CARNEGIE MELLON UNIVERSITY

SCALABLE GRACEFUL DEGRADATION FOR DISTRIBUTED EMBEDDED SYSTEMS

A DISSERTATION
SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

in

ELECTRICAL AND COMPUTER ENGINEERING

by

Charles Preston Shelton

Pittsburgh, Pennsylvania
June, 2003

Abstract

Distributed embedded computer systems are at the heart of many safety-critical systems such as airplanes, automobiles, and elevators. These systems have higher dependability requirements than general-purpose computer systems, as a system failure can cause human injury. However, these systems typically also have tight cost constraints, meaning there is a limit on the amount of design effort and redundant resources that can be spent making the system dependable. Traditional fault tolerance techniques of installing multiple identical backup systems may be cost prohibitive. Additionally, demand for more sophisticated system features has led to significantly more complex software being incorporated into these systems, and software design defects have become a major impediment to system dependability.

Graceful degradation mechanisms can potentially provide increased system dependability without having to provide redundant system resources. A gracefully degrading system tolerates partial system failures by providing reduced functionality with the remaining available system resources. In general, distributed embedded systems are designed to optimize performance and functionality with complex control algorithms and high quality sensors and actuators. The resources already designed into the system can provide some level of redundancy because not all of these system optimizations are required for the system to satisfy its primary requirements. Graceful degradation can exploit existing resources to provide increased dependability when partial system failures occur.

Designing a gracefully degrading complex software system is a significant challenge. Existing best practice consists of specifying all possible combinations of system failures, and designing a distinct system response for each combination. For a system with N failure modes, the design effort required for an ideal gracefully degrading system is O(2^N) which is clearly intractable for a complex distributed embedded system.

This thesis presents a scalable approach to building gracefully degrading distributed embedded systems. We define graceful degradation in terms of system utility: a generic measure of the system's ability to satisfy its functional and dependability requirements. An ideal gracefully degrading system minimizes the cumulative loss of system utility as successive system failures occur. We present a system model that enables scalable specification of system-wide graceful degradation. Our model views a distributed embedded system as a set of components that are either software components, sensors, or actuators. A system with N components that can each fail independently has 2^N possible distinct system failure configurations, one for each possible combination of failed components.

Defining the system's ability to gracefully degrade would traditionally require specifying the relative system utility of all 2^N possible failure combinations. We avoid this exponential complexity by exploiting the structure in the system's architecture to partition components into subsystems. We view each subsystem as a configuration of components that changes utility when components are removed due to failure or added via repair. We then view the system as a composition of subsystems that each contribute to overall system utility. Our model reduces the complexity of the system utility analysis from O(2^N) to O($N*2^k$) where k is the maximum number of components in any one subsystem.

We apply our system model to representative system architectures and identify some design techniques that can improve graceful degradation. We apply these design techniques to two distributed embedded systems and demonstrate how they enable scalable graceful degradation and increased system dependability. Our model also allows us to evaluate traditional fault tolerance techniques in terms of their ability to provide graceful degradation, and we can explicitly identify tradeoffs between the cost of graceful degradation mechanisms, in terms of design effort and redundant resources, and system dependability.

Acknowledgments

This thesis is dedicated to my wife, Tricia, without whom I could never have come so far so fast. Her love and support have sustained me through the arduous process of graduate school. I would also like to thank my parents and extended family, who have always believed in me. Also, my new "Pittsburgh Family" of in-laws, especially my mother- and father-in-law, have given me a support system that I needed so much and never imagined was possible away from home.

I thank my advisor, Philip Koopman, for all of the advice, guidance, and encouragement he has given me. This work would not have been possible without him. Also, I thank my committee members, Alan Baum, Bruce Krogh, and David Garlan, who provided time and support to meet with me and give me constructive criticism. My fellow students have made my graduate school experience both enjoyable and memorable. Thank you all for your help and support.

Graduate school is not free, and I appreciate all of the organizations that have provided financial support for my research. This work was sponsored by Lucent Technologies, the General Motors Collaborative Research Laboratory at Carnegie Mellon University, the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298, and the Pennsylvania Infrastructure Technology Alliance.

Table of Contents

List of Figures

List of Tables

1 Introduction

Our society has become increasingly dependent on complex, distributed embedded systems for critical activities. Airplanes, automobiles, and medical diagnostics systems are examples of safety-critical embedded computer systems that must continually provide dependable service in the face of harsh environmental conditions, partial system failures or loss of resources, or human error. Current techniques for assessing dependability properties such as reliability and availability typically focus on determining whether the system is working "perfectly" (i.e., provides 100% functionality) or has failed. However, reality is often somewhere between those two extremes.

Often a distributed system, after suffering some component failures, has enough resources to satisfy some or all of its primary objectives, even though it cannot fulfill all of its requirements completely. Not all system states of degraded functionality may be explicitly specified, but they are necessary to tolerate some failures. Degraded operating modes are especially important when cost precludes providing enough additional redundant resources to maintain total system functionality.

This thesis explores scalable techniques for specifying and designing *graceful degradation* into distributed embedded systems. Intuitively, the term graceful degradation means that a system tolerates failures by reducing functionality or performance, rather than shutting down completely. An ideal gracefully degrading system is partitioned so that failures in non-critical subsystems do not affect critical subsystems, is structured so that individual component failures have a limited impact on system functionality, and is built with just enough redundancy so that likely failures can be tolerated without loss of critical functionality. This is especially important for embedded systems, as they typically must maintain higher levels of dependable operation with fewer system hardware and software resources than general purpose computer systems.

Specifying and designing system-wide graceful degradation is not trivial. Graceful degradation mechanisms must handle not only individual component failure modes, but also combinations of component failures that can have a cumulative effect on the system's ability to continue operation. The previous best practice for specifying graceful degradation required identifying all system failure modes individually, as well as identifying all possible combinations of these failure modes [Herlihy91]. Then, a separate system recovery response was defined for each possible failure mode combination. Thus, specifying graceful degradation became exponentially complex with the number and type of possible failure modes. Typical graceful degradation design techniques emphasize adding complete component redundancy to preserve perfect operation when failures occur, or designing several redundant backup system configurations that must be tested and certified separately to provide a subset of system functionality with reduced hardware resources. These techniques have a high cost in both additional hardware resources and complexity of system design, and might not use system resources efficiently.

In general, it should be possible to provide graceful degradation in distributed embedded systems because a significant portion of a system's resources is used for optimization of certain properties, or increased system functionality. If a partial system failure occurs, the system can gracefully degrade by using these resources to preserve some basic level of functionality at the expense of losing the "auxiliary" system functionality or sacrificing high performance. We can define the minimum functionality required for primary missions, and treat optimized functionality as a desirable, but optional, enhancement. For example, the primary function of an elevator is to safely deliver all its passengers to their destinations. This can be accomplished, albeit very inefficiently, if the elevator moves slowly in the hoistway, stops at every floor, opens the doors at each floor, and does not compromise the safety of the passengers. Most elevators have much more functionality, such as responding to passenger input and only stopping at requested floors, as well as providing passenger feedback. However, if a few of the elevator buttons are broken, this should not cause the elevator to shut down. Similarly, much of a car's engine control software is devoted to emission control and fuel efficiency, but loss of emission sensors should not strand a car at the side of the road.

1.1 Problem Statement

Graceful degradation could be a mechanism for achieving high dependability in distributed embedded systems that have limited redundant system resources. When faults occur, the system may shed some functionality or reduce performance, but will continue to provide service. Unfortunately, specifying and designing a gracefully degrading system currently requires exponential design effort with the number of component faults that are considered. In the worst case, a separate system recovery mechanism must be designed for each possible combination of system faults that can occur. For distributed embedded systems that may have hundreds or thousands of individual processing nodes that each may host several software components, each of which can encounter system faults, this is infeasible.

This exponential design effort may offset any savings gained from not building dedicated redundant backup systems, and may not be feasible for human system designers with limited design time. In order for system-wide graceful degradation to be practical, the design effort required to specify, design, and implement graceful degradation mechanisms should be *scalable* with the design complexity of the system. In other words, the complexity that specification and design of system-wide graceful degradation adds to the system should not be greater than the total complexity of the system's design and architecture. Prior to this research, we have not seen any work that addresses the problem of scalability for specifying and designing graceful degradation. This thesis is a first step towards a *methodology for scalable graceful degradation* in distributed embedded systems. Our ultimate goal is to reduce the design effort necessary to build gracefully degrading systems so that it is tractable for system designers.

This research proposes an architectural system model, an analysis technique, and architectural design techniques to achieve scalable graceful degradation in distributed embedded systems. We present a system model that enables scalable specification and analysis of graceful degradation and has helped us to identify some system architecture properties that may contribute to a system's ability to degrade gracefully. We then apply this model to two representative distributed embedded system designs and identify: (i) how well these systems gracefully degrade, and (ii) the parts of the system that we could modify to improve graceful degradation.

We define graceful degradation in terms of system utility: a measure of the system's ability to satisfy its specified functionality and dependability requirements. A system that has all of its components functioning properly has maximum utility. A system degrades gracefully if component failures reduce system utility proportionally to the sum of all the components that have failed. Utility is not all or nothing; the system provides a set of features, and ideally the loss of one feature should not hinder the system's ability to provide the remaining features. It should be possible to lose a significant number of components before system utility falls to zero.

We focus our analysis on distributed embedded computer systems. Distributed embedded systems are usually resource constrained, and thus cannot afford complete hardware redundancy. However, they have high dependability requirements (due to the fact that they must react to and control their physical environment), and have become increasingly software-intensive. These systems typically consist of multiple compute nodes connected via a potentially redundant real-time fault-tolerant network. Each compute node may be connected to several sensors and actuators, and may host multiple software components. Software components provide functionality by reading sensor values, communicating with each other via the network, and producing actuator command values to provide their specified behavior.

Our system model provides a means for assessing graceful degradation by evaluating the relative utility of system configurations. Our framework achieves scalable analysis by partitioning the system into subsystems based on component input and output interfaces, and restricting utility analysis to individual subsystems.

Rather than specify the relative utility values of all possible configurations of the system, we determine only the utility values of configurations of each subsystem, and then combine these values to evaluate the utility of all possible system configurations.

This framework enables tractable analysis and design of graceful degradation in distributed embedded systems. We can use the model to explicitly identify tradeoffs among the design effort required for graceful degradation mechanisms, the cost of redundant resources, and the improvement to the robustness of the system. We can also use the model to evaluate the graceful degradation of the system implementation and ensure that it matches the system design and dependability requirements.

This work is a part of the RoSES (Robust Self-Configuring Embedded Systems) project and builds on the idea of a configuration space that forms a product family architecture [Nace2000]. Each point in the space represents a different configuration of hardware and software components that provides a certain utility. Removal or addition of a component to a system configuration moves the system to another point in the configuration space with a different level of utility. For each possible hardware configuration, there are several software configurations that provide positive system utility. Our model focuses on specifying the relative utility of all possible software component configurations for a fixed hardware configuration. For a system with N software components, the complexity of specifying a complete system utility function is normally $O(2^N)$. Our model exploits the system's decomposition into subsystems to reduce this complexity to $O(N*2^k)$, where k is the maximum number of components within a single subsystem. When we have a complete utility function for all possible software configurations, we can identify how well the system gracefully degrades by examining the differences in utility among different system configurations.

A scalable specification of system-wide graceful degradation enables scalable analysis and design of graceful degradation. We can rank the relative utility of different system configurations and identify which components and subsystems provide significant system utility contributions. We can then target these components and subsystems for graceful degradation design improvements, rather than adding design complexity to the entire system. We can also use the system utility model to validate the graceful degradation ability of the system implementation. If we compare the utility of different system configurations predicted by the model to the ability of these configurations to satisfy system requirements in the implementation, we can evaluate whether the implemented system actually achieves graceful degradation.

1.2 Thesis Contributions

This research provides four major contributions towards designing gracefully degrading distributed embedded systems:

- A structural model derived from the system's software architecture specification that enables scalable specification of graceful degradation in embedded systems, and expresses many current hardware and software fault-tolerance techniques in a single framework.
- Proposed design principles that will promote system-wide graceful degradation in distributed embedded systems that were identified as a result of applying the system model.
- A tractable analysis technique that uses the model to provide hints to where to focus design effort for improving graceful degradation and can validate that the implementation achieves graceful degradation.
- Two case studies in which we applied our system model and design techniques to representative distributed embedded system applications and observed how well they could gracefully degrade.

The framework we have developed makes it possible to quantitatively assess how well the system will gracefully degrade due to the particular system properties developed in the software architecture.

1.3 System Context

The system's software architecture embodies the system behavior and functionality, but it must be considered with the rest of the computer system as well. Our goal is to identify and systematically measure what properties of the system’s software design contribute to graceful degradation in distributed embedded systems. However, for the complete system to degrade gracefully, other system properties must be addressed as well. Since this work only addresses the particular architectural style that is common for distributed embedded systems, we make some assumptions about these properties that match this type of system and puts the software system in an “ideal” context:

- System hardware resources satisfy all processing, memory, and bandwidth requirements for the software system.
- The system is scheduled so that all working components satisfy real-time requirements, and failure recovery mechanisms have been considered in the schedule such that they do not cause additional timing faults.
- The fault model assumes that all components are fail-fast and fail-silent, and that these failures are detectable by other system components.
- The system communication mechanisms are assumed reliable and the software architecture is specified at the level of component inputs and outputs.

These assumptions are non-trivial, but determining how to achieve them and how they impact the system is outside the scope of the research proposed here. Our focus is on design techniques for graceful degradation that tolerate combinations of component failures.

Real-time embedded control systems are typically designed to be time-triggered [Kopetz97], meaning that processing and network communication are periodically scheduled. A software component may be implemented as a real-time task that periodically processes inputs and produces outputs. Thus, a timing failure in a component will manifest as an output not being updated before its deadline, and not being available for other components to process. This matches our fail-fast, fail-silent fault assumption. Other components that receive the component's outputs will detect that the component has failed because it missed its deadline and did not produce its outputs.

Any failures of interest must be detectable by other system components. If the other components never detect a component failure, the system cannot recover from it. This work focuses on how to design the system to automatically recover from failures rather than attacking the issues of failure and fault detection. We make a common assumption that most component failures will be fail-fast and fail-silent, and that failures only manifest as the loss of a outputs from a component. The communication interface will aid failure detection somewhat, as invalid messages can be detected if they do not follow the communication protocol. However, the problem of determining when a component is sending valid but incorrect information is an open question that currently cannot be overcome without costly replication and approaches such as Byzantine-agreement algorithms [Lamport82].

Our view of the software architecture is at a level of abstraction that defines the components and their interfaces but not the detailed design of the components or communication mechanisms. The architectural connectors are represented by system variables that represent the data values passed among software components. The communication implementation must satisfy communication requirements such that data outputs from components are available as inputs for other components to satisfy real-time deadlines and provide functionality. The software architecture described in our system model is separated from the network communication implementation and should not need to know the details of how data is transmitted among components.

For distributed embedded systems, we assume that the network is a fault-tolerant broadcast bus that transmits all messages to all nodes periodically, ensuring that all software components receive their inputs. However, changing the communication architecture does not affect the validity of our software model, as long as all working components receive their inputs from other working components. There could be distributed middleware that ensures that messages are delivered in time for real-time deadlines to be met, and can optimize message delivery when software components reside on the same hardware node. A survey of communication architectures for embedded control systems is presented in [Rushby2001].

1.4 Thesis Outline

The rest of this thesis is organized as follows: Chapter 2 discusses prior and related research areas for graceful degradation, dependability, embedded systems, and software architecture. Chapter 3 introduces our system model for specifying graceful degradation with an illustrative example, and shows how we can apply this model to traditional fault tolerance and dependability techniques. Chapter 4 shows how we applied this model to a more complex automobile navigation system and describes design techniques for achieving graceful degradation. We also present an analysis method for using the model to identify which parts of the system should receive more graceful degradation design effort and to validate graceful degradation in the system implementation. Chapter 5 describes a case study with the design and implementation of a distributed embedded elevator control system. Chapter 6 describes a case study with an autonomous robot navigation system. Finally Chapter 7 ends with conclusions and future work.

2 Related Work

This thesis draws on work from several different research areas to address the problem of scalable graceful degradation in distributed embedded systems. In this chapter we will examine current research in graceful degradation, dependability and fault tolerance, embedded system architecture and design patterns, and software architecture.

2.1 Graceful Degradation

Previous work on formally defining graceful degradation for computer systems was presented in [Herlihy91]. That work proposed constructing a lattice of system constraints that identifies what tasks the system can accomplish based on which constraints it can satisfy. A system that works perfectly satisfies all constraints, and a system that encounters failures might satisfy a looser set of constraints and still provide functionality, but is degraded with respect to some system properties. The difficulty with this model is that in order to specify the relaxation lattice, it is necessary to specify not only every system constraint, but also how constraints are relaxed in the presence of failures. It further requires determining how constraints interact and developing a recovery scheme for every possible combination of failures in order to move between points in the lattice. Because all combinations of component failures must be considered, specifying and designing graceful degradation is exponentially complex with the number of system components.

Other work on graceful degradation has focused on developing formal definitions [Jayanti99, Weber89], but has not addressed how to apply these definitions to complex system specifications, nor how to overcome the problem of exponential complexity for specifying failure modes and recovery mechanisms. The concept of multitolerance was proposed in [Arora98] to provide a unifying mechanism for providing dependability and graceful degradation by classifying all possible types of faults and designing separate mechanisms called detectors and correctors to minimize their effects on the system. However, global detectors and correctors must be specified for every distinct failure in the system, and every combination of detector and corrector mechanisms for different fault classes must be analyzed to ensure that they do not negatively interact to decrease system dependability.

Research on implementing graceful degradation for tolerating missed deadlines and solving quality of service constraints [Abdelzaher97, Mittal98, Ramanathan97] has focused only on processor load and timing-related faults rather than application faults due to component failures. The graceful degradation observed is only in terms of system performance rather than reduced or different functionality. Research effort in building self-healing systems is ongoing [WOSS2002], and may be complementary to gracefully degrading systems. Self-healing systems might incorporate mechanisms for graceful degradation to prevent interruption of service while the system recovers from a failure.

The term "graceful degradation" has been used informally in many different situations to mean anything from fault tolerance to quality of service guarantees. Graceful degradation has been identified as a desirable property for dependable systems and has been studied in early reliability research [Losq77, Ng77], but the focus was mainly on evaluating graceful degradation in terms of traditional hardware reliability models. Our work differs from previous research in that we provide a framework for explicitly defining what software system properties graceful degradation covers, and how graceful degradation affects system dependability.

2.2 Dependability and Fault Tolerance

Dependability covers a range of system properties such as reliability, availability, and maintainability. A taxonomy of dependability properties and related concepts of fault definitions, diagnosis, and recovery are listed in [Avizienis2001]. Traditional reliability and availability models tend to focus on hardware architecture and configurations rather than software, and the notion that a system can only move between the states of perfectly working and failed when faults occur. A software reliability model based on software architecture was described in [Wang99], but required knowledge of individual software component reliabilities (a difficult problem in its own right), and did not specifically address graceful degradation or include a notion of a partially working system.

Traditional fault tolerance relies on redundant resources to provide dependability, and can tolerate a limited number and type of system faults. Hardware replication strategies such as triplex modular redundancy [Rennels84] provide redundant copies of software running on separate processors to tolerate hardware faults, but cannot prevent a fault due to a software design defect that will affect all copies of the software. Software fault tolerance techniques such as N-version programming rely on multiple design efforts to build multiple distinct software modules that provide the same functionality but will not have the same design defects, ensuring that they will not fail due to a correlated defect [Avizienis85]. However, this technique requires twice or more the design effort to build multiple software modules, and it is controversial whether this actually prevents correlated software defects [Knight85, Koopman99]. Both hardware and software fault tolerance techniques have a cost either in terms of replicated resources, design effort, or both. Additionally, if enough faults occur to fail all of the backups, the system will then become very brittle and susceptible to catastrophic failures.

Survivability and performability are related to our concept of graceful degradation. Survivability is a property of dependability that has been proposed to define explicitly how systems degrade functionality in the presence of failures [Knight2000, Knight2003]. Performability is a unified measure of both performance and reliability that tracks how system performance degrades in the presence of faults [Meyer78, Meyer93]. Our work differs from survivability in that we are interested in building implicit graceful degradation into systems without specifying all failure scenarios and recovery modes *a priori*. Also, we focus on distributed embedded systems rather than on large-scale critical infrastructure information systems. Performability relates system performance and reliability, but our concept of graceful degradation addresses how system functionality can change to cope with component failures. Military systems have long used similar notions to provide graceful degradation (for example, in shipboard combat systems), but had scalability limits and were typically limited to a dozen or so specifically engineered configurations.

Other researchers in dependable distributed systems define graceful degradation as a combination of performability and real-time quality of service [Verissimo2001]. Real-time quality of service specifications define levels of performance that the system can maintain given available system resources. As resources are lost, system performance will degrade and some system services may be stopped to provide resources for other services that are mission-critical. However, this view of graceful degradation only deals with system hardware resources such as network bandwidth or processor utilization, and only focuses on the effects of timing faults or resource overload faults.

In contrast, our view of graceful degradation is that it is a general mechanism that can refer to any individual system property or set of properties in the presence of any set of defined faults. We use system *utility* as the general combined metric for whatever properties the system is required to satisfy, and we specify a fault model that explicitly states what faults the graceful degradation mechanism should cover. Beyond performance and reliability, functionality, security, availability, maintainability and other system properties could potentially degrade in the presence of system failures. These properties may not be quantitatively defined, but may have several levels of service that can be ranked in terms of utility. These levels of service may also map to different forms of system functionality that cannot be mapped to a resource quality of service model.

The system faults identified may be design defects that fail software and hardware components in addition to timing faults or resource overload faults that make system resources unavailable. There may be multiple faults that manifest as the same failure behavior and can be handled with one mechanism. Our goal is to provide a framework for evaluating graceful degradation that can be tailored to a system's fault model and system requirements. We have built some assumptions about system utility and system faults into our model, but attempted to make their definitions explicit and extensible.

2.3 Embedded Systems

Current industry practice for dealing with faults and failures in embedded systems focuses on the traditional approaches of fault-tolerance and fault-containment [Rushby99]. Software subsystems are physically separated into different hardware modules. Additionally, system resources, such as sensors and actuators, that are commonly used may be replicated for each subsystem. That approach provides assurance that faults will not propagate between subsystems since they are physically partitioned, and fault tolerance is achieved by replicating resources and subsystems. Typically, failures are dealt with by having separate backup subsystems available rather than shedding functionality when resources are lost. This approach is a restricted form of graceful degradation, in that it tolerates the loss of a finite set of components before suffering a complete system failure. However, this methodology is costly because of its required high level of redundancy. Other research on designing graceful degradation for manufacturing control systems [Adlemo95] did not address how to overcome the difficulty of dealing with increasing combinations of possible failure modes.

A promising approach to achieving system dependability is NASA's Mission Data System (MDS) architecture [Dvorak2000, Rasmussen2001]. This system architecture is being designed for unmanned autonomous space flight systems that must complete missions with limited human oversight. Their architecture focuses on designing software systems that have specific goals based on well defined state variables. The software is decomposed based on the subgoals it must complete to satisfy its primary goal. The software is not constrained to a particular sequence of behavior, but rather must determine the best course of action based on its goals. The potential difficulties with this approach include the effort required to decompose goals into subgoals, and conflict resolution among subgoals at run time. Our framework differs from MDS in that we specifically focus on behavior-based subsystems and the coordination among them through system communication interfaces.

2.4 Software Architecture

We also draw on research from the software architecture community to explore how a system's high-level organization can influence its ability to gracefully degrade. Well-known system decomposition strategies have been codified into architectural patterns that have become common knowledge. Architectural principles have become recognized as a major part of the system design process [Bass98, Shaw96]. Work has also been done on fitting architectural patterns into a taxonomy based on their system properties as a resource for choosing certain architectural styles for certain systems [Kazman97, Shaw97]. There have been several papers on applying certain architectural patterns to specific embedded system domains and real-time distributed systems [Banks94, Boasson98, Botti2000, Ravindran97, Rostamzadeh95], but we have not found any research focusing on developing a generalized methodology for system-wide graceful degradation using architectural properties.

Our system model focuses on distributed embedded system architectures, and defines software in terms of components that represent real-time tasks, and system variables that represent data communicated between these tasks. This is somewhat similar to the traditional software architecture view of components and connectors. If an embedded system architecture specifies the set of system components and their input and output interfaces, this should be enough information to express the system in terms of our model.

Many architecture description languages (ADL) have been proposed for expressing a system's software architecture. In [Medvidovic97] a comprehensive set of ADL's is examined in terms of what system properties they can express. Our software system model is not a substitute for an ADL or architecture specification, but is derived from these structures to primarily highlight the components defined in the system and the dependencies among them. We use the Acme ADL [Garlan2000] to formally specify the semantics of our software component model. This formal specification provides unambiguous definitions of the framework of our model, making it accessible to other architects familiar with ADL's. Additionally, the tool support available for Acme may provide a foundation for automating our system model analysis.

3 System Model for Graceful Degradation

Our system model for specifying graceful degradation is based on identifying the relative *utility* of all possible valid system component configurations. Overall system utility may be a combination of functionality, performance, and dependability properties, based on the requirements of the system for the services it must provide. For a system that is a set of N software components, sensors, and actuators, the total possible system configurations are represented by the system's power set. Thus, there are 2^N possible system configurations. If we specify the relative utility values of each of these 2^N configurations, then we can determine how well a system gracefully degrades based on the utility differences among different software configurations.

Our model enables complete definition of the system utility function without having to evaluate the relative utility of all 2^N possible configurations. Our model splits the system into orthogonal software and hardware views so that we can specify the utility of all software configurations without considering the hardware system, but still see the effects of hardware redundancy mechanisms on graceful degradation. A software data flow graph enables scalable system utility analysis by partitioning the system into subsystems and identifying the dependencies among software components. Our system utility model is based on the system's software configurations. It is primarily concerned with how system functionality changes when software components fail, and the effect of software fault tolerance techniques on system utility. A hardware allocation view enables mapping hardware failures to software component, sensor, and actuator failures for utility analysis. The hardware view also represents the effect of hardware replication on system dependability for hardware reliability and availability analyses.

We focus on real-time distributed embedded computer systems, which allows us to make several assumptions about a system's organization and fault model. Such systems are often composed of autonomous periodic tasks (e.g. reading a sensor value, updating a controller output) that only communicate via state variables (e.g. sensor data values, control system parameters, actuator command values). Examples of such systems include automotive and avionics control systems. Therefore our model of communication among software components is based on data flow rather than control flow, and assumes a fault-tolerant, broadcast real-time network.

The fault model for our system uses the traditional fail-fast, fail-silent assumption on a component basis, which is best practice for this class of system. Individual components are designed to shut down when they detect an unrecoverable error, meaning they no longer provide their outputs to the rest of the system. The loss of a component's outputs enables the other components in the system to detect the component's failure, and prevents an error from propagating through the rest of the system. All faults in our model thus manifest themselves as the loss of outputs from failed components. Software components either provide their outputs to the system or do not. Hardware component failures cause loss of all software components hosted on that processing element. Network or communication failures can be modeled as a loss of communication between distributed software components.

3.1 Data Flow and Dependency Graph

The data flow graph shows how information flows in the system from sensor inputs, through software components, to actuator outputs. Each vertex in the graph is a sensor, actuator, or software component, and each edge in the graph is a *system variable* that represents communication among components. This data flow graph can be directly generated from the system design's software component definitions and interface specifications.

If the system has a software architecture specification, we can generate the system from the component and connector view of a system's software architecture. The components are software components that represent real-time tasks that produce periodic outputs, sensors, and actuators. The connectors are the system variables that represent data communicated among components. Since the class of embedded systems we are examining deal primarily with data flow at the application level, they generally resemble the pipe-and-filter architectural style [Shaw96] in this dependency graph view. Section 3.2 presents a formal representation of the system's component model as an architectural style in the Acme ADL.

