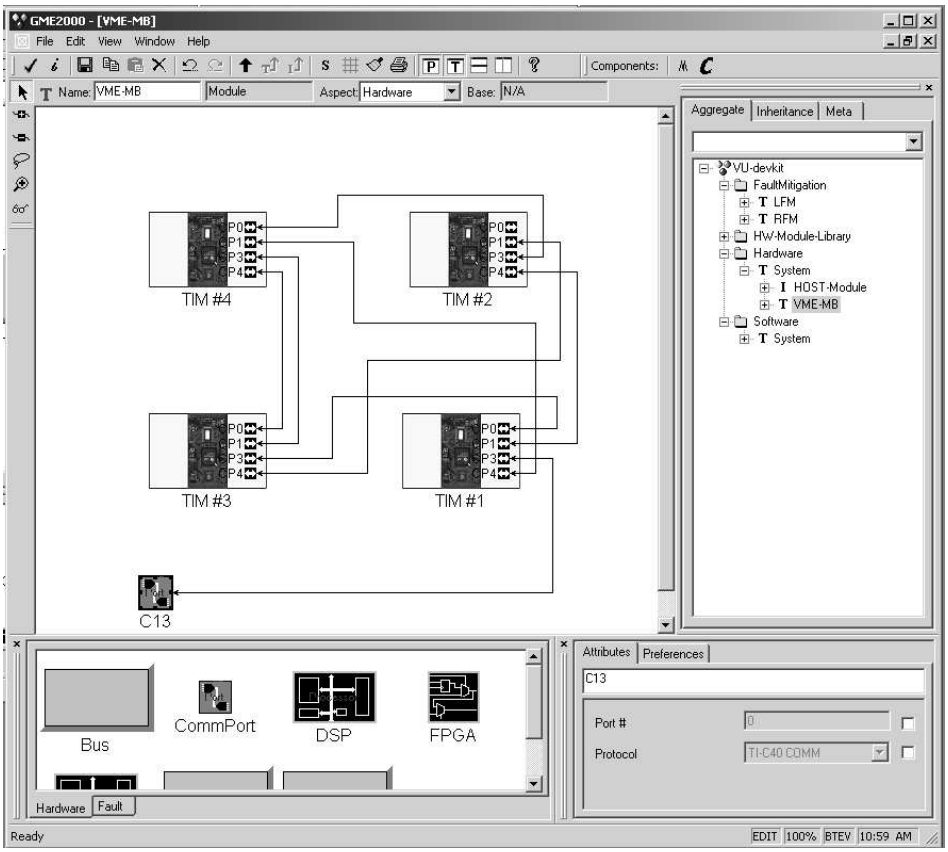

Figure 9. Structural model of a VME-Motherboard

The hardware model shown in the figure above is the specification for a single motherboard consisting of four TIM cards. The models are hierarchically decomposed, so focusing down into the TIM model would reveal the inner component models contained within.

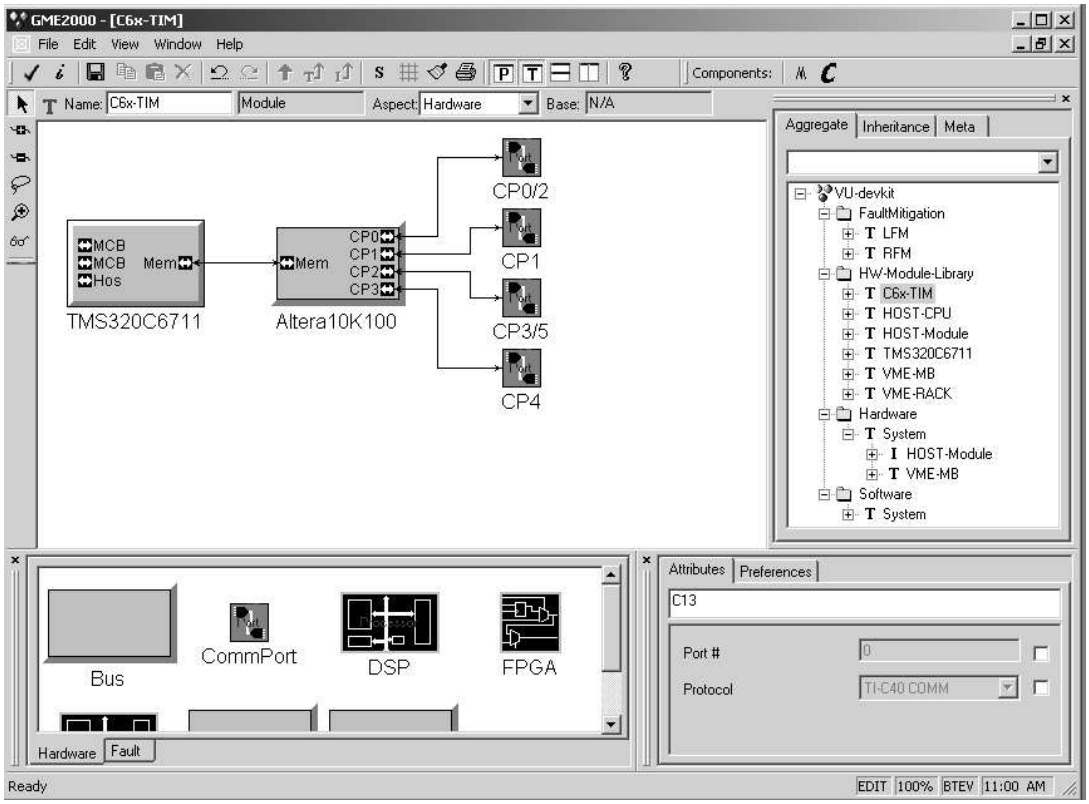

Figure 10. Model of a C6x-TIM

Figure 10 shows the decomposition of a C6x-TIM DSP module, complete with models representing communication ports, FPGA logic, and a TMS320C6711 processor. The modeling environment supports an arbitrary number of hierarchical levels. This is helpful for modeling large-scale systems, as blocks can be easily modeled and manipulated which contain hundreds or thousands of components [2].

Hardware Module Library

Models of domain-specific hardware components may remain fairly static, that is, unchanged, for the lifetime of the development cycle. This is true in the case of the BTeV environment with regards to the models for DSP modules (C6x-TIM) and several other hardware components. For these cases, a hardware model can be added to the Hardware Module Library, in which copies of hardware model compositions can be kept without fear of affecting the final system generation. Hardware components in the library are also hierarchical, and copies of any hardware component modules can be dragged in to the current design from the library and utilized with little effort. This feature is beneficial to the system designer, especially in cases where the designer is not an expert in composing efficient large-scale embedded systems. Modules can be created by the embedded systems expert end imported into the GME with little effort as well.

