MODELING AND SIMULATION OF LARGE SCALE REAL TIME EMBEDDED SYSTEMS

By

Divya Vashishtha

Thesis

Submitted to the Faculty of the Graduate School of Vanderbilt University in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE in Electrical Engineering

May, 2004

Nashville, Tennessee

Approved:

Professor Theodore Bapty
Professor Sandeep Neema
To my family
This thesis was made possible by the contribution of a lot of people at various stages. I would like to thank Professor Theodore Bapty for the continuous inspiration and support that I got from him while working on my thesis. Working with him was a great learning experience. I also want to thank Dr. Sandeep Neema, for helping me out at various stages of my research. I owe tremendous gratitude to both of them for their patient support and encouragement during the research process.

I would like to thank Brandon Eames for patiently answering my numerous research related questions, and Shweta and Steve for all the discussions that I had with them and the support I got from them for my work.

I want to thank Mummy and Papa for their faith in me and for all they did to support me in every way; I could not have accomplished any of this had it not been for their careful nurturing. Thanks to Akshay for being there for me always. I also want to thank my family for their love and support.

This research was sponsored by the National Science Foundation under the grant number ACI-0121658, as part of the BTeV project at Fermi National Lab.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEDICATION</td>
<td>ii</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>iii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF ABBREVIATIONS</td>
<td>viii</td>
</tr>
</tbody>
</table>

Chapter

I. INTRODUCTION ...............................................................................................................1
   - Embedded Systems ................................................................. 1
   - Simulation ................................................................................. 2
   - Model Based System Design and Synthesis ......................... 2
   - The BTeV Project ........................................................................ 4
   - The Problem Statement .......................................................... 5
   - Overview of the Report ............................................................ 5

II. SIMULATIONS: A DISCUSSION.....................................................................................7
   - Benefits of Simulation ............................................................ 7
   - What to Simulate? ................................................................. 8
   - Steps Involved in a Simulation ............................................... 8
   - Types of Simulation ............................................................... 9
   - Simulation Software .............................................................. 10
   - Matlab Simulink and Stateflow ............................................. 10

III. THE SYSTEM, FAULTS AND FAULT MITIGATION .................................................12
   - The BTeV Experiment and Trigger Algorithms .................. 12
   - Faults, fault mitigation requirements ................................. 13
   - Experimental Data Acquisition and Analysis ....................... 14
   - GME in the Development Cycle .......................................... 16
   - Where and How Simulation comes into Picture? ............... 20

IV. THE SIMULATION ..........................................................................................................21
   - Component Model Library ...................................................... 21
   - Discussing Simulation Components in Detail ...................... 22
   - Simulating the Hardware ....................................................... 22
   - Simulation of Data Communication Structure .................... 26
     - The Channel ................................................................. 26
     - The Stream .............................................................. 29
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulating the OS</td>
<td>32</td>
</tr>
<tr>
<td>The Scheduler</td>
<td>32</td>
</tr>
<tr>
<td>Simulating the Software Processes</td>
<td>33</td>
</tr>
<tr>
<td>V. AUTOMATIC GENERATION OF SIMULATION MODELS</td>
<td>35</td>
</tr>
<tr>
<td>Why automatic generation</td>
<td>35</td>
</tr>
<tr>
<td>Overview of Matlab API</td>
<td>36</td>
</tr>
<tr>
<td>Mapping of Models to Simulation Components</td>
<td>39</td>
</tr>
<tr>
<td>Interpreter Structure</td>
<td>39</td>
</tr>
<tr>
<td>Basic Classes in the interpreter</td>
<td>39</td>
</tr>
<tr>
<td>Logic flow in the Interpreter Code</td>
<td>41</td>
</tr>
<tr>
<td>VI. SAMPLE TEST CASE, RESULTS AND DISCUSSION</td>
<td>43</td>
</tr>
<tr>
<td>Case Study 1: System Failure Simulation</td>
<td>43</td>
</tr>
<tr>
<td>Case Study 2: Simulation Generation from GME Models</td>
<td>51</td>
</tr>
<tr>
<td>VII. CONCLUSIONS AND FUTURE WORK</td>
<td>55</td>
</tr>
<tr>
<td>Conclusion</td>
<td>55</td>
</tr>
<tr>
<td>Future Work</td>
<td>55</td>
</tr>
<tr>
<td>Generalization</td>
<td>56</td>
</tr>
<tr>
<td>Hardware Level Changes</td>
<td>56</td>
</tr>
<tr>
<td>Kernel Level Changes</td>
<td>56</td>
</tr>
<tr>
<td>Execution Level Changes</td>
<td>57</td>
</tr>
<tr>
<td>Modeling Fault Scenarios</td>
<td>57</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>58</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>1.</td>
<td>The Multi-Graph Architecture .........................................................4</td>
</tr>
<tr>
<td>2.</td>
<td>The BTeV Experimental Setup .............................................................12</td>
</tr>
<tr>
<td>3.</td>
<td>Trigger Algorithm and Data Acquisition System .................................15</td>
</tr>
<tr>
<td>4.</td>
<td>Software aspect of GME Models ..........................................................17</td>
</tr>
<tr>
<td>5.</td>
<td>Hardware Aspect of GME Models ..........................................................18</td>
</tr>
<tr>
<td>6.</td>
<td>Sample mitigation strategy specification ..............................................19</td>
</tr>
<tr>
<td>7.</td>
<td>Single Processor Composition .............................................................23</td>
</tr>
<tr>
<td>8.</td>
<td>Processor communicating with other processors ....................................23</td>
</tr>
<tr>
<td>9.</td>
<td>Communication Port Block Diagram ....................................................24</td>
</tr>
<tr>
<td>10.</td>
<td>Behavior of PAU ....................................................................................25</td>
</tr>
<tr>
<td>11.</td>
<td>Matlab diagram of the Communication Port Model ....................................26</td>
</tr>
<tr>
<td>12.</td>
<td>Dataflow in the Processor ....................................................................27</td>
</tr>
<tr>
<td>13.</td>
<td>Stateflow Diagram of the Input Channel ...............................................28</td>
</tr>
<tr>
<td>14.</td>
<td>Out Channel Stateflow Model ................................................................29</td>
</tr>
<tr>
<td>15.</td>
<td>Stateflow Diagram of a Local Stream ....................................................30</td>
</tr>
<tr>
<td>16.</td>
<td>Stateflow diagram of an interprocessor In Stream ..................................31</td>
</tr>
<tr>
<td>17.</td>
<td>Stateflow diagram of Destination Stream ..............................................32</td>
</tr>
<tr>
<td>18.</td>
<td>Stateflow Model of Scheduler ...............................................................33</td>
</tr>
<tr>
<td>19.</td>
<td>Generic Behavior of a Process ..............................................................34</td>
</tr>
<tr>
<td>20.</td>
<td>The Simulation Generation Process .......................................................36</td>
</tr>
<tr>
<td>21.</td>
<td>Stateflow Hierarchy ..............................................................................38</td>
</tr>
<tr>
<td>22.</td>
<td>The Top view of Simulation Model .......................................................43</td>
</tr>
<tr>
<td>23.</td>
<td>The Communication Protocol Model ......................................................44</td>
</tr>
</tbody>
</table>
24. Looking inside processor type 1.................................................................45
25. Source process behavior model.................................................................46
26. Looking inside Processor type 2 ...............................................................46
27. Model of a destination process.................................................................47
28. Stream model with random fault generator............................................48
29. Behavioral model of stream with error ....................................................49
30. Error plot for the stream........................................................................50
31. Number of Filled Slots in the Stream....................................................50
32. Hardware Aspect of GME model............................................................51
33. Software Aspect of GME model...............................................................52
34. Simulation model, Top Layer.................................................................53
35. Processor simulation..............................................................................53
LIST OF ABBREVIATIONS

MIC – Model Integrated Computing
MGA – Multi-Graph Architecture
GME – Generic Modeling Environment
DSP – Digital Signal Processor
FPGA – Field Programmable Gate Array
PAU – Port Arbitration Unit
CPCR – Communication Port Control Register
DMA – Direct Memory Access
FIFO – First In First Out
CHAPTER I

INTRODUCTION

Increasingly more applications are being based on embedded systems frameworks. With an increase in the complexity of the system, designing these systems is becoming increasingly challenging. Such systems have a large number of real-time constraints on system performance. Satisfying all the system constraints imposes a strict requirement of accuracy in design and implementation of real-time embedded systems.

