Model Based Self Adaptive Behavior Language for Large Scale Real time Embedded Systems

Shweta Shetty, Sandeep Neema, Theodore Bapty
Institute for Software Integrated Systems, Vanderbilt University,
2015 Terrace Place, Nashville, TN 37235
{shweta.shetty, sandeep.neema, ted.bapty} @vanderbilt.edu
http://www.isis.vanderbilt.edu

Abstract

At Fermi lab, high energy physics experiments require very large number of real time computations. With thousands of processors (around ~1000 FPGA's , ~2500 embedded processors ,~2500 PC's and ~25,000,000 detector channels) involved in performing event filtering on a trigger farm, there is likely to be a large number of failures within the software and hardware systems. Historically, physicists have developed their own software and hardware for experiments such as BTeV [9] However, their time is best spent working on physics and not software development. The target users of this tool are the physicists. The tool should be user-friendly and the physicists should be able to introduce custom self-adaptive behaviors, since they can best define how the system should behave in fault conditions. The BTeV trigger system is being used as a model for researching tools for defining fault behavior and automatically generating the software. This paper presents a language to define the behaviors and an application scenario for the BTeV system and its expected fault scenarios. These self adaptive system tools are implemented using Model Integrated Computing. The Domain specific graphical language (DSL) is implemented within the Generic Modeling Environment (GME) tool, which is a meta-programmable modeling environment developed at Vanderbilt University.

1. Introduction

Software development for real time embedded systems can be difficult, as these systems are part of a physical environment whose complex dynamics and the timing requirements have to be adhered to. Embedded real time systems should produce not only the correct outputs, but should produce them at the right time. Furthermore, a reasonable "behavior" is expected from these systems even under the fault scenarios, when the hardware or the software components fail. Large-scale systems (i.e. 100's to 1000's of processors) represent further challenges. This work is motivated by a large scale real time embedded system under construction at Fermi Lab, where high energy physicists use massive facilities to delve into the basic composition of matter. We are investigating and developing a self-adaptive approach that relies on fault-mitigation to provide fault tolerance in this class of systems.

It has been argued that self-adaptive approaches and infrastructure help mitigate the complexity of tuning, and maintaining embedded systems [1][2]. Unfortunately, this benefit comes at the expense of increased complexity in 1) developing a self-adaptive application approach, and 2) infrastructure itself. Constructing a self-adaptive solution requires significant effort in designing and programming the adaptive behaviors at multiple levels of the software infrastructure, including: 1) the application, 2) the middleware, and 3) the operating system. Furthermore, the adaptive behaviors need to be coordinated, across the different levels of the software infrastructures, and across processor boundaries in an extremely large-scale distributed system. Clearly, a low-level programming based approach that hard-wires adaptive behaviors do not scale for this class of system. A system developed this way would be extremely brittle to support the evolving requirements. Automated tools supporting a higher-level of abstraction are required to assist the system developers in managing the complexity. Since the target users are the physicists themselves, the tool should be user-friendly and should offer higher-level abstractions that are easier to manipulate and maintain, and amenable to analysis. Finally, the tools must have the ability to synthesize low-level programming implementations from the higher-level abstractions, alleviating the need for a programmer to construct code, while ensuring consistency with the higher-level abstractions.

To address these issues we have applied Model-Integrated Computing (MIC) [4][5][6][7], a methodology for developing tool-driven embedded software solutions developed and refined over two decades of research at ISIS, Vanderbilt University. The key elements of an MIC-based approach are:

1. A **Domain-Specific Modeling Language (DSML),** the syntax and static semantics of which are precisely

defined using UML-based notations and OCL-based constraints in a meta-programming environment.

