A Technique for Specifying Dynamically Reconfigurable Embedded Systems

Osamah A. Rawashdeh and James E. Lumpp, Jr.
Department of Electrical and Computer Engineering
University of Kentucky
Lexington, KY 40506, USA
859-257-8042
osamah@ieee.org, jel@engr.uky.edu

Abstract—This paper\textsuperscript{1,2} describes a framework for developing dynamically reconfiguring distributed embedded systems supporting graceful degradation. Graceful degradation allows embedded systems to reconfigure in response to faults, allowing the systems to reduce their level of service instead of suffering system failures. The approach is based on a graphical software specification technique. Software module dependency graphs are used to specify the interaction and interdependencies between software modules. Individual software modules can be specified with alternate implementations that may require different amounts of system resources. As failures occur, a system manager tracks system status and uses the dependency graphs to choose new system configurations to deploy. The proposed framework also supports traditional fault-tolerance techniques, such as fail-over programming, redundant calculations, and voting, making it an attractive alternative for the design of a wide range of embedded control applications. A high level description of the proposed system architecture as well as its fault detection and handling are presented followed by discussion of the software modeling.

1. INTRODUCTION

As embedded control systems become more complex, the probability of the failure of the system increases. Recognizing that no building block can be made completely error free, techniques are required to compensate or tolerate the failures. This is typically achieved through redundancy in hardware (and software) to enable fault detection and recovery by fault masking \textsuperscript{1}. A voting scheme can be used to detect the failure of one (or more) system blocks by comparing outputs from three or more replicated modules and assuming the majority are correct. This is an effective fault tolerance technique; however, redundancy for a non-trivial system can be complex and costly in terms of size, weight, cost, and power consumption. A more cost sensitive, but generally less effective, alternative to brute force replication and voting schemes is the employment of reasonableness checks to identify faults \textsuperscript{2,3}. Reasonableness checks entail checking against estimated states. The effectiveness of this method in detecting faults depends on the strictness of these states. Regardless of the fault detection scheme employed, fault masking occurs when a fault is detected, to allow for continued system operation. Traditionally, Fault masking is achieved by using the majority vote or the estimated state to replace the erroneous state.

Graceful degradation is an alternative powerful concept that masks faults by reconfiguring the system. Graceful degradation is defined formally as reacting adaptively to changes in the environment \textsuperscript{4}. A system that gracefully degrades suffers a proportional loss of system utility as individual software or hardware components fail instead of overall system failure \textsuperscript{5}. In our framework, a system can gracefully degrade by either reducing the quality of its outputs or reduce the number of outputs (i.e., shedding non-critical functionality). Graceful degradation is feasible in distributed embedded systems with resources dedicated to non-mission-critical functionality that can be diverted to support critical functionality when necessary.

In this paper we provide a framework system designers can use to specify and implement embedded control systems that support both traditional fault tolerance techniques as well as graceful degradation. The framework centers on a platform-independent modular software specification for embedded computation and a well-defined hardware architecture. A reliable centralized system manager monitors the system for faults (hardware and software) and

\textsuperscript{1} 0-7803-8870-4/05/$20.00© 2005 IEEE
\textsuperscript{2} IEEEAC paper #1435, Version 3, Updated December 10, 2004
reconfigures the system when necessary to provide continued, possibly degraded, system operation. While our framework requires the decomposition of the system into modules, each with more than one implementation, the potential benefits in the resulting design make it an attractive approach.

The following section provides a brief overview of related research efforts. In section 3, an overview of the complete framework is given. In section 4, the software architecture specification using software module dependency graphs is discussed in detail. Finally, section 5 summarizes the approach.

2. RELATED WORK

Research related to graceful degradation has been applied to the areas of traditional server-client networking, webpage design, and in real-time scheduling [6,7,8]. Early work on formally defining graceful degradation for computer systems was done by Herlihy and Wing in 1987 [4]. A relaxation lattice is used to describe system behavior in different operating conditions. A relaxation lattice is a lattice of specifications parameterized by a set of constraints. Changes in the environment force the system to satisfy a smaller set of constraints that is not optimal operation but is tolerated. The explicit nature of the lattice makes it necessary to examine a priori the system's response to every expected failure and failure combination.

