SPECIFICATION AND IMPLEMENTATION OF AUTONOMIC FAULT-MITIGATION BEHAVIORS FOR LARGE-SCALE REAL-TIME EMBEDDED SYSTEMS

By

Di Yao

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

August, 2005

Nashville, Tennessee

Approved: Date:

_____ _____

_____ _____

To My Parents

ACKNOWLEDGEMENTS

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

To start with I would like to thank Dr. Ted Bapty, my graduate advisor for giving me the opportunity to work on the BTeV project. His valuable ideas and guidance have helped me in developing this research and thesis. I would also like to give thanks to Dr. Sandeep Neema whose ideas opened new doorways during my research.

To all my team members and everyone actively involved in this project who gave their support and help to me when I needed it. To all the friends I made during graduate school. To Brandon Eames for allowing me to bug and pester him when he was free and when he was swamped.

Finally, I would like to give thanks to my parents for their contribution and encouragement throughout the journey. Without their support and encouragement, I would not have made it this far. I will always remember this quote that my mom has recited to me ever since I was little “Where there is will, there is a way”.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

MIC – Model-Integrated Computing.

DSML – Domain-Specific Modeling Language.

HEP – High Energy Physics Experiment.

FSM – Finite State Machine.

GCML – Graphical User Interface Configuration Modeling Language.

DTML – Data Type Modeling Language.

FMML – Fault Mitigation Modeling Language.

SIML – System Integration Modeling Language.

SIFT – Software Implemented Fault Tolerance.

GME – Generic Modeling Environment.

OCL – Object Constraint Language.

API – Application Programming Interface.

ABLE – Agent Building and Learning Environment.

IP – Internet Protocol

CHAPTER I

INTRODUCTION

The rapid advancement of computing technology results in the proliferation of the complexity and size of real-time embedded computer systems. The complexity of such systems results from a large number of components and interactions among components and with the environment. The mission-critical nature of the systems demands correct outputs at correct time with very small margin for tolerating errors.

Failures can, and do occur in these systems due to problems in hardware, software, or the environment. As consequences of failures are detrimental, faults must be detected and corrected as soon as possible without disturbing system operations. A range of fault-tolerance techniques exist, many of which are based on using redundant components [27]. Due to constraints imposed by budget, power and size, these traditional techniques may not always be feasible. Instead, these systems must be able to deal with failures while minimizing the amount of extra resources. Since there is no single 'perfect' system response to failures, the way in which any particular system handles failures is often dependent on the goals of the application and environment. These failure responses must be definable by the system designers.

The work presented in this thesis is motivated by the fault-tolerance requirements of a class of large-scale, real-time embedded systems used in high-energy physics experiments (HEP) for data acquisition and processing. These systems require thousands of processors to perform real-time computations and must be highly reliable. In order to

maintain reliability, effective fault-mitigation policies must be incorporated into the system design.

The work presented in this thesis is on the development of a tool for specifying and implementing fault-mitigation behaviors for large scale real-time embedded systems for HEP experiments using a model-based approach.

Overview of BTeV Experiment

HEP experiments use massive facilities to understand the properties and states of the basic building blocks of matter. The experiment of interest for this research is called the BTeV[1] experiment that is under development at Fermi National Accelerator Laboratory. BTeV's goal is to study charge-particle violation, mixing and rare decays of particles known as beauty and charm hadrons [2].

The experiment takes place in a particle accelerator, where enough energy is applied to protons and anti-protons moving in opposite directions to achieve relativistic speeds. The protons and anti-protons collide, breaking up into the most basic components of matter. The collisions are recorded for examination of detached secondary vertices from charm and beauty hadrons decays. The proposed BTeV detector layout is shown in Figure 1.

The experiment is designed such that particle collision occur every 132 nanoseconds with raw data rate at more than 14.8 Gbytes/sec. The collisions in the presence of a large magnetic field are recorded by the use of 30 planar pixel detectors, placed at fixed distances, providing a three dimensional data set. The results are carried to

[1] The BTeV experiment was cancelled recently. However, the work will be applied to other High Energy Physics applications.

localized processors that reconstruct the three-dimensional crossing data from the detectors to examine the trajectories for detached secondary vertices.

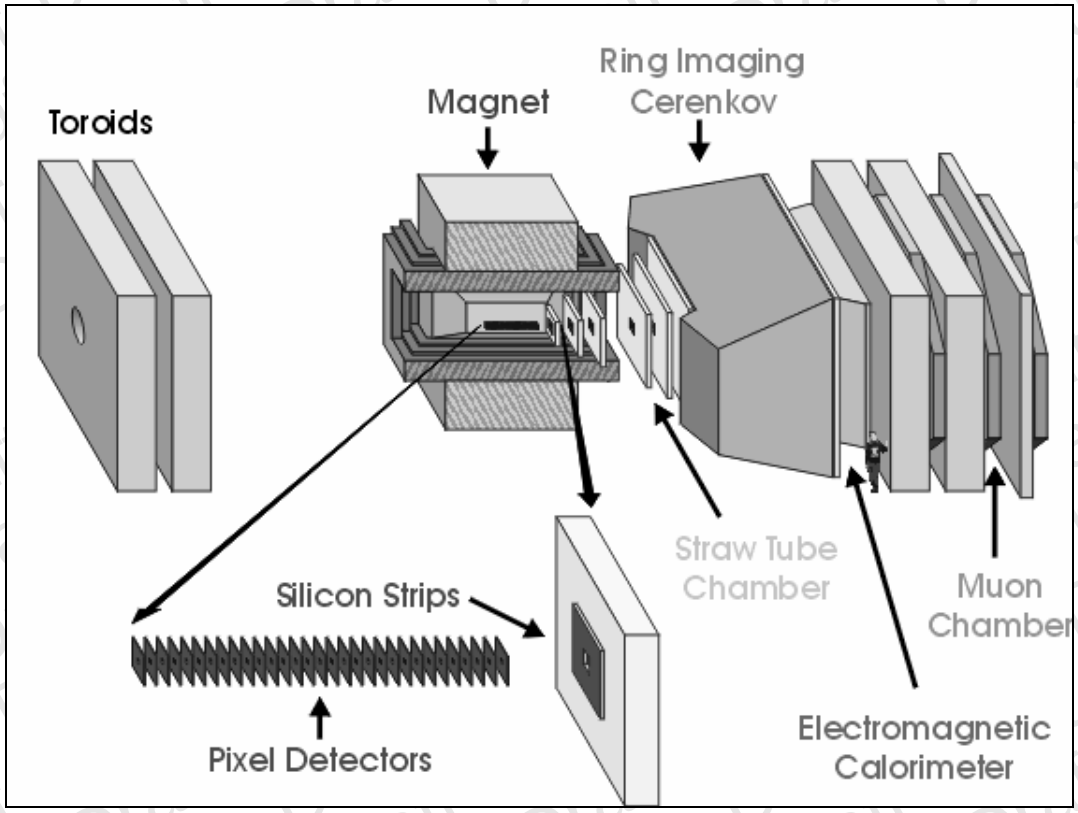

Figure 1 The BTeV Detector Setup

While the aim of the experiment is to find new phenomena occurring during these collisions, most of these collisions lead to already known collision behaviors and hence can be discarded without any loss. Furthermore, the data sizes from interaction are on the order of 2Kbytes per event with events occurring at a rate of 7.6MHz. The aggregate data rate is clearly too high to be blindly recorded. Therefore, data-dependent decision algorithms called Triggers must be executed online to dynamically compute an accept/reject decision. The trigger has three levels, all involving computations of collision data. The first level performs tracks reconstruction, primary vertex finding and

impact parameter computation [2]. All accepted event from first level are saved and passed onto the second and third level. Algorithm used in the second and third level reconstructs the event with better resolution and additional information to pick out events with potentially interesting vertex topologies [2]. These algorithms necessarily must be computed in real-time, although significant queuing is typically available.

While the actual accepted events occur infrequently, the high cost of operation of the experiment (facility, personnel, energy, etc.) and the large demand for the facility requires the system to 1) operate with excellent reliability, 2) sustain high computational performance and 3) maintain functional integrity over long periods of time [1] [3]. At the same time, the system cost must be minimized, precluding fault tolerant approaches that use redundancy such as triple-mode redundancy. Thus, the goals for fault mitigation are the following:

- Maintain the maximal application functionality for any set of component failures.
- Recover from failures as completely and rapidly as possible.
- Minimize the system cost.

The solution requires a certain degree of “self-awareness” in the system to accurately identify problems and tolerate failures. As faults will occur during operation, they must be corrected in the shortest possible time with as little human intervention as possible. The system must possess capabilities to execute automated recovery procedures, to make compensations for potentially changing resources by shifting loads or changing system thresholds, to reconfigure existing detection and recovery procedures, and to create new procedures.

These constraints and requirements are what drive the research and development of toolsets at the Institute for Software Integrated Systems, Vanderbilt University to design fault-tolerant large-scale real-time embedded systems. This thesis describes the work done to support the creation and deployment of such fault mitigating systems.

Autonomic Fault Mitigation Approach

The key features of the fault-mitigation approach in order to satisfy the requirements of the trigger and address the dependability problems associated with it is a hierarchical fault management framework that provides fault-mitigation behaviors customized for specific faults to systems. This framework uses software entities called fault managers which have mitigation knowledge from mitigation strategies embedded within them [4].

This fault-mitigation approach distinguishes two aspects of fault-mitigation: Structural and Behavioral.

- Structural refers to the location and relationship between fault management entities.
- Behavioral refers to the internal operation of the fault manager entities.

In large-scale systems, it is impractical for a single fault manager to manage all mitigation activities. A single manager will result in unpredictably large reaction times. The reaction time to any specific fault will depend on pending faults from the entire system, an indeterminate number. With an unpredictably large reaction time, a fault has the chance to propagate to other components, potentially inducing a cascade of failures. A single fault manager is also susceptible to single-point of failure where failure of this single fault manager disables the entire system's fault handling capabilities.

Due to these concerns, distributed fault managers, with distinctive fault-mitigation knowledge, should be employed. Specialized fault managers enable division of fault handling tasks so that mitigation actions could occur rapidly and concurrently, resulting in short reaction time. One or more fault managers are deployed to reside on each node in the system to provide protection to the running applications using their fault-mitigation knowledge.

Moreover, fault managers are composed in a hierarchical fashion in the system to form a management network that has a tree structure. Each manager has a specific control zone over which it has authority to take mitigation actions. Faults are handled by managers at the lowest level of occurrence in order to minimize propagation time. If a fault can not be resolved at a level, a request is propagated up to the immediate superior manager until it can be resolved. System-wide mitigation commands are typically broadcast down the hierarchy by the manager at the highest level (i.e. the root node in the tree structure).

For the HEP application space, we have defined three logical levels of hierarchy: Local, Regional, and Global levels.

- Local Managers perform basic fault actions that occur on local nodes in the system.
- Regional Managers deal with successively larger groups of nodes. Note that there can be multiple levels of regional managers.
- Global Managers have the authority to order system wide mitigation actions and responding to mitigation requests from Local and Regional

Managers. Note that there can be multiple global managers, serving as backups.

The fault managers operate concurrently along with the rest of the system. Responding to faults as they occur asynchronously, automatically making fault-mitigation decisions for faults in their zones only so they minimally perturb the system. The managers on each level only communicate with its subordinate or superior managers at a higher or lower level to coordinate fault-mitigation decisions.

Reaction time improves since mitigation decisions can be made closer to the fault source. Scalability also improves because new fault managers can be added easily at the appropriate hierarchy level without causing disruption to the existing fault-mitigation network as system size increases [20]. For example, if several new nodes are introduced in the system, they can either be added as individual nodes to the Local management level or they can be added as an entire region to the Regional management level.

Implementing fault managers with custom fault-mitigation knowledge and deploying them across the system require extensive knowledge of the runtime environment. While we can expect a computer engineer/scientist to work within the details of implementation, the domain knowledge to define system recovery actions is more with the application designer and users, in this case, the physicists. In addition, as mitigation behaviors progressively become more complex, it will become harder and harder to manage and evolve them. Therefore, a high-level tool that abstracts the runtime environment and facilitates the creation and management of the framework is needed.

Problem Statement

The goal of the research described in this thesis is to develop model-based tools for achieving autonomic fault-mitigation in large-scale real-time distributed systems and to demonstrate the applicability of the tools on examples from the BTeV application.

Developing such a tool involves the following:

1. Developing a domain-specific modeling environment for specifying autonomic fault-mitigation behaviors in the system. The modeling environment uses a domain-specific modeling language called Fault Mitigation Modeling Language (FMML) to model mitigation behaviors for fault managing entities and interactions with other components in the system.
2. Developing a translator to map high-level specifications to lower-level artifacts that is used by an execution platform to instantiate a hierarchical fault management framework that runs concurrently along with the system to mitigation failures.

This thesis describes the tool architecture, specification, and implementation. The tool is evaluated via a case study implementing a HEP application using this tool. The organization of the thesis document is as follows: Chapter II presents a survey of some of the existing tools in designing autonomic systems and fault-mitigation execution platform. Chapter III describes various other modeling tools developed for the BTeV system. Chapter IV and Chapter V give details about FMML and the translator. Chapter VI presents a case study demonstrating tool applicability using BTeV application as an example. Conclusions are drawn in Chapter VII.

CHAPTER II

BACKGROUND AND LITERATURE SURVEY

Background

Model-Integrated Computing

Model Integrated Computing (MIC) is a design methodology used for building embedded software systems [5]. Its key objective is to provide a way to design an embedded system by capturing its requirements and manage its evolution. Domain specific model is the key element in this approach. Using MIC technology one can capture the requirements, actual architecture, and the environment of a system in the form of high-level models. These models act as a repository of information that is needed for analyzing and generating the system.

The MultiGraph Architecture (MGA) provides a unified software architecture and framework for designing and building systems using the MIC approach. The MGA design process is comprised of three levels as shown in Figure 2. Synthesized, adaptable software applications are transformed from system models built in the Model-Integrated Programming Synthesis (MIPS) using model translators. The MIPS provides a Domain Specific Modeling Language (DSML) that governs how a system can be modeled. The formal semantics, syntax and visualization rules of a DSML are specified through a metaprogramming interface at the Meta-Level in the form of meta-models.

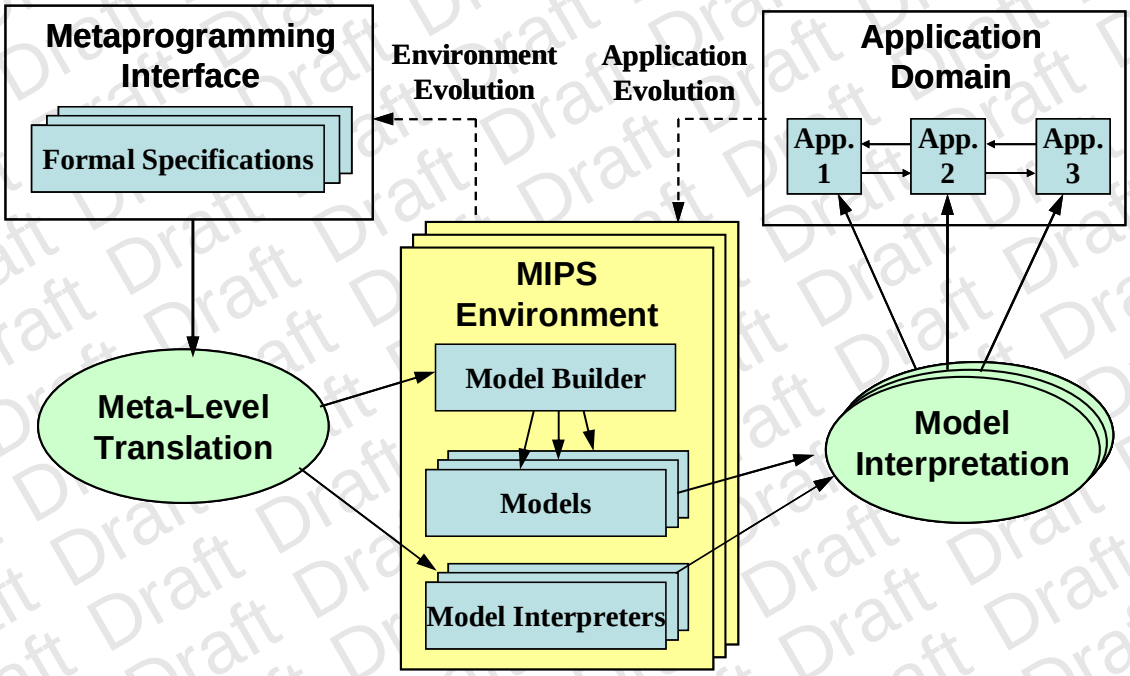

Figure 2 Design Process in Multi-Graph Architecture (MGA)

A toolkit called Generic Modeling Environment (GME) provides an environment for creating DSML and model translators. GME metamodeling environment provides a graphical interface similar to UML class diagrams [8]. In the metamodel, the user specifies the set of entities, their associations, groups and ordering that can be created in the domain models. The semantics of domain models are enforced in two ways. First, constraints specified by Object Constraint Language (OCL) are applied to models to enforce static semantics [9]. Second, dynamic semantics are enforced through model-interpreter that parses the models to generate source code, configuration or analysis files.