Simulation is a valuable capability that provides a means to design, test and evaluate a system before the actual implementation. A good simulation of a system can provide risk free analysis of not only the system in nominal conditions, but also in fault conditions which are difficult to create and test in the actual system. Furthermore, simulation can be used to develop and scale up systems before the physical hardware is available. Simulation is becoming an integral part of system design.

There is a growing design trend towards developing systems with high-level design tools. Model-Integrated Computing (MIC) is one of these approaches that permits graphical specification of systems, with automatic software synthesis. This approach can significantly lower the efforts required to design and implement complex systems. MIC-based system development requires design and test cycles. Integration of simulation tools with model-integrated design tool set provides an excellent framework for implementing the test part of the cycle in developing complex embedded systems. The ability to directly simulate MIC designs is the focus of this work.

This chapter first presents a brief overview of the general concepts that are involved in the research, followed by a very brief description of the BTeV project, for which this simulation is being developed. The chapter ends with an overview of this thesis.

Embedded Systems

Embedded systems are a class of computer-based systems consisting of both hardware and software designed for a specific application. Embedded systems are reactive real-time systems. The main concerns of a reactive system are timeliness and correctness. The very
simplest embedded systems are capable of performing only a small set of functions to meet a single pre-determined purpose. In more complex systems, many functions are integrated, often interacting in non-trivial ways. Multiple modes of operation, with varying behaviors is also characteristic of complex systems. The ability to be reprogrammed for these multiple modes implies that the same system can be used for a variety of applications.

The class of embedded system of interest for this research is a large, interconnected network of Digital Signal Processors. The application involves analysis and real-time processing of vast amounts of data. In addition to the normal embedded systems requirements, this system, by virtue of its application needs to be fault-tolerant and fault-adaptive.

The size of the system imposes a big challenge to the designer to design an error free system with the ability to withstand failures. Simulation becomes a very powerful tool in this case. If a simulation capability is available, system designers can test their designs even before implementing those on the actual hardware. These designs can be used to test known failure scenarios of the system, as well as to study other possible fault cases.

Simulation

Use of computer simulation in system design has been in common practice since the 1950s. Simulation is the art and science of representing a process or system in the form of an executable model which can be useful in decision making [1] [2] [3] [4] [5]. Computer based simulation modeling involves abstraction and simplification of the factors that are believed to play a crucial role in the operation of the system. This process of modeling requires correct data and information of the system based on the purpose of the simulation. A more detailed discussion on simulation is given in chapter II of this thesis.

Various tools, with varying simulation capabilities are available in the market. This research required a programmable simulation environment, with support for continuous and discrete simulation. Visualization of results was also needed. For the purpose of research, Matlab Simulink™ and Stateflow™ software provided the necessary capabilities.

Model Based System Design and Synthesis

The system under consideration is of a very large scale. Systems will range from 16 processors (our physical demonstration system) to the full-scale system of 2500 processors.
Accurately designing the entire system and maintaining hardware/software configurations is a big challenge. The traditional approach to system design is to first gather system information and perform a high-level design. Simulation is used to verify system design properties. Finally, implementation of the system is done by programmers manually. Such a method is feasible for simple embedded systems, where the level of detail is limited. As systems get more complex, this method of system design tends to be less practical. With increase in complexity of the system, a higher level system design methodology is required. Model-based system design is one approach to managing this complexity. A model is an abstraction of system or its components. A domain specific modeling environment [6] [7] matches the application domain concepts to the concepts in the language. This has advantages over a common modeling environment that it not only better captures the system information but also make the task of modeling easier for the designers, since they can focus on modeling the system at a higher level.

The domain specific modeling environment consists of three main components, namely:

- The meta-model: This is the formal definition of a modeling language. This definition consists of various aspects of the graphical language, such as components of the system, their connections and interconnections, attributes of various components and the constraints on the system.

- The modeling environment: This is a graphical environment implementing the language specified in the Meta-model. It is the tool where the modeler can design the system.

- The Interpreter: This is the tool that interprets the models and synthesizes executable code. The interpreter traverses through the models hierarchy and extracts semantic knowledge from the system. The interpreter hides from the modeler the low-level details by automatically synthesizing all the low-level code, data, configurations, and other structures needed to run the system.
Figure 1 shows the Multi-Graph Architecture (MGA) developed at the Institute for Software Integrated Systems, Vanderbilt University which provides a framework for modeling domain specific applications.

The **BTeV Project**

BTeV is a high energy physics experiment [8] [9], being conducted at Fermi National Laboratory to study particle collision. The Real Time Embedded System (RTES) group is a part of the BTeV project that is involved in developing aspects of the trigger and data acquisition system for the experiment. Developing this system involves creating new tools and methodologies for designing and implementing very large scale real-time embedded systems. These systems are characterized by high computational performance, high availability and dynamic reconfiguration. They must be maintainable and evolvable. The data collected from the experiment by pixel detectors is processed by the trigger application. The trigger serves to identify ‘interesting’ physics data and discard nonproductive results. Triggering occurs at
multiple levels. The first level of the trigger system checks all data with a gross filter. Second and third levels further retest and process at different levels of analysis.

Due to the importance of the data, the cost of running the facility, and the difficulty of experiment setup, faults in the trigger system could cost a great deal of time and money. Any loss of data due to a failure could cause big losses in experiment data quality. Due to these constraints, fault tolerance and adaptation is required of the system. Moreover, due to the size of the system, redundancy is not economically feasible. Hence the major focus of the RTES project is on developing fault-mitigation strategies and implementing those strategies such that the system is robust from the point of view of faults.

As discussed earlier, a complex system should be developed using a domain specific modeling environment. The trigger system is being developed in the domain specific environment called Generic Modeling Environment (GME). Besides designing the system and planning and developing fault mitigation strategies, GME models of the system can also be simulated using the interpreter feature of the system. This simulation helps in testing the fault mitigation strategies as well as testing the system design itself.

The Problem Statement

The aim of this research is to develop a methodology to automatically generate simulations of the system from the high-level, domain-specific design models. (This design environment is also used to generate the physical realization of the system.) The underlying simulation engine software chosen for this purpose is Matlab® Simulink® and Stateflow®. A Matlab component library of the basic building blocks of the system, including hardware, kernel, application, and fault mitigation engine, has been developed. An interpreter is being developed that traverses through the GME model hierarchy and assembles a system simulation using relevant blocks from the component library in Matlab. A detailed discussion of the generation and simulation environment is given in the following sections of this report.

Overview of the Report

Chapter II discusses the literature survey on simulation and various simulation tools available in the market. It discusses in brief, why Matlab has been chosen as the simulation software. Chapter III discusses the BTeV project in detail, including the requirement of trigger
application and its layout. Chapter III also discusses the high level system design tool used for the project. Chapter IV presents a detailed description of the simulation models of each component. Chapter V discusses the benefits of automatically generating the system simulation from the high level design tools, and explains how it is done. The Stateflow data structure is discussed in order to explain how Matlab supports this auto-generation along with a discussion on the development of the interpreter.