Software DataFlow Modeling

Data flow models capture the software requirements of the system. These data flow models are derived from those used in the ACS-MIDE and are made to closely resemble a directed data flow graph [7]. Inter- and intra-process communications are represented as streams and are assumed to be asynchronous.

Software primitives represent a single computational block or algorithm to be executed in the runtime system. Attributes for the software primitive specify a script name for the component, as well as the implementation file in which the script can be found. Software Compounds are similar to Primitives, with the exception that a Compound can contain other Primitives or Compounds.

Ports provide a means for connecting two software components with a stream. Ports also provide a mechanism to support hierarchical decomposition with the software dataflow model. More detailed discussion about software data flow modeling can be found in the ACS-MIDE publications [5][8].

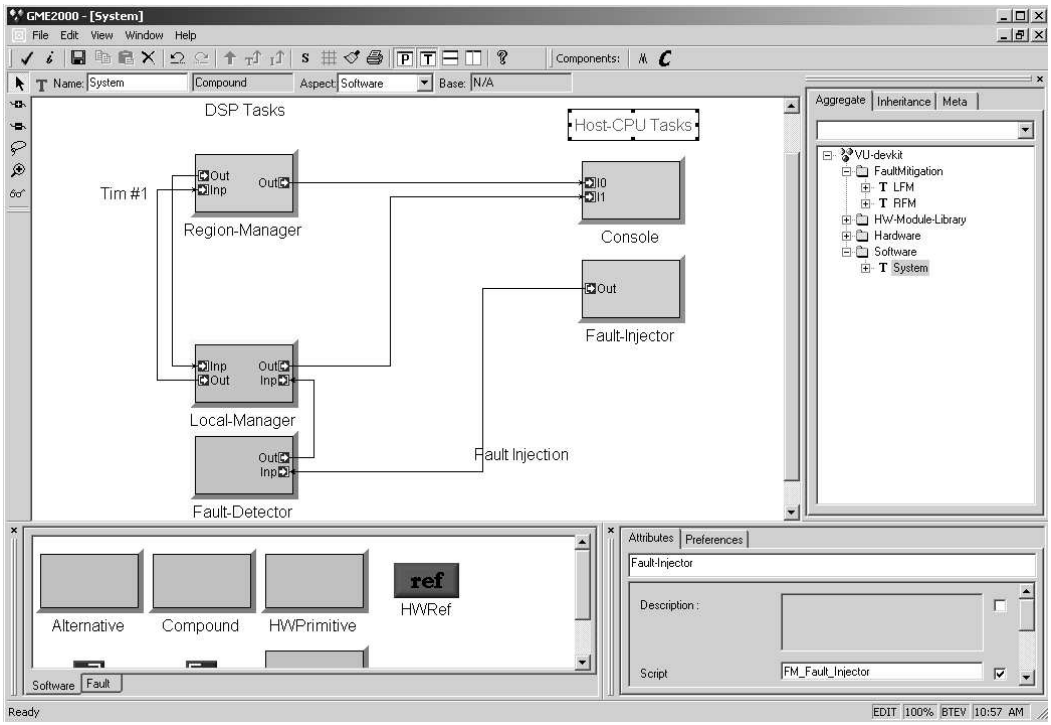

Figure 11. Data-flow modeling

Software processes are assigned to run on a specific hardware resource by placing a reference to the target hardware within the software primitive or compound, as shown in figure 11. Process tables and schedules for each node in the network are synthesized from these models.

Behavioral Modeling of Mitigation Processes

Fault mitigation behaviors are captured using a state-machine based paradigm that allows the specification of behaviors using Harel's StateCharts [25]. Mitigating actions are modeled based on the needs of the domain. The ease of modeling new mitigation behaviors using these models provides designers the much-needed room to evaluate alternative designs.

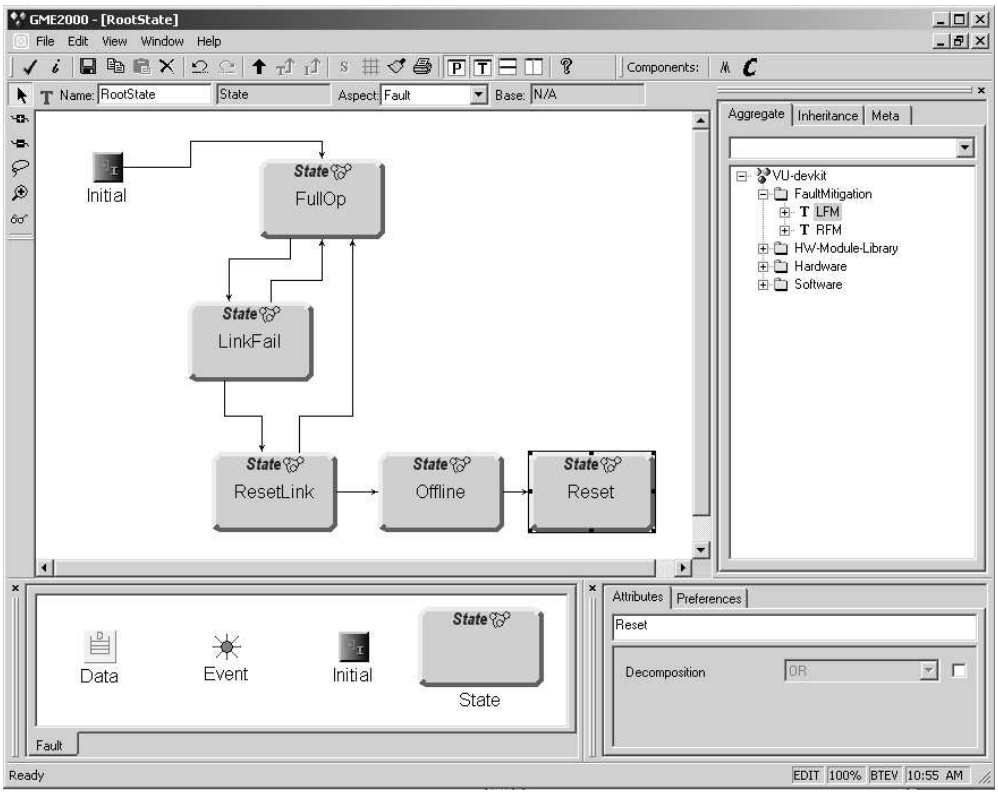

Figure 12. Behavioral models using modified StateCharts paradigm

Triggers, guards, and actions are specified as attributes of transitions. The paradigm utilizes a Mealy-based [25] state machine representation (shown in figure 12), where actions can only happen on the transition from one state to another. State parallelism is achieved through the use of AND and OR state decomposition. AND states

are states in which the contained state machines occur in parallel, OR states are states in which the contained states occur in series [25].