Software Implemented Fault Tolerance

Software Implemented Fault Tolerance is a dedicated software infrastructure capable of providing fault tolerance to user applications in a distributed environment.

Chameleon is a SIFT infrastructure developed at University of Illinois at Urbana-Champaign [10] that explicitly provides fault-tolerant services through a range of error detection and recovery mechanisms for applications. This section will give details about Chameleon since it is used as the execution platform for the fault management framework.

Chameleon provides fault-tolerant services through Adaptive Reconfigurable Mobile Objects of Reliability (ARMOR) [11]. ARMORs are processes that can be installed on multiple nodes to provide application specific fault-tolerance services. The Chameleon infrastructure is essentially a network of ARMOR entities residing on a single or multiple nodes linked together through a specialized messaging system.

Since fault-tolerant requirements may change from application to application, the ARMOR architecture has been structured such that it can be reconfigured to accommodate changing requirements. The ARMOR architecture may be reconfigured from a structural and a behavior level. The structural level refers to the number and type of ARMORs that run on nodes across a system. The behavior level refers to the functionality of individual ARMORs. These two levels of reconfigurability are further explored in the following paragraphs.

Three types of ARMORs have been developed: Manager, Daemon, and Common ARMORs. A detailed description of each type is given below [11]:

1. Manager ARMOR is the most authoritative object in the ARMOR hierarchy. They execute fault-tolerance strategies which allow the failure detection and recovery of subordinate ARMORs.

2. Daemon ARMORs are installed on every node participating in the Chameleon environment and provide a communication gateway for local ARMORs to ARMORs on remote nodes. They execute fault-tolerant strategies that allow error detection and recovery of other local ARMORs.
3. Specialized Common ARMORs execute different fault-tolerant strategies to provide for application dependability.

The functionality of an ARMOR is determined by a composition of building blocks called Elements. A set of core infrastructure Elements have already been defined and are stored in a behavior library that comes with the installation of Chameleon. ARMORs from the three types defined previously have a predefined set of core functionalities. The Elements of an ARMOR are executed by a microkernel inside the ARMOR.

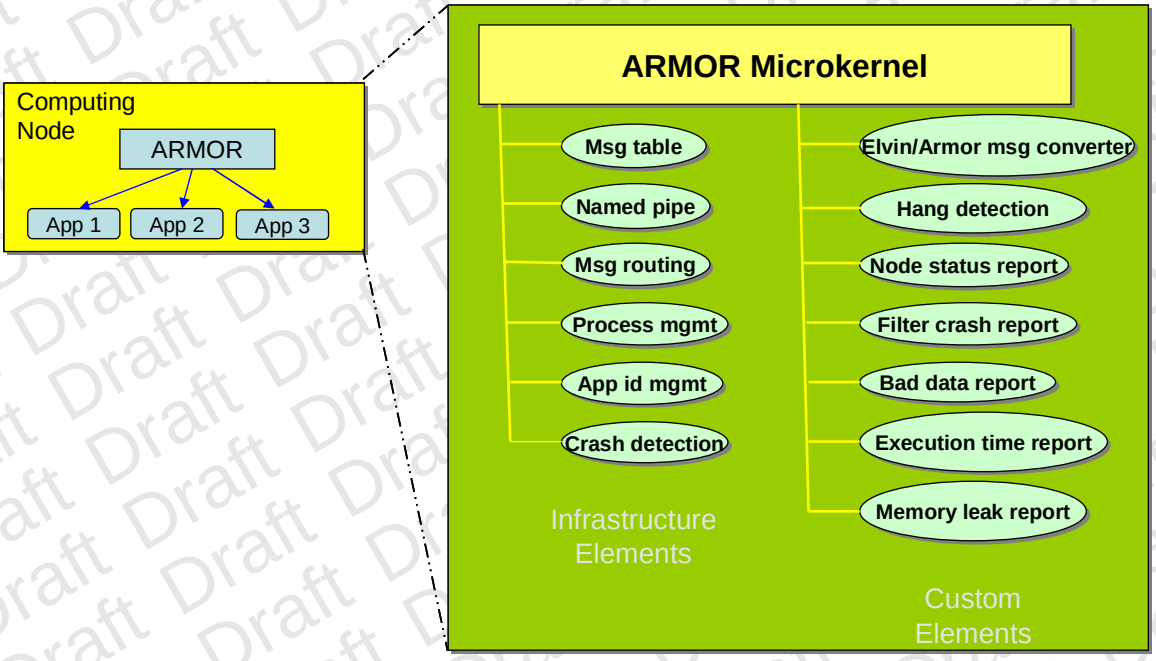

Figure 3 Internal Structure of an ARMOR

Due to the composable nature of ARMORs, behavioral level reconfiguration can be achieved by modifying the Element composition of ARMORs as well as implementing custom Elements. Modification to the Element composition can be achieved by adding or removing Elements from an ARMOR through a Element composition file. Custom Elements have specialized behaviors and can be added to an ARMOR through the composition file. However, they must conform to the common publish/subscribe messaging interface available in all Elements to be able to use ARMOR's internal messaging scheme. Figure 3 illustrates the makeup of an ARMOR.

Autonomic Computing

An immediate consequence of the increase in complexity and size of computing systems in the recent years is the rise in cost and difficulty in managing and maintaining such systems. Autonomic computing technologies and tools that enable automatic management of operations with minimal human intervention are being developed as a solution to this problem. By applying these technologies and tools, the resulting system will be able to regulate its functions in response to the environment in the same way that the human nervous system regulates body functions without conscious input [13].

An autonomic system must satisfy the following four key characteristics:

1. Self-Configuring – adapting to conditions in its environment by configuring system parameters
2. Self-Healing – discovering and diagnosing problems and using alternate ways to function without disruption
3. Self-Optimizing – using available resources and adjusting workloads to yield the maximum performance

4. Self-Protecting – anticipating, detecting, identifying, and protecting itself from possible threats

An autonomic system will be able to function well under unexpected conditions. Fault-tolerance based upon an autonomic approach can be cheaper, eliminating expensive redundancies characteristic of traditional fault-tolerance techniques [14]. This section will review some of the available technologies and tools for designing autonomic systems and applications and examine their capabilities in providing fault-tolerance.

Agent Building and Learning Environment

The Agent Building and Learning Environment (ABLE) is a Java-based toolkit for developing and deploying intelligent agent applications developed by IBM [15]. The framework provides a lightweight Java agent framework, a library of intelligent software components, a set of development and test tools, and an agent platform.

The ABLE agent framework is a software architecture that is built on the standard JavaBeans model. It allows algorithms to be packaged into JavaBeans called AbleBeans with a common interface. These AbleBeans can be connected to one another to form AbleAgents who can obtain information and perform actions with respect to applications they are responsible for. AbleAgents can be formed from other AbleAgents, in addition to AbleBeans.

ABLE provides a library of core AbleBeans for data access, machine learning algorithms, machine reasoning and interface engines. In addition, library provides a common data model composed of a set of data type classes for Boolean, Categorical, Discrete, Numeric and String literals, variables and fields. In addition to these core beans, users can wrap new or existing algorithms to create customized beans. Any new beans

must be wrapped by an AbleBean instance, associated with a BeanInfo file that specifies an members to be externalized, and a GUI Customizer class to allow users to set any algorithm attributes.

ABLE offers an interactive development and test environment. AbleAgents can be graphically constructed using the core AbleBeans library and other AbleAgents. AbleAgents can also be hand-coded and tested in this environment in the form of JAR files.

ABLE agent platform provides a set of services including agent life-cycle transitions and directory facilitators and agent communication functions that allow AbleAgents to form multiagent systems. The platform can support agents on multiple physical systems using Java Remote Method Invocation for communication.

ABLE toolkit can be used to add autonomic properties to applications by allowing users to create specialized agents who can monitor the external environment, plan reflex actions, learn behaviors, and take actions when problems occur. ABLE provides an architecture for construction of intelligent software components and agents using core beans that provide core functionality while allowing custom beans to be created for more specialized behaviors.

AutoMate

Grid computing attempts to use resources of many separated computers across a network to solve large-scale computation problems. AutoMate is a framework for enabling development of autonomic Grid applications that are capable of self-configuring, self-composing, self-optimizing and self-adapting. The idea is to construct autonomic applications as dynamic composition of autonomic components [16].

The framework consists of two key components: an autonomic component framework called Accord and an agent-based middleware infrastructure called Rudder. The Accord framework consists of four concepts [17].

1. Application Context – Defines a common semantic basis for components and applications. The semantic basis for describing application namespaces, component interfaces, sensors and actuators. Accord uses an XML-based language for describing functional and non-functional aspects of components.
2. Autonomic Component – Defines autonomic components as basic building blocks for autonomic applications. An autonomic component is a self-contained software module with specified interfaces and context dependent. A component has rules, constraints, and mechanisms for self-management and interacts with other components and systems via three ports to export information about their behavior, resource requirements, performance, interactivity and adaptability. An embedded rule agent monitors component state, controls execution of rules and cooperate with other agents to fulfill overall application objectives.
3. Rule Definition – Management and dynamic composition of autonomic components are guided by rules. Rules are If-Then expressions where the conditional part is a logical combination of component/environment sensors and events. The action part of a rule is a sequence of component/environment sensor/actuator invocations. There are rules for defining runtime functional behaviors of autonomic components and

interactions between components, their environment, and within an autonomic application.

4. Rule Enforcement – Rules are injected into components at run time to enable self-managing behaviors in applications. Rules for runtime functional behaviors are executed by an embedded rule agent. Rules for component interactions are dynamically injected into the interacting components.

Rudder is an agent-based middleware infrastructure to provide core capabilities for supporting autonomic composition, adaptations, and optimizations [16]. Self configuration is enabled through dynamic discovery and composition of new components and reconfiguration at run time. Self optimization enabled through dynamic switching of workflows and components. Self healing is achieved by restarting or replacing failed components. Self protection is enabled through reactive behaviors. The Rudder has two parts:

1. The agent framework of Rudder provides three types of peer agents: Component Agent (CA), System Agent (SA), and Composition Agent (CSA). Component Agents define interaction rules for component interaction/communication behaviors and mechanisms. System Agents monitor, schedule, and adaptively optimize physical resource utilization. CA and SA exist as system services. CSA are transient, generated to satisfy application requirements. Application and system dynamics an uncertainties are addressed by rules that enable applications to dynamically change flows, components, and component interactions [16].

2. Reactive tuple space – Tuple space provides coordination service for distributed agents and mechanisms for rule definition, deployment and enforcement. It executes runtime adaptive policies to allow coordinated application execution and optimized computational resource allocation and utilization. Programmable reactive behaviors defined using a tuple consisting of a Condition, Guard, and Reaction are supported dynamically. Condition associated triggering events with reactions. Guard defines execution semantics of the behavior. Reaction specifies the computation for the behavior.

vGrid

vGrid is a middleware architecture that sits on top of the existing Grid middleware, intelligently managing and executing autonomic applications with huge computational requirements over limited Grid resource [18]. It is an extension of AutoMate. vGrid is made up of three layers.

The first layer is called Autonomic Problem Solving Environment (Autonomic PSE). It provides application developers with a software development environment to design and construct scientific or engineering applications. The idea is to develop an autonomic application as a dynamic and opportunistic composition of autonomic components [18]. Individual modules of code are encapsulated into Fine Computation Unit (FCU) with information on data, operational rules, knowledge about its neighbors along with input and output ports for communication.

A group FCUs are managed in a collection called Virtual Computation Unit (VCU) with common properties. A VCU is given to a single Grid resource. A set of VCU

makes up the complete parallel application. The autonomic properties reside at the VCU level. A vGrid manager (VGM) uses information and policies about a VCU's behavior, resource requirements, performance, and interactivity and adaptively to autonomously change the VCU's configuration when needed.

Layer 2 is the vGrid Infrastructure Services enhances existing Grid middleware and runtime services to support autonomic Grid Applications [18]. The main components of the vGrid architecture include vGrid Manager (VGM), vGrid Resources, Open Grid Services, and Distributed Communication Service. The VGM itself is composed of Monitoring Engine (ME), Analysis Engine (AE), Planning Engine (PE), Knowledge Engine (KE) and Execution Engine (EE) for setting up and configuring application execution environment manages and controls all autonomic requirements.

KE stores all high level policies as implementation rules regarding different scenarios. Example rules are estimated/projected execution times, effect on overall performance, load on resource unit and etc. The PE uses these rules to plan appropriate strategy for particular scenario. Local AE computes actual value of an application characteristic (i.e. execution time) with the estimated value. If actual value exceeds accepted value, then Local PE is notified and plans adjustments. The adjustments are made by the corresponding Local EE. If adjustments can not be fulfilled on a local domain, then requests are propagated up to the global ME, AE and PE for determining a new domain for execution. The local engines do not have authority in making decisions outside their own Grid domain.

A distributed communication infrastructure with white board for coordinating all communications in the infrastructure among the engines and components. Layer 3 is

called the Autonomic Grid Application Execution Environment for monitoring and controlling actual execution of an application. It exists in each Grid resource domain. Its main components consist of Local VGM, local KE, ME, AE, PE, and EE. It sends VGM information about the VCUs being executed using the whiteboard.

Summary

The tools listed above have been applied to Grid applications or applications related to server diagnostics, administration and e-commerce whose requirements are different from HEP experiments. Significant amount of time and effort is required to customize and apply these tools to the Trigger system. Furthermore, if a manual development process is used, implementing the software for the tools can incur significant additional development cost as thorough knowledge of the tool and the specific implementation language is required.

Moreover, research in the field describes characteristics of autonomic systems and specific examples or middleware, but there is not a standardized mechanism for constructing these systems. The challenges include defining autonomic fault-mitigation behaviors, implementing these behaviors in software, integrating these with the application, and providing the middleware and underlying operating system. Moreover, coordinating the responses across a distributed system further complicates the design.

Therefore, automated tools are needed to help manage the complexity associated with designing and implementing autonomic response systems. The work described in this thesis is one such tools for designing fault mitigation behavior that aims to meet the requirements and challenges of HEP experiments using a model-based approach.

CHAPTER III

MODELING LANGUAGES FOR THE BTEV ENVIRONMENT

The Trigger system consists of several different aspects, ranging from hardware topology, communication architecture, software component configuration, fault-tolerance policy specification, data and message-types, message passing interfaces, run control, logging, online diagnosis, and deployment. These aspects of the system interact in varying degrees and are evolving on different timelines. In addition, there is a need to version-control the evolution of designs and design artifacts.

A set of narrowly-focused Domain Specific Modeling Languages (DSML), integrated through a high-level language, has been developed to address these concerns [19]. The topic of this thesis, the Fault-Mitigation Modeling Language, is one of the narrowly focused DSMLs for specifying fault-mitigation policies in the system. Details of this modeling language can be found in Chapter 4. This chapter gives an overview of the other modeling languages for the Trigger system in the BTeV environment to provide the background to show how the DSMLs integrate together.

Data Type Modeling Language

The systems running High-energy physics experiments are composed of thousands of distributed processors. Due to the size of the system and nature of the experiment, large amounts of data and message transfer takes place, both locally as well as between processors across the network. A publish-subscribe mechanism called Elvin [28] is used in the system for this purpose. Like other publish-subscribe message systems,

messages are routed to clients across the network based on subscriptions of message contents. Clients can both publish and subscribe to messages by invoking specific APIs provided by Elvin.

