A Methodology and Tool Support for the Design and Evaluation of Fault Tolerant, Distributed Embedded Systems

Mark Lee McKelvin Jr

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-35
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-35.html

April 29, 2011

A Methodology and Tool Support for the Design and Evaluation of Fault Tolerant, Distributed Embedded Systems

by

Mark Lee McKelvin, Jr.

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Alberto Sangiovanni-Vincentelli, Chair
Professor Francesco Borrelli
Professor Jan Rabaey

Spring 2011

A Methodology and Tool Support for the Design and Evaluation of Fault Tolerant, Distributed Embedded Systems

by

Mark Lee McKelvin, Jr.

Abstract

A Methodology and Tool Support for the Design and Evaluation of Fault Tolerant, Distributed Embedded Systems

by

Mark Lee McKelvin, Jr.

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Alberto Sangiovanni-Vincentelli, Chair

Embedded systems are becoming pervasive in diverse application domains, such as automotive, avionic, medical, and industrial automation control systems. Advancements in technology and the demand for sophisticated functionality to support a variety of applications are driving the increase in complexity of embedded systems, particularly in systems whose incorrect operation can result in significant consequences, such as financial loss or human life. As a result, these systems require high assurance to meet stringent constraints on reliability and fault tolerance, the ability to operate despite potential for components to operate incorrectly.

Reliability is an important design goal in distributed embedded systems that may be achieved by the provision of additional components in parallel or by improving component reliability. Thus, reliability in a fault tolerant system will be dictated by the combinations of components that operate incorrectly, or fail. Since, redundancy comes at a cost, the problem that designers face is determining which components to improve. Most existing approaches that seek to achieve better system reliability by determining levels of component redundancies and a selection of component reliabilities simultaneously do not consider the design of embedded systems. Of the approaches that do consider applications in the design of embedded systems, many do not consider the combinations of component failures, their location in the system architecture, and rate of failure due to the challenges and limitations of constructing reliability models that can express those characteristics.

In this dissertation, I present a design flow and a set of tools to support the design and analysis of distributed embedded systems with fault tolerant and reliability requirements using fault trees. A fault tree is a reliability model that is based on the failure characteristics of a system and its structure. The proposed design flow integrates the automatic generation and analysis of fault trees to enable the design

of fault tolerant architectures. I will apply this design flow to the evaluation of a fault tolerant control application and to the evaluation of architecture alternatives for an automotive application.

I'm glad I did it, partly because it was worth it, but mostly because I shall never have to do it again. - Mark Twain

Contents

List of Figures

List of Tables

Acknowledgements

I thank God for getting me through this! It has been a journey of ups and downs, but with the help, encouragement, and support from many people at Berkeley and beyond have helped to make it possible for me to complete this degree. I would like to thank first and foremost my advisor, Alberto Sangiovanni-Vincentelli for his patience, support, and for the opportunity to pursue the work in this dissertation. I also want to thank Professors Jan Rabaey and Francesco Borelli for being the other members of my dissertation committee and for providing helpful feedback. Thank you for being members of my qualifying committee, as well. I'd also like to thank Professor Brayton for being a member of my qualifying committee and providing valuable feedback.

I have met and interacted with many students, researchers, Professors, and staff who have in some way helped me get through these years at Berkeley, academically, socially, and mentally. I'd like to especially thank Claudio Pinello for his ealier ideas on the foundation of this work and for his support throughout. He has been my mentor on this work and I will always be grateful for having the opportunity to meet and interact with him. I thank Paolo Guisto for his encouragement, patience, energy, and advice. I would like to thank all the members of the Metropolis group that were patient enough to answer my questions and engage in thoughtful discussions, including Doug Densmore, Alessandro Pinto, Alvise Bonivento, Trevor Meyerowitz, Qi Zhu, Haibo Zeng, and Arkadeb Ghosal. Thanks to Professor Sanders for having a positive attitude, and I am grateful to have met you. Thank you Professor Subramanian for the trips to recruit other bright candidates for UC Berkeley. Thanks to Professor Goldberg for taking me as an advisee early in my studies at UC Berkeley and for providing the opportunity to work with a great group of students on cool robotics at a San Francisco high school.

Thanks to Mary Byrnes, Sheila Humphreys, Ruth Gjerde, the former MEP office - Michelle, Roni, Meltem - for always having a shoulder to lean on. Thanks to Carla Trujillo and Sheila, who both welcomed me to Berkeley when I first came. Sheila, thank you for everything you have done for me and for being a leader in diversity and equity in not only EECS but throughout the school. You all have made a profound impact on my well-being during my matriculation at U. C. Berkeley. I appreciate being able to come to your offices and shed a few tears over personal and academic matters or just come in to let off some steam. You really made this experience for me that much more bearable. You have provided outstanding guidance and support. Thanks to numerous staff members who I have met throughout campus, as well.

Thanks to all the friends I have met while at Berkeley. Thank you BGESS and BESSA for having a place for me to go when I felt there was no other place. Thanks to the people who I have met from BGESS, including Greg Lawrence, Hakim Weatherspoon, Ayodele Harris, Kofi Boakye, Kofi Inkabi, Nerayo, Miller Allen, Qwasi Kapori, Freddie, and the new faces who have also been supportive and a

pleasure to eat dinner with on Monday evenings. Thank you Rey Guerro for hanging out and playing basketball at the gym. Jorge, thanks for the weekly/monthly coffee breaks.

Last but not least, I'd like to thank my immediate and extended families for supporting me on this journey. Although my time has been wrapped up in school for most of my life, you have been the foundation from which I can stand tall, keep my head up, and never be afraid to accept a challenge, even when it means that I would be over 3,000 miles away. Thank you for teaching me to always say "thank you" with a smile and to be humble. You did not always know exactly what I was doing, and sometimes I didn't know either, but you always kept pushing me to never quit. Special thanks to my wife Tiffany and kids Breanna and Ayanna for understanding and giving me the time to do work even after long hours on campus. Now, we can spend more time at Disneyland! Thanks to my mom, "Snakelady", for her motivation and encouragement to keep going, despite the obstacles that have come my way thus far, my siblings for looking up to me and giving me the motivation to keep moving forward, my dad - Mark Sr., Evon, uncles, aunts, cousins young and old, and all of my family back in Georgia. Thank you "Bo", Sherri, and Scott for providing me a place to lay my head when I first came to California.

I'd also like to send special thanks to my Clark Atlanta University family for inspiration, motivation, encouragement, and for leading me on this path of pursuing a Ph.D., especially Dr. Musa Danjaji, Dr. Melvin Webb and staff, the entire Engineering Department, Dr. Jenkins, Mrs. Morgan for marking up my English papers, Dr. Brown for making me learn physics the right way, and Dr. Parker for cool chemistry labs, even though the lab was on a Friday evening in Atlanta. You all were my inspiration for higher education. Ryan Russell, Avery Cooper, Harold Gaines, "T" Hill, and Kennard Love, thank you for the experiences and support throughout. Thanks for the lasting relationships. And, thanks to the people of my hometown, Sylvester, Ga. You have influenced and motivated me in ways unimaginable, and no matter what happens, where I have been, or intend to go, you will always be "home".

Chapter 1

Introduction

Electronic systems are becoming pervasive in our daily lives as well as in aspects where their proper functioning is crucial. An electronic system that is embedded within the context of a larger system is often hidden from the immediate view of the end-user. Examples of such systems span from mass-produced consumer electronics, such as smart phones and automotive systems, to autonomous spacecraft systems and pacemakers. Consumer demands on more functionality for comfort, convenience, and safety, along with the pressures for suppliers to increase productivity to meet these demands, makes the design of these systems complex due to smaller dimensions of the electronic hardware, increasing use of software, and a greater number of interconnected parts. This complexity introduces more opportunities for parts or the system as a whole to function incorrectly.

As technology advances, electronic systems are increasingly being deployed in critical applications, such as applications whose incorrect operation may result in significant consequences. As a result, system designers are faced with the challenge to design these systems such that they tolerate defects or abnormal conditions that may result in the inability for the system to perform its function correctly. A *fault* is an abnormal condition, or defect, in one or more components of a system. When a fault is present and active, it can manifest into an *error*. An error then can result in the *failure* of a component or system. Thus, a *fault tolerant system* is a system that may continue to function as intended, potentially in a degraded mode of operation, despite the occurrence of faults. The ability for a system to perform its intended function for a specified amount of time is the *reliability* of a system, and it is a measure of fault tolerant design. This chapter provides a motivation for this work on the fault tolerant design of electronic systems for critical applications.

1.1 Characteristics of Embedded Systems

In the context of this dissertation, an *embedded system* is a specialized computing device that is designed to perform specific tasks as an integral component within the context of a larger system. Such systems are characterized by a continuous interaction with its environment using sensors and actuator devices. Most modern embedded systems contain one or more software programmable processing components, such as a Field Programmable Gate Array (FPGA), Application Specific Instruction Set Processor (ASIP), microprocessor, or micro-controller Technology advancements have made it possible to integrate more software code onto programmable devices, such small operating systems, application tasks, and software that manages the hardware resources. Software that executes on an embedded device is called *embedded software*. Such systems are characterized by a *heterogeneous architecture*, i.e., they consist of a tight integration of both software programmable and dedicated hardware components in a physical structure that is dictated by a specified behavior. Furthermore, an embedded system must satisfy strict requirements on the response time to inputs from the environment while executing on a limited set of hardware resources [42].

An embedded system that is composed of multiple processing components that interact with one another through a network is said to be *distributed*. In the context of a distributed embedded system, the processing components are referred to as *nodes*, and each node is an autonomous unit that provides hardware resources to the software *tasks*. A task is a unit of computation that computes the outputs of a function, and multiple tasks communicate with one another using units of data referred to as *messages*. *Channels* represent the communication medium by which the tasks exchange messages, and they may connect tasks locally on the same node or remotely on networked nodes. In the former case, message passing between tasks occur within a shared memory space that is located on the processing node. In the case that tasks must communicate remotely across the boundaries of a node, tasks communicate using communication networks, such as a communication bus, wireless channels, or Ethernet. A collection of tasks communicate to provide services that are viewed by the end user as a single application, and it is realized by a *distributed architecture*. A distributed architecture is a conceptual representation of the software and hardware components of a distributed system, their interfaces, methods of communication, behaviors, and properties that are of interest to the designer or developer.

1.2 Motivating Factors

Advancements in technology and the demand for more sophisticated functionality to support a variety of applications are key drivers in the increasing complexity of embedded systems. Given that hardware solutions can become costly to support complex applications, embedded software has become more prominent in embedded systems. Sophisticated applications, such as those employed within systems whose failure can result in severe consequences, require high assurance to meet stringent constraints on reliability. Such requirements leads to standards and certification processes, while simultaneously striving to meet market demands on product delivery. Moreover, embedded systems are moving away from large, expensive, and centralized architectures to smaller, less expensive, and highly distributed networks of embedded devices. This section will highlight the key factors that will motivate the work in this dissertation.

1.2.1 Complexity of Embedded Systems

The complexity of an embedded system, as indicated by the number of functions that are provided to the end user, is the outcome of several trends. Complex systems are systems whose functionality is a measure of the cognitive effort that is needed to understand a physical process under design and development [58]. The complexity in an embedded system extends from the number of features and functions provided by the electronic hardware, implementation of functions in the embedded software, and in the interactions between multiple processing units that are interconnected by communication networks to realize a set of functions. It is commonly accepted that as the complexity of components in a system increases with new functionalities, then designing the composition of components in a system also increases in complexity. Therefore, it is important to address the design of complex systems at the system level where the attention is focused on the behavior and structure of the system as opposed to limiting one to the details of its constituent parts. System complexity presents challenges to designers and developers as the nature of applications limit the comprehension of detailed functionality using traditional design methods.

Technology Advancements in Electronic Hardware

The impact of Moore's Law [82] on the semiconductor industry has spawned new fabrication processes, tools, and methods that enable the exponential increase in the number of transistors per silicon die. As a result of technology scaling [44], more features, or functions, can be implemented in hardware on a single die, such as analog components, digital logic, arithmetic logic units, memory, and intercon-

nect at lower costs. The decrease in feature size is having an adverse impact on the long-term reliability of electronic devices. Device miniaturization due to scaling reduces the thickness of semiconductor material in transistors. Although this causes a decrease in gate delay and interconnect, it also increases the current density. This phenomenon manifests itself into higher temperatures which causes an acceleration in the wear out of an electronic circuit, thus, decreasing its lifetime reliability. The effects include electromigration, stress migration, gate-oxide breakdown or time dependent dielectric breakdown, and thermal cycling [117]. The work by Srinivasan *et. al.* [116] highlights the impact of technology scaling on the reliability of microprocessors and its workload. The results of that work imply that as technology scales and more functionality is added to electronic devices, the failure rate will also increase. Therefore, the complexity in electronic hardware has a profound impact on the reliability of electronic devices. White [126] provides more details on the impact of technology scaling on electronic circuit reliability.

Embedded Software

Given the cost of having to either redesign or employ many hardware devices to support the growing demands on greater functionalities and services, embedded software is playing a much greater role in embedded systems. For example, in an recent article published by the Institute for Electrical and Electronics Engineers (IEEE) [20], it is reported that the new F-35 Joint Strike Fighter that is scheduled to be the next advanced military aircraft will require approximately 5.7 million lines of code to operate its electronic systems, and luxury automobiles have well in the neighborhood of 100 million lines of code to control functions such as airbag deployment, anti-lock brakes, and engine control that are distributed over 80-100 microprocessors. Hence, embedded software is a dominant factor that is contributing to system complexity.

The complexity of embedded software has a significant impact on the reliability of the system in which it is deployed. Software is a human activity that reflects the understanding of a problem and results in a physical artifact, or a software program. Woodfield [128] concludes that the complexity of the problem is directly related to the understanding by the programmer. Hence, as more sophisticated functions are implemented in software, the complexity of the software often leads to the inability for the programmer to predict the behavior of the software during operation [37]. This leads to errors in the design of software that may result in system failure. It is reported in a study by Lutz [76] on embedded software for distributed embedded systems that design errors in critical applications most commonly arise from a lack of understanding between the specification of system requirements and their implementation in embedded software. Techniques in the development of embedded software, such as testing, code reviews [53], and formal methods [91] are used to

improve software reliability by reducing design faults. However, as the complexity of software increase, the ability to achieve reliable software for embedded systems becomes more difficult [38, 76].

The consequences of errors in embedded software can be costly. For example, a design error led to the explosion of the Ariane 5 [64] spacecraft. An investigation revealed that the failure was caused by the inability to convert a 64-bit decimal value to a 16-bit integer. The cost of the Ariane 5 failure is reported to be over $500 million dollars. More recently, software-related errors are the cause of an increase in recalls by car manufacturers [100, 102]. A 2002 report by the National Institute of Standards [110] states that software failures cost the United States economy $59.5 billion dollars annually. Faults in embedded software have also resulted in loss of life upon system failure, as evidenced by failure in the Therac-25 drug delivery system [70] and the inability for the Patriot missile defense system to intercept an incoming missile due to a software error [86].

Distributed Architectures in Safety-Critical Applications

Due to lower costs in electronic hardware and the increasing use of software to meet the demand on more functionality, distributed architectures are more commonly used to support safety-critical applications. A *safety-critical* [57] application is one whose failure may result in a loss of life, severe financial loss, or cause serious harm to the environment in which it operates. Whereas reliability is an attribute of the system that involves the probability of failure, safety is concerned with the *consequences* of failures. Therefore, safety-critical systems are designed to be highly reliable and to mitigate the effects of failures. Common examples of distributed architectures to support safety-critical functions are found in avionics fly-by-wire control systems, as described by Yeh [133] and Isermann [46] describes several safety-critical applications in modern passenger vehicles (e.g., active steering and braking assistance). Moreover, emerging applications such as brake-by-wire [129] and steer-by-wire [45] will replace mechanical and hydraulics parts with no mechanical backup.

A distributed approach has several advantages, such as the ability to expand functionality and manage complexity by decomposing functions into more manageable components with well-defined interfaces [41]. However, as pointed out by Rumpler [112] the communication and coordination that is required in a distributed architecture causes an increase in complexity. Inherent in the trend to realize safety-critical functions on distributed architectures, there is a shift in the design and development of architectures from complete functions on a single node towards functions that share more common resources between nodes. Given that the automotive industry is very cost-sensitive and produces vehicles in mass quantities, the industry is realizing the benefits in applications that are realized by a network of shared

resources. Thus, the design of electronic controls in automotive systems illustrates this trend. Examples of this trend is observed in other industries as well, such as commercial and military avionic applications [14, 123, 127].

Example: Evolution of Architectures for Automotive Electronics

Introduced by the avionics community [124], the trend towards realizing embedded systems on a distributed architecture can also be observed in the shift from *federated* to *integrated* [41] architecture designs in modern automotive systems. Automotive systems have traditionally been federated, meaning that each function (e.g., braking or steering in a car) is developed in isolation and has its own node, or Electronic Control Unit (ECU) [99], that consists of hardware and software components that contain its own processing, input and output interface, application tasks, and direct connections to sensor and actuator devices with only minor communication between ECUs of other functions. The isolation between ECUs provides a strong barrier to error propagation because the systems supporting different functions do not share resources, the failure of one function has little effect on the continued operation of others [84]. Such containment of faults results in a fault containment region that defines an independent relationship between ECUs. Moreover, the federated approach to system design allows designers to manage the complexity of individual ECUs in isolation without analyzing or understanding the rest of the system, hence reducing the amount of complexity that are characteristic of distributed architectures where resources are shared. However, the federated approach is expensive, because each function has its own replicated resources, and the automotive industry is characterized by high-volume and low-cost products. So, recent applications are moving toward more integrated solutions in which some resources are shared across different functions to reduce hardware costs and make more efficient use of resources. The new danger is that faults may propagate from one function to another.

To address limitations imposed by the use of federated architectures, integrated approach merges nodes that were previously designed in isolation into systems where resources such as ECUs, communication channels, sensors, and actuators are shared between functions. This means that multiple functions can be supported by a single node and one function can be distributed over multiple nodes while sharing a limited set of communication resources. However, as opposed to federated architectures, an integrated approach lacks the isolation of nodes and their functions, and instead, promotes tight coupling of functions which leads to an increase in interactions between nodes. This complexity results in greater risks to the correct operation the system, particularly in the case where a resource or set of resources fail. A potential next step in this evolution of distributed architectures is one which capitalizes on the advantages of the federated approach while realizing functional integration and the reduction in electronic hardware of the integrated approach, such as distributed

control systems with autonomous decision making capabilities. Emerging standards, such as AUTOSAR [90], are proposing guidelines in the development of integrated architectures in automotive electronics.

As a consequence to the complexity of embedded systems that realizes safety-critical applications, the potential for failures that could lead to catastrophic loss increases. As pointed out by Charles Perrow [92] in an extensive analysis of on well-known disasters, he concludes that as our technologies become more complex from increasing interactions and tight coupling between system components (i.e. technical, human, and organizational), the chances of catastrophes due to system failures also increase. Hence, designers and developers must address these properties at the system level when such properties of interest are identified, addressed, and refined.

1.2.2 Requirements on Safety and Reliability

Safety and reliability are important quality attributes as embedded systems are increasingly realized by distributed architectures to replace mechanical and hydraulic subsystems. To clarify, reliability is an attribute of the system that involves the probability of failure, whereas safety is concerned with the *consequences* of failures. The automotive industry, which is characterized by low-cost and high-volume products serves as a primary example of the trend where mechanical subsystems that control critical functions are increasingly used in safety-critical applications. A safety-critical [57] application is one whose failure may result in a loss of life, severe monetary loss, or serious harm to the environment in which it operates. For example, in a modern passenger vehicle, complex software executes on a distributed architecture to realize safety functions and services that assists the end user in several active safety applications as identified by Isermann [46], such as steering assistance and lane departure [29]. Moreover, studies that are conducted in several countries, as summarized by Lie *et. al.* [72], demonstrate the effectiveness of electronic stability control in modern vehicles. The use of electronics in automotive control systems will continue to enable advanced functions that must operate without failure, such as steer-by-wire [50] and brake-by-wire [98] applications. Examples may also be found in markets that are characterized by low-volume and high-cost products in comparison to the automotive market such as avionics [120], autonomous spacecraft [23], and health care monitoring [47]. Since safety is achieved through the use of reliable components and processes, if not planned properly, an embedded system on a distributed architecture can decrease the overall reliability if a node fails and it causes a disruption to functions on other nodes. Design for reliability is the primary focus of this dissertation, whereas examples are extracted from safety-critical applications.

1.2.3 Time-to-Market and Design Productivity

Time-to-market is the window of time by which a product is conceived to the point at which the product is available on the market to the end-user. It is common knowledge in the market for consumer-driven electronic products that the greater the delay for a product to appear on the market, the opportunities to maximize profits are less and the chances to lose revenue are greater [109]. A 2008 VDC report [108] indicates that system integration, testing, and verification represents the greatest delay among engineering tasks for an embedded product. The report further shows that the primary causes are due to architecture design and specification (43.4%) and project management (44.7%) tasks. Furthermore, procedures that require additional design, test, and verification time are often required to ensure that the reliability requirements of an electronic product are met. Such procedures are specifically meant for certifying electronic systems across different industries where safety-critical applications are deployed. Some examples include, ISO26262 [8, 11] for automotive systems, DO-178B, and DO-254 for military systems. These procedures are used to structure design processes and flows throughout the life cycle of an electronic product, but require additional effort from system designers. Hence, these tasks are key factors that affect time-to-market for an embedded product.

Due to increasing complexity, time-to-market constraints, and increasing demand on reliability, the design of embedded systems is challenging. Across different industries that employ embedded systems, it is noted that the complexity of electronic systems is related to the design effort that is required to realize an embedded product. As illustrated in Figure 1.1 [44], design productivity is out paced by the increase in system complexity. The graph shows that the demand of software is currently doubling every 10 months, and the technology capabilities is doubling every 36 months due to technology scaling. On the other hand, the increase of hardware design productivity is well below the rate that technology is scaling, while the increase in productivity of hardware-dependent software increase is even slower by doubling every 5 years. Testing is often difficult, and testing embedded systems for safety-critical applications is particularly difficult owing to system complexity and requirements on reliability. The complexity of the problem grows much faster than the capabilities of existing design tools and methodologies. This is commonly referred to as the "design gap." As embedded systems become more complex, the challenge that is faced by system designers is to improve design productivity while satisfying reliability requirements..

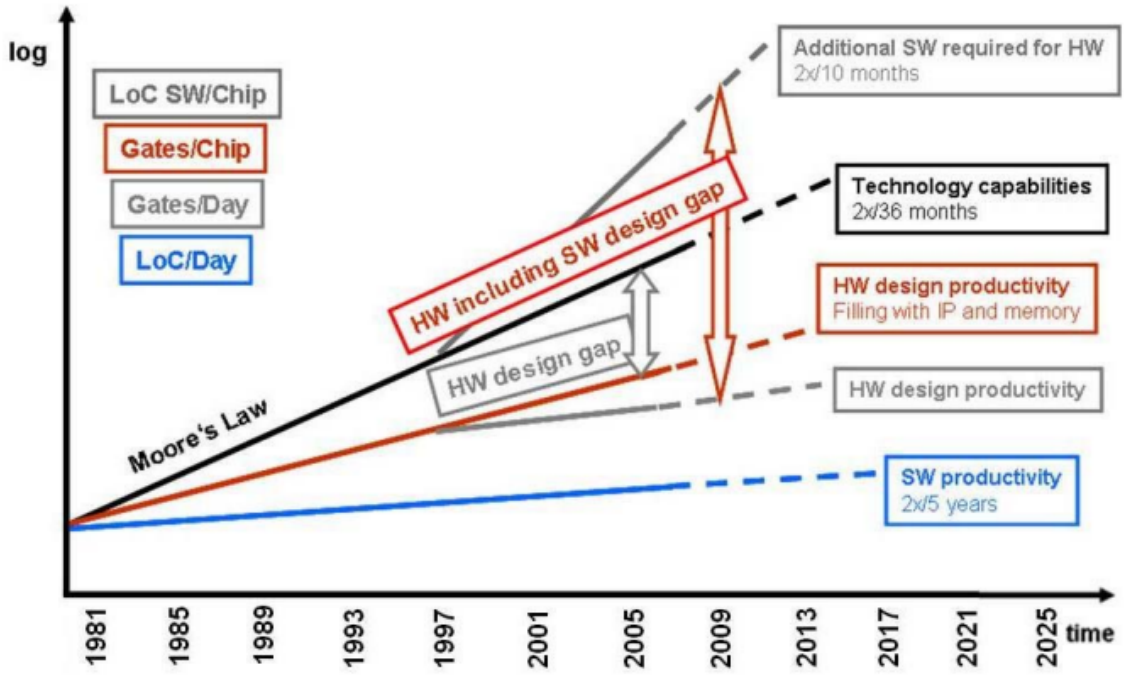

Figure 1.1. *An illustration of the design gap between the complexity of electronic hardware and software and design productivity. The graph plots the complexity of hardware and software components versus time.*

1.3 Electronic System Level Design

The challenges of designing complex electronic systems and improving design productivity under reliability requirements are primarily addressed by raising the level of abstraction at which the design is carried out and reusing existing components. By raising the abstraction of the design to a level that addresses the behavior and composition of components, unnecessary information is hidden from the designer who is left with a limited, and therefore manageable, set of choices to explore. Hence, a system can be composed out of fewer, yet more complex components where electronic components at lower levels are reused to support functions at a higher level. As a result, such an approach has the potential to increase design productivity since lower level components that have been verified can be reused to support a variety of higher level functions. In this work, the electronic system level will refer to the level of abstraction by which a distributed embedded system is designed. Thus, *system level design* takes into consideration the entire electronic-based system as opposed to only the individual components. The term "system level design" throughout this dissertation will refer to the design of electronic-based

systems. In system level design for electronics, the main properties of the system are captured in an abstract model which hides implementation details.

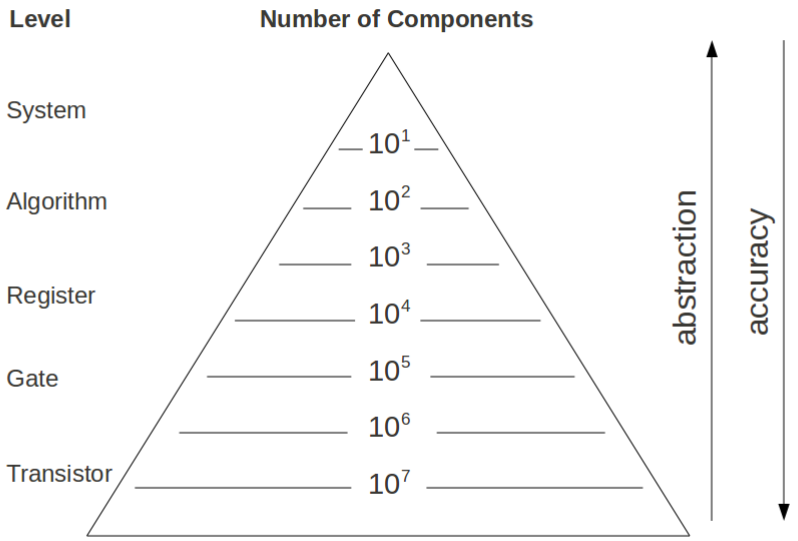

Figure 1.2. *An illustration of the relationship between abstraction levels and the number of components at each level.*

The relationship between abstraction level and the number of components in a System-on-Chip (SoC) design [35] is illustrated in Figure 1.2. An electronic system that is initially composed out of tens of millions of transistors may only require tens of thousands of RTL components. These in turn may be represented by multiple tens of algorithms where each algorithm is a sequence of operations, conditions, and loops that describes a function or set of functions that are provided by a microprocessor, or SoC implementation. By reducing the number of components to deal with at the same time, maintaining a system level view becomes easier. In contrast to the system level of a SoC design, communities such as automotive designers view the system level as a distributed embedded system that may be composed of multiple signal processors, micro-controllers, SoC devices, and mechanical parts such as sensors and actuators. The meaning of "system level" in this dissertation takes on the view of the latter.

A system level design approach emphasizes the use of models at throughout the design process from concept all the way down to an implementation. A design concept results in a *specification*, a formal or informal description of what the system does. Then, a system level design *methodology* defines a set of procedures by which the specification is realized, or implemented by lower level components. A system level approach to the design of distributed embedded systems allows the designer

to concentrate on the functionality, as defined in the specification, independently of the system's implementation early in the design process. A system model that is independent of its implementation naturally leads to abstraction, since lower level details have to be omitted. Abstract models of the system may be used to represent application behavior, architecture characteristics, and the relation between application and architecture. Then, these models can be evaluated to provide initial estimations on the performance, cost, or consumption of architecture resources. Moreover, modeling the system at a high level of abstraction minimizes the effort in model construction and speeds up simulation time. However, higher levels of abstraction also correspond to less accuracy between the model and the system's implementation of the specification.

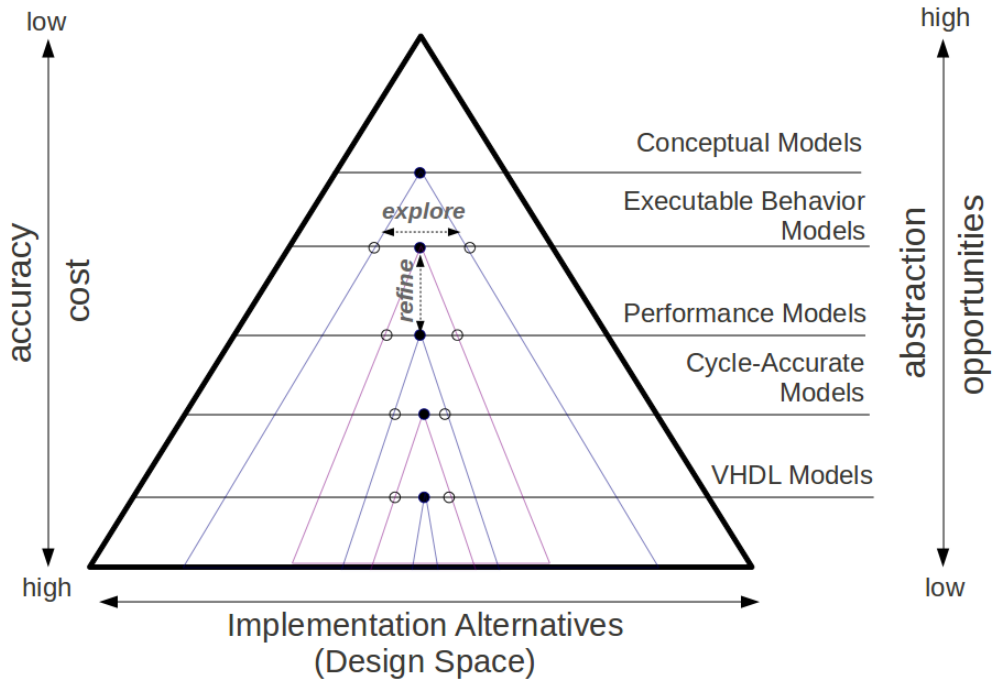

Figure 1.3. *The abstraction pyramid [55] illustrates the relationship between abstraction levels and modeling accuracy. Models at the top are more abstract and require relatively less effort to build, as models at the bottom incorporate more details and are more difficult to build while yielding less opportunity to explore alternative implementations.*

The abstraction pyramid [55] in Figure 1.3 shows different levels of abstraction for the design models. At a given level of abstraction, different solutions may be explored and one solution is chosen. This selected solution is refined and acts as the basis for potential solutions at a lower level of abstraction. Here, the refined design model allows different solutions at each abstraction level to be explored, but it also

excludes alternative solutions of the space of possible solutions, called the *design space*. To obtain the appropriate implementation, the solution of the higher level needs to be iteratively refined towards the lower levels by reducing the design space. The higher the levels of abstraction used for design entry, the more alternative realizations can be covered with the initial design model. The cost of the model construction and evaluation is higher at the more detailed levels of abstraction. However, the opportunities to explore alternative implementations is significantly reduced at these levels. Hence, methodologies for modeling, simulation and design of distributed embedded systems at the system level is of special interest.