2. A **Meta-Programmable Generic Modeling Environment (GME)** [8] that instantiates the DSML, resulting in a **Domain-Specific Modeling Environment (DSME)** which is highly customized for the needs of the system developers in the target domain.
3. System developers build **Integrated Multiple-View Models** in the DSME capturing information relevant to the target system from several aspects. The information captured includes adaptive behaviors, information processing architecture, and physical architecture of the target system.
4. **Model translators**, which generate inputs to various analysis tools, as well as synthesize various low-level artifacts for instantiating/deploying the system.

Utilizing these key capabilities of MIC, we (in collaboration with other research groups) are developing tools for designing self-adaptive solution that provides fault-tolerance via mitigation for a class of large-scale real-time embedded systems. The rest of this paper is organized in the following manner: Section 2 provides a description of System Development Environment, Section 3 describes the System Synthesis from models. Section 4 demonstrates the Results and the Experiments conducted with a prototype of our tools.

2. System Development Environment

The application scenario which we at Vanderbilt are working on is the BTeV system at Fermilab. The BTeV experiment [9] includes a trigger with approximately 5,000 CPU's. These are time critical event filtering applications running on trigger farms with thousands of processors that are likely to suffer from a large number of failures within the software and hardware systems. There are three main entities of the whole application: 1) EPICS Control System (System Operator Interface), 2) Modeling tools, and 3) ARMOR (Reconfigurable fault-tolerance entities) [6] (explanation of ARMOR is out of scope of this paper).The *EPICS* system provides a means of injecting faults into the system and also helps the user to see the effects of those faults on the system. EPICS provides the operator with all the necessary system information. Some of the system control is also available to the operator. The *modeling* environment provides a graphical language where a system developer (in this case the Physicists) can design and specify the system. The domain-specific graphical abstractions as provided in the modeling environment and the analysis and system synthesis capabilities are particularly attractive to the physicists. The modeling tool allows for the specification of system from several different aspects. The significant aspects are:

- Application **Data Flow**: the component-based specification of information processing,
- Hardware **Resources**: the physical computer hardware used in system implementation, and
- **Failure Mitigation Strategies**: the specification of how the system should detect and react to component and system failures.

Next we provide a detailed overview of these individual aspects to illustrate the modeling environment.

2.1. Data Flow Specification Language

As shown in the Figure 1 below, this aspect lets a system developer define the key software components and the flow of data between them. A standard hierarchical dataflow notation is used, where nodes (boxes) capture the software components (algorithms) and lines show the flow of data between nodes. These models can represent synchronous or asynchronous behavior, and a variety of scheduling policies. For the BTeV trigger, these are primarily ***asynchronous*** operation, with data-triggered scheduling.

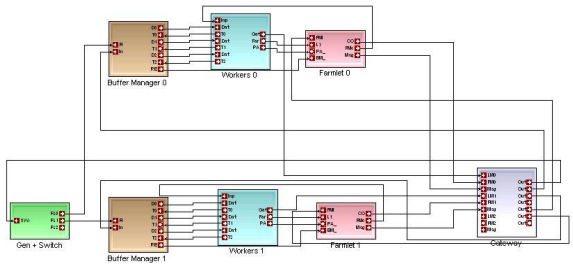

Figure 1. Data Flow Model

The primitive software components in the dataflow modeling are associated with a ***script*** that provides the implementation of the software component. Figure 2 shows a screen-shot of the dataflow modeling aspect with several processes and dataflow links between them. (The fault-manager processes are also depicted in the dataflow modeling view. These fault-manager processes are associated with fault-mitigation strategies described later.)

2.2. Resource Specification Language

This aspect defines the physical structure of the target architecture. Block diagrams capture the processing nodes (e.g. CPU-s, DSP-s, FPGA-s). Connections capture the networks and busses over which data can flow. One of the assumptions made here is that the hardware component is modeled exactly the same way as it is laid out physically.