The resulting state machines are synthesized to form component software processes and are realized in the form of C code. These generated components can be used within the software models just like existing software modules. This behavioral modeling capability allows the system designer to specify which fault mitigating behaviors the system will exhibit.

System Generation

From these models, the system implementation can be generated [4][14]. Artifacts of the generated implementation include:

- Schedules and process tables for each kernel to be run on each of the available processors. These schedules define where and when each software component will execute across the hardware.
- Network routing, defining the data paths from one logical process to another. Streams are the implementation mechanisms that realize these routes, both internal to a processor and across the network. If processes are not on adjacent processors, a forwarder is created on each intermediate node.
- Local and Regional Fault Manager implementation: The code to implement the mitigation strategies must be synthesized from the behavioral models. Critical factors in the generated code are adherence to real-time requirements, maximizing resilience, and proper execution of the modeled behavior.

- Simulations: The adaptation of such a large system is complex. Even simple rules applied in a distributed system may result in unexpected behavior. Work is currently in progress toward generating a simulation that faithfully emulates the system behavior, and allows evaluation of fault mitigation strategies.

Subsequent to system generation, the runtime system can be initiated. The runtime system implements the processes specified in the software data-flow models, using available resources specified in the structural models, and will react to faults in the system in accordance with the specified fault mitigation behavioral models.

CHAPTER IV

THE BTeV RUNTIME ENVIRONMENT

In the previous chapters, we have observed some design methodologies supported by existing real-time embedded systems development platforms. This chapter will document components of the BTeV runtime environment and provide explanations of their functionality.

Requirements

To keep the development process manageable, tools must be available to support the development process. An ideal tool should:

- Allow designers to specify fault mitigation behavior in domain-specific manner (as opposed to a machine-specific method)
- Integrate application specification and design, since the application and its fault-mitigation behavior are closely linked.
- Permit specification of the target hardware architecture, along with the available hardware resources and redundancies.
- Support analysis of the design, predicting or simulating the expected behavior of these complex systems.
- Support direct synthesis of software and system configuration artifacts.

In order to support these high-level tools, an underlying virtual machine is necessary. The virtual machine is implemented in the runtime environment described below. The runtime environment consists of:

- A Fault Mitigation API: This provides a 'standardized' interface to the fault adaptation functions of the runtime.
- A Fault Mitigation Messaging infrastructure, to pass fault detection status messages and fault mitigation actions across the network.
- A Fault Mitigation Kernel: providing the fault message routing system, processor scheduling, process and communication fault adaptation, etc.
- A Fault Monitoring capability:
- A Fault-Injection system: the fault injection system is used to simulate hardware or software faults to test the fault behavior

Fault Mitigation Kernel API

The fault mitigation approach takes a reflex-healing strategy. By this, we mean that rapid reflex actions are programmed into the system to provide a quick-but-limited response. Subsequent 'Healing' actions attempt to rebalance the system and compute new reflex actions. The Fault Mitigation Kernel provides a limited 'Reflex' action for rapid solution of system faults. For a specific system state (the set of functional resources and the set of executing tasks), the fault mitigation system supports the guaranteed ability to correct any ONE error (i.e. reflex actions are programmed for each of the expected failures). It is possible (but not guaranteed) that other reflex actions can succeed as well, unless they share resources with already-executed actions.

In order to support certain mitigating behaviors such as dynamic, transparent communication rerouting, certain features had to be added to the ACS kernel. These features manifest themselves in the so-called Fault Mitigation API (FM-API).

The FM-API provides a way of managing and implementing the fault-mitigation reflex actions. It performs activities for downloading reflex specifications, interfacing to fault detectors, and implementing the desired reflex actions. The reflex actions are stored in tables (for link replacement in link or processor failure) and in code modules generated from state machine specifications (for more complex algorithms).

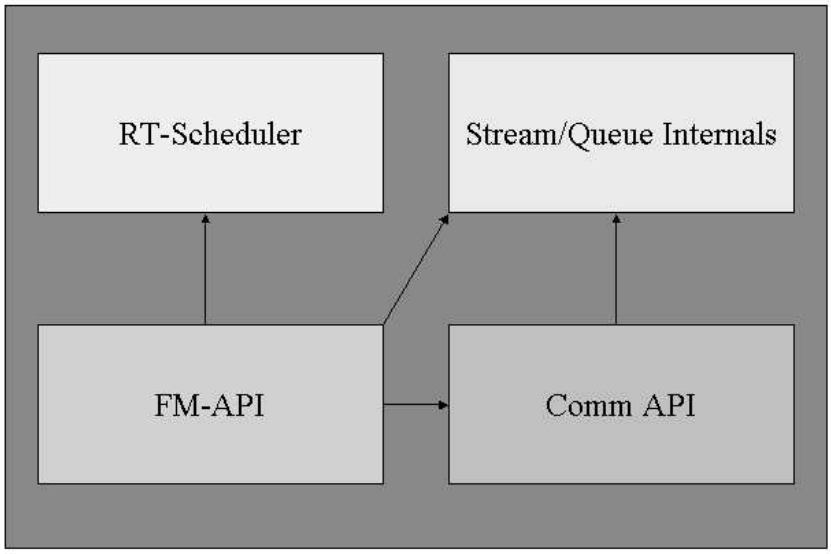

Figure 13. FM-API relationships in the modified ACS Kernel

Fault Mitigation Message Schema

Fault Mitigation Messages (FMM) are utilized to propagate notification of failures up through the FM component hierarchy, as well as reflex and healing actions back down to the component's healing interfaces.

In order to minimize bandwidth while providing the maximum flexibility, FM-Messages are specified to have variable length, based on the originator of the message. Table 1 shows the size and structure of the FM-Message header, while table 2 shows the size of each type of message as well as the payload contents. Figure 14 lists the code to implement the current message types, which are based on the message's origin.