Chapter VI presents test case discussions. Chapter VII provides the conclusion and a brief discussion of the proposed future work for the project.
SIMULATIONS: A DISCUSSION

As mentioned briefly in the introductory chapter, simulation is an important tool for making design decisions while developing any system. With an increase in the complexity of the system, the requirement of a system simulator becomes more prominent. A simulation is a simpler, executable representation of the system, developed to analyze and/or evaluate a particular aspect of the system [10]. In developing a system, it is required to find the faults early on in the system to avoid loss of valuable time and resources. Hence, simulations come in at a very early stage in designing the process. At this stage, the role of simulations is more towards understanding the execution of a system. In case of simpler systems, this is not as important. As the complexity of a system increases, there are a lot of interdependencies between various components of the system. Simulation can play a vital role in understand the system and how it may operate based on the current design.

Benefits of Simulation

With an increase in complexity of the system, system simulation becomes mandatory at each stage of design. Simulation is an important tool in the decision making process, a high level simulation of various design options can be used in finding out best designs that will maximize one or more performance measures [2] [3]. It is also useful in understanding how an existing system operates or how a proposed system might operate. A computer simulation can analyze models of arbitrary complexity. Worst case scenarios with simultaneous occurrence of independent functions can be studied, which could be difficult to achieve in the actual system. Simulation also enables an extensive observation of internal signals in a system, which is not possible in most of the actual systems due to the limitations on the I/O pins. External signals can be probed more easily as well. Also, due to more controllability and observability, it can be easier and faster to find errors in the simulations than actually debugging the system. In addition, simulation design can be changed more easily and quickly than the actual system, which result in a shorter debugging cycle and fewer prototype hardware design changes, a great benefit to the system designer. All these benefits, though various, are all interrelated. But it should be noted
that designing and using a simulation itself takes some effort. The purpose of integrating the simulation with the design tool aims to minimize this effort.

**What to Simulate?**

While simulation of a system can help better understand the system, not all systems are fit to be simulated, both from the point of view of resources and time. It has been observed that it is better to simulate systems with certain properties that make the effort of simulating worth it, than the others [2] [4]. Such systems have the following characteristics:

- These systems are *dynamic*: These are the systems with behavior that varies with time. These variations in the system behavior can be based on well understood relationships with the environment or could be totally random.
- These systems are *interactive*: Such systems consist of a number of inter-dependent components, whose interactions define the distinct behavior of the system.
- These systems are *complex*: There are a lot of interactions within the system, which consist of components whose individual dynamics need careful consideration and analysis in order to understand the whole system.

It is for these reasons that understanding the system without any aid becomes difficult. In the case of a simpler system, it would be easy to evaluate how the system would react in a given condition. But when the system is complex and dynamic, with significant internal interactions, considering all the factors and predicting the system’s behavior in a given condition becomes difficult. Here, simulation comes into picture.

**Steps Involved in a Simulation**

The process of simulation can be divided into the following steps [1] [3](These steps are similar to those involved in designing a system itself):

- Problem Definition: A clear problem definition is the first step in any design. The modeler should understand clearly, what the aim of the simulation is and what the system involved is.
- System Analysis: once it is clear to the modeler, what is expected out of the simulation, a thorough system inspection and understanding is required. The modeler should understand what the system inputs and random factors are at this point of time.
• Analysis of Input Distribution: Each random variable in the system should be examined and its distribution pattern should be studied.

• Model Building: Once the system components are understood, a model of the system should be created. The model is a simple representation of the system, consisting of all the required details.

• Designing and Coding the Simulation: Once the system model has been specified, the simulation can be designed and coded. This stage would depend on the simulation software in use.

• Verification of the Simulation Program: This can be seen as a “de-bugging” process of the simulation. At this stage, the modeler examines the simulation design to make sure that it best represents the system.

• Output Data Analysis Design: At this stage, it is required to decide what data will be collected from the simulation and how this data will be analyzed to evaluate the performance measures of the system.

• Validation of the Model: The simulation models are checked to make sure that the simulation behavior represents the system behavior as desired. This is done by comparing the output data from the simulation execution to the output data of the system under similar situations.

• Experiment Design, Product Runs and Statistical Analysis: At this stage, the decision of what simulation will run is made. These are the simulations that will be executed for system evaluation. Once the experiment is designed, the product runs are made and simulation data is collected for evaluation. The collected data is then analyzed to make the design decision or other system analysis.

• Implementation: This is the stage where changes are made to the actual system based on the simulation analysis.

**Types of Simulation**

Simulation can be categorized in various ways based on what the system to be simulated is, how it is simulated, etc. One of the important categories of simulation is based on what is being simulated [1] [2]. If the aim of the system is to evaluate the system performance based on
certain parameters, it is called a *Performance Simulation*. If the simulation focuses on what a system does, or in other words, the behavior of the system, it is a *Functional Simulation*.

Simulations can also be categorized as *Discrete Event Simulation* or *Continuous Simulation*. A discrete event simulation is the simulation where behavior of the system is based on the behavior of the components of the system, usually called entities and their interactions. Each entity’s behavior is modeled as a sequence of events. On the other hand, a continuous event simulation looks at the behavior of aggregate variables rather than on individual components. Also, at place of looking at the system’s behavior at discrete events, the continuous simulation is interested in looking at smooth changes in the system’s behavior in continuous time, for example, the changing weather patterns of an area.

The simulations done for this project are functional in nature. These simulations are behavioral simulations of the system, used to better understand the system and see how it behaves in given situations. All the simulations done for the project are discrete in nature, with each component being modeled independently. Hence these are Discrete Event Simulations.

---

**Simulation Software**

A lot of simulation software packages are available in the market. These are both application specific as well as general simulation software. For the purpose of research we required a simulation tool that would have a lot of functionality built in so that we would not have to spend time developing the simulation from scratch. It also would provide us with the flexibility to develop new components that could be integrated in simulations as seamlessly as the built-in components can be. Consequently, Matlab Simulink and Stateflow were chosen as the simulation platform.

**Matlab Simulink and Stateflow**

Matlab is an integrated, interactive environment used for mathematical computations, modeling as well as simulation of complex logical (StateChart representation) as well as mathematical systems (e.g. mathematical representation of a control system) [11]. Matlab Simulink and Stateflow are Simulation tools developed by The MathWorks, Inc.
Simulink [12] [13] is a GUI based interactive environment used for modeling, simulating and analyzing a wide variety of dynamic systems. The system is modeled in terms of block diagrams. Matlab provides a wide range of block models through its component library that are sufficient for modeling most of the components. Along this it also provides method to extend the library by letting the user define blocks specific to his/her needs.

Matlab Stateflow [14] [15] is a graphical design and development tool that can be used to visually model, simulate and analyze complex reactive systems based on finite state machine theory. It provides an easy tool to simulate, modify design and evaluate a system’s behavior. Stateflow is based on Statechart notation [16] [17] [18]. It extends the traditional Statecharts by providing graphical functions, temporal logic and integration with Simulink. This integration enables simulation of very complex systems. This combination of Simulink and Stateflow, along with visual representation of complex behavioral simulations and the ability to extend the component library were the main factors that lead to the selection of Matlab as the simulation tool for this project.
CHAPTER III

THE SYSTEM, FAULTS AND FAULT MITIGATION

The BTeV Experiment and Trigger Algorithms

At Fermi National Laboratory, high energy physics experiments are conducted to understand the basic composition of matter [19] [8] [9]. These experiments require a huge computational structure. The experiment takes place in a one mile diameter ring, where a series of particle accelerators apply enough energy to protons and anti-protons to achieve relativistic speeds. The particles and anti-particles moving in opposite directions collide, breaking up into the most basic components of matter.