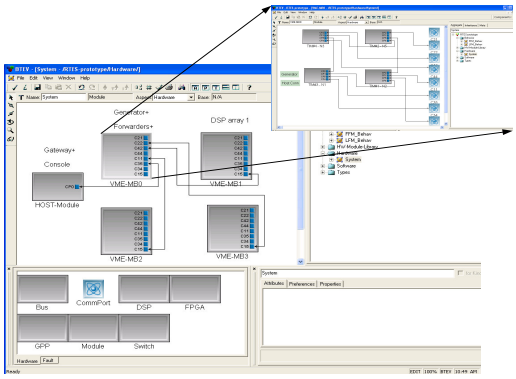

Figure 2. Hardware Component

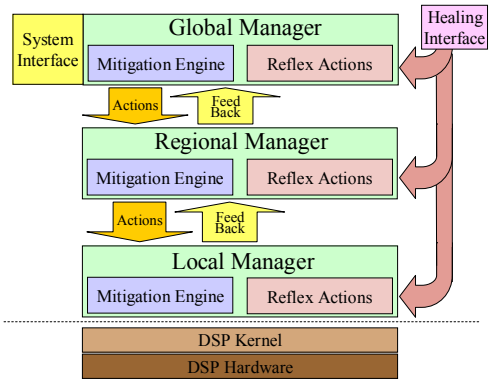

Figure 3. Hierarchical Fault Mitigation

2.3. Fault Mitigation Strategy

The motivation behind the fault mitigation strategy language is generally the desire for integration of prior knowledge into the fault detection system and to use a recursive narrowing of fault probabilities to aid in the identification of symptoms. The goals of the fault mitigation mechanism as indicated above are threefold:

1. Maintain the maximal application functionality for any set of component failures
2. Recover from failures as completely and rapidly as possible.
3. Minimize the system cost, resulting from excessive hardware redundancy.

These goals are contradictory in nature. Maintaining the maximum application functionality for a set of component failures requires redundant resources if performance is not to be compromised. However, increasing redundancy increases the cost of the system.

In our approach we have defined three different hierarchical levels of control: local, regional, and global fault-managers. Hierarchy is a simple, yet powerful concept that has been applied and proven in many types of complex and large-scale organization. The BTeV trigger system is similar in terms of complexity and scale. Clearly, in a system of this size, sending all fault-information to a centralized fault-manager for a mitigation decision is not a scalable approach. Reaction time would suffer in small systems, and be increasingly large as systems are scaled up. The *Local* managers are the leaf nodes, responsible for sensing faults and implementing actions. *Regional* managers handle successively larger regions of hardware. *Global* managers are the top-level fault mitigation agents, interfacing with external systems and/or users. These levels roughly correspond to the inherent hierarchical organization of the BTeV Trigger architecture.

Faults are handled at the lowest layer possible. At any specific level, if mitigation is not possible, due to resource availability, or lack of sufficient contextual information, the fault is promoted to the next level of fault manager.

2.3.2. Fault Mitigation Language

This aspect of the modeling environment defines the fault-mitigation strategy. A Statecharts-like [13] notation is provided that allows a developer to define various failure states. Conditions to enter or leave those states, along with actions to be performed when state transitions occur are also defined. The language can be summarized at a high level as follows:

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

The first step in the development of a new language is to specify the syntax and the visualization in the GME [2]