1.4 State of the Art in Fault Tolerant Design

Designing an effective fault tolerant system requires a through and comprehensive analysis to fully understand, quantify potential failures, and assess the effectiveness of fault tolerant strategies. Fault tolerant strategies consist of some form of redundancy, but the choice of strategy and the amount of redundancy to apply is based designer experience and intuition. Reliability is a measure that is commonly used to evaluate the fault tolerance of the system, and there are well developed techniques to support the reliability evaluation of conventional systems. However, when applied in practice, reliability evaluation techniques can yield ambiguous or incomplete results as designers base the analysis on a qualitative description of the systems functionality. This conceptual description describes how components behave and subsystems of components are interconnected to fulfill the for which they are intended. The description may include how component failures can propagate to other components through their interconnections and affect the system functionality. For conventional systems, it is possible to evaluate how a system can fail to perform the function for which it was designed by using a high-level qualitative description of the functionality. Such descriptions are captured in combinatorial reliability models such as reliability block diagrams [104], fault trees [28, 104], and state-based models such as Markov models [106].

However, as systems become more complex, system designers are faced with the challenge of understanding system performance in the presence of failures early in the design process. Addressing fault tolerance earlier in design allows the designer to identify and explore the system's architecture. Thereby, avoiding weaknesses in the architecture design that are discovered late in the design cycle when the architecture has been committed. Moreover, ad-hoc approaches may lead to overly conservative designs that are expensive.

1.5 Related Work

The fault tolerant design of reliable systems is achieved by provision of additional components in parallel or by improving the reliability of components in the system. Kuo *et. al.* [60] provides a comprehensive survey of different approaches that address the design of reliable systems by formulating the problem either to maximize the system reliability under resource constraints or to minimize the total cost that satisfies the demand on system reliability. The approaches in the survey consider the reliability optimization of hardware or software components separately, and only a few approaches consider the design of systems with integrated hardware and software components [61]. Wattanapongsakorn *et. al.* [125] demonstrates the use of a Simulated Annealing algorithm towards maximizing the reliability under cost constraints and minimizing the cost of the system under reliability constraints for several models of fault tolerant architectures. The system model is a specification of subsystems that are in series, and for each subsystem, a discrete set of hardware and software components may be chosen. Each component choice has a known cost and reliability. Levitin [71] provides a heuristic algorithm that evaluates the reliability of a general fault tolerant system that contains hardware components and considers different software versions. The algorithm evaluates both the system reliability of a given hardware configuration and the execution times of software tasks. However, in the literature on reliability optimization of hardware and software systems, the algorithms do not consider heterogeneous set of hardware component choices or complex interactions between software components since the components are assumed to be in a series and parallel configurations. Therefore, less attention is given to more complex system configurations and their performance estimates.

Due to the complexity in embedded systems, the aforementioned approaches are limited. As a result, reliability optimization techniques for distributed embedded systems are integrated into system level design activities. Jhumka *et. al.* [48] describes an approach to determine an allocation of tasks to a distributed set of nodes such that dependability is the key attribute for determining alternative architecture solutions. In that work, dependability is defined by the failure probability, a measure of how critical a task is relative to other tasks in the system, and a factor that determines the amount of redundancy for a given task on a distributed network. The criticality and redundancy factors are parameters that are manually specified in an analytical reliability model. Yang *et. al.* [131] addresses the problem of determining a task allocation and hardware redundancy policy for distributed embedded systems using a Genetic Algorithm. In this work, the problem is to determine a minimal cost design under performance and reliability constraints. The resulting design, or deployment, is measured based on the cost of executing tasks, communication time between tasks, number of components used in the deployment, and risk. The risk cost is function of the number of components in the deployment and their failure rates that measures the amount of loss that is incurred upon system failure.

The amount of loss is a factor that is manually specified by the designer based on experience or field data regarding the cost a system failure. The related work discussed so far relies on the user to provide information on the criticality of a task to determine which task gets replicated.

The following related work considers the system topology to determine which tasks to replicate. Xie *et. al.* [130] describes a task allocation and scheduling algorithm that supports the duplication of critical tasks in the design of distributed embedded systems. A task is determined to be critical based on a heuristic that considers the distance of a task from the last task in a task graph describing the application of a system, the worst-case execution time of a task, and the start time for a task on an allocated processing unit. Lukasiewycz *et. al.* [75] exploits data redundancy as a way to improve the reliability in the design of distributed embedded systems using an approach that encodes information on signal transmissions into a reliability model based on Binary Decision Diagrams (BDDs). In that work, an algorithm for the design exploration based on data redundancy is implemented. Construction of the BDD is based on the data dependence between communicating tasks in their distributed system model. Thus, in that approach, the reliability model, as encoded in the BDD, is constructed based on the system configuration. The approach described by Papadopolous *et al.* [88] automatically generates reliability models from Mathworks Simulink models to evaluate fault tolerance in automotive systems. The reliability models that are created include a Failure Modes and Effects Analysis and a Fault Tree Analysis of the system under study. The topology of the system model along with annotations of failure modes for each component is used to create the reliability models.

In this work, a reliability model is generated based on the failure characteristics of components in the system model. Reliability estimates are obtained from an analysis of a generated fault tree from the system model, and the reliability evaluation is integrated into the design flow to measure the fault tolerance and reliability performance of the system design. The analysis drives the exploration of fault tolerant design solutions by replication of critical components, as determined by an analysis of the components whose failure contributes most to the reliability of the system.

1.6 Problem Statement and Contributions

The problem that is addressed in this work is the architecture design of fault tolerant systems that are deployed in distributed embedded systems. To address the challenges in the design of fault tolerant embedded systems, there is a need for a system level design methodology and a set of tools that support a structured and correct-by-construction approach that will enable the exploration of architecture

alternatives. A design methodology is proposed in this work along with a tool flow that supports this design methodology is implemented to support the system modeling, evaluation of fault tolerant and reliability requirements, and exploration of alternative architectures. The key enabler is the ability to generate and evaluate a fault tree of the system model quickly as compared to the manual construction of fault tree as is done in practice.

In particular, this dissertation contributes the following:

- A design flow that integrates a fault tolerant and reliability evaluation based on the analysis of fault trees.
- A fault tree construction method and a tool set that facilitates automatic fault tree generation and evaluation from a system model. The tool also enables the import and export of fault trees to and from existing fault tree analysis programs, in particular the FaultTree+ [43] tool that is developed by Isograph and and the Galileo [119] tool that is developed by the University of Virginia.
- An algorithm and a tool implementation that enables the allocation of application tasks to hardware resources and the evaluation of fault tolerant and reliability properties of the system model using fault tree analysis.
- A data model that used to capture and modify the architecture design along with quantities of each component in the system model, such as failure rate and component cost.

1.7 Organization of Dissertation

Th remainder of this dissertation is structured as follows:

- **Chapter 2** provides a background on redundancy techniques that are used in fault tolerant designs, and it provides an overview of methods for reliability modeling and evaluation;
- **Chapter 3** provides an overview of the proposed approach to the design of distributed embedded systems with an emphasis on fault tolerance and reliability;
- **Chapter 4** presents a method to automatically generate a fault tree from a given system model that is used in the design flow to speed up the fault tolerance and reliability evaluation of the system;

- **Chapter 5** describes an application of the proposed design flow to the design of electronic system architectures and a set of supporting tools that are used to implement the design flow; and
- **Chapter 6** concludes with remarks on the advantages and limitations in this work and a possible extension into future work.

Chapter 2

Fundamental Concepts of Fault Tolerance and Reliability Analysis

This chapter introduces some concepts in the design and analysis of fault tolerant systems. The coverage of this chapter begins with describing faults and their characterization. Then, some techniques that are used in the design of fault tolerant systems are discussed, including a brief overview of the most common forms of redundancy. The chapter concludes with a discussion on the mathematical background for the reliability modeling of a binary state system.

2.1 Faults, Errors, and Failures

The successful operation of a system is determined by the ability for the system to provide service correctly. Correct service is delivered when the system implements a specified function. A *fault* is an abnormal condition, or defect, in one or more components of a system that may result in the inability for a system to perform its intended function. The source of faults can vary from physical defects such as material defects caused by manufacturing processes to environmental disturbances, such as electromagnetic radiation to human factors, such as design defects resulting from incorrect specifications or organizational factors. A fault may or may not cause a system to deviate from its intended function.

An *error* is a discrepancy between the intended behavior of a system and its actual behavior inside the boundary of a component within the system. When

a fault is *activated*, it means that the fault has been observed or is active during system operation. Activation of a fault places the system component in an erroneous state. Thus, an error is an erroneous state of a system component that may lead to the system operating incorrectly, or *failure*. A system is said to have failed if it cannot deliver its intended function. Hence, the error must propagate to the system boundary before the system is in a failed state. A system consists of a set of interacting components, therefore the system state is the set of its component states. A fault originally causes an error within the state of one or more components, but system failure will not occur as long as the error does not reach the boundary of the system.

2.1.1 Fault Propagation

Failures are observable errors that occur at the system boundaries. Fault propagation is a key mechanism by which a fault propagates into a component or system failure. The activation of a fault into an error may propagate into a system failure, as observed by Laprie [63]. The fault propagation represents the causal relationship between a fault, error, and failure. Since the output data from one function may be fed into another function in a system, failure in one function may propagate to the input of another function as a fault. Therefore, a chain reaction can occur in multiple component systems that have many interacting parts.

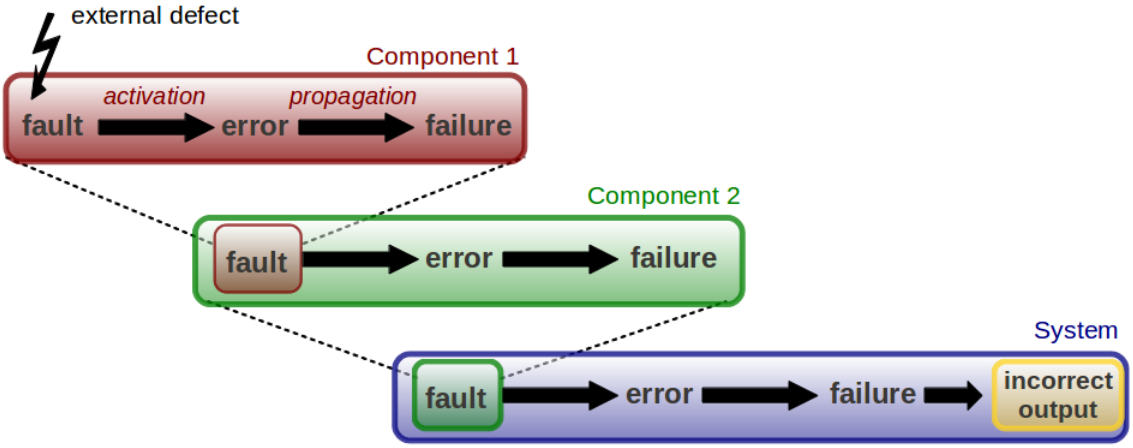

Figure 2.1. *An illustration of fault propagation in an example.*

Figure 2.1 illustrates this concept where arrows represent a causal relationship between a fault, error, and failure and its impact on two system components. In component 1, a fault is introduced into its boundaries. Activation of this initial fault

causes an error. The error then propagates into a failure of component 1. From this illustration, a failure in component 1 causes a fault in component 2. As a result, the fault in component 2 manifests into a system failure where the expected output is incorrect. Unless the chain reaction is countered, the result is a system wide failure. Hence, it is important to address this chain reaction throughout the life of the system at different levels of abstractions for the system. Examples to illustrate this concept follows:

Example: Failure of a Boolean Logic Gate

Suppose that two wires A and B are logical inputs (assumes A and B are logical values in the set $\{0, 1\}$ to a boolean logic gate. The expected output of the logic gate is 0 when A and B are both 1. Now, consider that a manufacturing process created a short (the fault) in wire A such that the value on wire A is always 1 (the error). The only case where the fault will result in an incorrect output value of 0 is when $A = 0$. Hence, an incorrect output of the logic function results in a failure of the logic function.

Example: Failure of a Software Function

Assume that a software programmer created the following code for computing the acceleration within the guidance system for an autonomous spacecraft according to the specification that $force = mass * acceleration$.

```
float ComputeAcceleration(float force, float mass) {
     return (force/mass) + mass;
}
```

The $force$ and $mass$ are input variables used to calculate the acceleration of the spacecraft. The programmer adds $mass$ to the acceleration calculation, hence, a software fault is introduced into the function with the summation of $mass$ as an additional term. As long as this snippet of code is never executed by the software program within the guidance system, then the fault is not activated. However, upon execution of this code, the code becomes active and computes an incorrect value for the acceleration. Hence, an error in the software has occurred. If there are no error detection mechanisms in place to catch such erroneous behavior of this function, then the error propagates into a failure of the guidance system. The failure of the guidance system, if not detected, can be propagated into a failure of the spacecraft to complete its mission.

2.1.2 Failure Modes

Failures may be characterized based on type and duration for any component in a system. A *failure mode* is a type of failure that can occur for a component within a system. It describes *how* a component may fail. For instance, an electrical circuit composed of a series combination of resistors and a power source may fail if the power source fails to provide an electrical energy to the circuit, a wire connection shorts a circuit component, or if the resistors were defective. In the circuit, each component failure is considered a failure mode because it describes how the component may fail to operate as intended. Furthermore, a failure mode that induces multiple components to fail simultaneously is a *common failure mode*. To understand a common mode of failure, the notion of a *fault containment region* [51] is introduced . A fault containment region is a collection of components that operates correctly regardless of any arbitrary fault outside the region. An arbitrary fault in a fault containment region cannot cause the hardware outside the region to misbehave or fail in any manner. Faults cannot propagate across containment regions but their effects can by way of error propagation. Therefore, a common mode failure is a component failure that affects multiple fault containment regions. As in the example of the electrical circuit, if the power source fails to produce any electrical energy, then the entire circuit fails to operate. Thus, failure of the power source is a common mode failure. However, if each component in the circuit has its own, power source, then the failure of the power source in the circuit does not affect the components that have independent power sources.

A fault or failure may be characterized by the time for which it persists in the system. A fault is said to be *permanent* if it continues to exist until it is repaired, such as a software defect, or bug. A *transient* fault is one that occurs and disappears at an unknown frequency, such as electromagnetic radiation that occurs and disappears. An *intermittent* fault is one that occurs and disappears at a frequency that can be characterized, such as a loose contact. For instance, a communication link may be in a failed state permanently, or in the case of a wireless link where there exists a momentary disruption, the link failure may be induced by a transient fault caused by the momentary disruption. Given that a system designer can place emphasis on any level of granularity of system components, it is most useful to consider failure modes that are observable and can be quantified by a frequency, or rate of failure.

Another way to classify faults is by their underlying cause. Design faults are the result of design failures, like the software coding example in Section 2.1.1. While it may appear that in a carefully designed system all such faults should be eliminated through fault prevention, this is usually not realistic in practice. For this reason, many fault tolerant systems are built with the assumption that design faults are inevitable, and that mechanisms need to be put in place to protect the system against them. Operational faults, on the other hand, are faults that occur during

the lifetime of the system and are invariably due to physical causes, such as processor failures.

2.1.3 Fault Models

In addressing faults, a method is needed to model their effects. A *fault model* is a logical abstraction that describes the effects on the function as a result of a fault. A fault model typically contains assumptions on boundaries of the system and components, the failure modes that are of interest, and the frequency of occurrence of failures. Fault modeling can be made at different levels of abstraction in a system. The lower the level of abstraction is more accurate in modeling the effects of a fault, but the more computationally intensive the method needed. The higher the level of abstraction, the less accurate and computationally expensive it is in representing the fault effects. As an example, the stuck-at fault model is a commonly used model for gate-level circuit testing. In comparison, transistor-level fault modeling assumes the stuck-open or stuck-short fault model. It is more accurate, but also more computationally expensive to use in circuit testing.

2.2 Fault Tolerant Design

Fault tolerance is defined as a property of a system that enable it to deliver the expected service in the presence of faults. As noted by Avižienis *et. al.* [3], there are two complementary approaches to addressing faults. The first approach, called *fault prevention*, attempts to ensure that the system will not and does not contain any faults during design, implementation, or operation. The two aspects of this approach are:

- **Fault avoidance.** The techniques in this category attempt to avoid introducing faults into the system. Examples that employs techniques are system design methodologies, quality control, and organizational management strategies.
- **Fault removal.** Techniques in this category are used to find and remove faults which were inadvertently introduced into the system, such as testing and analysis tools.

The second approach assumes that faults will occur because it is impractical to prevent the occurrence of all faults in a system, especially in a system with complex

interactions between electronic components. *Fault tolerance* is a property of the system that enables it to continue operating (potentially with degraded performance) in the presence of one or more faults, or component failures. Thus, fault tolerance employs techniques to tolerate faults. Hence, fault tolerance are characteristic of the mechanisms throughout the life cycle of a system that are employed to reduce the impact of faults that may propagate into system failure. Fault tolerant systems are systems which exhibit the ability to tolerate faults and component failures that may lead to system failure.

2.2.1 Approaches to Fault Tolerance

Designing an effective fault-tolerant system requires a through and comprehensive analysis to fully understand and quantify potential failures and assess the effectiveness of fault tolerant mechanisms. The common mechanisms as identified by Anderson and Lee [69] are: *error detection*, *damage assessment*, *error recovery*, and *fault treatment and continued service*.

- **Error detection.** In order to tolerate a fault, it must first be detected. Since internal states of components are not usually accessible, a fault cannot be detected directly, and hence, its manifestations, which cause the system to go into an erroneous state, must be detected. Thus the usual starting point for fault-tolerance techniques is the detection of errors.

- **Damage assessment.** Before any attempt can be made to deal with the detected error, it is usually necessary to assess the extent to which the system state has been damaged or corrupted. If the delay, identified as the latency interval of that fault, between the manifestation of a fault and the detection of its erroneous consequences is large, it is likely that the damage to the system state will be more extensive than if the latency interval were shorter.

- **Error recovery.** Following error detection and damage assessment, techniques for error recovery must be utilized in an attempt to obtain a normal error-free system state. In the absence of such an attempt (or if the attempt is not successful) a failure is likely to ensue. There are two fundamentally different kinds of recovery techniques. The backward recovery technique consists of discarding the current (corrupted) state in favor of an earlier state (naturally, mechanisms are needed to record and store system states). If the prior state recovered to, preceded the manifestation of the fault, then an error free state will have been obtained. In contrast a forward recovery technique involves making use of the current (corrupted) state to construct an error free state. An example of error recovery is *error masking*. This form of error recovery

works by allowing a number of identical modules execute the same functions, and their outputs are voted to remove errors created by a faulty component.

- **Fault treatment and continued service.** Once recovery has been undertaken, it is essential to ensure that the normal operation of the system will continue without the fault immediately manifesting itself once more. If the fault is believed to be transient, no special actions are necessary, otherwise, the fault must be removed from the system. The first aspect of fault treatment is to attempt to locate the fault; following this, steps can be taken to either repair the fault or to reconfigure the rest of the system to avoid the fault.

2.2.2 Redundancy Techniques

The concept of *redundancy* describes the existence of more than one means for performing a required function in an item [10]. Redundancy techniques are applied to increase the reliability of a system. Such techniques for achieving fault tolerance in embedded systems occurs in the form of hardware, software, time, or information.

Hardware Redundancy

Hardware redundancy refers to the provision of extra hardware components to either provide fault masking [93] or by automatically switching to a functioning component once a faulty component is detected. Fault masking is a means by which the effects of faults do not propagate to other components in the system by the presence of multiple components. A common approach to fault masking is to use identical components to perform the same computations at the same time. A representative example of fault masking is *N-Modular Redundancy*. Figure 2.2 illustrates a triple-modular redundant configuration. In this configuration, three computers compute the same outputs at the same time. The outputs are then passed through a circuit called a voter. The function of the voter circuit is to determine which output is in error when a majority vote is observed. This is a common system configuration in flight control systems [1].

Another general approach to hardware redundancy involves the removal or replacement of faulty components within the system in response to the detection of a fault or error. Such techniques are referred to as *dynamic hardware redundancy* techniques. When an error is detected in a component, a redundant spare component is selected automatically to replace the faulty component. At the time the spare is selected, then it becomes active in the system, unlike fault masking which requires all redundant components to be active in the system. Dynamic redundancy

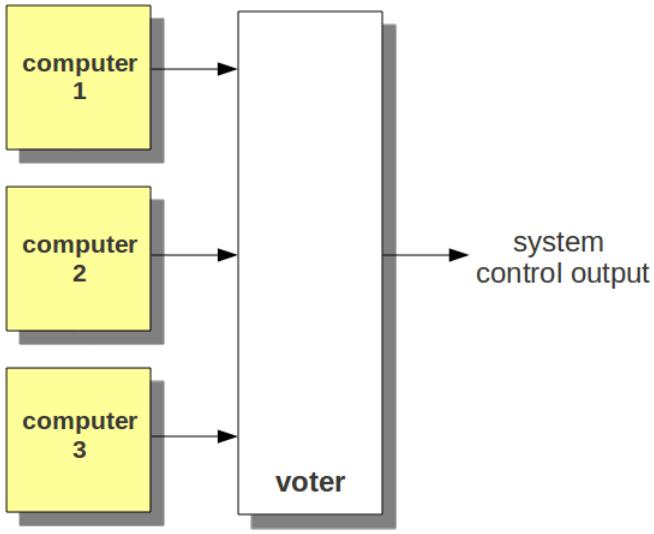

Figure 2.2. *An illustration of a triple-modular redundant system configuration.*

techniques in hardware can be used in combination with fault masking to provide fault tolerance in a system, as demonstrated in the work by Avižienis [2].

Software Redundancy

The redundancy of software refers to the use of additional software routines, code, or complete programs to check the correctness of software components. Similar to its hardware counterpart, *N-Version Programming* [4] is an approach that relies on multiple software components that are functionally equivalent. Multiple versions of a software task are independently developed by different designers from the same specification in a concept called *design diversity* [73]. The idea is that each of the versions of a software routine are executed simultaneously, and the results of each redundant component are voted upon to mask hardware and software faults.

The use of *recovery blocks* [103] is an effective software fault tolerance technique that is applied in embedded system design. This approach uses multiple components to perform the same task where each component is functionally equivalent. Redundant software components for a task are categorized as primary and secondary. The primary task executes first, and once its execution has completed, an acceptance or verification test is performed on its output. If the results of the primary component fails the test, the secondary component executes after rolling back to the state at which the primary task was invoked. This continues for all redundant components until an acceptable result is obtained or all redundant components have been exhausted. In contrast to N-Version Programming where the redundant components

are executed in parallel, the redundant software components in the use of recovery blocks are executed serially.

Software checking is a general method for checking the correctness of software routines. It is widely applied in the software design of fault tolerant and safety critical systems. This technique requires that computation results are known a priori, or that correct computations are within a given boundary or range of values. This can be accomplished by applying insertions at key points within software components. As an example, a check can be applied to determine if an invalid instruction code is executed. Another example of software checking is to compare measured performance of a software application with its predicted performance based on an accurate model. For embedded systems that are required to meet strict timing deadlines, a *watchdog timer* is often used to detect timing or omission errors that may result from a software or hardware component failure. A watchdog timer is a daemon process that checks if a component is actively working. The daemon process periodically sends a signal to the application and checks the return value. Lyu [77] provides a comprehensive summary of techniques in software fault tolerance, including software checking, recovery blocks, and N-Version Programming through design diversity.

Time Redundancy

Time redundancy refers to the repetition of a given computation in multiple instances at different points in time. Data redundancy, which is characterized by sending multiple instances of the same data at different times is a form of time redundancy in the software domain. Time redundancy is useful for detecting permanent, transient, and intermittent types of faults using hardware or software components. In embedded system design, time redundancy impacts the scheduling of software tasks, and as such, it is implemented in fault tolerant scheduling approaches. An example of time redundancy can be found in applications requiring fault detection in digital circuits. In Figure 2.3, bits of data are transmitted from a transmitter to a receiver at time t. At time $t + \delta t$, the complement of the original data is sent, and since the faulty line causes the data and its complement to be 1, then the faulty line can be detected. This method is called *alternating logic* [107], and it is useful in determining stuck-at-faults in digital circuits. It can be extended to larger systems by comparing the data that is sent at both instances of time.

Information Redundancy

This form of redundancy refers to the addition of extra information to allow fault detection, fault masking, or fault tolerance. Some examples of added information include error detecting codes, such as *Hamming Codes* and error correcting codes,

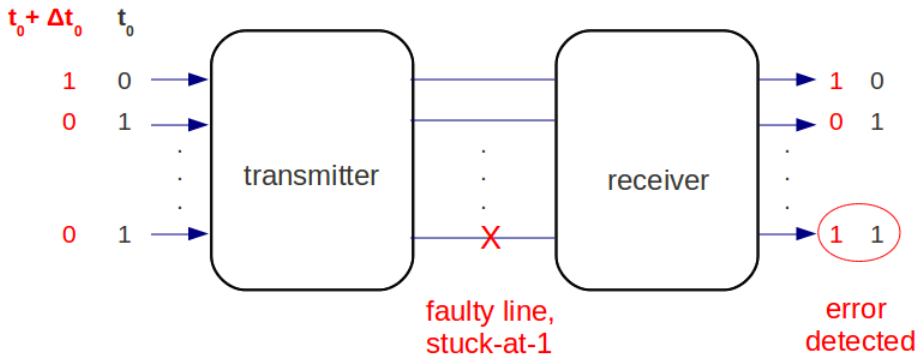

Figure 2.3. *An example illustrating time redundancy using alternating logic.*

such as *cyclic redundancy checks* [81]. One of the simplest forms of error detecting code is the *parity bit*. The parity bit is a bit of information that is added to a set of bits to ensure that the total number of bits with the value "1" in a set of bits is even or odd. Additional examples and details of error correcting and error detection codes can be found in Moon [81].

Since hardware redundancy is expensive, software redundancy techniques are more readily adopted in system design. Software implementation tends to be a less costly solution to redundancy because it usually consists of only adding additional lines of code at the expense of increased complexity, and it can be used to implement time and information redundancy techniques. Yet, in cost-sensitive safety critical systems and systems that require high reliability, it is common to find a mix of hardware and software approaches to fault tolerance. To determine the most effective method of fault tolerance to be employed in a system, it is necessary to measure the system reliability, which can be performed using a reliability analysis.

2.3 Reliability Analysis of Multi-Component Systems

The design of a fault tolerant system that is composed of multiple components aims at enabling the system to be reliable in the presence of faults. Failure of system components may manifest into system failures. A system is said to be *reliable* [87] if it can continue to provide a specified function correctly. Understanding that it is impractical to have a perfectly reliable system, by applying fault tolerant mechanisms,

the system can remain available to the user, possibly with reduced functionality and performance, rather than faults manifesting into a system failure. The Institute of Electrical and Electronic Engineers Standard Dictionary of Electrical and Electronics Terms defines *reliability* as the ability of an item to perform a required function under stated conditions for a stated period of time [101]. For a conventional system with no fault tolerance, it is natural to think that the system will deliver its function only if all the components are operational. This is not the case in a fault tolerant system, which may require either additional software, hardware, and human resources for design and test to deliver a function despite the presence of certain failures. Therefore, in a fault tolerant system, reliability will be dictated by the combinations of components at any time. Hence, reliability is one measure of the effectiveness of fault tolerance in systems.

2.3.1 Structure Function of a System

The *structure function* of a system relates the functional relationships among the components in the system to the functioning or non-functioning state of the system. A system is assumed to consist of n components and, without loss of generality, let $N = 1, 2, ..., n$ denote the set of components. It is assumed that the state of the system is given at a fixed moment in time, and the state of the system depends only on the state of its components. The system may also be in either a functioning state or failed state. The functioning state represents the state of the component by which it is working according to specification. Throughout this dissertation, it is assumed that each component can be in one of two states at any given instance in time, a functioning state or failed state. If a component is not functioning correctly or not functioning at all, then it is assumed that the component is in a failed state. To indicate the state of the i^{th} component, a binary indicator variable x_i, is assigned to component i with the following convention,

$$x_i = \begin{cases} 1, & \text{if } i \text{ is in a functioning state,} \\ 0, & \text{if } i \text{ is in a failed state.} \end{cases}$$

Let the state vector, $\mathbf{x} = (x_1, x_2, ..., x_n) \in \{0, 1\}^n$, describe the states of components in N. Given the system can only be in one of two states, the assumption that the state of the system is completely determined by the states of its components implies the existence of a Boolean function $\phi(\mathbf{x}) : \{0, 1\}^n \rightarrow \{0, 1\}$.