Figure 2: The BTeV Experimental Setup
The experiment is designed such that the particle collisions occur every 132 nanoseconds. The collision results are detected by the use of pixel detector planes, placed at fixed distances, providing a three dimensional data set. The raw data rate is more than 14.8 Gbytes/sec. 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 are of no use to the experimenters. These can be discarded without any loss. Also, it should be noted that the rate of generation of raw data is very high; storing all this data would require huge storage space, which can clearly be avoided if the raw data is analyzed for its relevance before it is saved. Hence data-dependent save/discard decision algorithms are used and only the useful data is stored for the next processing stage. These algorithms are called Trigger Algorithms. This trigger algorithm and Data acquisition process together is the system of interest for this research.

As the occurrence of desired data is so infrequent, it is necessary that the system should be very reliable. To ensure that the system is always available, the systems should be designed such that it is fault adaptive.

**Faults, Fault Mitigation Requirements**

Because the trigger hardware must be highly available, the system must be fault adaptive and fault tolerant. Hence the system should be designed such that the faults occurring in the system are corrected semi-autonomously, with the least human intervention possible and in the shortest time possible.

One of the common ways of gaining fault tolerance in a system is through redundancy. The basic trigger system is very large and expensive, and budgets are limited. A triple redundancy solution will cause the cost of the system to go up by more than 3 times. Due to budget constraints, this triple redundancy is not possible. Instead the system itself has to be designed with a smaller amount of redundancy (e.g. 10%), with the ability to automatically adjust itself to the faults, and mitigate those faults based on certain predefined algorithms and the minimal spare capacity.

In order to create such a system, or make an existing system fault-adaptive, the first requirement is to understand how a given fault propagates through the system. Understanding the effect of a given fault on the system using functional simulation models can help plan more effective fault mitigation strategies. Moreover, the actual hardware for the system will not be
available for a few years. Architecture, algorithm, and design tool decisions, however, must be made before the final hardware construction.

Experimental Data Acquisition and Analysis

As mentioned before, BTeV experiments will produce huge amount of data, of the order of 1.5 terabytes per second. This high rate is contributed to by the exceptionally high rate of collisions, the large number of particles that are produced from these collisions and the large number of sensor channels in the pixel detectors that collect the data. Recording all this data is impossible; hence the BTeV trigger system is being designed such that it analyzes data in real time to decide what data is to be kept for subsequent offline analysis. The design of this trigger system is discussed in brief here.

In the trigger system, an FPGA-based preprocessing network will interface with the sensors and prepare the data for the DSP’s, performing low-level algorithms. The FPGA-based algorithms are expected to be relatively fixed and predictable.

The trigger system will have three levels of data analysis, Level 1 (L1), Level 2 (L2) and Level 3 (L3). The primary computational engine for the L1 trigger algorithm will consist of ~2500 Digital Signal Processors (DSP’s). The Texas Instruments TMS320C6x DSPs is currently being planned to be used for executing the trigger algorithms. Figure 3 represents the trigger system’s layout. In the L1 trigger, these 2500 processors will be connected in an application-specific network, with a hardware-based interconnection fabric [19]. Each processor will have its own local memory, with no shared memory. Special-purpose management processors will be distributed throughout the network to assist with management/diagnostics tasks. These processors must accomplish their tasks within a strict real-time deadline. Likewise, they must respond to faults and rapidly mitigate the effects within time-bounds. The L1 trigger is being designed to reject 99% of the interactions that occur in the Collider.
The remaining 1% of the collision data is passed to the L2/L3 triggers which are a set of ~2500 Linux-based computers organized in a loosely coupled network (i.e. Ethernet and Myrinet). The L2/3 reduces the relevant data by a factor of 20 by discarding further the unimportant data. Since the first level of filtering is performed on the DSP’s, the hard real-time requirements for the Linux machines are relaxed.

Such a vast system requires many interactions between its components. The size of the system, as well as the huge number of these interactions increase the complexity of the system, thus making it a challenging task for the system designer to design the system including each and every detail.
GME in the Development Cycle

As the system complexity increases, a strong requirement arises for a design tool that will keep the development process manageable. This tool must have certain properties to be able to reduce the complexities and the rising costs of developing such a complex system. These properties are as follows:

- The tool should allow the designer to specify fault mitigation behavior in a domain-specific manner.
- The tool should be able to integrate application specifications and design as the fault mitigation behavior is closely related to the applications.
- The tool should be able to design the target hardware architecture along with the available hardware resources and redundancies.
- The tool should be able to support design analysis and testing through simulations etc.
- The tool should be able to support direct synthesis of software and other system artifacts such as boot strapping of a network of DSPs etc.

GME is being used for the system design tool. A brief overview on GME has been given in the introductory chapter. GME provides a platform to design both the hardware and software composition of the system. GME uses the Model-Integrated Computing techniques. The models developed using MIC techniques are generally multi-aspect models capturing relevant information of the system. The models have proper representation of dependencies and constraints included in the aspects. Also, any number of translators can be developed for the system for different kind of analysis or data capturing, these translators are called interpreters in GME.

GME is being used as the ‘happy medium’ between the ultimate flexibility of designing ‘raw code’ and a fixed application. The paradigm being developed for the BTeV project allows a multi-aspect modeling of the system (The BTeV paradigm was developed in conjunction with others in the VU RTES group. It is used as a basis for system simulation and system generation. Its details are presented here as it is the starting point for system simulation.). As shown in Figure 6. The various aspects of the modeling paradigm are:

1. Application data flow: Define the flow of data and the processing of that data. A graph-based representation uses nodes to capture the processing steps (algorithms) with lines showing the data flow between nodes. These models can represent synchronous or
asynchronous behavior, and a variety of scheduling policies. For BTeV project, these are primarily asynchronous operations, with data triggered scheduling.

Figure 4: Software aspect of GME Models

2. Hardware Aspect: Define the physical structure of the target computer/network. Block diagrams capture the processing nodes (e.g. CPUs, DSPs, FPGAs). Connections capture the networks and busses over which data can flow.
3. Failure mitigation strategies: As hardware resources fail, or software crashes render a node non-functional, mitigation action must be rapidly executed to resume system function. The strategy to fix an error is often application and resource-specific. The user must be able to define these behaviors to reallocate resources, modify algorithms, etc, to bring back some level of functionality.
These models contain the design information of the system. From these models, using the interpreters, the software for the system, as well as the simulation can be generated. Among the artifacts generated are:

- Schedules and process tables for each of the processors, defining what executes and when it executes across the hardware.
- Network Routing, defining the paths from one process to another. Streams implement these routes, both internal to a processor and across the network. If processes are not on adjacent processors, a multi-hop path must be created.
- Local and regional Fault Managers: The code to implement the mitigation strategies must be synthesized from the behavioral models. Critical factors in the code are adherence to real-time requirements, maximizing resilience, etc.
- Simulations: The models are translated to Matlab codes that can be executed to build the simulation models of the system. This part is discussed in detail in the following chapters.
Where and How Simulation comes into Picture?

The primary purpose of the simulation is to predict the behavior of the DSP network, both in normal operating conditions and during faults. A detailed understanding of the behavior of individual elements is required to assist in the design and refinement of the architecture as we approach final designs. At the same time, understanding of the aggregate behavior of the full system is required.

Rather than creating each simulation by hand, auto-generation of simulations not only makes the simulation process easy and accessible to anyone who doesn’t have any prior experience with simulation, but also ensures an accurate simulation of the system, as the information for the simulation design is gathered from the GME models which also generate the system’s configuration files.

With this background in the system, the design tools and simulation, we can now look at the simulation design itself. The following chapters discuss the simulation process in detail.
With the understanding of what is desired out of the system, and what the system design tools are, we can now discuss the building blocks of the simulation. These building blocks are categorized based on the simulation requirements of granularity [20]. Hence a communication protocol consists not only of the decision making and handshaking component but also the FIFO buffer which actually implements the decision. This chapter presents a detailed discussion on each of these components, their behavior and the simulation model that captures the behavior.