Table 1. FM-Message header structure

Type	**Size**	**Contents**				
Header	12 bytes	msgSize	msgStatement	msgOrigin		

Table 2. Fault-Mitigation Message sizes and contents

Payload Type	**Size (B)**	**Contents**				
PHYS_APP	20	ID	CID	Nature	PC[]	TickCount
FPGA	12	CID	DSP-ID	Usage		
FARM_MGR	12	DSP-ID	Usage	Reason		
REGION_MGR	8	FARM-ID	Errline			

```
typedef struct {// message header
      unsigned int fmsgSize;
      FMsgStatement fmsgStatement;
      FMsgOrigin fmsgOrigin;
}FMsgHdr;
typedef struct {// Actual FMSG_DSP_Arch message structure
      FMsgHdr fmsgHeader;
      // DSP Software Arch Specific Elements
      unsigned int ID;
      unsigned int CID;
      unsigned int PC;
      unsigned int tickCount;
      unsigned short nature;
}FMSG_DSP_Arch;
typedef struct {// Actual FMSG_Physics_App message structure
      FMsgHdr fmsgHeader;
      // DSP Physics Application Elements
      unsigned int CID;
      unsigned short state;
      unsigned short numOfSegments;
      unsigned int stack[16];
}FMSG_Physics_App;
typedef struct {// Actual FMSG_FPGA message structure
      FMsgHdr fmsgHeader;
      // FPGA Specific Elements
      unsigned int CID;
      float percentUsg;
      unsigned int DSP_ID;
}FMSG_FPGA;
typedef struct {// Actual FMSG_Farmlet_Manager message structure
      FMsgHdr fmsgHeader;
      // Farmlet Manager Specific Elements
      float average;
      unsigned int DSP_ID;
      unsigned short reason;
}FMSG_Farmlet_Manager;
typedef struct {// Actual FMSG_Region_Manager message structure
      FMsgHdr fmsgHeader;
      // Regional Manager Specific Elements
      unsigned int farm_ID;
      unsigned int errAddress;
}FMSG_Region_Manager;
```

Figure 14. Code listing for FM-Message types

Runtime System

The runtime environment for the BTeV system is built atop an adaptive, real-time, embedded kernel (ACS). Communication (both inter- and intra-process) in the DSP network is achieved through queuing streams. The ACS kernel provides a stream-per-connection facility. Each of the streams implements an asynchronous logical queue. The logical queue can exist on a single processor or can span across two adjacent processors.

The connections between streams are typically configured at application load-time and remain static throughout the execution. In this work, the ability to change stream mapping is used to help implement fault mitigation. In the kernel, processes are scheduled in a round-robin fashion. Again, the typical use of the kernel is to configure the schedule at load-time, but here we use schedule reconfiguration for fault-mitigation.

Figure 15 shows the overall structure of the runtime system. The runtime assumes a high-level planning system that can configure applications, devise fault mitigation plans (programs), both as part of a design phase and dynamically, in response to detected faults. The interface to the high-level tools is dubbed the Healing Interface.

The runtime supports a hierarchy of fault mitigation 'managers'. The hierarchy provides some level of scalability to the system. Problems are detected and mitigated at the lowest level possible. This allows the actions to be more rapid (i.e. Reflex) and to ensure that the corrective action has as localized an impact as possible.

There is assumed to be only one Global Manager. The Global Manager interfaces with the synthesis tools and with any user interface or facility control system (via the System Interface). The global manager has control over the entire set of system resources. Global actions permit reallocation of resources during particularly large or serious fault conditions. In the BTEV example, the Global Manager will control ~2500 DSPs and possibly 2500 Linux workstations.

Regional Managers form the bulk of the fault mitigation network. Regional Managers can themselves form a hierarchy, as appropriate for the size and desired fault behavior of the system. They provide a way to implement gradually expanding mitigative actions. They also provide a path from the lowest level managers to the Global Manager.

Local Managers are the leaves of the graph. They interface directly to diagnostics and computational components, receiving fault reports. As possible the Local Managers will solve the problem locally and inform the parent Regional Manager of the problem and solution. When resources are not available, or the mitigation prescription is not local, the problem is passed up a level to the Regional Manager. Any mitigation actions that are developed at a higher level, are eventually passed down to the local manager for implementation.

Interactions between levels of hierarchy occur through both direct messaging from the mitigation engines, as well as through the Healing Interface. Mitigation engines calculate reflex actions to be taken when the detection of a fault occurs. When such a fault occurs, each node in the network takes a predetermined reflex action, and notice of both the fault and the resulting action are propagated up the hierarchy using the fault-mitigation message schema. These reflex actions are expected to occur within minimal time frames: Local Mitigation <1ms, Regional Mitigation 1-100 ms, Global Mitigation: 100ms-2sec.

Concurrent with the local, regional, and global reflex actions, fault information is passed up to the high level tools. At a longer time scale 2-100 sec, the tools will re-plan the system. When a solution as to how best to mitigate future faults is formulated (based on the application specific behavioral models), new reflex actions are propagated back down the hierarchy through the healing interface. Long-term healing actions, such as reallocating resources for more optimal system performance, are continuously being calculated and sent through the healing interface as well.