Definition 2.1 (**Structure Function**)**.** The structure function is a function $\phi(\mathbf{x})$ that relates the state of components of a system to the state of the system, and it has the following convention,

$$\phi(\mathbf{x}) = \begin{cases} 1, & \text{if system is in a functioning state,} \\ \\ 0, & \text{if system is in a failed state.} \end{cases}$$

The set $\{0,1\}^n$ is fitted with the operations addition $(+)$ and multiplication $(\cdot)$, and it has the common properties of associativity, commutativity, and distributivity. The "zero" element is denoted as $\mathbf{0} = (0,\ldots,0)$, the "unity" element as $\mathbf{1} = (1,1,\ldots,1)$, and the "inverse" element as $\bar{\mathbf{x}} = 1-\mathbf{x} = (1-x_1, 1-x_2, \ldots, 1-x_n)$. The operations addition and multiplication among elements of $\{0,1\}^n$ are carried out as follows:

$$\begin{aligned} x+y &= (x_1+y_1, x_2+y_2, \ldots, x_n+y_n) \\ x\cdot y &= (x_1\cdot y_1, x_2\cdot y_2, \ldots, x_n\cdot y_n). \end{aligned}$$

The set $\{0,1\}^n$ is partially ordered with the relationship "$\leq$". That is to say for state vectors $\mathbf{x}$ and $\mathbf{y}$, if $y_i \leq x_i$ for $i = 1,2,\ldots,n$ with $y_i < x_i$ for some i, then $\mathbf{y} \leq \mathbf{x}$.

In practice, it is commonly assumed that by replacing a failed component with a functioning component does not deteriorate the system given that the state of the other components remains the same. For example, replacing a failed component in a working system usually does not make the system fail. Mathematically, this assumption implies that the structure function $\phi(\mathbf{x})$ is an increasing function of $\mathbf{x}$. A structure function $\phi(\mathbf{x})$ is said to be *monotone* if the following conditions are met:

1. $\phi(\mathbf{0}) = 0$ and $\phi(\mathbf{1}) = 1$, and
2. for state vectors $\mathbf{x}$ and $\mathbf{y}$, if $\mathbf{x} \geq \mathbf{y}$, then $\phi(\mathbf{x}) \geq \phi(\mathbf{y})$.

This means that if the state of all components in the system are failed, then the system is failed, and similarly, if all components in the system are functioning correctly, then the system is functioning correctly. Since $\phi(\mathbf{x})$ is a monotonically increasing function, an improvement can be made on the state of the system if improvements are made on the state of its components. For the reliability analysis of systems, a component whose state that does not affect the state of the system will have no affect on the structure function or the system reliability. Thus, a component x_i is said to be *relevant* to the structure of a system if there exists at least one state vector $\mathbf{x}$ such that $\phi(\mathbf{x}) = x_i$ [6]. Otherwise, the component is said to be *irrelevant*. This means that when components are collectively in certain states, as specified by $(x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$, then $\phi(x_1, x_2, \ldots, x_n) = x_i$, and thus, the state of the system is dictated by the state of component i. The structure function for a given

system can be reduced to one with less components to consider by removing irrelevant components. Therefore, this dissertation will consider the structure function of coherent system structures. A system with structure function $\phi(\mathbf{x})$ is *coherent* if $\phi(\mathbf{x})$ is monotonic and every component is relevant [6].

The structure function is used to describe the state of a system in relation to the state of its components for any system. To illustrate, the structure function is used to define some fundamental structures, in particular, the series, parallel, and k-of-n structures. The k-of-n structure is a generalization of the series and parallel structures.

Example: Series Structure

A series system functions if and only if all of its components are functioning. Hence, its structure function is given by,

$$\begin{aligned} \phi(\mathbf{x}) &= \prod_{i=1}^{n} x_i \\ &= x_1 \cdot x_2 \ldots \cdot x_n \\ &= \min(x_1, x_2, \cdots, x_n). \end{aligned} \tag{2.1}$$

An illustration of a series structure is shown in Figure 2.4. The idea is that if a signal is initiated at point a then for the signal to reach point b, it must pass through all the components in the system. Hence, all components must be functioning correctly.

Figure 2.4. *A series structure.*

Example: Parallel Structure

A parallel system functions if and only if at least one its components are functioning. Hence, its structure function is given by,

$$\begin{aligned} \phi(\mathbf{x}) &= 1 - \prod_{i=1}^{n}(1 - x_i) \\ &= x_1 + x_2 + \cdots + x_n \\ &= \max(x_1, x_2, \cdots, x_n). \end{aligned} \tag{2.2}$$

An illustration of a parallel structure is shown in Figure 2.5. It follows that a signal at the input can reach the output if at least one component is functioning.

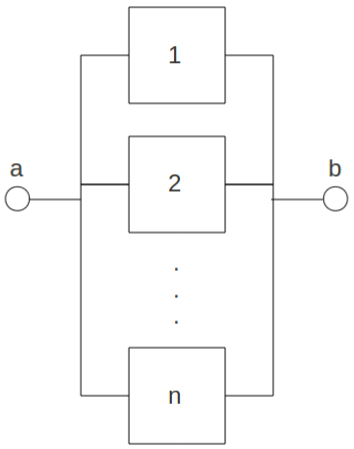

Figure 2.5. *A parallel structure.*

Example: The *k*-of-*n* Structure

A generalization of the series and parallel structures is the k-of-n structure. Such a system functions if and only if at least k of the n components are functioning. Hence, its structure function is given by,

$$\phi(\mathbf{x}) = \begin{cases} 1, & \sum_{i=1}^{n} x_i \geq k \\ 0, & \text{otherwise} \end{cases} \tag{2.3}$$

The series and parallel systems are special cases of the k-of-n structure. A series system can be represented by a n-of-n structure, and a parallel system can be represented by a *1*-of-n structure. An illustration of the 2-of-3 structure is shown in Figure 2.6.

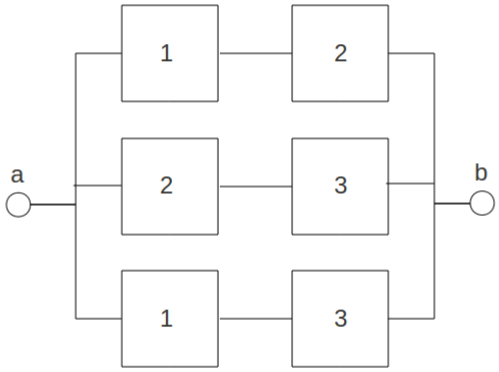

Figure 2.6. *A k-of-n structure where* $k = 2$ *and* $n = 3$.

Example: Series-Parallel Structure

A series-parallel structure is a system that contains a combination of components in series or parallel. Consider a system that contains four components. The system functions if and only if components 1 and 2 both are functioning and at least components 3 and 4 are functioning. Its structure function is given as,

$$\begin{aligned} \phi(\mathbf{x}) &= x_1 \cdot x_2 \cdot \max(x_3, x_4) \\ &= x_1 x_2 (1 - (1 - x_3)(1 - x_4)) \\ &= x_1 x_2 (x_3 + x_4 - x_3 x_4). \end{aligned} \tag{2.4}$$

An illustration of the series-parallel structure in Equation 2.4 is shown in Figure 2.7.

2.3.2 Minimal Path and Cut Sets

Minimal cut and path sets are introduced as a way to determine the structure function of any system as a series arrangement of parallel structures or as a parallel arrangement of series structures. They are particularly useful to determine the

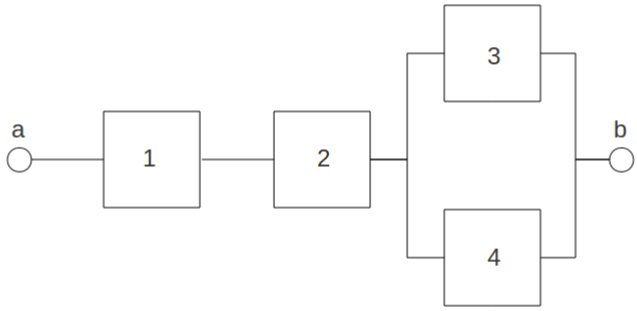

Figure 2.7. *A combined series-parallel structure.*

structure function of systems that may not be obtained directly from a k-of-n structure. Moreover, a coherent structure function can be expressed in a reduced form in terms of minimal path or cut sets by removing any irrelevant components in the structure function. As an example, Figure 2.8 illustrates a set of five components in a configuration such that it cannot be classified as having a series or parallel structure. Therefore, the concepts of minimal cut and path sets are introduced as a way to determine the structure function of any system.

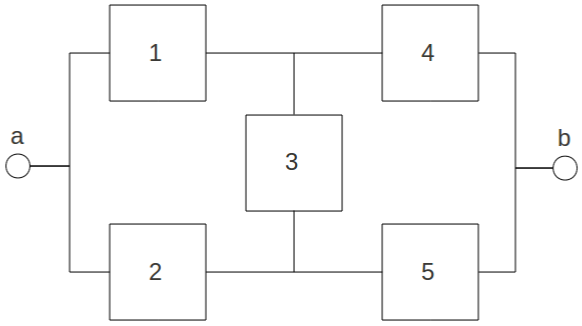

Figure 2.8. *A bridge system structure.*

A state vector $\mathbf{x}$ is called a *path vector* if $\phi(\mathbf{x}) = 1$, and $\mathbf{x}$ is called a *cut vector* if $\phi(\mathbf{x}) = 0$. If $\phi(\mathbf{y}) = 0$ for all $\mathbf{y} < \mathbf{x}$, then $\mathbf{x}$ is a *minimal path vector*. The state vector $\mathbf{x}$ is called a *minimal cut vector* if $\phi(\mathbf{y}) = 1$ for all $\mathbf{y} > \mathbf{x}$.

Definition 2.2 (**Minimal Path Set**)**.** For a state vector $\mathbf{x}$ with n components, if $\mathbf{x}$ is a minimal path vector, then the set $K = \{i : x_i = 1\}$, where $i = 1, 2, \ldots, n$ is a minimal path set.

Definition 2.3 (**Minimal Cut Set**)**.** For a state vector $\mathbf{x}$ with n components, if

$\mathbf{x}$ is a minimal cut vector, then the set $C = \{i : x_i = 0\}$, where $i = 1, 2, \ldots, n$ is a minimal cut set.

In other words, a path set is a set of components whose correct functioning at a given instance in time will simultaneously ensure that the system functions correctly independent of the states of the other components. Therefore, a minimal path set does not contain another path set. On the contrary, a cut set is a set of components whose failure to function correctly at a given instance will simultaneously ensure system failure independent of the states of the other components, and a minimal cut set does not contain any other cut set. To illustrate, the bridge configuration in Figure 2.8 is revisited.

Example: Minimal Cut Sets of the Bridge System Structure

The structure function of the bridge system configuration is obtained by determining the set of components whose simultaneous failure will result in system failure. Thus, the minimal cut sets are $x_1, x_2,\ x_1, x_3, x_5,\ x_2, x_3, x_4,$ and x_4, x_5. Therefore, the structure function is expressed as

$$\phi(\mathbf{x}) = x_1x_2 + x_1x_3x_5 + x_2x_3x_4 + x_4x_5. \tag{2.5}$$

An illustration of the structure function in 2.5 is shown in Figure 2.9.

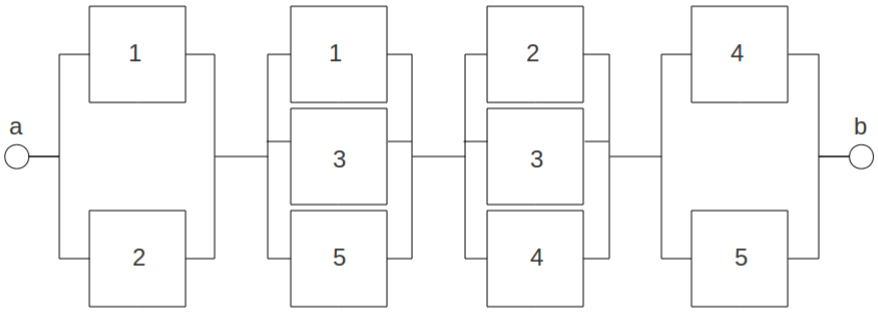

Figure 2.9. *Minimal cut set representation of the bridge system structure.*

The initial system configuration is equivalent to the system formed by its minimal paths in parallel, where each path is represented by a series system having as components the components in the path. Hence, the relationship between the

minimal paths and structure function is given as,

$$\phi(\mathbf{x}) = 1 - \prod_{j=1}^{k} \Big(1 - \prod_{i \in K_j} x_i\Big) \tag{2.6}$$

where k is the number of minimal paths in $K = (K_1, K_2, \ldots, K_k)$ of the system. In addition, the initial system is equivalent to the system formed by its minimal cut sets in series, where each cut set is represented by a parallel system having as components the components of the cut set. This relationship is given as,

$$\phi(\mathbf{x}) = \prod_{j=1}^{c} \Big[1 - \prod_{i \in C_j} (1 - x_i)\Big] \tag{2.7}$$

where c is the number of minimal cut sets in $C = (C_1, C_2, \ldots, C_c)$ of the system. Thus, the minimal path or cut sets are used to define the structure function of any system in a reduced form, where the reduced form is equivalent to the original structure of the system.

2.3.3 System Reliability

Reliability is the ability for a system to function as intended for a stated period of time. Since reliability can be characterized as a random phenomenon, there is a relationship that describes the occurrence of failures over time called the *failure distribution function*. Suppose that the time to failure T has the probability density function $f(t)$. Then, the failure distribution function $F(t)$ is the probability of an item failing in the time interval $0 < \tau \leq t$, and it is denoted as

$$F(t) = Pr(T \leq t) = \int_0^t f(\tau)\, d\tau \tag{2.8}$$

where $t \geq 0$ and $Pr(\cdot)$ denotes the probability. Hence, the *reliability function*, $R(t)$ is the probability of a component which does not fail in the time interval $0 \leq \tau \leq t$, and it is denoted as

$$R(t) = 1 - F(t) = Pr(T > t). \tag{2.9}$$

In practice, the failures of different components of a system are characterized by different failure distribution functions. Some common common failure distributions in reliability analysis include the Normal Distribution for modeling wear out failures

in mechanical components, the Exponential Distribution which is used to model the characteristics of other types of distributions and to fit a statistical distribution to failure data from a representative sample. As a result, it is of interest to to describe the reliability of a system in terms of the reliability of it components.

The reliability of a system with coherent structure is obtained from the reliability of its components. Assume that a component x_i in the state vector $\mathbf{x}$ for $i = 1, 2, \ldots, n$ is a binary random variable $\mathbf{X}_i$ of Bernoulli such that,

$$\mathbf{X}_i(t) = \begin{cases} 1, & \text{if } i \text{ is in a functioning with reliability } p_i, \\ 0, & \text{if } i \text{ is in a failed state with reliability } 1 - p_i \end{cases}$$

where $p_i = R_i(t)$ is the reliability of component i and $q_i = 1 - p_i$ is the unreliability of the component i. Note that the reliability of component i is in terms of its reliability function, hence, p_i is a function of t. Then, $\phi(\mathbf{X}(t)) = \phi(X_1(t), X_2(t), \ldots, X_n(t)) \in \{0, 1\}$ is also a binary random variable of Bernoulli. Since $\phi(\mathbf{x})$ is a function of the state vector $\mathbf{x}$, the system reliability $R_S(t)$ is defined as the probability that $\phi(\mathbf{x}) = 1$, and as such,

$$R_S(t) = Pr(\phi(\mathbf{X}(t) = 1) = E[\phi(\mathbf{X}(t)] \tag{2.10}$$

where $\mathbf{X}(\mathbf{t}) = (X_1(t), X_2(t), \ldots, X_n(t))$, $E[\cdot]$ denotes the expected value of a random variable, and the unreliability of the system is denoted by $Q_S(t) = 1 - R_S(t)$. As a result, the state of the system is a state of its components. The states of the components in the system may be stochastically independent or dependent on one another. When the components are independent, then the reliability of the system is a function of the reliability of its components denoted by $R_S(\mathbf{p})$, where $\mathbf{p} = (p_1, p_2, \ldots, p_n)$ is the component reliability vector. The reliability of the system reflects the relationship between system reliability and component reliabilities of each distinct system structure ϕ.

Kaufmann *et. al.* [52] shows that for a monotonic structure function, the reliability function of the system $R_S(t)$ is also monotonic. This means that for component reliability vectors $\mathbf{p}$ and $\mathbf{q}$, if $\mathbf{p} \geq \mathbf{q}$, then $R_S(\mathbf{p}) \geq R_S(\mathbf{q})$ which allows for a quantitative comparison of system structures in terms of their reliabilities. Since the calculation of the reliability of a system is known to be NP-hard [5], the method of minimal path and cut sets is commonly used to approximate the reliability of a system in terms of an upper and lower bound on the exact system reliability. In Equation 2.12 Barlow *et. al.* [7] shows that there exists a lower bound that is obtained through minimal cut sets and an upper bound that is obtained through

minimal path sets,

$$\prod_{j=1}^{c}\Big(1-\prod_{i\in C_j}(1-p_i)\Big)\leq R(\mathbf{p})\leq 1-\prod_{j=1}^{k}\Big(1-\prod_{i\in K_j}p_i\Big) \tag{2.11}$$

where $C=(C_1,C_2,\ldots,C_c)$ is the set of minimal cut sets for a coherent structure function ϕ with n components, and $K=(K_1,K_2,\ldots,K_k)$ is the set of minimal path sets for ϕ. In addition, the reliability of a coherent system is higher than that of a series system and lower than that of a parallel system, as expressed by the following trivial bounds,

$$\prod_{i=1}^{n}p_i\leq R(\mathbf{p})\leq 1-\prod_{i=1}^{n}(1-p_i). \tag{2.12}$$

The reliability of a system with coherent system structure is a function of the reliability of the components in the system. Therefore, an improvement in the system reliability will be directly influenced by the improvement of its components.

2.3.4 Measures of Component Importance

The concept of *component importance* describes ways by which the contribution of a component on the state of the system and the system reliability are measured. When assessing a system, its performance depends on that of its components. Component importance takes the form of a relative ranking among components in the system. Some components play a more important role in causing or contributing to system failure than others. Importance measures are used to numerically rank the contribution of each component to reflect the susceptibility of the system to the occurrence of the component's failure. The measurements of a component's importance are made relative to the other components in the system. Therefore, such measurements are useful to identify weaknesses in the system with respect to reliability, quantifying the impact of component failures, and establishing a direction and priority of actions that are used to improve the system reliability.

To evaluate the importance of different aspects for a system, a set of importance measures are well defined and widely adopted in engineering practice. Different importance measures are based on slightly different interpretations of component importance [9, 32]. For example, in one aspect of system design, a measure may be used to obtain the greatest gain in system reliability that results from improving the reliability of components that are most critical to the system reliability. The primary factors that influence component importance include the component's location in the system and the reliability of the component. When the component

reliabilities are given, then importance measures may be used to quantitatively assess the component importance. The Birnbaum Importance, Criticality Importance, and Fussell-Vesely Importance measures are considered.

Birnbaum Importance

In 1968, Birnbaum [9] introduces the concept of component importance, and one of the more fundamental importance measures is the Birnbaum Importance measure. This importance measure is analytically defined as,

$$I_i^B = \frac{\partial R_S(t)}{\partial x R_i(t)} = R_S(t; R_i = 1) - R_S(t; R_i(t) = 0) \tag{2.13}$$

where I_i^B is the Birnbaum Importance measure of component i, $R_S(t)$ is the system reliability at time t, $R_S(t; R_i(t) = 0)$ is the system reliability at time t given component i is failed, and $R_S(t; R_i(t) = 1)$ is the system reliability at time t given component i is perfectly working. This importance measure represents the maximum loss in system reliability when component i switches from the condition of perfect functioning to the condition of certain failure.

Criticality Importance

The Criticality Importance measure extends the Birnbaum Importance to account for component unreliability $F_i(t)$, with respect to the system unreliability $F_S(t)$. It is analytically defined as,

$$I_i^{CR} = I_i^B \frac{F_i(t)}{F_S(t)} \tag{2.14}$$

where $F_i(t) = 1 - R_i(t)$ is the component unreliability and $F_S(t) = 1 - R_S(t)$ is the system unreliability.

Fussell-Vesely Importance

The Fussell-Vesely Importance measure I_{FV} for a component i is given as,

$$I_i^{FV} = \frac{R_S(t) - R_S(t; R_i(t) = 0)}{R_S(t)}. \tag{2.15}$$

The Fussell-Vesely Importance measures the maximum decrease in system reliability that is caused by component i.

Additional reliability importance metrics may be obtained, and they are similarly defined such that the end result is a measure on the rate of increase of system reliability with respect to the reliability of each component in a system. In computing such measures, an analytical relationship between system reliability and component reliability is required, and that relationship is obtained from the structure function of the system. The structure function provides the mathematical basis for reliability models and their probabilistic evaluation to support the design of fault tolerant systems. This dissertation will utilize the fault tree as the reliability model to describe the structure function of a distributed embedded system and to assess the fault tolerance of the system.

2.4 Summary

The main objective of this chapter is to introduce the reader to the underlying concepts in fault tolerance and reliability analysis. In fulfilling this objective, fault tolerant techniques are introduced as methods for preventing the manifestation of component faults into system failures. In addition, system reliability analysis is presented in terms of the structure function, a mathematical relationship between the state of components in a multi-component system and the state of the system. The structure function is used to model the system behavior under the influence of component failures. With a probabilistic study of the structure function, quantitative measures of the system in terms of its components may be obtained.

Chapter 3

Design Methodology and Flow for the Fault Tolerant Design of Distributed Embedded Systems

This chapter introduces platform based design as a methodology which is applied to the proposed design flow that is described in this dissertation. The basic concepts of platform based design are introduced, and this is followed by an overview of the proposed design flow that supports the design of fault tolerant embedded systems. The main components of the design flow are described.

3.1 Design Methodologies for Embedded Systems

A *design methodology* describes a procedure, or design flow, that takes a design specification as input and produces an implementable system. Design methodologies for embedded systems have traditionally followed one of two approaches. In a *bottom-up* approach, the design of an electronic system began with simple components that are composed hierarchically into complex components as the design progresses through different levels of abstraction in an effort to support multiple applications. As the design proceeds to higher levels of abstraction, the components are stored in a library so that they may be reused in the next level of abstraction. The specification of a specific application is then satisfied by customizing the imple-

mentation. This requires making design choices by intuition and design experience. The problem that arises with this approach is that at the lower level of abstraction it is difficult to anticipate the needs of the higher level of abstraction. Hence, this approach results in over designed systems that may not meet the specification and issues with the integration of heterogeneous components [24]. Bottom-up approach to embedded system design in common in component-based design flows [13, 18, 25, 39].

In contrast to a bottom-up approach, a *top-down* begins with a system level view, and then proceeds to realize a detailed implementation. In this approach, the design specification begins at the system level, and it is then refined in a step-wise procedure to an implementation. With each refinement step, more implementation detail is added to the description of the system. Potentially after each refinement step, an analysis of the implementation that results from the prior design step is used to evaluate the effects of the implementation. The evaluation enables decisions on design parameters that may influence the next design step. By starting at the highest level of abstraction, fewer components have to be considered. Hence, this approach allows the use of views at higher levels of abstraction to enable early trade off analysis, accelerate verification, and reduce iterations that may occur in late stages of the design cycle. Moreover, the details of the system can be fined tuned at the expense of a larger set of design solutions, thus, resulting in an increase in time-to-market. Synthesis-based design flows, such as those found in custom integrated circuits designs [26, 113, 118] and hardware/software co-designs [36, 40] are examples of top-down methodologies.

Rather than a purely bottom-up or top-down approach to the design of embedded systems, a *meet-in-the-middle* approach draws from both approaches. The term, "meet-in-the-middle" is introduced in the design of electronic systems with the work by De Man *et. al.* [78] on the synthesis of multiprocessor chips for digital signal processing. In general, a design begins with a high level specification, and it is refined until a set of predefined components from a library can be instantiated. Each refinement step consists of selecting a valid composition of library elements that are characterized by their cost of usage and performance measures (i.e. monetary, communication, utilization, power consumption, etc.) that correctly implements a specification. Requirements are driven from the high levels of abstraction through the system hierarchy to improve productivity while an opposing force of feasible designs flow upward to ensure that higher level design decisions are implementable by a library of designed components whose function is verified.

3.2 Introduction to Platform Based Design

Platform based design [30, 114] is a design methodology that has been used in several application domains [15] to cope with the pressures of increasing demands on design productivity, reducing manufacturing costs, and meeting time-to-market requirements. Platform based design is based on the concept of *platforms*. A platform is an abstraction layer that hides the details of several possible implementation refinements of the underlying layers. The design methodology is a meet-in-the-middle approach that may be applied to all levels of abstraction in the design of embedded systems. A key principle of platform based design is the *orthogonalization of concerns*, i.e. the separating the various aspects of a design to allow more effective reuse of components and exploration of design solutions [54]. Ideally, any design aspects that are orthogonal to one another can be represented. Yang *et. al.* [132] and Lee *et. al.* [67] demonstrate the use of aspects in the design of system level frameworks for modeling heterogeneous systems. Additional examples of various design aspects in the context of embedded systems include *fault model and application behavior* in fault tolerant design of embedded control systems [96], *computation and communication* in communication-based design [54] and networked control systems [97], and *behavior and performance* [134] in the design of automotive architectures.

The two main design concerns emphasized in the Platform Based Design paradigm are *function* and *architecture*.

- **Function.** An abstract view of the behavior that the designers want the system to provide, i.e. "what the system is supposed to do," is described by the function of the system. The function layer does not express any of the implementation decisions. It is independent of the architecture and often an executable specification of the behavioral aspects of the system. In general, the function represents an upper layer, or higher level of abstraction level. The information that is needed to realize the behavior of the function is provided by the function layer as specifications. The specifications contain information about the function, such as the sequence of execution, deadlines to complete execution, priority, and amount of data that may be communicated within the function. Hence, the specifications are propagated towards a lower layer of of abstraction as constraints.

- **Architecture.** A configuration of resources or components that are capable of realizing the function, i.e. "how the function is implemented," is described by the architecture. It represents the lower layer, or a lower level of abstraction level. An architecture a set of components that may be abstract or characterized by physical quantities to support a function at the expense of some cost, i.e. time, power, or area. The components in an architecture can be viewed as being able to service different behaviors of the function. Therefore, the architecture captures the performance aspects of realizing a function. However, the

architecture does not specify which services or when they will be utilized. The concept of the architecture is that the services it provides will be consistent with the rules of execution and composition of components in the function. Hence, an estimation on the performance of the ability for the architecture to support a function is used to constrain the set of behaviors that can be realized by the architecture.

Mapping is the design activity that allows the function to be realized, or assigned to, the services that are provided by the architecture. A mapping represents a particular point in the design space of the system, and it defines the process by which the function and architecture meet, as illustrated in Figure 3.1. After a mapping is

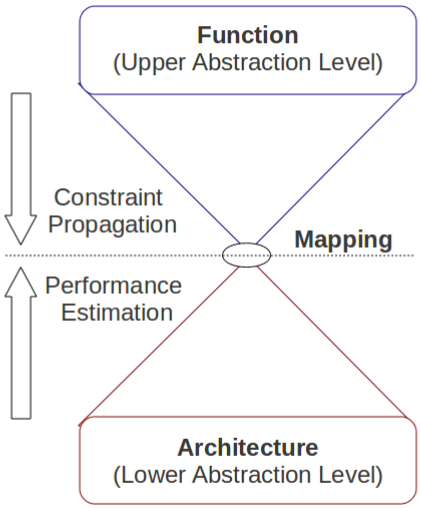

Figure 3.1. *An illustration of platform based design.*

complete, either manually or automatically with the aid of tools, the performance and cost of the system can be evaluated by estimating the cost and performance of library elements. Mapping is facilitated by formal descriptions in the form of design models that represent different aspects of the design methodology. Formal models, or *models of computations*, are abstract representations of a physical entity, such as the function and behavior of the system under study. In particular, such design models describe how a system design is specified, captured, and transformed into other model views or a more refined platform. As an example, a designer can model a system in Simulink [80] and use a code generation tool, such as TargetLink's dSPACE [121] or Real-Time Workshop [79] to map the functionality of the design onto a specific hardware platform. If a final implementation is available, then an exact performance and cost can be obtained. However, a system is often incomplete in early design stages. Hence, the performance and cost of a mapping are estimated by experience and intuition, simulation, or analytical methods. Based on the results of the evaluation, a procedure can be carried out to explore different points in the

design space, referred to as *design space exploration*. A method that is similar to platform based design is discussed in Kienhuis *et. al.* [56], and it features a Y-chart approach.

In this context, a *platform* is a library of components that can be assembled at a specific level of abstraction to define a set of solutions to a design. A platform contains all valid compositions of library elements, and each valid composition defines a design solution. A *platform instance* is a particular design solution. Each platform represents a layer in the design process at which the design space exploration is possible or desired. The result of mapping is a refinement of the original specification and of the platform instance, and the platform then plays the role of a new function at the lower level of abstraction. In the design process, each successive refinement step consists of selecting a platform instance that correctly implements a specification. The process continues until the abstraction level is close enough to the implementation.

The idea behind a platform based design methodology is to hide unnecessary details of the architecture so that the designer can focus on describing important parameters of the function at a higher level of abstraction, while limiting the design choices to a set of platform instances. This approach can be useful in the design of fault tolerant architectures for distributed embedded systems. The separation of function and architecture enables tasks and processes that define the function to be distributed across processing elements in a distributed network in a way that does not affect the function when the architecture components in the network change. This opens up the degrees of freedom by which the function and architecture may be modified to meet fault tolerant and reliability requirements. Platform based design limits the number of design choices to those that may be implemented by the platform. Hence, it enables more efficient design exploration as compared to a top-down approach, for instance. Moreover, this approach makes the evaluation of fault tolerant architectures possible with respect to a number of metrics, such as time, cost, power, and resource utilization.

3.3 Design Flow with Fault Tolerance and Reliability Evaluation

The design flow that is proposed in this dissertation is based on the Platform Based Design methodology, and it is illustrated in Figure 3.2. The main contribution of this design flow is the integration of reliability modeling and analysis for the performance evaluation of fault tolerance in a distributed embedded system. The proposed design flow begins with a specification of the system model using a specific

model of computation to define the functionality and architecture of the system's design. From this model, a mapping results in a platform model, or a deployment. The deployment is then used to construct a reliability model which is used to evaluate the reliability and fault tolerance of the system under design. Driven by a reliability and fault tolerance evaluation technique, this dissertation considers the replication of components in the FTDF graph and the mapping of function specification to a platform instance. This section provides an overview of the design flow in this dissertation.