To illustrate the model, we present a hypothetical automotive brake-by-wire system. We constructed this example by adapting a real anti-lock braking system design described in [Jurgen99] from a centralized electro-mechanical system to a distributed software control system. We also added a vehicle dynamics subsystem to represent an active stability control feature. Figure 3.1 shows the data flow graph for this system, with all of the software components, sensors, and actuators necessary for braking functionality on the left front (LF) wheel of the car. The brake

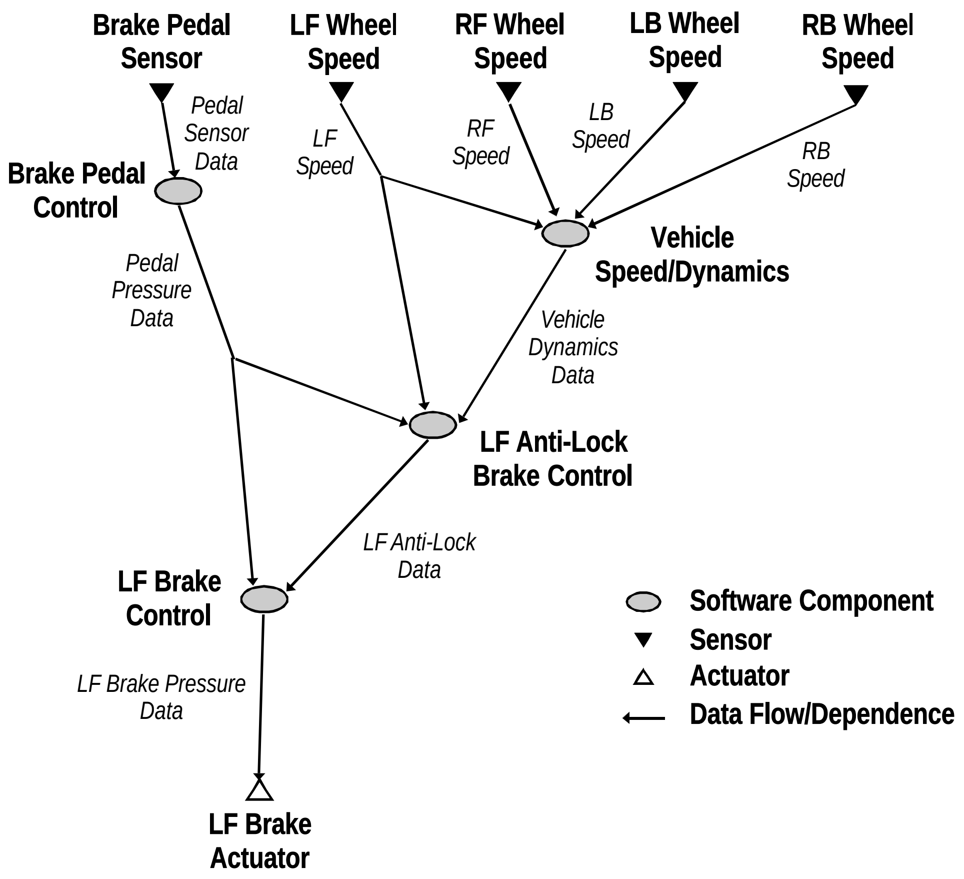

Figure 3.1. Data Flow Graph for the Left Front Wheel Brake Actuator.

controller sends brake commands to the brake actuator for the LF wheel, and the brake controller receives input from the pedal controller (which monitors the pedal sensor for driver brake commands) and anti-lock braking software. The anti-lock software also receives input from the pedal controller, as well as the LF wheel speed sensor (to detect when the wheel locks) and vehicle dynamics software component (to maintain stability of the vehicle). The vehicle dynamics software monitors all four wheel speed sensors to calculate the overall vehicle speed. The right front (RF), left back (LB), and right back (RB) wheel braking subsystems have similar data flow graphs, and they all receive data from the pedal controller and vehicle dynamics software.

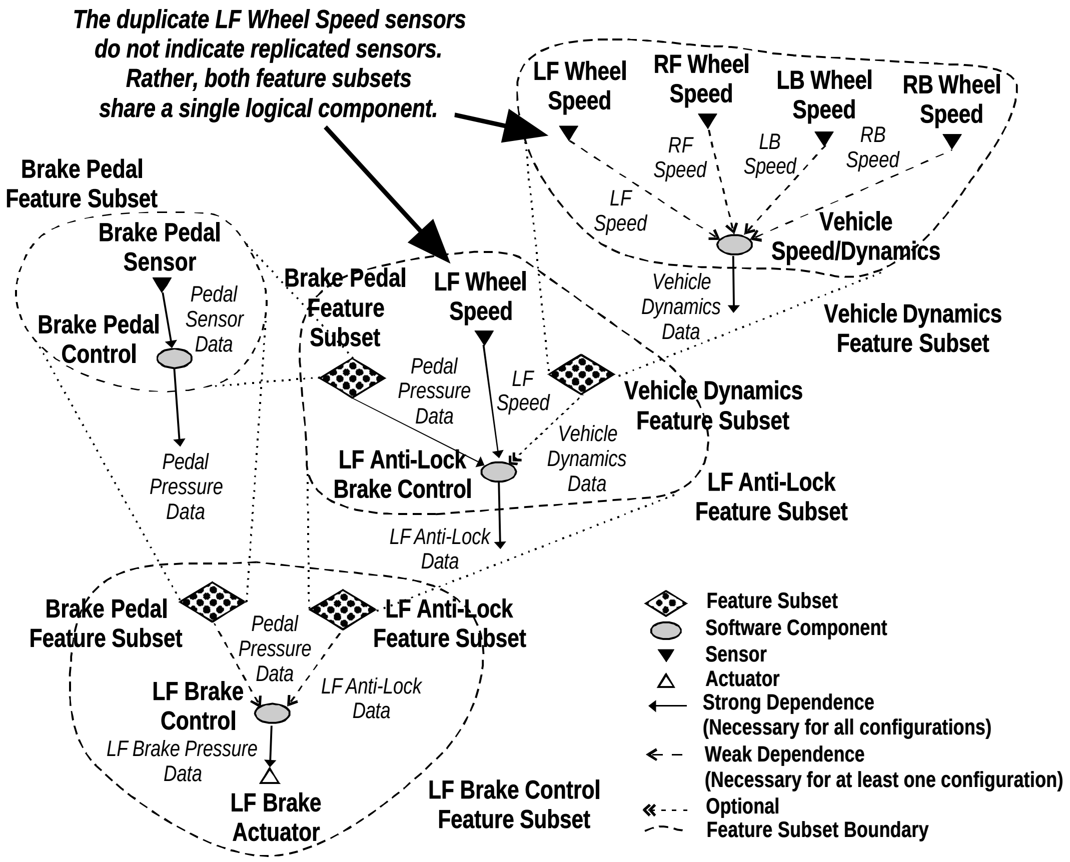
The duplicate LF Wheel Speed sensors do not indicate replicated sensors. Rather, both feature subsets share a single logical component.
LF Wheel Speed
RF Wheel Speed
LB Wheel Speed
RB Wheel Speed
RF Speed
LB Speed
RB Speed
LF Speed
Vehicle Speed/Dynamics
Vehicle Dynamics Data
Vehicle Dynamics Feature Subset
Brake Pedal Feature Subset
Brake Pedal Sensor
Pedal Sensor Data
Brake Pedal Control
Brake Pedal Feature Subset
LF Wheel Speed
Pedal Pressure Data
LF Speed
Vehicle Dynamics Feature Subset
Vehicle Dynamics Data
LF Anti-Lock Brake Control
LF Anti-Lock Feature Subset
Pedal Pressure Data
LF Anti-Lock Data
Brake Pedal Feature Subset
LF Anti-Lock Feature Subset
Pedal Pressure Data
LF Anti-Lock Data
LF Brake Control
LF Brake Pressure Data
LF Brake Actuator
LF Brake Control Feature Subset
Feature Subset
Software Component
Sensor
Actuator
Strong Dependence (Necessary for all configurations)
Weak Dependence (Necessary for at least one configuration)
Optional
Feature Subset Boundary

Figure 3.2. Feature Subset Definitions and Component Dependencies.

Based on the data flow graph, we can group the components into subsystems based on the outputs they provide. We define these subsystems in our model as *feature subsets*. A feature subset is a set of components (software components, sensors, actuators, and possibly other feature subsets) that work together to provide a set of output variables. Feature subsets may or may not be disjoint and can share components across different subsets. Each feature subset can be viewed as a subgraph of the system data flow graph, where other contained feature subsets are represented as components. Figure 3.2 shows the feature subset definitions for the braking system with respect to the LF wheel. The LF brake control feature subset contains the LF brake actuator, the LF brake control software component, and the brake pedal and LF anti-lock braking feature subsets. The LF anti-lock braking feature subset is composed of the LF anti-lock braking software component, the LF wheel speed sensor, and the brake pedal and vehicle dynamics feature subsets. The vehicle dynamics feature subset contains the vehicle dynamics software component and all four wheel speed sensors. Note that feature subsets can share components if they require similar information. For example, both the brake control and anti-lock braking feature subsets contain the brake pedal feature subset as a component that provides the *Pedal Pressure Data* system variable. Additionally, the LF wheel speed sensor is a component in both the LF anti-lock braking and vehicle dynamics feature subsets. These shared components only represent one logical instance in the software data flow view, and whether or not they are replicated in hardware will be visible in the hardware allocation view (see Section 3.3).

The data flow graph can also represent dependency relationships among components. Each component may provide functionality without all of its specified inputs. For example, the brake control software only needs input from either the pedal control software or the anti-lock braking software. The anti-lock braking output is preferred because it provides better vehicle stability while braking, but if it is not available, normal braking is still possible with the pedal control output.

We annotate the data flow graph with a set of dependency relationships among components (Figure 3.2 illustrates this for the example brake-by-wire system). These relationships are determined by each component's dependence on its input variables, which might be strong, weak, or optional. If a component is dependent on one of its inputs, it will have a dependency relationship with all components that output that system variable. A component *strongly* depends on one of its inputs (and thus the components that produce it) if the loss of that input results in the loss of the component's ability to provide its outputs. A component *weakly* depends on one of its inputs if the input is required for at least one configuration, but not required for at least one other configuration. For example, the vehicle dynamics software component requires at least one wheel speed sensor to perform calculation of vehicle dynamics, but it can still provide its output without inputs from all four wheel speed sensors. Additionally, a component can be weakly dependent on multiple components that redundantly output the same required system variable. If an input is *optional* to the component, then it may provide enhancements to the component's functionality, but is not critical to the basic operation of the component. For example, the anti-lock braking software can produce its outputs with only the *Pedal Pressure* and *Wheel Speed* system variables. The *Vehicle Dynamics Data* system variable enhances the anti-lock braking functionality, but is not required for basic operation.

These dependency relationships will enable us to eliminate invalid configurations (configurations that have zero utility) for each feature subset based on whether or not components that provide required system variables are present in each configuration. Any valid feature subset configuration must contain all of the components necessary to satisfy all strong system variable dependancies within the feature subset. At least one component that provides each distinct system variable must be present in the configuration. All other configurations can be eliminated as invalid. For system variables that are considered optional in a feature subset, the presence or absence of the components that output these variables in a configuration may affect the utility of the feature subset, but will not affect whether or not that feature subset is valid. For any valid configuration without an optional component, the same configuration that only differs by the addition of that optional component must also be valid and must be evaluated.

Weakly dependent system variable inputs cover all variables that are required as inputs for some components in some feature subset configurations, and are optional inputs for those components in other feature subset configurations. The only situation in which we have used the weak dependency relationship is when there are multiple components that have semantically related output variables that can serve as redundant backups for inputs to other components. In this situation, all configurations in which at least one of the components that can provide one of the weakly dependent outputs are valid. The weakly dependent relationship is intentionally broad so that more complex dependency relationships can still be represented in our model without having to redefine the basic semantics. These dependencies are only used in a model to reduce the number of valid configurations that must be evaluated in each feature subset.

3.2 Acme Specification of the Software System View

The Acme ADL [Garlan2000] provides a mechanism for generating a formal specification of an architectural style by defining the semantics of component and connector interaction. The architectural style of our software system view represents software components, sensors, and actuators as components, and system variables as connectors, along with the basic rules of how they should interact. The Acme specification of our software component model is listed in Appendix A. Our Acme specification only covers the system component and interface definitions.

Each component has a set of input and output ports that specify which input variables they receive and which output variables they produce. Each component's input port also has a dependency property associated with it that specifies whether the input's dependency is strong, weak, or optional for that component.

Acme currently does not have a mechanism to accurately represent our hierarchical feature subset definitions. Acme uses recursive component definitions to represent hierarchical component decomposition, but the hierarchy is strict and components at different levels of the hierarchy are not visible to one another. Acme also allows specification of groups of associated elements (components and connectors), but the current semantics for group definitions do not allow one group to contain another as an element. Feature subsets, on the other hand, represent sets of components that form logical subsystems but do not encapsulate all of the interfaces of the components they contain. Feature subsets also allow multiple feature subsets to contain the same component instance, and allow one feature subset to contain another as a component without strong encapsulation.

Our definition of feature subsets is essential to our model's ability to provide scalable specification of system-wide graceful degradation. It is common in embedded systems for separate subsystems to share resources and information but not necessarily have a strict hierarchical structure that ensures that subsystems are disjoint and layered. Acme only allows specification of disjoint hierarchical subsystems, so an Acme architectural description of this type of system could have only one level where all components and connectors are visible, but the subsystem definitions are obscured. Thus, system utility evaluation cannot be partitioned to individual subsystems because they are not visible in the architecture description.

Feature subsets provide a mechanism for evaluating the utility of individual subsystems as if they were disjoint while preserving connections and common dependencies to other subsystems.

3.3 Hardware Allocation Diagram

The hardware allocation diagram provides information about which processors are tied to sensors and actuators, and where software components are allocated in the hardware system. The hardware structure of the system defines the set of available processing elements that form a distributed system. The fault-tolerant network topology is described in terms of which hardware nodes can communicate with each other. Each hardware component has sensor, actuator, and software components mapped to it, defining the hardware configuration. Sensors and actuators are physically connected to particular nodes in the system, and software components are allocated to nodes. There may be multiple sensors, actuators, or software components allocated to different hardware nodes for redundancy. This view of the system allows us to assess the system's ability to tolerate hardware failures by identifying which software components, sensors, and actuators are affected by a processor failure.

Figure 3.3 shows a possible hardware allocation for our example brake-by-wire system. Software components, sensors, and actuators for brake control in each wheel are allocated to separate processors, while the components for vehicle dynamics and brake pedal control are allocated to other processors. Notice that in hardware there are dual redundant brake pedal sensors and brake pedal controllers

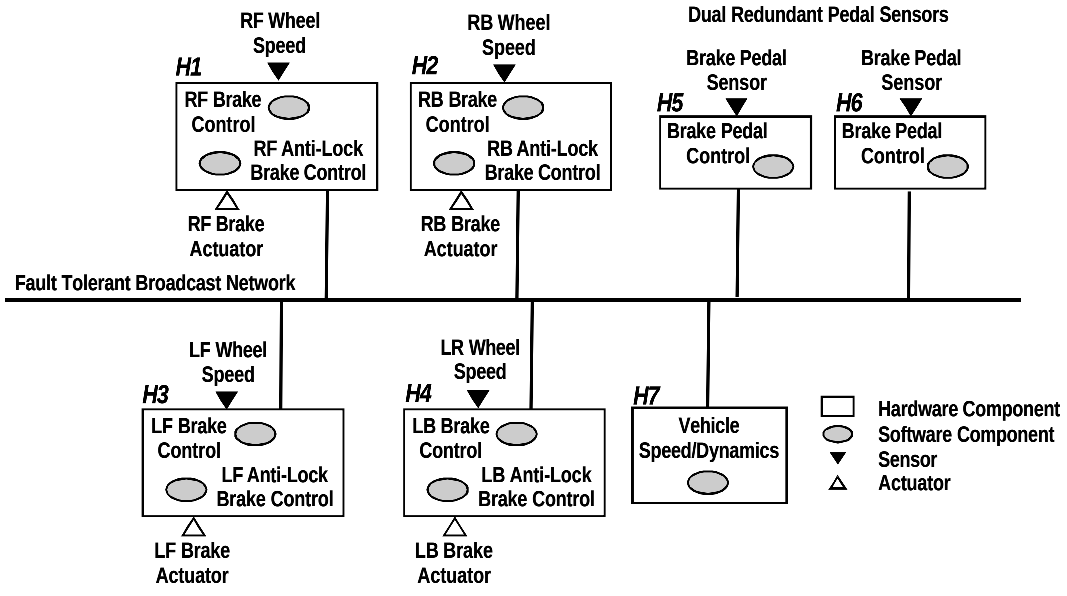

Figure 3.3. A Hardware Allocation Diagram for the Brake-By-Wire System.

to provide increased reliability. This redundancy is orthogonal to the software data flow graph. A software component that is replicated in hardware only represents one logical software component in the software data flow system view. This means that redundancy management mechanisms such as replica determinism are below our level of abstraction and are implicit in our view of the software architecture.

We assume that the hardware allocation mapping is static during operation (one could envision dynamic allocation, but that is beyond the scope of this work). If there are redundant hardware nodes, loss of one node will not change the software configuration if there is another copy still available. A hardware node failure that removes a set of software components, sensors, and actuators from the system will alter the software configuration by the loss of those components. Therefore, we can focus on analyzing the utility of the software configuration to assess graceful degradation.

3.4 Utility Model

Our utility model exploits the system decomposition captured in the software data flow view to reduce the complexity of specifying a system utility function for all possible software configurations. We have already grouped the system components into several feature subsets based on their communication interfaces, and these feature subsets encapsulate functional subsystems. Rather than manually rank the relative utility of all 2^N possible software configurations of N components, we restrict utility evaluations to the component configurations within individual feature subsets. We specify each component's utility value to be 1 if it is present in a configuration (and providing its outputs), and 0 when the component is failed and therefore not in the configuration. Utility values of other than 0 or 1 for individual components are precluded by the fail-fast, fail-silent assumption.

We also make a distinction between *valid* and *invalid* system configurations. A valid configuration provides some positive system utility, and an invalid configuration provides zero utility. For graceful degradation we are interested in the utility differences among valid system configurations, as the system is still considered "working" until its utility is zero. In general, there are many "trivially" invalid system configurations. A system configuration that strongly depends upon a component that is failed provides zero utility regardless of what other components are present. For example, a car with no brake actuators at all cannot provide its basic system functionality and is already failed, so examining the rest of the system's component configuration is unnecessary. However, there is still a set of multiple valid configurations that must be ranked for system utility, and we use our feature subset definitions to specify the utility of these system configurations.

Configurations that are invalid due to weak dependence on multiple missing components can be eliminated as well. If a feature subset has m components, where these components can be software components, sensors, actuators, or other feature subsets, then we can define the utility of the feature subset U_f as:

$U_f = H_f(u_1, u_2, \dots , u_m)$

Where $u_1 \dots u_m$ are the utility values of each component, and H_f is the utility function of the feature subset. Since we are restricted to the feature subset and not the entire system, we only need to rank the relative utility of 2^m possible component configurations to completely specify the feature subset's utility function. For many systems, $m << N$, making this task tractable. If there are p feature subsets in the entire system, and the number of components per feature subset is bounded by $k << N$, then we must evaluate a maximum of $p*2^k$ component configurations to specify the utility functions of all feature subsets.

To determine the utility of system configurations, we must be able to relate the utilities of individual feature subsets to overall system utility. We can view the system as providing several orthogonal functional *capabilities* that can be implemented by one or more feature subsets. Capabilities are "top-level" feature subsets that encapsulate all other feature subsets in a hierarchical subsystem decomposition. Once we have determined the relative utility values of feature subset configurations, we can determine the utility of a system configuration by specifying the utility of the configurations of the system's functional capabilities.

Each functional capability may have multiple feature subsets that can implement the required functionality. For graceful degradation, the system designers may create multiple feature subsets of varying utility for each capability. The utility of

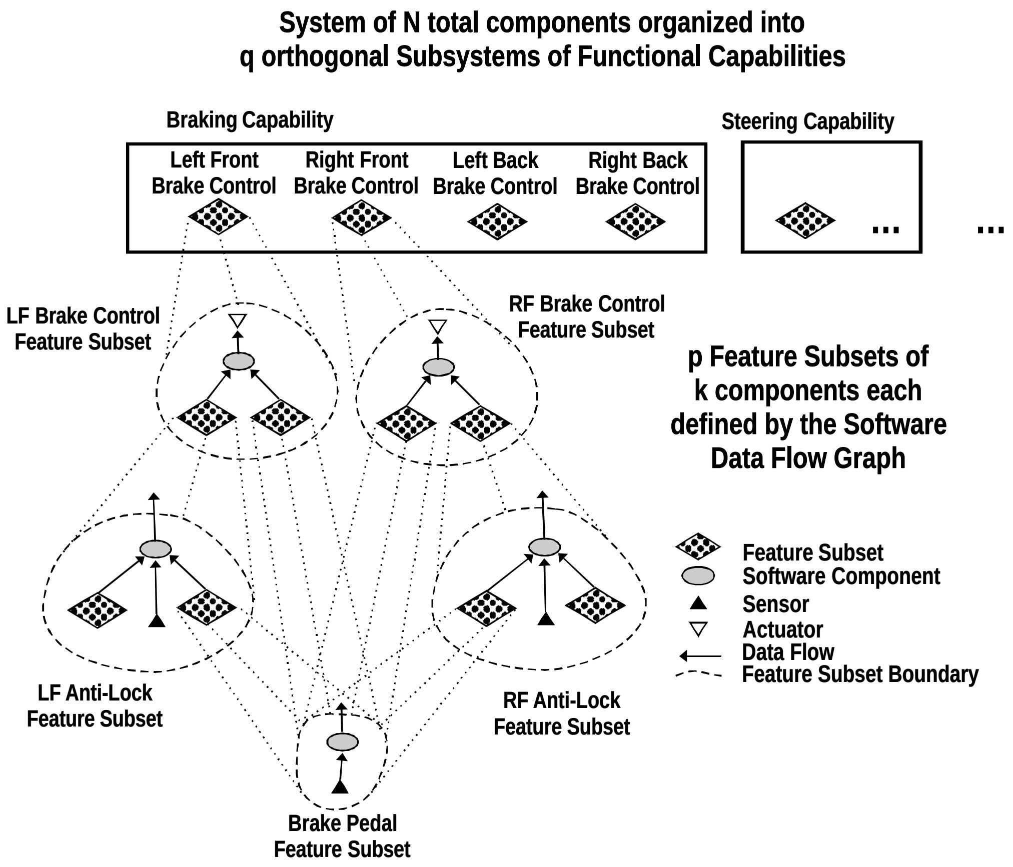
System of N total components organized into
q orthogonal Subsystems of Functional Capabilities
Braking Capability
Steering Capability
Left Front Brake Control
Right Front Brake Control
Left Back Brake Control
Right Back Brake Control
...
...
LF Brake Control Feature Subset
RF Brake Control Feature Subset
p Feature Subsets of k components each defined by the Software Data Flow Graph
Feature Subset
Software Component
Sensor
Actuator
Data Flow
Feature Subset Boundary
LF Anti-Lock Feature Subset
RF Anti-Lock Feature Subset
Brake Pedal Feature Subset

Figure 3.4. Top-down View of System Decomposition into Capabilities.

the capability is then dependent on which of its feature subsets are present in the system. For a capability with t feature subsets, there are 2^t possible configurations of that capability. Figure 3.4 illustrates a top-down view of the system decomposition into functional capabilities and feature subsets for a portion of our brake-by-wire system. The braking capability is one of several functional capabilities an automotive system provides. This braking capability is composed of four braking feature subsets to drive the brake actuators on each wheel of the car. The utility of the braking system will be different depending on which of the four brake control feature subsets are present. Having all four brakes provides maximum utility, but having a single pair of brakes on either the front or back wheels will provide more utility than having a pair of brakes on only the left or right side.

If the feature subsets that provide a capability are not present in the system configuration, then that capability will have a utility of zero. Otherwise each capability will have a utility value based on its feature subset configuration. Thus, for a system with q capabilities, the system utility function can be specified by evaluating the relative utility values of 2^q capability configurations, assuming we have already generated all of the utility functions for all feature subset and capability configurations.

For a system of N software components, sensors, and actuators, one would normally evaluate the relative utility of 2^N system configurations to manually define the system utility function. Using our model, we first evaluate the utility of all configurations of up to k components in each feature subset, which is 2^k configurations for each of p feature subsets. Then, we evaluate the utility of all 2^r configurations of up to r feature subset alternatives in each functional capability, and repeat this for each of q capabilities. To determine the relative utility of system configurations, we evaluate the utilities of 2^q possible configurations of q capabilities. The number of individual utility values that must be assigned (i.e., the complexity of specifying the complete system utility function) is:

$$\textit{(max feature subset configs)} + \textit{(max configs in capabilities)} + \textit{(capability configs)} =$$
$$(p*2^k) \qquad + \qquad (q*2^r) \qquad + \qquad (2^q)$$

For the expected situation of $q, r \leq k$ and $p \leq N$ this utility model requires $O(N*2^k)$ complexity to specify a utility function for 2^N system configurations. Table 3.1 summarizes the parameters of our system utility model. For systems in which $k << N$, meaning that individual subsystems have few components compared to the total

Table 3.1. Key Parameters of the Utility Model.

Parameter	Description
N	Total # of system components (software, sensors, actuators)
p	Total # of feature subsets
q	Total # of system functional capabilities
k	Maximum number of components in any feature subset
r	Maximum number of feature subsets in any capability

number of system components, this utility model enables a scalable definition of the system utility function.

3.5 Scalable Generation of the System Utility Function

If we apply this model to our brake-by-wire subsystem example, we can see the scalability benefits. Our example software system has five sensors (four wheel speed sensors and one brake pedal sensor), four actuators (the four brake actuators), and ten software components (four brake controllers, four anti-lock braking software components, one vehicle dynamics algorithm, one brake pedal controller). This makes a total of 5+4+10 = 19 components, which can have $2^{5+4+10} = 2^{19} =$ 524,288 possible system configurations. We first eliminate all of the invalid component configurations that cannot provide the functionality of at least one brake actuator, but are still left with 89,600 possible valid configurations.

Using our model, there are p = 10 feature subsets in the system (four brake control feature subsets, four anti-lock braking feature subsets, one vehicle dynamics feature subset, and one brake pedal control feature subset). The largest of these feature subsets, vehicle dynamics, has k = 5 components. There are r = 4 braking feature subsets (one for each wheel) that make up the braking system capability. If we were looking at the entire automotive system, braking would be one of several functional capabilities such as steering and acceleration. Since we are only looking at the braking subsystem for this example, q = 1 (i.e., one subsystem). Based on these parameters, the maximum number of subsystem configurations for which we have to assign utility values would be:

$p*2^k + q*2^r + 2^q = 10*2^5 + 1*2^4 + 2^1 = 338$

This is a significant reduction, but we can do better. This calculation gives us the maximum bound assuming that all feature subsets have 5 components each, but if we look at the individual feature subsets, most actually have fewer components. The brake pedal feature subset has two components, each of the four anti-lock braking and brake control feature subsets has four components, and only the vehicle dynamics feature subset has five components. Using this information, we can lower the number of configurations evaluated to:

(feature subset configs) + *(feature subset configs in capabilities)* + *(cap. configs)* =

$1*2^2 + 8*2^4 + 1*2^5 \quad + \quad 1*2^4 \quad + \quad 2^1 \quad = 181$

Furthermore, there are multiple invalid configurations within each feature subset that can be eliminated because they are missing a required component. Using the dependency information in the software data flow view, we can immediately identify these configurations. For example, the brake pedal feature subset will only provide utility if both the brake pedal sensor and brake pedal control software are present in the system. Any other configuration of this feature subset is invalid. Likewise, the vehicle dynamics feature subset cannot provide any utility without the vehicle dynamics software algorithm. Also, at the capability level, we will need at least one functioning braking feature subset out of the four available, and the braking capability must be present in the system to provide positive utility. If we continue this examination for every feature subset, we are left with 1 system capability configuration with a working braking capability, 15 feature subset configurations for the braking capability, and 36 component configurations across the 10 feature subsets for a total of 52 subsystem configurations. We only need to specify the utility values of these 52 subsystem configurations to determine the relative system utility of any of the 89,600 valid system configurations. Once we specify the relative utility of all valid possible component configurations, we can assess how well the system gracefully degrades as components fail.

A system designer with domain knowledge should be able to assign utility values within individual subsystems based on how he or she ranks the different configurations within each subsystem. If we assume that a feature subset's utility is dependent on its functioning components' utility values, we can specify its utility function H_f by generating a separate utility function for each valid feature subset configuration. Table 3.2 shows an example specification for the left front brake control feature subset and its encapsulated feature subsets. Note that our framework requires the system designers to specify the parameters of the utility functions for each feature subset configuration, and our specification for this system is an arbitrary example.

Each configuration is identified by which of the feature subset's components are functioning. The brake pedal feature subset only has one valid configuration in which both of its components must be present, and this by definition provides maximum utility for this feature subset. All other configurations of this feature subset provide zero utility. There are only two valid configurations of the LF

Table 3.2. Example Utility Specification for the LF Brake Control Feature Subset.

Feature Subset	Configuration	Utility Function
Brake Pedal	{Pedal Sensor (u_{ps}), Pedal Controller (u_{pc})}	$U_{Brake\ Pedal} = H_{f1}(u_{ps}, u_{pc}) = 1$
	All other configurations	$U_{Brake\ Pedal} = 0$
LF Anti-Lock	{LF Anti-Lock Brake Control (u_{lfal}), LF Wheel Speed Sensor (u_{lfws}), Brake Pedal Feature, Vehicle Dynamics Feature}	$U_{LF\ Anti\text{-}Lock} = H_{f1}(u_{lfal}, u_{lfws}, U_{Brake\ Pedal}, U_{Vehicle\ Dynamics}) = 0.7 + 0.3{*}U_{Vehicle\ Dynamics}$
	{LF Anti-Lock Brake Control(u_{lfal}) , LF Wheel Speed Sensor(u_{lfws}), Brake Pedal Feature}	$U_{LF\ Anti\text{-}Lock} = H_{f2}(u_{lfal}, u_{lfws}, U_{Brake\ Pedal}) = 0.7$
	All other configurations	$U_{LF\ Anti\text{-}Lock} = 0$
LF Brake Control	{LF Brake Control (u_{lfbc}), LF Brake Actuator (u_{lfba}), LF Anti-Lock Feature, Brake Pedal Feature}	$U_{LF\ Brake\ Control} = H_{f1}(u_{lfbc}, u_{lfba}, U_{LF\ Anti\text{-}Lock}, U_{Brake\ Pedal}) = 0.4 + 0.6{*}U_{LF\ Anti\text{-}Lock}$
	{LF Brake Control, LF Brake Actuator, LF Anti-Lock Feature}	$U_{LF\ Brake\ Control} = H_{f2}(u_{lfbc}, u_{lfba}, U_{LF\ Anti\text{-}Lock}) = 0.4 + 0.6{*}U_{LF\ Anti\text{-}Lock}$
	{LF Brake Control, LF Brake Actuator, Brake Pedal Feature}	$U_{LF\ Brake\ Control} = H_{f3}(u_{lfbc}, u_{lfba}, U_{Brake\ Pedal}) = 0.4$
	All other configurations	$U_{LF\ Brake\ Control} = 0$

anti-lock feature subset; one with the vehicle dynamics feature subset, and one without it. Since we know all other components must be present for the feature subset to provide utility, we do not have to specify the utility function based on their values.