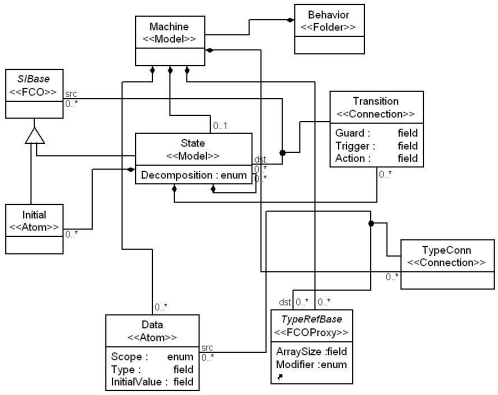

Figure 4. Meta-Model describing the Language

meta-modeling environment. The meta-model for the state machine language uses the UML [6] based notations describing the different associations and interactions between the components. Figure 4 shows the metamodel components. In metamodel we can see that *Behaviour* Component contains *Machine* which in turn contains the *State*. The *Machine* can be considered as *Fault Manager* whose behavior can be described using StateChart concepts. Based upon whether they receive or send data the *Ports* can be specialized as *Input* or *Output*. The Behavior state machines perform actions based on triggering conditions. The *Triggers* is defined as a Connection which has the *attributes* which specify the triggering condition and action to be performed. The action code which is the *fault handling* code is mainly written in C and based on the type of the guard/trigger the action would be either to send the message (action, error, statistical) upstream or downstream. These triggering conditions take the form of messages. Messages are utilized to propagate notification of failures up through the hierarchy. In order to minimize the bandwidth while providing the maximum flexibility, the messages are specified to have a variable length, based on the originator of the message. Considering the BTeV trigger architecture as an asynchronous message passing system, the messages defined for this prototype are typically:

- *Fault/Error Message*, reporting errors in hardware, or application.
- *Control messages*, both decision requests and commands that force parameters to change in the running system
- *Statistical messages* are periodic in nature.

The Fault Mitigation Strategy Language overall supports following features

1. Hierarchy - The user can specify Hierarchical State Machines using concurrent models which reduces the complexity significantly. Each state of a hierarchical state machine can contain a whole state machine and so on.
2. Asynchronous Message Passing - Considering the Asynchronous nature of the architecture the state machine generation can handle asynchronous nature of the messages coming in the states.
3. Reactive - Continuous interaction with environment; system responses depend on input, system state and time.

Currently, we have a fairly simple language for specifying the fault handling aspect. The Action code of the fault handling aspect is written in C which is not highly intuitive for a person who is not well versed with the Kernel functions. Since the focus of this tool is to make domain specific fault handling language, the action functions should focus more on the fault handling aspect of BTeV trigger system. We have few predefined fault handling functions which would be called from the Action component and whose implementation is available in the Kernel. Also, we plan to model the real time constraints accurately and make the behavior more temporal based by adding timers.

3 System Generation

The overall system is generated once the models and the fault behaviors have been defined by the user. Several low-level artifacts are generated from the models in order to derive an implementation. System synthesis performs the following key activities:

- Dataflow synthesis – This involves mapping a dataflow depiction in models into a set of software processes, and inter-process communication paths. This mapping derives the execution order or schedule of the processes executing on the processors. The communication paths between software processes must be setup such that the software process itself is unaware of the location of other software processes that it is communicating with. However, the mapping process alone cannot enforce location-transparent communication. It relies on some capabilities in the runtime execution infrastructure in order to facilitate this. The overview of the runtime system is described in [14].
- Fault managers synthesis – This involves synthesizing, code for fault-managers from the behavioral specifications provided in models. The user specifies the behavior using the tool with appropriate knowledge of the whole system.

3.1 Fault Manager Code Synthesis

The Figure 5 shows the mapping of self adaptive behavior language defined in the models into the code which is linked to the runtime.

The following steps define the mapping algorithm:

1. For each fault manager (software) component model in the dataflow models, an associated state machine model that defines its mitigation strategy is located. This state machine model defines the behavior to be synthesized for each fault manager.
2. Given the behavior, the set of defined states is collected. In the above example, there are 2 distinct states: NOMINAL_FM, and REROUTE_LINK. An **enum** construct is generated in the source code.
3. Based on the trigger interface variables in the state model, a function prototype is defined for the state transition step function, with a parameter for current state and each of the trigger input & output variables. This function is used to compute next states and to read & write input and output messages. In the example, there is the current state, "**cs"**, followed by three (3) inputs and four (4) outputs.
4. Next, the body of the behavioral state transition step function is defined. For each state, a **case** segment is defined. Here, there are two **case** entries, one per state.
5. Within the **case**, the code is generated to implement the guard conditions, in the form of **if** clauses. The example shows one guard condition, **if(det_fault)**.