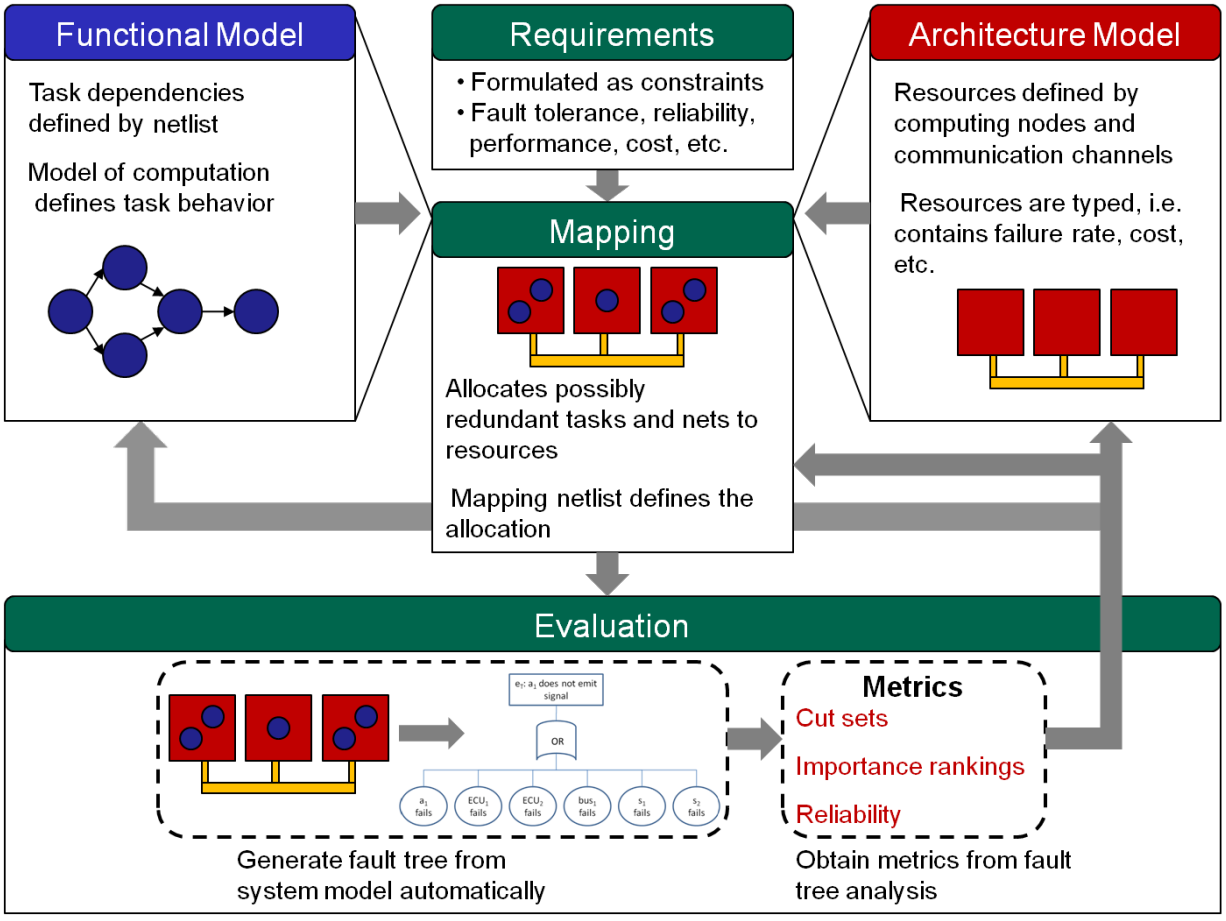

Figure 3.2. *The design flow for fault tolerance and reliability evaluation.*

3.3.1 System Requirements

Requirements on a distributed embedded system must satisfy a set of requirements that are needed to operate according to specification, including temporal, fault tolerance, reliability, and resource requirements. Requirements may be expressed by classifying them as design *objectives* or *constraints*. Design objectives are generally used to compare design alternatives by providing quantitative characteristics that expresses a level of compliance between a candidate design and the requirements. Constraints are requirements that must be satisfied in the design. As an example, requirements on timeliness of a design can be described as a constraint, yet, as a design matures and more information is provided to enable a more refined characterization of timing characteristics of the design may result in describing timing properties as design objectives.

A distributed embedded system may cover several types of requirements, including temporal, fault tolerance, reliability, and requirements on resource consumption. Timeliness requirements must be respected in a distributed embedded system to deliver a predictable and deterministic behavior in addition to the compliance to the functional specification. Temporal constraints are typically defined in classical scheduling theory in terms of *deadlines* to be maintained by the termination of execution of individual tasks and *precedence relations* which requires termination of a task prior to the execution of another task that has another task that causally depends on its results. Synchronous system implementations such as time-triggered architectures are frequently used to ensure that temporal requirements are met. In the case of fault tolerant design, often the number of replicas and redundancy are related to the reliability of the system. Such requirements imply that the result of a function in both time and value must be correct despite external perturbations that may induce faults. Hence, techniques need to be applied to achieve a desired level of system reliability. Another set of requirements involve the availability of resources. Some tasks can only be allocated to a subset of available computing nodes due to the need of certain resources, such as a sensor function that must be bound to a sensor resource. Such constraints are treated as binding requirements. Other factors such as utilization, are used to ensure that computing or communication resources are available. Requirements of the system are used to drive designs towards solutions that can be realized or refined into lower level constraints as the refinements in the design flow are made.

3.3.2 System Specification with Fault Tolerant Data Flow

The computational model that is chosen to describe the system is *Fault tolerant data flow* (FTDF). FTDF is introduced by Pinello *et. al.* [95] as a modeling formalism for the design of periodic, control applications. In the original work by

Pinello, FTDF networks are used for specifying fault tolerance in safety critical, real-time control systems. Its syntax and semantics enable formal analysis, synthesis techniques, and a structure which is used in this dissertation for constructing a reliability model for the system. This section will review the basic concepts of FTDF networks.

Signals

The terminology of the Tagged Signal Model (TSM) [68] is used to define FTDF networks more precisely in an actor-oriented framework [67]. The FTDF model of computation is constructed using the TSM by considering a set of values, $\mathcal{V}$ and a set of partially ordered tags, $\mathcal{T}$. The set of values represent the type of data that can be exchanged by objects in the model, whereas the set of tags carries an order relationship that is used in the model to represent a notion of precedence, such as time, causality, or execution order. A change in the set of values in the system is denoted by a set $\{(t, v)|t \in \mathcal{T}, v \in \mathcal{V}\}$, where v is the new value at "time" t. A *signal* s is a subset of $\{(t, v)|t \in \mathcal{T}, v \in \mathcal{V}\}$. Let the symbol $\perp$ denote the absence of a value, such that $\mathcal{V} = \mathcal{V} \cup \{\perp\}$. Since it is more useful to express the change in a value v at some t, this dissertation will limit the discussion to signals that are functional. A *functional signal* is a partial function $s : \mathcal{T} \rightarrow \mathcal{V}$. Thus, let $s(t) \in \mathcal{V}$ denote the value of signal s at tag t. From this point on, a "signal" will refer to a "functional signal". Signals are organized into tuples, hence, a *tuple* of N signals is denoted by $\mathbf{s}$, and the set of all such tuples is S^N. Let the set of input signals be S^I and the set of output signals be S^O, where for N signals, (S^I, S^O) is a partition of S^N. Hence, signals are considered to be partitioned into input or output signals.

Actors

Signals interact through *actors*, where an actor produces and receives signals on *ports* [74]. Each element in a tuple of signals corresponds to a port on an actor such that an actor A with N ports is a subset of S^N. A particular $\mathbf{s} \in S^N$ is said to satisfy the actor if $\mathbf{s} \in A$ is called a *behavior* of the actor. Thus, an actor asserts constraints on the signals at its ports, and the the possible behaviors of an actor can be observed at a set of ports. Ports are either inputs or outputs to an actor A. Let $A \subseteq S^N$ where $I \subseteq 1, 2, \ldots, N$ denotes the indices of the input ports, and $O \subseteq 1, 2, \ldots, N$ denotes the indices of the output ports such that $I \cup O = 1, 2, \ldots, N$ and $I \cap O =$. Given a tuple of signals $\mathbf{s} \in A$, let $\pi_I(\mathbf{s})$ define the projection of $\mathbf{s}$ on the input ports of A and $\pi_O(\mathbf{s})$ be the projection signals on the output ports of A. Then, an actor is said to be *functional* if

$$\forall \mathbf{s}, \mathbf{s}' \in A, \pi_I(\mathbf{s}) = \pi_I(\mathbf{s}') \Rightarrow \pi_O(\mathbf{s}) = \pi_O(\mathbf{s}').$$

An actor is a function from input signals to output signals. In particular, an *actor function* F_A is defined as,

$$F_A : S^N \rightarrow S^M, \tag{3.1}$$

where $N = |I|$ and $M = |O|$ denote the number of input and output ports respectively. It is common to express the exchange of data between actors using *tokens* in an actor-oriented framework. A token is a pair (t, v) where $t \in \mathcal{T}$ and $v \in \mathcal{V}$. Since a signal is the set of (t, v) pairs, this implies that a signal is a stream of tokens. Thus, a functional actor consumes and produces tokens, and tokens are passed between actors. For the remainder of this dissertation, the term "actor" will refer to a "functional actor".

Fault Tolerant Data Flow

FTDF is a variant of the data flow [65] MoC that models a system as periodic executions of actors. A sequence of actors in a FTDF model pass data in the form of tokens on every iteration according to the precedence order dictated by the data dependencies. This operation implies that the each actor is modeled as a sequence of atomic reactions, and the overall system model iterates through the reaction with a period, T_{max}. At each reaction, an actor presents a new set of tokens on its output ports, where the tag of a token indexes the reaction of an actor. In addition, a token within the FTDF MoC is appended with a *valid* field to record the boolean outcome of some fault detection algorithm (e.g. majority voting, checksum, CRC) since FTDF is designed to be fault model independent. Thus, a token is a pair (t, v) such that $t \in \mathbb{Z}$ is an integer representing the reaction of an actor, and $v = (data, valid)$ represents *data* of some type and a boolean-valued *valid* field.

In addition to the function of an actor, the behavior of an FTDF actor is also defined in terms of a *firing function* $f : S^N \rightarrow S^M$ and a *firing rule* $U \subset S^N$, for N-tuple of input signals and M-tuple of output signals. A firing rule is a guard condition that must be satisfied by a finite sequence of input tokens to the actor, and a firing function determines whether or not the condition is met. When considering a firing function f with the actor function F_A, the actor function is defined by $F_A(\mathbf{s}) = f(u).F_A(\mathbf{s}')$ if there exists a $\mathbf{u} \in U$ such that $\mathbf{s} = \mathbf{u}.\mathbf{s}'$ or $F_A(\mathbf{s})$ is the empty sequence [65], where $\mathbf{s}, \mathbf{s}' \in S^N$ is a finite set of tokens for each sequence in $\mathbf{s}$ and $\mathbf{s}'$, and "." is the concatenation of sequences. This implies that for a set of input tokens of an actor, the actor can *fire*, or an atomic execution of its function, only if the firing rule for the actor is satisfied. When an actor fires, it completes a reaction to the set of input tokens. For FTDF network, a repeated firing of all actors in the model defines the operation of the FTDF model.

As an example, suppose an actor has three input ports. Then a firing rule $U = \{(*, *, *)\}$ denotes the case that a token must have a value at each of the three

input ports to the actor for an iteration, where "*" denotes that a value for the token exists. This is an illustration of a typical firing rule for a data flow actor [66]. In contrast, suppose that an actor can fire if only two of three inputs have a value for their token. Then, the combinations of N input ports with tokens that satisfy the firing condition is given by each element in U, where each element is of size $N = 3$, is denoted by,

$$U = \{(*, *, *), (\perp, *, *), (*, \perp, *), (*, *, \perp)\},$$

where $\perp$ denotes the absence of a value for a token. In this case, the actor is said to have a *partial* firing rule, which means that the actor can fire on a subset of input tokens whose values are present. This is how FTDF models fault tolerance, since there may be the case that an input token does not have a value because a predecessor actor failed to emit an output token.

Actors in FTDF are typed, and as such, they are classified by their behavior on ports. A *sensor* actor, $a \in A_S$, reads data from a sensor device, as an *actuator* actor, $a \in A_C$, updates an actuating device with data, where A_S is the set of *sensor* actors and A_C is the set of *actuator* actors. These actors interact with the plant in a feedback control application. An *input* actor, $a \in A_I$, performs a merge function, such as sensor fusion or a majority voting strategy, which takes a collection of input tokens and produce a single output token. An *output* actor, $a \in A_O$, is designed to balance the load on *actuator* actors. A *task* actor, $a \in A_T$, is responsible for the computation workload, as an *arbiter* actor, $a \in A_R$, is used to select input tokens to present to the output of the actor. Finally, a *memory*, $a \in A_M$, is an actor that stores tokens at tag t, for use in the following period, tag $t + 1$. Memory actors serve as unit delay actors since other types of actors are stateless and their code is executed as a result of an atomic reaction. Actors in $\{A_S \cup A_C \cup A_O \cup A_T \cup A_M\}$ require all inputs to fire, and *input* and *arbiter* actors may use partial firing rules. Therefore, the set of actors that may be used in a FTDF model is a partitioned set of typed actors, such that

$$A = A_S \cup A_C \cup A_I \cup A_O \cup A_T \cup A_R \cup A_M.$$

Actors communicate via objects called *communication media*. A *media*, $m \in M$, is an actor that is used to merge input tokens and distribute the input tokens to possibly many actors, where M is the set of *media* in a network of FTDF actors.

3.3.3 Functional Model

The functional model is a formal description of the application tasks that are to be executed and supported in the system. Each actor in the FTDF model of

computation is an application task that computes, sends, and receives data to and from other actors through communication tasks, which are described by the communication medium. With these basic elements of FTDF, a definition for a network of FTDF actors is given as a graph where each node in the graph is either an application task or communication medium. This graph is the specification for the functional model of the system.

Definition 3.1 (FTDF Graph [95]). Given a set of actors A and communication media M, a FTDF graph $\mathcal{G}$ is a pair (V, E) where $V = A \cup M$ is the set of vertices and $E \subset (A \times M) \cup (M \times A)$ is the set of directed edges.

A FTDF graph $\mathcal{G}$ is bipartite, and actors A are connected via a communication medium M. The data dependencies of $\mathcal{G}$ determine the order in which actors may fire and data communication may occur. Since edges are directed, $e = (m, a) \in E$ denotes that an actor $a \in A$ receives a token from a medium $m \in M$ on its input port, and $e = (a, m) \in E$ denotes that an actor $a \in A$ sends a token from its output port to the input port of a medium $m \in M$. A FTDF graph $G = (V, E)$ is said to be *legal* if,

1. for a FTDF graph $\mathcal{G}' = (V', E')$, where $V' = V - A_M$ and $E' = E \cap (V' \times V')$ is acyclic,
2. $\forall v \in A_I$, $pred(v) \subset A_S \cup A_M$ and $\forall v \in A_S$, $succ(v) \subset A_I$,
3. $\forall v \in A_C$, $pred(v) \subset A_O$ and $\forall v \in A_O$, $succ(v) \subset A_C \cup A_M$,
4. $\forall v \in A_S$, $\{v^- | v^- \in pred(v)\} = \emptyset$ and $\forall v \in A_C$, $\{v^+ | v^+ \in succ(v)\} = \emptyset$,

where an edge $(e_i, e_j) \in E$, denotes that the edge is directed from e_i to e_j and e_j is the *successor* of e_i denoted by $succ(e_i) = e_j$ and e_i is the *predecessor* of e_j denoted by $pred(e_j) = e_i$. These conditions mean that a FTDF graph contains no causality cycles, and any loops that are in the graph must pass through a *memory* actor; the results of *sensor* actors must always be combined using *input* actors, and *actuator* actors must always be driven by *output* actors; and every iteration begins with a *sensor* actor and ends with a *actuator* actor, and if *memory* actors are present in the graph, then their execution also begins at the beginning of an iteration along with the *sensor* actors. In addition, an actor ca

Finally, FTDF graphs can express redundancy, i.e. one or more actors may be replicated. This implies that any two actors in $\mathcal{R}(v)$ are of the same type and must compute the same function. Replicas of an actor $v \in A$ are denoted by $\mathcal{R}(v) \subset A$, such that $\mathcal{R}(v) \neq v$. When a FTDF graph contains a set of replicated actors, the graph is called a *redundant FTDF graph*. The properties listed above are recognized in the construction of the reliability model for a redundant FTDF graph. The properties above are not exhaustive, and the reader is referred to Pinello [94] for more information on additional properties for FTDF graphs.

3.3.4 Architecture Model

The architecture model is a set of computing and communication resources that provide a set of services to the application tasks that are described in the functional model. The architecture model can be represented by a library of parametric components that may be characterized by physical quantities. The resources represent physical components, such as ECUs, buses, and wires, that are available to the designer for use in a system. The physical components may be obtained from a library of off-the-shelf components. The architecture model is specified by a connected graph of processing nodes (i.e. the ECUs) and communication channels (i.e. the buses and wires). The graph that describes an instance of the architecture is an *architecture graph*, $\mathcal{PG}$.

Definition 3.2 (**Architecture Graph**)**.** An architecture graph is a bipartite graph $\mathcal{PG} = (R_{\mathcal{PG}}, E_{\mathcal{PG}})$ where the vertices $R_{\mathcal{PG}} = \{P \cup C\}$ in $\mathcal{PG}$ represents resources, the processing nodes P, and communication channels C. The set of edges $E_{\mathcal{PG}} \subseteq (P \cup C) \times (C \cup P)$ are undirected. Each edge $e \in E_{\mathcal{PG}}$ connects a vertex in P to one in C such that the edges in $E_{\mathcal{PG}}$ induce a symmetric binary relation $\sim$ on $R_{\mathcal{PG}}$. This means that for each edge $(r_1, r_2) \in E_{\mathcal{PG}}$ the vertices r_1 and r_2 are said to be adjacent to one another, which is denoted by $r_1 \sim r_2$.

The processing nodes of an architecture graph will be referred to as $ECUs$ and the communication channels are referred to as *channels*, such as buses or copper wire. Note that the architecture graph does not dictate data dependencies between functions. The data dependencies between function tasks are dictated by the functional model. Each resource $r \in R_{\mathcal{PG}}$ is characterized by a set of parameters $Q = \{q_1, q_2, \ldots, q_N\}$ where N is the number of parameters that are of interest to the designer. These parameters are inherent characteristics of the resource, such as the size of memory, bandwidth, transmission rate, the monetary cost, and a failure distribution. The parameters distinguish different types of resources that are available in a library of resources. In practice, the library may represent the set of off-the-shelf resources that are available to the designer or a set of virtual components that acts as placeholders until resources with more accurate performance and cost measures are available.

3.3.5 Fault Model

In the design of fault tolerant systems, the designer must be able to design against faults that may occur in the system. A *fault model* is an abstraction of the type of faults that may occur in a particular system. This requires assumptions on the type, frequency, and effects of faults on system components. For system level models of distributed embedded systems, the types of failures can be of type omission, value, or timing failures. The faults that may occur for a component

in a distributed system are described by a set of *failure modes*, where each failure mode for a component represents a faulty state of the component. For example, in a distributed system, an ECU may crash due to any number of reasons, including being subjected to excessive radiation. A crash failure is a permanent omission failure that results in the ECU to no longer perform as specific. Suppose that the event that the ECU crashes is given by the random event A and the event that the excessive radiation was the cause of the crash is given by random event B. Then, the event that the ECU crashes due to excessive radiation is given by the random event $E = A/B$. This random occurrence is called a *fault event*. In this case, the fault event E is a failure mode of the system.

Components in the platform model, including actors, signals, ECUs, and channels, can have multiple failure modes. A failure mode for a component forces the analyst or designer to concentrate on the expected or potential failures that can occur with each component in the system; hence, targeting the fault model towards specific component failures. If a system exhibits one or more failure modes, then ideally the fault tolerant design of the system would prevent the effects of the occurrence of those failure modes from propagating into failure of the system. For this reason and for reasons of expressing failure modes in the reliability model, a fault model is described by a set failure modes, where a failure mode for a component in the function and architecture models is a relation between the component and a fault event.

3.3.6 Replication and Mapping of the System Specification

In the design of fault tolerant embedded systems, employing redundancy strategies through replication leads to additional costs on the system. So, it is important to consider which components in the system should be subjected to additional redundancy. Once the components are selected for replication, it is then the goal of the mapping process to determine how to assign the redundant components to the architecture resources. The mapping process in embedded system design consists of a selection of architecture resources (allocation) and the assignment of components from the functional model to the resources in the architecture model in space (binding) and time (scheduling).

Replication

To fulfill the reliability and fault tolerant requirements, active redundancy of software components is applied. The application tasks are assumed to be time-triggered, periodic tasks. Using software redundancy implies that some tasks in the functional model may be replicated, and when tasks are replicated, the semantics

of the function that is computed by the tasks must be preserved. This means that the data dependency between tasks should be consistent when tasks are replicated. In this dissertation, the application tasks are replicated according to the relative contribution of component failure to system failure as measured by the reliability importance factor of the component. To do this, the application is modeled with a redundant FTDF graph, and then it is mapped onto the architecture platform under constraints that are imposed by the architecture components. This procedure results in a procedure to replicate tasks, bind them onto a selection of resources, and ensure that the tasks are scheduled under any specified resource constraints. Determining the relative importance of components in the platform model is done by the construction and analysis of a reliability model.

Mapping

Distributed embedded systems for safety critical systems are heterogeneous in nature, and they must often compute application tasks correctly with limited physical resources. Their design at the system level early in the design process entails the consideration of a set of constraints on the application requirements and resource consumption. More specifically, the constraints are defined as the conditions that limit the possible designs from an assignment of application tasks to resource components for which the system has to satisfy to ensure the correct behavior once it executes on a set of resources. Constraints that flow from a previous refinement step are also refined into a set of constraints for subsequent refinement steps. By following this strategy throughout the design, beyond the system level, ideally the requirements on the system flow down to implementation as they appear as refined constraints of the system requirements at lower levels of abstraction.

In this design flow, the mapping is defined as a relation between the system functional model, described by a FTDF graph $\mathcal{G}$ that may contain a set of redundant actors, and the architecture model, as represented by an architecture graph $\mathcal{PG}$. The platform model is a model that represents the result of the mapping process, and it defines a platform instance. It can be used for different purposes and analysis by hiding unnecessary details and exporting only the necessary amount of information. Given a set of control algorithms specified as a possibly redundant FTDF graph $\mathcal{G}$ and an architecture graph $\mathcal{PG}$, a fault tolerant deployment is a platform model that can be defined as another graph to model the redundant allocation of actors and communication media onto the platform model.

Definition 3.3 (**Platform Graph**)**.** A mapping of $\mathcal{G}$ onto $\mathcal{PG}$ is a directed graph $\mathcal{L} = (L_V, L_E)$ where $L_V = (R_{PG} \times A_G)$ is the set of vertices and L_E is the set of edges. In L_V, the resources $R_{\mathcal{PG}} = \{P \cup C\}$ denotes a set of ECUs P and the set of channels C in the platform graph $\mathcal{PG}$, and $A_{\mathcal{G}} = (A \cup M)$, denotes the set of actors A and the set of media M in the FTDF graph $\mathcal{G}$. Actors and media in A can be

replicated. A vertex $v \in L_V$ with $v = (r, a)$ for $r \in R_{\mathcal{PG}}$, $a \in A_{\mathcal{G}}$ means that an actor or medium a is allocated to resource r. An edge $e \in L_E$ with $e = (v_1, v_2)$, $v_1 = (r_1, a_1)$, and $v_2 = (r_2, a_2)$ connects v_1 to v_2 for $v_1, v_2 \in L_V$. When replicas are introduced into the platform graph, the data dependencies must be preserved between communicating actors as specified in the FTDF graph, $\mathcal{G}$.

Example: Mapping of FTDF Actors onto Architecture Resources

To illustrate the mapping of actors in a FTDF graph onto a set of architecture resources in an architecture graph, consider the following:

1. Two actors $a1, a2 \in A_{\mathcal{G}}$ are mapped to the same ECU $p_1 \in P$ from the architecture graph $\mathcal{PG}$, and the actor a_1 sends a token to a_2 such that no channel from $\mathcal{PG}$ is involved, then $v_1 = (p_1, a_1)$ and $v_2 = (p_1, a_2)$ where $v_1, v_2 \in L_V$ are nodes in the system architecture graph $\mathcal{L}$.

2. One actor $a \in A_{\mathcal{G}}$ is mapped to an ECU $p \in P$, and it transmits data on a channel $c \in C$ of the architecture graph $\mathcal{PG}$, then $v_1 = (p_1, a_1)$, $v_2 = (c_1, m_1)$, and $(v_1, v_2) \in L_E$ of the platform graph $\mathcal{L}$.

In Case 1, since two actors are mapped to the same ECU, no explicit channel is necessary because the implicit assumption is that actors that are located on the same ECU will exchange tokens using shared memory on the ECU. In Case 2, only one actor is mapped to an ECU and it transmits tokens across the boundaries of the ECU for which it is mapped. The *media* actor that receives tokens from the sending actor must also be mapped to a resource from the library of resources. This implies distributed communication, where the token must be transmitted across a channel to its destination. These two cases in the example reflect the previous two cases of actor-to-actor communication on a same ECU through memory and the dependency between an actor and a communication medium.

3.4 Reliability Modeling and Analysis with Fault Trees

Reliability modeling and analysis using fault trees presents a view of the system as a combination of component failures that lead to system failure. The purpose is to obtain insight as to how a multi-component system may fail and to quantify the possible failure or fault events that lead to the system failure. Reliability analysis

with fault trees consist of constructing the fault tree model, obtaining its minimal cut sets, and analyzing the fault tree model by quantifying the occurrence of fault events. In this section, a more precise definition of a fault tree is presented along with methods for its analysis.

3.4.1 Fault Tree Graph

A fault tree is a graphical representation of the systems structure function in the form of a fault-oriented logical diagram [28, 104]. The fault tree is more generally described as a graphical representation of the logical combination of component failures, or fault events, that result in system failure. In this dissertation the terms "fault tree graph" and "fault tree" will be used interchangeably. The fault tree graph is defined below as an interconnection of fault events.

Let $\phi(\mathbf{x})$ be the structure function of a coherent system with N components and state vector $\mathbf{x} = (x_1, x_2, \ldots, x_N) \in \{0,1\}^N$, and let $B = \{+, \cdot\}$ be the set of boolean operators that denote the addition and multiplication operators of $\{0,1\}^N$ respectively. $e_T = \phi(\mathbf{x})$ such that $\phi(\mathbf{x}) = 1$, and the set of *basic events* is defined as, $E_B = \{e_i : x_i \in 0, 1, \forall i = 1, 2, \ldots, N\}$. In words, the top event is the failure of the system and a basic event is the failure of a system component. If the state of the system component can be represented by multiple failure modes, then each basic event is a failure mode for that component. Similarly, the state x_i for component i can be represented by multiple state variables. The set of *intermediate events* is defined as $E_G = \{g(f(E_s)) : E_s \subseteq E_B \text{ and } g(f(E_s)) \neq e_T\}$. Intermediate events are functions of basic events where $(g, f) : E_B \rightarrow E_G$, and the outcome of an intermediate event is determined by its operator from the set B. Although a top event is the function of basic events, it is uniquely defined, and so it is not considered to be in the set of intermediate events.

Definition 3.4 (**Fault Tree Graph**)**.** A fault tree graph is a directed acyclic graph $F_G = (E, A)$, where

- $E = \{\{e_T\} \cup E_G \cup E_B\}$ is the set of vertices;
- $A = \{(e_i, e_j) | e_i, e_j \in E\}$ is the set of edges such that an edge is considered to be directed from e_i to e_j where e_j is the *successor* of e_i denoted by $succ(e_i) = e_j$ and e_i is the *predecessor* of e_j denoted by $pred(e_j) = e_i$;
- $\forall (v_i, v_j) \in A$, $\exists v_r$ that can reach v_i and v_j by a path;
- the fault event e_T is a distinguished node called the *root node* of F_G which can reach all other nodes in E by a unique *path*, where a path in F_G is defined as a sequence of q nodes $(v_0, v_1, \ldots, v_q)$ such that $(v_i, v_{i-1}) \in A$, $v_0 \neq v_q$, and no vertex is repeated (i.e. no cycles);

- the $|pre(e_T)| = 0$ for the root node e_T, where the notation $|\cdot|$ denotes the number of elements;
- $\{v|v \in E_B\}$, the $|succ(v)| = 0$; and
- $\{v|v \in \{\{e_T\} \cup E_G\}$, the $succ(v) \in E_G$ or $succ(v) \in E_B$, i.e. every intermediate event and top event must have another intermediate event or basic event as a successor.

A fault tree graph is a connected graph of fault events. A fault tree graph shares the same properties as a coherent structure function with the addition that fault events are monoform, i.e. basic events do not exist in complemented form. So, a fault tree graph can be evaluated just as is done with the evaluation of the structure function for a coherent system in terms of the reliability estimation of system components and a topology for the system.

3.4.2 Methods for the Obtaining Minimal Cut Sets

Analysis of a fault tree is the process of computing minimal cut sets and making a probabilistic assessment of the fault tree with the intent to obtain qualitative and quantitative measures of the system that the fault tree models. The approaches to computing the minimal cut sets from a given fault tree are based on boolean reduction techniques. This section will discuss algorithms and methods for finding the minimal cut sets of a fault tree. The techniques for evaluation using probabilistic methods, as discussed in Chapter 2, are also discussed.

Direct Method of Boolean Reduction

Reducing a fault tree to its minimal cut set can be performed directly by applying boolean reduction rules. It is assumed that the fault tree represents a coherent structure function of the system, as is discussed in Chapter 2, where the basic events of the fault tree can be either in a functioning or failed state. The relationship between the state vector of a structure function and basic events of a fault tree is that a basic event denotes the state of a component. If the component has multiple failure modes, then each failure mode constitutes a basic event in the fault tree.

Let F_G be a fault tree that is composed of a top event e_T, a set of gates E_G that represent intermediate fault events, and a set of basic events E_B where $y_T = e_T$ is a variable that represents the gate of the top event, $y_j \in E_G$ for $j = 1, 2, \ldots, N$ are variables for N intermediate gates, and $x_i \in E_B$ for $i = 1, 2, \ldots, M$ are variables for M basic events of fault tree F_G. The minimal cut set is obtained by applying the

following reductions,

$$\begin{aligned} x_i^n &= x, && (3.2)\\ nx_i &= x_i, && (3.3)\\ x_i + x_i \cdot y_j &= x_i, && (3.4) \end{aligned}$$

where $n \in \mathbb{N}$ and "+" and "$\cdot$" are boolean operators for *OR* and *AND* gates respectively. The *AND* operator symbol "$\cdot$" will be omitted and implicit for convenience. An example of applying boolean reduction to compute the minimal cut sets of a fault tree F_G is given in the example below on "Obtaining Minimal Cut Sets using Boolean Reduction", given the notation above.