In this hypothetical system, we might determine that the vehicle dynamics data contributes 30% utility to the anti-lock braking algorithm, and specify our utility functions accordingly. Similarly, there are three valid LF brake control feature subset configurations in which either the Brake Pedal feature, LF anti-lock feature, or both are available for the brake controller. Here we assume that the anti-lock braking system contributes 60% utility to the brake control system. Since the LF anti-lock feature subset depends on the brake pedal feature subset, the configuration in which the anti-lock feature is working but the brake pedal feature is not should never occur. However, we still specify its utility in the LF brake control feature subset since we treat them as independent components.

Though not shown in the table, there are 15 possible vehicle dynamics feature subset configuration utility functions that are dependant on which wheel speed sensors are working. These functions cannot be collapsed into a single linear utility function, because the vehicle dynamics feature subset may have more or less utility based on which wheel speed sensors are functioning. For example, the vehicle stability information may be better if both front wheel speed sensors are working than if the left front and right back wheel speed sensors are working.

This utility function specification can be the same for each of the four brake control feature subsets since they are not directly coupled. Then we can specify the utility functions for each of the 15 possible configurations of the braking capability based on the utility values of the four brake control feature subsets. Each braking capability function can be of the form:

$$\begin{aligned} U_{Braking\ System} &= H_{fc}(U_{LF\ Brake\ Control}, U_{RF\ Brake\ Control}, U_{LB\ Brake\ Control}, U_{RB\ Brake\ Control}) \\ &= w_{LF}*U_{LF} + w_{RF}*U_{RF} + w_{LB}*U_{LB} + w_{RB}*U_{RB} \end{aligned}$$

in which any of the brake control feature subsets that have zero utility can be eliminated. The four weights $\{w_{LF}, w_{RF}, w_{LB}, w_{RB}\}$ should be specified with different values for each of the 15 possible valid capability configurations based on the expected behavior of the braking system when the different combinations of brake actuators on each wheel are working.

3.6 Assumptions of Our Model

Our model is never any worse than having to consider 2^N system configurations of N components, and in typical cases will be a significant improvement. To attain these improvements we rely upon several assumptions with regard to how these software systems are designed. First, we assume that the parameters of the utility function for each feature subset configuration are independent of the configuration of any other feature subset in the system. We only define different utility functions for different feature subset configurations, in which a configuration specifies whether a component is present and working (providing positive utility) or absent and failed (providing zero utility).

When a feature subset is treated as a component in a higher-level feature subset, that component can potentially have different utility values based on its current configuration, rather than just 1 for working and 0 for failed as with individual software components, sensors, and actuators. This could potentially mean that in order to define the higher-level feature subset's utility function, we would have to define a different utility function for every possible utility value for every feature subset contained as a component in the higher-level feature subset. However, this is only necessary if the encapsulated feature subsets are strongly coupled within higher level feature subsets. Similarly, individual components could have utility values other than 0 or 1 if a different fault model were applied that allowed partial component utility. These components could be modeled as "logical" feature subsets with an internal utility specification, and would not affect the number of feature subset configurations required to specify the system utility function.

Because system architects generally attempt to decouple subsystems to the degree possible, we assume that encapsulated feature subsets are not strongly coupled. Additionally, the high-level capabilities could also be coupled. For example, if a braking capability has degraded utility, it might mean that high utility in a steering capability is worth much more to the system than if the brakes were functioning normally. If some subsystems are strongly coupled, one could apply multi-attribute utility theory [Keeney76, Keeney92] to deal with the added system complexity within the model. In the worst case, if it is not possible to separate utility evaluations across feature subsets, we can still confine our utility evaluation to valid system configurations rather than all 2^N possible configurations.

Defining the system functional capabilities requires grouping the system feature subsets according to the functionality they provide. The feature subsets within a capability may be functionally equivalent and represent system-level redundancy, or they may coordinate their functionality to provide a general system service. In our brake by wire example, the four brake control subsystems are viewed as isolated feature subsets that contribute individual utility to system braking ability, but they can also be viewed as feature subsets that coordinate their behavior to provide enhanced functionality. In this view, each brake control feature subset definition would be modified to reflect that each brake controller software component receives the system variable outputs of the other three brake control feature subsets as optional inputs. Although the number of valid system configurations does not change, this would increase the number of valid configurations in each brake control feature subset that must be specified from 3 to 24, to account for each case of how an

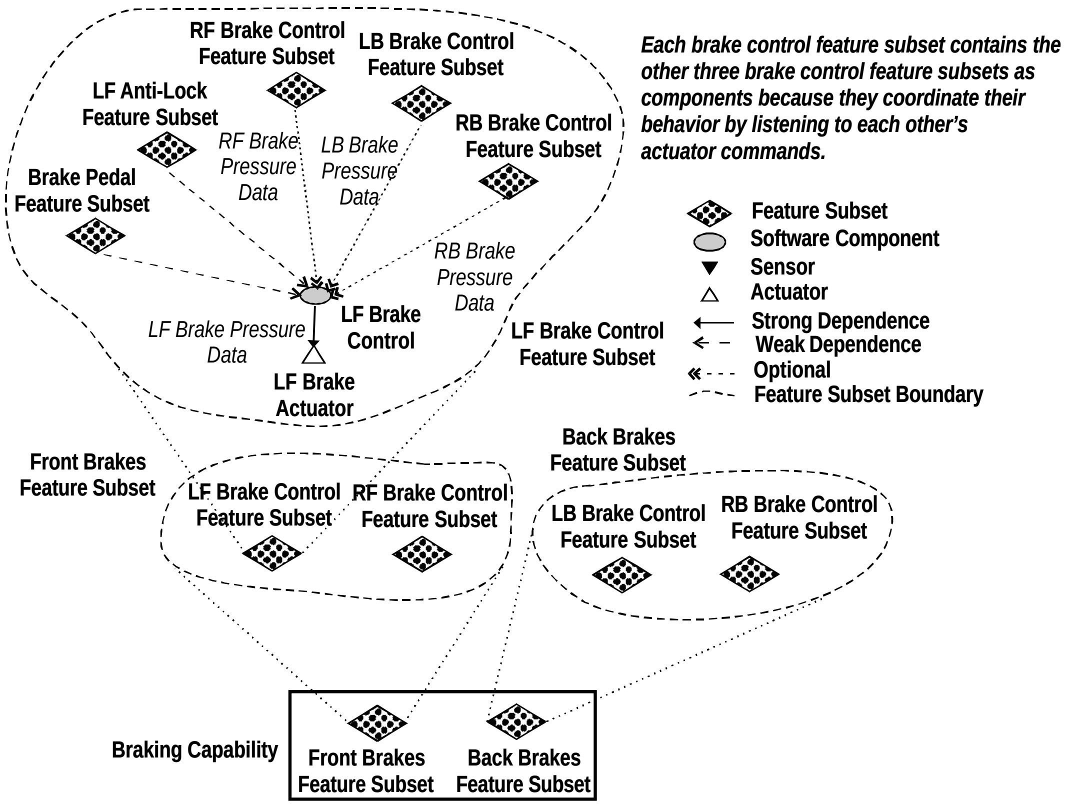
RF Brake Control Feature Subset
LB Brake Control Feature Subset
LF Anti-Lock Feature Subset
RF Brake Pressure Data
LB Brake Pressure Data
RB Brake Control Feature Subset
Brake Pedal Feature Subset
RB Brake Pressure Data
LF Brake Control
LF Brake Pressure Data
LF Brake Actuator
LF Brake Control Feature Subset
Each brake control feature subset contains the other three brake control feature subsets as components because they coordinate their behavior by listening to each other's actuator commands.
Feature Subset
Software Component
Sensor
Actuator
Strong Dependence
Weak Dependence
Optional
Feature Subset Boundary
Front Brakes Feature Subset
LF Brake Control Feature Subset
RF Brake Control Feature Subset
Back Brakes Feature Subset
LB Brake Control Feature Subset
RB Brake Control Feature Subset
Braking Capability
Front Brakes Feature Subset
Back Brakes Feature Subset

Figure 3.5. Alternate Brake-by-Wire System Feature Subset Organization.

individual brake controller subsystem's utility changes when the other subsystems are lost.

It might also be reasonable to group the four brake control feature subsets into two front and back feature subsets since these subsystems are coupled by wheel axle. These two feature subsets would each have 4 possible configurations (3 of which are valid) of their two feature subset components. Then the brake control capability would have 4 possible (and 3 valid) configurations of the two front and back brake control feature subsets. This alternative feature subset organization is shown in Figure 3.5. This capability definition may more accurately reflect how the braking subsystems are related, and aid the system designer in constructing a more accurate utility model. However, this also requires specifying a total of 130 configuration utility functions rather than 52. This example illustrates the tradeoff between the expressiveness of the system utility function and the detail of the functional capability specification. A more detailed capability specification may more accurately model how each feature subset affects system utility, but will require that more configurations be evaluated to specify the system utility function.

We also assume that the system is "well-designed" such that combinations of components do not interact negatively with respect to feature subset or system utility. In other words, when a component has zero utility, it contributes zero utility to the system or feature subset, but when a component has some positive utility, it contributes *at least* zero or positive utility to the system or feature subset, and never has an interaction with the rest of the system that results in an overall loss of utility. Thus, working components can enhance but never reduce system utility. We assume that if we observe a situation in which a component contributes negative utility to the system, we can intentionally deactivate that component.

Our utility model only deals with software system configurations, and we do not directly account for hardware redundancy as a system utility attribute. However, in general hardware redundancy mechanisms will not affect system functionality, but rather hardware system reliability or availability. Since we have separated the hardware and software views of the system, we can still perform traditional dependability analysis on the system's hardware configuration. To analyze tradeoffs between system functionality and dependability, we could again apply multi-attribute utility theory to judge the relative value of the software configuration's utility and the hardware configuration's reliability and availability to the system’s overall utility. This analysis may include factors such as system resource costs and hardware and software failure rates.

3.7 Traditional Fault-Tolerance Techniques

Our goal is to provide a common representation of system component configurations that can be used to analyze system-wide utility and graceful degradation. For our model to be useful, it must be readily applicable to current software system design techniques and architectural approaches. Because our model is directly generated from component and interface definitions, we should be able to apply it to systems early in the design process.

Although our model emphasizes graceful degradation, it must also be able to represent common dependability techniques. Beyond that, it is desirable for a model of graceful degradation to have commonly used fault-tolerant computing techniques as special cases of graceful degradation. We demonstrate elements of the generality of our system model by showing how hardware redundancy, software fault tolerance techniques (described in [Lyu95]) such as recovery blocks, multi-version software redundancy, self-checking programming, analytic redundancy, and the simplex architecture can be represented in our model.

3.7.1 Hardware Redundancy

Hardware components that replicate identical copies of software can mask hardware faults and prevent them from affecting the system. Hardware redundancy treats replicated software components as one logical component in the software system.

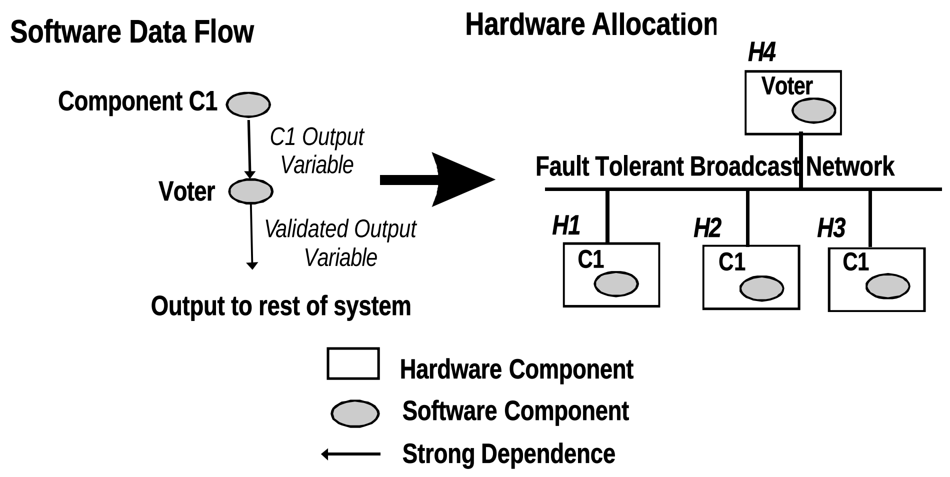

Figure 3.6. Hardware TMR in the Data Flow Graph and Allocation Views.

Therefore hardware redundancy is expressed only within the hardware allocation diagram within our model. Figure 3.6 shows how we represent hardware triplex modular redundancy (TMR) [Rennels84] in our system model. Three redundant software components logically represent only one component within the software data flow graph, but that software component is mapped to three processing elements in the allocation diagram. In the implementation, these components would attach a node ID to their output variables when they are sent over the network so that the voter component could distinguish between the different sources of the output, but these redundant component outputs represent the same system variable type in the software data flow graph.

Hardware redundancy can be combined with software fault tolerance techniques to improve system dependability. Using two different views for hardware and software fault tolerance techniques permits a separate analysis of these two approaches. The next few sections describe how our model represents current software fault tolerance techniques.

3.7.2 Recovery Blocks and Temporal Redundancy

Recovery blocks [Randell75] use checkpointing to prevent errors in computations from corrupting system state. A snapshot is made of system state before a computation. If the computation fails or its output fails the acceptance test, system state can be restored to the checkpoint and the computation can be retried. Multiple alternate algorithms are used to avoid having the same computation fail repeatedly. If the first alternate fails, the system is rolled back to the checkpoint, and the next algorithm is executed until there is a successful completion. If all of the alternates fail, then the subsystem reports a computation failure. If the recovery block only has one algorithm for computation, and executes it multiple times, this special case represents temporal redundancy.

Temporal redundancy takes advantage of the transient nature of some faults. In a time-triggered embedded system, a transient component fault can be automatically tolerated because a missed output for one period will be recovered in the next period. It is only when output values become stale (no new value for several periods) that a fault manifests as a failure. In an event triggered system, a component may receive requests as inputs to provide its outputs. In this case, if a component does not provide its outputs, the component that sends the request as input can retry the operation to tolerate a transient fault. In our model, this would be represented as a cycle within the data flow graph between the component that makes the request and the component that outputs a response.

Distributed recovery blocks replicate some or all of the alternate algorithms across multiple hardware nodes, requiring mechanisms to synchronize state

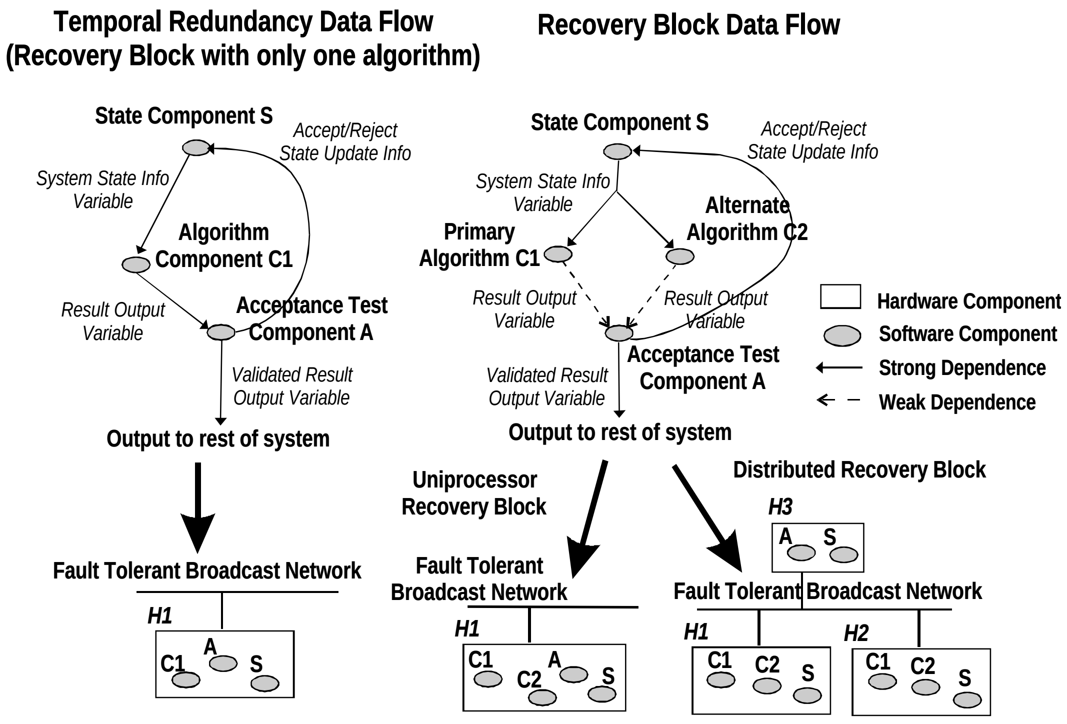
Temporal Redundancy Data Flow
(Recovery Block with only one algorithm)
Recovery Block Data Flow
State Component S
Accept/Reject State Update Info
System State Info Variable
Algorithm Component C1
Result Output Variable
Acceptance Test Component A
Validated Result Output Variable
Output to rest of system
Fault Tolerant Broadcast Network
H1
C1
A
S
State Component S
Accept/Reject State Update Info
System State Info Variable
Primary Algorithm C1
Alternate Algorithm C2
Result Output Variable
Result Output Variable
Acceptance Test Component A
Validated Result Output Variable
Output to rest of system
Hardware Component
Software Component
Strong Dependence
Weak Dependence
Uniprocessor Recovery Block
Distributed Recovery Block
H3
A
S
Fault Tolerant Broadcast Network
H1
C1
C2
A
S
Fault Tolerant Broadcast Network
H1
C1
C2
S
H2
C1
C2
S

Figure 3.7. Temporal Redundancy and Recovery Block Model Descriptions.

between the various alternates. Figure 3.7 shows examples of a recovery block and a distributed recovery block, as well as a simple example of temporal redundancy.

3.7.3 Multi-Version Software Redundancy

Multi-version software redundancy (also known as N-version redundancy) [Avizienis85], is represented in the data flow dependency graph, but not necessarily in the allocation diagram. If three software components implement the same input and output interfaces, they can provide software redundancy and are represented as three software components in our data flow graph that have the same input and output system variables. A voter component receives the outputs of the different software component versions and uses majority voting to determine the correct

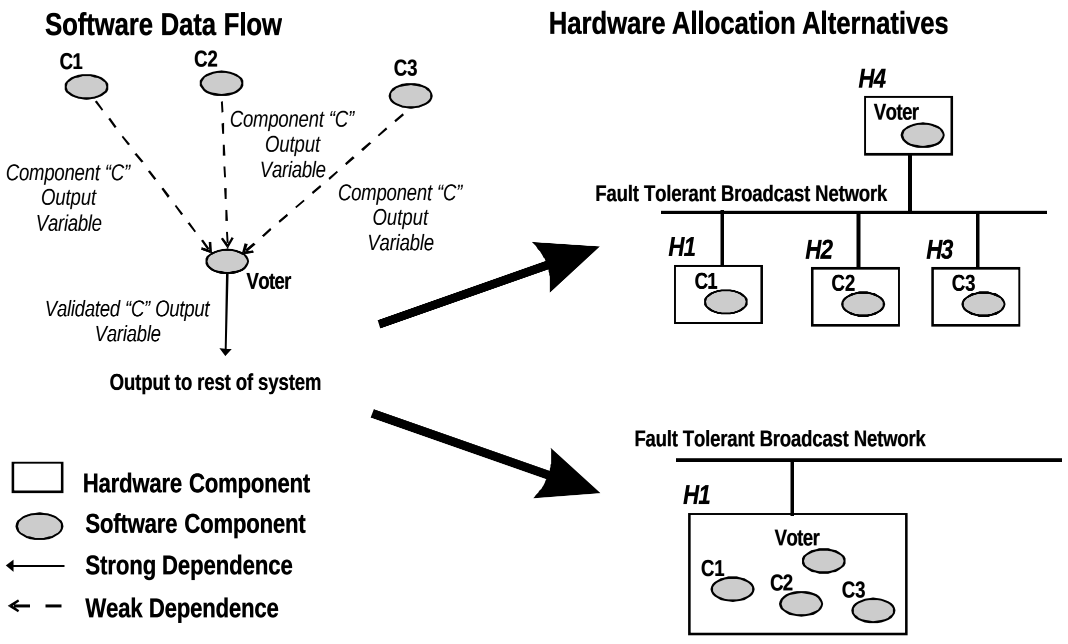
Software Data Flow
C1
C2
C3
Component "C" Output Variable
Component "C" Output Variable
Component "C" Output Variable
Voter
Validated "C" Output Variable
Output to rest of system
Hardware Component
Software Component
Strong Dependence
Weak Dependence
Hardware Allocation Alternatives
H4
Voter
Fault Tolerant Broadcast Network
H1
H2
H3
C1
C2
C3
Fault Tolerant Broadcast Network
H1
Voter
C1
C2
C3

Figure 3.8. Multi-Version Software Redundancy.

output to send to the rest of the system. These components may all be allocated to the same processing element, or be distributed across multiple processors. Software redundancy will affect system utility measured by our model if the individual components implement different algorithms that provide different levels of quality in their outputs.

Multi-version software redundancy schemes are designed primarily to prevent software defects from causing system failures. According to this methodology, independently designed and verified software components should not share similar or identical software defects, and should not be susceptible to similar software failures. Thus, these components should provide higher reliability by serving as redundant backups for one another's software defects. Figure 3.8 shows an example of multi-version software redundancy in the data flow graph and hardware allocation views of our model. As shown in the figure, this scheme could either have its components distributed across multiple nodes, or they could be allocated to the same node. If the primary dependability concern is software defects rather than hardware faults, it might be cost effective to spend the extra design effort on multiple software versions while conserving hardware costs by allocating all of the components to the same node.

3.7.4 Self-Checking Programming

N Self-checking programming [Laprie87] can take advantage of multi-version software redundancy and recovery blocks to increase overall reliability. N Self-checking programming has multiple algorithms that run in parallel with the results passed to a voter. Each variant itself is a set of components that cross check their results against each other before passing them to the voter component. These interior components can be implemented as another multi-version redundancy scheme or as a recovery block. Using this technique, software components can be organized into a hierarchy that keeps faults from propagating across multiple alternates.

Figure 3.9 shows an example of this hierarchy in a software system as well as a possible hardware configuration in our model. As shown in the figure, each self-checking component is itself a feature subset that contains two components that provide the same outputs. Within each self-checking component feature subset, the results of the algorithms are compared against each other to detect an error. Only when the results agree will their outputs be passed on to be output by the self-checking feature subset. A voter component compares all of the results of the

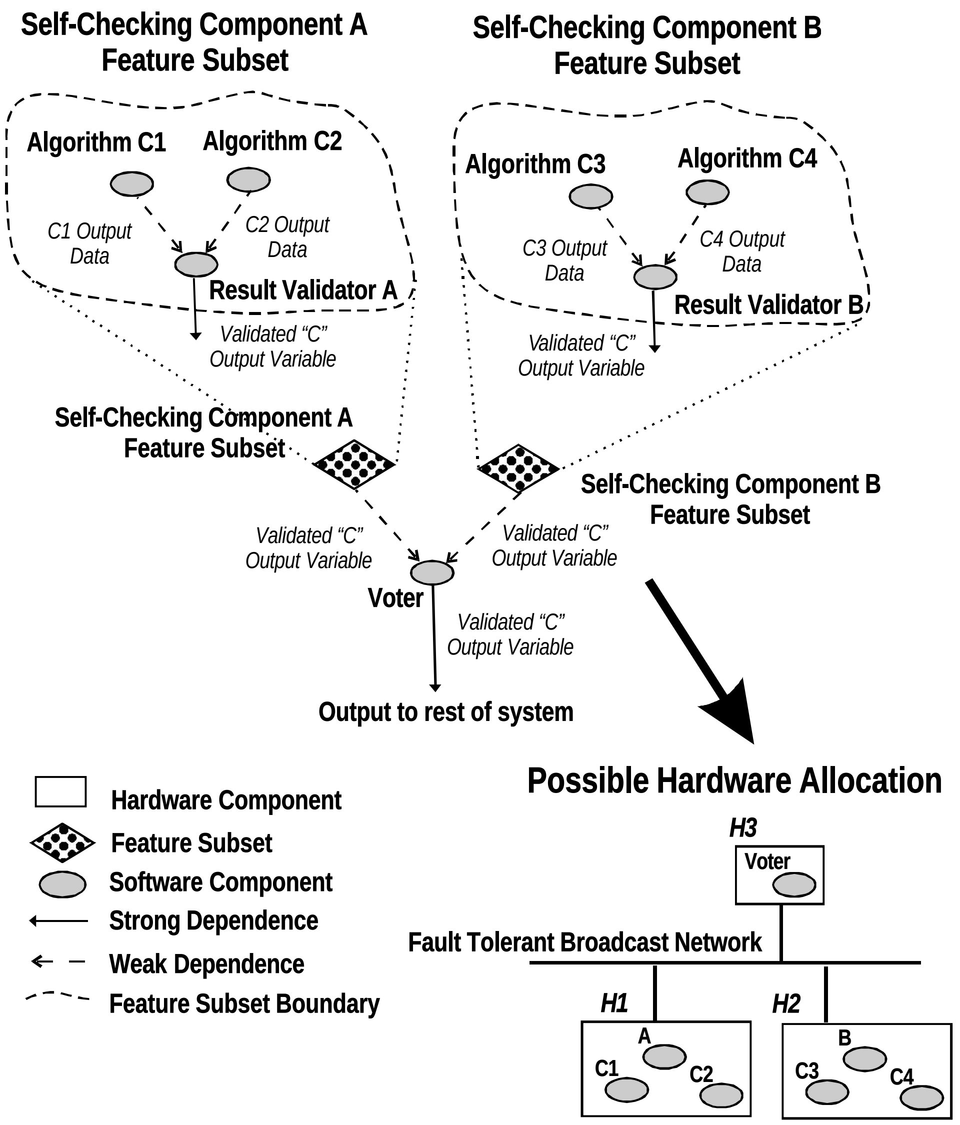
Self-Checking Component A Feature Subset
Self-Checking Component B Feature Subset
Algorithm C1
Algorithm C2
C1 Output Data
C2 Output Data
Result Validator A
Algorithm C3
Algorithm C4
C3 Output Data
C4 Output Data
Result Validator B
Validated "C" Output Variable
Validated "C" Output Variable
Self-Checking Component A Feature Subset
Self-Checking Component B Feature Subset
Validated "C" Output Variable
Validated "C" Output Variable
Voter
Validated "C" Output Variable
Output to rest of system
Hardware Component
Feature Subset
Software Component
Strong Dependence
Weak Dependence
Feature Subset Boundary
Possible Hardware Allocation
H3
Voter
Fault Tolerant Broadcast Network
H1
H2
A
C1
C2
B
C3
C4

Figure 3.9. Self-Checking Software With Hierarchical Component Organization.

self-checking components to provide an output. In hardware, components may be allocated to nodes according to which self-checking feature subsets they comprise.

3.7.5 Analytic Redundancy

Analytic redundancy [Patton93] allows the system to take advantage of multiple sources of heterogeneous information. For example, if a system has sensors for measuring the temperature, pressure, and volume of a gas, loss of any one sensor

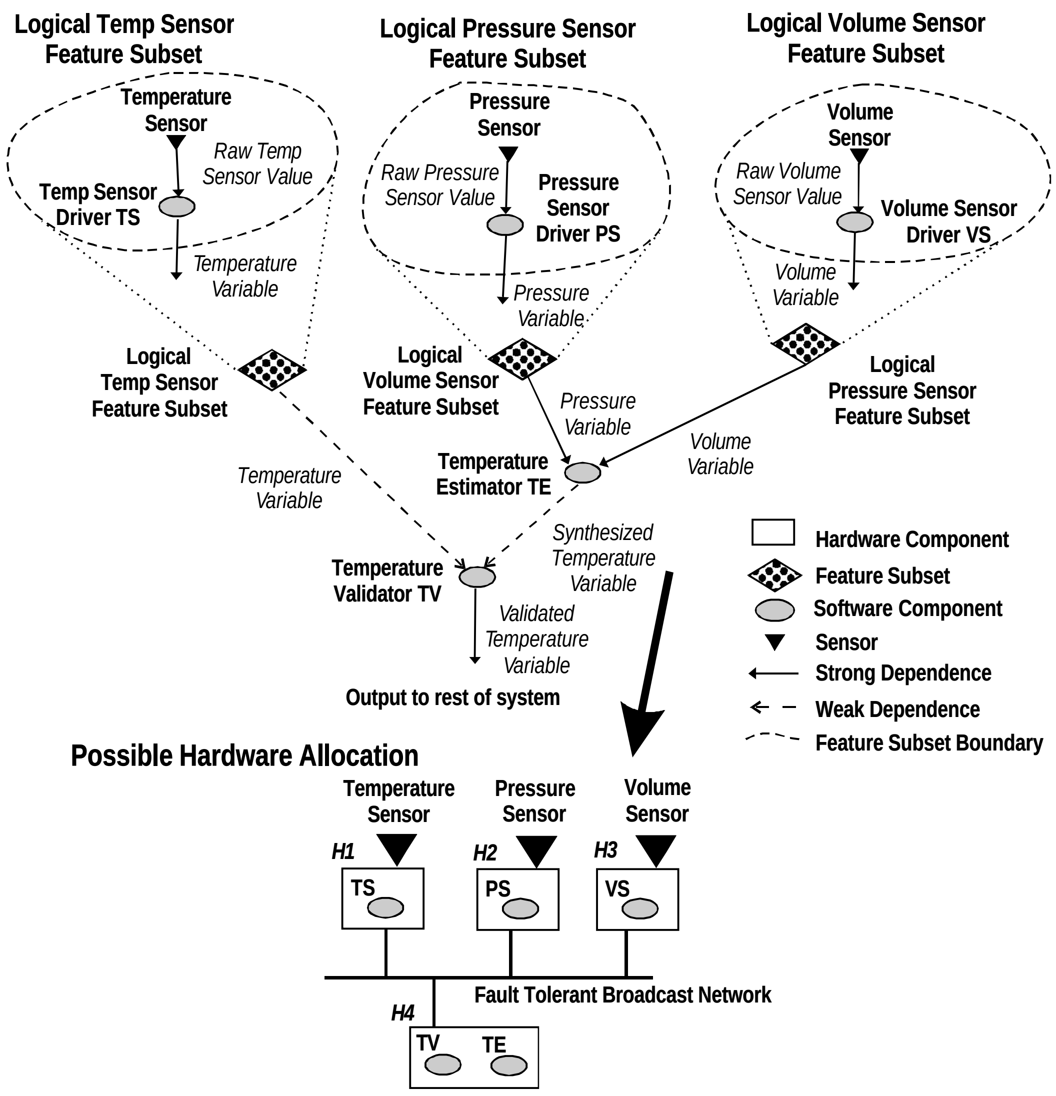

Figure 3.10. Analytic Redundancy to Tolerate a Temperature Sensor Failure.

input can be mitigated by creating a synthetic sensor value based on the other two sensors. The synthesized input may not be as accurate as a working sensor, but will still provide some level of functionality and generally cost less than redundant sensors.