Researchers are also applying graceful degradation to distributed embedded systems [5,9]. The RoSES project is defining a framework to enable analysis and design of scalable gracefully degrading distributed embedded systems [5]. The RoSES reconfiguration algorithm finds all possible software configurations of the system for every hardware configuration and selects the highest utility software configuration to run. Finding a new configuration is done while the system is offline and considers non-critical functions, or features, of the system. RoSES assumes that the critical functionality of a system is made fault tolerant by other means.

An architecture for safety-critical reconfigurable systems is being developed at the University of Virginia [10]. This architecture uses an a priori defined set of specifications (i.e., possible configurations) to simplify assurance of critical system components. This architecture assumes a robust system core and a set of sub-systems with each subsystem having at least one redundant copy. A configuration is chosen that selects a combination of working sub-systems to achieve full system operation. Degraded operation is facilitated by providing complete backup sub-systems (HW and SW) with possibly different-qualities.

The Chameleon project is developing modeling techniques to specify and analyze the behavior of dynamically reconfiguring systems [9,11]. Chameleon is abstract and focuses on prediction and testability of configurations. All possible configurations are testable at design time.

The Simplex architecture was developed to provide robustness for complex software systems [12,13]. It has two main components: the high assurance and the high performance control system. The high assurance control system is a highly reliable implementation with limited but critical functionality. The high performance control system is more complex and is designed to maximize functionality. In normal operation, the high performance control system maintains system operation. The high assurance control system monitors system behavior and takes over if a fault is detected. The simplex architecture may be viewed as gracefully degrading to a single operating mode in the event of a failure.

The software module dependency graphs used in our framework share some aspects with success trees used to model industrial systems for fault diagnosis and risk analysis [14] and computer systems for reliability modeling [3]. In general, nodes in a success tree represent objects needed to achieve a common goal. The goal is the root of the tree. The objects at a level are connected through “and” and “or” gates to objects in higher levels. The “and” gate is used to denote a group of necessary operational objects and the “or” gate is used to denote redundant objects of which only one needs to be operational. Our dependency graphs can be viewed as extended success trees. We include combinatorial gates that allow the specification of purely optional objects as well as annotating objects with quality values. Furthermore, there are two kinds of objects in our graphs: data variables and software modules. We use the graphs to specify all possible combinations of data variable requirements for every software module. Hence, our graphs show the flow of data instead of the binary values “operational” and “non-operational” exhibited in success trees.

Our proposed framework supports the design and implementation of a scalable gracefully degrading system, where fault combinations of non-critical system components need not to be explicitly considered a priori. It attempts to provide fault tolerance for the entire system (hardware and software), including the critical components, and attempts to allow the reconfiguration to be done while the system is online. We achieve graceful degradation on a low level by platform independent software modules, with multiple degraded implementations, that can be scheduled on most any available processors in the system. In summary, we target safety-critical systems and develop an abstract framework that includes the specification of mechanisms such as: distributed fault detection, run-time reconfiguration, and resource management. In the following section, the overall system architecture of our approach is described.
3. System Overview

In this section, an overview of the framework is presented. First, the assumed hardware architecture is discussed. An introduction to the software module dependency graphs is then given and the runtime behavior of the system is discussed including fault detection and fault handling techniques used by the system manager.

Hardware Architecture

The architecture model is shown in Figure 1. The top right block represents the system manager subsystem. The four processing elements (PEs) represent the processing resources of the system. Each PE can host a number of input and output devices (sensors and actuators) that are represented as triangles in the figure. A robust real-time communication network interconnects all processors and the system manager.

Processing Elements— A PE is a hardware unit with a processor and a network interface(s). Each PE runs a local operating system and receives scheduling information from the system manager. The object code of software module implementations that can be scheduled on a PE is available in local non-volatile storage. A PE can connect to any number and combination of I/O devices. PEs with no I/O resources function as pure computational PEs.

I/O Devices— An I/O device is a hardware component, connected to a PE, that performs an input and/or output function. All sensors and actuators as well as bidirectional interface devices (e.g., communication radios) are considered I/O devices. The lowest level software modules that directly interface to I/O devices must be mapped to the PE that hosts the I/O device. To achieve redundancy, redundant I/O devices of critical functions should be hosted by distinct PEs. I/O devices of varying quality or capacity used to support graceful degradation of the system.