To reduce the complexity of the Elvin communication mechanism and match the messaging APIs to the tool requirements, an abstraction layer is developed over the Elvin APIs. This communication layer provides a standard way to marshal and de-marshal messages using the Elvin protocol while hiding the implementation details of the Elvin APIs. To further reduce burden on the user, the marshalling and de-marshalling code for messages are automatically from Data Type Modeling Language (DTML) models [19].

Users model the structure of messages used in the system using the Data Type Modeling Language. A message can contain message fields who represent simple data of type floats and integers to specify representation size (i.e. number of bits used) as well as composite data that can contain other composite data and simple data. Additional information such as array size or signed/unsigned representation can be specified through attributes of the message fields.

Figure 4 shows an example message model expressed in DTML. This particular message contains a composite data and two simple data. The composite data represents the header of a message which contains three simple data (node_id, region_id, and message_category). As mentioned in the previously, the marshalling and de-marshalling code for this message is automatically generated for the user based on the message structure specified in the DTML model.

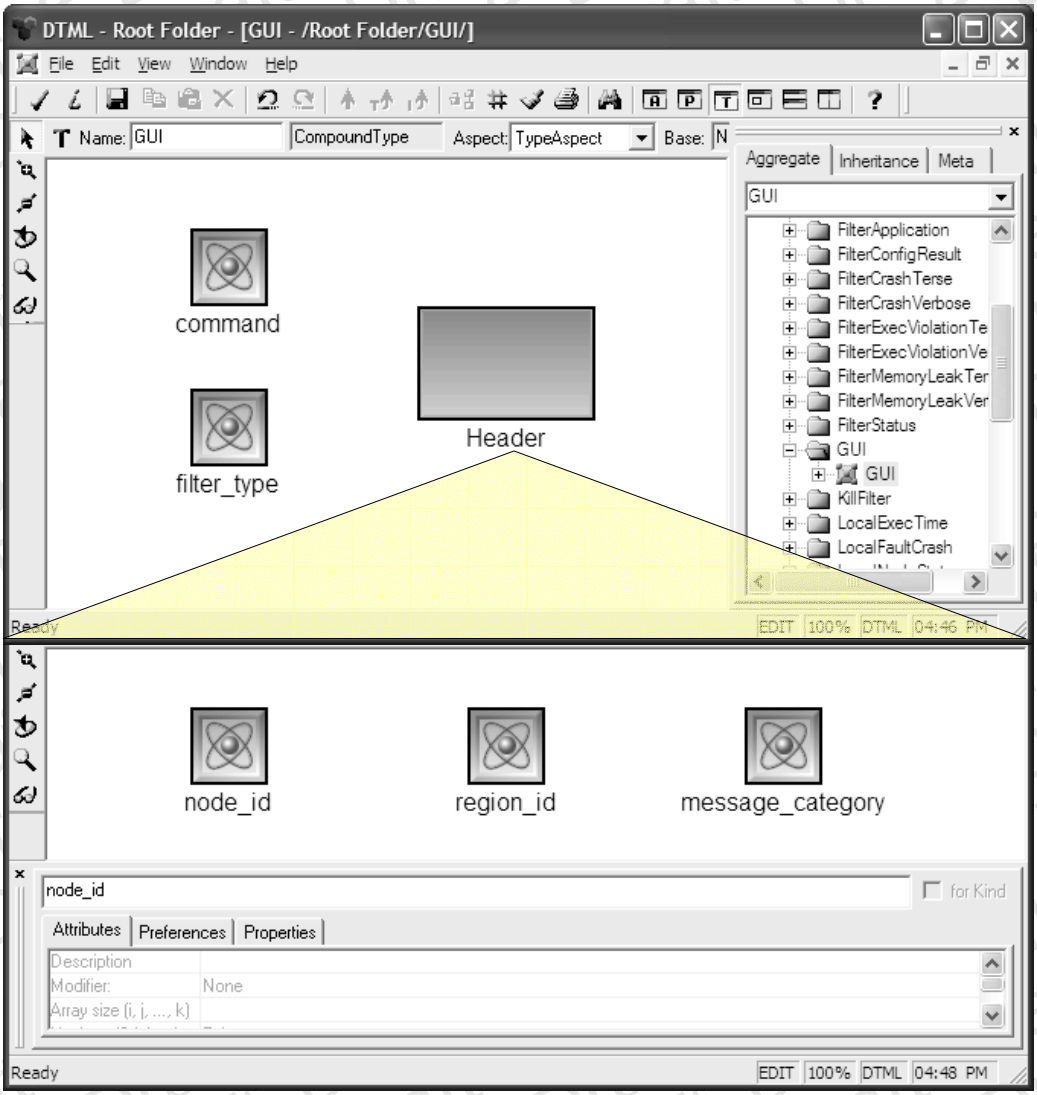

Figure 4 Example of a DTML Message Model

Graphical User Interface Configuration Modeling Language

Being able to monitor a system is essential to ensure that a system is functioning correctly. As a system evolves, its user interface needs to be reconfigured to reflect the changes in system requirements. Furthermore, configurable user interfaces would enable the physicists to dynamically view data and error conditions in ways that aid in system analysis. They would also allow users to dynamically configure and control the state of the system. The Graphical User Interface Configuration Modeling Language (GCML) is

a modeling language that provides an intuitive environment for users to configure user interfaces [21].

Graphical User Interface Configuration Modeling Language allows users to create multiple user interface panels as well as specify the displays and controls that are a part of these panels. The types of displays include Cartesian plot, histogram plot, checkerboards, and text box. The types of controls include pushbuttons, slider bars and editable text box. There are two aspects for the display and control objects, the Display aspect and the Dataflow aspect. The Display aspect is used to specify structural properties of display and control objects such as their physical location inside a user interface panel. The Dataflow aspect displays the data from the system that is to be displayed or controlled on a panel [21]. Figure 4 shows an example user interface with these two aspects. An additional component is provided in the Dataflow aspect called Computation Blocks that connect to incoming data to perform computation on the data before displaying. In Figure 5, data from a SystemStatus message is connected to a Computation Block for processing before they are displayed on a Checkerboard display in the MainControl panel.

Once the user has modeled the system user interface using the language provided, the tool is capable of generating the artifacts necessary to create the user interface. The artifacts include structural files for specifying the location of display and control objects in a panel and dataflow code for receiving monitoring messages and sending control messages. Currently, the tool generates software that is executed by Matlab to create the user interfaces.

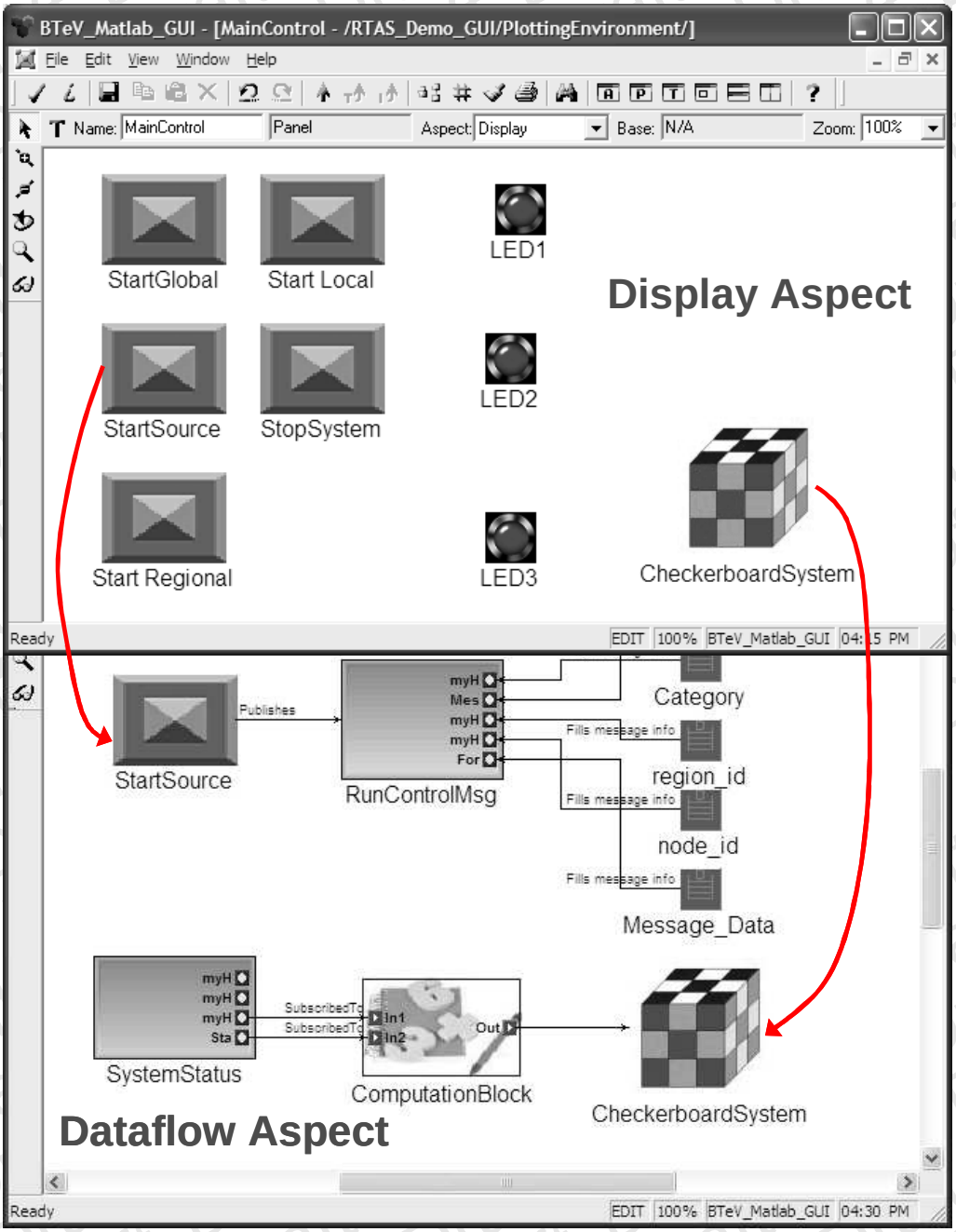

Figure 5 User Interface Model in GCML

System Integration Modeling Language

The System Integration Modeling Language (SIML) is a system level modeling language used for specifying general dataflow within the system, as well as the structural

and communication layout of the system. Typical objects represented in SIML are regions or partitions in the system, hardware components, software components, dataflow connections, node identification, and message routers. The information captured on system components, component hierarchy and interactions within the system in a SIML model are relevant for constructing system configuration files used in deploying the system.

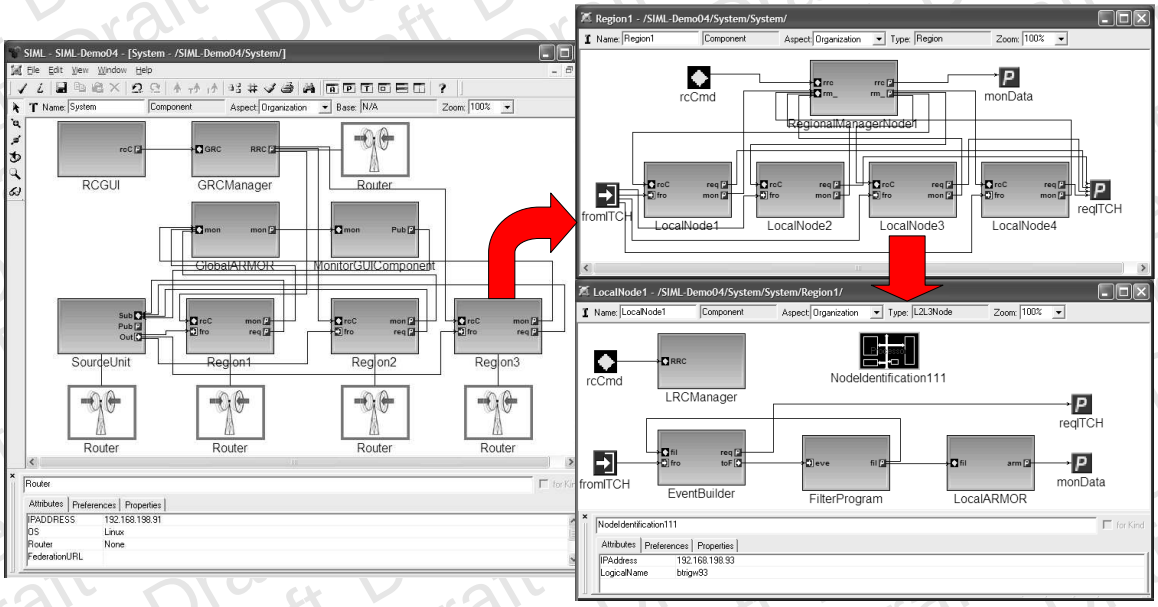

Figure 6 System Model Expressed in SIML

Figure 6 shows a system configuration model expressed in the System Integration Modeling Language for a prototype HEP experiment from a global view. This model defines a GlobalManager component (GRCManager), several Region components (Region1, 2 and 3), and several LocalNodes. Each Region component represents a logical partition of the system and contains a RegionalManager and several LocalNodes. LocalNode components can be further decomposed to show a LocalManager component

and two HEP application components. Processors are identified by their IP address using the NodeIdentification object shown in Figure 6. Data paths between the components at global level are captured by solid black lines to represent point-to-point communication, and Routers represent access points to publish-subscribe communication services.

Integration of Modeling Languages

A set of Domain Specific Modeling Languages has been developed for the Trigger system, each specifying a relevant aspect of the system. They are briefly summarized below:

1. The Data Type Modeling Language describes the data and message types used in the Trigger system.
2. The Graphical User Interface Configuration Modeling Language specifies the logging and online diagnosis interface to the system.
3. The System Integration Modeling Language determines the hardware topology, communication architecture, and software component configuration of the system.
4. The modeling language used for specifying custom fault-mitigation behaviors of the system, Fault Mitigation Modeling Language (FMML), is described in detail in Chapter 4.

These different languages are integrated together to form a complete description of the system. The System Integration Modeling Language serves as the high level language through which models of other DSMLs can be accessed, allowing integration of the modeling languages. This integration of the modeling languages is achieved using the concept of a Link type. A Link can be seen as a bridge between two graphical modeling

languages. A Link has attributes to identify the modeling language of the linked object, and file path of the linked object (relative to a CVS working-directory), and provides a concise interaction between different modeling languages [21].