Figure 3.10 illustrates a representation of analytic redundancy in our system model. In the figure, there are three sensors that each measure a different aspect of the environment (temperature, pressure, and volume). Each of the sensors provides different data and are not functionally equivalent, so they cannot provide traditional redundancy. However, the values these sensors measure are physically related to one another, and the value of one sensor (e.g. temperature) can be estimated by using the values of the other two sensors (volume and pressure). Thus, in the event of a temperature sensor failure, the temperature sensor data can be synthesized by using a software component estimator to process the other two sensors' data.

This example displays how graceful degradation mechanisms can provide tradeoffs between high dependability and constrained system resources. If the system used a brute force hardware replication strategy of dual-redundant hardware sensors, then the system could tolerate as many as three sensor failures, but at a high component cost. If the system designer is only concerned about tolerating the failure of one temperature sensor, analytic redundancy is a more affordable choice. When the sensor is lost, the rest of the system continues to synthesize the missing data, but it is degraded with respect to its accuracy.

One of the benefits of our model as demonstrated in this example is that it can show where there are opportunities to provide additional redundancy and fault tolerance with existing system resources. The software components communicate via well-defined system variable interfaces, and components that output similar or identical system variables can be used as redundant components even if their primary functionality is different.

3.7.6 Simplex Architecture

The simplex architecture [Bodson93] is a control system architecture for using design diversity to improve the reliability of a software control system, and provide some level of graceful degradation. It explicitly defines tradeoffs between low-performance, more reliable controllers that are less likely to introduce design defects, and high performance controllers that may contain more residual defects. Rather than develop multiple versions of software from the same specification and with the same requirements as in traditional multi-version software redundancy, the simplex architecture requires at least two different control algorithms with different specifications and requirements to be implemented as separate software controllers. One control algorithm is specifically designed so that it is as simple and reliable as possible and its control laws can be easily verified. These requirements lead to a software controller that sacrifices high performance for reduced complexity and fewer residual defects. A second control algorithm is designed to provide high-performance control at the cost of higher complexity and possibly more software defects. Within the control system, both controllers receive inputs from the same sensors, and an acceptance test decides whether to use the output from the high-performance, complex controller, or the simple, reliable controller. If the high-performance controller's output fails the acceptance test (possibly due to a software defect), the output from the simple controller can be used to maintain system stability with lower system performance.

The simplex architecture can be readily represented within our system model, as shown in Figure 3.11. Each control algorithm can be represented as a separate feature subset that receives data from the same sensors in the software data flow view. The acceptance test component is responsible for issuing commands to the actuator, and will only use the high-performance controller output if it passes validation. Otherwise, it will always use the output from the simple controller.

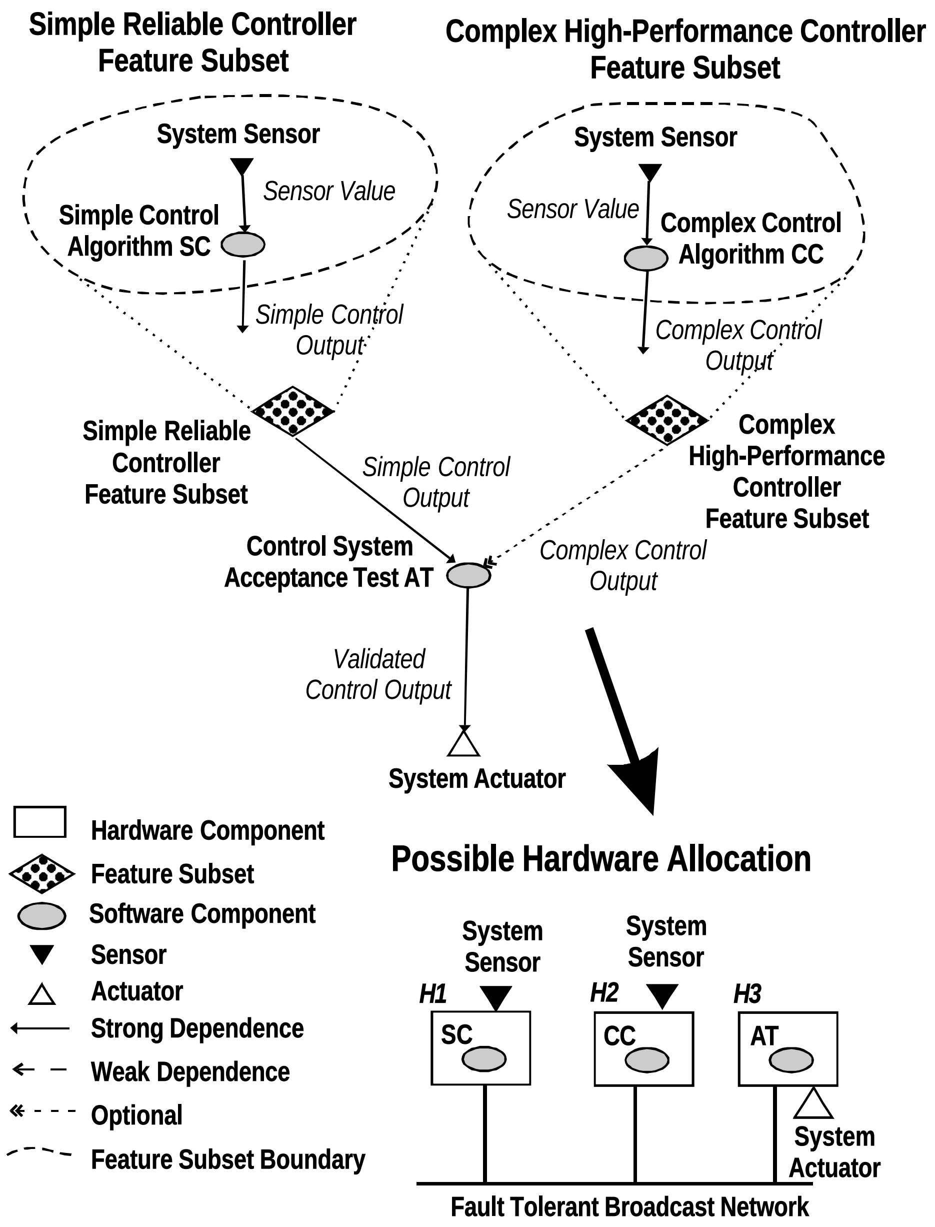

Figure 3.11. Simplex Architecture in the Software and Hardware Views.

Thus it treats the high-performance output as optional, and is strongly dependent on the output from the simple controller. In hardware, the different control algorithms may be allocated to different processors with their own sensors, with their control outputs broadcast on the network to be received by the acceptance test software component which outputs to the actuator.

3.8 Conclusions

Our system model provides a scalable approach to determining how well a system gracefully degrades. Since individual component failures simply transform the system from one configuration to another, we can evaluate how well the system gracefully degrades by observing the utility differences among valid system configurations. By exploiting the fact that systems are decomposed into subsystems of components, we can reduce the complexity of determining the utility function for all possible system configurations from $O(2^N)$ to $O(N*2^k)$, where N is the total number of software components, sensors, and actuators in the system, and k is the maximum number of components in any one subsystem. Data dependency relationships among components enable efficient elimination of invalid configurations from our analysis.

Our model consists of a software data flow graph for determining dependency relationships among software components, sensors, and actuators; a hardware allocation diagram that provides information about hardware replication; and a utility model that provides a framework for comparing the relative utility of system configurations. Since feature subset definitions are based on component input and output interfaces, they can be automatically generated from the software system data flow graph. We allow multiple feature subsets that require the same input system variable from another component to share that component. Feature subsets are in general not disjoint, and a component or feature subset encapsulated in one high-level feature subset may belong to several other feature subsets. This allows us to decouple subsystem utility analyses within our model, even if the system itself does not completely encapsulate its subsystems into a strict hierarchy.

We have shown that the model can represent traditional fault-tolerance mechanisms and evaluate how they affect system-wide graceful degradation. Since we have two orthogonal views of the software and hardware structure of the system, we can consider the effects of hardware and software replication separately. Hardware replication will affect the reliability of software components with respect to hardware failures, but will not affect system functionality and will be largely invisible to the software system. Software replication may affect system functionality as different software components that output the same interface may implement algorithms that output different levels of data quality. Our system model provides a common representation of heterogeneous redundancy mechanisms, as well as a scalable technique to evaluate how these mechanisms may affect system utility.

In the following chapter, we will apply this model to a more complex example of a distributed embedded system architecture. This example will drive our identification of architectural properties that contribute to graceful degradation. We will also show how we can use the model to identify parts of the system that may benefit from graceful degradation mechanisms, analyze the effectiveness of a system’s graceful degradation mechanisms, and evaluate whether the system implementation achieves the level of graceful degradation predicted by the model.

4 Architectural Properties for Graceful Degradation

Now that we have a scalable model for specifying graceful degradation, we can use it to identify likely architectural properties that improve a system's ability to gracefully degrade. A system's ability to gracefully degrade will improve with the number of possible valid configurations it can have, as well as with smaller differences in system utility between different configurations. Thus, properties that tend to increase the number of valid configurations within feature subsets and also tend to reduce the differences in utility provided by different feature subset configurations should make a system more gracefully degradable in the presence of multiple component failures. Our system model should provide a means to explicitly identify these properties.

Our model is designed primarily for examining the software organization of distributed embedded systems, with a lesser focus on its complimentary hardware and communication structures. Therefore, we focus on the system's software architecture in terms of component and connector [Shaw96] organization for mechanisms that should improve graceful degradation at the application level.

In this chapter we will use a typical example system that was specifically designed to have multiple graceful degradation opportunities. We will apply our model to this system's architecture and identify the properties that contribute to making this system gracefully degradable. Our goal is to develop a set of general techniques that should improve graceful degradation that can be applied across this class of distributed embedded systems.

The example system is the system architecture of a hypothetical automobile navigation system. It was originally designed as an example problem to drive the development of a hardware allocation and reconfiguration algorithm in [Nace2002]. It has many heterogeneous software components that have alternate means of providing system functionality, and should provide multiple graceful degradation opportunities. The system was designed as a product family architecture (PFA) in which different valid hardware/software configurations constituted different versions of the navigation system with differing utility values. Failure or addition of components moves the system from one product instance to another.

The original problem for which this system architecture was designed involved building an algorithm that could allocate software components to limited hardware resources to provide maximum system utility. Thus, the work in [Nace2002] was only concerned with finding valid software configurations that fit on available hardware and not with identifying all possible valid software configurations. Since the previous work focused on reconfiguration mechanisms rather than having backup redundancy available in the system, the allocation algorithm considered software components that provided the same functionality as mutually exclusive and were not allocated to the same configuration. Also, sensors and actuators were tied to hardware configurations and not considered as part of the software configuration. Thus, the view of software configurations in [Nace2002] was significantly constrained, and they manually assigned utility values to different software configurations, which were used by the reconfiguration algorithm to evaluate different allocations of software to hardware.

Our goal is to specify the relative utility of all possible software configurations in order to evaluate the ability of the system's software architecture to gracefully degrade. Therefore, we will not look at possible system hardware configurations, but rather assume that there are enough hardware resources available to support any possible software configuration. We will take the original PFA specification and build our system model to specify the system's feature subsets.

4.1 System Description

Our example is an automobile navigation system that provides turn-by-turn directions to the driver to his or her desired destination. The navigation system draws information from the vehicle's sensors to determine the car's current position, and uses a map database and path planning algorithms to determine what the driver's next action should be. The system can then provide feedback to the driver either through a color display in the car that provides visual output of the directions, through audio cues for turns via the car radio speaker, or through turning hints displayed by the car's turn signal indicators.

Figure 4.1 shows a data flow graph view of the system's software architecture. In the figure, there are multiple sensors, such as a GPS (global positioning system) sensor, engine sensor, or compass sensor, that can provide varying levels of information about the car's position and direction. The actuators available include the turn signal indicator, speaker, and display that can output directions to the driver. The adapters and features in the data flow graph represent software components that process the sensor inputs and provide outputs. The feature classes represent the functional subsystems they identified within the navigation system. We will not use these features in our model, but rather identify a set of feature subsets based on the data flow of the system.

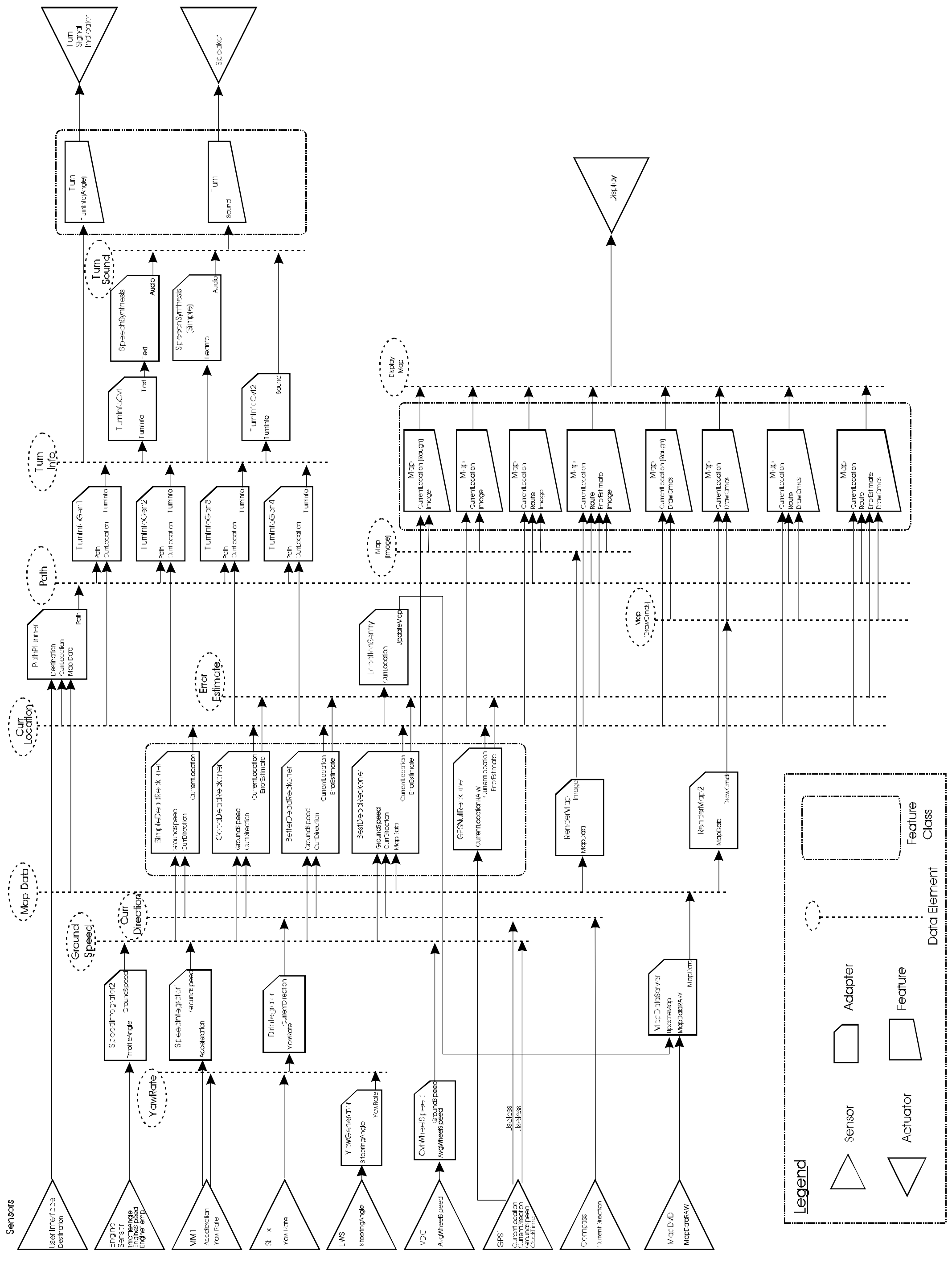

Figure 4.1. PFA Graph of the Navigation System from [Nace2002] (Used With Permission).

4.2 Specification of the System Utility Function

Since the system architecture is expressed as a data flow diagram, we can directly apply our system model and identify feature subsets. The data elements in Figure 4.1 will become our system variable definitions. We identified 22 system variables that are directly generated from the data elements described in the PFA graph (all system variable names are in italics): *AvgWheelSpeed, Acceleration, ThrottleAngle, SteeringAngle, YawRate, GroundSpeed, CurrentDirection, CurrentLocationRaw, CurrentLocation, MapDataRaw, MapData, UpdateMap, DesiredDestination, PathInfo, TurnInfo, TurnSound, TurnText, SpeakerCommands, TurnSignalCommands, MapDrawCommands, MapImage,* and *DisplayMap*. Note that we combined the current location and error estimate data elements into the single *CurrentLocation* system variable, since the error estimate can be implemented as an attribute of the location data. The sensors, actuators, adapters, and features will become our system components. There are 9 sensors, 3 actuators, and 33 software components for a total of 45 system components. Without our model this would require manually evaluating the relative utility of $2^{45} \approx 4 * 10^{13}$ system configurations. If we eliminate all of the invalid system configurations, there are approximately $6 * 10^{11}$ valid possible configurations for which we still must specify relative system utility values.

We will apply our system model to the data flow graph starting at the system actuators and working backwards to generate the system's feature subsets. At the system level, there are three functional capabilities (Turn Signal, Speaker, Display) derived from the three actuators available in the system that provide user functionality. The driver will receive navigation information as long as at least one

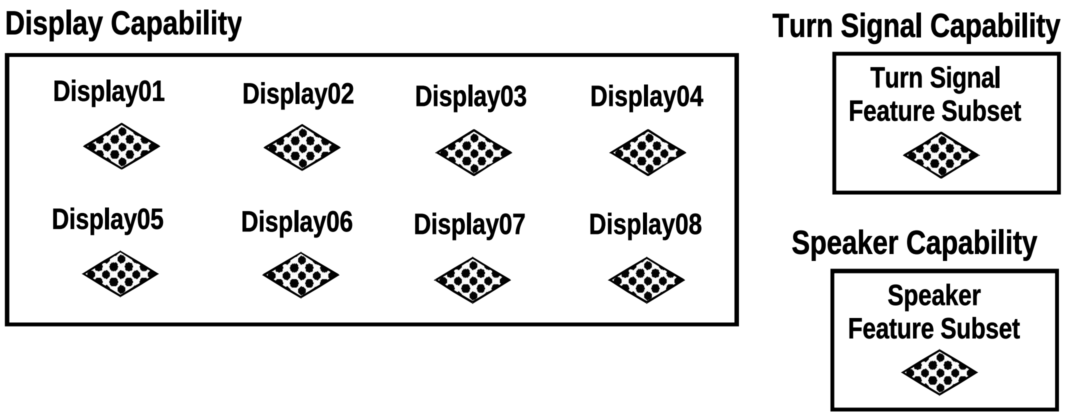

Figure 4.2. System-Level Functional Capabilities.

of these three capabilities provides positive utility. Figure 4.2 shows the top level functional capabilities and the feature subsets of which they are composed. The Turn Signal and Speaker capabilities each contain one feature subset, while the Display capability has eight feature subsets that can provide display functionality.

Figures 4.3 and 4.4 show definitions of the feature subsets which are encapsulated by the system-level capabilities. Figure 4.3 details the hierarchical feature subset definitions for the Turn Signal and Speaker feature subsets. Figure 4.4 defines the feature subsets from two of the eight available Display feature subsets. Based on the feature subset dependencies, we can see that for the system to provide any utility in a given configuration, that configuration must contain enough components for working Location and Map Data feature subsets. The Turn Signal and Speaker feature subsets depend on the TurnInfo feature subset, which in turn depends on the Path Planner and Location feature subsets. The Path Planner feature subset then also depends on the Location and Map Data feature subsets. Each of the eight possible Display feature subsets depends on at least a working Location and MapData feature subset.

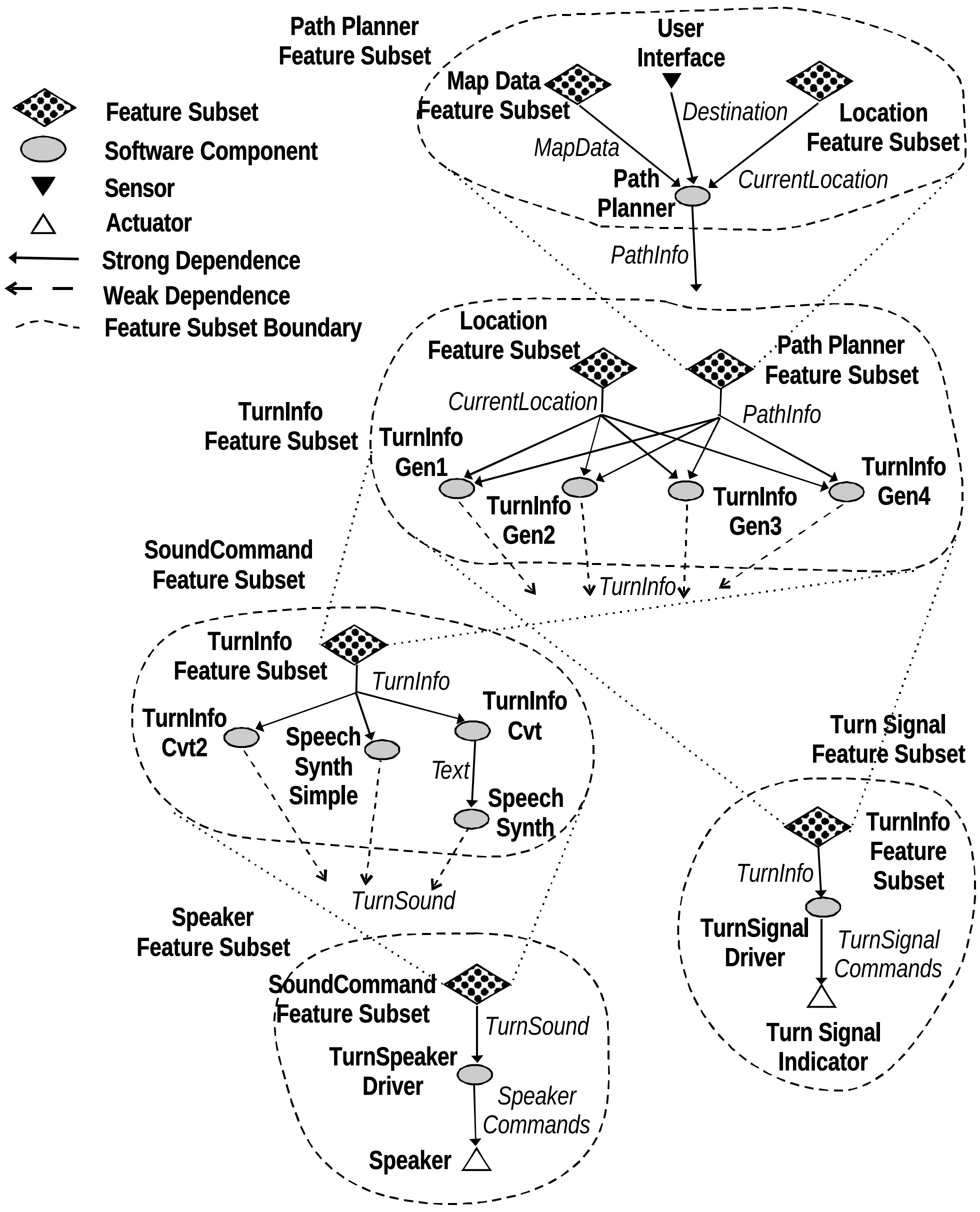

Figure 4.3. Expansion of Turn Signal and Speaker Feature Subsets.

Since the system functionality heavily depends on providing valid location data, it is not surprising that the designers focused their efforts on providing several levels of heterogeneous redundancy for this subsystem. Figure 4.5 shows the Location feature subset in detail. In addition to using the GPS sensor to get accurate location data, the system has several dead reckoning algorithms available in the event that GPS location data is lost. Similarly, the dead reckoning algorithms require both

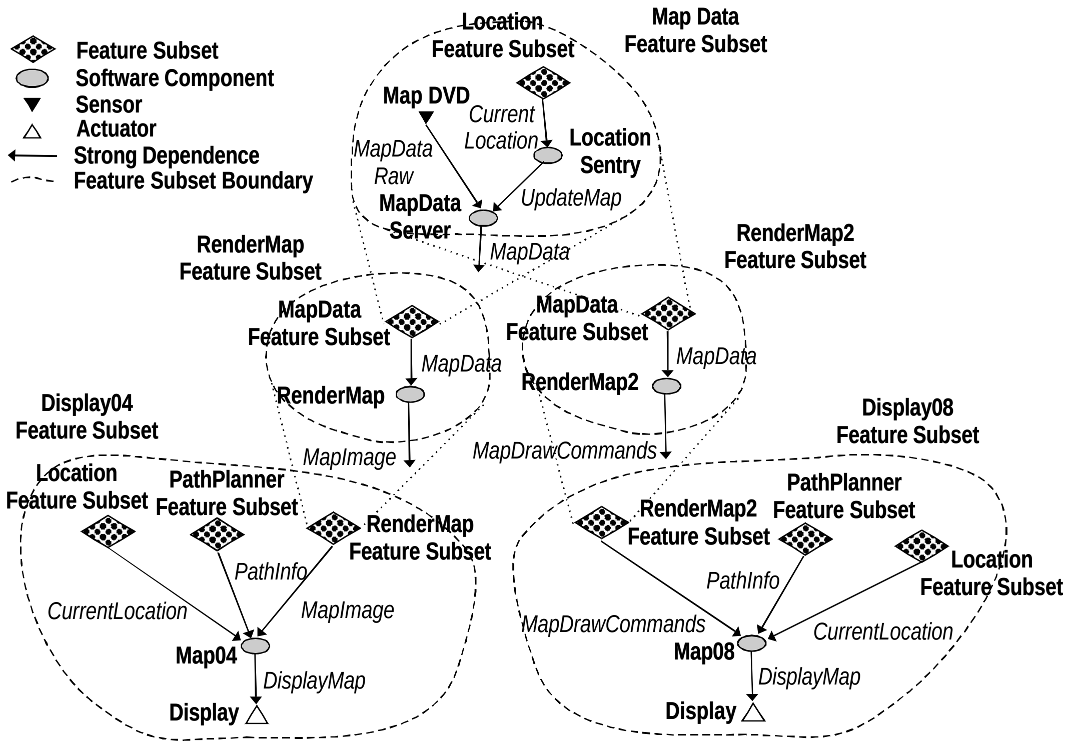

Figure 4.4. Partial Expansion of Display Feature Subsets.

speed and location data, which can be derived from multiple sensors within the car with a bit of data transformation. The graceful degradation of the location subsystem manifests in the loss of accuracy in the location data provided when the high accuracy components fail. This may reduce the effectiveness of the system's path planning navigation algorithms, making the directions that are provided to the user through the system actuators less accurate.