Communication Network— There are two types of communication links in the real-time communication network. The manager deploys software scheduling information to the PEs through the management link. The management link is also used to send messages from fault checkers, discussed later, to the manager. Fault checkers have more critical timing constraints because they notify the manager of failures. The second type of link is the data link that is used to move data and state variables between software modules. Depending on the application and its bandwidth requirements, the links may be implemented on one or more (possibly redundant) networks. In either case, the networks are transparent to the software modules and are handled by the local operating system.

Software Architecture

Overall system functions are represented as software module dependency graphs. Dependency graphs (DGs) are directed graphs that connect input devices to output devices via a set of nodes. Figure 2 shows a simple example of a DG. From the left to the right hand side: a sensor is connected to a software module (sensor driver) that outputs the sensor value as a data variable (sensor reading). A computational module uses the data to calculate an output data variable (actuator setting). The AND symbol specifies the dependency of the computational module on the sensor reading data variable. The actuator driver software module reads the actuator setting data variable and sets the actuator accordingly.

Software modules may operate with a set of input combinations as specified by the interconnection symbols. Therefore, several implementations of a module may exist. One implementation is chosen based on the available resources in the system. The system manager uses the DGs to find the system outputs that must be provided and it traverses the graph back to the sensors to find a possible implementation. This runtime behavior of the system is discussed in more detail in the following subsection. The specification of DGs and their application is described in detail in section 4.

Runtime Operation

Fault tolerance is achieved by detecting failures and
subsequently responding to their occurrence. In this section, the runtime-behavior is briefly discussed in terms of fault detection, fault handling, and configuration determination. Furthermore, new configuration deployment and system reconfiguration due to priority changes are briefly overviewed.

Fault Detection — There are two basic methods of detecting faults: a systematic approach and an application specific approach [2]. First, the systematic approach entails the replication of system components and employing a voting scheme. Faults are detected by comparing two or more sources of the same information. This traditional approach is an effective method of detecting faults but it is also a costly one. The application specific approach is based on application knowledge and checking data for reasonableness. This includes range checking, state estimation, and rate-of-change limits on specific data items. Both the systematic approach and the application specific approach to fault detection are supported by our framework.

The application specific approach is achieved in our framework through fault checkers. Every software module can have a fault checker component that acts as wrapper/monitor for critical data variables. The systematic approach can be implemented by duplicating sensors and actuators and adding spare PEs. Software modules can also be duplicated (or n-version implementations used). To detect faults in redundant components, software modules that have a voting function may be used. In either case, when a fault is detected, an error message is sent to the system manager for notification.

There are a number of other failure detection techniques that can be employed. Using internal timers, fail-and-stop errors of sensors may be detected by watching for time-outs on output data. It may be possible to detect failure of output devices if there is some form of feedback into the system. A PE failure may be detected by the system manager by monitoring the absence of heartbeat messages or by monitoring the status of the software modules scheduled on it. Furthermore, the local OS on a PE can detect violations (e.g., stack overflow, memory space violations, etc.) of software modules.

Fault Handling — The system manager is the supervising centerpiece of the dynamically reconfiguring system. The fault detection mechanisms discussed above provide the manager with the condition of the hardware resources and of software modules.

Both a masking and fail-safe fault tolerant behaviors [15] are supported. Masking fault-tolerance entails the ability of the system to mask faults from other system components guaranteeing continued correct operation of the system. Fail-safe behavior entails placing the system in a safe mode (where all output devices are in a predefined “safe” state) if a non recoverable fault has occurred. In our framework, fault masking is achieved by system reconfiguration. Fail-stop behavior can be implemented by implanting a fail-safe version of software modules that drive output devices so the output devices are set to a “safe” state when the rest of the system fails.

When a failure occurs, the manager finds a configuration for the critical functions first. The remaining resources are then allocated to run non-critical functionality.

Critical Sub-system Configurations — Because the critical functions of a system must be reliably carried out, the set of possible configurations may have to be determined at design time to allow for more rigorous verification and validation. In larger systems, this set may become too large. In such case, only a few selected configurations may be stored in order to save memory space. The minimal safe-mode configuration must always be part of the set. Having predefined configurations enables a rapid system reconfiguration minimizing system downtime.

Non-critical Sub-system Reconfiguration — No static configuration table is stored by the manager for non-critical functions. Instead, the manager searches the dependency graphs and finds the best configuration possible given the hardware resources available after the critical functions were scheduled.