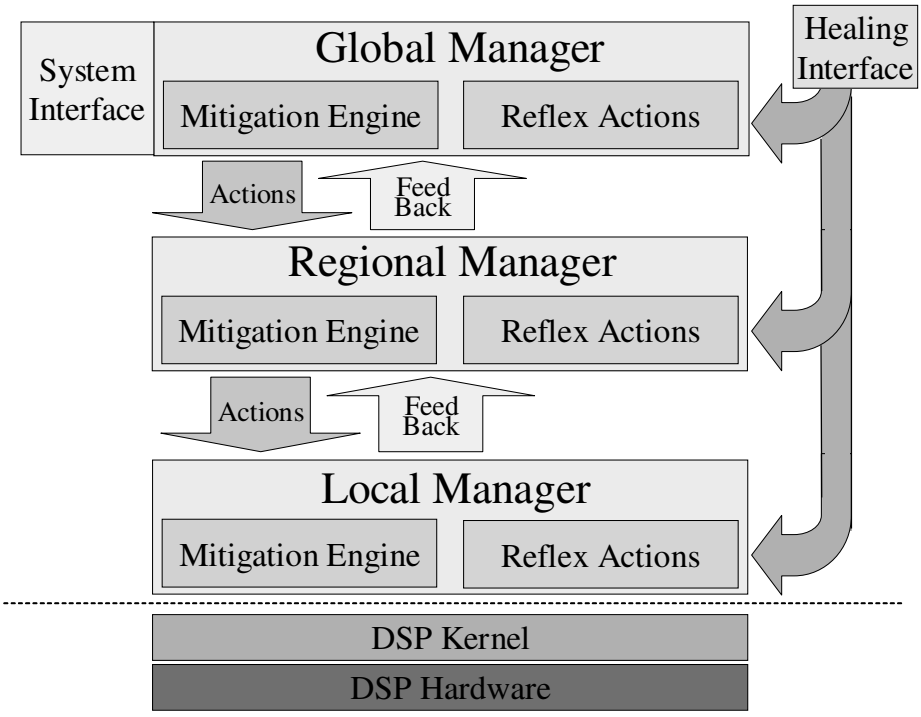

Figure 15. BTeV runtime environment

To keep data transfer rates among user applications as high as possible, no mechanisms for reliable data transfer (such as those found in TCP/IP) are used in the communication protocol. The computational overhead for communication between nodes in the system is kept to an absolute minimum through copy-less direct memory transfers provided by DMA engines in hardware. Extended monitoring of the health of such connections will be provided through direct hardware support mechanisms (watchdog timers and hardware communications monitoring).

In summary, the Fault Mitigation (FM) API for the DSP kernel provides the underlying mechanisms that support fault mitigating behaviors. The mitigation engines in both the regional and local management hierarchies rely on this tightly integrated, kernel-level support to carry out reflex and healing actions. These facilities form the tools that

the local and regional managers use to redistribute communication and computation load, thereby mitigating hardware and software failures in the network.

Runtime System Fault Mitigation Scenario

In order to explain the workings of the runtime system, we present an example scenario where the system detects an error and implements a sequence of operations to mitigate the fault.

The BTeV runtime system currently supports the simulation and recovery of a single communication error, as described earlier. Detection and recovery from this single communication link error forms the basis for additional recovery scenarios. Faults are assumed to be singular, with an additional assumption that the failure of a single component is not sufficient to render the system wholly unusable. (this implies a limited redundancy, or overcapacity of the system, but not on the order of TMR)

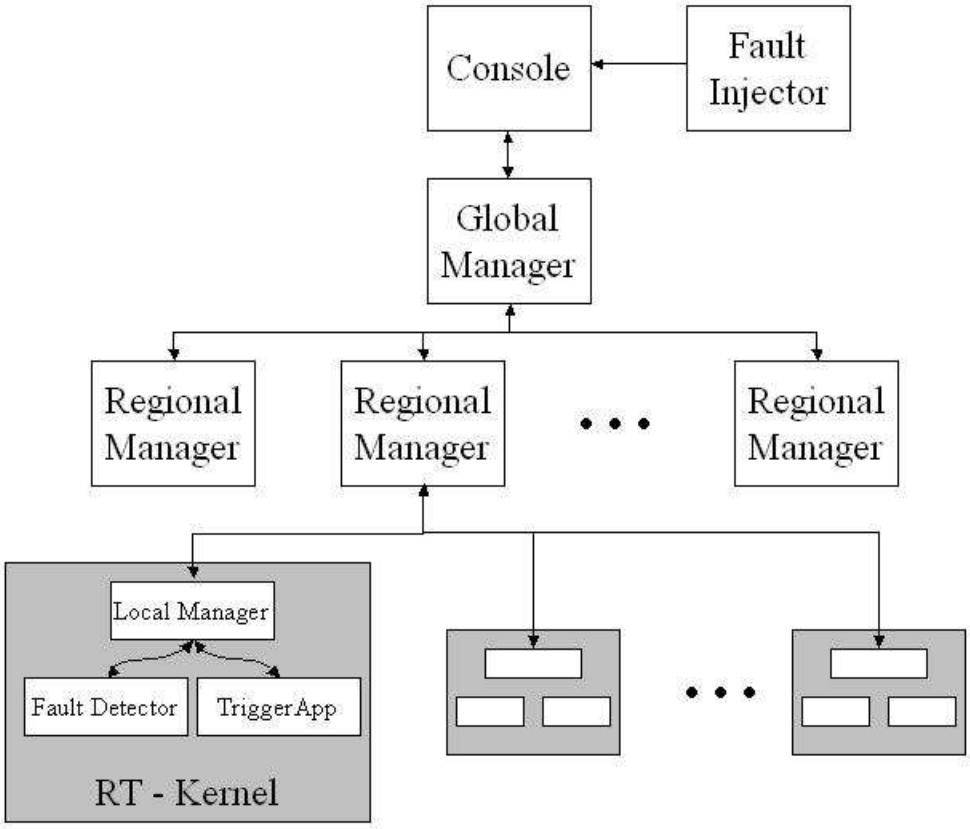

Figure 16. Runtime fault mitigation hierarchy

The following scenario details the actions of each component in the runtime system when a communication link fault occurs. The system is assumed to be running in a healthy, stable state, and that all communication links are functioning. Figure 16 shows the overall structure and hierarchy of the system. The fault injection mechanism connects to the console for simulating system faults. The console is the overall interface to the global manager, in addition to providing all the system bootstrapping, program load, reflex table loading, etc. Global, Regional, and Local Managers execute on a farm of DSPs. Local Managers execute on each of the DSP nodes, along with the Fault Detector and application code (Trigger App in the BTeV system).

The current entry point into to the system resides on a Windows-based host machine. This machine runs the user interface that allows the user to interact with the running system. Also running on the Windows-based host is the Console process, which allows components in the environment to output data to the screen. The user interface provides the option to inject a fault into the environment.

Data events are generated by the Data Generator and processed by the trigger applications on the DSP nodes, which send the results of their data calculations to the Console. They also send acknowledgement to the generator that they are fit to receive another data event. This process continues indefinitely, as is the case in a physics experiment.

To test the fault mitigation behaviors of the systems, a mechanism for fault injection is provided in the user interface running on the Windows host machine. It allows the operator to specify the type and destination of the fault. The type of fault to be injected in this case is a communication link failure, which is designated to trigger a LINK_FAILURE message. This will exercise the runtime system's ability to detect and mitigate a fault in a communication link.

The error is inserted into the environment through the Fault Injector. The fault injector creates the fault message type and populates the fields appropriately. Once created, the fault is injected into the network by sending the message to the detector residing on the selected destination node.

The Fault Detector monitors the hardware and software executing on its node for evidence of failure. Built into its fault monitoring routine is the ability to probe an incoming port for fault detection messages. When a message is detected, the Fault

Detector will accept the message, and simulate an actual fault. In the case of a LINK_FAILURE message being received, the action is simply to forward the message to the Local Fault Manager. Other types of failure messages may result in other actions (populating fields with current temperatures being recorded, stack dumps of running applications, etc.) to be taken before the message can propagate to a higher level of authority.