Example: Obtaining Minimal Cut Sets using Boolean Reduction

Suppose that a fault tree, F_G, is given as a function of gates, and its decomposition into basic events yields,

$$\begin{aligned} F_G &= y_1 \\ &= y_2 y_3 \\ &= (x_1 + y_4)(x_2 + y_5) \\ &= (x_1 + x_3 + x_4)(x_2 + x_3 + x_4). \end{aligned} \tag{3.5}$$

The fault tree is decomposed into its basic events. The fault tree F_G can now be expanded into a sum of products expression,

$$F_G = x_1x_2 + x_1x_3 + x_1x_4 + x_3x_2 + x_3x_3 + x_3x_4 + x_4x_2 + x_4x_3 + x_4x_4.$$

By applying the boolean reduction in Equation 3.2 yields,

$$F_G = x_1x_2 + x_3 + x_4. \tag{3.6}$$

Observe that Equation 3.6 can no longer be reduced by boolean reduction. Therefore, each term in the equation is a cut set K:

$$\begin{aligned} K_1 &= x_3, \\ K_2 &= x_4, \\ K_3 &= x_1, x_2. \end{aligned}$$

It is worth noting that the minimal path sets are obtained the same way as minimal cut sets, but by proceeding with the *dual* of the fault tree. The dual of a fault tree is obtained from the fault tree by replacing the *AND* gates with the *OR* gates, the *OR* gates with the *AND* gates, and the basic events with their complements. As an example, the dual of the fault tree F_G from Equation 3.6 is

$$\overline{F_G} = \overline{x_1 x_3 x_4} + \overline{x_2 x_3 x_4}. \tag{3.7}$$

MOCUS: Method for Obtaining Cut Sets

One of the more commonly used methods for obtaining minimal cut sets is the MOCUS algorithm by Fussell and Vesely [33]. MOCUS is representative of a class of top down approaches to obtaining minimal cut sets for a fault tree. The algorithm that starts with the gate for the top level event and proceeds to decompose each higher level gate until only basic events are obtained. The algorithm assumes that the fault tree is given with only *AND* and *OR* operators.

Although MOCUS is widely used as the core of many computer codes for fault tree analysis, very little is given about its implementation. The algorithm begins by defining two sets, each to hold a term in a sum of products form. The terms could be in terms of a gate or basic event. One set, P, holds the sum of product terms that are terminal (composed of only basic events) and the other set, R, holds terms that contains gates that are to be further decomposed. The algorithm works as follows:

1. Initialize R with the top event e_T: $R = e_T$
2. Initialize P to be empty: $P =$
3. For each product $\pi \in R$,
4. If π is terminal, then move π to P
5. Else, a gate variable is selected in π,
   decomposed into additional product terms,
   insert product terms into R
6. Repeat until all variables are terminal (basic events).

The MOCUS algorithm requires two heuristics to do the following:

1. to select the next product term in R to process and

2. to select the gate to decompose and expand to product terms during the process.

Therefore, several methods and algorithms attempts to improve on the basic algorithm by employing heuristics and introducing different data structures.

MICSUP: Minimal Cut Sets Upward

The MICSUP algorithm, proposed by Chatterjee [21], is a representative of the class of algorithms that starts with the primary gates (gates with basic events only) and replace higher level gates with their inputs, which ultimately become a function of only basic events. This procedure continues "up" the fault tree until the top event is composed of a primary gate containing only basic events. The algorithm of MICSUP works as follows:

1. Initialize a set of primary gates: $S = allprimarygates$
2. While S contains primary gates,
3. Select a primary gate and replace with its input events
4. Apply boolean reductions
5. Replace the product terms in S
6. Apply boolean reduction to primary gate of top event
7. S results in only basic events

One of the challenges to this method is that the development of large fault trees result in having AND operators near the top, which yields quite a considerable number of Boolean terms. To address this challenge, Nakashima and Hattori [83] proposed the algorithm Ancheck.

3.4.3 Quantitative Evaluation of a Fault Tree

A quantitative analysis of a fault tree consists of a calculating the probability of the top event by starting with the probability of the basic events, as discussed in Chapter 2. In addition, importance measures can be carried out quantitatively once a fault tree constructed. This can be done directly by traversing the fault tree from its basic events back up to the top event and making the appropriate probabilistic calculations depending on whether or not an AND or OR gate is visited during the

traversal. On the other hand, if basic events are repeated (share common gates), then the minimal cut set are used to perform a quantitative analysis. This requires the reduction of the fault tree into its minimal cut sets, and then using minimal cut sets to analyze the fault tree quantitatively. The most common approaches uses the minimal cut set method to quantify a given fault tree. This method is based on the inclusion-exclusion theorem, and thus, it is not exact. Its wide use is contributed to the fact that it is a faster calculation than exact approaches and it yields results that are adequate for its use.

Recent research into the probabilistic assessment of fault trees focuses on the efficient use of a Binary Decision Diagram (BDD) [105] as data structures to perform exact probabilistic calculations for a fault tree. The BDD method does not analyze the fault tree directly, but converts the tree to a binary decision diagram, which represents the logical relationships for the top event. The difficulty, however, is in the conversion of the fault tree into a BDD. An ordering of the basic events in a fault tree must be chosen and this ordering can have a crucial effect on the size of the resulting BDD. The ordering can mean the difference between two extremes: a BDD with few nodes that provides an efficient analysis and the inability to produce a BDD at all. There is no universal ordering scheme that can be successfully used to produce a BDD for all fault trees, and no scheme has been found that will produce a BDD for some large fault trees. The effects of a large BDD can increase the computational time that is required to obtain probabilistic measures, and the size can have an adverse impact on the amount of computational memory utilized in the tool. Recent research on the use of BDDs in fault tree analysis is focused on applying alternative techniques that will increase the likelihood of obtaining a BDD for any given fault tree and ensuring that the probabilistic calculations are as efficient as possible.

3.5 Summary

This chapter provides an overview of the design methodology and its flow in the context of this dissertation. Platform based design is introduced as the underlying design methodology that supports this design flow. The design flow is composed of a system model, which specifies models of its functionality, architecture, and a system platform model that is the result of mapping. Fault tolerant data flow is described as a computational model that is used to model the functionality of an application within this framework, whereas the architecture model consists of a set of parameterized components that are selected from a library. Finally, the fault tree is introduced as the reliability model that enables this design flow such that reliability and fault tolerance of a system model can be analyzed and evaluated.

Chapter 4

Automating the Art of Fault Tree Construction

One of the first steps to performing a reliability analysis using fault trees is to construct a fault tree from a model of the system under design. This chapter focuses on the systematic construction of a fault tree from a system model and techniques for its analysis. A systematic approach to fault tree construction enables the automatic construction and analysis.

4.1 Background

The construction of a fault tree requires an in depth knowledge about the system that is under study. The system could represent a physical artifact, a human and organizational procedure, or process. In the context of this dissertation, the system will be a physical artifact, in particular, a system with electronic components. System analysts and designers must understand the structure, behavior, and the expected failure modes of the system. This understanding is acquired through experience and knowledge of a specific system, its components, or a similar system. There is no precise method for constructing a fault tree. However, a highly referenced publication by Vesely *et. al.* for the United States Nuclear Regulatory Committee [122] provides a general procedure to guide the construction of fault trees.

4.1.1 General Procedure for Constructing Fault Trees

Before construction begins, a preliminary analysis is performed to define and identify components in the system and their failure modes. The preliminary analysis occurs prior to fault tree construction. Identifying the system means that the system boundary must be specified. The system boundary dictates the input and output signals of the system, and it defines how the system interacts with its environment. Once the system is defined, it is decomposed into its components. The components are entities in the system that are not decomposed any further. For each component that is not decomposed any further, a set of failure modes are associated with those components. The failure modes represent the ways by which a component is said to have failed. After the preliminary analysis, the construction of a fault tree begins by defining a top event for the system; resolving this event into intermediate fault events until resolving them is judged to be impossible or useless according to the level of fidelity that is desired; and designating the failure modes of a component as a basic event. A failure modes and effects analysis (FMEA), is one such procedure for for identifying and analyzing the consequences of potential failure modes within a system. Subsequently, the general approach to systematically construct a fault tree is given in the following steps:

1. *Define the top event.* This means determining the undesirable event. The undesirable event is the top event of the system that may be resolved.

2. *Resolve the fault events.* This step requires that the top event is resolved in terms of intermediate events that immediately contribute to a top event and the resolving of the latter into the next level of intermediate events.

3. *End construction.* The construction is completed once the resolved events there are no more events to resolve, leaving only basic events.

The construction of a fault tree is an important step in the analysis of systems using fault trees. The resulting fault tree of the system is used to perform qualitative and quantitative analysis and an evaluation of those results so that corrective measures can be taken to address unsatisfactory measures. In practice, fault trees are constructed manually from design specification documents and intuition by highly skilled and experienced subject-matter experts since it requires expert knowledge about the system. Due to increased system complexity, constructing fault trees manually is difficult, time-consuming, and inconsistencies may arise the fault tree and the intended behavior of the system it models. For these reasons, analysts have been led to conceive systematic techniques to construct fault trees with a view to automate its steps.

4.1.2 Approaches to Automatic Fault Tree Construction

Approaches to automatic fault tree construction varies in terms of the type of input data structure used to systematically generate the fault tree and the way failure modes are attributed to the system components. In many cases, the input data structures used for generating the tree are typically not the system design models. Many design models do not contain enough information, such as failure modes and error propagation, to support the fault tree construction. Instead, such information appears in reports that are used to build data structures from which fault trees are generated.

In traditional approaches, labeled directed graphs in Lapp and Powers [62], Bosche [12], Ju *et. al.* [49] are constructed to explicitly describe the cause-and-effect relationships between process variables and its environment. The graphs are constructed manually from design documents and knowledge of the system behavior, and as such, they were represented by separate models. These graphs are then transformed into a fault tree. In a method developed by Fussell [34], each component in the schematic of an electrical system has a small fault tree that is used to embed failure modes of the component. The system fault tree is composed from the individual fault trees of the components. Failure modes are described by a small set of discrete values, and the intermediate data structures that describe the fault trees and their failure modes can become complex. The result is that the data structures for manipulating the fault trees and failure modes are not easy to build, reuse, and keep consistent with the system design models since changes in system behavior and requirements means that the fault tree must change accordingly. Such changes are made by hand.

Subsequent approaches began to focus on integrating enough detail in the models that are used for designing and simulating the system, for example into block diagram schematics and behavioral models. These models describe the structural dependencies and the behavior of components in the system, and each of the two models were built independent of one another. Moreover, none of these methods considered applications for embedded systems. De Vries [27] develops an approach for quantifying failure modes in analog electrical circuits. In contrast, methods such as the ones reported in Vemuri *et. al.* [59] and Papadopoulos *et. al.* [89] address digital systems where each component in the system model is annotated with a small set of discrete failure modes. In these approaches, a parameter of a component is qualitatively labeled as or , as it represents a discrete failure mode. A chronology of traditional approaches to automatic fault tree construction is provided in Carpignano *et. al.* [16]. The main differences in systematic and automated approaches include the type of model that is used to describe the function of the system and the faulty behavior of the system, the degree at which the analyst can reason about the system, and expressiveness of the resulting fault tree is when compared to the system from which it is derived.

More recent approaches [59, 89] have focused on the automatic construction of fault trees for safety critical systems. The work most closely related to this paper is Papadopolous *et. al.* [89], where a methodology is presented that enables fault tree construction based on a composition of both a hazard analysis on the architectural and functional components of the system. Having the composition of the two enables where each component is annotated with failure modes, the system level fault tree is constructed automatically. In Vemuri *et. al.* [59], focus is given on how to create a descriptive modeling language called RIDL, which stands for the Reliability Embedded Design Language. The purpose of RIDL is to use a framework with related syntax and semantics to describe fault trees and their construction. This gives a natural modeling environment for reliability engineers to work, and it also removes the ambiguity in interpretation of component reliability requirements. Primitives such as single point failure components, support component, repeated components are available in the environment to encode a variety of systems from a reliability perspective. Since these primitives in the modeling language have precise semantics, a fault tree can be constructed systematically. By applying a systematic technique to fault tree construction, the fault tree may be generated automatically to support the design flow that is presented in Chapter 3 and enable faster evaluations of architecture designs for the system model as compared to manual, ad-hoc methods.

4.2 Automatic Fault Tree Construction of Fault Tolerant Data Flow Models

In this section, an algorithm for automatic fault tree construction from a redundant, mapped FTDF graph is formulated and described. The resulting fault tree is static, i.e. contains only *AND*, *OR*, and *K-of-N* logic operations. It is represented by a series of Boolean relationships. The fault tree describes how faults in the execution platform may lead to faults in the functionality and ultimately to violations of the specifications, i.e. to system failures. A recursive algorithm operates on the deployed FTDF graph model to produce a system fault tree. The problem is defined more precisely below.

4.2.1 Assumptions

The system specification is modeled by a platform graph, given as $\mathcal{L} = (L_V, L_E)$, where $L_V = (P \cup C) \times V$ is the set of vertices and L_E is the set of edges. The

graph $\mathcal{L}$ contains a set of redundant actors that describes a periodic, time-triggered application. In L_V, P is a set of ECUs, C is a set of communication channels, and V is the set of actors and media that describes the application, as defined in Section 3.3.6. The unmapped application is represented by the FTDF graph $\mathcal{G}$, and the set $V \subseteq G$. The set $P \cup C \subseteq \mathcal{PG}$, where $\mathcal{PG}$ is the platform graph that models the set of architecture resources. A vertex $l \in L_V$ with $l = (r, v)$ means that an actor or medium v is mapped to resource r. An edge $e \in L_E$ with $e = (l_1, l_2)$, $l_1 = (r_1, v_1)$, and $l_2 = (r_2, v_2)$ connects l_1 to l_2. Graph $\mathcal{L}$ preserves the data dependencies between its components as specified in the functional model of the unmapped FTDF graph of $\mathcal{G}$. The replication strategy ensures that the data dependencies are preserved and that the application tasks execute correctly, i.e. without fault.

The fault model assumes fail silence on nodes in $\mathcal{L}$. Fail silence is a fault that results in permanent omission failure of the ECUs and communication channels in the platform graph. Thus, the failure mode is omission failure. It is assumed that software components have the ability to detect value and timing faults. This can be accomplished by an internal detection mechanism that checks or asserts conditions on the value of input data. Since the application is assumed to be time-triggered, then timeouts are used to detect timing failures. Hence, the application tasks fail when the ECU fails to execute or when communication channels fail to transmit messages.

The fault tree is described by a fault tree graph $\mathcal{F} = (E, A)$, as defined in Definition 3.4, where E is the set of fault events and A is the set of directed edges that connect vertices. In the fault tree generation method, it is useful to partition the set E into the set of basic events E_B and the remaining intermediate events E_G such that $E_G \cup E_B \subseteq E$. The top event represents the failure of the actuators in the system to execute. Since the assumption is on fail silence, it implies that no input tokens are received at the input of the *actuator* type actors. Thus, the top event e_T is specified as a function of the set of *actuator* actors in PG.

4.2.2 Problem Statement

Given a platform graph $\mathcal{L}$, a top event e_T (and the logic operation which outputs it), generate a fault tree graph $\mathcal{F}$ and a correspondence map $f_{\mathcal{F}} : L_V \rightarrow \mathcal{F}$, such that:

- the set of basic events, E_B is in bijection with $P \cup C \cup A_S \cup A_C$ and indicates the failure of resources in the architecture,
- $f_{\mathcal{F}}(l)$, where $l = (p, a)$, returns the gate $g_1 \in \mathcal{F}$ that indicates the faulty/missed execution of actor, a on ECU, p, and

- $f_{\mathcal{F}}(l)$, where $l = (c, m)$ returns the gate $g_1 \in \mathcal{F}$ that indicates the faulty transmission of the data dependency m on channel c.

The problem as stated transforms a mapped FTDF graph as a platform model into a fault tree graph. The platform model $\mathcal{L}$ is specified by the designer, as well as the top event, which is specified in terms of a logical expression that describes the failure of the *actuator* actors. Under the fail silence assumption, this means that the *actuator* actors are unable to execute due to not receiving any input tokens. The correspondence map $f_{\mathcal{F}}$ is constructed by traversing the platform graph and transforming the failure modes of components in the platform graph into fault events in E_B and E_G. The set of basic events E_B represent the failure mode of ECUs, communication channels, *actuator*, and *sensor* type actors. The failures of those components are the inability to execute for ECUs and the inability to transmit data for communication channels, as specified by omission failure. Thus, those components failure appear as basic events in the resulting fault tree. The intermediate events E_G of the fault tree will denote the combination of basic events that result in the top event occurring.

It is noted that the top event e_T can be deduced by the semantics of the FTDF model of computation. For example, each *output* actor A_O is annotated by the designer with the minimum number of *actuator* types that it must be able to update in order to achieve a correct actuation of the control algorithm. For example, consider two *actuator* types, $a_1, a_2 \in A_C$. The top event e_T could be described by the output of an logical *AND* operation where the inputs to the logical operation are the failure to update the actuators a_1 and a_2, i.e. receive no input tokens. In general, designers may want to specify different top events to assess other aspects of the system response to faults in terms of other signals or actors within the platform graph. For this reason, the top event is taken as input into the fault tree generation problem, and for reasons of simplifying the discussion, the top event is specified in terms of the *actuator* actors.

4.3 Fault Tree Construction Algorithm

The fault tree construction algorithm defines the correspondence map $f_{\mathcal{F}}$ where each node in the platform graph maps to a fault event in the resulting fault tree. The platform graph, $\mathcal{L}$, exhibits strong structural dependencies amongst its actors. This dependency, along with semantics of the computational model, provides the necessary information to build the fault tree of a system modeled using FTDF. Thus, a recursive procedure is implemented in an algorithm to traverse the fault tolerant platform graph and generate a fault tree of the system.

The pseudocode for creating the fault tree for a given platform graph $\mathcal{L}$ and a

specification for the top event $e_T = f(A_C)$ as a function of actuators in the platform graph is presented in an algorithm that traverses the platform graph and perform a transformation of nodes in the platform graph to a set of fault events in the resulting fault tree. The algorithm starts with the execution of the *GenerateFaultTree* routine shown in Algorithm 4.1 by identifying the set of actors that are actuator types in the platform graph. For each actuator in the set, the subroutine *DevelopSubTree* in Algorithm on line 5 generates a fault graph for nodes that are in the path between the selected actuator and each actor of a sensor type that is also along that path in the platform graph by performing a graph traversal. The result is a fault graph where the actuator is the source node of the fault graph, denoted by the fault event e_i in the algorithm where $i = 1, 2, \ldots, |A_C|$. In the construction of e_i by Algorithm 4.2, the sensor typed actors are mapped to basic events, and the sensor actors mark the end of graph traversal. Once the set of fault graphs for the collection of actuator actors are constructed, then the graphs are combined into the resulting fault tree $\mathcal{F}$ by using a logic gate that relates the combination of actuator fault events as defined by the top event specification e_T. The subroutine *AddGate* on line 8 constructs the top level gate e_{TG} for e_T. The top level gate along with e_T are composed, and the fault tree $\mathcal{F}$ is returned from the routine.

The subroutine *DevelopSubTree* is the core of the fault tree generation as it performs the traversal of the platform graph. During the traversal, the algorithm creates and stores fault graphs after transforming components in the platform model into fault events, and it composes the fault graphs together as each actor is visited in the traversal of the system platform model. It is illustrated in Algorithm 4.2. When a vertex $l \in L_V$ of graph $\mathcal{L}$ is visited, the algorithm creates and stores a $\mathcal{F}_{Sub_l}$ at l. This fault graph is then appended to $\mathcal{F}$, and the operation is called recursively on each input of vertex l until a *sensor*, $a_s \in A_S$, is reached at which point the recursion ends.

The routine *DevelopSubTree* first generates initial parameters. The operation *pre(a)* returns the set of sources of actor a, and *minFire(a)* returns the number of inputs that are needed to fire actor a, and it depends on the firing rule of a. The number of inputs to fire an actor is given by the designer as a parameter when the FTDF graph is initially specified. Next, fault event of fault graph $\mathcal{F}_{Sub_l}$ is created and a number of fault events that are based on the type of actor encountered during traversal are also created. Mathematically, when the fault events of the actor is created, the result is a tree since the leaf nodes are not shared, and the root of the tree is the fault event for the actor. For example, event e_3 is created as a basic event for a *sensor* if $a \in A_S$ or a basic event for an *actuator* $a \in A_C$. A *sensor* or *actuator* basic event corresponds to an abstraction of a fault in the electro-mechanical hardware of the sensor or actuator devices. The algorithm adds an *OR*

Algorithm 4.1: GenerateFaultTree

Input : A platform graph $\mathcal{L} = (L_V, L_E)$ and a top event specification $e_T = f(A_C)$
Output: A fault tree $\mathcal{F}$

```
1  begin
2      E_i ← {∅};
3      for a_i ∈ A_C do
4          l_{a_i} ← (p_i, a_i) ∈ L for p_i ∈ P where P ⊂ L_V;
5          e_i ← DevelopSubTree(l_{a_i});
6          E_i ← e_i ⋃ E_i;
7      end
8      e_TG ← AddGate(e_T, E_i);
9      return F(e_TG, e_T)
10 end
```

gate with events e_1, e_2, and e_3 (if applicable). Here, e_1 corresponds to a fault of the ECU, e_2 is a fault that describes when the actor cannot fire because of missing tokens. If $a \in A_S$, the subroutine terminates the recursion immediately.

For each medium that connects actors a and a_j, an input fault event is created for that medium. The input fault event specifies that a medium was unable to deliver a token at the input of actor a. The routine then checks for the location of the medium, whether it is a connection between actors on the same ECU via shared memory or if it is a channel that connects two actors across different ECUs. If the medium is on the same ECU, no immediate event is created and the algorithm is called recursively to further develop that event. The routine then creates an event for each edge l_k, and it adds each event to an *OR* gate. An event created at this point models the fact that remote data from actor instance l_k is not delivered to actor instance $l = (p, a)$ when either the channel used is faulty or the remote actor fails to execute. The input edges to l_k are further developed by a recursive call to *DevelopSubTree*. The *InputFaultEvent* event (e_j) is then composed of the logical *AND* of the fault events of the various replicas generating the input. The choice of the *AND* composition derives from the fail silent assumption; changing the fault model will impact how these events must be composed. At the end of the algorithm, notice that the second level of the tree (second from the root event) is created with a *K-of-N* operation. The *K-of-N* operation is useful to create a logical relationship between a subset of inputs. This is a one-to-one correspondence to the *input* and *arbiter* actors, which have partial firing rules. More specifically, the *K-of-N* operation requires that input events must occur before an output event to occur at that intermediate event. When $x = y$, this simplifies to an *AND* operation and when $x = 1$, this simplifies to an *OR* operation. As an example, let $y = |pre(a \in A_I)|$ and $m = |minFire(a)|$. Then for the *K-of-N* operation of actor a, $x = (ym + 1)$. This means that if x tokens are not

present at the input of actor a, then the *K-of-N* operation for actor a produces a fault event $e = 1$ at its output. Essentially, the routine composes subtrees to create the system fault tree. When the subtrees of each node in the platform graph are composed, some leaf nodes of the tree are shared between different trees, and it is at this point that the fault tree is more generally a fault graph.

4.4 Complexity of the Fault Tree Construction Algorithm

The implementation of the fault tree construction is described. A platform graph is given as input to *DevelopSubTree*. Actors in the mapped graph are traversed in a depth first traversal scheme from *actuator* to *sensor* actors. As each actor in the mapped graph is visited once through recursion of *DevelopSubTree*, a subtree of fault events is created and stored in memory. Each subtree contains levels of fault events that contribute to the failure of that actor instance. The first level corresponds to the failure of the actor instance in the platform graph. Each actor instance will have a first level fault event, therefore, its memory complexity is $O(1)$. The second level is a combination of three fault events that must occur for the actor instance to fail. Fault events at the second level are an actor not receiving enough input tokens to fire, the ECU for which the actor instance is mapped fails, or the actuator hardware fails to update the actor. The second level results in a memory complexity of $O(1)$. The third level contains events for determining when an input to the actor instance is faulty, as in the case where an actor instance cannot fire due to the lack of input tokens. Fault events on the third level depend on the number of inputs to an actor instance, and they are created and stored in the first recursive call to *DevelopSubTree* on each input with a memory complexity of $O(D)$, where D is the average number of inputs per actor instance in the mapped FTDF graph. The fourth level constructs subtrees that determine how fault events occur for the source actor that feeds into the actor instance. The second call to *DevelopSubTree* occurs at this point, and since a subtree of fault events are developed for the inputs to the source actor, it has a complexity of $O(D)$. At the point where the algorithm reaches a *sensor* actor, the recursion ends since *sensor* actors have no input events to further develop. Furthermore, in the case that an actor is replicated (i.e. it contains the same actor dependencies, but located on a different ECU), the subtree of the replicas will be the same, and each subtree is stored individually. The resulting memory complexity for developing a subtree for each actor instance in the mapped graph is $O(D^2)$. Given a total number of M actor instances in the mapped graph,

Algorithm 4.2: DevelopSubTree

Input : $l = (p, a) \in (P \times A) \subset L_V$
Output: $\mathcal{F}_{Sub_l}$

Begin;
$N \leftarrow |pre(a)|$;
$M \leftarrow minFire(a)$;
$\mathcal{F}_{Sub_l} \leftarrow CreateActorEvent(l)$;
$e_1 \leftarrow CreateEcuBasicEvent(p)$;
$e_2 \leftarrow CreateInputFaultEvent(a, N)$;
if $a \in A_S$ **then**
    $e_3 \leftarrow CreateSensorBasicEvent(a)$
end
if $a \in A_C$ **then**
    $e_3 \leftarrow CreateActuatorBasicEvent(a)$
end
$\mathcal{F}_{Sub_l} \leftarrow AddGate(\mathcal{F}_{Sub_l}, OR(e_1, e_2, e_3))$;
if $a \in A_S$ **then**
    return $\mathcal{F}_{Sub_l}$ `/* terminal case, end recursion */`
end
for $m_j \in pre(a)$ **do**
    $a_j \leftarrow pre(m_j)$;
    $e_j \leftarrow CreateInputFaultEvent(m_j, l)$;
    if $l_{local} \leftarrow (p, a_j) \in pre(l)$ **then**
        $e_{local} \leftarrow DevelopSubTree(l_{local})$
    end
    for $l_k \leftarrow (c, m_j) \in pre(l) \cap (C \times m_j)$ **do**
        $e_k \leftarrow CreateRemoteInputEvent(l_k)$;
        $e_c \leftarrow CreateChannelBasicEvent(c)$;
        $e_{ra} \leftarrow CreateRemoteActorsEvent(l_k)$;
        $e_k \leftarrow AddGate(e_k, OR(e_c, e_{ra}))$;
        for $l_r \in pre(l_k)$ **do**
            $e_r \leftarrow DevelopSubTree(l_r)$;
        end
        $e_{ra} \leftarrow AddGate(e_{ra}, AND_{\forall e_r}(e_r, e_{local}))$;
    end
    $e_j \leftarrow AddGate(e_j, AND_{\forall e_k}(e_k, e_l ocal))$;
end
$e_2 \leftarrow AddGate(e_2, VOTE_{\forall e_j}(N, M, (e_j)))$;
return $\mathcal{F}_{Sub_l}$;

the resulting memory complexity is $O(MD^2)$. The execution time complexity for the algorithm is $O(D^{2M})$ since each call to *DevelopSubTree* is $O(D^M)$. The performance of the algorithm is computationally expensive and highly dependent on the number of actors and communication channels in the mapped FTDF description.

4.5 Experimental Case Study

In this section, to test the fault tree construction algorithm, an example of an "on-off", or "bang-bang", controller for an inverted pendulum is discussed. The purpose of this case study is to demonstrate the fault tree generation technique on a tractable example, and to illustrate its use within the design flow that is described in Chapter 3. Since manual construction of fault trees are highly dependent on subjective factors such as specified failure modes, system topology, and the analysts understanding of the system, a complex example for this work would be difficult to evaluate the generated fault tree. The example is simple enough to validate the quality of the fault tree that is produced by the fault tree generation algorithm by inspection and using simulation to validate that the design is according to specification. The task dependencies of the digital controller are described by a FTDF graph in Figure 4.1. In Figure 4.2, a platform graph that contains three ECUs and two communication channels is also shown.

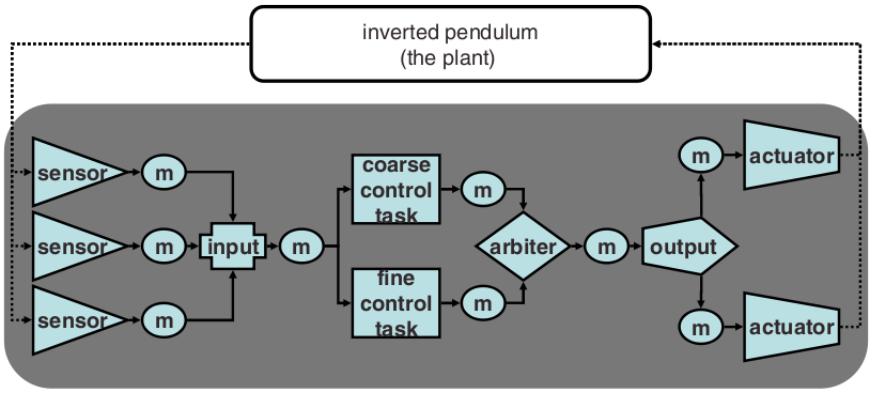

Figure 4.1. *Function model for a distributed control example, the inverted pendulum case study.*

The controller consists of three position sensor devices, one *input* actor that performs sensor integration and assesses the current pendulum position, two different *task* actors that represent two controllers (coarse and fine) that require different computing power, one *arbiter* that selects the output of one of the controllers (the fine one, whenever available), and an *output* actor that directs the control action to

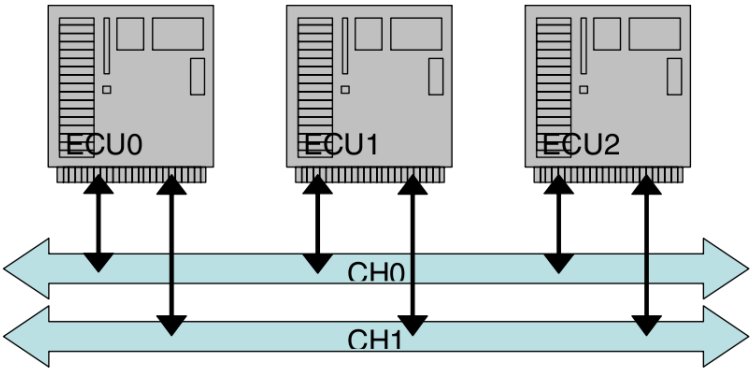

Figure 4.2. *Distributed architecture model for the inverted pendulum control case study.*

two different actuators. The architecture platform model of Figure 4.2 and consists of three ECUs and two communication channels. Each ECU samples one of the three sensors, while and drive actuator and respectively. The plant describes the mechanical dynamics of an inverted pendulum on a moving cart. This is a classical example that is commonly used in benchmarks for the design of feedback control systems.