We were able to completely specify the navigation system with 24 feature subsets, the largest of which had 6 components, and 3 functional capabilities that encompass all feature subsets. This means there is an upper bound of $24 * 2^6 = 1536$ feature subset configurations that must be evaluated to specify the utility functions for all feature subsets. However, this assumes that all feature subsets have the maximum 6 components, and that all feature subset configurations are valid. We

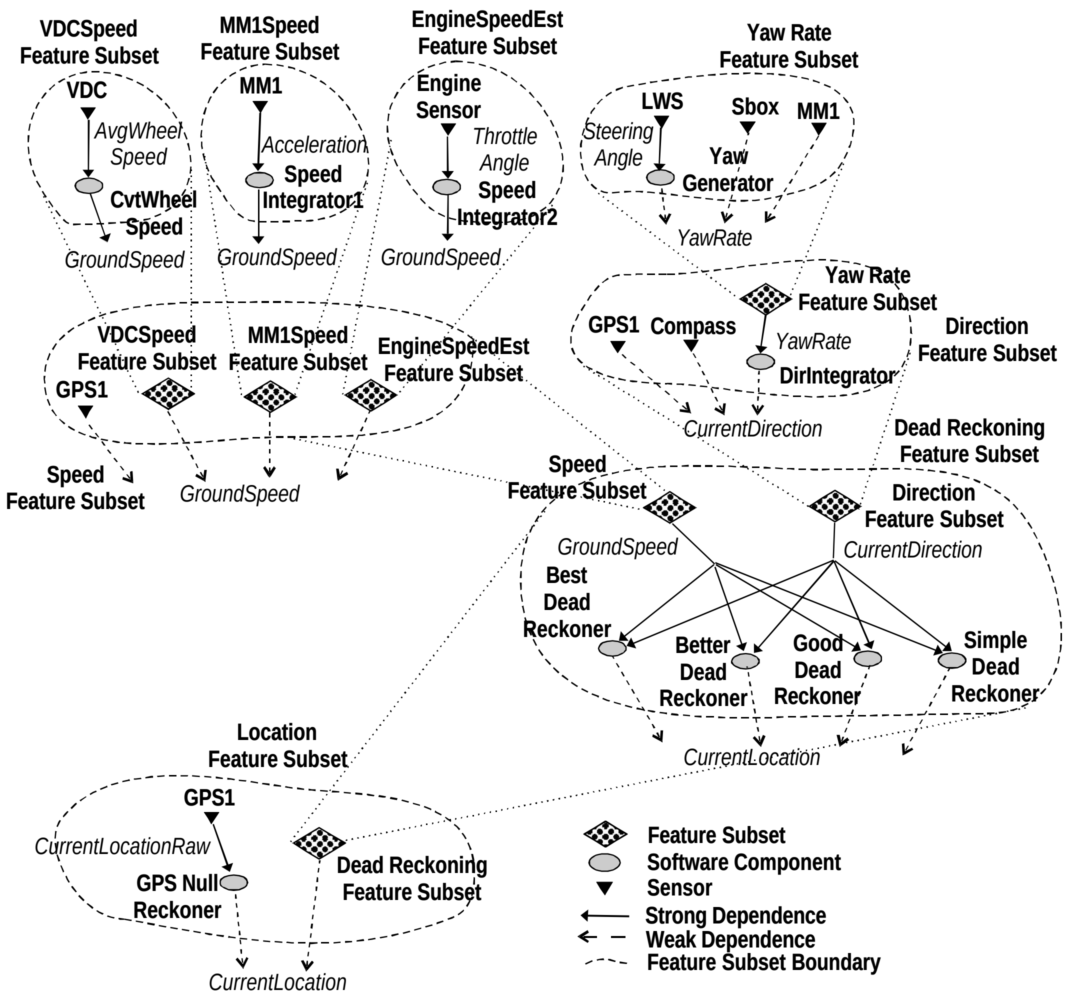

Figure 4.5. Expansion of the Location Feature Subset.

were able to eliminate many invalid feature subset configurations based on component dependencies. The total number of feature subset configurations we had to specify was 106.

If we look at the capability configurations there is one valid feature subset configuration for both the Turn Signal and Speaker capabilities, and $2^8 - 1 = 255$ valid configurations for the Display capability that has eight feature subsets. Although the Display capability has 255 valid configurations, we will not have to specify their utility values individually. Since there is only one Display actuator it can only receive inputs from one of the eight map components at a time to provide utility. Therefore, we only need to rank the relative utility of the 8 Map feature subsets to specify the utility of the Display capability. When multiple feature subsets are available in the system configuration, the Display will only use the one that provides the most utility. The other feature subsets are treated as backups.

Since we need at least one working capability to provide positive system utility, there are $2^3 - 1 = 7$ possible configurations at the system level that must be specified to complete the system utility function. We have a total of 123 configurations that must be specified (106 in feature subsets, 10 in functional capabilities, 7 system capability configurations) to evaluate the relative utility for all $6 * 10^{11}$ valid system configurations. Appendix B contains all of the feature subset definitions along with our utility specification.

4.3 Mechanisms that Contribute to Graceful Degradation

Since this example system was designed to provide a high level of graceful degradation, we can perceive characteristics of the architecture through our system model that seem to particularly enhance the system's ability to gracefully degrade.

Several aspects of this system's software architecture stand out:

- The architecture has well-defined interfaces among components that provide for logical partitioning of the system into subsystems.
- Subsystems that provide required functionality are targeted for an increased level of functional redundancy and brute-force redundancy.
- Heterogeneous redundancy is available to provide multiple alternatives for providing system outputs and completing system requirements.
- Subsystems are designed to be robust to input failures so that they can continue to provide utility when system variable inputs are not available.

We will examine each of these aspects in both the navigation system and the brake-by-wire system discussed in the previous chapter to derive a set of heuristics that should help improve graceful degradation for distributed embedded systems.

4.3.1 Well-Defined System Component Interfaces

In order for a system to provide graceful degradation, the individual subsystems should be decoupled so that they can tolerate failures from other parts of the system. One method of decoupling subsystems is to define a set of system interfaces that restrict the amount of state that is passed among components. In distributed embedded systems, these interfaces should map to a set of system state variables that represent the key data elements that are required across different subsystems. In our system model the interfaces are represented by the set of defined system variables. Since the system components and subsystems are only coupled through system variables, making individual subsystems and components robust to losses of inputs should improve the system's ability to gracefully degrade.

Unfortunately, producing well-designed component interfaces is a fundamental problem of software and system architecture. One of the key insights of software architecture is that the interfaces among components can have as much of an impact on the system as the components themselves. The software architecture view of components and connectors emphasizes specifying the connectors in as much detail as the components.

Within the context of distributed embedded systems, we reduce the general interface problem to only specifying what data will be passed in the system state variables. System designers must have domain knowledge so that they can identify what internal transformations are useful from sensor data values to actuator command values. If we are dealing with a real-time control system, we can use the control system parameters as a starting point for identifying system variables.

The system variables should provide logical partitioning of the system into subsystems, as well as computational "checkpoints" that represent intermediate steps in the system's processing. For example, in the automobile navigation system, two of the major system variables are the *CurrentLocation* and *MapData* variables. Every part of the system that provides functionality depends on receiving these two data variables. The PathPlanner component requires their data values to provide an accurate path to the destination. All of the Map components that output to the

Display actuator must have access to map and location information to give the driver any information about where the car is traveling. However, the PathPlanner and Map components do not require the *CurrentDirection* and *GroundSpeed* variables that were used to calculate the *CurrentLocation* variable in some instances. Thus, in order to ensure that the system gracefully degrades, we can either provide multiple sources for these data elements, or design components that require them as inputs to tolerate their loss.

We can use our system model to recognize whether a system's interface is more or less conducive to providing graceful degradation. Since feature subsets are defined based on component interfaces, in general, the number of feature subsets will scale with the number of defined system variables. If the interface has many system variables, there will be many feature subsets defined in the system relative to the number of components in each feature subset ($p >> k$), and we will have to consider building mechanisms into each one to tolerate input failures. This may be cost-prohibitive if a large fraction of these feature subsets are required to provide any system utility. However, if the interface has few system variables, there will be few feature subsets defined relative to the number of components in each feature subset ($p << k$). This would seem to indicate that feature subsets are large, monolithic, and complex. Then, using brute-force redundancy to completely replicate these feature subsets would seem to be the best way to achieve graceful degradation. Unfortunately this would also be expensive in terms of system resources. There should be a "sweet spot" in which the number of system variables and feature subsets are within an order of magnitude of the number of components per feature subset, where the graceful degradation techniques we propose at the subsystem level should have the maximum effect for the least system design effort and resource cost.

Beyond understanding the system’s problem domain and applying the traditional approach of modular system decomposition, we do not have any new insight on how to design system interfaces that are ideal for graceful degradation. For this research, we have focused on developing techniques that will be scalable given that the system component interfaces are designed so that we have well-partitioned logical subsystems. Since this is already a goal for well-designed system architectures, and we have reduced the scope to defining state variables in distributed embedded systems, this should be a reasonable assumption.

4.3.2 Targeted Redundancy for Critical Subsystems

As a general graceful degradation approach, this mechanism is similar to the simplex architecture [Bodson93]. We build a subsystem with multiple functionally redundant software components that provide tradeoffs between their complexity and the accuracy of their outputs. The simpler components should be less prone to failures, but will not provide as much utility as the more complex components. We could also add brute-force hardware redundancy for subsystems that are identified as single points of failure. It may be more feasible to apply brute-force redundancy such as replicated system resources to only those parts of the system that are identified as mission-critical, rather than replicating all of the system’s components.

The best example for this type of redundancy in the navigation system is the Location feature subset. The location data is critical for the system’s ability to provide its navigation functionality. Therefore, the location subsystem must be especially resilient to component failures. However, implementing redundant backup systems with complete functionality may be cost-prohibitive. For example, the GPS sensor and software may be an expensive set of components to replicate. The dead reckoner subsystem provides functional redundancy to the GPS sensor, with a tradeoff of reduced accuracy of location data. Similarly, the TurnInfo feature subset contains multiple software components that may have different algorithms for providing real-time turn information to be communicated to the driver through the turn signal indicator and speaker.

In the brake-by-wire system, discussed in Chapter 3, the Brake Pedal and Anti-Lock feature subsets represent critical redundant functionality that trades accuracy and performance for dependability. Each brake controller has direct access to the data from the brake pedal controller in the event that the anti-lock braking subsystems for each wheel fail. With this data, the system can still provide low-performance braking functionality. Additionally, the Brake Pedal feature subset is replicated in hardware since it can be a single point of failure in the system and requires high reliability.

4.3.3 Heterogeneous Redundancy

We define heterogeneous redundancy as subsystems that are designed to provide different functionality when the system is operating normally, but can be used as redundant backups at reduced utility when failures occur. Heterogeneous redundancy is similar to analytic redundancy but is broader in scope.

Heterogeneous redundancy covers not only components and subsystems that are considered functionally equivalent, but also subsystems that may satisfy the same requirements with different functionality.

Heterogeneous redundancy can take many forms. There may be several sensors available in the system that monitor different aspects of the environment that are physically related, such that one sensor’s data can be synthesized by applying a transform function to another sensor’s data. For example, if a system has sensors that monitor temperature, pressure, and volume of a gas, a software component can be designed to implement a transform function to synthesize the output of one sensor based on the readings of the other two. Thus, one sensor failure could be tolerated with this transformer component, without having to add redundant sensors. Another example of heterogeneous redundancy would be multiple sensors that have varying degrees of accuracy, such as the GPS speed sensor versus the wheel speed sensor in the navigation system. Both sensors may output essentially the same data, but in different formats that must be translated to a system variable interface. One sensor may provide more utility because its data is more accurate and reliable, but the other sensor can provide a redundant backup.

In the automobile navigation system, heterogeneous functionality is available at the system level. The main requirement of the system is to provide navigation aid to the driver by giving turn-by-turn real-time directions. The system has three distinct actuators that can provide system utility: the turn signal lights on the dash board, the car’s radio speaker, and the visual display installed in the car with the navigation system. In general, we would expect the visual display to provide the most utility and give the driver the most information about the driving route. However, if the display suffers a failure, the system can still provide utility via the turn signal lights and audio cues from the speaker. The failure of any of these three functions is compensated by the availability of navigation information from the other two actuators, even though they are not designed to provide the same or equivalent functionality. This redundancy is a consequence of the functionality built into the system.

In the brake-by-wire system, there is heterogeneous redundancy by virtue of the fact that there is one braking subsystem for each of the four wheels in the car. During normal operation, each of the four brake actuators applies braking force to a separate wheel to stop the car. However, if one of the actuators failed, the car would still be able to brake, perhaps with an increased stopping distance. Even if three of the four brake actuators failed, one working brake actuator would still be useful in helping the driver regain control of the car.

4.3.4 Component Robustness to Loss of Inputs

Designing individual components (and feature subsets) to be robust to input failures complements designing subsystems to provide redundant sources of output system variables. If a component can tolerate the loss of a system variable when all of its input sources have failed, it may still provide reduced utility and prevent a system failure. This may not be possible in all situations, but we can identify some guidelines that might help implement this design approach. Within our system model, this would be represented by transforming arrows in the data flow graph that represent strong dependence on system variable inputs, to arrows that represent weak dependance or treat system variable inputs as optional.

One approach that can make components robust to a loss of input failures is to design a component with multiple algorithms that provide similar functionality with each possible combination of required inputs. Of course this will add a great deal of complexity to the component, as it must manage multiple algorithms as well as transition between algorithms when inputs are lost. Within a feature subset, these algorithms may be separated into multiple components, as with the Map feature subsets defined in the Display capability in the navigation system.

Another approach might be to initially specify the component's output to provide some "base level" utility with a minimum of system variable inputs and a default behavior. Then any other inputs that are available should be treated by the component as "advice" that modifies the default behavior in specific ways. For example, if the brake controllers in the brake-by-wire system lost both the brake pedal and anti-lock braking data from those subsystems, they could provide a default behavior that applies enough pressure to the brakes to hamper the car's acceleration, and then will cause the car to come to a gradual stop when the accelerator is released. This crippled functionality would signal to the driver that the car needs to be taken in for repairs, while providing at least a low level of transportation ability. This technique assumes that received inputs will not be erroneous, which is reasonable because of our fail-fast, fail-silent fault model.

In the navigation system, this technique is not readily visible in the system architecture. The designers originally intended reconfiguration to be the main mechanism of graceful degradation, so that when a component that provides functionality no longer had enough system inputs available, the reconfiguration manager would swap that component for another that required fewer inputs. However, it may not be feasible to require a reconfiguration action every time a component failure occurs, and it might be necessary for some components to be designed to continue operating even when all input data sources are lost. This is a defensive strategy for component design to guard against the event that multiple failures may occur that cause the complete loss of a system variable input.

4.4 Model Analysis and Graceful Degradation Implementation

The techniques described in section 4.3 focus on mechanisms that should increase the number of valid configurations within individual feature subsets. They contribute to graceful degradation because reducing the proportion of feature subset configurations that provide zero utility in general will translate to fewer system configurations that provide zero system utility. However, it is not feasible to simply apply all of these graceful degradation techniques to every feature subset in the system. Each technique has a cost in terms of increased design effort or additional system resources.

We can use our system model to analyze the system architecture to target which components and feature subsets should receive graceful degradation support. There are several properties in the architecture that can be used as indicators for which parts of the system should be improved with graceful degradation. For example, with our scalable system utility function generated from the system model, we can evaluate every configuration in which a single component or feature subset is not available. If any of these configurations are invalid (provide zero system utility) we know that the component or feature subset is a single point of failure. Any component that is a single point of failure in the system should have some graceful degradation mechanism installed to compensate for a failure, such as a redundant backup (Section 4.3.2), or a heterogeneous source of the component's output variables elsewhere in the system (Section 4.3.3). Similarly, a feature subset that is critical to providing system utility should contain multiple components that can provide its outputs to tolerate failures.

Another approach to improving system-wide graceful degradation could be to analyze which system variables are required inputs to a large number of components in the system. This information is available from the system model by counting how many sink output roles each system variable connector has. The more components that require any one system variable as an input, the more critical that system variable is to system utility. Therefore, we should maximize the number of components and feature subsets that can output that system variable. Depending on the resources available, both targeted redundancy (Section 4.3.2) and heterogeneous redundancy (Section 4.3.3) may be appropriate mechanisms to provide multiple components and feature subsets that output a system variable.

Designing components to be robust to input failures (Section 4.3.4) may be more difficult since multiple algorithms may be necessary for each software component to tolerate the loss of any required system variable inputs. Again, depending on the system resources available, it may be desirable to redesign all of the components that receive an input from a critical component so that they can tolerate the

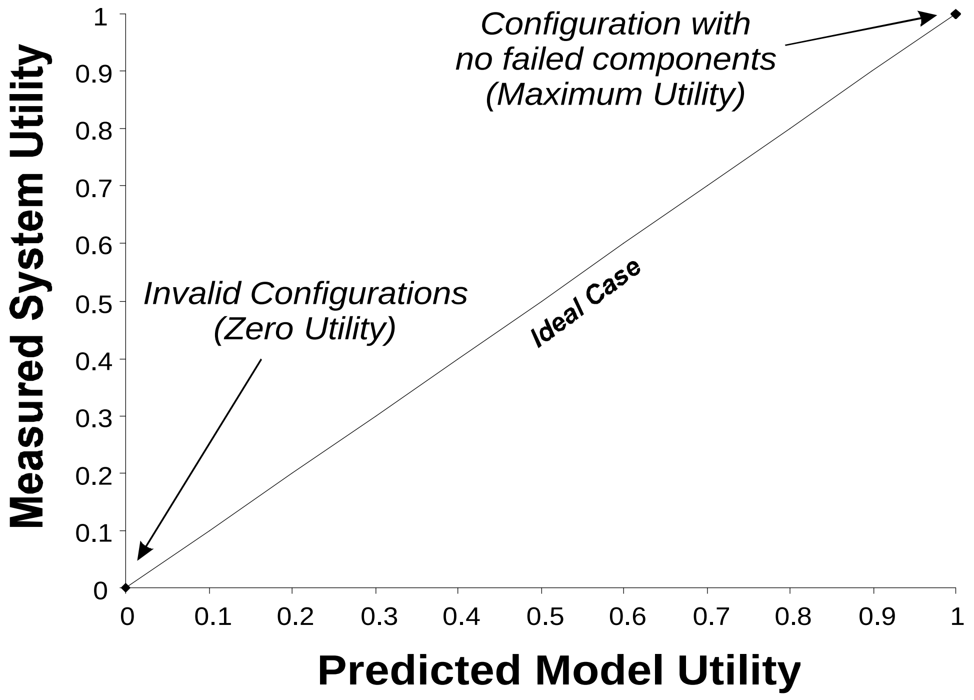

Figure 4.6. Graph of Ideal Case of Predicted Model Utility vs. Measured System Utility.

component's failure, rather than add redundant backup resources to the component. This effectively renders the component no longer a single point of failure.

The model analysis provides information about which feature subsets and components are critical to system utility, allowing us to target these parts of the system for graceful degradation mechanisms. Choosing which techniques to implement requires an analysis of the tradeoffs between the resources available in the system and the level of dependability required. Our scalable specification framework should enable these tradeoffs to be explicitly identified with the utility model and information about the resources required for system components and feature subsets.

In addition to using the model at design time to determine where graceful degradation mechanisms should be applied in the system, the model can also be used to validate whether or not the system implementation achieves the level of graceful degradation predicted. In an ideal case the utility model should perfectly reflect each component and feature subset's contribution to system utility. If we have a utility metric that incorporates all of the desired system properties defined in the system's requirements, and these attributes can be measured in the system implementation, then every system configuration's actual measured utility should equal the utility predicted by the model. If we graph each configuration's utility from the model versus its measured utility for all 2^N configurations, we should have a straight line with a slope of 1 as shown in Figure 4.6.

Unfortunately, in general this ideal case is not possible. Many system properties such as usability, maintainability, and dependability cannot be readily quantified, and it is nontrivial to combine these properties along with system functionality and performance into a single utility metric. Additionally, the utility function generated in the system model is based on the feature subset definitions and the assumption that each feature subset's utility can be evaluated independently for each configuration. Individual feature subset configuration utility functions may be based on the designer's domain knowledge and understanding of the components in the system. This may lead to a less accurate system utility function that does not capture all of the interactions between components in the system.

Rather than focus on absolute utility measurements that may be inaccurate, we can use the relative utility values of system configurations to rank all 2^N configurations in order by increasing utility according to the model. Then we may select a system property or set of properties such as performance and reliability that may be measurable for the system implementation, and use this measurement as a proxy for a system utility metric. If we graph the system configurations by comparing their utility value as predicted by the model and their system property metric that is a substitute for a system utility measurement, we expect a graph that may not be linear, but will be monotonically increasing such that configurations with higher utility values in the model will have higher system property measurements. If there are configurations that do not fit the curve in this graph (e.g. configurations ranked as low utility that have unusually high measured system properties or configurations ranked as high utility that have low measured system properties), they may indicate either an inaccuracy in the system model, a dependability problem in the system implementation, or a violation of the model's assumptions (described in section 1.3). We can apply this analysis iteratively to both refine the system model and identify dependability bottlenecks in the system implementation.

This analysis assumes that the utility values specified by the system model for all 2^N configurations will be accurate enough that in general configurations that actually have more utility will be ranked higher than configurations that actually have less utility. It also assumes that the properties selected to measure the system implementation are indicators of system utility as defined by the system requirements. The system designer should choose properties for this metric that are both quantifiable and general indicators of overall system utility. This may be difficult depending on which properties are considered important by the system requirements, and whether these properties have tradeoffs with one another. The current best practice for combining properties into a single utility metric is multi-attribute utility theory [Keeney76, Keeney 92].

4.5 Summary

In this chapter we have used an example of a complex distributed embedded system architecture to identify a set of properties that should improve graceful degradation when applied to a system's software architecture. Traditional brute-force redundancy techniques are resource intensive and may not be feasible for cost-sensitive embedded systems. Therefore we propose some techniques that apply limited redundancy in terms of both design effort and additional system resources to parts of the system to improve its ability to gracefully degrade.

We can use our system model for specifying graceful degradation to locate single points of failure in the system at the component and subsystem level. Then we can concentrate on adding redundancy to these parts of the system, implementing multiple subsystems that trade complexity for utility. We can add brute force hardware redundancy to smaller feature subsets and individual system components that are mission-critical. We can also identify natural heterogeneous redundancy that may be already designed into the system, and exploit it for graceful degradation. Finally, if we adopt a strategy for individual component and subsystem design such that they are robust to input failures, this will complement the targeted redundancy that emphasizes providing multiple output sources for system variables.

We also outlined a general analysis technique that uses our system model for scalable graceful degradation to identify dependability problems in the system implementation. By measuring relevant system properties of different system configurations in the implementation and comparing these measurements to the utility values predicted by the model, we can validate the accuracy of the model. Any discrepancies between the model prediction and the measurements of the implementation may be traced to either an inaccuracy in the model or a problem in the system design and implementation.

The application of the graceful degradation techniques depends on already having well-defined system variable interfaces that create partitioned feature subsets. We do not think it is unreasonable to require an architecture that provides partitioned decoupled subsystems as a prerequisite for providing scalable system-wide graceful degradation, as this is a fundamental goal of system and software architecture design. In the next chapters we will demonstrate the application of these techniques on two representative distributed embedded systems, and evaluate how they contribute to system-wide graceful degradation. We also use the analysis methods we have outlined to validate the utility model compared to an approximate utility measure of the system implementation for each case study.

5 Case Study: Elevator Control System

To illustrate how we can apply our system model and graceful degradation techniques to a realistic distributed embedded system, we use a design of a relatively complex distributed elevator control system. This system was designed by an elevator engineer (my thesis advisor) and has been implemented in a discrete event simulator written in Java. This elevator system has been used as the course project in the distributed embedded systems class at Carnegie Mellon University for several semesters. Since we have a complete architectural specification as well as an implementation, we can directly observe how properties of the system architecture affect the system's ability to gracefully degrade by performing fault injection experiments in the simulation.

The architectural specification is in the form of a requirements document that specifies each system component's inputs and outputs, as well as their functional behavior. The component interfaces are specified as a message dictionary that represents all network messages that can be sent among components. We first describe the elevator system interface in detail, and then apply the graceful degradation techniques from Chapter 4 to the system architecture. We then apply our system model to the architecture to specify its ability to gracefully degrade. Finally, we run a set of experiments on the elevator system using both the original system architecture and the new architecture with our graceful degradation improvements. We fail several combinations of components and observe the effect on the system's ability to deliver passengers.

5.1 Elevator System Architecture

Our view of the elevator system is a set of sensors, actuators and software components that are allocated to the various hardware nodes in the distributed system. The nodes are connected by a real-time fault-tolerant broadcast network. All network messages can be received by any node in the system. Since all communication among components is via this broadcast network, all component communication interfaces map to a set of network message types.

Our elevator system architecture is highly distributed and decentralized, and is based on the message interfaces that system components use to communicate. System inputs come from "smart" sensors that have a processing node embedded in the sensing device. These sensors convert their raw sensor values to messages that are broadcast on the network. The software control system, implemented as a set of distributed software components, receives these messages and produces output messages that provide commands to the actuators that provide the system's functionality.

The elevator consists of a single car in a hoistway with access to a set number of floors *f*. The car has two independent left and right doors and door motors, a drive that can accelerate the car to two speeds (fast and slow) in the hoistway, an emergency stop brake for safety, and various buttons and lights for determining passenger requests and providing feedback. Since the sensors and actuators map directly to the message interfaces among components, we list all the possible interface message types along with their senders and receivers below to define the components and interfaces of the system architecture. In the following notation, the values within the "[ ]" brackets represent the standard replication of an array of sensors or actuators, and the values within the “( )” parentheses represent the values the sensor or actuator can output. For example, the Hall Call message type maps to an array of sensors for the up and down buttons on each floor outside the elevator that is *f* (the number of floors the elevator services) by *d* (the direction of the button; Up or Down) wide, and each button sensor can either have a value v of True (pressed) or False (not pressed). Unless otherwise noted, “f” represents the number of floors the elevator services, “d” represents a variable that indicates a direction of either Up or Down, “j” is a variable that is a value of either Left or Right (for the left and right elevator doors), and “v” denotes a value that can be either True or False.

The sensor message types available in the system include:

- **AtFloor[f](v):** Output of AtFloor sensors that sense when the car is near a floor.
- **CarCall[f](v):** Output of car call button sensors located in the car.
- **CarLevelPosition(x):** Output of car position sensor that tracks where the car is in the hoistway. x = {distance value from bottom of hoistway in millimeters}
- **CarWeight(w):** Output of car weight sensor that measures the aggregate weight of all passengers in the car. w = { weight in car in pounds }
- **DoorClosed[j](v):** Output of door closed sensors that will be True when the door is fully closed.
- **DoorOpen[j](v):** Output of door open sensors that will be True when the door is fully open.
- **DoorReversal[j](v):** Output of door reversal sensors that will be True when door senses an obstruction in the doorway.

- **HallCall[f,d](v):** Output of hall call button sensors that are located in hallway outside the elevator on each floor. Note that there are a total of 2f - 2 rather than 2f hall call buttons since the top floor only has a down button and the bottom floor only has an up button.
- **HoistwayLimit[d](v):** Output of safety limit sensors in the hoistway that will be True when the car has overrun either the top or bottom hoistway limits.
- **DriveSpeed(s,d):** Output of the main drive speed sensor. s = {speed value}, d = {Up, Down, Stop}

The actuator command messages available in the system are:

- **DesiredFloor(f, d):** Command from the elevator dispatcher algorithm indicating the next floor destination. d = {Up, Down, Stop} (This is not an actuator input, but rather an internal variable in the control system sent from the dispatcher to the drive controller)
- **DesiredDwell(n):** Command from the elevator dispatcher algorithm to the door controllers indicating how long the doors should remain open when stopped on a floor. n = { Integer dwell time in milliseconds } (This is also not an actuator input, but an internal control system variable that allows the dispatcher to affect the operation of the door motors)
- **DoorMotor[j](m):** Door motor commands for each door. m = {Open, Close, Stop}
- **Drive(s, d):** Commands for 2-speed main elevator drive. s = {Fast, Slow, Stop}, d = {Up, Down, Stop}

- **CarLantern[d](v):** Commands to control the car lantern lights; Up/Down lights on the car doorframe used by passengers to determine the elevator's current traveling direction.
- **CarLight[f](v):** Commands to control the car call button lights inside the car call buttons to indicate when a floor has been selected.
- **CarPositionIndicator(f):** Commands for position indicator light in the car that tells users what floor the car is approaching.
- **HallLight[f,d](v):** Commands for hall call button lights inside the hall call buttons to indicate when passengers want the elevator on a certain floor.
- **EmergencyBrake(v):** Emergency stop brake activated whenever the system state becomes unsafe and the elevator must be shut down to prevent a catastrophic failure.

For each actuator, there is a software controller object that produces the commands for that actuator. The drive controller commands the drive actuator to move the elevator based on the DesiredFloor input it receives from the dispatcher software object. The left and right door controllers operate their respective door motors. The safety monitor software monitors the elevator system sensors to ensure safe operation and activate the emergency brake when necessary. The various software objects for the buttons and lights determine when to activate the lights to indicate appropriate feedback to the passengers. The elevator control system consists of 9 + 4*f* sensors, 5 + 3*f* actuators, and 6 + 3*f* software components, for a total of 20 + 10*f* components in the system. Figure 5.1 illustrates how these system components are allocated to hardware nodes in the elevator's distributed control system. Each software component has a set of inputs and outputs that are specified

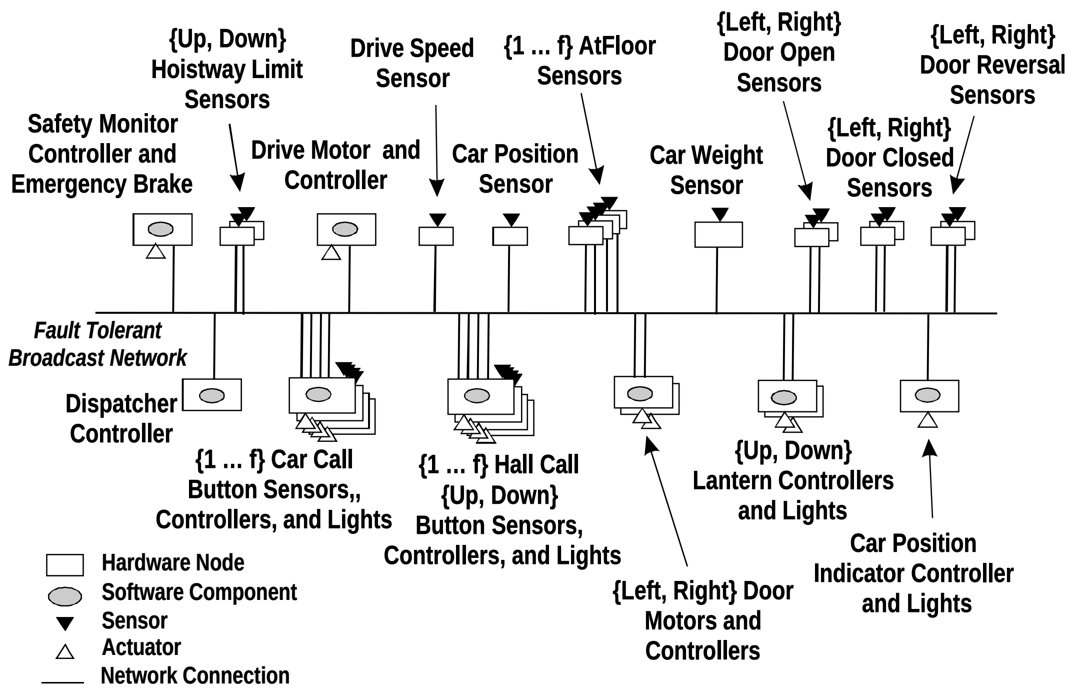
{Up, Down} Hoistway Limit Sensors
Drive Speed Sensor
{1 ... f} AtFloor Sensors
{Left, Right} Door Open Sensors
{Left, Right} Door Reversal Sensors
Safety Monitor Controller and Emergency Brake
Drive Motor and Controller
Car Position Sensor
Car Weight Sensor
{Left, Right} Door Closed Sensors
Fault Tolerant Broadcast Network
Dispatcher Controller
{1 ... f} Car Call Button Sensors,, Controllers, and Lights
{1 ... f} Hall Call {Up, Down} Button Sensors, Controllers, and Lights
{Up, Down} Lantern Controllers and Lights
Car Position Indicator Controller and Lights
{Left, Right} Door Motors and Controllers
Hardware Node
Software Component
Sensor
Actuator
Network Connection

Figure 5.1. Hardware View of Elevator Control System.

from the message interface. Appendix C describes the interface specification for each of these software components. All components are designed to require all of their inputs to provide their functionality.