Component Model Library

Matlab provides a Simulink component library with a number of basic mathematical and other simulation blocks. These blocks either implement some mathematic or logical operation or store or display data or information. A component library has been created in Matlab as part of the research to extend these libraries with functionality specific to the project. These blocks can be used in various combinations to create a desired simulation. The behavior of each component in the system can be divided in two categories. The first category consists of the generic behavior that all the components of same kind show, for example, all processes, independent of what software application they run, first checks all the input and output data streams before executing the data. The other category consists of specific behavior, which depends on the system design, for example the number of processes the scheduler is assigned to, is decided at the design time. Most of the components in our system primarily have generic behavior and require very little structural information to represent the actual system’ behavior. Using this information, we have created a library of such components in Simulink that only require some application specific information. These components are the “building blocks” used in constructing a full system simulation model. The following paragraphs discuss how the behavior of the components is represented in Stateflow models.
Discussing Simulation Components in Detail

The goal of the simulation is to predict the timing and behavior of the DSP network at multiple levels. In order to facilitate this we have created simulation models of various elements of the target system including components of the hardware and software layers of the DSP processors [21]. These include:

- The low-level communication port protocol (implemented in the hardware in the target DSP), including handshaking, bus access, byte multiplexing.
- Transfer of information to and from communication ports using DMA to software entities known as “streams” in the target RTOS.
- Behavior of the scheduler in the target RTOS for scheduling of software processes.
- Behavior of the software processes. These processes implement the Trigger algorithms and diagnostics facility.
- Behavior of the fault detection and messaging software elements.
- Behavior of the software processes responsible for fault mitigation.

Each of the components, that are part of the simulation are mentioned in detail in the following text.

Simulating the Hardware

The main component in the hardware whose behavior needs to be captured in order to simulate the system is the communication link between two processors. In the case of a single processor that doesn’t interact with another processor, the communication protocol is not required. The composition of this behavior is shown in Figure 7. The basic elements, processes, streams, and the scheduler will be described later. When inter-processor communication occurs, the simulation becomes more complex as the modeling of the communication is needed, as can be seen in Figure 8.
The communication protocol for these links is implemented in the hardware of the DSPs. As shown in Figure 9, the communication protocol consists of various components.
The actual physical links between two processors are wires that transmit data between the two processors. These wires transmit both data and hand-shaking signals between the processors. The first four wires are used for the handshaking and the rest are used for data exchange. These wires are bidirectional, the decision of which processor sends the data is made on the basis of handshaking. The handshaking takes place through a token exchange. The processor with token acts as the sender and the connecting processor acts as the receiver. The Port Arbitration Unit (PAU) of a processor decided the state of the token based on the states of connecting FIFO buffers and Communication Port Control Registers (CPCRs).

Each TI C6x DSP processor module that we use has four communication ports [22]. At the time of system restart, the comm. ports numbered 1 and 2 start as the sending ports (that is the ones with the token) and the ports numbered 4 and 5 start as the receiving ports. This particular port numbering scheme of 1, 2, 4, and 5 is due to historical reasons. The processing modules and the communication architecture employed are derived from TI C4x processors which used to have six communication ports numbered 0 thru 5. In the current set of processing
modules port numbered 0 and 3 are disabled owing to I/O limitations, and only the other four ports are actually available.

The FIFO buffer is used to store both the input data and the output data. The states of the FIFO play a role in deciding the token exchange. If the input FIFO is full, no more data can be received until the data is consumed by the processor. In this case, the processor doesn’t give token to the connected processor. In case the output FIFO is empty; the processor doesn’t request the token if it is with the other processor or passes it to the other processor if it is requested by the other processor. This behavior is captured in the Stateflow part of the model.

These FIFOs are connected to the DMA interface for data transfer inside the processor. For the purpose of this thesis, we will call the DMA interface a channel. Each communication port has one independent channel interface.
The kernel implements a simple asynchronous communication model. The data transfer takes place through two main components, the streams and the channels. The channel as mentioned before is the layer between the hardware and software. It abstracts the low level hardware communication structures. We will first discuss the channel structure followed by its simulation model.

The Channel

The data transfer between the physical links depends on the kind of hardware. Additionally, it also depends on the nature of the communicating device. The channel interface is used in order to provide a uniform interface to the kernel. This provides portability of the kernel as well as manages the heterogeneity of the target architecture. The current architecture is primarily composed of Texas Instruments’ C6x DSPs and Altera FPGA-TIMs. The channel interface is designed for TI C4x-Comm protocol.

Figure 11: Matlab diagram of the Communication Port Model
On the communication port end, the channel interfaces with the FIFO of the communication protocol, on the kernel side, the channel interfaces with data buffers called streams. A detailed discussion on streams comes later in this chapter, but for the purpose of understanding, streams can be seen as software FIFO buffers of fixed size. While each channel is associated with a unique communication port, multiple streams can be connected to a channel. The stream can be of type source, which will be sending data or of type destination, which will be receiving data.

The behavior of a channel can be categorized into two parts, the In-channel which receives data from the In-FIFO and the Out-channel that transmits data to the Out-FIFO. Figure 12 shows the data transfer within the processor.

The In-channel, based on the message type, either transmits the data to the destination stream or passes a dequeue acknowledgement to a source stream. The message received from the FIFO comes in words, the first word that comes is the size of the total data packet; this information is part of the message header. This message header also contains the information about the type of message, which could be a data packet or an acknowledgement. The channel distinguishes between these two types of messages, and after assembling it in its proper structure
based on the size, the channel transmits it to a destination stream. The stream number to which the message has to be transferred is also coded in the message header. This behavior is captured in a Stateflow simulation model as shown in Figure 13.

The out-channel reads messages from the streams and writes it to the output FIFO. Multiple streams can be connected to a channel. The source streams pass messages to be read by the processes on the connecting processors, while destination streams pass dequeue acknowledgements, to be passed to source streams on connecting processors. The channel maintains a list of stream data pointers. At a given time, only one pointer per stream can be listed, even if the stream has more than one slot filled. When the channel’s list has a stream pointer, it checks the status of the out FIFO, if there is an empty slot available, it writes message from the stream, one word in each slot until the whole message is enqueued to the FIFO. This behavior is captured in Stateflow as shown in Figure 14.
The Stream

A stream is a unidirectional data pipe, through which the source process sends data to the destination process. A stream is associated with exactly one source process’ port and one destination process’ port. In the kernel, the process ports hold pointers to the streams that are connected to it. A stream can transfer data from processes which are on the same processors, or between those on different processors. The behavior of the stream depends upon the position of its source and destination processes.

When both processes run on the same processor, the stream behavior is simpler. For the purpose of this thesis, let this stream be known as the local stream. The local streams implement a simple FIFO buffer. If there is a slot available in the stream, then at the time of source process execution, the process enqueues the data to the stream. In the same way, at the time of destination process execution, if there is data available in the destination stream, the process dequeues that data. Figure 15 shows the Stateflow diagram of the simulation of the local stream.
The interprocessor streams are associated with a source and a destination processes that execute on different processors. The process of data transfer differs between the streams based on whether the stream reads from a channel or writes to it. Let the interprocessor stream that writes to the channel be called the source interprocessor stream and the stream that reads data from the channel the destination interprocessor stream.

The data is enqueued to the interprocessor source stream in the same way as it is enqueued to a local stream, but the dequeue process differs. As mentioned in the discussion on channels, the source stream pointer is enqueued in the channel list if the source stream has data available to be sent, and there is no previous pointer in the channel list. The channel list is a kind of FIFO, hence when the channel list reads the stream pointer; the data is read from the stream, and passed to the output FIFO, word by word. Once the stream pointer is written to the channel list, the deq_ack_count (dequeue acknowledgement counter which keeps an account of how much data has been sent, and how many data receipt acknowledgements have been received which originated from the destination stream on the other processor) is incremented. When this stream receives the deq_ack originating from the destination stream on the other processor, it
decrements the deq_ack_count. When the deq_ack_count is equal to the total number of buffer slots in the stream, no more data can be read from the source process. This is to ensure that the no data is overwritten until it is read by the destination process. The behavior of the source interprocessor stream captured by the Stateflow is shown in Figure 16.