The fault tolerance requirements used to drive the synthesis of the redundant deployment consist of:

- execute the entire algorithm in absence of faults (default behavior)
- in the presence of a single ECU fault, guarantee the execution of the critical subset of the FTDF graph, so that at least one of the Actuators is updated.

The critical set is a set of actors that mandates the execution of at least one replica of:

- the *input* actor ,
- the coarse controller ,
- the *arbiter* actor , and
- the *output* actor .

The criticality of a task in this context is determined from user input, and it is annotated into the model and represented one of the actors in the critical set described above. In particular, in order to fire, actor requires that at least two of the *sensors*

Actor	ECU0	ECU1	ECU2
SEN0	X		
SEN1		X	
SEN2			X
IN		X	X
FUNc	X	X	
FUNf			X
ARB		X	X
OUT		X	X
ACT0	X		
ACT1			X

Table 4.1. *A mapping of the inverter pendulum case study with redundancy.*

deliver their data to it. The *arbiter* actor can fire with one of its two inputs present. As a result, a synthesis tool [95] performs the redundant mapping described in Table 4.1.

It is worth noting that the fine controller is not replicated, because it was not specified as part of the critical set. This mapping is guaranteed by construction to tolerate single ECU failures. However, there is no finer quantification of the degree of fault tolerance achieved by this particular implementation.

4.5.1 Analysis of Fault Behavior using Fault Trees

To measure and quantify the degree of fault tolerance of this case study, fault tree analysis is used to determine how and why the system may fail under a given set of failure patterns. A *failure pattern* in this context describes the set of failures that are assumed to occur in the system. In this case study, the failure patterns are assumed to be resource crash failures, i.e. permanent ECU omission failures and permanent communication channel failures. The fault tree that represents the controller is extracted using automatic fault tree generation. The fault tree is then analyzed by the Item Toolkit [115], a commercial tool distributed by Item Software. In this case study, a cut set analysis, reliability analysis, and quantification of importance measures are used.

	Cut Sets
1	SEN0, SEN1
2	SEN0, SEN2
3	SEN1, SEN2
4	ECU0, ECU1
5	ECU0, ECU2
6	ECU1, ECU2
7	CH1, CH0
8	ACT0, ACT1
9	CH0, ECU1
10	CH0, ECU2
11	CH1, ECU0
12	CH1, ECU2
13	SEN0, CH1
14	SEN1, CH0
15	SEN2, CH0
16	SEN0, ECU1
17	SEN0, ECU2
18	SEN1, ECU2
19	SEN1, ECU0
20	SEN2, ECU0
21	SEN2, ECU1
22	ACT0, ECU2
23	ACT1, ECU0

Table 4.2. *Minimal cut sets for the pendulum example.*

Cut Set Analysis

The cut set analysis permits to identify the combinations of events that generate a system failure. The list of minimal cut sets for the mapped pendulum example is presented in Table 4.2 where each row of the table represents a cut set, and each element in that set corresponds to an event. Since the failure of an architecture resource is a basic fault event, resources (i.e. ECUs, channels, sensors, and actuators) appear in the cut sets.

The name of the basic fault events correspond to architecture faults in Sensors, Actuators, ECUs, or communication channels. Based on the minimal cut sets for this example, it is clear that no single ECU failure leads to the system failure (assuming the top event is the failure of all actuators in the given system mapping). Moreover, no single channel failure leads to a system failure. Note that channel failures were

not part of the fault behavior specified to drive the deployment of the synthesis algorithm.

Reliability Analysis

The reliability analysis combines the reliability information about components into the reliability of the system. Starting from the mean-time-to-failure (MTTF) and the mean-time-to-repair (MTTR) of the basic events. Among other metrics, the system MTTF is computed. In this simple example, we assume a system lifetime of 5000 hours, all basic events have the same reliability, and an exponential distribution is assumed to describe the failure of the component over time, where the MTTF of a basic event is 2000 hours and the MTTR of a basic event is 8 hours. The Item Toolkit returns a MTTF of 11015.81 hours for the system. Based on the simple specification, one could expect a reliability higher than that of a single component failing (*MTTF (system)* ¿ 2000 hours). It is also observed that the solution provides a dual redundant system at its weakest points. Hence, assuming no repair, a $MTTF(system) \geq 1.5 \cdot MTTF(baseevent) = 3000hours$. Having a fault tree now allows experimenting with different mixes of more or less reliable components to explore design trade-offs.

Importance Analysis

The Item Toolkit also provides various Importance metrics for systems under analysis. The Importance metrics considered in this analysis include the Barlow-Proschan Importance, Birnbaum Importance, and the Fussell-Vesely Importance. expectations since ECU2 contains more actors than ECU0 or ECU1, thus making ECU2 a highly important component in the given system mapping shown in Figure 4.1. Table 4.3 provides the importance values for the basic events in each of four different system mappings. In the figure, the first column is the basic events. The last three columns are the importance metrics for each of three mappings.

4.5.2 Validating the Automatic Fault Tree Construction

The case study is to used to validate that the automatic fault tree construction does provide an accurate view of the failure behavior of the system under the stated conditions. The conditions are the expected failures that may occur in the system, as described by the fault behavior. In this case study, the fault behavior considers the ability to tolerate a set of failure patterns that originate from the architecture platform. Assuming the failures within the set of failure patterns occur

Basic Event	Map1	Map2	Map3	Map4
ACT0	0.043478	0.050154	0.05	9.83E-06
ACT1	0.043478	0.050055	0.025	9.83E-06
SEN0	0.108696	0.124988	0.125	0.247771
SEN1	0.108696	0.10001	0.125	0.00198
SEN2	0.108696	0.124888	0.125	0.247771
ECU0	0.130435	0.149866	0.125	0.248758
ECU1	0.108696	0.10001	0.125	0.00198
ECU2	0.152174	0.149866	0.15	0.248758
CH0	0.108696	0.075231	0.1	0.001481
CH1	0.086957	0.074933	0.05	0.001481

Table 4.3. *Importance of basic events on different system mappings.*

simultaneously, it is deduced by inspection and simulation using fault injection, as implemented and described by Pinello [94], that the fault tree does capture the possible failure patterns accurately when given a mapping. The failure patterns, and thus, the fault tolerant requirements of three additional mappings are considered below:

1. *Map 2.* The requirement is that all critical actors be executed in the presence of any two simultaneous ECU faults, i.e. corresponding to failure patterns {ECU0, ECU1}, {ECU1, ECU2}, {ECU0, ECU2}.

2. *Map 3.* In addition to requirements in Map 2, all critical actors must be executed also in the presence of any single channel fault, i.e. corresponding to failure patterns {Channel0}, {Channel1}.

3. *Map 4.* The requirements in Map 3 are considered with changes. The task dependencies in the sensor fusion algorithm is modified to correctly estimate the pendulum position, in addition to using a single measurement from the sensors. It contains a partial firing rule.

The fault tolerant requirements specified by the different mappings are compared with the fault tree that is generated from the system model. Each mapping represents a deployment, which is then used to evaluate the impact of different mappings on system reliability.

4.5.3 Impact of Different Mappings on Reliability

The results of the analysis show that the highest contributors to system failure for the first mapping shown in Figure 4.1, Map 1, are the ECU faults, then the sensors faults, the channels faults, and the least impact is due to actuator device faults. The first attempt at improving system MTTF is to improve reliability of the ECUs, and in general of each of the components. This would result in longer MTTF for the system.

The synthesis tool in [94] cannot meet the requirements for Map 2 and Map 4, for example because the failure of two ECUs makes it impossible to read two of the sensors, so the sensor fusion actor cannot fire and none of the successor actors can fire. Nonetheless, the synthesis tool introduces in the deployment more redundancy in the execution of actors, such that there are now three replicas of each of the critical actors. Also, in Map 3 and Map 4, communication is more redundant. After generating the fault trees for the three deployments, analyzing them in the Item Toolkit, and performing a timing analysis, the results in Figure 4.3 are obtained. The results in the figure show that the additional redundancy improves the MTTF

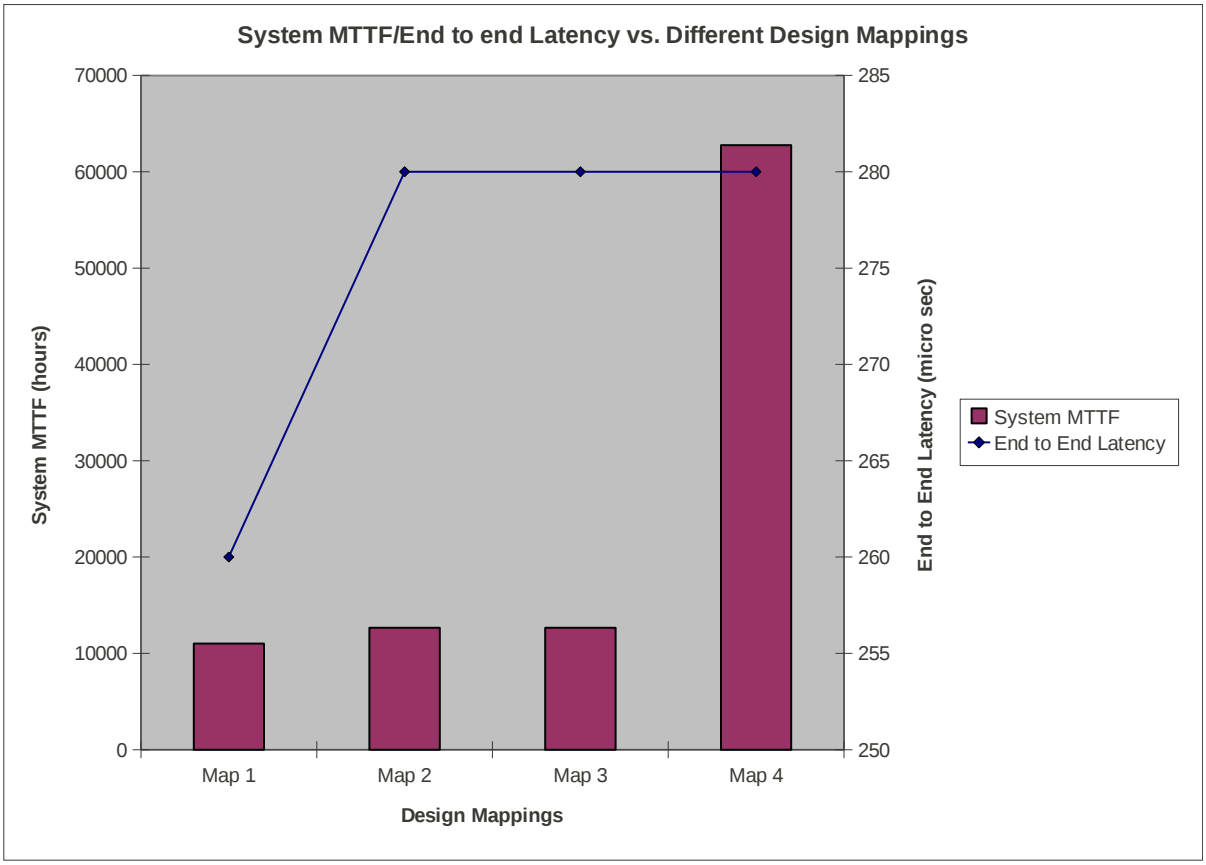

Figure 4.3. *A plot of mean-time-to-failure, the number of cut sets, and end-to-end latency of multiple mappings.*

only marginally, whereas the use of a more robust sensor fusion algorithm yields much better results in this example. The end-to-end latency is the result of a timing analysis for the four deployments and indicates the latency from reading the sensors to updating the actuators. From the results, it is clear that having more replicas to be executed on the same execution architecture leads to longer latencies. This information allows the system designer to trade-off between the desired replication and required latency through the controller. Thus, this method can be used to explore other design trade-offs that involve redundancy for achieving fault tolerance.

4.6 Summary

This chapter presents the problem of automatically constructing a fault tree from a given system model. Unlike previous work, the method employed in this work captures not only the syntax, but it also captures the semantics of the underlying model of computation. This can reduce the need for an additional model that describes the failure behavior of a system, as is done in many of the previous techniques. Moreover, the generated fault tree graph captures the functional dependencies of the application, thus, the fault tree captures the impact of platform failures on the correct function of the application without arbitrarily specifying a criticality to the application tasks, as is done is related work.

As shown in this section, the construction begins with a platform model for which describes the behavior of the system as a set of communicating tasks that are assigned to a set of ECUs and communication channels. The syntax and semantics of the computational model that describes the application is captured in the generated fault tree. In particular, since the computational model is a directed graph and the essence of fault trees stems from the capability of determining how a system can fail due to component fault events that may propagate throughout the system, it is observed that a graph traversal could be used to capture fault events in the fault tree. The fault tree captures failure modes that are described in terms of the behaviors and signals that characterize the system model. Thus, the fault tree captures omission, value, and timing failures of the signals and actors in the model that may occur. A case study is also presented that shows the feasibility of this method, and it is used to validate that the fault tree that is derived from the system model is correct.

Chapter 5

Architecture Exploration and Tool Support for Fault Tolerant Designs

This chapter describes the problem of determining a set of system architectures for a distributed application that satisfies the requirements of a given application subject to fault tolerant and reliability constraints. In system level design for fault tolerant architectures, the goal of an exploration of the system architecture is to find a set of solutions for a given specification of a distributed application. An algorithm that performs the exploration of fault tolerant architectures is proposed. The algorithm is driven by a fault tree generation and evaluation of the generated fault tree as described in Chapter 4. Then, the set of tools that supports the design flow and an example of architecture exploration on an automotive case study will be described.

5.1 Background

Section 1.5 discussed some works that are related to addressing the problem of improving the reliability of a system. The goal is to select the number and type of components that are utilized in a system such the cost of the system is minimal with respect to system level reliability constraints. A general formulation that models this problem with a series-parallel configuration as in Figure 5.1 when given as a

priori the cost and reliability of each component in the system is given as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_i \sum_j c_{ij} z_{ij} \\ \text{subject to:} \quad & \prod_i f_i \geq R_S \\ \text{where} \quad & f_i = 1 - \prod_j (1 - r_{ij})^{z_{ij}}, \\ & z_{ij} \in \mathbb{Z}^*. \end{aligned}$$

The decision variable z_{ij} is an integer value that denotes the number and type of component j that is used in subsystem i of a series-parallel configuration. In this formulation, the system reliability R_S is expressed as a function of the components' reliability r_{ij} by way of f_i. The specific form of f_i depends on how the system components are configured and the reliability function that is adopted for a particular component. The objective function is then defined as a function that seeks to minimize the cost of a design solution represented by the combination of the values for z_{ij}. The problem is known to be NP-hard [22] due to the form of the reliability function of the system, f_i which is non-linear, thus a number of heuristic techniques have been employed to solve it. The problem of selecting from a set of components with known reliability where components may be redundant is so-called the *reliability-redundancy allocation* problem. The problem may be formulated with either reliability or cost as the objective function, whereas in the case that reliability is the objective, the problem would be considered one in which the reliability is maximal subject to a cost constraint. Its solution includes two parts: the component choices and their corresponding optimal redundancy levels.

In the context of distributed embedded systems, the configuration is not readily a series-parallel configuration. To utilize the formulation of the reliability-redundancy problem, the system must transformed into a series-parallel configuration. Few existing works consider the design of distributed embedded systems. In the works that do consider replication and selection of components in system level design, the limitations are that those works do not consider the effects of failures, hence, how error propagates through the system is not captured. Furthermore, failure patterns are considered only in the case where an importance value is given for a specific task based on user input. In the design flow of this dissertation, failure patterns are considered as fault tolerant requirements on the system operation, and the use of fault trees for reliability modeling and analysis allows the evaluation of the fault tolerant requirements of the system under an assumed fault model. The fault tree allows a complex system structure to be decomposed into minimal cut sets, which provides a way to evaluate the fault tolerance and reliability requirements of the system in terms of possibly multiple failure modes for a component. The

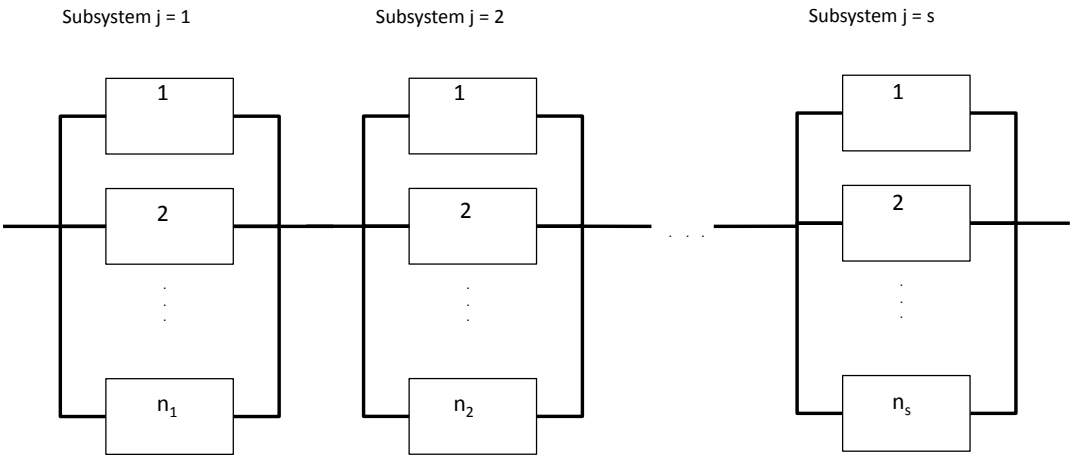

Figure 5.1. *Example of a series-parallel configuration of components.*

form of the minimal cut sets is the same as a series-parallel configuration where each component in the configuration is a failure mode that describes the failure of a component, hence, the failure of component may contain multiple failure modes. By using the quantitative analysis results of the fault tree, the importance of tasks are determined in terms of component failure distribution, which may differ depending on failure modes considered, and the structure of the system.

5.2 Problem Formulation

The goal is determine a system architecture that meets reliability and fault tolerant constraints. To achieve this goal, active replication of architecture resources is applied. It is assumed that a system platform graph $\mathcal{L} = (L_V, L_E)$, a reliability requirement $Q_{req} \in \mathcal{R}$, and a fault tolerant requirement Φ are given. The fault tolerant requirement is described by a set of failure patterns, or a combination of resources, that must be tolerated. The system platform graph, as defined in Definition 3.3, represents an initial system configuration. It is also assumed that the reliability p_j and the unit cost c_j of each component in the resource architecture model $R_{\mathcal{PG}} \subseteq L_V$ of $\mathcal{L}$ is given a priori. The expected output is a platform graph $\mathcal{L}'$ that meets the requirements Q_{req} and Φ.

Let n denote the number of task instances in the functional model $A_G \subset L_V$ of $\mathcal{L}$ and m be the number of resources in the resource architecture model of $\mathcal{L}$. A variable, $d_{ij} \in \{0, 1\}$ denotes a mapping decision to allocate task i onto resource

of type j where $d_{ij} = 1$ means that task i is allocated to resource j and $d_{ij} = 0$ otherwise.

Let α and β denote variables that determine the allocation and binding respectively. The variable α is a constraint on the resulting platform graph $\mathcal{L}^{'}$ that means an allocation can only contain the available resources, since resources could be chosen from a library. They are the resources that are used in the selection of resources for the implementation of the design. The choice of resources depend on a selection of more reliable components or active redundancy, as formulated by the reliability-redundancy allocation problem. The variable β is a subset of the edges in L_E of graph $\mathcal{L}$, where each software task in the application is bound to a hardware resource that executes this task at runtime. Since, all data dependencies must be satisfied for the system to function correctly according to specification, then a pair (α, β) is *feasible* if all data dependencies are correct. This means that when replicating a function task,

- for every edge, $e = (v_i, v_j) \in E$ of the FTDF graph in the functional model, where v_i points to v_j in the functional model, there is an edge (v_i, v_j) that points to every replicated actor in the redundant FTDF graph,
- for every edge, $e = (v_i, v_j) \in E$, starting with the actor v_j, there is an edge $e = (v_i, v_j)$ at every replicated actor of v_i, and
- each replica of actor v_i has to be bound to a resource $r \in R$ in such that no other replica of v_i is bound to the same resource.

The constraints are representative of an active redundancy policy. The constraints enforces that each replica must be deterministic, that is, that for the same set of input tokens, each replica computes the same set of output tokens, and the output of each replica goes to the input of each successor actor as specified in the non-redundant FTDF graph. Furthermore, since a resource can fail, it is useless to assign a task and its replica to the same resource.

The constraints also considers the fault tolerant and reliability requirements, Φ and Q_{req} respectively, where the fault tolerant requirement is defined by the failure patterns to tolerate, as in the example of Section 4.5, and Q_{req} is the minimum system reliability that must be obtained. The reliability of each component can be obtained by estimation based on experience, prediction of the failure distribution, or by testing the load on the actual resource component and inject faults to determine its reliability. Additional constraints that affects the consumption of resources that are used to achieve fault tolerance and reliability requirements may also be added

as constraints. The problem is described as follows:

$$\begin{aligned} \text{minimize:} \quad & C_S = \sum_{j=1}^{m} f(c_j, z_j, d_{ij}) \text{ for } i = 1, 2, \ldots, n \\ \text{subject to:} \quad & Q_S(z_j, p_j, d_{ij}) \geq Q_{req} \\ & FT_S(x_j, d_{ij}) \subseteq \Phi \text{ such that } \Phi \text{ is satisfied} \\ & \exists(\alpha, \beta) \text{ for } \alpha \subseteq R_{\mathcal{PG}}, \beta \subseteq L_E. \end{aligned}$$

In this formulation, the objective is to minimize the unit cost of the system C_S, as a function of a selection of components with known reliability and unit cost. The decision variables are in the number and type of resource j, as given by z_j, and the allocation and binding of tasks from the functional model onto the selected resources of the architecture model, as represented by d_{ij}. The selection and allocation are subject to constraints on a minimum reliability of the system Q_S that must be met and if the fault tolerant requirement Φ is satisfied. The fault tolerant of the system FT_S is determined by an evaluation of the system configuration as a function of z_j and d_{ij}. The evaluation of the fault tolerant requirement is obtained by evaluating the structure function, as defined in Definition 2.1, of the system configuration.

5.3 An Algorithm for Exploring Architecture Alternatives

An algorithm is proposed that describes how the use of fault tree analysis can be used in evaluating alternative architectures. The algorithm begins by reading an initial system platform graph, S_k along with a top event e_T. The objective is to reduce the number of components instantiated in the design that satisfies the set of requirements that are given as design constraints. The goal of the design exploration is to find a solution to the problem that is formulated in Section 5.2. The general flow for the algorithm is illustrated in Figure 5.2. The variable k is an iteration number that is initialized to 0. The initial platform graph is stored in a data model, from which a fault tree generation step translates S_k and e_T into a fault tree $\mathcal{F}_k$ according Algorithm 4.1 that is shown in Section 4.3. An evaluation of the fault tree at iteration k yields the tuple $< mcs_k, Q_k(t), IMP_k >$, where mcs_k is the set of minimal cut sets, $Q_k(t)$ is the reliability of e_T, and IMP_k is the set of importance metrics. The algorithm then checks to determine if the fault tolerant requirement Φ is satisfied and if the system reliability Q_{S_k} meets the required reliability of the

system. If Φ is not satisfied or $Q_{S_k} < Q_{req}$, then an improvement strategy is applied to generate a new configuration S'_k such that the constraints of Section 5.2 are met. To achieve this goal, the improvement strategy may apply a solution to the reliability-redundancy allocation problem where the minimal cut sets are represented by the series-parallel configuration. The strategy that is applied in the case studies from Sections 4.5 and 5.5 ranks the most critical basic events, which corresponds to resources in S_k, according to descending order of importance determined by an ordering of IMP_k. The higher the importance value, the higher the priority in selecting a new architecture (either by replication or replacement of a resource with higher reliability). Furthermore, once a selection or replication of resources has been performed, the problem of mapping must be considered. This problem is also addressed in the improvement strategy. A feasible mapping results in a new configuration S'_k that is stored in a data model. The value of k increments, and the algorithm iterates since a mapping produces a new configuration that depends on the data dependencies of the functional graph of S'_k, the algorithm iterates to evaluate the new mapping. The algorithm continues to iterate until the fault tolerant and reliability requirements are met, thus producing a new configuration S'_k that gets stored into a data model.

5.4 Supporting Tool Chain

This section describes the set of tools that supports the design flow and the exploration algorithm. The tools include a data model that is used as a repository to store the system model as well as the fault tree models that are generated. The data model allows for the transformation of fault events back into the corresponding components in the system model by allowing the user to perform queries. A set of fault tree generation and analysis tools are also used in the design flow. The advantage is that they may be applied more generally to cases where a user simply wants to generate a fault tree and analyze it.

5.4.1 A Data Model for Design Capture

The design needs to be specified and captured in a data structure that allows ease of generating the fault tree from the model and storing the resulting fault tree and designs. It is of advantage that the function and architecture models are specified separately per the design methodology to allow flexibility in the different set of design implementations that may be realized. The root node of the XML structure is the design specification that contains the different system models. The system spec-

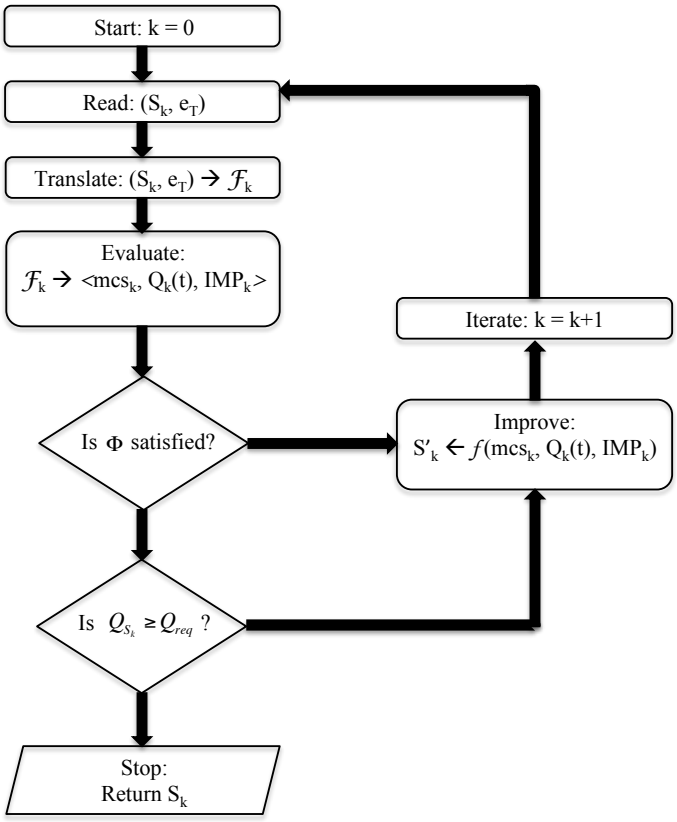

Figure 5.2. *An algorithm to support the design exploration of alternative architectures.*

ification has three major parts, including the function model, architecture model, and system platform model.

A library node within the XML structure is included, and it represents the set of hardware resources that are available to the designer. It is presumed that for any resource component that may be used in a design, then it is specified in the library by way of adding it as an entry into the library node of the data model. Each resource has a type and associated quantities that characterizes the resources. In the case of this example, the quantity is the failure rate. Additional characteristics of the resource may be added in the library for a specific type of hardware resource, and such quantities may be acquired from data sheets, manufacturer records, or experience.

Figure 5.3 illustrates how the application in the functional model is captured in the data model as a set function tasks. The function blocks are typed, and they are given a name. The nets represent the dependency relation between the function blocks. Together, typed function blocks and the dependency relationship between function blocks help facilitate the automatic fault tree generation, and the designer

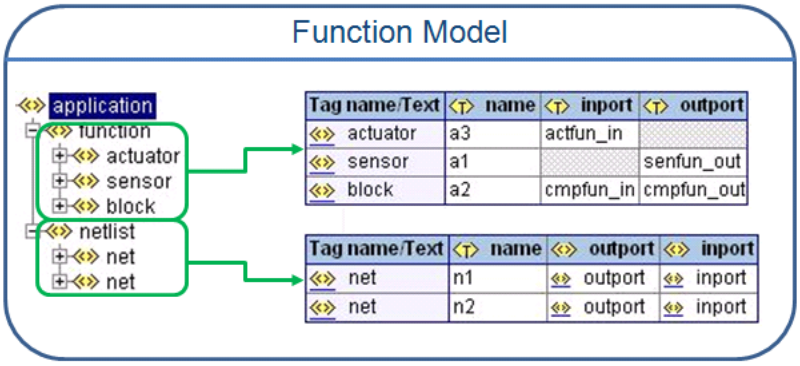

Figure 5.3. *Specification of the function model in the data model.*

can reason about how the model objects are related. As an example, in the case where only a subset of input signals are required for a block to execute, a special rule that governs that behavior can be set. As such, we use the execution rule in this work to determine branches in the generated fault tree that addresses the behavior where all input signals are not necessary for the function block to execute.