The LINK_FAILURE message is then received by the Local Fault Manager. Once the fault message has been received, the Local Manager will proceed through the predefined state-machine-based behavior (i.e. generated from the system models) that determines how to handle the given fault. The contents of the fault message are maintained throughout the state machine execution, as data contained in the message will be used to trigger or guard against state transitions. In this case, the LINK_FAILURE message will trigger the state machine to execute the defined behavior to handle such an event.

In this case, informing the Regional Manager of the fault is the first action in the behavior. In this step, the designer of the Local Manager's behavior has recognized that a link fault message results in coordinated actions that need to be taken on more than one node. This behavior avoids some of the synchronicity errors that would occur if local management simply executed the appropriate steps in the reflex table. If other nodes were not synchronized, they would continue to communicate with the faulty node on normal channels. Premature switching of communication channels could result in a disjoint network graph. To alleviate these conditions, the behavioral models dictate that all LINK_FAILURE messages result in immediate action to inform higher-level

management of the condition. The Regional Manager is responsible for coordinating the mitigation actions to ensure these errors do not occur.

Since the Fault Injector has prepackaged a correctly formatted LINK_FAILURE message, the Local Manager need only forward that message up to a Regional Manager. After doing so, the behavioral models dictate that the Local Manager return to a normal listening mode, in which it waits for a response from the Regional Manager. Other faults are assumed to not occur until the entire LINK_FAULRE fault is detected and fully mitigated.

While Regional Managers are similar to Local Managers, they differ in that they operate on a higher level of the hierarchy. Although in the BTeV runtime environment the Regional Manager runs on the Windows Host machine, it can also reside on the distributed DSP network.

The Regional Manager will receive notice from the LFM that a LINK_FAILURE has occurred, and will then transition through its predefined behavior to handle the fault. For mitigating communication failures, the Regional Fault Manager will send LINK_REROUTE messages to all of its local managers. This is the so-called reflex action, as it needs to happen as soon as possible to restore communications throughout the network. Also, the Regional Manager will send messages to the Console indicating that link failures are present in the network. It will then proceed to listen for other messages form the Local Managers.

When a Local Manager receives a LINK_REROUTE message, all processes on the local DSP are halted while the communication links are rerouted. A table containing reflex actions for given link failures is kept in memory. An index into the reflex table is

sent with the LINK_REROUTE message. Each node executes the actions in the table corresponding to the link that has been designated as faulty. Code executed by each DSP kernel to initiate a reflex action is given in figure 17.

```
if(l_rcin->command == REFLEX_ACTION){
  Log_printf("LocalFM: Taking Appropriate Reflex Action...\n");
  linkfailed = l_rcin->param[0];
  for (k=0; k<rc_table[linkfailed].num_instr; k++){
    if (rc_table[linkfailed].instructions != NULL){
      if(rc_table[linkfailed].instructions[k].input_not_output == 1){
        FM_swap_ports_input(\
        rc_table[linkfailed].instructions[k].process_id,
        rc_table[linkfailed].instructions[k].orig_port,
        rc_table[linkfailed].instructions[k].new_port);
      }
      else{
        FM_swap_ports_output(\
        rc_table[linkfailed].instructions[k].process_id,
        rc_table[linkfailed].instructions[k].orig_port,
        rc_table[linkfailed].instructions[k].new_port);
      }
    }
  }
}
```

Figure 17. Code listing for reflex action execution

The model interpreters have determined, at design time, the actions for rerouting streams through other hardware channels. Each process can enqueue messages to a set of streams, and data in each stream is sent to other DSP nodes via a hardware channel. Once a channel has been determined to be faulty, the data in the streams waiting to go out that channel must be copied to a new stream that is wired to a different (undamaged) channel. Also, the stream table inside the kernel must be altered to associate the original process with this new stream.

At certain points in the network it is necessary to create forwarding processes to carry data across the network. In the case of a LINK_REROUTE condition, new forwarding processes may also have to be created to support the original logical communication across a slightly modified hardware layout. Steps to perform these actions, if necessary, are also given in the reflex table.

Once the communication has been rerouted, the execution of processes continues. This scenario, as stated above, assumes that no other communication links fail during the critical execution of the detection and recovery action. Also, it is assumed that the network has not reached a minimum state, where any link failure would result in unrecoverable loss of communication in at least one point on the network.

CHAPTER V

A PROTOTYPE SYSTEM USING THE BTeV RUNTIME ENVIRONMENT

Software Model Component Description

The constructed prototype system's software model contains representations for both the physics application (PixelTrigger) as well as components for fault mitigation (Global, Regional, and Local Fault Managers). Also modeled are facilities for fault detection and injection, as well as a console component for displaying test messages to the screen during runtime.