This elevator system is decomposed into several distinct subsystems that have control over different parts of the elevator system. Although the elevator has well-defined component and interface definitions, this system was not particularly designed for graceful degradation. Many of the subsystems are tightly coupled because they depend on each others' outputs. For example, the drive controller will not move the elevator unless it receives commands from the dispatcher, and the dispatcher cannot effectively service all floors without receiving inputs from all of the elevator hall call and car call buttons. In the next section we will apply some of the mechanisms we identified in Chapter 4 to this elevator architecture to give the system the ability to gracefully degrade.

5.2 Adding Graceful Degradation to the Elevator System

In order to apply the graceful degradation techniques we identified, we first examined which parts of the system should be considered mission-critical, and which could be considered functional enhancements that can be lost without causing a system failure. An elevator system's most basic requirements are that it protect passenger safety while using the system and transport passengers to their destination floors without stranding them or trapping them in the elevator. Other services typically associated with an elevator system, such as providing appropriate passenger feedback and efficiently processing passenger requests, can be considered "optional" functionality. As long as the elevator maintains passenger safety, and can (eventually) service all floors, the elevator can still be considered "working."

Based on the software components defined for this elevator system, the safety, drive control, and door control subsystems are responsible for satisfying the basic elevator requirements. Therefore, we can significantly improve the system's ability to gracefully degrade if we can ensure that these subsystems can tolerate failures from all other parts of the system. We can achieve this by redesigning system components to tolerate multiple input failures, and by adding redundant components to critical subsystems.

The safety monitor component requires all of its sensor inputs to keep track of the elevator's state and ensure that the elevator does not violate its safety conditions. If a violation is detected, the safety monitor will trigger the emergency brake and will cause a complete shutdown of the elevator system. Therefore we are unlikely to find any graceful degradation opportunities in the safety subsystem, as all of the components are required to prevent a catastrophic system failure that may harm the passengers. A loss of any of the sensor inputs by the safety monitor will by design trigger an emergency shut down.

There are several modifications we can make to the drive controller that will make it robust to input failures, which is one of the graceful degradation strategies we propose in Section 4.3.4. The drive controller depends on the dispatcher to provide commands that tell the drive where to move the elevator in the hoistway, but we can redesign the drive controller component to tolerate the loss of the dispatcher input. If we design the drive to have a default behavior such that it will periodically visit every floor when the input from the dispatcher is not available, this will guarantee that the drive controller can continue to provide elevator service if the dispatcher fails. When the dispatcher is working and providing inputs to the drive controller, the drive controller lets the dispatcher command override its default behavior.

The drive controller also uses the floor, drive speed, and car position sensors to determine what commands to send to the drive motor to travel in the hoistway. At the drive motor's slow speed, the elevator only needs floor sensor data to reliably stop level with a floor. In order to travel faster in the hoistway, the drive controller uses the car position sensor to calculate the appropriate stopping distance to determine when to decelerate from fast to slow before approaching a destination floor. We can ensure that the drive controller will tolerate car position sensor failures by designing it to only command the drive motor to fast if the car position sensor's input is available, and to immediately command the drive motor only to slow if the car position sensor data is lost. This will significantly impact elevator performance, but will continue to guarantee basic elevator requirements.

We can also redesign the dispatcher component to be robust to hall call and car call button failures. The dispatcher implements an algorithm to efficiently process passenger requests by listening to these button inputs. If a button for a particular floor fails, we do not have to change the algorithm, but can simply synthesize an internal request for that floor periodically that will be incorporated into the dispatcher's regular algorithm. This will guarantee that the dispatcher does not "starve" floors on which all of the buttons have failed. This is an extension of our strategy for heterogeneous redundancy and making components robust to input failures, described in Sections 4.3.3 and 4.3.4. The dispatcher takes over the responsibility of generating hall call and car call sensor data internally when some of these sensors fail. This provides a redundant backup for these sensors in the context that their floors continue to be periodically processed into the dispatcher algorithm, even thought their buttons are disabled. This also has the benefit of making the dispatcher software component robust to input failures.

Additionally, since the AtFloor sensors are a critical resource for the elevator system (nearly every subsystem requires Atfloor sensor data), we add redundant "virtual" AtFloor software components that can synthesize AtFloor messages based on data from the car position and elevator drive speed sensors. If some of the physical AtFloor sensors fail, these software sensors can be used as backups. These virtual AtFloor sensors implement data transformations to provide backup Atfloor sensor data and will only transmit their messages when the real Atfloor sensors fail. This draws on the techniques for targeted and heterogeneous redundancy, described in Sections 4.3.2 and 4.3.3. The AtFloor sensors are a critical system resource, and thus a target for additional redundancy to improve system dependability. We can provide this redundancy by synthesizing AtFloor sensor data from other sensors already available in the system, rather than having to add additional physical sensors.

All of the changes made to the implementation to add these graceful degradation mechanism did not require a large amount of code. Ignoring the simulation code that was not altered for either system, the objects that implement the elevator control system in the original design had a total of 1,659 lines of code. The gracefully degrading system had a total of 1,807 lines of code in the control system implementation. This is only a 9% increase in code size to implement all of our modifications for graceful degradation.

Our goal was to identify what changes we could make to the software system to give the elevator system the ability to gracefully degrade. Thus, although we identified the critical parts of the system that could benefit from receiving brute force hardware redundancy, we did not add it to our system. Rather, we evaluated the system's ability to gracefully degrade by restricting our component fault injection only to those parts of the system that were enhanced with graceful degradation mechanisms. The mission-critical sensors, actuators, and software components (components that make up the safety, drive control, and door control subsystems) only represent 25% of the total components in the system. We know that if we fail parts of the system that provide mission-critical functionality, the system will fail, but we are interested in ensuring that any failures in the other 75%

of the components in the system will not cause a complete system failure, but rather will allow the system to gracefully degrade.

5.3 Specifying the Utility Function of the Elevator Control System

We can use the component and interface specifications of the elevator control system to apply our system model for graceful degradation. We will not reproduce the entire system data flow graph here, but rather show the subgraphs for each feature subset we identified and how we performed our analysis using these subsystem definitions. These feature subsets will be defined based on the system architecture after we have made the graceful degradation improvements. Many of these improvements will manifest in the model as data flow arcs between components that are optional rather than strongly dependent relationships. The feature subset definitions of the original elevator system architecture would each have only one valid configuration since each subsystem is dependent on all of its inputs to provide utility.

There is a significant amount of functional repetition in the elevator system. There are two door controllers and two sets of door sensors and actuators, but their only functional difference is that one controls the left door and one controls the right door. Similarly, all of the car call and hall call button software components are implemented as objects that are instantiated at run time from one car call class and one hall call class. However, in the software view of our model, these objects are all considered distinct components that make up distinct feature subsets. Most of these sets of components have one instantiation per floor. They cannot be considered logically equivalent because each component in the set only provides functionality for a single floor.

These replicated components form nearly identical feature subsets that have identical sets of valid configurations, and are replicated on a per floor basis. Thus we can take advantage of this replication to reduce the number of utility specifications. One utility specification can be applied to the entire set of replicated feature subsets. This replication does not represent redundancy for fault tolerance or graceful degradation, but rather is a consequence of distributing functionality for multiple identical system behaviors. The elevator treats every floor the same, so one base class can be designed that generates distinct software objects per floor for similar functionality.

At the system level, there are seven major functional capabilities that provide utility: the safety monitor, drive control, door control, hall call buttons and feedback lights, the car call buttons and feedback lights, the car lantern lights, and the car position indicator lights. These capabilities can each be represented by a single feature subset that encapsulates other subsystems in the elevator architecture. These feature subsets are outlined in the following section.

5.3.1 Elevator Feature Subsets

In the elevator system, there are several functional subsystems that map to feature subsets. The primary control systems in the elevator operate the drive and the door motors. Their feature subsets are defined by the inputs and outputs of the drive controller, and left and right door controller software objects. Figure 5.2 displays

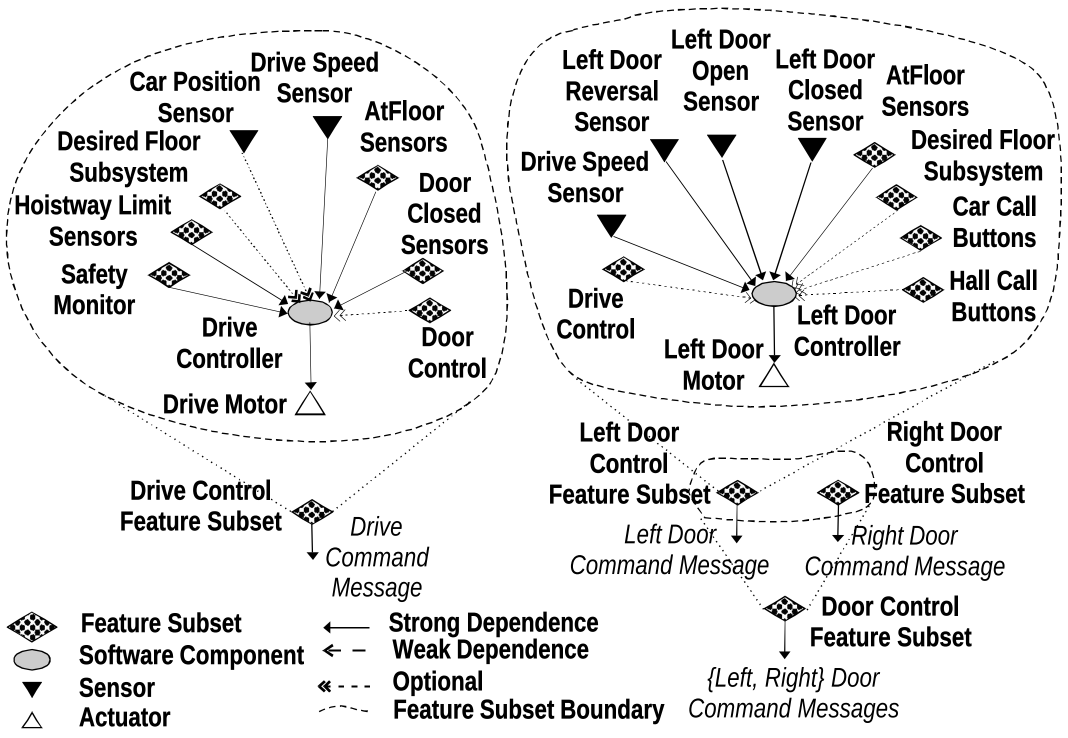

Figure 5.2. Feature Subset Graphs of the Door and Drive Control Subsystems.

these feature subsets and the dependency relationships among their components. In the diagrams we annotated the output variables of each feature subset. The left and right door control feature subsets are nearly identical with the exception of which door sensors and actuators they contain, so only the left door control feature subset is shown in detail. Both door control feature subsets can be covered with one utility specification that is applied separately based on their configurations.

These feature subsets are responsible for controlling the drive and door actuators, but they also output their command variables over the network to the rest of the system. This allows subsystems to loosely coordinate their operation without being strongly coupled and dependent on each other. For example, the Door Controllers must receive inputs from the Drive Speed sensor in order to safely operate the door only when the elevator is not moving. However, the Door Controller can also use the command output from the Drive Control feature subset to anticipate when the elevator will stop based on the command sent from the Drive Control feature subset, thus allowing more efficient door operation via sending the door open command slightly before the elevator is level with the destination floor. The Drive Control feature subset encapsulates all of its components, so that it is represented as a single component that outputs the Drive command system variable in the Left and Right Door Control feature subsets. Likewise the Door Control feature subset encapsulates all of the components in the Left and Right Door Control feature subsets.

These feature subsets also contain several identical components, such as the Drive Speed and AtFloor sensors. These components do not represent multiple copies of the same component in the software data flow view, but rather that these feature subsets overlap and share some of their components. The feature subset graphs show dependencies among components, but not whether individual components are replicated for multiple subsystems. There may be multiple redundant sensors installed in the system, but the information about how components are allocated to hardware would be visible in the hardware architecture and is orthogonal to the software data flow view of our system model.

We defined several other feature subsets for our elevator system in addition to the Door Control and Drive Control feature subsets. The safety monitor software component and its inputs and outputs defines the Safety Monitor feature subset. The Safety Monitor feature subset is responsible for detecting when the elevator system state becomes unsafe, such as the doors opening while the elevator is moving, the doors failing to reverse direction if they bump into a passenger while closing, or the

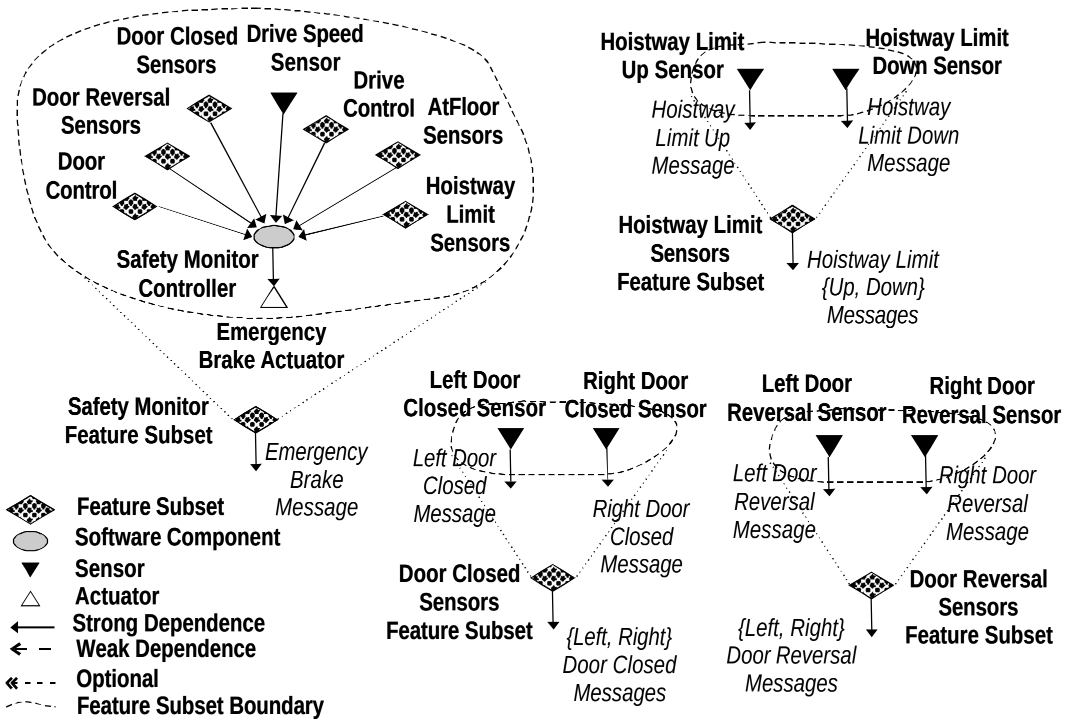
Door Closed Sensors
Drive Speed Sensor
Drive Control
Door Reversal Sensors
AtFloor Sensors
Door Control
Hoistway Limit Sensors
Safety Monitor Controller
Emergency Brake Actuator
Safety Monitor Feature Subset
Emergency Brake Message
Feature Subset
Software Component
Sensor
Actuator
Strong Dependence
Weak Dependence
Optional
Feature Subset Boundary
Hoistway Limit Up Sensor
Hoistway Limit Down Sensor
Hoistway Limit Up Message
Hoistway Limit Down Message
Hoistway Limit Sensors Feature Subset
Hoistway Limit {Up, Down} Messages
Left Door Closed Sensor
Right Door Closed Sensor
Left Door Closed Message
Right Door Closed Message
Door Closed Sensors Feature Subset
{Left, Right} Door Closed Messages
Left Door Reversal Sensor
Right Door Reversal Sensor
Left Door Reversal Message
Right Door Reversal Message
Door Reversal Sensors Feature Subset
{Left, Right} Door Reversal Messages

Figure 5.3. Feature Subset Graphs of the Safety Monitor Subsystem.

elevator overrunning into the top or bottom of the hoistway. In any unsafe situation, the safety monitor must trigger the emergency brake actuator that shuts down the elevator system to prevent a catastrophic failure. Figure 5.3 shows the Safety Monitor feature subset along with some of the sensors from which it receives inputs. The safety monitor must receive inputs from both the Door and Drive Control feature subsets to ensure that their commands are consistent with the elevator's actual operation determined from the drive speed and door sensors.

The Door Control, Drive Control, and Safety Monitor feature subsets represent the critical elevator subsystems that provide an elevator's basic functionality. An efficient elevator should also respond to passenger requests to move people quickly to their destination floors. The Drive Controller listens to the DesiredFloor system variable to determine its next destination, and this variable is the output of the Desired Floor feature subset. The Desired Floor feature subset contains the

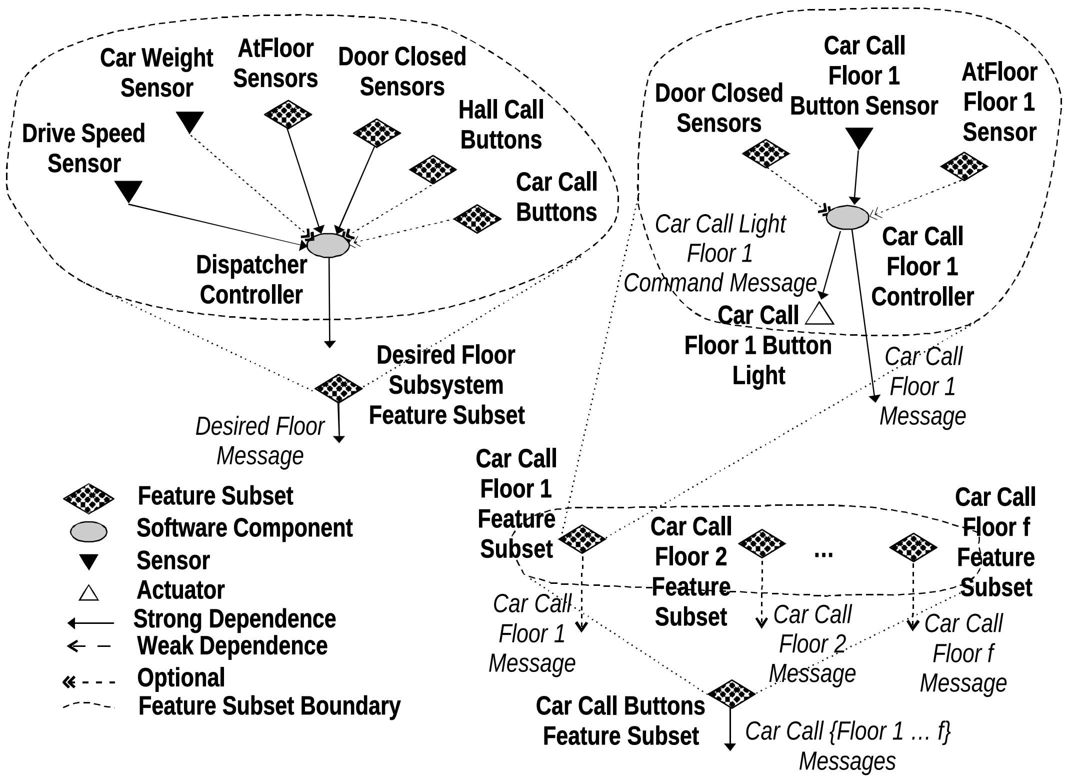

Figure 5.4. Feature Subset Graphs for the Desired Floor and Car Call Button Subsystems.

dispatcher software component that implements the algorithm for determining the next floor at which the elevator should stop. The dispatcher also outputs the DesiredDwell system variable to the door controllers to control how long they hold the doors open on a floor.

The dispatcher receives inputs from the car call and hall call buttons to determine passenger intent and compute the elevator’s next destination. The dispatcher can also use the elevator’s weight sensor to determine when the car is full, thus bypassing hall call requests in favor of car call requests to unload the car. The car call and hall call buttons in turn form their own feature subsets that provide the button sensor messages to the rest of the system, but also control the button lights to provide appropriate passenger feedback. Figures 5.4 and 5.5 show the feature

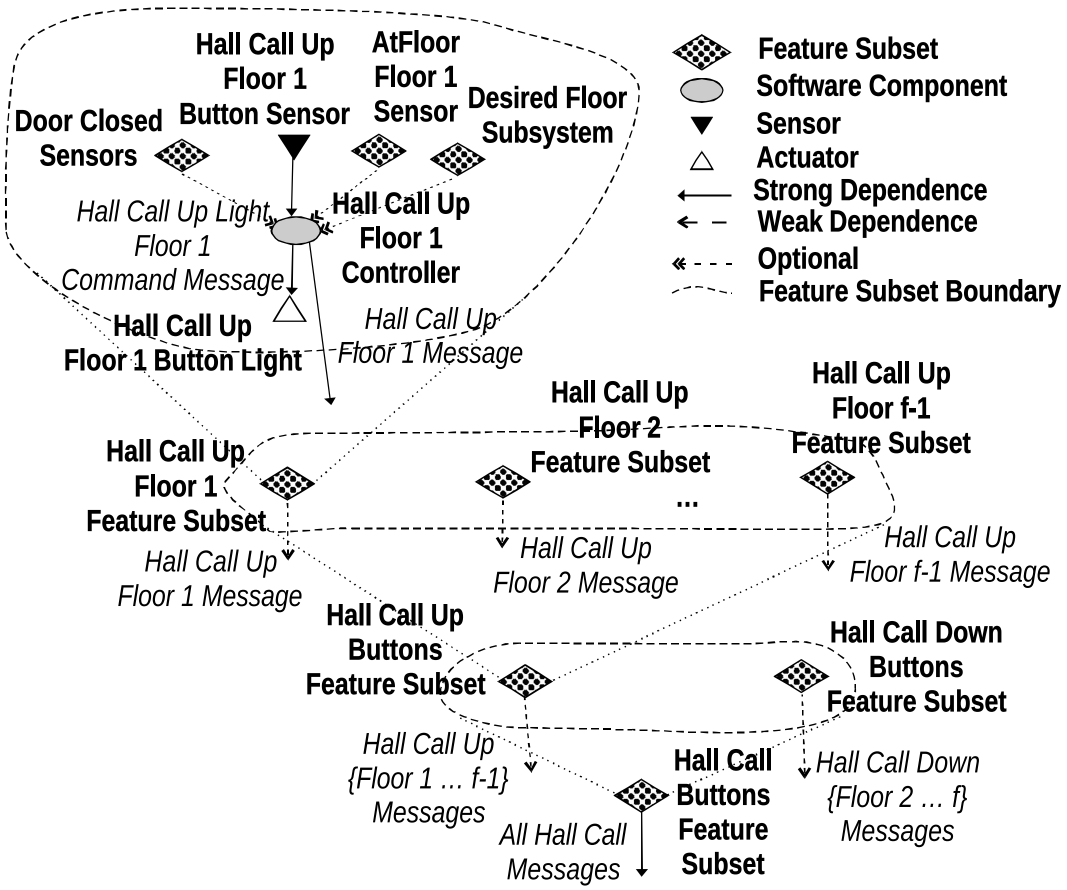
Hall Call Up Floor 1 Button Sensor
AtFloor Floor 1 Sensor
Desired Floor Subsystem
Door Closed Sensors
Hall Call Up Light, Floor 1 Command Message
Hall Call Up Floor 1 Controller
Hall Call Up Floor 1 Button Light
Hall Call Up Floor 1 Message
Feature Subset
Software Component
Sensor
Actuator
Strong Dependence
Weak Dependence
Optional
Feature Subset Boundary
Hall Call Up Floor 2 Feature Subset
Hall Call Up Floor f-1 Feature Subset
Hall Call Up Floor 1 Feature Subset
...
Hall Call Up Floor 1 Message
Hall Call Up Floor 2 Message
Hall Call Up Floor f-1 Message
Hall Call Up Buttons Feature Subset
Hall Call Down Buttons Feature Subset
Hall Call Up {Floor 1 … f-1} Messages
Hall Call Buttons Feature Subset
Hall Call Down {Floor 2 … f} Messages
All Hall Call Messages

Figure 5.5. Elevator Hall Call Button Feature Subsets.

subset definitions for the Desired Floor, Car Call and Hall Call feature subsets. The feature subsets for the car call and hall call buttons are similarly defined for each floor since each car call and hall call software controller have similar input and output interfaces and are replicated instances of the same base class. Each car call and hall call controller outputs the value of its respective sensor on the network for the rest of the system, but only sends the command messages for its button light to its actuator.

In order to encourage people to move quickly in the elevator, the car lantern and car position indicator lights provide feedback to let the passengers know the elevator's current traveling direction, and the elevator's next floor destination.

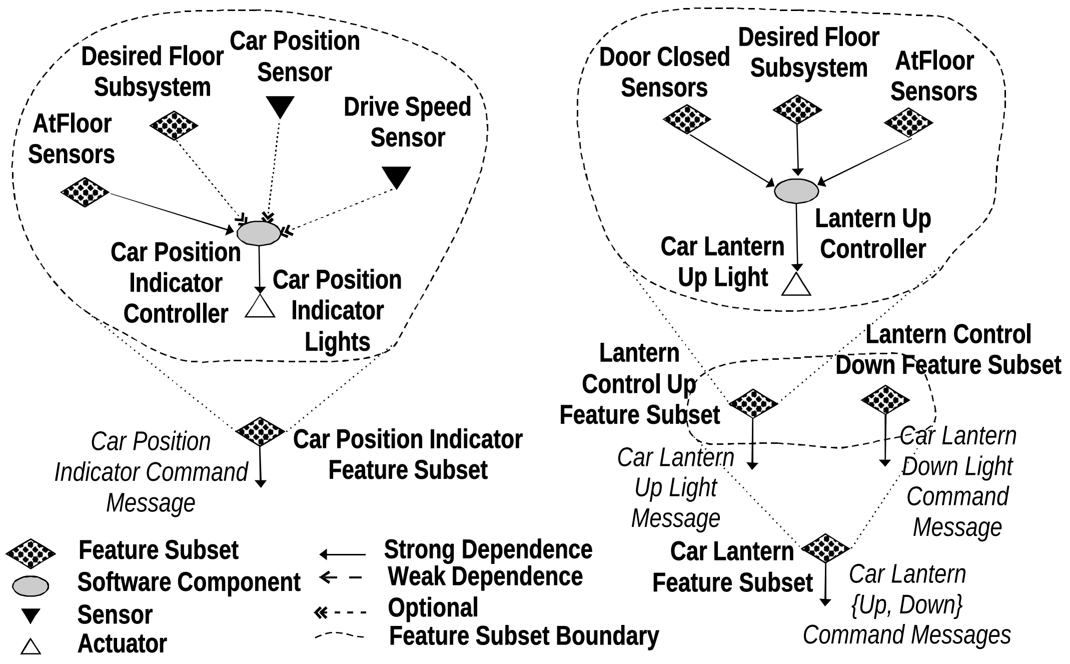
Desired Floor Subsystem
Car Position Sensor
Drive Speed Sensor
AtFloor Sensors
Car Position Indicator Controller
Car Position Indicator Lights
Car Position Indicator Command Message
Car Position Indicator Feature Subset
Door Closed Sensors
Desired Floor Subsystem
AtFloor Sensors
Lantern Up Controller
Car Lantern Up Light
Lantern Control Up Feature Subset
Lantern Control Down Feature Subset
Car Lantern Up Light Message
Car Lantern Down Light Command Message
Car Lantern Feature Subset
Car Lantern {Up, Down} Command Messages
Feature Subset
Software Component
Sensor
Actuator
Strong Dependence
Weak Dependence
Optional
Feature Subset Boundary

Figure 5.6. Car Position Indicator and Car Lantern Feature Subsets.

Figure 5.6 displays the feature subsets for the Car Position Indicator and Up and Down Car Lantern light subsystems. These features are not essential for the elevator's basic operation, but provide information to the passengers to help them use the elevator more efficiently.