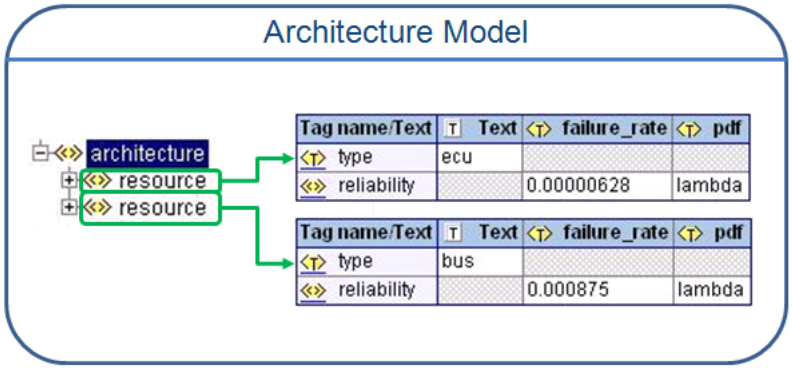

Figure 5.4. *Specification of the architecture model in the data model.*

In Figure 5.4, the hardware model of the example is captured. Each resource type, ECU and BUS, are characterized by failure rate and an exponential probability distribution. The fault tree analysis uses the quantities to evaluate and assess a design since this type of input data is needed by the fault tree analysis tool.

The results of the mapping of the functional and architecture models into a system platform model is captured in Figure 5.5. The system platform model integrates the hardware resource and application models into a single design implementation. This model specifies for the functional model, which tasks and nets are mapped to which ECUs and communication channels from the architecture model. This is denoted by the *resource* node in the figure. The advantage is that the application is modeled independently of the hardware that is used to execute the application, hence, enabling exploration of design alternatives using degrees of freedom in the

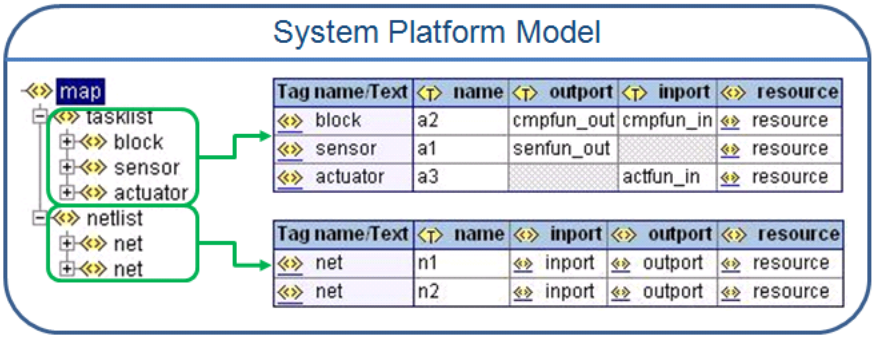

Figure 5.5. *Specification of the system platform model in the data model.*

application model, the hardware resource model, and the architecture model by changing the allocation policy.

The final part of the data model captures the fault tree that is generated from the system platform model. The structure in the data model for the fault tree is illustrated in Figure 5.6. The fault tree is captured as a list of fault events that are identified by several parameters, including the logic operation, a unique fault event name, the type (basic fault event, intermediate, or top event) and the set of fault events that are used as input to the event's logic operation.

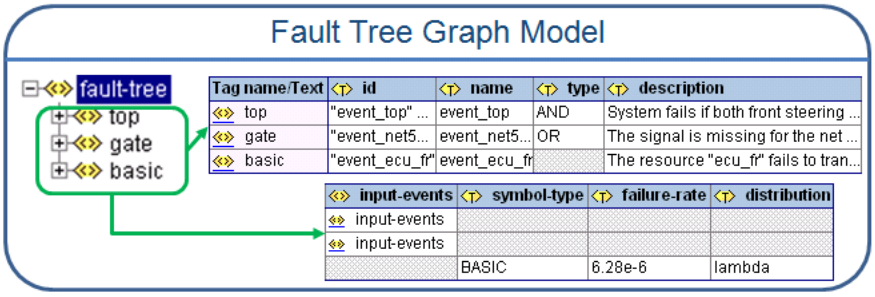

Figure 5.6. *Specification of the fault tree graph in the data model.*

5.4.2 Fault Tree Construction Tools and Analysis

The fault tree construction is implemented such that it automatically translates a system platform model into a fault tree. The analysis of the generated fault tree is done by porting the generated fault tree over to an external fault tree analysis tool. The tools that have been used in this work include the ITEM Software fault tree module, the Isograph FaultTree+, and Galileo from the University of Virginia. The external tools requires an interface to allow the fault tree to be read and accessed for analysis.

FTGen: Automatic Fault Tree Generation Tool

The Fault Tree Generation tool (FTGen) is the artifact of the automatic fault tree construction algorithm within this design flow. It reads a model specification from the data model, and it produces a fault tree that contains a set basic events and gates. The basic events represent the failure modes of the system, and the gates represent intermediate fault events. The fault tree generation tool translates a fully connected system model that is represented in the data model to a fault tree that is also stored in the data model.

The tool executes in two phases. In the first phase, a fault tree is created for each actor and signal, which will be referred to as a functional block and net respectively. Note that the blocks are typed (i.e. function, sensor, actuator, state memory, etc.) and mapped to a resource (i.e. processor). Nets, like blocks, are also mapped to a type of resource (i.e. processor, bus). The resource for nets can be processor, or ECU, and communication channel. When mapped to a ECU, the tasks that communicate exchange data through local memory on that ECU. During this first phase of the implementation of the fault tree construction, a hash table is created to store the subtrees that are created by visiting a net or node in the data model. The second phase will assemble each tree into a system fault tree characterized by a top event, basic events, and intermediate fault events, called gates. The fault events are uniquely named, and they may be duplicated at the end of this phase. Events with the same name are considered to be duplicates. This occurs when a fault event is referenced more than once by a visit to a predecessor node in the fault tree graph. As a result, a final phase removes duplicate events based on the event's name. The fault events are hashed based on their names, since the names are unique once mapped to a resource. The final phase creates an output file, which is the resulting fault tree that is stored into the data model. The implementation of the fault tree construction follows that of Section 4.3. This implementation differs from an original implementation that stored the entire subtree for a given task and net and recursively developed fault events. In that recursive implementation, the subtrees had to assembled, and this required a search. The hash implementation on the current and recent implementation is more efficient.

Fault Tree Analysis Tools

A set of external tools are used for the analysis of the generated fault tree. These tools are well-known to industry as tools that support the design and analysis of reliable systems and other systems by which could benefit from reliability models. In particular, the FaultTree+, Item Toolkit with Fault Tree Module, and the Galileo tools were used. The Galileo tools is an academic tool that allows the input of a fault tree, and it analyzes the fault tree to capture minimal cut sets, system reliability,

and other measures. This tool was used primarily for ease of access. For larger designs, this tool was too slow in its analysis to be integrated into the design flow automatically. The industrial tools were much faster and offered more capabilities, however, they were not as accessible and proved to be a challenge to import external fault trees. For those reasons, a module that extends FTGen is created that performs the fault tree analysis. It is faster than the Galileo tool, yet it offers less capabilities as the industrial tools. However, it allowed ease of accessing the generated fault tree, and it provides the ability to reduce the cut sets to a minimal set of cuts, computes the system and cut set reliability, and it computes several importance factors.

5.5 Architecture Exploration of an Automotive Steer-by-Wire Application

The design flow and tool support discussed in this dissertation is applied to an automotive control application. The system hardware architecture is part of an experimental vehicle [19] with multiple by-wire features, including steer-by-wire, brake-by-wire, propulsion-by-wire, and shift-by-wire. The application tasks are distributed across a total of ten ECUs that communicate using a combination of FlexRay and Controller Area Network (CAN) buses. The FlexRay bus is used primarily for communication between ECUs to support the data traffic of the by-wire applications. The CAN buses are used for non-safety critical functions, and twisted copper wire is used for hard-wired data transport to and from actuators and sensors. The fault tolerant objectives of the experimental vehicle influenced the system architecture. The primary objective of interest in this case study is to tolerate single point failures in the architecture.

The design model that is used to apply the methods and tools in this work is extracted from a Matlab/Simulink model that was used in an actual prototype of the system. The model contains application tasks as a data flow diagram by which the prototype can be simulated. In the functional model, the set of functions that are connected via directed edges represent the flow of signals between blocks. Blocks are artifacts of the Matlab/Simulink design environment. Consequently, this model is used in this work as a basis for automatically generating fault trees. Without loss of generality, a subset of the experimental vehicle model was extracted to capture the spatial redundancy in the architecture (a subset of signals and blocks) at a level of abstraction that would enable manual assessments of the front steering subsystem and to manually specify the functionality separately from the architecture. The front steering subsystem is a part of the steer-by-wire application that is presented in Cesiel *et. al.*[19]. In particular, the steer-by-wire application is designed to be

fault tolerant. Therefore the fault tolerant objectives of the system can be evaluated. In this work, the ability of the system to tolerate single point failures and at least three simultaneous failures of hardware resources are assessed.

For the purpose of this work, the original model is abstracted according to the model specification described above. The baseline model is in a centralized architecture where the four main tasks of the steer-by-wire subsystem are contained within one ECU, and they are executed under a periodic, time-triggered schedule with a predefined (static) scheduling order that does not change during a sequence of invocations. A brief description of this set of tasks is as follows:

- **T1**: Captures and conditions raw sensor signals and distributes the conditioned signals to all other controllers.
- **T2**: Provides fault detection and management, performs sensor fusion, and sets reference signals for computing the control law.
- **T3**: Computes the control law and sends copies of its results to each controller and to task T4.
- **T4**: Receives control commands and processes the signals for driving the actuators.

Each task in the model contains multiple sub-tasks that carry out the task execution. An additional architecture is assessed, and the baseline tasks are replicated, increasing the number of subtasks in the abstracted model.

5.5.1 Problem Statement

Given a model of a distributed control system $S = \mathcal{L}$, find a design $S = \mathcal{L}'$, that contains an assignment of tasks to resources at minimal cost such that a fault tolerant requirement and reliability requirements of the system are satisfied. To achieve this goal, a key enabler of this method is to be able to evaluate S qualitatively and quantitatively with respect to fault tolerance. The method proposed automatically generates a fault tree to achieve the evaluation, thus, enabling the exploration of alternative architectures which are evaluated more quickly than current practical methods.

The problem that is addressed can be stated as follows. Let $F = f_1, f_2, ..., f_m$ be a set of m application tasks, $N = n_1, n_2, ..., n_q$ be a set of q signals between tasks, and $R = r_1, r_2, ..., r_k$ be a set of k resources. Also, let x_j be the number of resource type a_j for $j = 1, 2, ..., k$, d_{ij} for $i = 1, 2, ..., m + q$, and $j = 1, 2, ..., k$. Given an initial allocation of tasks to resources S, find a new allocation S, that minimizes

Resource Type	Failure Rate, λ (failures/hour)
Copper Wire	1.0×10^{-6}
Sensor Type 1	6.06×10^{-4}
Sensor Type 2	6.06×10^{-5}
Processor (ECU)	6.28×10^{-6}
Motor (Actuator)	7.90×10^{-7}
CAN Bus	2.6×10^{-7}
FlexRay Bus	8.75×10^{-4}

Table 5.1. *Estimated failure rates for resources in the automotive case study.*

$C_S(d_{ij}, xj)$ such that $\phi(S) < \phi$, is satisfied for the fault tolerant requirement, ϕ, the reliability that is required Q_{req}.

5.5.2 Data Sources

Data used to perform a quantitative analysis on the generated fault trees for the experimental vehicle described in the case study is presented. In particular, failure rate and failure probability estimates are summarized for the architecture resources in the case study. Table 5.1 illustrates the estimated failure rate in failures per hour under the assumption of an exponential distribution for the failure rate of components during components useful life. The estimate for the processor is obtained by assuming a basic failure rate from [85] and applying an environment factor for commercial vehicles. Sensor and motor estimates are from [17], the estimates are taken for mobile ground vehicles. Two types of rotary position sensors are used in the case study, so for the purpose of making quantitative comparisons as opposed to making an objective quantitative measure, it is assumed that one sensor is an order less reliable. The failure rates in Table 1 are to only give a coarse estimate of the system reliability, and more accurate measurements can be made with more accurate input data.

The bus failure rates are obtained from public literature based on the CAN bus protocol. Error detection and signaling mechanisms in the CAN bus protocol may fail when an inconsistent frame omission occurs [111]. An inconsistent frame omission is an error where the $(N-1)$ bit of an N bit end of frame delimiter is not detected, resulting in inconsistent frame omissions and inconsistent message duplicates. An inconsistent frame omission is the condition where messages can be delivered to receiving nodes in duplicates, and an inconsistent message duplicate is the condition where a message is delivered to only a subset of receiving bus nodes. Since architecture resources can fail silent in our fault tree generation framework,

we consider the probability of failure resulting from message omission errors in a CAN bus.

To the best of knowledge at the time of writing this thesis, failure data is not available in published literature for a FlexRay bus protocol. Hence, an estimate of the probability of an inconsistent message omission failure in FlexRay is obtained from measurements in for the Time Triggered CAN (TTCAN) bus protocol [31]. It is shown in [111] that when the sending node of a TTCAN message fails, its failure results in an inconsistent message omission. Since automatic retransmission of messages due to error does not occur in TTCAN or FlexRay protocols, the effect of not transmitting a frame is identical to the failure of a sending node.

5.5.3 Experimental Results

An abstraction of the Simulink model is given and used as the baseline architecture. It is expected that this system does tolerate single point failures in the resource architecture, and the top event is the failure for the actuators (motors) to receive any data. Two additional architectures are explored and used to compare. Since, the baseline architecture is highly redundant, the objective is to reduce the parts count while still maintaining the ability to tolerate single point failures in the hardware architecture.

- **Architecture 1**: This architecture contains one ECU, and on this ECU, each of the four application tasks is allocated.
- **Architecture 2**: This architecture contains a dual redundant configuration where an additional ECU is added to Architecture 1, and the additional ECU also has all four application tasks mapped to it.
- **Architecture 3**: This architecture is the baseline configuration. It contains four ECUs, and the four tasks described in the case study are allocated to each ECU.

A simplified illustration of sensor and actuator placement in the case study architectures is given in Figure 5.7. The figure illustrates how the sensors and actuators in this cases study model are distributed across the available architecture resources. Using the baseline configuration, Architecture 3, the minimal cut sets are used to validate that the configuration does satisfy the fault tolerant requirement. This is verified by inspection of the signals and blocks in the Simulink model of the system, and the resulting minimal cut sets are shown in Table 5.2. Each minimal cut set is numbered to distinguish between other minimal cut sets, and each minimal cut set contains a set of basic events which correspond to resource failures.

Minimal Cut Sets			
#	Set of Basic Events	#	Set of Basic Events
1	{ ecu3-fail, can2-fail}	33	{pos-sens2-fail, pos1-1-fail }
2	{ ecu4-fail, can2-fail}	34	{swa2-2-fail, pos1-1-fail }
3	{ ecu1-fail, can2-fail}	35	{motor2-fail, pos1-1-fail }
4	{ can2-fail, can1-fail}	36	{wire2-fail, pos1-1-fail }
5	{ can1-fail, pos-sens2-fail}	37	{can2-fail, swa1-1-fail }
6	{ ecu2-fail, can1-fail}	38	{ecu2-fail, swa1-1-fail }
7	{ pos1-sens-fail, can1-fail}	39	{pos1-sens-fail, swa1-1-fail }
8	{ ecu3-fail, can2-fail}	40	{pos-sens2-fail, swa1-1-fail }
9	{ wire2-fail, can1-fail}	41	{swa2-2-fail, swa1-1-fail }
10	{ ecu3-fail, ecu2-fail}	42	{motor2-fail, swa1-1-fail }
11	{ ecu4-fail, ecu2-fail}	43	{wire2-fail, swa1-1-fail }
12	{ ecu1-fail, ecu2-fail}	44	{can2-fail, motor1-fail }
13	{ swa2-2-fail, ecu1-fail}	45	{can2-fail, wire1-fail }
14	{ wire2-fail, ecu1-fail}	46	{can1-fail, motor2-fail }
15	{ ecu3-fail, pos1-sens-fail}	47	{ecu3-fail, motor2-fail }
16	{ ecu4-fail, pos1-sens-fail}	48	{pos1-sens-fail, pos1-1-fail }
17	{ ecu1-fail, pos1-sens-fail}	49	{ecu4-fail, motor2-fail }
18	{ ecu3-fail, pos-sens2-fail}	50	{ecu4-fail, wire2-fail }
19	{ ecu4-fail, pos-sens2-fail}	51	{ecu2-fail, motor1-fail }
20	{ ecu1-fail, pos-sens2-fail}	52	{ecu2-fail, wire1-fail }
21	{ ecu3-fail, swa2-2-fail}	53	{ecu1-fail, motor2-fail }
22	{ ecu4-fail, swa2-2-fail}	54	{pos1-sens-fail, motor1-fail }
23	{ can2-fail, pos1-sens-fail}	55	{pos1-sens-fail, wire1-fail }
24	{ ecu2-fail, pos1-sens-fail}	56	{pos-sens2-fail, motor1-fail }
25	{ pos1-sens-fail, pos1-sens-fail}	57	{pos-sens2-fail, wire1-fail }
26	{ pos-sens2-fail, pos1-sens-fail}	58	{swa2-2-fail, motor1-fail }
27	{ swa2-2-fail, pos1-sens-fail}	59	{wire1-fail, motor2-fail }
28	{ motor2-fail, pos1-sens-fail}	60	{swa2-2-fail, wire1-fail }
29	{ wire2-fail, pos1-sens-fail}	61	{wire2-fail, motor1-fail }
30	{ can2-fail, pos1-1-fail}	62	{motor1-fail, motor2-fail }
31	{ ecu2-fail, pos1-1-fail}	63	{wire2-fail, wire1-fail }
32	{pos1-sens-fail, pos1-1-fail}		

Table 5.2. *Minimal cut sets for the baseline architecture in automotive case study.*

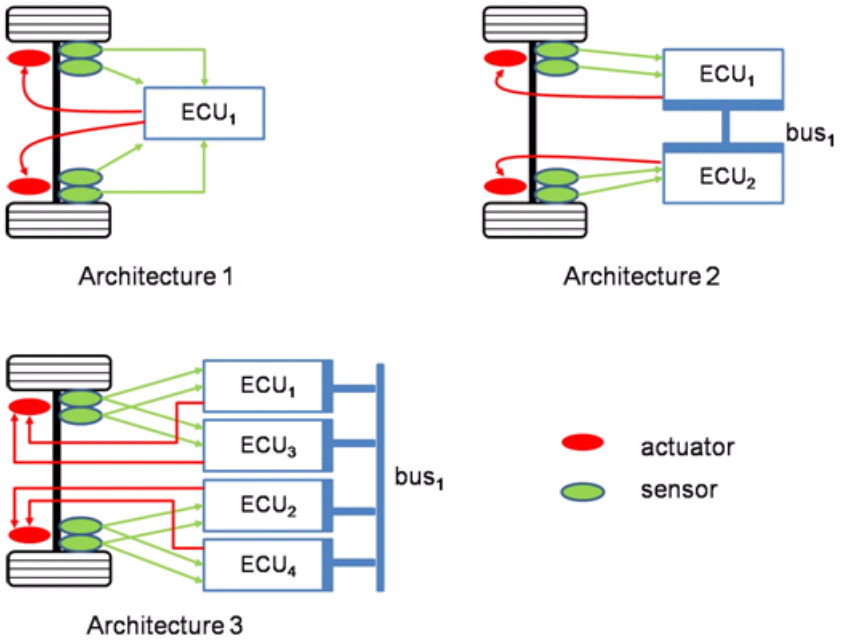

Figure 5.7. *Topology of the architectures of the automotive case study.*

In Table 5.2, there is no minimal cut set with a single basic event, thus, the design satisfies the fault tolerant requirement that there are no single point of failures in the architecture. Furthermore, it is shown from these results that failure to the FlexRay bus does not appear in the list of minimal cut sets. This results from masking of omission errors detected by the application using voting strategies and application redundancy.

Then, an investigation occurred on how the architectures would compare to one another based on their reliability by comparing architectures when given the initial failure data estimates in Table 5.1. The purpose is to observe the impact of the most critical resources, as determined by a relative ranking of the importance measures for each architecture in Figure 5.3. The ranking is relative to the other resources in the design model. The Birnbaum importance measure is represented in the table for each architecture. Architecture 2 is the result of the replication of those resources that have the higher importance values in Architecture 1, and Architecture 3 is a measure of the baseline architecture. Across each of the measured architectures, one can observe that failure of the sensor resources are most critical based on the architecture configuration and estimated failure rate of the sensors. The events that correspond to failure of sensor resources include "pos1-sens-fail", "pos-sens2-fail", and "pos1-1-fail". Thus, one method of improvement is to replace the sensor with more reliable components that can achieve the same function or by adding redundant sensors. The importance measures drive the exploration process from the single architecture to the dual ECU architecture by replicating the most

Birnbaum Rankings	Arch1	Arch2	Arch3
1	pos1-sens-fail	pos1-sens-fail	pos1-sens-fail
2	pos-sens2-fail	pos-sens2-fail	pos-sens2-fail
3	ecu2-fail	pos1-1-fail	pos1-1-fail
4	can1-fail	swa1-1-fail	swa1-1-fail
5	can2-fail	swa2-2-fail	swa2-2-fail

Table 5.3. *A ranking of resource failures as basic events according to the Birnbaum importance measure.*

critical resources relative to other resources in the model. One can observe that the importance measures are dependent on the failure distribution of the resource and its position in the topology of the architectures, according to Equation 2.13.

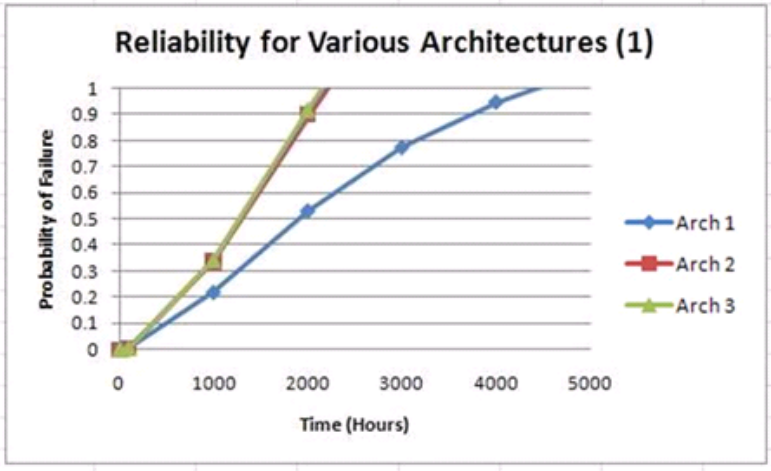

Figure 5.8. *Reliability of automotive case study architectures for initial failure data estimates.*

The results in Figure 5.8 and Figure 5.9 show that a substantial increase in reliability for the different architectures can be obtained when the failure rate of the sensor values are decreased by a factor of 10-1 failures per hour since the sensors were most critical consistently across Architectures 1, 2, and 3 according to the Birnbaum importance. This implies that improvements in the systems reliability can be achieved by replacing architecture resources with resources that perform the same functionality at a lower failure rate. This of course is assumed to come at a higher cost, but it adds another degree of freedom that may be exploited by the designer. Furthermore, it aids in the choice of off-the-shelf components that may be used in the system.

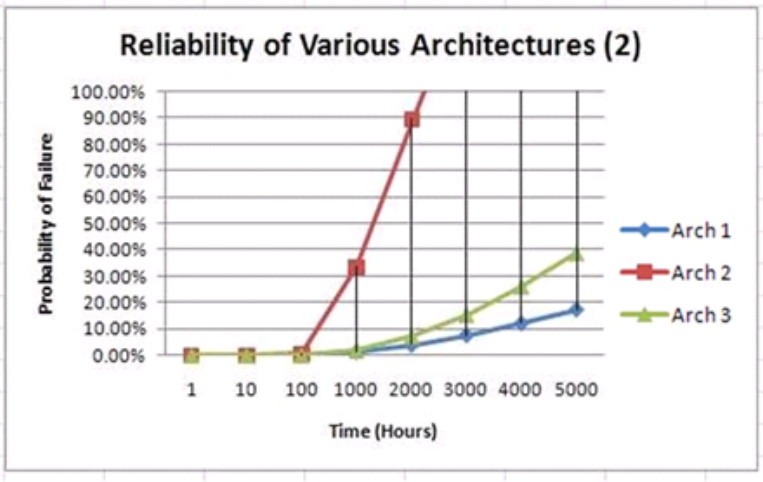

Figure 5.9. *Reliability of automotive case study architectures after improving failure data estimates.*

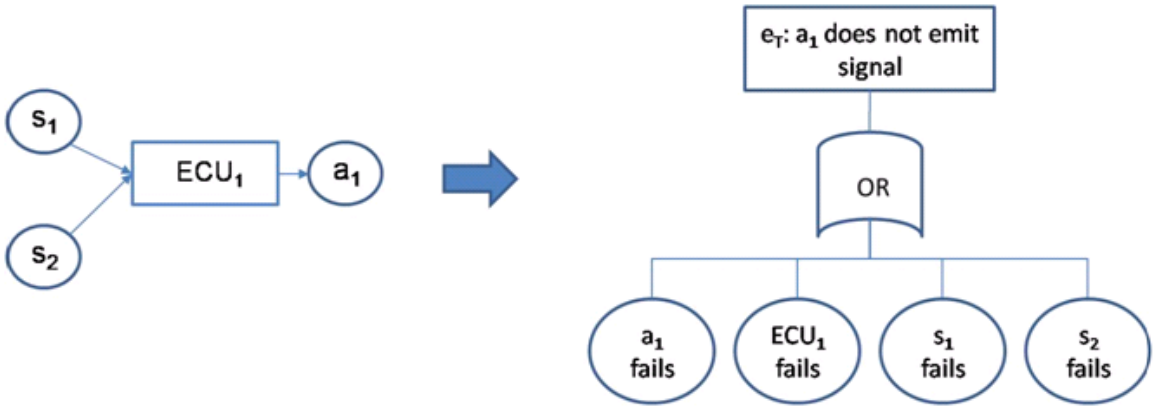

Figure 5.10. *Sensor and actuator topology with a single ECU architecture.*

Figure 5.10 shows a single ECU architecture that takes input signals from two independent sensor devices s_1 and s_2, and after processing the input signals, an output signal is emitted to the actuator a_1. The sensors are needed for the a_1 to emit a signal. Assume that the ECU contains a set of tasks that communicates with one another via shared memory, since the tasks are all on the same ECU. Furthermore, assume the ECU, sensor resources, and actuator resource can fail silently. The fault tree that is generated is given in the figure. Let the probability of failure be 0.10 for each resource. Then, from the generated fault tree and the equations for computing the probability of the top event, the probability of failure for the top event in this architecture can be approximated as 0.40. This results in a reliability of 60%.

Consider the architecture shown in Figure 5.11 where ECU2 is introduced as a replica of ECU1. This means that ECU2 contains the same set of tasks as ECU1 and

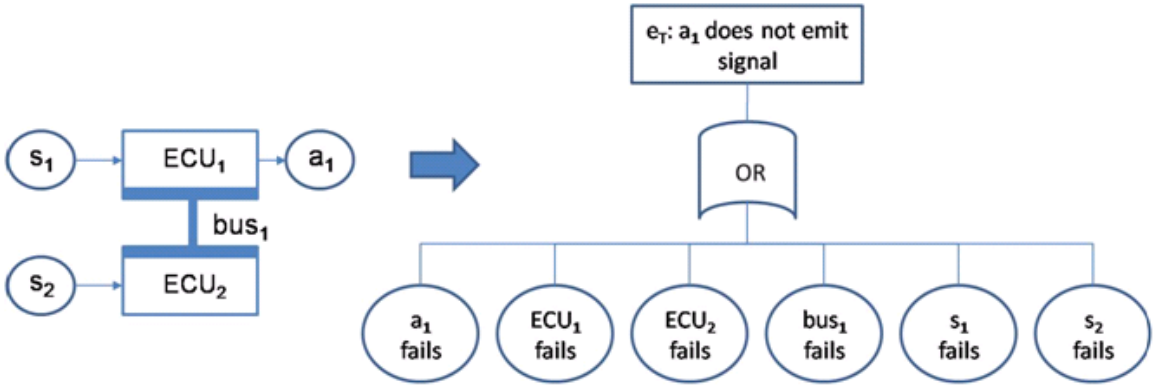

Figure 5.11. *Sensor and actuator topology with a dual ECU architecture.*

FTGen	Arch1	Arch2	Arch3
# Model Objects (Blocks/Signals)	62	176	239
# Fault Events	114	299	399
Time to Construct Fault Tree (milliseconds)	1272	1431	1563

Table 5.4. *Fault tree construction time for the FTGen tool.*

the communication between tasks on each ECU are transmitted via BUS1. Sensor s_2 is directly connected to ECU2 as compared to Figure 5.10. Therefore, to emit a signal at the actuator a_1, signals from sensor s_2 are required to be received by ECU1 over the bus. Based on the same assumptions that are previously made, the fault tree that is generated is shown in Figure 5.11. An analysis reveals that the reliability is approximately 50% given that each resource has a failure probability of 0.10. Therefore, the reliability is lower for the dual ECU configuration. By observing the simplified architectures of the case study in Figures 5.7, one can see a similar architecture as in the example. One can deduce that the architectures in the case study exhibit a similar characteristic when compared to one another.

In this design flow, it is important to be able to create, evaluate, and modify the design quickly. So, it is of interest to know how the tools in our tool chain measured up with the case study architecture models. Table 5.5.3 shows the size of events that are created from the architecture models of the case study. It is observed that the time to automatically generate a fault tree is approximately linear in the size of the model. The results also suggests that if the same model abstraction is used by an analyst as is read by the automatic fault tree generation tool, then the fault tree generation tool creates a fault tree significantly faster than by hand, which for Architecture 3 as an example, could take on the order of days to generate manually.

Analysis Tools	Arch1	Arch2	Arch3
# Cut Sets	15	56	63
Galileo Analysis Time (seconds)	4	61	n/a
FaultTree+ Analysis Time (seconds)	0.3	1.1	1.7
FTGen Analysis Module (seconds)	0.2	1.2	2.1

Table 5.5. *A comparison on the time (in seconds) to analyze the fault trees for the given architectures in the automotive case study.*

Table 5.5 shows how the fault tree analysis tools that are used in this case study, perform on these architecture models. The tools that were used include Galileo, FaultTree+, and the FTGen analysis module, which is a self-developed tool. The number of cut sets for each tool is the same when no compaction is used (this is specific to the FaultTree+ tool since it uses a method to analyze fault trees more compactly when selected). The steps involved in the fault tree analysis includes parsing the input fault tree after importing into the tool, generating the cut sets, and performing an importance analysis and reliability analysis of the imported fault tree. In the table, it can be observed that the FaultTree+ tool, as expected, performs faster than the Galileo tool as the number of cut sets that are generated by each tool increases. Although the quantitative results of each tool were similar, the discrepancy in the time it takes to acquire the cut sets and quantitative results is due to the accuracy of the quantitative analysis algorithm and possible data structures used by each tool. Observe that for Architecture 3, the Galileo tool was unable to complete the cut set analysis, so the Galileo tool was aborted after 20 minutes and not results were obtained. Both tools provide interfaces to import and export a fault tree, but in each tool the analyst is still required to do some manual tasks such as open up an application interface and import the generated fault tree. The analysis module in FTGen performed relatively well to FaultTree+ in each architecture.