In the destination interprocessor stream, the data dequeue to destination process is similar to that of the local streams, but the data enqueue from channel is different. If there is a slot available in the stream, and if the channel receives data for the stream, then the data is enqueued to the stream. When the data is dequeued to the destination process, a deq_ack is generated, and queued to the out channel in the same way as any other data. This is the acknowledgement that decrements the deq_ack_count of the source stream on the other processor from where the data is originally sent. The behavior of destination stream, as captured in Stateflow is shown in Figure 17.
The scheduling of processes in a processor is non-preemptive. The scheduler that is implemented is a simple round-robin scheduler. In this scheduling policy, each process is given a chance to execute at its specific order is the list. The process makes the decision of execution based on the process definition and the availability of input data.

This behavior is captured in Stateflow by using variables modified by scheduler that signal the process to execute, and variables controlled by process that signal the scheduler if the process is running or not. This behavior captured in Stateflow can be seen in Figure 18.
A process is the software task that is executed on the processor. Each process can have multiple input and output ports connected to streams. These processes run the trigger applications, regional and other managers as well as other fault mitigation applications. A process is executed only when it receives the signal from the scheduler. Once the scheduler signals the process to execute, the process checks all its input ports for data. If any of the input streams do not have data available to be dequeued, the process is not executed and the control is returned to the scheduler which passes it to the next process. But if all the required inputs are available, it next checks for empty slots’ availability in the output stream, again, incase any of the slots required are unavailable, it sends the control back to the scheduler, but if all the slots are available, it reads data from all the input streams. Next it does the assigned computations and then enqueues the results to the output streams. This behavior of the process is shown in Figure 19.

Figure 18: Stateflow Model of Scheduler
The components discussed in this chapter together comprise of the library components that have been developed for the project. These models have been developed at the level of detail based on the simulation requirements. With the understanding of the system, the design tools and simulation components, we can now proceed to looking at how the simulation is generated from the system design models.
BTeV is a large scale system. Designing and simulating such a large system by hand is not only difficult but also error prone. As mentioned before, GME is being used for system’s high level design. An interpreter generates the system configuration files for the actual hardware based on the system design in GME [23]. Another interpreter is used to generate system simulation in Matlab Simulink and Stateflow. This chapter first discusses why auto-generation of simulation from GME models is useful; next it presents an overview on Simulink and Stateflow API's that facilitate the auto-generation, and finally it describes how the interpreter is designed.

Why Automatic Generation

The system is being designed in GME. The system configuration files are generated by these models that are developed in GME. Hence auto-generation of simulation models from the same source facilitates consistent representation of the actual system. When the system design changes one simply needs to change the GME design models, and quite literally, a press of a button regenerates the new simulation models. In absence of this, manually modifying the system simulation model could be quite cumbersome and error prone given the size of the system. Keeping these factors in mind, the generation of simulation models has been automated using both the Matlab provided API and application specific functions to generate these models from a Matlab script file. The Matlab script file is in turn generated from the GME models.

Hence first from the GME models, the Matlab script file is generated. On compiling and executing this script file in Matlab execution environment, the Simulink-Stateflow models are generated. This generation process is shown in Figure 20.
Here, it becomes necessary to explain why the interpretation of GME models generates script files instead of directly generating Simulink files. Matlab has provided an API to generate models in Stateflow and Simulink which is very concise and simple. Direct generation of Simulink files would require generation of a large, proprietary format, unnecessarily cluttered file. Each component generation would require generation of a long list of definitions that need not be an important feature of the component.

In order to understand the generation of script files, the understanding of how the Stateflow and Simulink APIs work is required. A short overview of this API is given in following section.

**Overview of Matlab API**

The interpreter designed for the modeling environment, as mentioned above, generates a Matlab script file. The script file requires some basic definitions and commands before actually adding blocks to the models. Some of these basic commands are used for:

- Opening a new system (The model file)
- Loading Simulink to the Matlab environment
• Checking if Stateflow environment has been loaded, if not loading it.

Addition of a new Block in Simulink is done through the \texttt{add\_block} command or function. This takes in arguments such as the library block to be copied, the location in the model where it will be copied, the location vector in the format \texttt{[left, top, bottom, right]} coordinates of the block and based on the library block, and some other arguments.

Blocks can be connected using the \texttt{add\_line} command. This command takes in arguments for the parent block where the line will show up, the source block along with its port number from where the line originates and the destination block along with its port number where the line ends, an optional argument of \texttt{auto-routing} of the line can also be passed to the function. These two commands are the key \texttt{API} functions used for constructing simulation models.

Stateflow uses a different \texttt{API} than Simulink [24]. Before discussing the Stateflow \texttt{API}, let us first discuss the Stateflow data structure. The Stateflow follows a hierarchical pattern as shown in Figure 21.

Stateflow Root is the parent of all Stateflow \texttt{API} objects. It is automatically created when the Stateflow \texttt{API} function \texttt{sfnew} is called. In our case, it is called when Stateflow is loaded to Matlab. The machines in Stateflow are similar to models in Simulink. We can get a handle for a machine by using the command:

\[
\text{Machine} = \text{root.find(’-isa’, ’Stateflow.Machine’)}
\]

This returns a list of currently open machines in the workspace. The problem that was mentioned above was that by loading Stateflow to the system, and then opening a new model, we have two Stateflow machines at place of just one that is provided by \texttt{sfnew}. This is so because now even the library models are loaded. The solution to this problem is in checking each machine’s name, and comparing it with the name of the project file. This will return the model’s machine-index in the list.
Each Stateflow block in the machine is a chart. A chart contains Data, Events, Transitions, Junctions, Note and States. For the purpose of project we would be using all these except the Note component. A state can further contain the above mentioned components as well as itself.

The Stateflow provides an API of functions that can be used to directly access these models, as well as create them. The API provides a method to create a new file, using the `sfnew` function; this adds a Stateflow block to the system. For the project, we could be dealing with more than one Stateflow machines, or more than one Stateflow machines could be open in the Matlab workspace at a given point of time. Hence, to work on the correct machine, so that the code doesn’t modify any other machines, and works properly, the code first scans though the list of currently opened machines, and finds the one associated with the project.

The interpreter, after loading the Stateflow, will do the following necessary steps in order to get the correct handle of the machine, before adding information to the models:

- Create a Stateflow block
- Get the root handle
- Get the list of machines
Get the index in the above list corresponding to the current machine
Get the machine handle

A similar method will be used each time a new Stateflow object is created, to find out that object’s handle in the list of all the similar objects. With this background on what we want to generate using the interpreter we can now look into the design of the interpreter itself.

Mapping of Models to Simulation Components

In order to generate system files, that configure and run the actual system, an interpreter is used, that translates the models developed in GME to system code. The interpreter written as part of the research uses the data structure defined in the above mentioned interpreter to maintain consistency. A short overview of these data structures is given in the next section. Following this is a discussion on the structure of the interpreter, mentioning how the individual components in the models are mapped to Matlab script code and Simulink models.

Interpreter Structure

GME provides an extensible COM-based programming interface, referred to as the Builder Object Network (BON) to access the models. An interpreter has been developed using this interface. The interface has been extended using the DECLARE and IMPLEMENT macros (A detailed discussion of these macros is given in the GME manual as part of the High Level Component Interface for GME [23]). Specific data structure and classes have been used for all the hardware as well as software components. The hardware component classes that are directly useful for this project are defined in nwnode, nwport and nwstream header and cpp files, while the software component related information is stored in dfnode, dfport and dfstream header and cpp files.