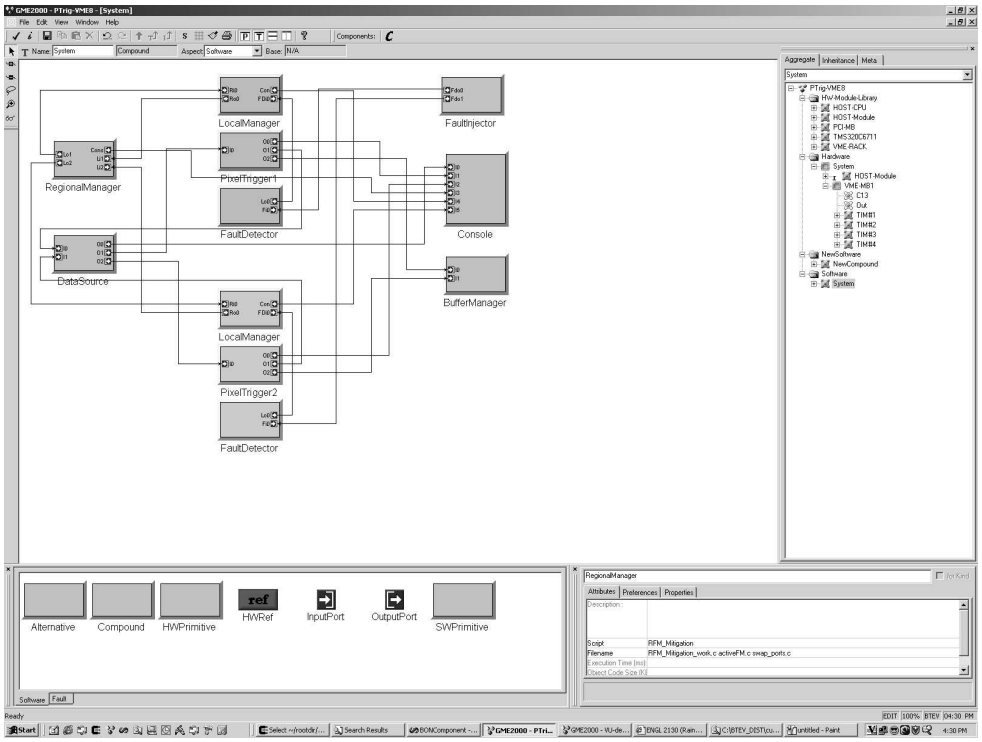

Figure 18. Prototype software component model

A hierarchy of fault mitigation agents in the BTeV experiment includes provisions for Global, Regional, and Local Fault Managers. These managers react to fault messages received by a number of sources. Global Fault management was not implemented for the prototype, as a single Regional Manager was adequate to manage the given number of nodes (eight).

Local Fault Manager

A Local Fault Manager is placed on every DSP node in the network. Figure 18 shows the tight coupling between local management and local fault detection components. Local Managers can receive Fault Mitigation Messages from a local Fault Detector, as well as other Local Managers.

Local Fault Managers are modeled as state machines in the BTeV modeling paradigm. The BTeV modeling paradigm provides facilities to model application behavior in a manner like Harel's StateCharts [25]. Interpreters transform the hierarchical state machines into a flattened form. The new models are then interpreted, resulting in the generation of standard C code to implement the state machine. These generated software components manifest themselves as components in the software models, and the Local Fault Manager is one such manifestation.

The Local Fault Manager will react to certain fault messages based on their specified state machine based behavioral models. Exact fault actions to be taken are application dependant, and may change as the needs of the system change.

Although no restrictions are in place to govern strict homogeneity of all Local Manager behavior patterns in the network, it is assumed that every instance of a Local Manager is derived from the same state machine model. In the future there may be

benefits to having alternate behaviors for Local managers based on factors such as physical proximity to a Regional Manager or specific application code being run on the node. This may also lead to very serious design space exploration problems, where the number of possible execution scenarios grows beyond reasonable control. Currently, the restriction that all Local Manager share the same behavior is for simplicity only.

Local Managers are able to receive, create and distribute Fault Mitigation Messages. Typically, Fault Mitigation Messages are received from the local Fault Detectors, and the messages propagate up through the hierarchy until the Global Manager is able to initiate a Reflex Action.

Regional Fault Manager

Regional fault management is placed more sparingly in the network, usually dependant on physical constraints of the system. For the BTeV prototype, there is a single Regional Manager. This is due mostly to the fact that a maximum of eight DSP nodes have been utilized. It is reasonable to assume that in larger systems a single Regional Manager might be present for every eight DSP nodes. Again, this is largely dependant on the hardware topology and communication paths. Using the hardware available, it may not be feasible to have more than eight DSP nodes serviced by a single Regional Manager due to the latencies incurred by communication forwarding that must be present (with current hardware available in the ISIS lab, it is difficult to implement both redundant communication links as well as high node parallelism, each node has only four available external communication links).

Regional Manager behaviors are modeled using state machines in the same manner as Local Managers. One Regional Manager is present in the system, and

therefore no assumptions are made as to the homogeneity of regional manager behavior. It can be assumed, however, that all regional Managers in the network will behave in a similar manner. The number of Regional Mangers may typically be an order of magnitude smaller than the number of physical nodes in the system (and therefore Local Mangers).

Fault Detector

Fault detection in the system is handled by the so-called Fault Detector. Each physical node in the network contains a Fault Detector. Provisions for fault detection made in the BTeV environment are mostly forward-looking in nature, as no hardware facilities to detect communication/application failure are present at this time.

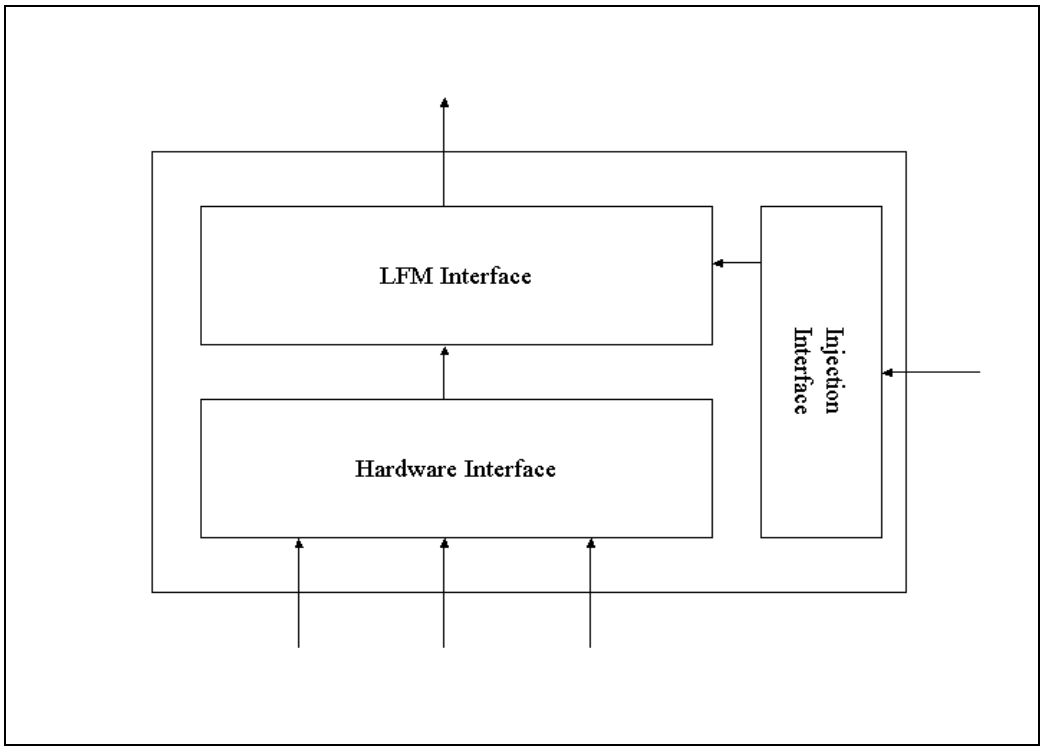

Figure 19. Fault detector interfaces

A significant feature of the Fault Detector is the fault injection interface. With the implementation of this feature an external source may force a failure to be discovered, in order that the system failure scenarios may be tested in a soft manner, without the need for actual hardware failure. Upon receipt of a message on the fault injection interface, the detector will forward that message the Local Fault Manager residing on its node.