One essential subsystem that is required by all of the other major elevator subsystems is the AtFloor Sensors feature subset. Nearly every feature subset strongly depends on AtFloor sensor information to provide functionality. For example, the Drive Control and Door Control feature subsets need the AtFloor sensor information to correctly operate the drive and door motors. Since this is such a critical feature in the elevator system, our elevator design also has redundant software components. The virtual AtFloor software components can synthesize AtFloor sensor messages from the car position and drive speed sensors when the physical AtFloor sensors fail. Thus they are included in the AtFloor sensor feature

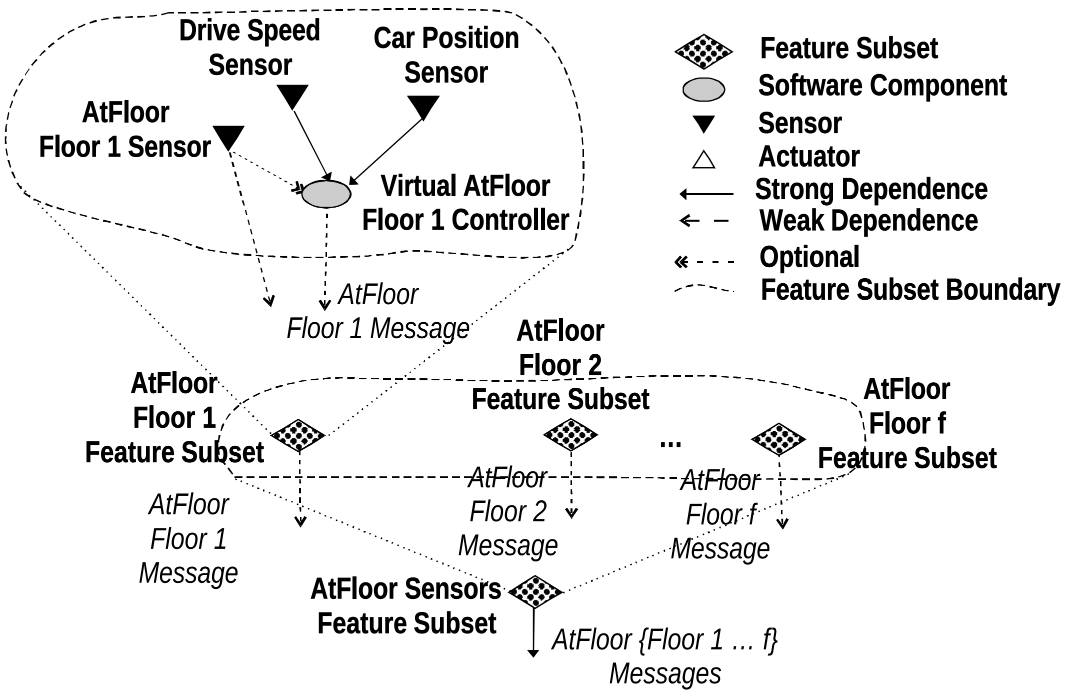

Figure 5.7. AtFloor Subsystem Feature Subset Graph.

subset graphs. Figure 5.7 shows the AtFloor feature subset description for the elevator system in our model. As with other replicated components, the virtual AtFloor sensor components are software objects instantiated from the same base class.

5.3.2 Utility Analysis

The gracefully degrading elevator system has a total of 20 + 11*f* system components (there are an *f* extra components, representing each of the virtual Atfloor software components that we added to the system), meaning there are $2^{20 + 11f}$ possible system configurations. The system can provide basic functionality if the minimum components necessary to operate the drive motor, door motors, and maintain safety are present. Thus these 17 components (drive controller software, drive speed sensor, drive motor, left and right door controller components, left and right door motors, all door sensors, safety monitor software, hoistway limit sensors, emergency brake actuator), in addition to the components required to provide valid AtFloor feature subsets for each floor, are fixed and must be present in every valid configuration. All other components (such as the button lights and sensors and passenger feedback lights) can be considered optional and present in any configuration. There are $2 + 9f$ optional components that can have $2^{2 + 9f}$ possible configurations.

Enough components to provide working AtFloor feature subsets for each floor must be present as well. Therefore, on each floor there must be a working AtFloor sensor or a working virtual AtFloor component with a working car position sensor. If the car position sensor breaks, then all AtFloor sensors must work, assuming the worst case scenario that the elevator must service at least one passenger on every floor. Since all the AtFloor sensors must work in this situation, they are fixed and have one configuration. However, the virtual AtFloor components can either work or not work since their failure will not affect the availability of the AtFloor system variables, making 2^f valid combinations for the various virtual AtFloor components. If the car position sensor works, then one or both AtFloor sensor and virtual AtFloor component must work for each floor, so the only invalid combinations are when both have failed for at least one floor. This means there are 3 valid combinations per floor, making 3^f valid combinations out of the possible 2^{2f}. Thus there are $2^f + 3^f$ valid combinations of components in the AtFloor feature subset.

The total number of possible valid system component configurations after eliminating all configurations that will always have zero utility is $(2^f + 3^f)(2^{2 + 9f})$.

We ran our elevator simulator tests with seven floors, so this is approximately 9 * 10^{22} configurations that still must be manually ranked. This is a significant reduction from the $2^{97} \approx 2 * 10^{29}$ total possible system configurations, but still intractable for specifying system-wide graceful degradation. However, we can exploit the structure of the system design captured in the feature subset definitions to reduce the number of configurations we must rank to completely specify the system utility function.

We have defined 16 + 4*f* distinct feature subsets in the elevator system. If *f* is small, the largest feature subsets are the left and right door control feature subsets, with 11 components each. Thus we must rank a maximum of 2^{11} = 2048 configurations in any one feature subset.

Since we can determine the valid and invalid configurations in each feature subset by examining the component dependencies, we can significantly reduce the number of configurations we must consider in each feature subset. For example, in the left and right door control feature subsets, 7 of the 11 components are required for the feature subset to provide utility, meaning we only need to consider the 16 possible configurations of the 4 optional components. If *f* is large, the number of configurations in feature subsets that contain *f* components (AtFloor, Car Call, and Hall Call Up/Down) will dominate. However, these feature subsets contain components that are largely orthogonal since each component's functionality is restricted to a different floor. Therefore we can simplify the utility specification of these feature subsets to a linear combination of the utility values of their components, requiring only that we specify *f* weights for each component utility in the feature subset. Table 5.1 summarizes the number of valid configurations that

Table 5.1. Valid Configurations in Each Feature Subset.

Feature Subset	# Replicated Feature Subsets	# Valid Configurations per Feature Subset	Total Valid Configurations
Drive Control	1	8	8
Left/Right Door Control	2	16	32
Top Door Control	1	3	3
Door Closed Sensors	1	1	1
Door Reversal Sensors	1	1	1
Hoistway Limit Sensors	1	1	1
Safety Monitor	1	1	1
Desired Floor	1	8	8
AtFloor per floor	*f*	9	9*f*
Top AtFloor	1	*f*	*f*
Car Call per floor	*f*	8	8*f*
Top Car Call	1	*f*	*f*
Hall Call per floor	2*f* - 2	16	32*f* - 32
Top Hall Call Up/Down Buttons	2	*f - 1*	2*f* - 2
Top Hall Call Buttons	1	3	3
Lantern Control Up/Down	2	1	2
Top Car Lantern	1	3	3
Car Position Indicator	1	8	8
Totals:	**16 + 4*f***		**37 + 53*f***

must be assigned utility values in each feature subset for a total of 37 + 53*f* feature subset configurations that must be considered across the entire elevator system. For our seven-floor elevator, this totals 408 valid feature subset component configurations for the entire system.

We can then determine overall system utility by composing the system configurations of the capability feature subsets that provide system functionality. In the elevator system, these feature subsets are the Drive Control, Door Control, Safety Monitor, Car Call, Hall Call, Car Lantern, and Car Position Indicator feature subsets. All other feature subsets are encapsulated within these seven subsystems that provide external system functionality. Since the Drive Control, Door Control, and Safety Monitor feature subsets must be present to provide minimum elevator functionality, that leaves only $2^4 = 16$ possible configurations of the other four capability feature subsets in the system. Once we specify the relative utilities of these 16 configurations in addition to the 408 total feature subset configurations, we can completely specify the system utility function. We have greatly reduced the number of configurations we must evaluate from $9 * 10^{22}$ system component configurations to 424 feature subset configurations to assess system's ability to gracefully degrade. Appendix D has the complete utility function specification for the elevator control system.

5.4 Experimental Validation

We tested two hypotheses with these simulation experiments. The first is whether the changes we made to the elevator system architecture actually improved the system's ability to gracefully degrade. We can measure this by running simulations of both the original elevator system and our gracefully degrading elevator system, and observing which system can more efficiently deliver passengers. The second hypothesis is whether our system model accurately predicts the relative utility of system configurations, so that we can use it as a measure of graceful degradation for different configurations of a single system.

We performed a set of fault injection experiments on a simulated elevator implementation of both system architectures. A discrete event simulator simulates a real time network with message delay that delivers broadcast periodic messages between system components. Each software component, sensor, and actuator is a software object that implements its message input and output interface to provide functionality. Sensor and actuator objects interact with the passenger objects that represent people using the elevator. Each simulation experiment specifies a passenger profile that indicates how many passengers attempt to use the system, when they first arrive to use the elevator, what floor they start at, and their intended destination. We can specify which elevator system configuration to simulate by setting which components are failed at the start of the simulation.

5.4.1 Experimental Setup

We selected a subset of the possible valid elevator system configurations that represented a wide range of possible component failures. The configurations we selected for evaluation included the configuration in which only the minimum required components for basic operation were present, as well as the configuration in which all of the components were working. We tested several configurations in which different subsets of car call and hall call buttons were failed so that the elevator could not receive all passenger requests. We also picked configurations in which the dispatcher component was failed so that no destination commands were sent to the drive controller. There were a total of 70 configurations tested for both the original and gracefully degrading elevator architectures.

We also generated a set of passenger arrival profiles with which to test each of the system configurations. Each profile had 50 passengers, all arriving within 5 minutes of each other on different floors to use the elevator. Elevator systems usually deal with three types of traffic: two-way, down-peak, and up-peak [Strakosch98]. Two-way traffic assumes random passenger requests between floors. Down-peak traffic is characterized by 90% of the requests from passengers coming from a random start floor and traveling to the first floor. up-peak traffic is characterized by 90% of the requests from passengers coming from the first floor and traveling to a random destination floor. The other 10% of passenger requests in both up-peak and down-peak traffic profiles are random two-way requests. Our experiments included 10 randomly generated passenger profiles for each type of traffic for a total of 30 passenger tests. The total number of simulations we ran were 2 elevator architectures x 70 configurations per elevator x 30 passenger profiles per configuration = 4200.

Although this is a small number of configurations compared to the total number of possible valid system configurations, we can extrapolate these results to the space of system configurations because it is largely constructed of components that are replicated per floor. The dispatcher and car call and hall call button subsystems are mainly responsible for the elevator's performance and functionality. These components are strongly decoupled and provide equal utility contributions to the system per floor. If we test enough failure combinations that account for the failures of each button individually, as well as the dispatcher component, we should have enough data to determine how well the system gracefully degrades.

The simulated passengers in this system are predictable in that they will wait indefinitely for the elevator to service their requests. They will not take the stairs if the wait is too long, and they will never exit the elevator if it does not stop at their destination floor. Since we are only concerned with testing how the elevator can compensate for its own failures, we did not implement passenger behavior that would bypass elevator failures. Work on examining how users can improve system dependability can be found in [Latronico2001].

5.4.2 Original versus Gracefully Degrading Elevators

We can compare the original and gracefully degrading elevator systems by measuring how many passengers each system delivered during the simulation runs. The average number of passengers delivered per configuration was calculated by taking the mean of the number of passengers delivered for all 30 passenger profiles for each configuration and dividing by 50 (number of passengers per simulation test) to get the average percentage of passengers delivered per simulation. Every configuration of the gracefully degrading elevator delivered 100% of its passengers for each simulation test. The original elevator system frequently stranded passengers both in the car and on each floor waiting to be serviced when any of the car call and hall call buttons were broken.

Figure 5.8 shows a bar graph of the average percentage of passengers delivered per simulation for each configuration of the original elevator system. Only three configurations successfully delivered all passengers in every simulation run. These configurations corresponded to situations in which only the passenger feedback lights were failed (car position indicator in configuration ID #4, and the car lanterns and car position indicator in configuration ID #5), and the configuration in which no components were failed (configuration ID #69). Only one test out of all of the simulations run for the other configurations managed to deliver all 50 of its

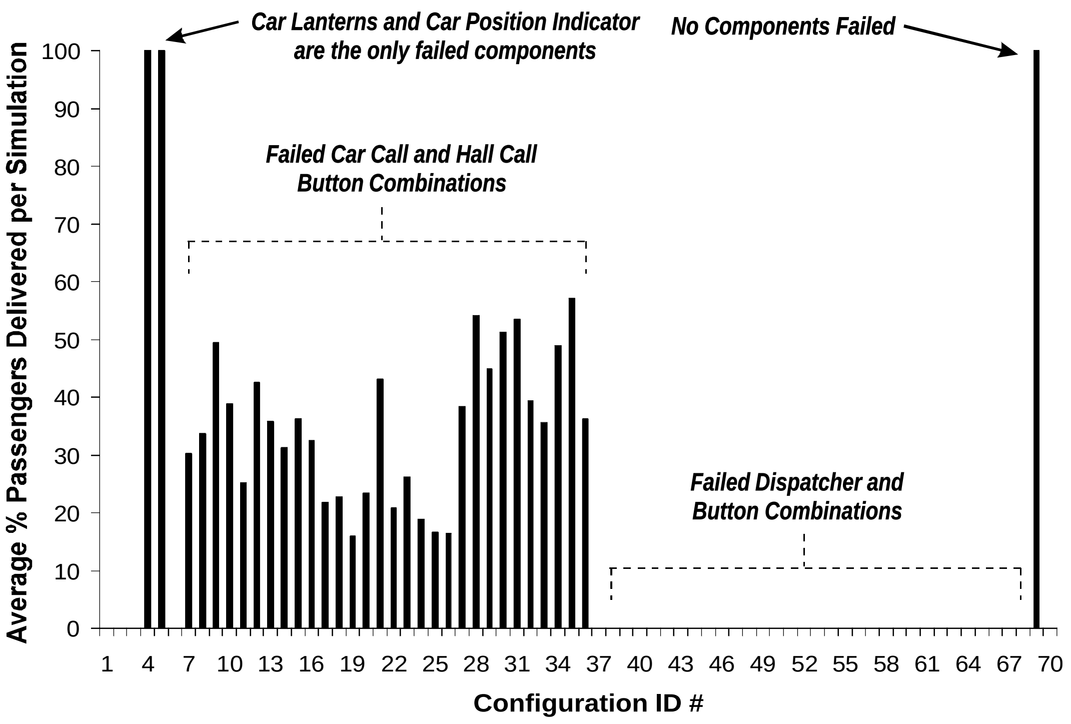

Figure 5.8. Average % Passengers Delivered for the Original Elevator System.

passengers (one of the two-way test profiles for configuration ID #30). Many of the configurations could not deliver any passengers at all because the drive controller could not move the elevator due to the fact that the dispatcher was one of the failed components.

For one of the configurations, the original system violated a safety condition and triggered the emergency brake in all of its simulation runs. This configuration corresponds to when several of the AtFloor sensors were failed (configuration ID #70). This is not surprising given that these are critical sensors required by the drive controller to safely operate the elevator, and the original system has no safeguards against their failure beyond the safety monitor.

5.4.3 Validation of our Utility Model

The result in the previous section shows that our gracefully degrading system can tolerate combinations of component failures that would prevent the original system from satisfying its requirements. It is certainly more fault-tolerant than the original system, and displays some level of graceful degradation. However, we would like to evaluate how well our system model accurately predicts the relative change in system utility due to component failures. We can analyze the relative performance of each of the configurations of the gracefully degrading system to observe whether the system exhibits a gradual drop in utility as components fail. We should also be able to examine discrepancies between the model utility predictions and the utility measured in the system implementation to improve the system's ability to gracefully degrade.

In general, system utility should be a measure of how well the system fulfills its requirements, and could incorporate many system properties such as performance, functionality, and dependability. An elevator system's primary function is to efficiently transport people to their destinations, minimizing how long passengers must wait for and ride in the elevator. Therefore, in our simulation experiments, we use the elevator's average performance per passenger as a proxy for measuring system utility. We track how long it takes for each passenger to reach his or her destination, from the time they first arrive to use the elevator to the time they step off the elevator at their intended floor. This is a relatively simple performance metric, and could be modified to account for worst case passenger travel times in addition to average passenger travel times. However, since this is a simulated environment and the passengers have simple behavior patterns, modifying the performance metric

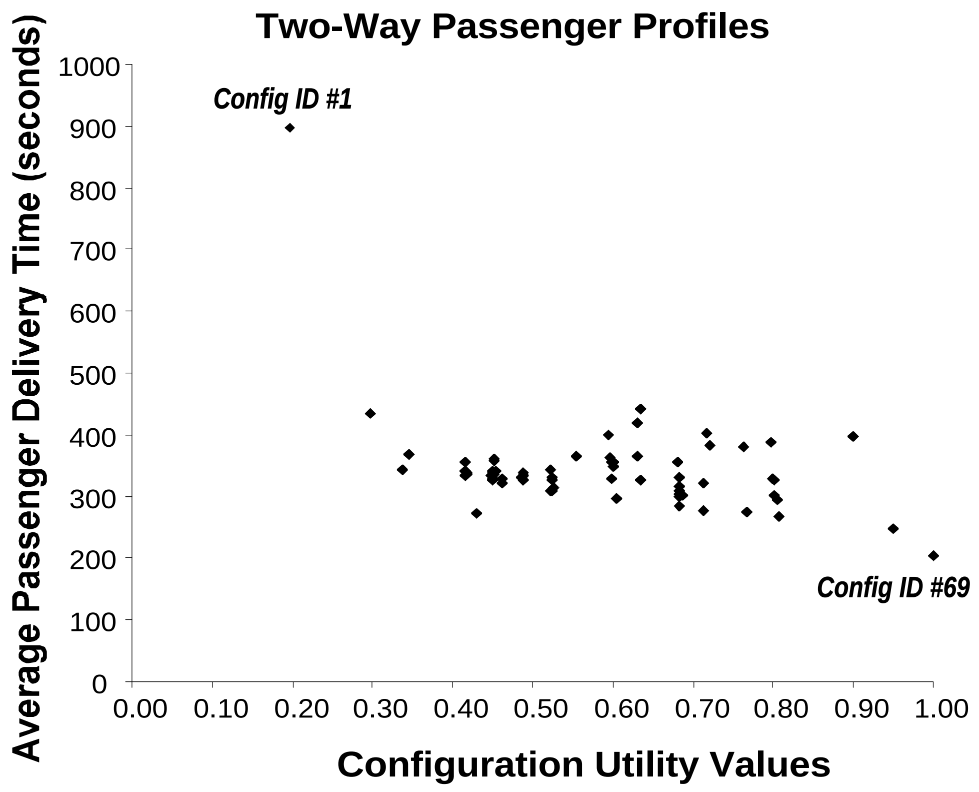

Figure 5.9. Average Elevator Performance vs. Model Utility for Two-Way Profiles.

did not seem necessary. In the data we examined, changing the performance metric did not significantly affect the relative order of the configurations tested.

We measured the average performance of each system configuration for each simulation test, and grouped the results according to the type of passenger profile tested. Then we took the mean of the average passenger delivery times for each of the three passenger profile types: two-way, down-peak, and up-peak, for each configuration. If our model accurately predicts system utility, we should see configurations that have higher utility measures achieve better average performance. Figures 5.9, 5.10, and 5.11 graph the utility of the tested system configurations versus the average elevator performance per passenger per simulation for each of the three profile types. In these graphs, better elevator performance translates to lower average passenger delivery times. The system configurations on the horizontal axis are ordered by utility, so the measured average passenger delivery time should decrease as utility increases to indicate better performance for system configurations that provide more utility. Appendix E lists all 70 configurations tested with a description of which components are failed in that configuration, its specified utility value, and the data for each of the simulation tests.

For the random two-way traffic profiles (Figure 5.9), the data indicates that the model accurately approximates relative system utility for the configurations tested. The configuration with the most components failed and the least utility (ID #1) has the longest average passenger delivery time at about 898 seconds per passenger. The configuration in which no components have failed (ID #69) has the shortest time with about 203 seconds per passenger. There is some variance in the performance measurements for configurations with similar utility values, but there is clearly a general trend of better average performance for systems with higher utility values. The configurations in the middle of the graph differ by which combinations of car call and hall call buttons have failed, and this can have a significant effect on elevator performance depending on the particular passenger requests.

For the down-peak traffic profiles (Figure 5.10), a similar trend of increasing system performance with increasing utility is visible. However, there are several outlying data points in the range of utility values of about 0.30 to 0.60 for which the configurations perform much better on the down-peak traffic profiles than their utility scores would seem to indicate. After investigating these configurations, we found that the major difference between them and the other configurations was that

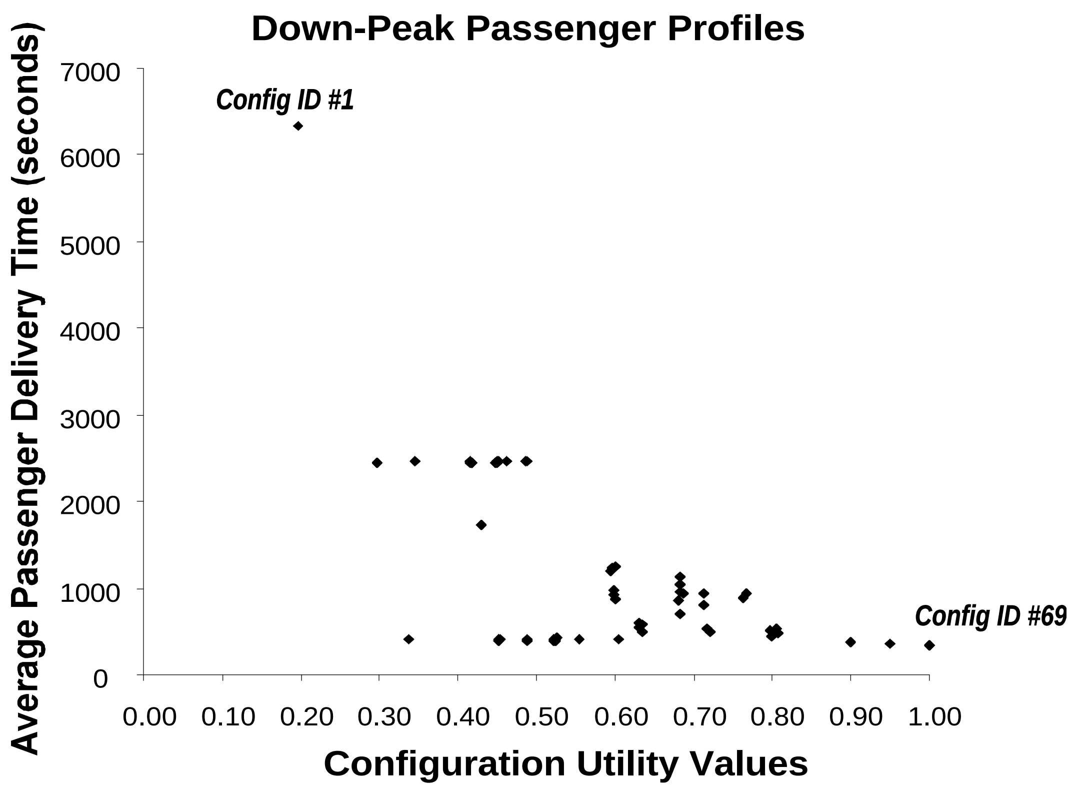

Figure 5.10. Average Elevator Performance vs. Model Utility for Down-Peak Profiles.

they had a working car call button for floor 1. Down-peak traffic is characterized by having 90% of all passengers request floor 1 as their destination.

One feature of our simulator is that the passenger behavior is such that they will continue to press their desired button until it either lights up or the elevator arrives on their desired floor. Also, the doors will reopen when a button press is detected on the desired floor, reducing the likelihood of the passenger being delayed by a door reversal. For configurations in which their desired button is broken, they will waste time pressing the button before exiting the elevator, and the doors will not respond to button presses, which can cause delays when multiple passengers are trying to exit the elevator on the same floor. Since the car call button for floor 1 is especially important to the elevator's operation on down-peak traffic, its presence or absence has a disproportionate effect on the system's performance, and is not accounted for

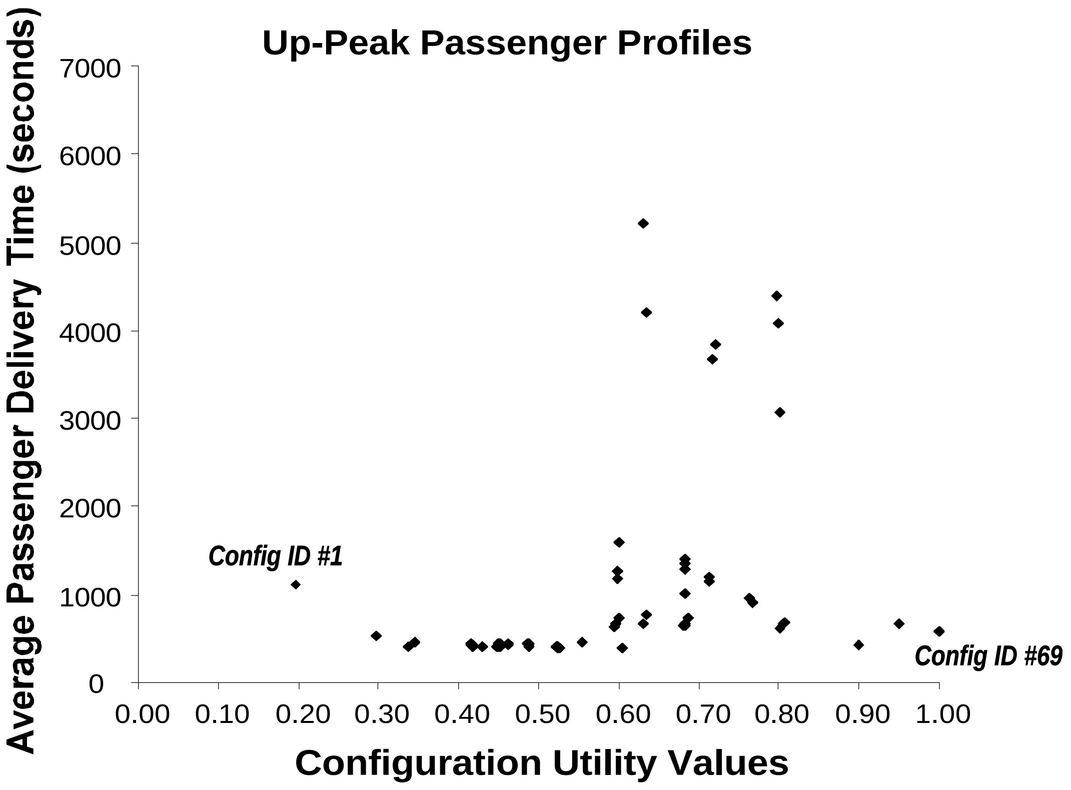

Figure 5.11. Average Elevator Performance vs. Model Utility for Up-Peak Profiles.

in our utility model. Thus, configurations in which the car call button is working will perform much better than their utility values indicate.

For the up-peak traffic profiles (Figure 5.11), the model does not seem to be as accurate at predicting relative system performance. Many configurations that supposedly have higher utility values and more working components perform much worse than configurations with low utility values. This is also due to an unforseen interaction between the characteristics of the up-peak traffic profiles and the graceful degradation mechanisms implemented in the system. Since up-peak traffic is characterized by 90% of the passengers arriving on the first floor to use the elevator, the drive controller's default algorithm for visiting floors is actually better suited for this traffic than the dispatcher's high performance algorithm.

The default drive controller starts at floor 1, stops at every floor until it reaches the top floor, and then returns to the first floor to repeat the process unless it receives an override destination from the dispatcher. For up-peak traffic, this will be very efficient since most passengers arrive on the first floor and exit on other floors. The dispatcher's algorithm will only perform reasonably well for up-peak traffic if the first-floor hall call button is working. If the first-floor hall call button is broken, the dispatcher will visit floor 1 periodically, but it will not process the first floor as frequently as it should for maximum performance, given that 90% of the passengers arrive there. All of the extreme outlying points in Figure 5.11 were traced to configurations in which the dispatcher was working but the first-floor hall call button was not.

Our utility specification gave equal weights to the utility contributions from all hall call buttons and most car call buttons. We did give the car call button for floor 1 more relative utility value than the other car call buttons, but this did not take into account its interaction with the door controller. Our experiments indicate that the utility model was relatively accurate for the general case of random two-way elevator traffic patterns, but was less accurate for the down-peak and up-peak traffic profiles. This was partially due to the fact that efficiently processing the up-peak and down-peak passenger profiles heavily depends on processing the first-floor button requests. When the first-floor hall call and car call buttons fail, the system's performance is severely degraded, and our utility model does not account for this. These tests indicate that additional hardware redundancy should be added to these first-floor buttons since they are critical to system performance for the up-peak and down-peak passenger profiles.