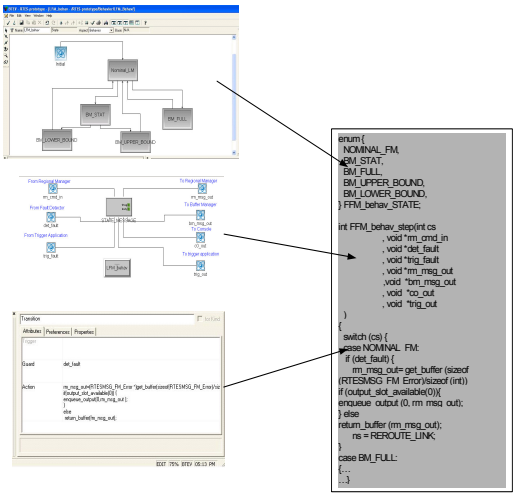

Figure 5. Mapping of the Model with Generated Code

6. Within each of the **if** transition steps, the **action** code is inserted. This is based on the action attributes that the user specified when creating the behavioral model. The action shows creation of a message (**rm_msg_out**), followed by a conditional transmission of the message to another behavioral process.
7. Steps 4-6 are repeated for each **state**, **guard,** and **action** specified in the model.
8. Steps 1-7 are repeated for each physical resource in the models.
9. If there are hierarchical models, then we have nested switch case statements generated.

This generated code is compiled and linked with the dataflow code generated to create a set of executable models for the system. Resource models are used to create configurations for the loader to install and initialize the system with the generated executables.

4 Results and Experiments

The tools for modeling, analyzing, and synthesizing large-scale, parallel, fault adaptive real-time systems have been developed and prototyped using the Model-integrated computing infrastructure at Vanderbilt University. These tools were used to model and synthesize a scaled down representative prototype of BTeV system. The prototype and some observations are described below. The prototype contained 16 embedded DSP processors and two PC workstation. These processors were configured in an application-specific topology to reflect the dataflow of the BTeV trigger algorithms. Figure 2 shows this example. There are approximately 20 concurrently executing processes, with around 100 interconnections. As shown in the Figure 2 we can see there are 4 VME boards and each board has four (4) DSP's. Each hardware resource is referenced to one of the software components.

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

The tools allowed full generation of all executable code. The time to modify a behavior and implement it across the array of processors was approximately 10 minutes. This represents a very large reduction in the effort and time required to adapt the system behavior.

The modeling language was reviewed by practitioners in the high energy physics community. While some details will take training to become natural to the system implementer, the basic concepts in the modeling language were natural to the domain. Overall, the physicists surveyed were encouraged by the results. The prototype was demonstrated at Supercomputing 2003.

5 Acknowledgments

This work is supported by NSF under the ITR grant ACI-0121658. The authors also acknowledge the contribution of other RTES collaboration team members at Fermi Lab, UIUC, Pittsburg, and Syracuse Universities.

6 References

[1] Laddaga R., "Active Software," in Robertson, P., Shrobe, H., Laddaga, R. (eds.): Self-Adaptive Software, LNCS 1936, Springer Verlag, February 2001

[2] Robertson P., "An Architecture for Self-Adaptation and its Application to Aerial Image Understanding," in Robertson, P., Shrobe, H., Laddaga, R. (eds.): Self-Adaptive Software, LNCS 1936, Springer Verlag, February 2001

[3] Osterwel L., and Clarke L., "Continous Self-Evaluation for the Self-Improvement of Software," in Robertson, P., Shrobe, H., Laddaga, R. (eds.): Self-Adaptive Software, LNCS 1936, Springer Verlag, February 2001

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

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

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

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

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

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

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

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

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

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

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

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

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

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