Fault Injector

Fault Injectors allow failure scenarios to be tested in software. The Fault Injector resides on the host machine, as interactive input is needed by the user to initialize fault messages. The Fault detector creates a valid Fault Mitigation Message based on user input and sends it directly to the Fault Detector of the specified node.

A simpler approach might have been to allow faults to be sent directly to the Local Manager. A logical abstraction between fault injection and fault detection was necessary in order to reinforce the notion of transparent testing of the fault mitigation infrastructure, using facilities that closely mimic those which will be present in the final BTeV L1 trigger [24].

PixelTrigger

A typical worker node could be comprised of any software component that implements some domain-specific data processing application. In the case of the BTeV environment prototype system, a worker might implement the code for a pixel trigger, or may simply calculate π as a means for placing a computational load on the system.

For the current BTeV prototype, workers implement a preliminary revision of the actual trigger code to be used in the production system, and are referred to as

PixelTriggers. PixelTriggers receive event data from the Data Generator and process each event as if it were running live. Upon completion of processing an event, the keep/toss decision made by the PixelTrigger is sent to the console and a new set of event data is requested.

Console

The console object implements a very simple round-robin polling of its input ports, grabbing data as it arrives to display on the command window. The console resides on the host and was necessary to view status messages produced by components running on the DSPs, as no facility for displaying text is present in the DSP network.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

Conclusions

The BTeV runtime environment is able to provide a glimpse into the future for how development platforms allow users to compose and realize embedded systems. It will be used as we progress with the NSF/Fermilab system development to implement and test systems gradually increasing in size. With the addition of a Fault Mitigation API, the ACS kernel has been extended to support fault mitigation behaviors that require application-transparent stream rerouting, fault injection and primitive debugging facilities. By leveraging concepts of MIC with an execution environment that supports such fault mitigation actions, the BTeV runtime environment can allow rapid modeling, generation, execution, and evolution of fault tolerant RTE systems. This environment allows physicists and researchers of other disciples to quickly build robust and resilient embedded systems in which to run their applications without requiring expertise in the area of real-time embedded systems. With the improved runtime environment, along with capabilities for hardware, software, and behavioral modeling, the BTeV environment is able to provide a solid platform to support research surrounding fault mitigative, large-scale, real-time embedded system development.

Future Work

Several improvements are planned for the BTeV environment. The continuing progress of this project is a testament to the need for more robust tools for researching the development these types of systems.

First on the list of future improvements is the addition of live network status monitoring from within the design environment (GME 2000). GME implements a COM+ interface to allow remote querying and updating of the models. This will be done in accordance with fault detection, allowing models of the network to be updated to reflect the current health of the system at runtime.

Another planned improvement to the BTeV environment is the use of dynamic model interpretation inside the network structure. This so-called "live-model" would exist on the network and be updated and re-interpreted during runtime as network failure events occur. This would push the model interpretation process from a static a-priori event to a live runtime occurrence.

Real-time behavior is critical for these types of systems. The timing characteristics of the runtime and mitigation infrastructure should be characterized. Feedback of this information to the upper level tools will permit accurate simulation, and timing guarantees on the fault-adaptive properties of the system.

The runtime system capabilities will evolve as the tools are developed. New facilities are anticipated, along with a much closer coupling between runtime and design tools.

REFERENCES

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

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

[3] 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.

[4] Bapty T., Neema S., Nordstrom S., Shetty S., Vashishtha D., Overdorf J., Sheldon, P., "Modeling and Generation Tools for Large-Scale, Real-Time Embedded Systems", ECBS, (accepted), Huntsville, AL, April 4, 2003.

[5] Neema S., "System Level Synthesis of Adaptive Computing Systems", Ph. D. Dissertation, Vanderbilt University, Department of Electrical and Computer Engineering, May 11, 2001.

[6] Scott J., Bapty T., Neema S., Sztipanovits J., "Model-Integrated Environment for Adaptive Computing", Proceedings of the Military and Aerospace Applications of Programmable Devices and Technologies Conference, CD-ROM Reference D5, Greenbelt, MA, September, 1998.

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

[8] 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.

[9] Sprinkle J., Karsai G., Ledeczi A., Nordstrom G., "The New Metamodeling Generation", IEEE Engineering of Computer Based Systems, Proceedings p.275, Washington, D.C., USA, April, 2001.

[10] 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.

[11] Agrawal A., "Hardware Modeling and Simulation of Embedded Applications, Master's Thesis, Vanderbilt University, Electrical Engineering & Computer Science, May, 2002.

[12] Davis J., “Integrated Safety, Reliability, and Diagnostics of High Assurance, High Consequence Systems”, Ph.D. Dissertation, Vanderbilt University, Electrical Engineering, May, 2000.

[13] 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.

[14] Gajski D. D., Vahid F., Narayan S., Gong J., “Specification and Design of Embedded Systems”, *Englewood Cliffs, NJ: Prentice-Hall*, pp. 1-6.

[15] Bradley D. W., Tyrrell A. M., “A Hardware Immune System for Benchmark State Machine Error Detection”, Proceedings of the Congress on Evolutionary Computation 2002 (CEC2002), 2002.

[16] Bradley D. W., Tyrrell A. M., “Immunotronics : Hardware Fault Tolerance Inspired by the Immune System”, Proceedings of the 3rd International Conference on Evolvable Systems (ICEES2000), 2000.

[17] Ortega C., Mange D., Smith S.L., Tyrrell A.M., “Embryonics: A Bio-Inspired Cellular Architecture with Fault-Tolerant Properties”, Genetic Programming and Evolvable Machines, Volume 1-3, Pages 187-215 July 2000.

[18] 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.

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

[20] Becker T., “Application-Transparent Fault Tolerance in Distributed Systems”, Proceedings Second International Workshop on Configurable Distributed Systems, pp36-45, March 1994.

[21] Bowen N.S., Antognini J., Regan R.D., Matsakis N.C. “Availability in parallel systems: Automatic process restart”, S/390 Parallel Sysplex Cluster, IBM Systems Journal, Volume 36, Number 2, 1997.

[22] Hadjicostis C.N., “Coding Approaches to Fault Tolerance in Dynamic Systems”, BRIMS Seminar Series, Hewlett-Packard Laboratories, Bristol, UK, April 1999.

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

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

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