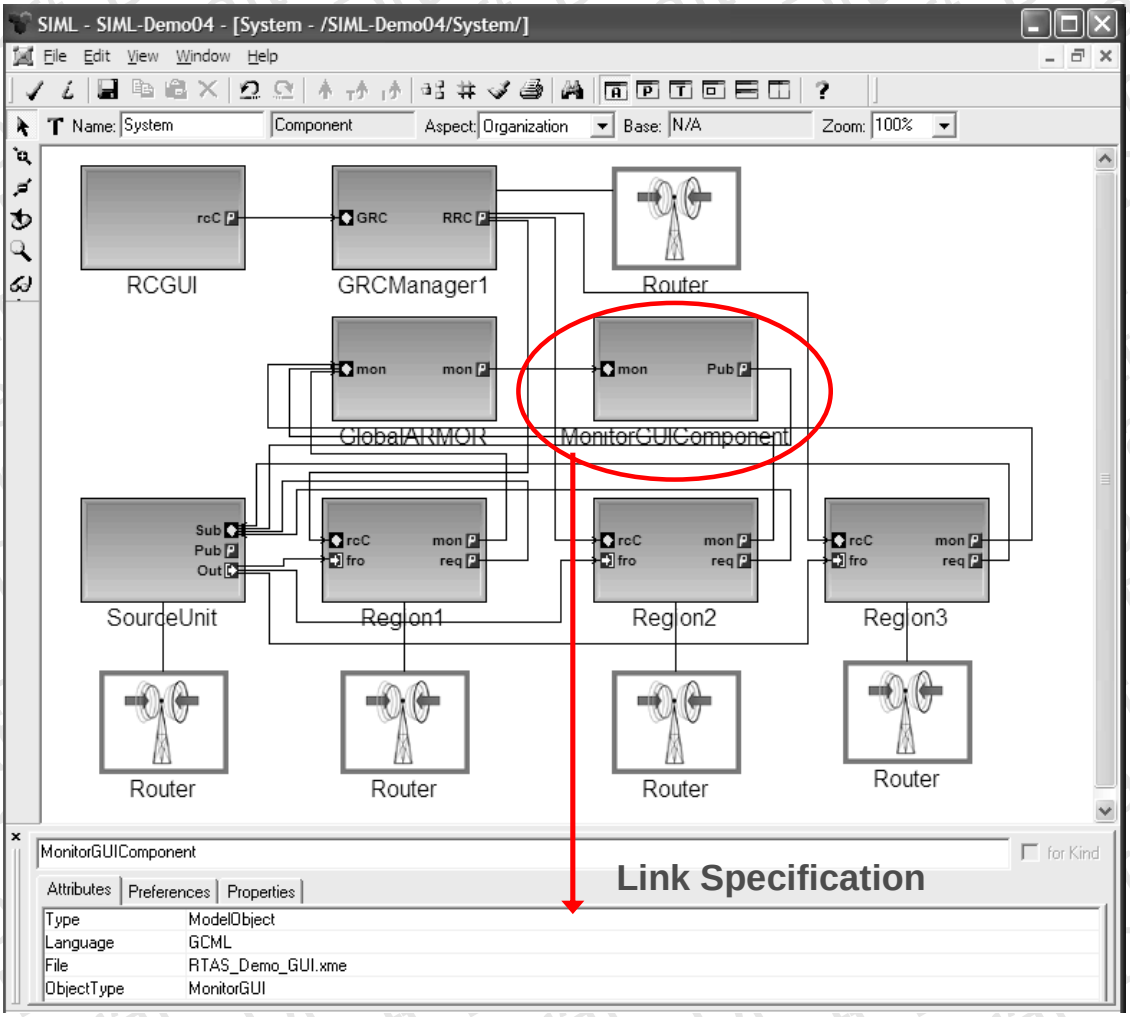

Figure 7 Specification of Link in SIML

A Link has specialized tools to facilitate the process of Link creation and Link navigation, as the creation of a Link is non-trivial and may require an intricate mapping between concepts in different modeling languages [19]. These tools relieve the system designers from the onus of Link creation and reduce the possibility of error occurring

during Link creation and navigation processes. As a result of this integration, SIML component can be decomposed to a model expressed in one of the many other languages in the tool suite through a Link type.

The MonitorGUIComponent in Figure 7 is one component that can be decomposed to a user interface model created using GCML. The Link attributes associated with this component are link type, name of the modeling language, the filename of the user interface model, and object type. Other components in the SIML that are associated with models of other languages via a Link are Global, Regional, and Local fault managers.

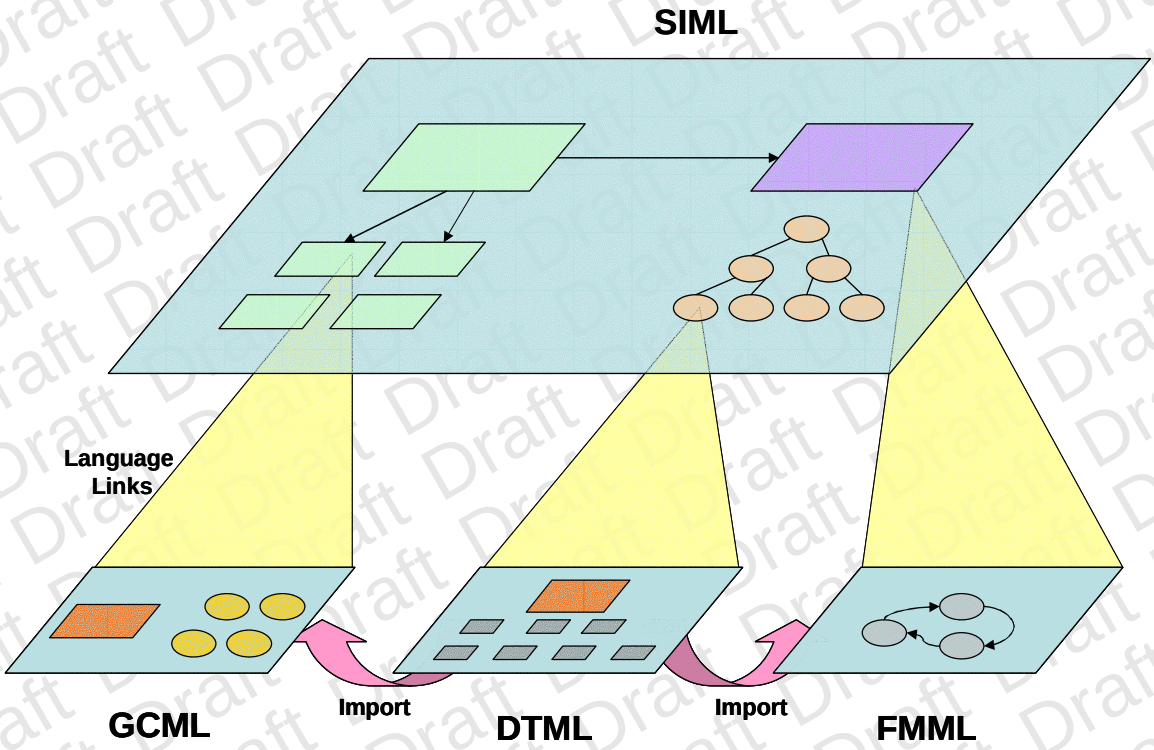

Figure 8 Integration of Modeling Languages

Data types and messages from Data Type Modeling Language models can be imported into the Graphical User Interface Modeling Language models and Fault Mitigation Modeling Language models. During the import process, data types and messages from DTML models are copied into the GCML and FMML models that use these to specify the information to be extracted or filled during system execution. Figure 8 Illustrates how the modeling languages integrate with each other [21].

CHAPTER IV

FAULT MITIGATION MODELING LANGUAGE

Chapter 1 presented a fault mitigation approach using a hierarchical fault management framework where fault managers on multiple hierarchy levels can individually or cooperatively make mitigation decisions. Recall that this mitigation approach distinguishes two aspects: Structural and Behavioral. Structural refers to the placement of specialized fault managers within a system. Behavioral refers to the collective set of fault mitigation behaviors for all the fault managers in the framework. Together, they form the foundation for system-wide fault mitigation.

A high-level tool called the Fault Mitigation Modeling Tool has been developed for defining and instantiating a fault management framework with customized fault detection, mitigation and recovery capabilities. The Domain Specific Modeling Language of the tool, Fault Mitigation Modeling Language (FMML), provides an environment for creating fault mitigation behaviors and recovery policies for the system under fault conditions. FMML also allows capturing information pertaining to the structural makeup of the fault mitigation framework. This chapter presents the concepts behind the FMML.

Domain-Specific Modeling Language

A Domain Specific Modeling Language allows a system designer to describe a system in terms of a domain. It enables system designers who have domain knowledge but not necessarily the implementation knowledge to design a system through use of

models with a strict set of concepts from a domain. The models can be analyzed to verify required properties of the design and used to generate the final system.

DSMLs are declarative, have precise semantics and use domain-specific symbols [7]. A DSML is specified by the five-tuple of concrete syntax (C), abstract syntax (A), semantic domain (S), and semantic and syntactic mapping (M_S and M_C)

$$L=<C, A, S, M_S, M_C> \quad (1)$$

The purpose of an abstract syntax, A, is to define the data structures that can represent models via concepts, relationships, and integrity constraints using Unified Modeling Language (UML) class diagrams and Object Constraint Language as metalanguage. The concrete syntax, C, can be seen as a mapping of abstract syntax onto a specific domain of rendering [7]. Modelers can directly interact with the concrete syntax by manipulation of graphical objects that are renderings of the underlying objects of abstract syntax. The purpose of semantics is to define the meaning of models that we create using the DSML. Semantics can be broken down into two parts: semantic domain and semantic mapping. The semantic domain, S, is defined by mathematical formalisms. Syntax mapping, M_C, is mapping of A→C, whereby syntactic constructs are assigned to elements of the abstract syntax. Semantic mapping, M_S, is a mapping of A→S where syntactic concepts are related to concepts in the semantic domain.

The syntax and static semantics of a DSML can be specified using a technique called metamodeling. The Generic Modeling Environment (GME) provides an environment for creating metamodels to describe the entities, their attributes, and their relationships that are available in the target DSML using UML class diagrams and OCL syntax. Once a metamodel has been created, the DSML can be generated using a meta-

translator. Fault Mitigation Modeling Language is defined by a metamodeling language using the Generic Modeling Environment tool. The metalanguage defines the structural and behavioral aspects of a hierarchical fault management framework.

Structural Concepts

Fault managers reside on all nodes within a system, each providing some specialized fault-mitigation services. There are four general types of fault managers that can be instantiated for a fault-mitigation framework. The metamodel in Figure 9 shows the four types of fault managers available in the modeling language using a language similar to UML class diagrams. Note that fault managers are called ARMORs in the Chameleon execution platform. Therefore, the four types of fault managers are called ManagerArmor, LibraryArmor, DaemonArmor and CustomArmor in the metamodel. A brief description of each type of fault managers is as follows:

1. ManagerArmor – Monitors other fault managers in the systems. It keeps location and type information of all the existing fault managers in the system. (A)
2. LibraryArmor – Provides specific predefined behaviors to applications. (B)
3. DaemonArmor – Exchanges heartbeat and registration information with ManagerArmor and other fault managers. (C)
4. CustomArmor – It has behavior of one of the other three types of fault manager, specified via a reference (D), with additional custom mitigation behaviors.

The first three types are composed of predefined behaviors available from a behavior library. The fourth type is intended for customization by the system designer. Custom fault managers can contain any number of arbitrary fault-mitigation behaviors in addition to behaviors from the behavior library.

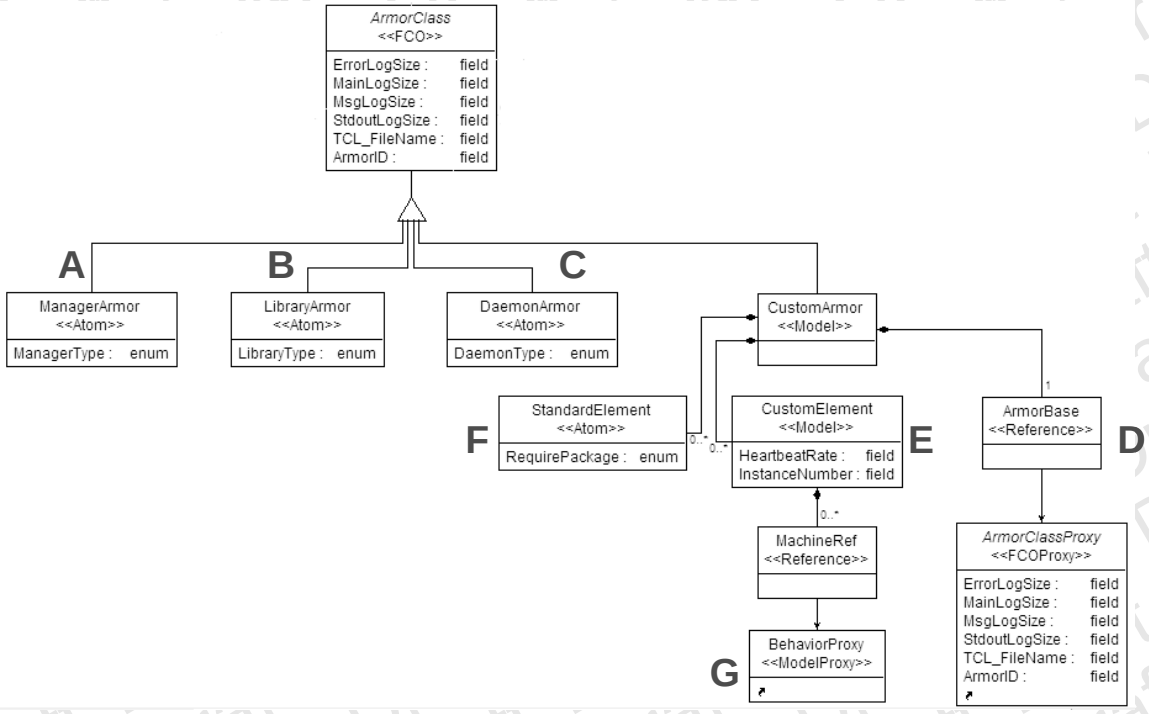

Figure 9 Metamodel of Structural Concepts

Behaviors can be seen as basic elements that can be plugged into a fault manager. Two types of behaviors are available for plugging into a CustomArmor fault manager: custom behaviors that are called CustomElements (E), and predefined behaviors from the behavior library that are called StandardElement (F). The relationship between the behaviors and custom fault managers are constructed using containment relationship. A CustomElement object contains a behavior reference to a custom mitigation behavior model (G) that opens the behavior model it refers to when double clicked.

The global view of fault managers' placements within a system is captured using the System Integration Modeling Language where fault managers reside on LocalNode, RegionNode, and GlobalNode components in the system. Fault managers' behavior models are integrated into a SIML model using Links. Figure 10 shows that a LocalNode contains a LocalARMOR component that has a Link to a FMML model. The FMML model specifies a Custom type fault manager and a Daemon type fault manager.

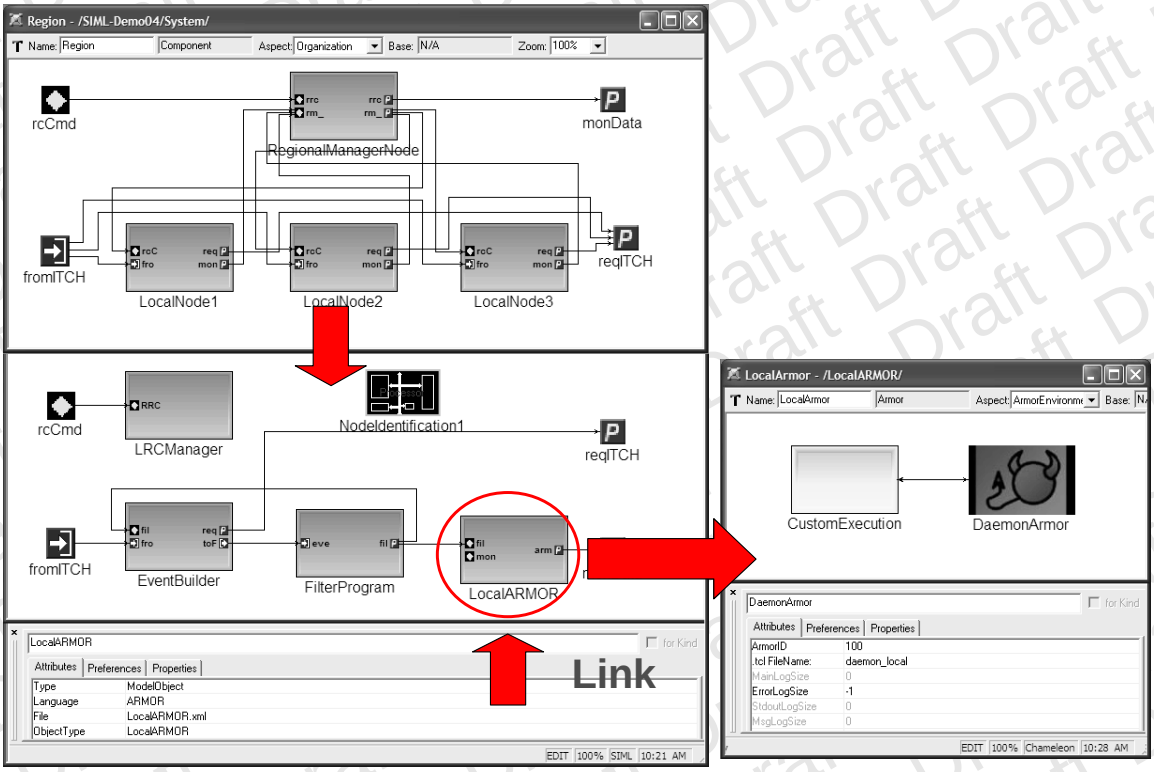

Figure 10 Location Specification of Fault Managers on a Local Node

Behavioral Concepts

The State machine-based formal specification methodologies have long been used to design and specify reactive systems. The FMML uses Statechart [23] notation for describing the fault-mitigation behavior of fault managers. Using FMML, failure states

and progression of behavior under failure can be defined as well as interactions between fault managers and with other components of the system.