Basic Classes in the interpreter

This section first presents a discussion on the data structures and classes, followed by a brief overview of the interpreter code.

The $CNWNode$ class holds the information of individual hardware components such as a processor. The class $CNWNode$ contains a number of functions that can be used to gather information such as the list of software processes mapped to the processor as well as functions
that take care of low level behavior of the processor. This class has been extended to incorporate
functions and information required to generate a correct Matlab script. The first function that has
been added is used to uniquely identify the processor with a name. Matlab requires a unique
name of each component at a given level of hierarchy. A function is used to access the name of
the processor. Two more functions have been added that return a list of software processes and a
list of data streams respectively that run on the given processor.

The class $CNWPort$ represents a single communication port, and stores information such
as the stream it is attached to or the list of streams that it has to multiplex data between.

The class $CNWStream$ represents the physical wire that connects port of one processor to
the port of the other processor. This class stores only the source and destination communication
ports.

As mentioned before, the classes with $df$ (data flow) prefixes represent the software
components of the system. Hence $CDFNode$ represents a software process. This class can store
information like the script that it would execute, the input and output ports (these are the
software ports that have been discusses in previous chapters and whose software implementation
is discussed below) that are associated with the process, the memory that will be allocated to the
process and so on. Similar to a processor, the process is also allocated a unique name to be used
by Matlab script to differentiate it from other processes at the same level of hierarchy. A function
is used to set this name based on some decision and a $get$ function is used to extract this Matlab
specific name. The components and behavior of a process varies depending on whether the
process is a source process, destination process or both. The Matlab script requires this
information to select a corresponding process behavior component from the component model
library. Hence a function is used that sets the $type$ variable and another that reads this value.

The $CDFPort$ represents the software ports, i.e., the ports associated with the processes.
This class contains information related to the port such as whether it is an input port or output
port and the data stream it is attached to.

The $CDFStream$ represents the data flow stream in a processor. It contains information
regarding the stream such as the source and destination nodes (process), source and destination
processors, ports to which it is attached and so on.
Logic flow in the Interpreter Code

The code for the interpreter starts with the initialization function that writes out the code to create a new Simulink project to a script file. After this, the interpreter writes to the script file, a set of commands to load the Simulink into the Matlab workspace. When Simulink is loaded to the workspace using this method, a machine for sflib is created by default along with the machine for the project. Hence, first we need to find the correct machine handler amongst the two that is associated with the project. This is the next set of commands written to the script file by the interpreter.

Besides loading up Stateflow to the workspace, the initialization function also generates a few Matlab script files that contain reusable code. These are the codes that would return the chart handler of a given chart from a given chart list at a given level of hierarchy and code that would add data to a given chart without special data definitions or with special data definitions.

Once the initialization is done, the interpreter next scans through the hardware node list for the hardware components, and for each component, it assigns a unique Matlab identifier name and coordinates for the hardware component. Once these are set for a component, it then writes a command to the script file that would add a sub-system block in the Simulink for the hardware component. This sub-system would contain all the software processes, communication protocol behavior and other components like streams and channels for data transfer.

As mentioned before, each processor has a scheduler, a round robin scheduler in this case, that decides what process is executed next. Once the sub-system is created for the processor, the next step is to set a scheduler block in the subsystem. The interpreter writes to the script file, an add_block function, which takes in as arguments, the name of the library scheduler, the sub-system name where the scheduler will be place, and the coordinates where the scheduler block will be placed in the sub-system. Various multiplexers and de-multiplexers are added to the block along with the scheduler to convert data to an array of data and vise versa. A delay block is also added to the scheduler, as the interaction between the processes and scheduler is that of a feedback, and without a delay in the data transfer, it will not be a correct representation of the actual system. Moreover, such feedback systems without delays in Matlab cause an algebraic loop, and can not be executed.

Once the scheduler is added to the system, the next step is to add the communication protocol block. The interpreter checks the comm.-port list for a given processor, checks the
comm.-port number, and writes a command to the script file to add a comm.-port to the processor sub-system based on the type.

Next the system checks the list of dfnodes, and for each node, which is actually a process, places the process block. Along with adding the correct process, it also checks if the process is a source, a destination or both, and in each case, places the other enqueueing or dequeueing components for the simulation. It also writes code to the script commands that would connect the appropriate ports on the process to the corresponding ports on the scheduler or the multiplexer and de-multiplexer related to the scheduler. Once the interpreter goes though all the nodes of the process list for a processor, it next adds code for streams.

In the processor, the streams are listed as dfstreams. The interpreter scans through this list and for each stream checks if it is a local stream, or interprocessor source or interprocessor destination stream. Next according to type of stream, it writes code to the script that places the correct stream block from the components library and places it in the system. The code makes sure that streams do not graphically superimpose each other, by placing each stream at a fix distance from the previous stream. Once the main components are placed in each of the processors, the interpreter next uses the nwstream list and connects the processors, this corresponds to the physical link through wire between the processors.

This is a brief overview of the interpreter. This interpreter generates the Matlab script files as mentioned before. The following chapter presents a test case, where a real DSP layout model in GME is simulated in Simulink using the interpreter.
CHAPTER VI

SAMPLE TEST CASE, RESULTS AND DISCUSSION

Until now we have discussed the benefits of simulating a large scale embedded system, the modeling in detail, and the process of generating simulations. This chapter presents two case studies which will better illustrate the concepts that have been discussed in this thesis.

Case Study 1: System Failure Simulation

The first case study that we discuss is a simple case of two processors running compute processes. The processes on one of the processors are source processes, producing data, and the processes on the other processor are sink processes, which take the data generated by the source processes as input. Before discussing the results and usefulness of using the simulation, let us first look at the structure of the model. The following paragraphs discuss the structure of the system, and components of the system that are specific to this system.

Figure 22: The Top view of Simulation Model
Being a multi-processor system, the channels and communication protocol play a vital role in the data transfer. Figure 22 shows the two processors’ connections. Looking inside the source processor, named processor type 1 in the simulation, there are two components, the kernel, and the communication protocol. The communication protocol is almost the same in both the processors, except that the type 1 processor starts as a sender and type two processor starts as the receiver in data transmission. The communication protocol’s model is shown in Figure 23.

![Figure 23: The Communication Protocol Model](image)

The other components of the processor type 1 are the processes, the source streams, the scheduler, and the channel. Figure 24 shows a view inside the processor model, the block in blue is the scheduler, whose model has been discussed before in this thesis. The blocks in green are the processes. These are the source processes, which are connected to source streams, and there execution depends only on the scheduler, as there are no input streams connected to these. The blocks in yellow are the source interprocessor streams, and the magenta block contains sub block that model the channel’s input and output behavior.
Figure 25 presents the Stateflow block of the source process. This behavior is same for all the source processes, the computation or task takes place in a separate block that is triggered by variables set in this block.

The behavior model of source interprocessor stream as well as the in and out components of the channel has been discussed in previous chapters.
Figure 26 shows the components of the processor type 2 model. The scheduler, destination interprocessor stream and channel models are the same as have been discussed.
before. The main difference is in the behavior of the process. The process behavior Stateflow model is shown in Figure 27.

Till now we have discussed a sample system, its design and layout. In order to demonstrate the usability of this system, in capturing the behavior of the system in cases of faults, let us consider a simple failure scenario in which there are no data slots available in the stream. In the nominal case, the source would not enqueue any data to the stream. But due to a failure in the system, the source erroneously reads that there is a slot available in the stream. It enqueues data to the stream, but the stream is already full, so a failure occurs.

The system that has been modeled is for the ideal case, where no such failure occurs. In order to study fault scenario, we have to change the system to incorporate the fault.