Module to PE Assignment — The successful assignment of software modules to PEs is based on the availability of free resources on the PE (e.g., memory, processing power, and communication bandwidth) to run the module. Modules may be scheduled to PEs with slack if they do not have hard deadlines.

New Configuration Deployment — After the manager determines a new system configuration in response to failure(s), the appropriate control messages are sent to the local OS on each PE. These messages indicate which software to schedule on the PE and in which mode each software module should operate.

Reconfiguration due to Priority Changes — The current focus of this work is on dynamic system reconfiguration to provide fault tolerance. In some embedded applications priorities of system sub-functions may change. To accommodate this property, the priorities of output software modules may be a function of a “mission state” variable. When the state of the mission changes, the system manager performs a reconfiguration process with the updated priorities of the system sub-functions. Hence, the landing data-flow sub-graph modules of an autonomous aircraft, for example, will only be consuming hardware resources in the final stages of its mission. Therefore, the system manager can reconfigure the system due to changes in the function priorities as it does when faults are detected.
Conclusion

In this section, an overview of the system components and the runtime behavior was described. In the following section, software module dependency graphs are described with examples.

4. MODULE DEPENDENCY GRAPHS

Structuring embedded applications in a modular fashion can assist in fault prevention in several ways. Breaking complex software functions into modules encourages encapsulation and software reusability. The software module dependency graphs show software module dependencies and redundancies giving the designer an overview that may reduce errors.

Dependency graphs (DG) provide the basis for determining system configurations and making scheduling decisions. Dependency graphs are directed graphs that specify the flow of data and the dependencies among software modules. The software architecture of the system is represented as a set of dependency graphs for each system function (e.g., system output). DGs show the flow of information from input devices to output devices through a series of software modules that perform the required computations. Data variable nodes represent the information passed between modules. The data variable requirements of a software module are represented in the graphs by a set of interconnection symbols (comparable to logic gates). The software modules that form the basic degradable units of our architecture and the dependency graphs are described in this section.

Software Modules

A software module is a quantum of executable code that is schedulable by the OS running on a PE. There are three kinds of software modules: input, output, and computational software modules. Input software modules are software modules that obtain data produced by input devices (i.e., sensors) and have one or more data variables as output. Output software modules send data to one or more I/O devices (e.g., actuators). Output software modules are classified as mission-critical or non-mission-critical. Non-mission-critical output software modules are assigned a priority number according to their significance to the overall system operation. Computational software modules have data variables as inputs and as outputs. The inputs, outputs, and function of a software module are captured by its module specification.

Only one implementation of a software module is scheduled at any given time. The local OS manages the exchange of the input and output variables with other modules over the system network. A module’s state variables, on the other hand, contain state information that is necessary when the module is restarted (possibly in a different operational mode) to continue correct operation. The restarting of modules occurs after a system reconfiguration. Therefore, all state information must be saved by modules and stored in non-volatile memory by the system manager. State storage intervals, or checkpointing intervals, may be static and set at design time for every software module. It has been argued that static checkpointing schedules are a reasonable assumption for real-time embedded applications [16].

Each module is annotated with its resource requirements and with a quality value associated with its output. The higher the quality value the better the overall system operation. The highest quality output is produced when the highest quality inputs to the module and sufficient processing resources are available.

Graphical Representation of DG Elements

There are seven different types of nodes in a DG: input devices, output devices, data variables, and three combinatorial interconnection symbols. Nodes are connected through edges implying unidirectional data flow and are annotated with a bandwidth requirement. On one end of the graph, input devices connect directly to input software modules and produce data variable(s) as outputs. On the other end of the graph, output software modules connect directly to output devices. Between these, there are one or more of the following set of nodes (in order): data variables, a combination of interconnection symbol(s), software modules, and then data variables.

Figure 3- Node Types in a DG

A software module with a single operating mode is represented as a square node (top of Figure 3). The Node is labeled with the function name. Modules with more than one operating mode can use one or more combinations of
input variables and produce one or more quality versions of output variables. A three mode software module is shown in Figure 3.

Data variables are outputs of software modules and are connected to their source module by an edge. Data variables are also inputs to software modules and are connected to their destination modules through interconnection symbols (discussed below). A data variable is represented as a circular node in the DGs (Figure 3). They are labeled with their name and an integer \( m \) that specifies the number of different quality versions available.