The features of the behavioral specification are as follows:

- State defines the status of a computing node at a particular instant in time.
- Transitions implement the change of states. These occur with a particular event (application failure, status or recovery reports, etc).
- Transition objects are annotated with Trigger, Guard, and Action. The Guard is a condition that must be true for a transition to occur. The Action is a specific action that is executed when a transition is taken (e.g. reset event variables, data variables, decrementing event counts, etc).
- Transition objects contain Messages objects that act as triggering events to activate transitions.
- Lines connect States and Transition objects represent transitions between States, capturing the progression of States.

Each mitigation behavior is specified using a hierarchical state machine. Figure 11 shows the metamodel that define the syntax and semantics of the behavior state machines. The *Machine* acts as a container that encapsulates mitigation behaviors, one container per behavior described using hierarchical state machine concepts. States of the system are defined by a set of *State* objects. Each *State* object can be further decomposed to two or more concurrent substates or mutually exclusive disjoint substates like Statechart. *Initial* denotes the default state a system is in through an *InitialConn* type connection to the default state. Each behavior state machine contains a set of *Data* for use in keeping track of current and next states, system information, thresholds, and error

counts. *Transition* denotes progression of states due to a triggering event. *Trigger, Guard, and Action* are attributes of a *Transition* object to specify the triggering event, guard and action to be performed as a result of transitioning to the next state. Note that unless the *Guard* attribute is fulfilled, a state transition will not take place and actions specified in the *Action* attribute will not take place. User-defined actions could be computing parameters for reconfiguration or statistics for tracking trends in the system. The user can also create and send new events as messages to other components or fault managers in the system.

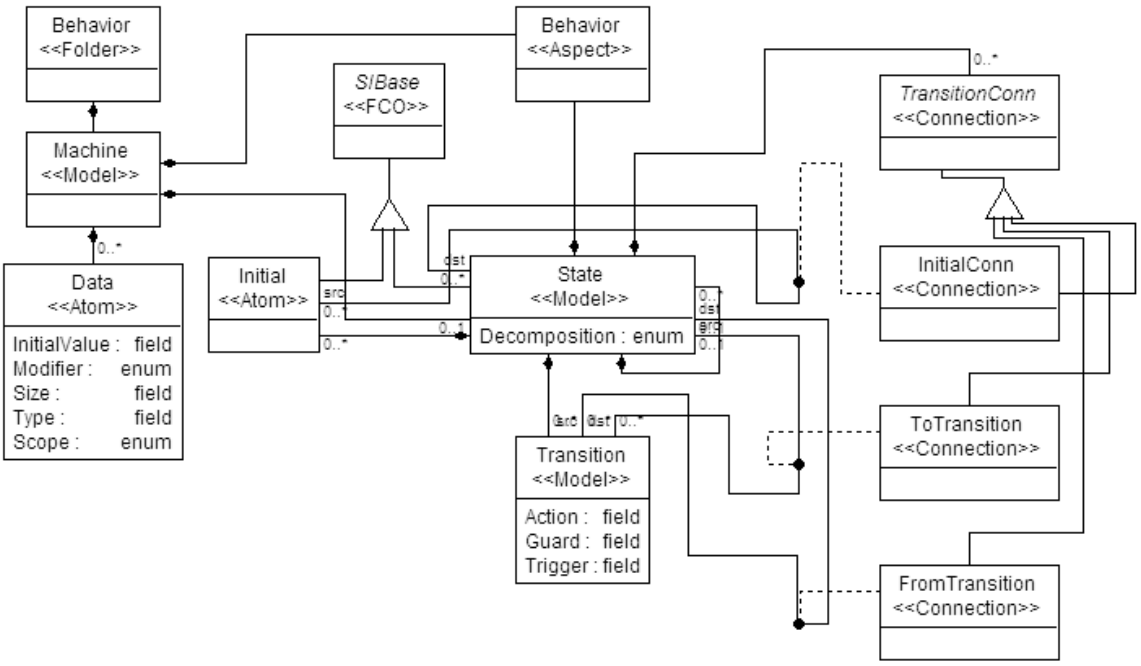

Figure 11 Metamodel of Mitigation Behavior Specification

As stated previously, events in the form of messages act as inputs and outputs events to the behavior state machines. The input events are used to trigger a state transition while the output events are sent due to a state transition. Events can carry

system information from another component or it can be a command message from a superior fault manager or user interface. Events between fault managers and components in the system are exchanged via the Elvin publish/subscribe mechanism.

The metamodel in Figure 12 shows the specification of input and output messages. A Transition object can contain *Message* objects through UML containment relationship (A). Message objects can contain message fields that are of type *Int8, Int16, Int32, Int64, or Real64* (B). These types are the exact same types used by the DTML. This is because the messages specified in DTML models are copied into the *MessageImport* folder (C) of FMML models using a helper tool called a plug-in. By copying the message structures from a central message model, we can be sure that the messages used by all modeling languages have the same structures.

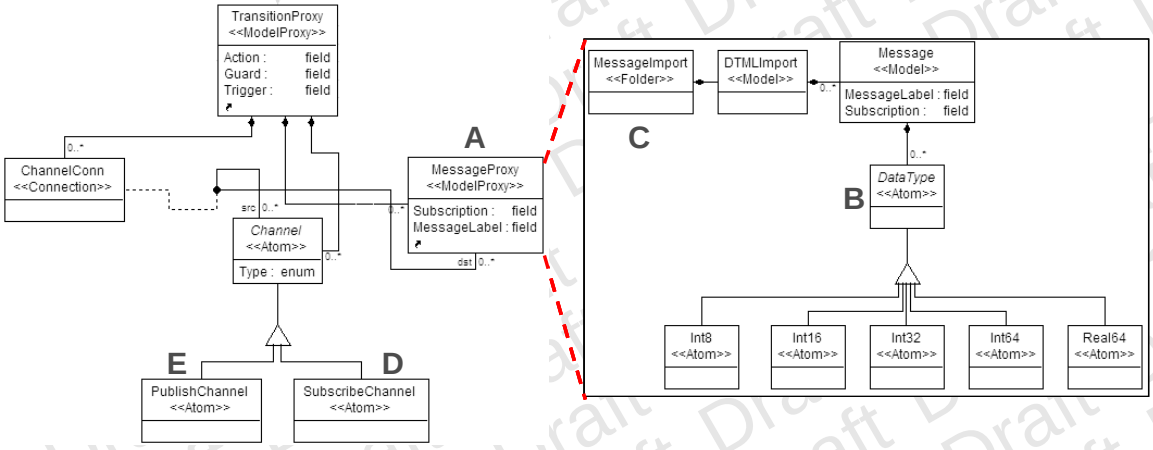

Figure 12 Specification of Input and Output Events in Fault Managers

Sources of messages can be divided into two groups, either an internal or external channel. Internal channel messages are from the same fault manager. External channel message are from a different fault manager or other system components. Likewise,

messages are sent via an internal or an external channel. Figure 12 shows these key concepts using a metamodel. Input events are denoted by connecting a *SubscribeChannel* object (D) to a *Message* via a *ChannelConn* type connection. Output events are denoted by connecting *PublishChannel* object (E) to a *Message* object. The *Type* attribute indicates the type of source or destination a message is from or intended for; whether message is for another mitigation behavior of the same fault manager or another system component.

Figure 13 shows a behavior model expressed in FMML [20]. In this model, there are two possible system states, NOMINAL and FAULT, with NOMINAL being the default state. Arrival of an AsteroidData message triggers Transition1 to be evaluated. The guard condition is checked first. In this model, the guard condition is verifying whether a state variable called “parameter” is below a threshold value. The value of the parameter variable comes from the AsteroidData message. If the guard condition evaluates to true, a RequestData message is sent to another component that has knowledge of how to handle such message and state moves from NOMINAL to FAULT. If the guard condition evaluates to false, no state transition takes place. Likewise, another AsteroidData message triggers evaluation of Transition2 and transition executes depending upon evaluation of the guard condition.

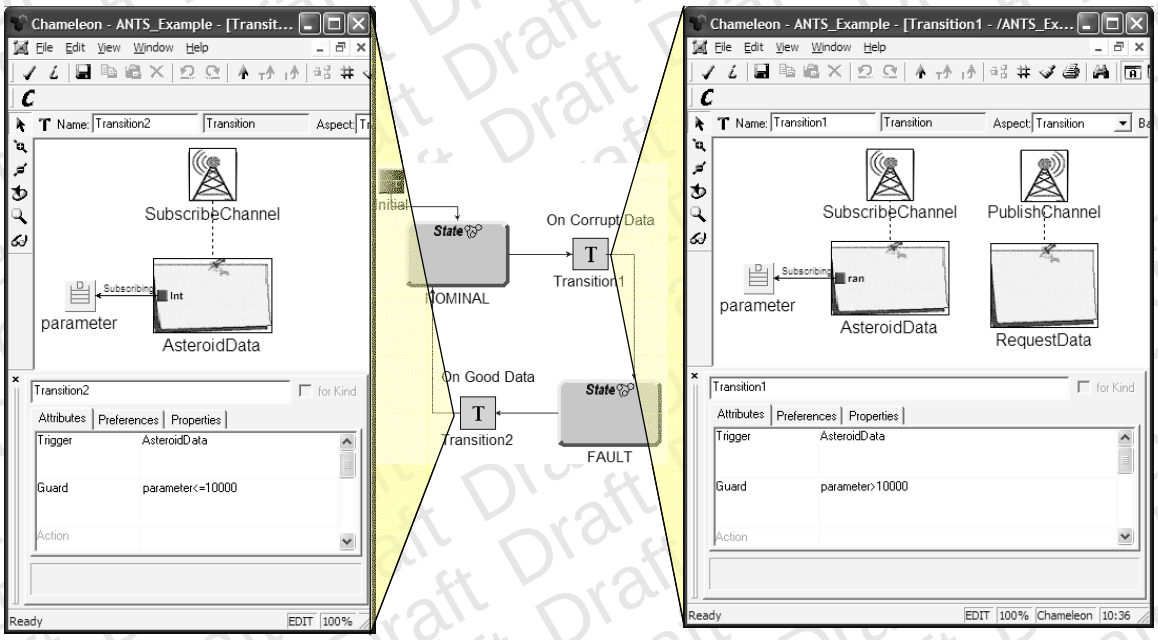

Figure 13 Mitigation Behavior Model in FMML

Extensions to Level 1 Fault Mitigation Research

The current fault-mitigation language focuses on the requirements of Level 2/3 data filtering of the BTeV Trigger system. It is built upon prior work done for the Level 1 data filtering of the Trigger system. Details of the Level 1 fault-mitigation language can be found in [24][25]. The current language uses Statechart notations to specify mitigation behaviors like the Level 1 tool but with some extensions.

The current language allows temporal behavior by using temporal events to trigger transitions and temporal guard conditions. A behavior can be activated periodically by a message that is sent at a user-define rate. This feature can be useful for behaviors that are required to update the user interface at a constant rate or monitoring of important tasks that require frequent verification. Determination of whether to remain or switch to a different state can be dependent upon how long it has been since entrance into the state by checking the elapsed time in the guard condition. This can be used to

determine if an error is intermittent or persistent. An example scenario would be to transition into a probation state when an error has occurred. If the same error repeatedly occurs within a certain time period then that could be an indication of a serious error and calls for thorough examination of the error. Then the system administrator would be notified. There are many other variations of this scheme.

The current language is also integrated with another modeling language, the Data Type Modeling Language. Due to the number of nodes in the system and the large amount of messages exchanged among these nodes, it is important to ensure the consistency in the messages used by all the components. This feature is achieved by using a utility program that imports messages from DTML into the FMML models. The message fields of the imported messages are represented as ports allowing direct copy and paste of messages and assigning/retrieving data to/from the messages.

CHAPTER V

MAPPING DOMAIN MODELS TO IMPLEMENTATION

With the Fault Mitigation Modeling Language defined, users can create models of fault managers along with their mitigation behaviors. Through the use of a model translator, these models can be mapped to artifacts used by an underlying execution platform to instantiate a fault-mitigation framework, thereby, bridging the gap between high level mitigation behavior designs to low level source code [7].

Execution Platform Artifacts

Currently, Chameleon is the execution platform for the BTeV prototype system. A detailed description of it can be found in Chapter 2. Chameleon possesses several key capabilities for executing a hierarchical fault management framework. These capabilities are listed below:

1. Fault Manager Specification and Instantiation - In Chameleon, fault managers are called Adaptive Reconfigurable Mobile Objects of Reliability (ARMOR).
2. Mitigation Behavior Instantiation - Fault-mitigation behaviors can be added and removed from fault managers. ARMORs in Chameleon support multiple mitigation behaviors in the form of pluggable building blocks called Elements, each implementing a custom behavior.
3. Communication - It provides mechanism for reliable communication between fault managers and within the fault managers.

Artifacts for creating and instantiating fault managers, specifying custom mitigation behavior, and communicating with other system components and fault managers are generated from Fault Mitigation Modeling Language models. Figure 14 shows the translation process from FMML models to artifacts.

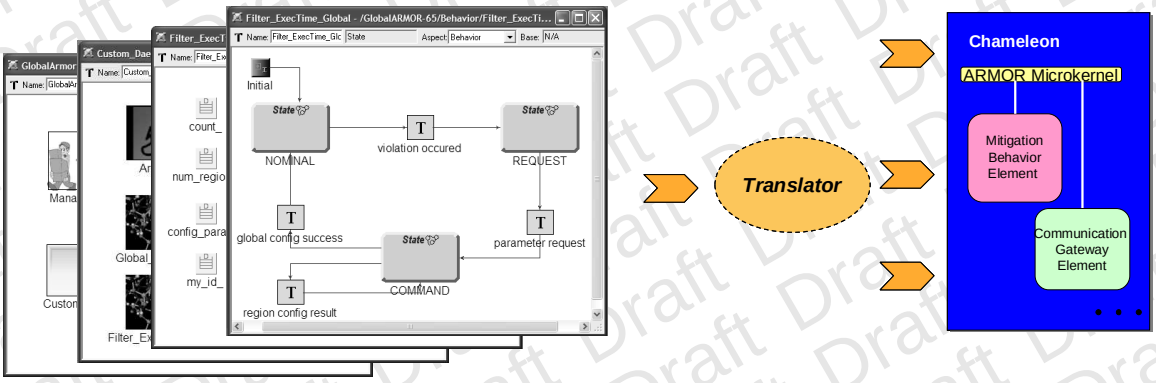

Figure 14 Mapping Process and Resulting Artifacts

Fault Manager Specification and Instantiation Script Generation

A microkernel in the Chameleon environment is responsible for instantiating fault managers. The instantiation process is invoked through a shell script that supplies the microkernel with a manager id and name of a Tcl script file. Every manager in the system must have a unique id for identification. The id and Tcl file name come from attributes associated with the fault manager object in the FMML model.

The Tcl script file contains various structural information about a fault manager, ranging from id of the fault manager to the name of the mitigation behaviors this manager possesses. The names of the behaviors come from name of StandardElement and CustomElement objects that a fault manager contains.