5.5.4 Discussion

The results demonstrate two basic advantages of the method: first, the fault tree is generated automatically and thus quickly and consistently with the system model as compared to manually. Secondly, we are able to show that the system in the case study exhibits no single point of failure for the top failure event to take place. Again, while this is done by visual inspection, this step can be easily automated. Therefore, both the automatic generation and the assessment of cut sets could be included in automated design exploration.

Limitations to using a fault tree analysis are exhibited by this approach. Quantitative results are not objective. They are interpreted to be relative to the accuracy

of the input data to the fault tree analysis. Therefore, this method is most useful to assist in making architectural decisions early in the design phases, support safety cases in conjunction with other reliability and safety analysis, and to validate fault tolerant requirements in the architecture.

5.6 Summary

This chapter presented the set of design tools that are implemented to carry out the design flow based on the construction and analysis of a fault tree. The design flow provides a way to explore different design solutions while integrating fault tree construction and analysis into the design flow. The design flow and the tools described are used to explore alternative architectures for a distributed steer-by-wire model. The design flow captures the different steps and tools for each step within the design methodology that is introduced.

Chapter 6

Conclusions

The design for fault tolerant, distributed embedded systems is an increasingly complex task and it is difficult to manage with current practices. In this dissertation, a methodology and a set of tools to support the design distributed embedded systems with fault tolerant and reliability requirements is presented. The methodology is based on the platform based design method. This methodology allows for a separation in the modeling of the function and architecture, as a mapping step in the design method allows for function tasks to be allocated onto architecture resources.

6.1 Benefits of Proposed Methodology

The example case studies that are presented highlights the benefits of a methodology that integrates reliability modeling with fault trees into a system level design framework. Unlike previous work that relies on simple structure models to select components within a complex system that should be replicated, this work considers the design of embedded systems. This means that there are hardware and software components that must interact together. Moreover, this design flow considers how failures to components within a distributed embedded system will effect the state of the failure state of the system. This includes the ability to measure the effects of fault propagation using minimal cut sets concept and the ability to quantify the failure distribution of system components. The method aides the designer in making choices on the number and type of components that should benefit from added redundancy or improved reliability. This is useful in a system level design as the

performance of the system as a whole can be impacted and suffer from the faults that propagate into system failures.

In practice, the automatic fault tree generation can be used as a tool to aide the designer to make high level trade-off decisions in the architecture design of a distributed embedded system. Fault trees are very popular in industry as they are used to make safety cases and provide evidence to support reliable and fault tolerant designs. Currently, fault tree are still performed much by hand in practice, and the tool FTGen would be a good aid to system designers as it would allow for quicker, more efficient construction of fault trees.

6.2 Drawbacks

This approach to the design of fault tolerant systems in distributed architectures has some drawbacks. The fault tree model is good for viewing the failure scenarios of a system based on a presumed fault model that can be defined in terms of several failure modes for a single component. But, the fault tree model inherently has some shortcomings. In this work, the fault tree is said to be static, i.e. it does not consider priority of events or where order of events are important. The use of a static fault tree models only simultaneous failures for system structures where the dependencies between components are static, i.e. the topology does not change. Static fault trees are not suited to modeling repair processes, and although the fault tree in this work is generated automatically, the drawback is that the size of the generated tree can be very large depending on the number of components, dependencies between components, and level of detail from which the tree is generated. Finally, this work is limited to omission failures, whereas, value and timing failures would be interesting to investigate.

6.3 Future Directions

Directions that are of interest in moving this work along can be targeted towards the following areas:

- *Support for state based reliability modeling.* This work considers the use of fault trees as the reliability model of choice, however, other notable models are state-based. In particular, Markov models and Bayesian inference models are making ground in their use as reliability models. These models are useful to model state transitions that may not be correct according to a design specification. These models allow the designer to capture different modes and

states within a complex system that data flow oriented models may not be able to capture.

- *Software reliability, quantification, and estimation.* Given that more software is used in systems and many reliability models originated to address hardware failures, there is a need for more research in the area of software reliability prediction, reliability modeling for software using traditional fault trees has been a topic of interest. Even more so has been the prediction of software reliability. This area is gaining attention as system designers need more accurate ways of predicting software failures. It is challenging to predict and estimate software failures, and as such, the current body of work in this area predominantly relies on the use of previous software projects that are similar in size and structure. As software becomes more important in safety critical systems, the ability to predict and estimate failures more accurately would add value to the area.

Bibliography

[1] J. Aplin, "Primary flight computers for the boeing 777," *Microprocessors and Microsystems*, vol. 20, no. 8, pp. 473 – 478, 1997. [Online]. Available: http://www.sciencedirect.com/science/article/B6V0X-4GW41CM-4/2/15ff064161dbc81c71c28316605e931d

[2] A. Avižienis, "Design of fault-tolerant computers," in *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference*, ser. AFIPS '67 (Fall). New York, NY, USA: ACM, 1967, pp. 733–743. [Online]. Available: http://doi.acm.org/10.1145/1465611.1465708

[3] A. Avižienis, "Fault-tolerance and fault-intolerance: Complementary approaches to reliable computing," in *Proceedings of the International Conference on Reliable Software*. New York, NY, USA: ACM, 1975, pp. 458–464.

[4] A. Avizienis, "The n-version approach to fault-tolerant software," *Software Engineering, IEEE Transactions on*, vol. SE-11, no. 12, pp. 1491–1501, December 1985.

[5] M. O. Ball, "Computational complexity of network reliability analysis: An overview," *Reliability, IEEE Transactions on*, vol. 35, no. 3, pp. 230 –239, 1986.

[6] P. F. Barlow, Richard E., "Importance of system components and fault tree events," *Stochastic Processes and their Applications*, vol. 3, no. 2, pp. 153–173, 1975, cited By (since 1996) 22. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-0000695488&partnerID=40&md5=df564d760e30615618a4c6c2a18e480d

[7] R. E. Barlow and F. Proschan, *Statistical Theory of Reliability and Life Testing: Probability Models*. Holt, Rinehart, and Winston, 1975.

[8] M. Bellotti and R. Mariani, "How future automotive functional safety requirements will impact microprocessors design," *Microelectronics Reliability*, vol. 50, no. 9-11, pp. 1320 – 1326, 2010, 21st European Symposium on the Reliability of Electron Devices, Failure Physics and

Analysis. [Online]. Available: http://www.sciencedirect.com/science/article/B6V47-50P47VD-3/2/4c60f4b8ba1792de6fee98df0dec4213

[9] Z. W. Birnbaum, "On the importance of different components in a multicomponent system," University of Washington, Seattle, Washington, Technical Report 54, May 1968.

[10] A. Birolini, *Reliability Engineering: Theory and Practice*, 6th ed. Springer, 2010.

[11] M. Born, J. Favaro, and O. Kath, "Application of iso dis 26262 in practice," in *CARS '10: Proceedings of the 1st Workshop on Critical Automotive applications*. New York, NY, USA: ACM, 2010, pp. 3–6.

[12] A. Bossche, "Computer-aided fault tree synthesis i (system modeling and causal trees)," *Reliability Engineering and System Safety*, vol. 32, no. 3, pp. 217 – 241, 1991. [Online]. Available: http://www.sciencedirect.com/science/article/B6V4T-4829XT1-11/2/fa07e9ff0ab5ea044d19a4bb7ce174d1

[13] B. Bouyssounouse and J. Sifakis, *Embedded Systems Design: The ARTIST Roadmap for Research and Development (Lecture Notes in Computer Science)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[14] R. W. Brower, "Lockheed f-22 raptor," in *The Avionics Handbook*, C. R. Spitzer, Ed. CRC Press, 2001.

[15] L. P. Carloni, F. De Bernardinis, C. Pinello, A. Sangiovanni-Vincentelli, and M. Sgroi, "Platform-based design for embedded systems," in *The Embedded Systems Handbook*, R. Zurawski, Ed. CRC Press, 2005.

[16] A. Carpignano and A. Poucet, "Computer assisted fault tree construction: a review of methods and concerns," *Reliability Engineering and System Safety*, vol. 44, no. 3, pp. 265 – 278, 1994, special Issue On Advanced Computer Applications. [Online]. Available: http://www.sciencedirect.com/science/article/B6V4T-4829Y10-4K/2/5f2cd80e2994579869655d362bafc78a

[17] R. A. Center, *Nonelectronic Parts Reliability Data*, 1985.

[18] W. Cesário, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, and M. Diaz-Nava, "Component-based design approach for multicore socs," in *DAC '02: Proceedings of the 39th annual Design Automation Conference*. New York, NY, USA: ACM, 2002, pp. 789–794.

[19] D. Cesiel, M. C. Gaunt, and B. Daugherty, "Development of a steer-by-wire system for the gm sequel," in *Society of Automotive Engineers (SAE) 2006 World Congress & Exhibition*, April 2006.

[20] R. N. Charette, "This car runs on code," *IEEE Spectrum*, February 2009.

[21] P. Chatterjee, "Fault tree analysis: Min cut set algorithms," Operations Research Center, University of California, Berkeley, Tech. Rep. ADO774100, 1974.

[22] M.-S. Chern, "On the computational complexity of reliability redundancy allocation in a series system," *Operations Research Letters*, vol. 11, no. 5, pp. 309 – 315, 1992. [Online]. Available: http://www.sciencedirect.com/science/article/B6V8M-48MPT46-9G/2/bc19188fc360cfbbd36628d33fb0796d

[23] S. Chien, R. Sherwood, D. Tran, B. Cichy, G. Rabideau, R. Castao, and A. Davies, "The autonomous sciencecraft embedded systems architecture," in *In IEEE Int. Conf. on Systems, Man and Cybernetics*, 2005, pp. 3927–3932.

[24] P. Chou, R. B. Ortega, and G. Borriello, "Interface co-synthesis techniques for embedded systems," in *ICCAD '95: Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*. Washington, DC, USA: IEEE Computer Society, 1995, pp. 280–287.

[25] I. Crnkovic, "Component-based approach for embedded systems," in *In Proceedings of 9th International Workshop on Component-Oriented Programming*, 2004.

[26] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[27] R. C. De Vries, "An automated methodology for generating a fault tree," *Reliability, IEEE Transactions on*, vol. 39, no. 1, pp. 76 –86, 1990.

[28] J. B. Dugan, S. J. Bavuso, and M. A. Boyd, "Dynamic fault-tree models for fault-tolerant computer systems," *Reliability, IEEE Transactions on*, vol. 41, no. 3, pp. 363–377, Sept. 1992.

[29] N. M. Enache, M. Netto, S. Mammar, and B. Lusetti, "Driver steering assistance for lane departure avoidance," *Control Engineering Practice*, vol. 17, no. 6, pp. 642 – 651, 2009.

[30] A. Ferrari and A. Sangiovanni-Vincentelli, "System design: traditional concepts and new paradigms," 1999, pp. 2 –12.

[31] J. Ferreira, "An experiment to assess bit error rate in can," in *In Proceedings of 3rd International Workshop of Real-Time Networks (RTN2004*, 2004, pp. 15–18.

[32] J. Fussell, "How to hand-calculate system reliability and safety characteristics," *Reliability, IEEE Transactions on*, vol. R-24, no. 3, pp. 169 –174, Aug. 1975.

[33] J. Fussell and W. Vesely, "A new methodology for obtaining cut sets for fault trees," *Transactions American Nuclear Society*, vol. 15, no. 1, pp. 262–263, 1972.

[34] J. B. Fussell, "Synthetic tree model: a formal methodology for fault tree construction," Ph.D. dissertation, Georgia Institute of Technology, December 1972.

[35] D. D. Gajski, R. Domer, J. Peng, and A. Gerstlauer, *System Design: A Practical Guide with Specc.* Norwell, MA, USA: Kluwer Academic Publishers, 2001.

[36] ——, *System Design: A Practical Guide with Specc.* Norwell, MA, USA: Kluwer Academic Publishers, 2001.

[37] R. L. Glass, "Persistent software errors," *IEEE Transactions on Software Engineering*, vol. 7, no. 2, pp. 162–168, 1981.

[38] ——, "Sorting out software complexity," *ACM Communications*, vol. 45, no. 11, pp. 19–21, 2002.

[39] G. Gössler and J. Sifakis, "Composition for component-based modeling," *Science of Computer Programming*, vol. 55, no. 1-3, pp. 161–183, 2005.

[40] R. K. Gupta and G. De Micheli, "System synthesis via hardware-software co-design," Stanford, CA, USA, Tech. Rep., 1992.

[41] R. Hammett, "Flight-critical distributed systems: design considerations," *Aerospace and Electronic Systems Magazine, IEEE*, vol. 18, no. 6, pp. 30–36, June 2003.

[42] S. Heath, *Embedded Systems Design.* Newton, MA, USA: Butterworth-Heinemann, 2002.

[43] I. Incorporated. (2010, October) Fault tree analysis software: Faulttree+. [Online]. Available: http://www.isograph-software.com/

[44] International Technology Roadmap for Semiconductors. (2009) Executive summary. [Online]. Available: http://www.itrs.net/Links/2009ITRS/2009Chapters_2009Tables/2009_ExecSum.pdf

[45] R. Isermann, R. Schwarz, and S. Stölzl, "Fault-tolerant drive-by-wire systems," *Control Systems Magazine, IEEE*, vol. 22, no. 5, pp. 64–81, oct. 2002.

[46] R. Isermann, "Mechatronic systems–innovative products with embedded control," *Control Engineering Practice*, vol. 16, no. 1, pp. 14 – 29, 2008.

[47] R. Jafari, "Medical embedded systems," PhD Dissertation, University of California, Los Angeles, 2006.

[48] A. Jhumka, S. Klaus, and S. A. Huss, "A dependability-driven system-level design approach for embedded systems," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 372–377.

[49] S.-N. Ju, C.-L. Chen, and C.-T. Chang, "Constructing fault trees for advanced process control systems application to cascade control loops," *Reliability, IEEE Transactions on*, vol. 53, no. 1, pp. 43 – 60, 2004.

[50] R. K. Jurgen, *Distributed Automotive Embedded Systems*, 2007.

[51] L. M. Kaufman, S. Bhide, and B. W. Johnson, "Modeling of common-mode failures in digital embedded systems," January 2000, pp. 350 –357.

[52] A. Kaufmann, D. Grouchko, and R. Cruon, *Mathematical Models for the Study of the Reliability of Systems*, ser. Mathematics in Science and Engineering. New York: Academic Press, 1977, vol. 124.

[53] C. F. Kemerer and M. C. Paulk, "The impact of design and code reviews on software quality: An empirical study based on psp data," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 534–550, 2009.

[54] K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. L. Sangiovanni-Vincentelli, "System-level design: orthogonalization of concerns and platform-based design," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 19, no. 12, pp. 1523–1543, December 2000.

[55] B. Kienhuis, E. Deprettere, P. van der Wolf, and K. Vissers, "A methodology to design programmable embedded systems," in *Embedded Processor Design Challenges*, ser. Lecture Notes in Computer Science, E. Deprettere, J. Teich, and S. Vassiliadis, Eds. Springer Berlin / Heidelberg, 2002, vol. 2268, pp. 321–324.

[56] B. Kienhuis, E. Deprettere, K. Vissers, and P. Van Der Wolf, "An approach for quantitative analysis of application-specific dataflow architectures," in *Application-Specific Systems, Architectures and Processors, 1997. Proceedings., IEEE International Conference on*, July 1997, pp. 338–349.

[57] J. C. Knight, "Safety critical systems: challenges and directions," in *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*. New York, NY, USA: ACM, 2002, pp. 547–550.

[58] H. Kopetz, "The complexity challenge in embedded system design," May 2008, pp. 3–12.

[59] K. Kumar Vemuri, J. B. Dugan, and K. J. Sullivan, "A design language for automatic synthesis of fault trees," in *Reliability and Maintainability Symposium, 1999. Proceedings. Annual*, 1999, pp. 91 –96.

[60] W. Kuo, V. R. Prasad, F. A. Tillman, and C.-L. Hwang, *Optmial Reliability Design: Fundamentals and Applications*. United Kingdom: Cambridge University Press, 2000.

[61] W. Kuo and R. Wan, "Recent advances in optimal reliability allocation," in *Computational Intelligence in Reliability Engineering*, ser. Studies in Computational Intelligence, G. Levitin, Ed. Springer Berlin/Heidelberg, 2007, vol. 39, pp. 1–36.

[62] S. A. Lapp and G. J. Powers, "Computer-aided synthesis of fault-trees," *Reliability, IEEE Transactions on*, vol. R-26, no. 1, pp. 2–13, 1977.

[63] J.-C. Laprie, "Dependable computing and fault tolerance : Concepts and terminology," June 1995, p. 2.

[64] G. Le Lann, "The ariane 5 flight 501 failure - a case study in system engineering for computing systems," INRIA, Research Report RR-3079, 1996. [Online]. Available: http://hal.inria.fr/inria-00073613/en/

[65] E. A. Lee, "A denotational semantics for dataflow with firing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/ERL M97/3, 1997, a copy may be found at ¡a href="http://ptolemy.eecs.berkeley.edu/publications/papers/97/dataflow/"¿http://ptolemy.eecs [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1997/3167.html

[66] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," in *Proceedings of the IEEE*, vol. 75, no. 9, September 1987, pp. 1235–1245.

[67] E. A. Lee and S. Neuendorffer, *Actor-oriented models for codesign: balancing re-use and performance*. Norwell, MA, USA: Kluwer Academic Publishers, 2004, pp. 33–56. [Online]. Available: http://portal.acm.org/citation.cfm?id=1137958.1137961

[68] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 17, no. 12, pp. 1217–1229, December 1998.

[69] P. A. Lee and T. Anderson, *Fault Tolerance: Principles and Practice*, J. C. Laprie, A. Avizienis, and H. Kopetz, Eds. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1990.

[70] N. G. Leveson and C. S. Turner, "An investigation of the therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, 1993.

[71] G. Levitin, "Reliability and performance analysis of hardware-software systems with fault-tolerant software components," *Reliability Engineering and System Safety*, vol. 91, no. 5, pp. 570 – 579, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/B6V4T-4GGWG9B-1/2/591e11df6d8be7d02b73db861b8d1e28

[72] A. Lie, C. Tingvall, M. Krafft, and A. Kullgren, "The effectiveness of electronic stability control (esc) in reducing real life crashes and injuries," *Traffic Injury Prevention*, vol. 7, no. 1, pp. 38–43, 2006.

[73] B. Littlewood and D. R. Miller, "Conceptual modeling of coincident failures in multiversion software," *Software Engineering, IEEE Transactions on*, vol. 15, no. 12, pp. 1596 –1614, Dec. 1989.

[74] X. Liu, "Semantic foundation of the tagged signal model," Ph.D. dissertation, EECS Department, University of California, Berkeley, December 2005. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2005/EECS-2005-31.html

[75] M. Lukasiewycz, M. Glaß, and J. Teich, "Exploiting data-redundancy in reliability-aware networked embedded system design," in *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, ser. CODES+ISSS '09. New York, NY, USA: ACM, 2009, pp. 229–238. [Online]. Available: http://doi.acm.org/10.1145/1629435.1629468

[76] R. R. Lutz, "Analyzing software requirements errors in safety-critical, embedded systems," *IEEE Transactions on Software Engineering*, 1993.

[77] M. R. Lyu, Ed., *Software Fault Tolerance*. John Wiley and Sons, Inc., 1989. [Online]. Available: http://www.cse.cuhk.edu.hk/~lyu/book/sft/index.html

[78] H. Man, J. Rabaey, P. Six, and L. Claesen, "Cathedral-ii: A silicon compiler for digital signal processing," *IEEE Design and Test of Computers*, vol. 3, pp. 13–25, 1986.

[79] Mathworks, *Real-Time Workshop version 7.6*, Natick, MA, U.S.A., September 2010. [Online]. Available: http://www.mathworks.com/products/rtw/

[80] ——, *Simulink version 7.6*, Natick, MA, September 2010. [Online]. Available: http://www.mathworks.com/products/simulink/

[81] T. K. Moon, *Error Correction Coding*. New Jersey: John Wiley and Sons, Inc., 2005. [Online]. Available: http://www.cse.cuhk.edu.hk/~lyu/book/sft/index.html

[82] G. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, no. 19, April 1965.

[83] K. Nakashima and Y. Hattori, "An efficient bottom-up algorithm for enumerating minimal cut sets of fault trees," *Reliability, IEEE Transactions on*, vol. R-28, no. 5, pp. 353–357, 1979.

[84] R. Obermaisser, P. Peti, and F. Tagliabo, "An integrated architecture for future car generations," *Real-Time Systems*, vol. 36, no. 1-2, pp. 101–133, 2007.

[85] U. D. of Defense, "Military handbook reliability prediction of electronic equipment (mil-hdbk-217f)," 1991.

[86] G. A. Office, "Patriot missile defense: Software problem led to system failure at dhahran, saudi arabia," United States General Accounting Office, Washington, D. C., Report GAO/IMTEC-92-26, February 1992. [Online]. Available: http://www.fas.org/spp/starwars/gao/im92026.htm

[87] S. Osaki, "Performance/reliability measures for fault-tolerant computing systems," *Reliability, IEEE Transactions on*, vol. R-33, no. 4, pp. 268 –271, October 1984.

[88] Y. Papadopoulos and C. Grante, "Evolving car designs using model-based automated safety analysis and optimisation techniques," *Journal on System Software*, vol. 76, no. 1, pp. 77–89, 2005.

[89] Y. Papadopoulos and M. Maruhn, "Model-based synthesis of fault trees from matlab-simulink models," in *Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, ser. DSN '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 77–82. [Online]. Available: http://portal.acm.org/citation.cfm?id=647882.738074

[90] A. Partnership. (2010) Autosar: Automotive open system architecture. [Online]. Available: http://www.autosar.org

[91] D. A. Peled, D. Gries, and F. B. Schneider, Eds., *Software reliability methods*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2001.

[92] C. Perrow, *Normal Accidents*. Princeton University Press, September 1999.

[93] W. Pierce, *Failure-tolerant Computer Design*. Academic Press, 1965.

[94] C. Pinello, "Design of safety-critical applications, a synthesis approach," Ph.D. dissertation, EECS Department, University of California, Berkeley, December 2004.

[95] C. Pinello, L. P. Carloni, and A. L. Sangiovanni-vincentelli, "Fault-tolerant deployment of embedded software for cost-sensitive real-time feedback-control applications," in *In Procs. of Design Automation and Test in Europe*. IEEE Computer Society, 2004.

[96] C. Pinello, L. P. Carloni, and A. L. Sangiovanni-Vincentelli, "Fault-tolerant distributed deployment of embedded control software," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, no. 5, pp. 906–919, May 2008.

[97] A. Pinto, L. P. Carloni, and A. L. Sangiovanni-Vincentelli, "A communication synthesis infrastructure for heterogeneous networked control systems and its application to building automation and control," in *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*. New York, NY, USA: ACM, 2007, pp. 21–29.

[98] P. Pop, P. Eles, and Z. Peng, *Analysis and Synthesis of Distributed Real-Time Embedded Systems*, 2004.

[99] P. Pop, P. Eles, Z. Peng, and T. Pop, "Analysis and optimization of distributed real-time embedded systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 11, no. 3, pp. 593–625, 2006.

[100] B. Popa. (2009, September) Volvo recalls 26,000 cars worldwide due to faulty software. [Online]. Available: http://www.autoevolution.com/news/volvo-recalls-26000-cars-worldwide-due-to-faulty-software-10772.html

[101] J. Radatz, *The IEEE Standard Dictionary of Electrical and Electronics Terms*. New York, NY, USA: IEEE Standards Office, 1997.

[102] K. Rajagopalan. (2010, February) Electronics causing problems for toyota, ford and chevrolet. [Online]. Available: http://www.frost.com/prod/servlet/market-insight-top.pag?docid=192381791

[103] B. Randell, "System structure for software fault tolerance," *SIGPLAN Not.*, vol. 10, pp. 437–449, April 1975. [Online]. Available: http://doi.acm.org/10.1145/390016.808467

[104] M. Rausand and A. Høyland, *System Reliability Theory: Models, Statistical Methods and Applications Second Edition*. Wiley-Interscience, 2003.

[105] A. Rauzy, "New algorithms for fault trees analysis," *Reliability Engineering and System Safety*, vol. 40, no. 3, pp. 203 – 211, 1993. [Online]. Available: http://www.sciencedirect.com/science/article/B6V4T-481DTN8-6K/2/d65a42d9c259fa4242ef3b757f5539d7

[106] A. L. Reibman and M. Veeraraghavan, "Reliability modeling: An overview for system designers," *Computer*, vol. 24, pp. 49–57, April 1991. [Online]. Available: http://portal.acm.org/citation.cfm?id=103287.103291

[107] D. A. Reynolds and G. Metze, "Fault detection capabilities of alternating logic," *Computers, IEEE Transactions on*, vol. C-27, no. 12, pp. 1093 –1098, December 1978.

[108] K. Rowe. (2008, October) Planning to fail: Current project management and system architecture specification activities causing delays in software engineering and system integration. [Online]. Available: http://ontargetembedded.blogspot.com/2008/10/planning-to-fail.html

[109] K. Rowe. (2010, February) Time to market is a critical consideration: Being first is key, unless you can be substantially better. being third means being out of luck. [Online]. Available: http://www.eetimes.com/discussion/guest-editor/4027610/Time-to-market-is-a-critical-consideration

[110] RTI, "The economic impacts of inadequate infrastructure for software testing," Health, Social, and Economics Research, Research Triangle Park, NC 27709, Research Report for the National Institute of Standards, 2002.

[111] J. Rufino, P. Veríssimo, G. Arroz, C. Almeida, and L. Rodrigues, "Fault-tolerant broadcasts in can," in *Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, ser. FTCS '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 150–. [Online]. Available: http://portal.acm.org/citation.cfm?id=795671.796875

[112] B. Rumpler, "Complexity management for composable real-time systems," *Object-Oriented Real-Time Distributed Computing, IEEE International.*

[113] A. H. Salek, J. Lou, and M. Pedram, "An integrated logical and physical design flow for deep submicron circuits," *IEEE Transactions on CAD*, vol. 18, pp. 1305–1315, 1999.

[114] A. Sangiovanni-Vincentelli. (2002) Defining platform-based design. [Online]. Available: http://www.eetimes.com/electronics-news/4141729/Defining-platform-based-design

[115] I. Software. (2010) Fault tree software item toolkit module.

[116] J. Srinivasan, S. P. Adve, P. Bose, and J. A. Rivers, "The impact of technology scaling on lifetime reliability," June 2004, pp. 177–186.

[117] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "The case for lifetime reliability-aware microprocessors," *Computer Architecture, International Symposium on*, vol. 0, p. 276, 2004.

[118] J. E. Stine, J. Grad, I. Castellanos, J. Blank, V. Dave, M. Prakash, N. Iliev, and N. Jachimiec, "A framework for high-level synthesis of system-on-chip designs," in *MSE '05: Proceedings of the 2005 IEEE International Conference on Microelectronic Systems Education*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 67–68.

[119] K. J. Sullivan, J. B. Dugan, and D. Coppit, "The galileo fault tree analysis tool," in *FTCS '99: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*. Washington, DC, USA: IEEE Computer Society, 1999, p. 232.

[120] D. L. Swanson, "Evolving avionics systems from federated to distributed architectures," in *17th Digital Avionics Systems Conference*. Bellevue, WA, USA: IEEE Press, 1998.

[121] I. TargetLink, *dSPACE version 3.1*, Wixom, MI, U.S.A., 2010. [Online]. Available: http://www.dspace.com/en/inc/home/products/sw/pcgs/targetli.cfm

[122] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl, *Fault Tree Handbook*. Washington, DC: U.S. Nuclear Regulatory Commission, 1981.

[123] C. B. Watkins, "Integrated modular avionics: Managing the allocation of shared intersystem resources," October 2006, pp. 1–12.

[124] C. B. Watkins and R. Walter, "Transitioning from federated avionics architectures to integrated modular avionics," October 2007, pp. 2.A.1–1 –2.A.1–10.

[125] N. Wattanapongsakorn and S. P. Levitan, "Reliability optimization models for embedded systems with multiple applications," *Reliability, IEEE Transactions on*, vol. 53, no. 3, pp. 406–416, 2004.

[126] M. White, "Scaled cmos technology reliability users guide," Jet Propulsion Laboratory, Pasadena, CA, USA, JPL Publication 09-33, January 2010.

[127] B. Witwer, "Systems integration of the 777 airplane information management system (aims): a honeywell perspective," November 1995, pp. 389–393.

[128] S. N. Woodfield, "An experiment on unit increase in problem complexity," *IEEE Transactions on Software Engineering*, vol. 5, no. 2, pp. 76–79, 1979.

[129] W. Xiang, P. C. Richardson, C. Zhao, and S. Mohammad, "Automobile brake-by-wire control system design and analysis," *Vehicular Technology, IEEE Transactions on*, vol. 57, no. 1, pp. 138–145, January 2008.

[130] Y. Xie, L. Li, M. Kandemir, N. Vijaykrishnan, and M. Irwin, "Reliability-aware co-synthesis for embedded systems," *The Journal of VLSI Signal Processing*, vol. 49, pp. 87–99, 2007, 10.1007/s11265-007-0057-6. [Online]. Available: http://dx.doi.org/10.1007/s11265-007-0057-6

[131] B. Yang, H. Hu, and S. Guo, "Cost-oriented task allocation and hardware redundancy policies in heterogeneous distributed computing systems considering software reliability," *Computers and Industrial Engineering*, vol. 56, no. 4, pp. 1687–1696, 2009.

[132] G. Yang and A. Sangiovanni-Vincentelli, "Separation of concerns: Overhead in modeling and efficient simulation techniques," 2004. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.62.5543

[133] Y. Yeh, "Triple-triple redundant 777 primary flight computer," vol. 1, February 1996, pp. 293–307.

[134] H. Zeng, A. Davare, A. Sangiovanni-Vincentelli, S. Sonalkar, S. Kanajan, and C. Pinello, "Design space exploration of automotive platforms in metropolis," in *Society of Automotive Engineers (SAE) 2006 World Congress & Exhibition*, April 2006.