**Figure 4-** Representation of I/O Interface

Input devices and output devices are shown at the bottom of Figure 3. They connect with an edge directly to their corresponding input/output software module without going via a data variable node (e.g., Figure 4). Bidirectional I/O devices, such as communication links, are graphically represented with two blocks: a sensor and actuator for incoming and outgoing data, respectively.

**Figure 5-** Data-flow Graph Gates

Three types of symbols used to interconnect software modules with data variables are shown in Figure 5. The output of the logic symbol (D) feeds either into an input of another gate or connects directly to a software module. The inputs to the AND and OR symbols can be data variables or outputs of other symbols. The inputs of the XOR symbol are always data variables. The AND symbol can have one or more inputs and exactly one output. All inputs (A, B, C) to an AND are required by the object connected to its output. Secondly, the OR symbol can have one or more inputs and exactly one output. The object connected to its output requires at least one of the inputs (A, B, C) to be available, but as many as possible is preferred. If there is only one input, then this input is not necessary but is preferred. Finally, the XOR symbol can have two or more inputs and exactly one output. The XOR gate specifies that exactly one of the input data variables (A, B, C) is required by the object connected to its output. The highest quality (Q value) input is preferred.

**Figure 6-** Graphical Representation of a Software Module

Figure 6 further illustrates the use of the interconnection symbols. It shows a partial dependency graph with a single software module and its I/O variables. The outputs of the software module are data variables \( X \) and \( Y \). The module can produce these two output variables when the following input variables are available (i.e., required inputs): the variable \( D \), the variable \( E \), and either \( A \) or \( B \). There may be more than one quality version of these necessary inputs (e.g., \( a, b, d, e, f, g, \) and/or \( h \) may be \( > 1 \)). The higher quality these inputs are the higher quality the output \( X \) and \( Y \) will be. The partial DG also identifies an optional input variable \( C \) by use of a single input OR symbol. The quality of the data variables \( X \) and \( Y \) is improved if input \( C \) is available and is further improved if the quality of \( C \) is improved (when \( c > 1 \)).
The quality of data variables is set by the software modules producing the variable; it is computed by the \textit{quality function} of the module that is a function of the input variables and their qualities. To simplify the DGs, the data variable node \( Y \) is labeled with an integer \( m \) to indicate that there are \( m \) different quality versions possible of \( Y \). Graphically, this is represented as shown Figure 7 (a). The Figure 7(b) shows the equivalent exploded view of the same data variable \( Y \).

Figure 8- A Software Module with three Implementations and its Abstraced View

A software module with three modes of operations is shown in Figure 8 (a). Each mode has a combination of required inputs. A system is designed using this detailed view to show every operating mode of a software module as a block with its own inputs and outputs. Module 1, for example, can work with either input \( A_1 \) or \( B_1 \). Furthermore, the module produces \( X_1 \) and one of two versions of \( Y \). These dual quality output variables can be further exploded by using the equivalent representation shown in Figure 7. An abstracted view maybe used to view the dependency graphs to hide low level information (Figure 8 (b)). This view shows that the software module has 3 different modes and two output variables: \( X \) and three versions of \( Y \). Furthermore, the graph shows that the module requires the input variables: \( A \) or \( B \) or \( C \) and one version of \( D \). If at least one of the input conditions is met, the outputs \( X \) and \( Y \) will be produced. However, it is not clear from this graph which implementation uses which inputs to produce which version of the outputs. There must be an implementation that can handle every combination of input variables. An implementation can cover more than one operating mode. This decision is made by the designer.

**System Dependency Graph Example**

Figure 9 shows an abstracted view of a DG for a hypothetical flight control system with an emergency parachute function for a simple autonomous aircraft. The sensors are along the left side of the graph and the actuators are on the right side. The system has two 3-axis gyroscopes and two 3-axis accelerometers for redundancy. Using the XOR symbols, only one source for the roll rate and acceleration rate is required by the pitch and roll angle estimating software modules. The estimated angles are both required by the flight stability control module in order to produce the settings of the control surfaces (elevator and rudder). A third output of this control module is the parachute release command that is asserted if the UAV loses flight stability. The parachute release command can also be asserted by the parachute release logic module that watches for excessive airspeeds or drastic changes in altitude. Both release commands are desirable but are still optional inputs to the parachute actuator driver module. The actuator driver module will release the chute if it receives a command to do so. If these variables were not available (i.e., optional inputs are not available), the actuator driver module would release the parachute by default.