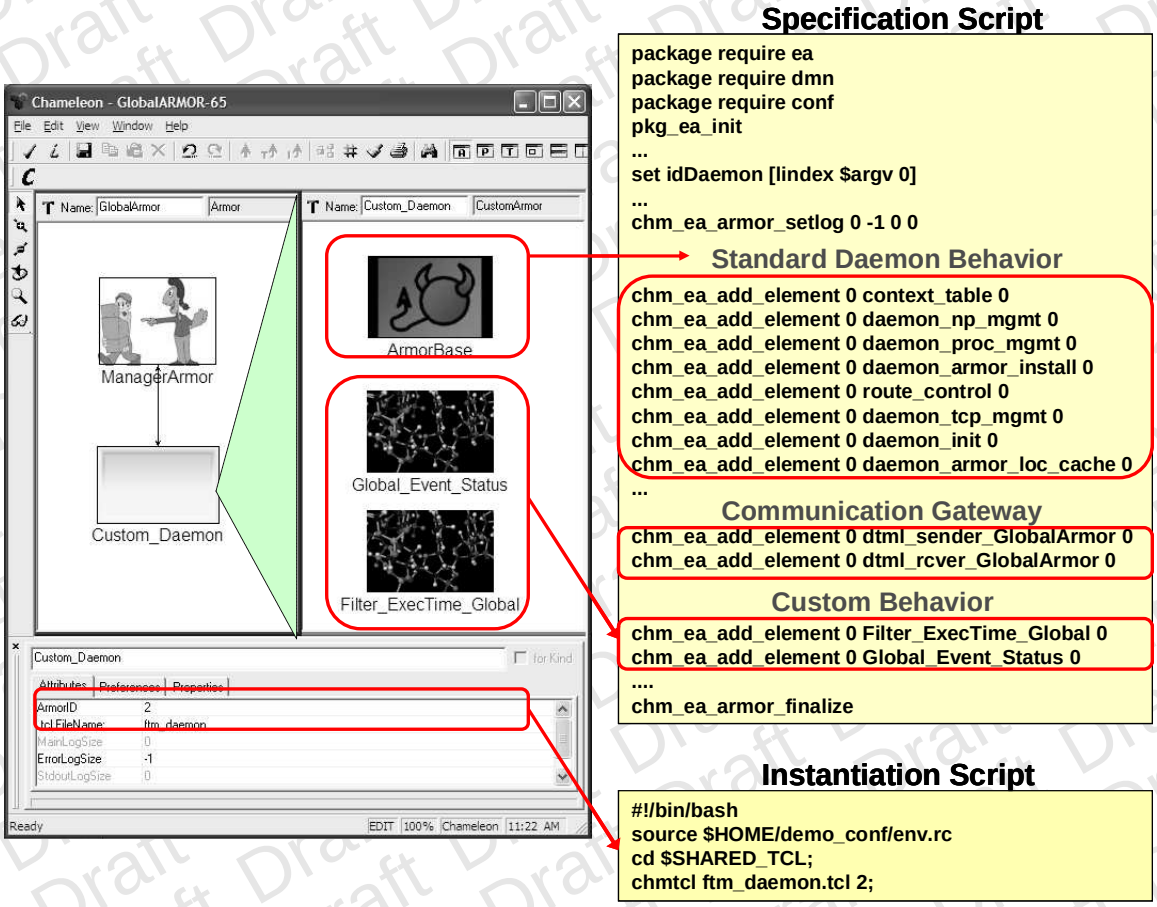

Figure 15 Mapping of Instantiation and Specification Scripts

Figure 15 illustrates the mapping of a Tcl script and an instantiation script from a CustomArmor object named Custom_Daemon in a FMML model. The Custom_Daemon object contains two custom mitigation behaviors that maps to corresponding instantiation code in the Tcl script. The instantiation code for the communication gateway is automatically added to the Tcl script. The ArmorBase of the Custom_Daemon indicates that it has the basic behavior of Daemon type fault managers in addition to the custom behaviors. So the basic behavior instantiation code is also added to the Tcl script. The instantiation script uses the ArmorID and Tcl FileName attributes for identifying the

unique id of the fault manager and the specification file when fault manager is instantiated.

Fault Managers Communication Gateway Generation

Publishing an Elvin message involves creating a message by specifying its various message fields and then broadcasting the message via an Elvin router. Subscribing for an Elvin message involves creating an subscription expression, providing a message callback function. Both processes involve calling marshalling and de-marshalling APIs from a library of general APIs that uses the Elvin-specific APIs internally.

Fault managers need to exchange messages with other components in the system to monitor and track failures in the system. The messaging scheme used in the system is a publish/subscribe mechanism through Elvin. Since ARMORs only recognizes ARMOR messages, messages sent using the Elvin protocols must be translated into an equivalent ARMOR message representation. Likewise, outgoing ARMOR messages must be translated into an equivalent Elvin message representation.

The two way translation processes is encapsulated within two custom gateways of a fault manager; one for sending and one for receiving Elvin messages. These two gateways are implemented as C++ classes with some fixed functions for instantiation themselves used by the ARMOR microkernel. The receiving gateway takes an Elvin message and encapsulates it within an ARMOR message. Custom mitigation behaviors can then proceed to process the ARMOR message. Likewise, any ARMOR message sent by custom mitigation behaviors are stripped, with the resulting Elvin message sent by the sending gateway via Elvin. Figure 16 illustrates the translation processes.

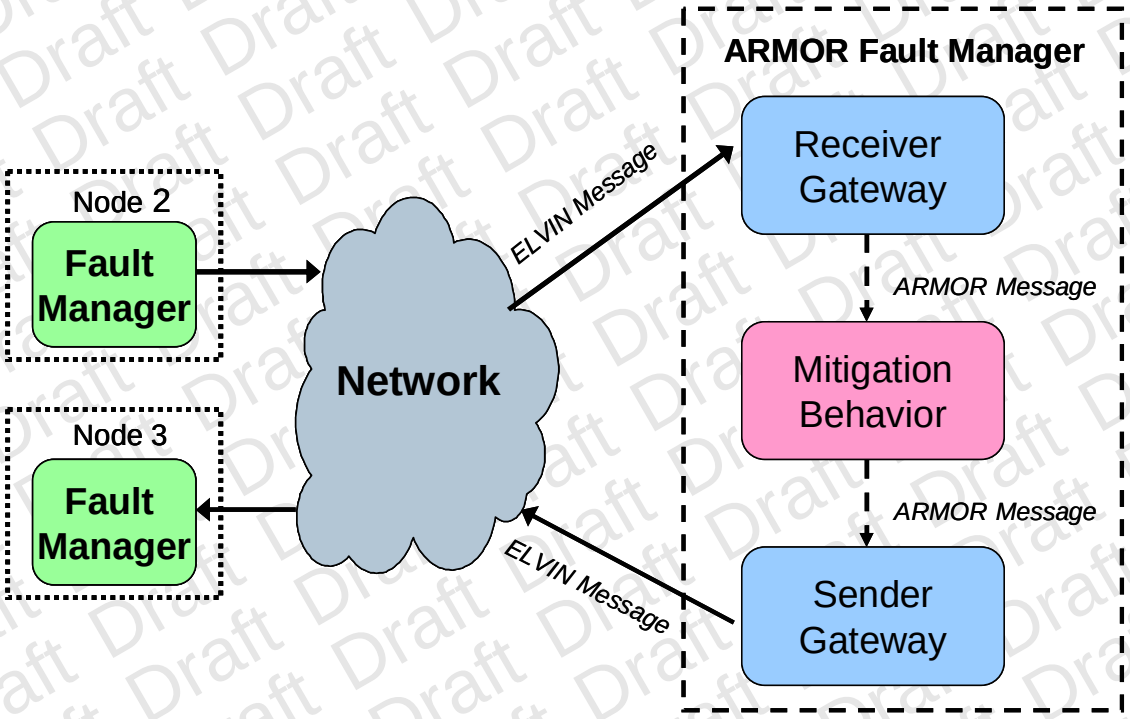

Figure 16 Message Translation Process Using Communication Gateways

Since not all fault managers receive and send the same messages, the gateways must be customized based the types of messages modeled in FMML. This information comes from the Transition objects in the FMML model. Only the non-Daemon type of fault managers will have the gateway behaviors.

Recall that Transition objects contain input and output messages associated with a state transition. The model translator translates all messages connected to a SubscribeChannel object with the type attribute equal to Elvin to a callback function and associates the callback with a corresponding Elvin subscription expression in the receiver gateway. The subscription expression comes from an attribute of a Message object.

The model translator translates all messages connected to a PublishChannel object with the type attribute equal to Elvin to a function call to strip out the Elvin message from an ARMOR message based on a defined message type in DTML and a publish API call

to publish message via Elvin. An ARMOR message's associated DTML message type is denoted by the Message_Type attribute of a message object.

Fault Mitigation Behavior Generation

The mitigation behaviors are modeled as hierarchical state machines in FMML. Each behavior model is implemented as a C++ class with three core parts: a state machine function, incoming message processing functions, and outgoing message specification functions.

Mapping of Messages

The Data objects in the model are transformed to class variables by the translator. Inside the incoming message processing functions, data of the incoming messages are assigned to class variables. The assignment is specified by the message assignment in FMML models. A similar mapping occurs for creating outgoing messages inside the outgoing message specification functions from models. Figure 17 illustrates the mapping from models to code.

Receiver Gateway Source File

```
class callback0:public CallbackArmor<CurrentCrossing>
{
  public:
    callback0(ControlsConnection *cc, void *p) :CallbackArmor<CurrentCrossing>(cc,
p) { }
      void invoke(CurrentCrossing* msg)
      {
        printf("Callback. Recieved CurrentCrossingmessage.\n");
        Receiver_Gateway *Rcver=(Receiver_Gateway*)this->element_ptr;
        mc_message_ct *pmc = new mc_message_ct;
        mc_bundle_ct *bundle= pmc->push_bundle();
        ...
        bundle->idArmorDest = Rcver->get_armor_id();
        pmc->push_op(bundle,MSG_CURRENT_CROSSING);
        pmc->set_field(MV_MSG_CURRENT_CROSSING,msg,sizeof(CurrentCrossing));
        Rcver->get_armor()->execute_context(pmc);
        ...
        }
};
...
thread_return_ct Receiver_Gateway::elvin_subscribe(void *ar)
{
SubscriptionData sub00;
  sub00.lang_option=equals;
  sub00.field = "category";
  sub00.ValueType =StringValue;
  sub00.un_val.sval ="Current_Crossing_Number";
  list0.push_back(sub00);
  callback0 *p0 = new callback0(er->con, er);
  er->con->subscribe( auto_ptr<callback0>(p0), list0);
...
}
```

Sender Gateway Source File

```
void Sender_Gateway::msg_ChangeFilterConfig_handler (mc_message_ct *pmc)
{
  printf("SENDING ChangeFilterConfig message\n");
  ChangeFilterConfig *msg = (ChangeFilterConfig*)pmc->
                get_data(MV_MSG_CHANGE_FILTER_CONFIG);
  msg->myHeader.region_id = this->region_id;
  msg->myHeader.node_id=this->node_id;
  con->publish(msg);
}
```

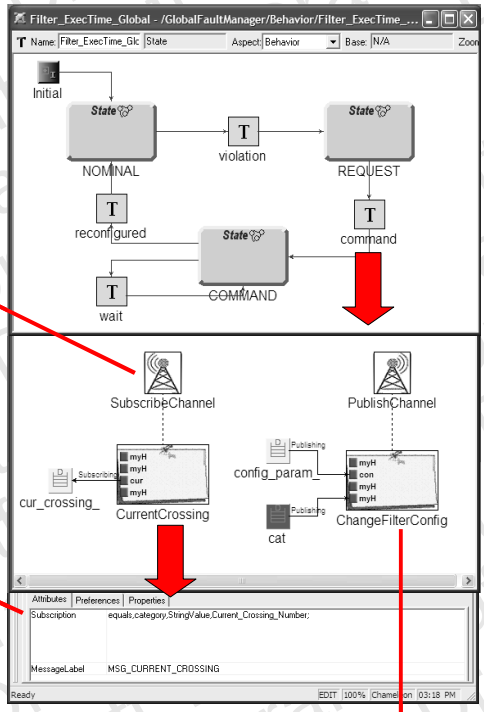

Figure 17 Mapping to Communication Gateway Source Files from Model

Mapping of Behaviors

The behavior model is mapped to a nested switch statement inside the state machine function of the custom Element. Nested switch statement was chosen for its simplicity, small memory footprint, and modularity. Other typical implementations of state machines include state table, state design pattern, or a combination of the other implementations [29]. Figure 18 illustrates the mapping process. The various mappings are listed below:

1. A set of defined states is collected from the model. An enum construct defining the states is created as a class variable in the custom Element class.
2. For each state, a case segment is created.
3. For each state, its outgoing transitions' guard condition is transformed to if and else if clauses under the corresponding case label. The action code of transitions are inserted inside the if and else if clauses. Action code is based on the action attributes of the transition objects from the behavior model.
4. For each outgoing message in a transition object in the model, a call to the corresponding outgoing message processing function is added inside the if or else if clause.

Behavior Source File

```
void Filter_ExecTime_Global_ct::run_statemachine
(mc_message_ct *pmc)
{
  cur_time_=time(NULL);
  switch(cur_state_)
  {
  case REQUEST:
    if(msg_current_crossing_flag)
    {
      printf("Current State= REQUEST\n");
      config_param_=cur_crossing_+N;
      next_state_=COMMAND;
      send_changefilterconfig(pmc); }
    break;
  }
...
  cur_state_ = next_state_;
}
...
void Filter_ExecTime_Global_ct::msg_current_crossing_handler
(mc_message_ct *pmc)
{
  printf("msg_current_crossing_handler Handler\n ");
  msg_current_crossing_flag = true;
  CurrentCrossing *msg = (CurrentCrossing*)pmc->
                get_data(MV_MSG_CURRENT_CROSSING);
  cur_crossing_ = msg->current_crossing;
  run_statemachine(pmc);
  msg_current_crossing_flag = false;
}
...

void Filter_ExecTime_Global_ct::send_changefilterconfig
(mc_message_ct *pmc)
{
  printf("send_changefilterconfig \n");
  ChangeFilterConfig *msg = new ChangeFilterConfig;
  msg->myHeader.category = "Change_All_Filter_Config";
  msg->config_param = config_param_;
  mc_bundle_ct *bundle = pmc->push_bundle();
  bundle->idArmorDest = get_armor_id();
  pmc->push_op(bundle, MSG_CHANGE_FILTER_CONFIG);
  pmc->set_field(MV_MSG_CHANGE_FILTER_CONFIG, msg,
sizeof(ChangeFilterConfig));
  delete msg;
}
...
```

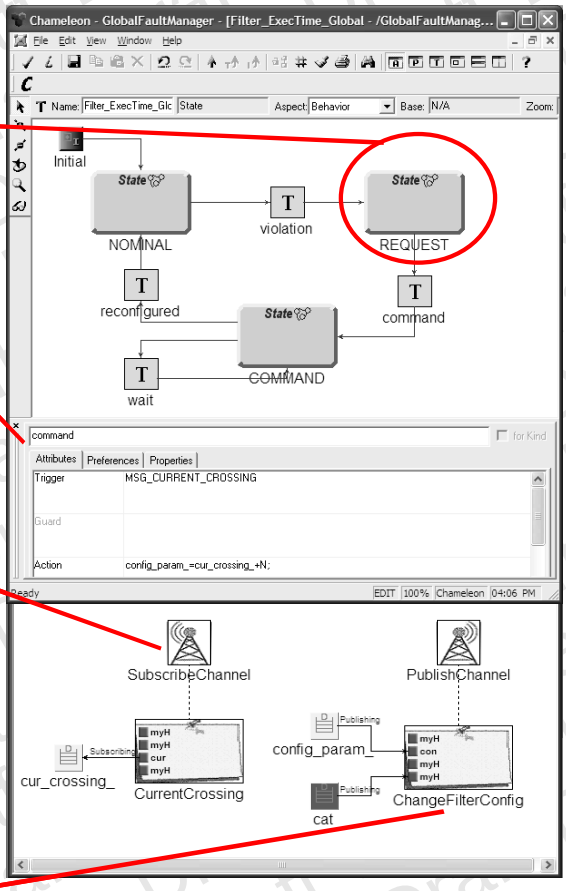

Figure 18 Mapping to Behavior Source File from Model

Summary

Through the model translation process, four types of artifacts are generated from Fault Mitigation Modeling Language models. Automatically generating these artifacts shields the designers of mitigation behaviors and fault management framework from the implementation details of Chameleon, which would require thorough knowledge in several programming languages and understanding of Chameleon execution environment. As system fault-tolerance requirements change, it is easier and less error-prone to evolve

and maintain the models accordingly and generate new artifacts than it is to modify the four types of artifacts themselves. Furthermore, generating artifacts saves development time since the translator makes sure that the models are built following a set of rules and the translator transforms information in all models the same way, artifacts are less prone to errors.

CHAPTER VI

CASE STUDY

As a proof-of-concept, several prototype BTeV Trigger systems were constructed using the tools developed. The demonstration consisted of a set of representative architectures (computing clusters of various sizes), a target runtime infrastructure (Chameleon + Elvin), a set of real and simulated application components (prototype physics applications), a set of models in the various DSML's, and the generated artifacts. Multiple versions were implemented to test the flexibility of the design tools and to assess their scalability. The tools were also evaluated by designers outside the tool development group, to assess ease-of-use factors. This chapter describes the prototypes in sufficient detail for a broad-stroke understanding of the application and its results. A set of performance results are also presented.