We can simulate this fault scenario using the random function generator available in the components’ library in Simulink. The fault scenario system is modeled such that if the stream is full, and the random event occurs, the can_enq value, that transmits the status of the stream to its
source, is changed. Now the source reads that there is a slot available in the stream, and enqueues data to the stream. Consequently, a fault occurs. The stream’s model using the random generator is shown in Figure 28. The block in green (small square on the left side), is the random fault generator, which sends as input, data to the interprocessor stream’s Stateflow block. The modified Stateflow behavior model is shown in Figure 29.

The number of slots filled in the stream can be calculated in the simulation, and can be output to a scope (available in the component’s library for Simulink) and can be analyzed offline, post-simulation. The simulation is designed such that if there is an enqueue from the process, and there are no slots available in the stream, it will go in the error state (shown as the error_state in Figure 29). This changes the value of the err variable, which is used to trace the stream’s functioning. Figure 30 shows the error plot for the stream. This figure plots the err variable over time. This is useful in studying when the error occurred. Figure 31 shows the number of filled slots in the stream. These two plots are examples of ways to set traces on the
system to understand it better, observe how the states change over a period of time, or to see when an error occurred and how it affect the component of interest of the system.

Figure 29: Behavioral model of stream with error
Figure 30: Error plot for the stream

Figure 31: Number of Filled Slots in the Stream
Case Study 2: Simulation Generation from GME Models

For the second case study, we will consider a GME model of a given system, and see how the simulation model is generated.

Figure 32 shows the diagram of hardware aspect of the system model in GME. This model contains information such as inter-processor connection. Figure 33 shows the software model of the system.

Figure 32: Hardware Aspect of GME model
This model contains information such as what processes are connected, the data transfer direction, and the processor mappings. From the point of view of simulations, these are the main aspects that capture the system information. On interpreting this model, a Matlab script file is generated. When the Matlab script file is executed, the simulation model is generated.

Figure 34 shows the generated system simulation. This is the topmost layer of the system, and shows the processors and the connection between these processors. Each processor has four ports, but not all are connected at a given time. This information is generated from the hardware topology modeled in the hardware aspect.
Figure 34: Simulation model, Top Layer

Figure 35: Processor Simulation
Looking inside one of the processor, as shown in Figure 35: Processor Simulation some of the components, like the scheduler and the comm. ports are automatically generated with the processor. While the scheduler has no corresponding model in GME, the ports are generated based on the ports associated with the processor. The rest of the simulation is generated on information generated by combing the various aspects.

The process blocks are generated based on the process list that is associated to a processor. This association is generated by the explicit mapping of software onto hardware in the GME models.

The data transfer between the processes is done through streams. The particular behavior of stream is determined by whether it is a source, destination or local stream. This is in turn decided based on the hardware mapping of the source and destination process. In case there is a data link between processes that are on different processor, and there is no direct physical link modeled between these processors, the system then routes the data through a third processor which is connected to both the processors. Hence there will be more processes running on each processor than are modeled in GME. These extra processes are the forwarding processes.

Hence case study 1 shows how a simulation model can be used to study a scenario, case study 2 shows how from a GME model, the system can generate a Matlab simulation in Simulink and Stateflow, using the Matlab API. The case study 2 gives a short overview of which components are interpreted to generate which simulation components. The following chapter concludes this thesis with a brief summary of the project, proposed future work and conclusion.
CHAPTER VII

CONCLUSIONS AND FUTURE WORK

Conclusion

Behavioral simulation is a useful tool to analyze and design embedded systems. The benefits of simulation can be seen at every stage of system design. The system for which the simulation tools have been designed is large scale and real time. It has very small tolerance for faults, and hence need to be designed to adapt to the inevitable system fault. This requires a good understanding of the possible failure scenarios and adaptation techniques. Powerful, easy-to-use tools are needed to simulate the system as-designed. A model-based simulation capability has been described, that uses the same models as are used for system synthesis, greatly reducing the effort to simulate and decreasing the potential for errors in simulation.

The simulations have been developed in Matlab Simulink and Stateflow. A library of building blocks for the simulation has been developed. This library consists of Simulink-Stateflow models of very low level components of the system. The simulation design is bottom up approach, so first each component is designed, and then using these building blocks, and the design information from the modeling framework (GME models), the simulation is generated.

In order to automatically generate simulations of any given hardware, an extension to the current work is required. The rest of the concluding chapter presents proposed future work that can help enhance the usefulness of the tool by extending it to simulate other kinds of embedded systems, as well as to develop a more effective way to generate fault scenarios in the system.

Future Work

The work that has been discussed in this project is focused towards developing the simulation for systems that would be created for BTeV project. This implies that the models are for specific hardware, kernel etc. Hence the first set of proposed future work focuses on the extension of the simulation to incorporate other components, to increase the simulation scope.

The first proposed extension to the research focuses on developing the building blocks of a more generic system, and takes a bottom up approach. The aim of this research would be to provide a framework for simulation design for the nominal case. The second proposed extension
to this research is to develop tools that would simulate fault scenarios. The following sections talk about these suggested extensions in detail.

Generalization

As mentioned before, the work done was with the aim of developing simulation tools in Matlab Simulink and Stateflow that would represent possible system designs for BTeV. The BTeV project will use a fixed set of hardware, software kernel and applications. Hence, for this phase of the project, we were only required to develop the simulation for a DSP with a standard behavior. There is, however, always a possibility that the BTeV system might shift to a different hardware architecture, operating system, etc. Moreover, being a useful tool to develop embedded systems, this tool should be made more generic, such that it can simulate other systems easily.

The bottom up approach used in developing this system is useful in doing this extension, by modifying individual components instead of remodeling the whole system. Following paragraphs look at each of the component categories and how a change would affect the modeling, starting with the lowest level.

**Hardware Level Changes**

For the current system, the hardware is modeled using two components, the sub-system block, which contains all the components that execute on the DSP, and the communication protocol block, that simulates, using the Simulink and Stateflow functionalities, the behavior of communication protocol of the actual hardware. A change in this hardware will correspond to a change in this communication protocol, a new block that would represent the behavior of the new communication protocol will have to be developed. Also, the interface which acts as the layer between the hardware and the software will have to be modified to adjust to the new protocol.

**Kernel Level Changes**

A kernel change could imply a different way of data transfer within and between processors. A kernel level change could be more involved that the other changes, and would require not only remodeling the behavior of each of the software components such as the data streams and execution flow of a process, but also there interconnection. Because of these changes, it is anticipated that the interpreter might have to be modified quite a bit.
Execution Level Changes

This would involve changes of the scheduler. The system is modeled right now in such a way that any number of processes can run in parallel, the scheduler sends a token to each process, directing it to execute or wait. Hence changing the execution method would only require changing the scheduler block. Very few changes in interpreter are anticipated for this case. For example, if the scheduler is changed from pre-emptive to priority based, the priority rule can be sent as input to the scheduler or can be calculated by the scheduler, and the processes it selects can be executed using the same execute or wait tokens.

Modeling Fault Scenarios

The scope of the current research is to develop the basic components to simulate a given system in its nominal state. Future work could be done to develop a simulation tool to generate fault scenarios using the GME models. This would help to understand the combined behavior of the system in these failure cases. The next step towards developing such a tool would be to modify the BTeV paradigm for GME models such that failure information can be represented and extracted for simulation purposes. Some modifications in the interpreter are also anticipated to incorporate the generation of these simulations.
REFERENCES


[12] [http://people.msoe.edu/~saadat/Matlab.htm](http://people.msoe.edu/~saadat/Matlab.htm)


[23] [http://www.isis.vanderbilt.edu/Projects/gme/GMEUMan.pdf](http://www.isis.vanderbilt.edu/Projects/gme/GMEUMan.pdf)