It is valuable to note that the number of possible input combinations to a software module does not necessarily equal the number of module implementations. Similarly the number of implementations of a module does not necessarily equal the number of output variable versions. In Figure 8, there are only 2 implementations of the pitch angle estimator that can operate with the different versions of the pitch rate and 2 versions of acceleration rates for every axis. Similarly, There are two versions of the flight stability control unit that operate using two different versions of input variables but the output is always the same quality. There are no fixed rules to specify these relationships. Rather, they are set at design time, through the implementation of the module, and can be represented in the normal view of the DG.
Figure 9- Example of an Abstracted View of a System Dependency Graph
Classification of Data-Flow Graphs

All output software modules (software modules that interface with output devices) are classified as either safety-critical or non-safety-critical. Furthermore, each output module is assigned a priority value to describe its importance to the overall system function. All modules in a particular graph inherit the classification and priority of the output module they support. The classification and priority information is needed by the manager for reconfiguration.

Level of Software Modularization

There is no firm rule to specify to what level the software should be modularized when developing a DG for a specific application. However, some general rules may be followed. The function a module implements should be small enough to simplify verification and validation and possibly allow for reusability. The modules should not be so small that the overhead to the multi-tasking OS and the manager search algorithms become a limiting factor. Excessively large modules will limit the number of possible system configurations. Every module should further more have a reasonably distinct function such that examining the DG will allow the identification of critical functions to the overall system operation and allow the identification of their redundancies and single points of failure.

5. CONCLUSION

We have introduced a framework for designing gracefully degrading distributed embedded control systems. A system designed using the framework provides fault tolerance by using idle resources or resources providing non-critical functionality as backup for critical functions, eliminating the need for redundant copies of a sub-system and reducing the overall cost, size, weight, and power consumption. The framework eliminates the need to specify and design for all possible failure modes affecting non-critical system functions. It uses a modular software design approach that can simplify verification of software components on a low level and encourages reusability. The framework also decomposes the system into sub-functions, each represented by data-flow graphs that can give the designer a better overview of dependencies and failure conditions. Decoupling the software design from the target hardware may simplify design and testing by defining uniform software interface standards. The framework allows the incorporation of traditional fault tolerance techniques such as redundancy and voting. The techniques provide a systematic way of designing and implementing a dynamically reconfiguring control system.

Theses techniques are being applied to the development of a high-altitude Unmanned Aerial Vehicle (UAV) being developed at the University of Kentucky. Up to date, traditional fault tolerance techniques were employed [17,18]. It is anticipated that the design methodology will be refined through this application. The resulting UAV avionics system will be used to experiment with the fault tolerance capability of the system by inducing combinations of hardware and software failures and observing the system’s response and behavior.

ACKNOWLEDGMENTS

The authors would like to thank David Feinauer, Casey Harr, David Jackson, Garrett Chandler, and Adam Groves. Their reviews and suggestions have greatly improved the quality of this paper. This work was supported in part by the Kentucky Science and Engineering Foundation (KSEF), the Kentucky Space Grant Consortium (KSGC) and the NASA Workforce Development Program.
REFERENCES


BIOGRAPHY

Osamah A. Rawashdeh is a Ph.D. Candidate in the Department of Electrical and Computer Engineering at the University of Kentucky. He received his BS and MS in Electrical Engineering from the University of Kentucky in 2000 and 2003 respectively. He is a member of IEEE, ACM, ARRL, and AIAA. His research interests include reconfigurable computing and distributed fault tolerant embedded systems.
James E. Lumpp, Jr. is an Associate Professor in the Department of Electrical and Computer Engineering at the University of Kentucky. He received the BSEE and MSEE degrees from the School of Electrical Engineering at Purdue University in 1988 and 1989 respectively, and the PhD from the Department of Electrical and Computer Engineering at the University of Iowa in 1993. He joined the faculty at the University of Kentucky in 1993. He is a Member of IEEE, ACM, AIAA, and SWE and the Center for Advanced Networking and the Advanced Science and Technology Commercialization Center at the University of Kentucky. He has worked as a visiting scientist or intern at AT&T Bell Labs, Rockwell International, Northrop Defense Systems Division, and Oak Ridge National Laboratory. His research interests include distributed embedded systems, safety critical systems, high-performance distributed computing, and unmanned aerial vehicles.