As mentioned in Chapter 1, the Trigger system is embedded inside a particle collider system providing three stages of experiment data filtering. Level 2/3 designates the second and third stages of data filtering. Timing deadlines are somewhat relaxed for Level 2/3 systems, and the filtering algorithms run much longer and are much more precise than of those in HEP Level 1 systems [22].

The system was tested with respect to its ability to recover from a number of fault scenarios. Dual processor worker nodes running a Fermilab distribution of Scientific Linux and a command and control user interface were used. Figure 19 shows an overall architectural view of the system.

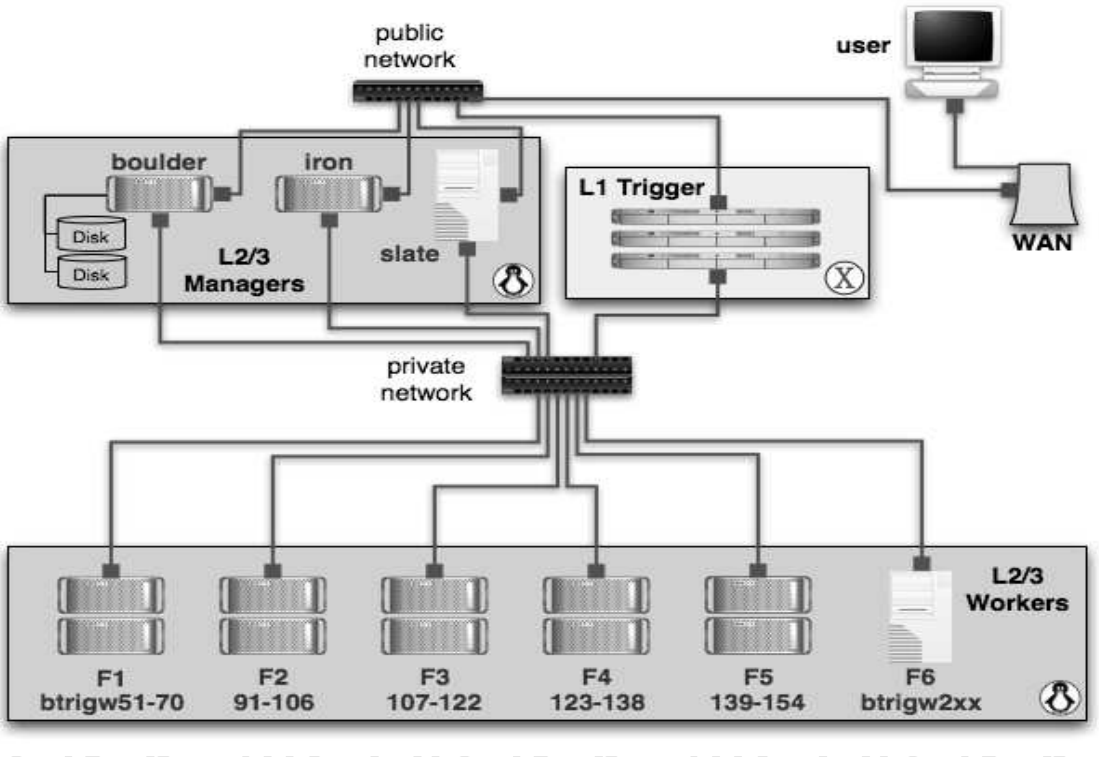

Figure 19 Level 2/3 Prototype System Setup (courtesy of H. Cheung, Fermilab)

The prototype system divides the worker nodes into three levels of hierarchy, Global, Region, and Local levels. A Global Manager residing on a GlobalNode governs all the Regions in the system. A Region consists of a RegionNode and several LocalNodes. Each Region is governed by a Regional Manager that resides on the Region Node. The Local level constitutes a node with an individual Local Manager to monitor two HEP applications running locally on that node [20]. The physics applications receive sample physics data from a Data Source application that is running on a separate node. The fault management architecture is illustrated in Figure 20.

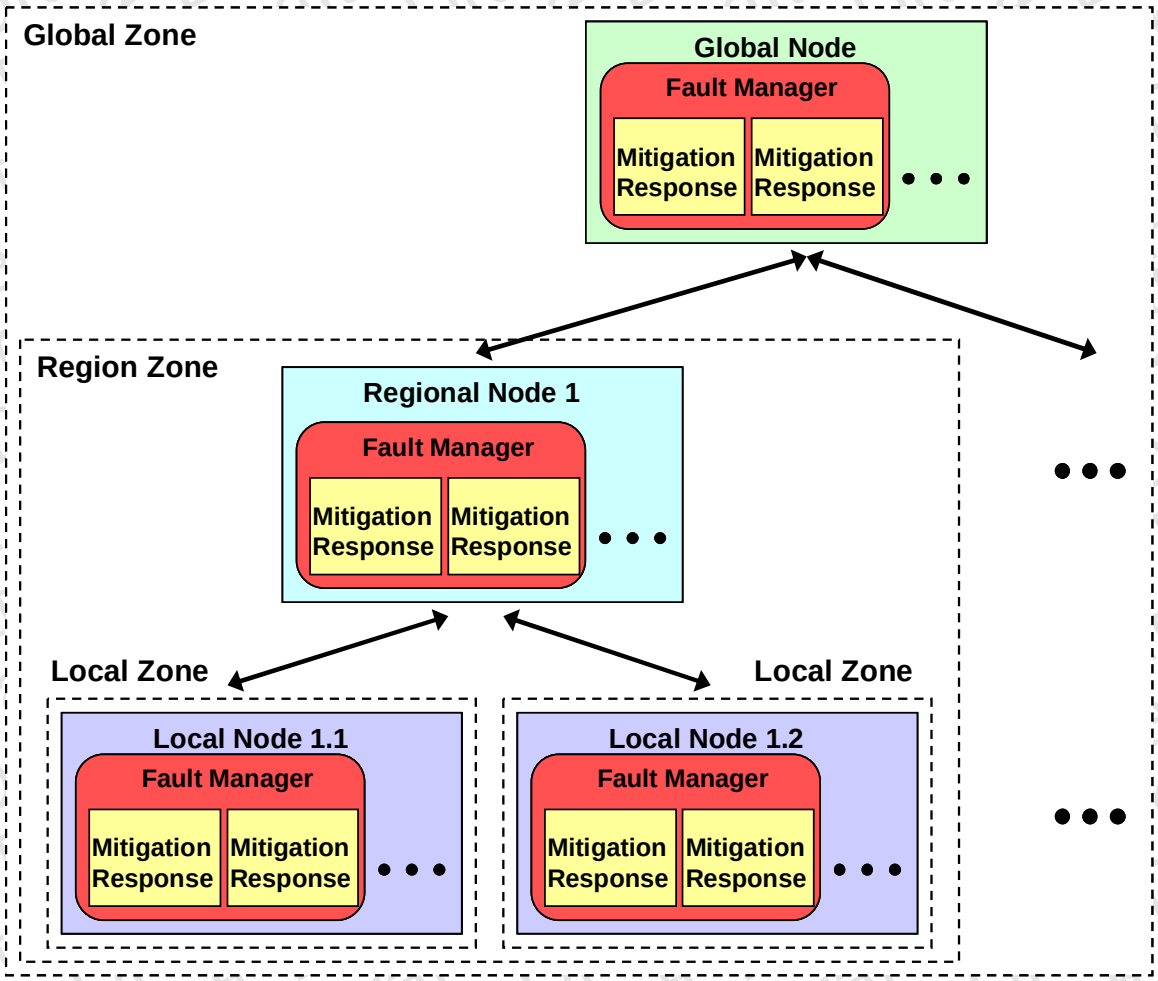

Figure 20 Fault Management Architecture in the Prototype System

Seven fault scenarios were constructed to represent typical fault conditions known to occur in HEP data processing systems. The fault scenarios are the following:

1. HEP application crash
2. Corruption of data stream for HEP application
3. HEP application caught in an infinite loop
4. HEP application exponential slowdown
5. HEP application memory usage violation
6. Individual HEP application processing time violation on LocalNode

7. Region execution time violation of HEP application

Various fault injection controls are present to the user on the control user interface for creating the fault scenarios listed above in the prototype system. The following sections look at the fault-mitigation behaviors and fault recovery timelines in two of the fault scenarios.

Corruption of Physics Data Stream

A common failure that can occur in HEP experiments is the temporal corruption of physics data leading to incorrect filtering results or even crash of physics application. The physics filtering applications are generally provided with the ability to detect corrupted data streams. It is extremely important to notify the physicists running an experiment upon detection of such fault so that the affected portions of the data stream can be marked as corrupt and ignored. The Local Manager should also be notified of this fault so that any trend regarding data stream corruption may be detected by the fault management system.

Presently, the Local Manager behavior focuses on detection and reporting of the data stream corruption to the system operator via user interface. There are two reporting modes available; a verbose mode and a terse mode. While every data corruption is reported in the verbose mode, a frequency of corruption occurrence is reported in the terse mode. The frequency of occurrence is simply the number of data corruptions within an adjustable time period. The default reporting mode is verbose but the mode can be easily changed via a control button on the user interface.

This scenario can be created by injecting a “Bad Data” fault from the user interface into the 65-node system. The Data Source upon receiving this command will

send a single corrupt data to a physics application on a local node. Figure 21 shows the event timeline during the monitoring process of corrupt data. The legend for the events is in Table 1.

Table 1 Event Legend for Data Stream Corruption Fault Scenario

Event Label	**Event Description**
LM1	Received processing time report form HEP application
LM2	Detection of HEP application encountering corrupt data
LM3	Notification to user interface of detection

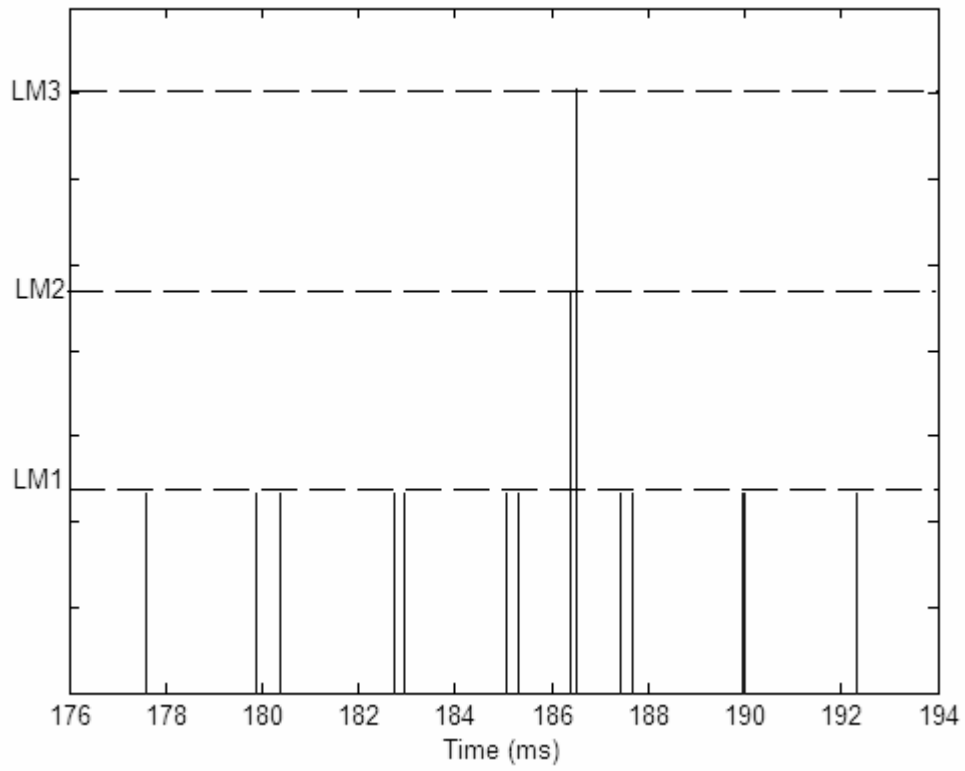

Figure 21 Mitigation Time Plot of Data Stream Corruption by Local Fault Manager

According to the timeline, the detection (LM2) and reporting of corruption (LM3) takes around 0.1 milliseconds. This time is affected both by ARMOR's internal messaging system and Elvin. Currently, mechanisms for detecting trends in data corruption have not been included in this behavior. This behavior will be enhanced to include such mechanism in the future.

This scenario tests how affective the Local Manager is able to detect and report the detection of bad physics data streams. Since it does not have control over the physics data that are generated, timely detection and reporting is necessary to notify the system administrator who has the authority to perform various checks on equipments and detector environment to figure out the cause of the corruption.

Region Timing Violations During Data Filtering

A typical occurrence is a decrease in the overall Level 2/3 data filtering system throughput due to the HEP application taking too long in processing individual data. HEP applications employ filtering algorithms whose execution times are data dependent. As conditions in the particle accelerator change due to a higher density of particle collisions, the characteristics of collision data also changes in turn affecting the behavior of the filtering application. The HEP experiments run at a constant physical periodicity, so slowing down the data acquisition is not considered a viable mitigation behavior. Instead, physicists can reconfigure certain parameters of the HEP applications such that they have shorter execution times.

In the prototype system, this timing violation is detected when the average processing time of one or more regions is higher than a reasonable threshold. Once a violation is detected, an immediate reconfiguration of HEP application in all regions is initiated. The behavior to detect and mitigate such a violation requires cooperation among the fault managers from all three levels of hierarchy. As Local Managers have only a small view of the entire system, they do not know the processing times of HEP applications on other nodes. Regional Managers have knowledge of HEP application processing times in its control zone from Local Managers to detect violation but they do not have the authority to initiate a global HEP application reconfiguration. The Global Manager does not detect execution time violation of a region but it does have the authority to initiate a global HEP application reconfiguration once it is notified of such violation.

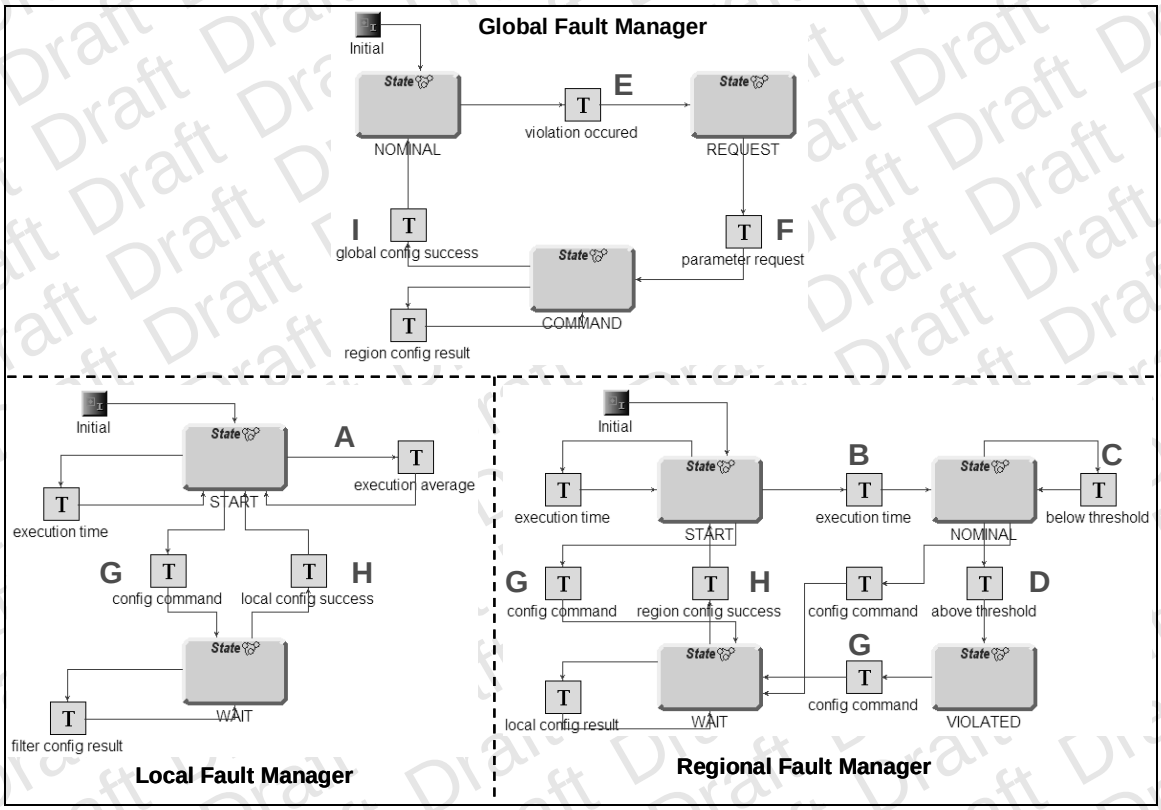

Figure 22 Mitigation Behaviors of Local, Regional and Global Fault Managers

Figure 22 shows the mitigation behavior model of the Local Manager, Regional Manager and Global Manager for this particular type of fault. The specific detection and mitigation steps are described below along with the corresponding transitions from Figure 22:

1. Local Managers of a region compute the average of every ten data execution times of the physics applications on its node and send the average to the Regional Manager. (A)
2. The Regional Manager computes a moving average of the execution times that it receives from the Local Managers in its region. (B)

3. If the regional execution time is below a threshold then repeat Step 2. (C) Otherwise, send a violation message to the Global Manager and go to a VIOLATED state. (D)
4. The Global Manager upon receipt of the violation message, requests a processing parameter from the data source. (E)
5. The Global Manager sends a command message to all the physics application to change their configuration. Depending on this processing parameter, not all physics applications need to change their configuration. (F)
6. The Local Managers and Regional Managers transition to a WAIT state upon receipt of the configuration change message. (G)
7. In this WAIT state, the Local Managers wait for reconfiguration result messages from the physics applications. The reconfiguration result messages are propagated up the management hierarchy to the Global Manager. (H) Upon receipt of reconfiguration messages from all regions, the Global Manager returns to a NOMINAL and the scenario are complete. (I)

Using the built-in logging feature of the custom mitigation behavior, the time of occurrence for the above events was recorded. Results from several runs of the scenarios were compared and the mitigation times were similar. Timing data from one of the runs is used to construct a timeline that shows the sequence of events from violation discovery to the regional reconfiguration event propagations in Figure 23. Table 2 gives the legend for events in Figure 23. The events are represented by bars labeled with event type. The color

of the bar indicates the management level; green for Local level, blue for Regional level and red for Global level. Please note that the clocks on the node where the fault managers reside have not been synchronized with each other.

Table 2 Event Legend for Regional Execution Time Violation Fault Scenario

Event Label	**Event Description**
LM1	Detection of command to reconfigure HEP application
LM2	Notification that HEP application has been reconfigured
RM1	Detection of slow regional processing time
RM2	Detection of command to reconfigure HEP application
RM3	Notification of all HEP applications in the region has been reconfigured
GM1	Detection of reconfiguration request
GM2	Requesting configuration parameter
GM3	Received configuration parameter
GM4	Command to initiate HEP reconfigurations in the system
GM5	Notification that HEP applications has reconfigured in the system

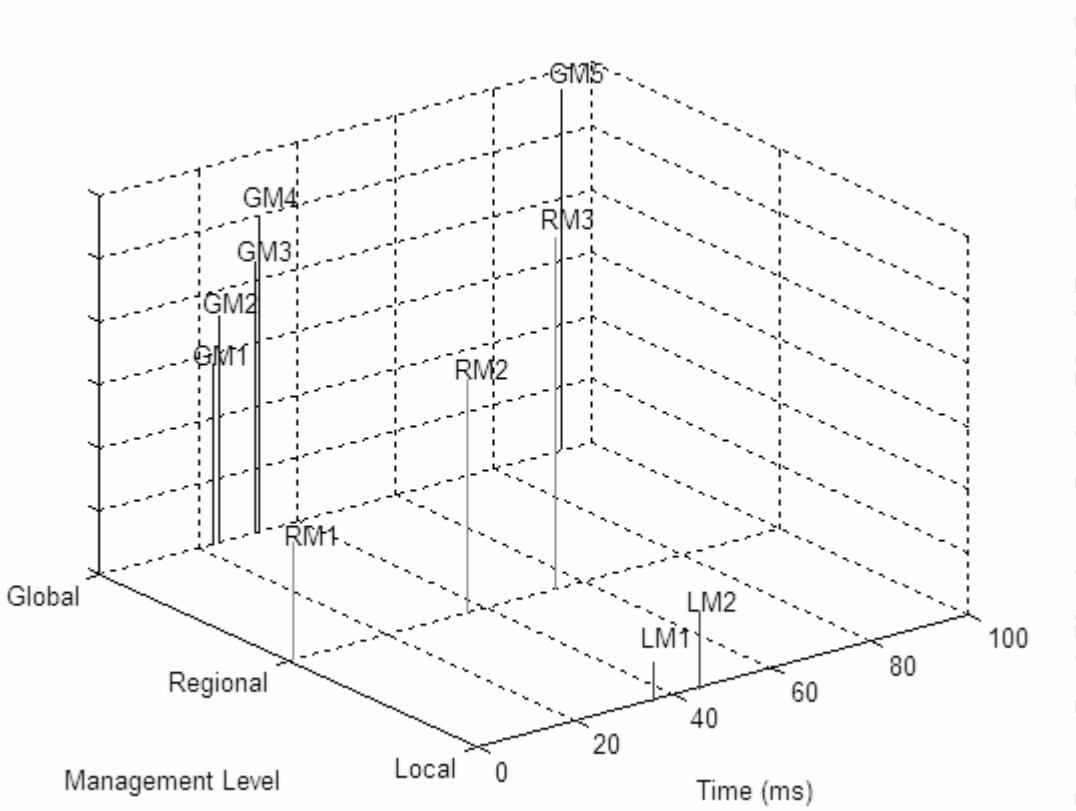

Figure 23 Mitigation Time Plot of Regional Execution Violation by Three Levels of Fault Managers

Figure 23 shows the normalized times at which detection and mitigation events occur during this scenario. The sequence of events is (RM1, GM1, GM2, GM3, GM4, RM2 & LM1, LM2, RM3, and GM5). From the figure, we can see that about 22.07 milliseconds pass between detection of timing violation by the Regional Manager (RM1) and start of Global Manager's corrective actions (GM2). The time it took for the initiation of reconfiguration initiation (GM4) to the finish of reconfiguration and reporting of all HEP applications (GM5) took about 61.47 milliseconds.

The prototype system for this timeline consisted of 65 nodes. The same scenario can be run for different system sizes (5, 35, 65, 250 and more), the resulting data can be compared to study the performance of the hierarchical fault management system in

mitigating behavior that require cooperativeness among management levels. This would certainly be a part of future work.

Evaluation of Case Study

Using the different Domain Specific Modeling Languages, we were able to model the complete Trigger prototype systems, generate implementation and runtime software, and execute the system. Scalability of the prototype system from 5, 17, 35, to 65 computing nodes was easily achieved using different architecture specification models expressed using SIML. Mitigation behaviors are easily expressed using state machine concepts and artifacts generated by invoking the corresponding translator relieving designers from burden of hand-coding them.

Designers outside the tool development group agreed that developing the prototype system using the model-based tools was more intuitive as it provides domain-specific concepts that are context specific. Configuration is also made easier using graphical constructs that represent these concepts. Automatically generated software reduces development time since software do not need to be hand-coded, allowing designers to spend more time on designing rather than coding. The performance of generated software was comparable to hand-generated software.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

Conclusions

Designing large-scale, real-time embedded systems is not a trivial task due to the size and real-time constraints on the system. Fault tolerance and reliability requirements further complicate the process. Redundancy based fault-tolerance techniques may not be feasible, so systems must have the ability to detect, monitor, and correct faults automatically. This is possible by employing a hierarchical fault management framework that enables automatic fault mitigation.

Designing and implementing the management framework requires extensive knowledge of the runtime environment. To aid designers who may not have this knowledge and to speed up the design cycle, and to easily explore various alternative solutions, a model-based approach is used that uses domain-specific languages and concepts. Using this approach, a modeling tool was developed that has a domain-specific modeling language and a model translator.

The domain-specific modeling language enables system designers to implement customized fault-mitigation behaviors to target specific faults and specify interaction between the fault managing entities and other components in the system. The model translator translates the models to artifacts that are used to configure, specify, and deploy the hierarchical fault management framework.

This tool was integrated with other modeling languages described in Chapter 3. The tool suite was used to design, implement, and deploy a prototype Trigger system for

the BTeV experiment. Several fault scenarios were implemented for the Trigger system. The behaviors involved mitigation actions at the local fault manager level as well as coordinated behaviors between different management levels. The system was successfully demonstrated at 2005 IEEE Real-Time and Embedded Technology and Applications Symposium.

The model-integrated approach proved effective for system implementation. The entire system software was generated from the MIC tools. Performance was shown to be adequate, comparable to hand-generated code. System scalability was demonstrated by migrating architectures from 5 nodes to 65 nodes within time period of an hour.

General user comments were positive. Users were able to completely redefine the fault mitigation behavior with very little knowledge of the underlying implementation. Approximately 7 mitigation behaviors were implemented and tested within a 2 week period. The visibility of the behavior as models also served to make the system behavior much more understandable, and therefore manageable.

Future Work

Several areas related to this research need to be explored. First of all, the current tools simplify the definition of behaviors and automate the production of code to implement these behaviors. As these behaviors become complicated via large individual state machines and cooperative, distributed state machines, the effective behavior becomes very difficult to understand. Tools for verifying behavior will help in producing reliable, well-understood systems. The state machine based formalism for describing mitigation behaviors allows static model checking techniques to be applied for analyzing the behaviors. Some of the existing tools that can be used include SMV, SPIN and

UPPAL. Fault Mitigation Modeling Language models can be transformed to equivalent formats used by one of the verification tools using a model translator.

Currently, we are not using any simulation tools to predict the mitigation behaviors. Use of simulation tools, like Simulink Stateflow, will enable better design during design time. Since the mitigation models already uses Statechart notations, only some modification will be needed to transform FMML models to an equivalent Stateflow model for simulation. However, a simulation framework would need to be developed so that Stateflow models can be directly plugged into the framework. The framework needs to have a fault injector so that various faults could be injected during simulation.

The prototype system has been limited to ~65 nodes. The tool should be used to model, generate, and analyze more complex behaviors on prototype systems with larger number of nodes. Observations can be made to study how well the tool allows the behaviors to be scaled up according system size.

Performance measurements of the system have been relatively limited. As of now, mitigation action timings have been taken for the 65 node prototype system. More extensive performance measures on mitigation timings should be taken for studying how performance of the fault management framework relates to system size and improvements that could be made to the behaviors for minimal mitigation time. Additionally, this can help to design behaviors that can fulfill real-time requirements. So that real-time mitigation deadlines can be incorporated into system mitigation behaviors. These performance measures can be used in the simulations described above.

REFERENCES

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

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

[3] J. Kowalkowski, "Understanding and Coping with Hardware and Software Failures in a Very Large Trigger Farm", 2003 Computing in High Energy and Nuclear Physics, La Jolla, CA, USA, March 2003.

[4] S. Neema, T. Bapty, S. Shetty, S. Nordstrom, "Developing Autonomic Fault Mitigation Systems", *Journal of Engineering Applications of Artificial Intelligence Special Issue on Autonomic Computing and Grids*, Elsevier 2004.

[5] J. Sztipanovits, G. Karsai, "Model-Integrated Computing", *IEEE Computer*, vol. 30, no. 4, pp. 110-112, April, 1997.

[6] G. Nordstrom, J. Sztipanovits, G. Karsai, A. Ledeczi, "Metamodeling – Rapid Design and Evolution of Domain-Specific Modeling Environments", *Proceedings of the IEEE ECBS '99 Conference*, Nashville, TN, pp. 68-74, April 1999.

[7] G. Karsai, J. Sztipanovits., A. Ledeczi, T. Bapty, "Model-Integrated Development of Embedded Software", *Proceedings of the IEEE*, Vol. 91, Number 1, pp. 145-164, January, 2003.

[8] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason IV, G. Nordstrom, J. Sprinkle, P. Volgyesi, "The Generic Modeling Environment", *Workshop on Intelligent Signal Processing*, Budapest, Hungary, May 17, 2001.

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

[10] Z. Kalbarczyk, R. K. Iyer, S. Bagchi, K. Whisnant, "Chameleon: A Software Infrastructure for Adaptive Fault Tolerance", *IEEE Transaction on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 560-579, June 1999.

[11] S. Bagchi, B. Srinivasan, K. Whisnant, Z. Kalbarczyk, R. K. Iyer, "Hierarchical Error Detection in a Software Implemented Fault Tolerance (SIFT) Environment", *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 2, pp. 203-224, 2000.

[12] K. Whisnant, Z. Kalbarczyk, R. K. Iyer, "A System Model for Dynamically Reconfigurable Software", *IBM Systems Journal, Special Issue on Autonomic Computing*, vol. 42, no. 1, pp. 45-49, 2003.

[13] IBM Autonomic Research Webpage, http://www.research.ibm.com/autonomic

[14] M. Hiller, "Software Fault Tolerance Techniques from a Real-Time Systems Point of View: An Overview", *Technical Report* No. 98-16, Dept. of Computer Engineering, Chalmers University of Technology, Sweden, 1998.

[15] J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, W. N. Mills III, and Y. Diao, "ABLE: A toolkit for building multiagent autonomic systems", *IBM Systems Journal*, Volume 41, Number 3, 2002.

[16] M. Parashar, "AutoMate: Enabling Autonomic Applications on the Grid", *Autonomic Computing Workshop, 5th Annual International Active Middleware Services Workshop*, Seattle, WA, 2003.

[17] Z. Li, H. Liu, M. Parashar, "Enabling Autonomic, Self-managing Grid Applications", *Proceedings of the 4th International Workshop on Grid Computing (Grid 2003)*, Phoenix, AZ, USA, IEEE Computer Society Press, pp 34 - 41, November 2003.

[18] B. Khargharia, et. al, "vGrid: A Framework for Building Autonomic Applications", *Autonomic Applications Workshop,* 10th International Conference on High Performance Computing (HiPC 2003), Hyderabad, India, December 2003.

[19] S. Shetty, S. Nordstrom, S. Ahuja, D. Yao, T. Bapty, S. Neema, "System Integration of Large Scale Autonomic Systems Using Multiple Domain Specific Modeling Languages", *Engineering of Autonomic Systems*, *IEEE ECBS Workshop on Engineering of Autonomic Systems*, Greenbelt, MD, April, 2005.

[20] D. Yao, S. Neema, S. Nordstrom, S. Shetty, S. Ahuja, T. Bapty, "Specification and Implementation of Autonomic Large-Scale System Behaviors Using Domain-Specific Modeling Language Tools", *Proceedings of the 2005 International Conference on Software Engineering Research and Practice*, Las Vegas, NV, June, 2005.

[21] S. Ahuja, D. Yao, S. Neema, T. Bapty, S. Shetty, S. Nordstrom, "Dynamically Reconfigurable Monitoring in Large Scale Real-Time Embedded Systems", *Proceedings of the IEEE SoutheastCon*, Fort Lauderdale, FL, April 2005.

[22] S. Ahuja, et. al, "RTES Demo System2004", *Special Issue on High Performance, Fault Adaptive, Large Scale Embedded Real-Time Systems,* ACM SIGBED Review, July 2005, Volume 2, Number 3.

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

[24] S. Shetty, “Towards Developing Tools and Technologies for Modeling Faults in Large Scale, Real Time, Reactive Embedded Systems”, *M. S. Thesis*, Vanderbilt University, 2004.

[25] S. Shetty, S. Neema, et. al, “Model-Based Self-Adaptive Behavior Language for Large Scale Real-Time Embedded Systems”, IEEE Conference on Engineering of Computer Based Systems, Brno, Czech Republic, 2004.

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

[27] M. Hiller, “Software Fault Tolerance Techniques from a Real-Time Systems Point of View: An Overview”, *Technical Report No. 98-16*, Department of Computer Engineering, Chalmers University of Technology, Sweden, 1998

[28] Elvin Webpage http://elvin.dstc.edu.au

[29] Miro Samek. *Practical Statecharts in C/C++*. CMP Books, 2002.