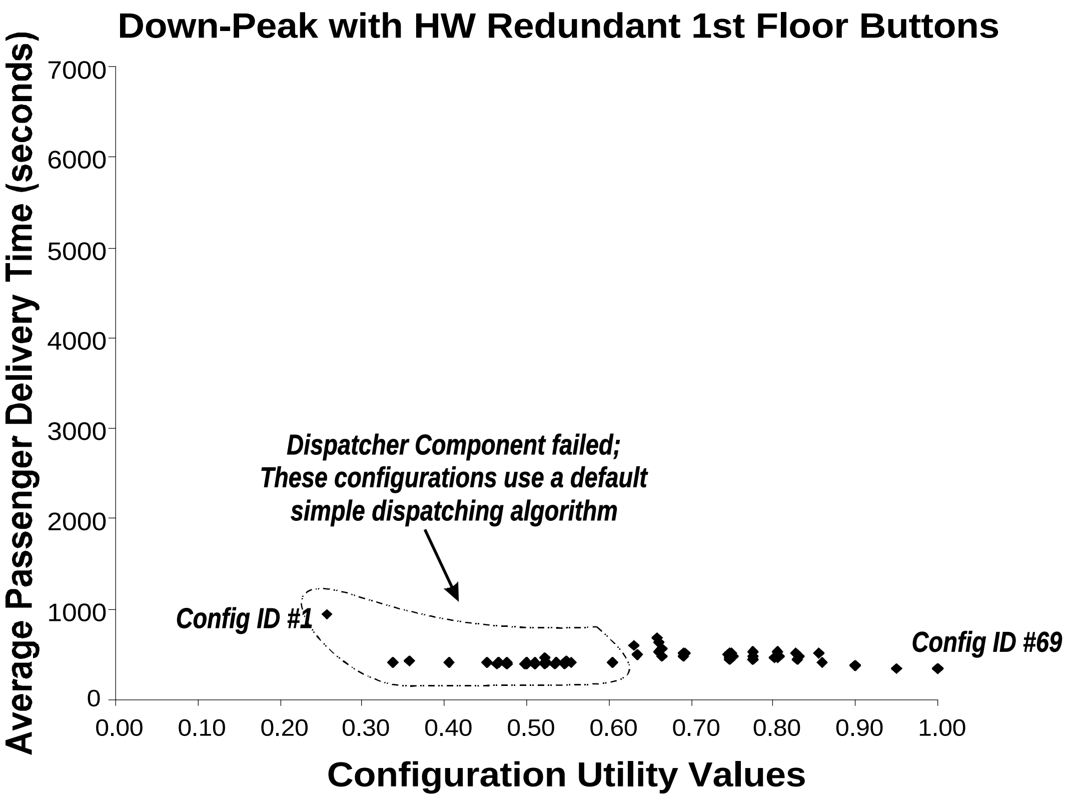

Figure 5.12. Average Elevator Performance vs. Model Utility for Down-Peak Profiles with HW Redundant First-Floor Buttons.

We reran the up-peak and down-peak passenger profile simulations with modified test configurations in which all the components in the original configurations were failed except for the first-floor car call and hall call buttons. These configurations represent situations in which the first-floor buttons have redundant hardware backups that mask a single button failure. Figures 5.12 and 5.13 show the results of these experiments. Once the first-floor buttons are removed from the possible failure configurations, our model more closely matches the performance of the elevator on the up-peak and down-peak passenger profiles. Additionally the performance of nearly all of the configurations significantly improves, as all of the average passenger delivery times for all configurations are less than 1,200 seconds, compared to the previous experiments in which some

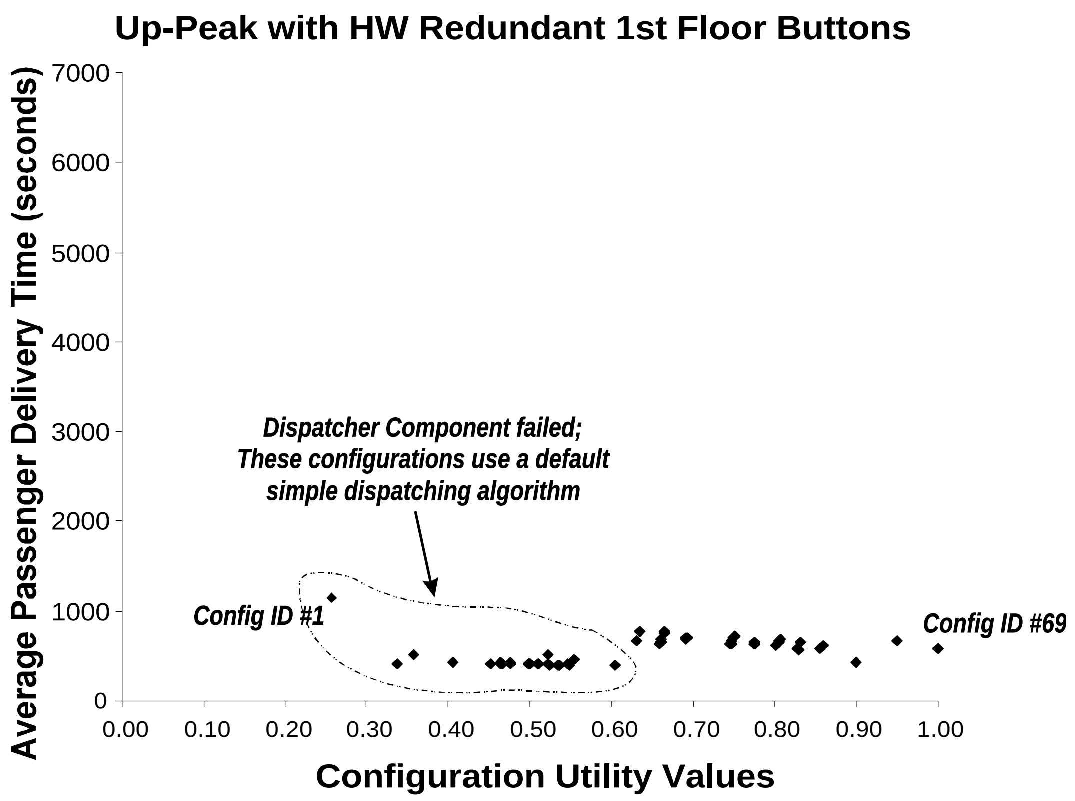

Figure 5.13. Average Elevator Performance vs. Model Utility for Up-Peak Profiles with HW Redundant First-Floor Buttons.

configurations had average passenger delivery times as bad as 5,000 or 6,000 seconds.

The dispatcher was designed to optimize performance for the random two-way passenger profiles, so the model does not completely match the observed performance for the up-peak and down-peak profiles, even with the first-floor buttons always working. One way to overcome this might be to specify utility functions that directly address these specific traffic patterns by giving more weight to the utility contributions from subsystems that particularly affect the performance under the special conditions. Then we could use multi-attribute utility theory [Keeney76, Keeney92] to combine the utility measures from the three traffic profile types based on which of these traffic types were considered more important to the system's operation. It might also be feasible to design a different dispatching algorithm for each traffic profile. This would also require that the elevator have some mechanism for switching modes based on the current traffic pattern.

Another solution might be to build a separate system utility function for each system mode of behavior that significantly changes utility measures. Each of these utility functions generates a system utility attribute. For each utility function, a new set of utility specification parameters would be generated for the system. Then general system utility would be a linear combination of the different utility attributes. This effectively multiplies the number specifications required for a complete utility function, but does not cause an exponential explosion.

5.5 Conclusions

We applied our system model and graceful degradation techniques to an elevator control system architecture to evaluate how well our model can measure the relative utility of system configurations, and whether the techniques we propose actually improve the system's ability to gracefully degrade. Since individual component failures simply transform the system from one configuration to another, we can evaluate how well the system gracefully degrades by observing the utility differences among valid system configurations. In the elevator system, we used our system model to generate a complete system utility function for all $8.54 * 10^{22}$ valid system configurations by only examining 424 subsystem configurations.

The experiments on a simulated implementation of the elevator control system produced several interesting results. The original elevator design could only tolerate failures in the car position indicator and car lanterns without failing to deliver passengers. However, our gracefully degrading elevator design could withstand up to a loss of 75% of the system's components and still provide service to all passengers, albeit at greatly reduced system performance. Every system configuration tested on the gracefully degrading elevator delivered all passengers to their destinations in all simulation tests, satisfying the minimum elevator system requirements despite a loss of system functionality. We also showed that the utility model was a good approximation for relative system utility among system configurations when the elevator traffic was random.

Our utility model did not account for how the elevator's performance would depend on particular components for up-peak and down-peak traffic profiles. Specifically, the first-floor hall call and car call buttons are necessary for the elevator to provide acceptable utility in these traffic profiles. Since our utility model assumed that each button subsystem per floor provided an equal contribution to the overall system utility, this affected our model's accuracy. This seems to indicate that even though replicated subsystems may be similarly designed and may appear homogeneous, system architects should also pay attention to how these subsystems are used in different system operating modes when evaluating their utility contribution. In our elevator system model, it would be reasonable to give more weight to the utility values of the first-floor hall call and car call button feature subsets relative to the other button feature subsets because they have a large impact on elevator performance in the up-peak and down-peak modes. In cases where multiple subsystems are affected by changing system modes, it may be necessary to specify multiple utility functions based on the different characteristics of these modes' operational profiles.

For graceful degradation in the elevator system we designed the software components to have a default behavior based on their required inputs, and to treat optional inputs as "advice" to improve functionality when those inputs are available. For example, the Door Control and Drive Control components can listen to each other's command output variables in addition to the Drive Speed and Door Closed sensors to synchronize their behavior (open the doors more quickly after the car stops), but only the sensor values are necessary for correct behavior. Likewise, the Drive Control component has a default behavior that stops the elevator at every floor, but if the Desired Floor system variable is available from the output of the Dispatcher component, then it can use that value to skip floors that do not have any pending requests. Also, the Door Control component normally opens the door for a specified dwell time, but can respond to button presses to reopen the doors if a passenger arrives.

We did not explicitly design failure recovery scenarios for every possible combination of component failures in the system, but rather built the individual software components to be robust to a loss of system inputs. The individual components were designed to ignore optional input variables when they were not available and follow a default behavior. This is a fundamentally different approach to system-wide graceful degradation than specifying all possible failure combinations to be handled ahead of time. Properties of the software architecture such as the component interfaces and the identification and partitioning of critical system functionality from the rest of the system seem to be key to achieving system-wide graceful degradation. This case study demonstrates the applicability of our model and techniques to adding graceful degradation to distributed embedded system designs.

6 Case Study: Autonomous Robot Navigation System

Now that we have demonstrated that we can apply our graceful degradation techniques to an existing system design, we will use our system model to guide building a gracefully degrading system from scratch. The CMU Mobot (Mobile Robot) competition [Mobot2003] provides a rich problem domain in autonomous vehicle navigation that has several opportunities for graceful degradation mechanisms. We decided to build a gracefully degrading mobot that could tolerate multiple sensor and software failures and still complete a race course within our lab.

This chapter details our attempt to build a gracefully degrading mobot system using our methodology. Section 6.1 describes the mobot navigation problem in detail. Section 6.2 shows how we used our system model for graceful degradation to drive our system architecture design and the techniques we used. Section 6.3 details our implementation and how we built the graceful degradation mechanisms into the system. Sections 6.4 and 6.5 conclude with the results of our case study.

6.1 Mobot Navigation Problem

The mobot competition involves designing an autonomous robot that can successfully navigate a race course in the shortest amount of time. The race course is on an outdoor concrete sidewalk with a white line painted on light gray concrete pavement. There are several gates placed at different points on the course which the mobot must pass through in sequential order. At the end of the course the white line has forks in several directions, and the mobot must pick the path that passes through the gates in the correct order. Contestants are given the locations of the gates prior to running the race so they can program the mobot ahead of time to follow the correct path. Each mobot is timed individually and runs the course alone without interference until it either completes the course or stops making progress.

At first look this seems to be a straightforward line following problem. With the appropriate sensors and a good line-following algorithm, the only design challenge would seem to be detecting the points in the course where the line diverges and taking the correct path. However, there are several features in the environment of the course that make this problem more interesting. Most mobot designs use some sort of visual sensor to track the line, such as infrared sensors or an embedded camera. Since the mobot course is outdoors, the ambient lighting conditions can vary greatly depending on the weather. Also, the contrast between the white line and light gray sidewalk pavement is not very large, and the cracks between blocks in the sidewalk disrupt the continuity of the line. These conditions make line following more difficult, and the mobots are susceptible to frequently losing track of the line in the course.

In order to explore graceful degradation opportunities, we generalized the line following problem to a navigation problem. The line can be treated as a high quality accurate source of position information that a general navigation system uses to complete the course. Other sources of location information could include tracking the cracks in the pavement as the mobot passes over them, and using shaft encoders on the wheels to measure distance traveled. A navigation algorithm that can take advantage of the course layout and keep track of its position could anticipate curves in the line and make more accurate turns to finish the course more quickly. Additionally, the navigation algorithm should be able to tolerate line sensor failures and continue to complete the course. If parts of the navigation system fail, the default line following algorithm can still take over and attempt to complete the course.

We constructed a smaller mobot course in our lab to run graceful degradation tests. The lab course uses semi-white masking tape for the line, which contrasts with the multicolored carpet. Although we do not have to deal with ambient light changes in the lab, distinguishing the carpet from the line is still a challenge because the carpet is composed of speckles of colors from near- white to black.

6.2 System Architecture Design

Our goal was to explore software design techniques for graceful degradation. Rather than completely design the robot hardware, we selected a hardware platform that provided some of the basic sensors and actuators, could easily accommodate additional sensor devices, and used a processor that had accessible software programming support. The basic hardware we started with was an ARobot mobile robot kit [Arrick2003] with a Basic Stamp 2 processor [Parallax2003]. The robot consisted of a chassis with three wheels. The front wheel axle has a drive motor for movement with a shaft encoder for position measurement. The rear wheels are connected to a servo motor that provides rear wheel steering. The front of the chassis has two whisker sensors for front collision detection. An embedded coprocessor on the robot controls the drive motor, servo motor, and encoder, and communicates via serial I/O to the Basic Stamp processor on the robot. The whisker sensors are directly connected to the Basic Stamp's I/O channels. The Basic Stamp can be programmed to send commands to the coprocessor to control the motors and read encoder values, and read the values of the whisker sensors. The additional I/O channels on the Basic Stamp can be used to connect additional sensing devices to the robot.

With this basic hardware platform, we focused on building a navigation system that could gracefully degrade when sensors and parts of the navigation system fail. We took a top down approach and started by specifying the actuators, system variables and software subsystems that are required for the navigation system. Figure 6.1 shows the feature subset diagram for the main navigation system.

The Navigation feature subset provides logical movement commands to the Actuator Control feature subset, which is responsible for moving the mobot to complete the course. Within the Navigation feature subset, the navigator software component receives data from the Line Follower, Direction, and Collision Detection feature subsets, which are derived from sensor subsystems built on the mobot that are described in the next section. These sensor subsystems can provide enough data to the navigator to reliably follow the line in the course, and also avoid obstacles that are sensed by the collision sensors. This is the basic functionality of a typical mobot system. Having the Direction feature subset so that the mobot knows what direction it is facing can prevent the mobot from making excessive turns that could throw it off course. Without any other navigation functionality, if the mobot ever lost track of the line, it would not be able to complete the course.

In order to compensate for a possible failure of the Line Follower feature subset, the Navigator also receives data from the Path Planner feature subset. This feature subset contains a Path Planner software component that receives data from Location

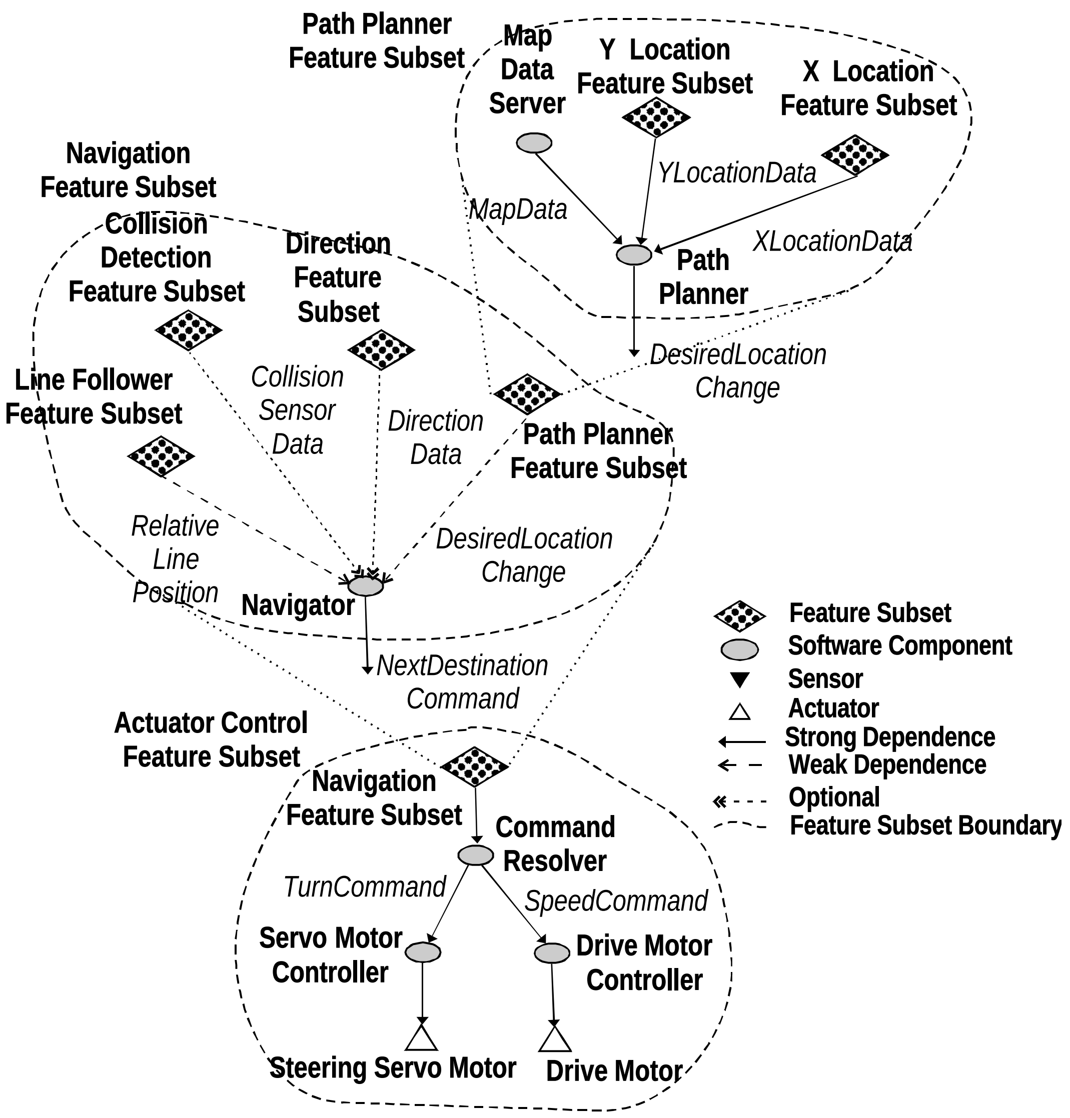

Figure 6.1. Navigation and Actuator Control Feature Subsets.

feature subsets and a Map Data component to synthesize the mobot's current position, and determine the best route for completing the course. The Map Data component provides a list of waypoint coordinates that will take the mobot on a path through the course. The Path Planner will then send a suggested location change command relative to the mobot's current position, that the Navigator can interpret into actual movement commands for the drive and servo motors. This navigation is based on using all available sensors to monitor the mobot's current location on the course, and doing a simple waypoint navigation based on knowledge of the course geography.

This is an example of using heterogeneous redundancy (Section 4.3.3) to provide graceful degradation for a critical subsystem (Section 4.3.2). The Navigation feature subset can continue to provide utility with either simple line following functionality, or high-performance location navigation.

The location data is split into separate subsystems for the X (axis parallel to course's direction) and Y (axis perpendicular to the course's direction). The X and Y Location feature subsets use heterogeneous redundancy to provide graceful degradation of the location data when sensors fail. Several sensor subsystems contribute to the location feature subsets and provide utility to the rest of the system.

6.2.1 Sensor Subsystems

We designed several sensors based on the hardware already available in the ARobot kit, and we also mounted some new sensors on the chassis. One of the major sets of sensors we added was the line following sensor subsystem. This subsystem consists of an array of infrared (IR) phototransistors and infrared light emitting diodes (LED) mounted close to the ground on the front of the mobot's chassis. Each sensor is a pair of one phototransistor and one LED. The LED shines IR light on the ground, which is reflected back up to strike the phototransistor. More light will be reflected from the white line than the rest of the carpet, so the change in the state of the phototransistor can detect whether or not the sensor is passing over the line. Six sensors are mounted an inch apart on the front of the mobot, to form an array that is normally perpendicular to the line when the mobot is travelling on the course. If the mobot starts to deviate from the line, it can be detected by the change in the IR sensor values.

Using these IR sensors, we can to a certain extent detect the relative position of the center of the mobot to the line. When the mobot is to the left of the line, the right sensors will sense the line, and when the mobot is to the right of the line, the left sensors will sense the line. As long at least one IR sensor is over the line, we can estimate the mobot's relative position, and the Navigator component can use the Line Follower's output to calculate a turn that brings the mobot's center closer to the line. Figure 6.2 shows the Line Follower and IR Sensors feature subsets, along with additional sensor subsystems we added.

In addition to using the IR sensors to follow the line, we can also use them to detect when the mobot has reached a decision point on the course. These decision points are characterized by a fork in the line that will continue the path in two directions. The IR sensors can detect this when both left and right sensors that are on opposite sides of the mobot sense that they are on the line. Once this is detected, we can choose which path to take. The Decision Point software component can maintain a list of decision points in the course and keep a history of how many decision points have already been passed to make the correct choice. Alternately, if course map data and location information are available from the Map Data component and X Location feature subsets, this will also provide reliable information about the correct fork to take on the path.

In Figure 6.2 we also have the feature subset definitions for Collision Detection and Crack Detection. The Collision Detector simply receives data from the whisker

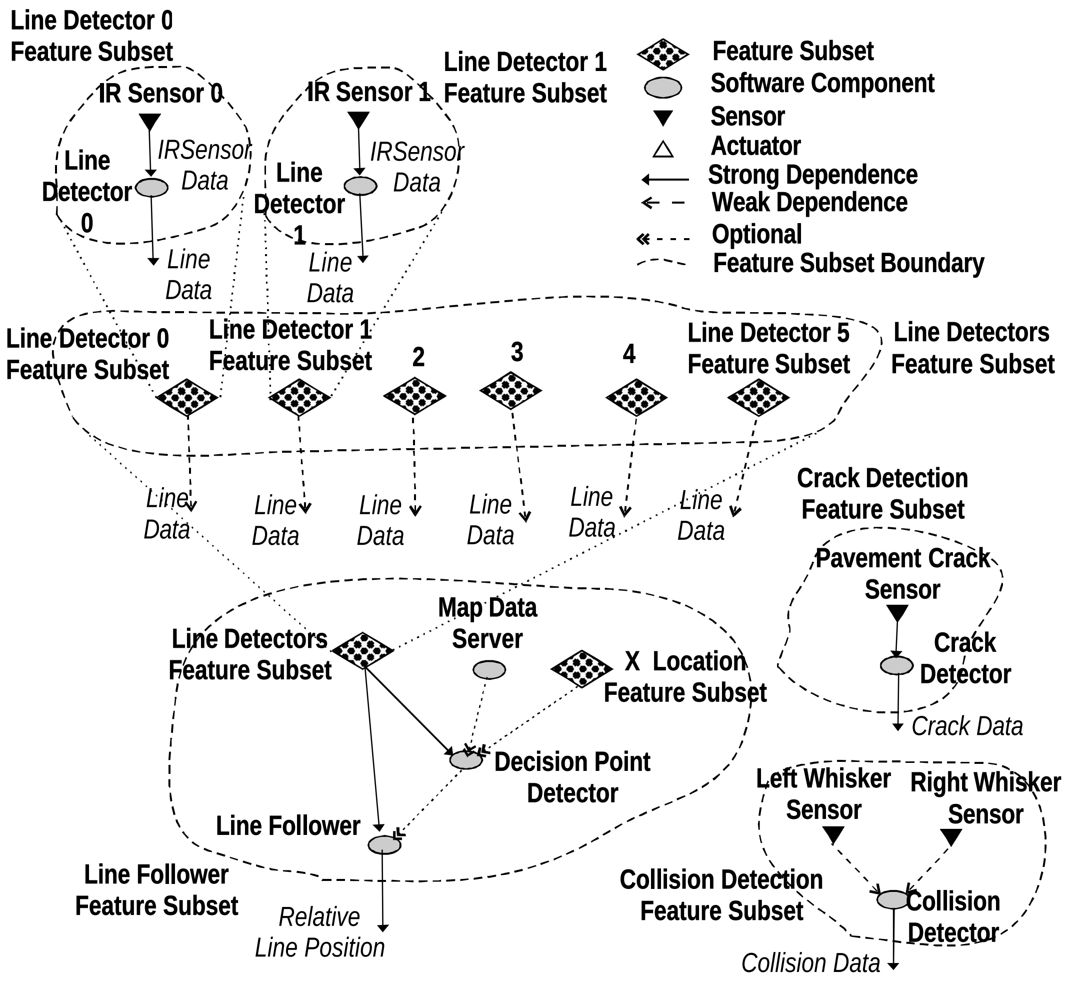

Figure 6.2. Line Follower, IR Sensors, Crack Detection, and Collision Detection Feature Subsets.

sensors on the front of the mobot. These whiskers are metal wires that stick out in front of the mobot’s frame. When the mobot strikes an object, these wires are pushed backwards and touch terminals on the mobot that complete a circuit. The middle of the course is free of obstacles, so when the mobot has a collision, this indicates that mobot is moving off course, and must backtrack. This can also be used as a rough position measurement, as it is confirmation that the mobot is at the edge of the course.

The Pavement Crack Sensor is a metal wire that drags along the ground on the back of the mobot. When the mobot passes over a crack, the wire dips lower, causing a hook on the other end of the wire to touch a terminal on the mobot. Since the sidewalk cracks are spaced at regular intervals on the course, this can be used as another source of position data.

A shaft encoder is already included on the front wheel's axle for distance measurements. We also added low resolution wheel revolution sensors on the rear wheels. These sensors consist of IR sensors mounted on the back of the mobot, with strips of white tape placed at regular intervals on the rear wheels' black tires. We can sense wheel revolutions by counting the number of times the white strips on the wheels pass under the IR sensors. These sensors are used for position tracking and dead reckoning, which is described in the next section.

6.2.2 Dead Reckoning and Location Subsystems

The dead reckoning subsystem uses the position tracking wheel sensors to estimate the mobot's change in position. Figure 6.3 shows the details of this feature subset. To calculate the mobot's speed and direction, we can use the outputs of the Command Resolver software component that sends the speed and direction commands to the actuators. Of course, this assumes that the actuators can precisely execute commands from the Navigation subsystem without any error or inaccuracy. Therefore, we can also use the wheel position sensors to estimate the mobot's change of speed and direction, and compare it with the Command Resolver's outputs. The mobot's speed and direction estimates are combined with the wheel

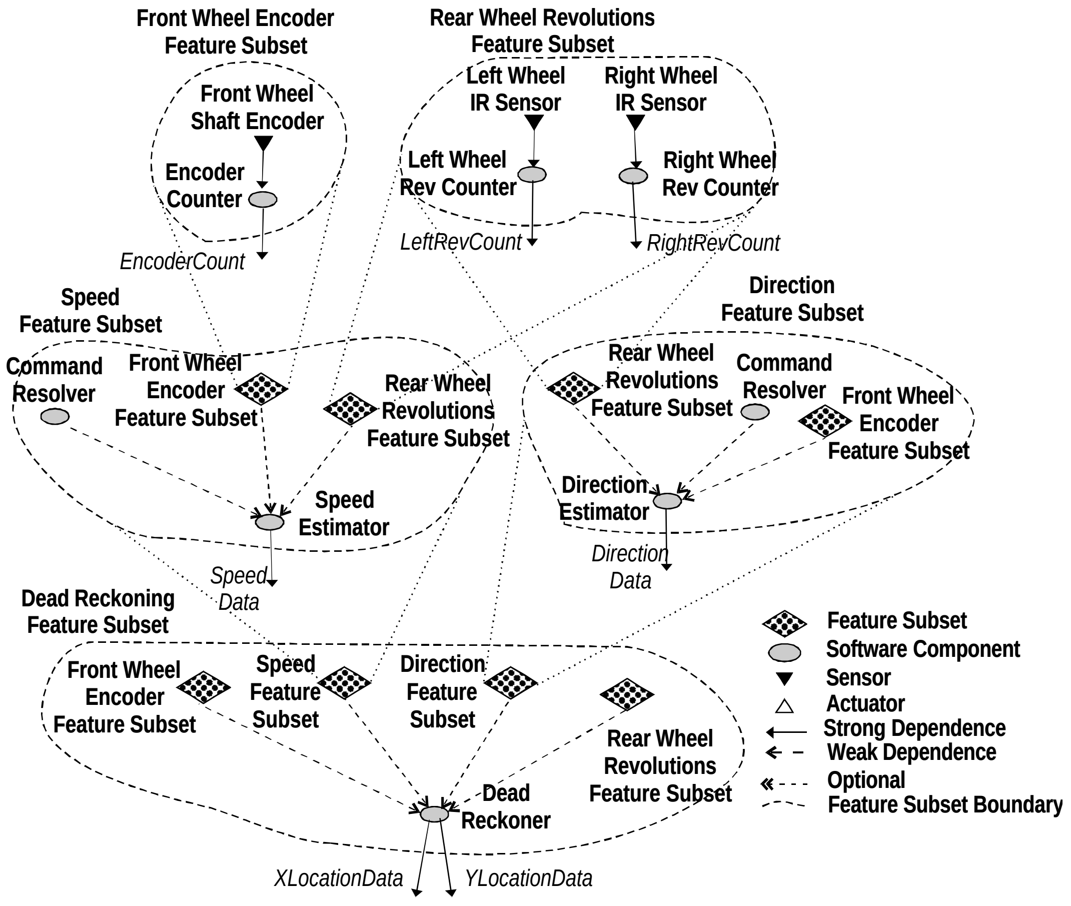

Figure 6.3. Dead Reckoning and Related Feature Subsets.

position data to estimate the mobot's location, which is output by the Dead Reckoning Feature Subset as the *XLocationData* and *YLocationData* system variables.

The Dead Reckoning Feature Subset provides location data to be incorporated into the X and Y Location feature subsets. These feature subsets collect position estimates from the multiple sensor subsystems and synthesize the location data used by the Navigation feature subset. Figure 6.4 displays the X and Y Location feature subsets.

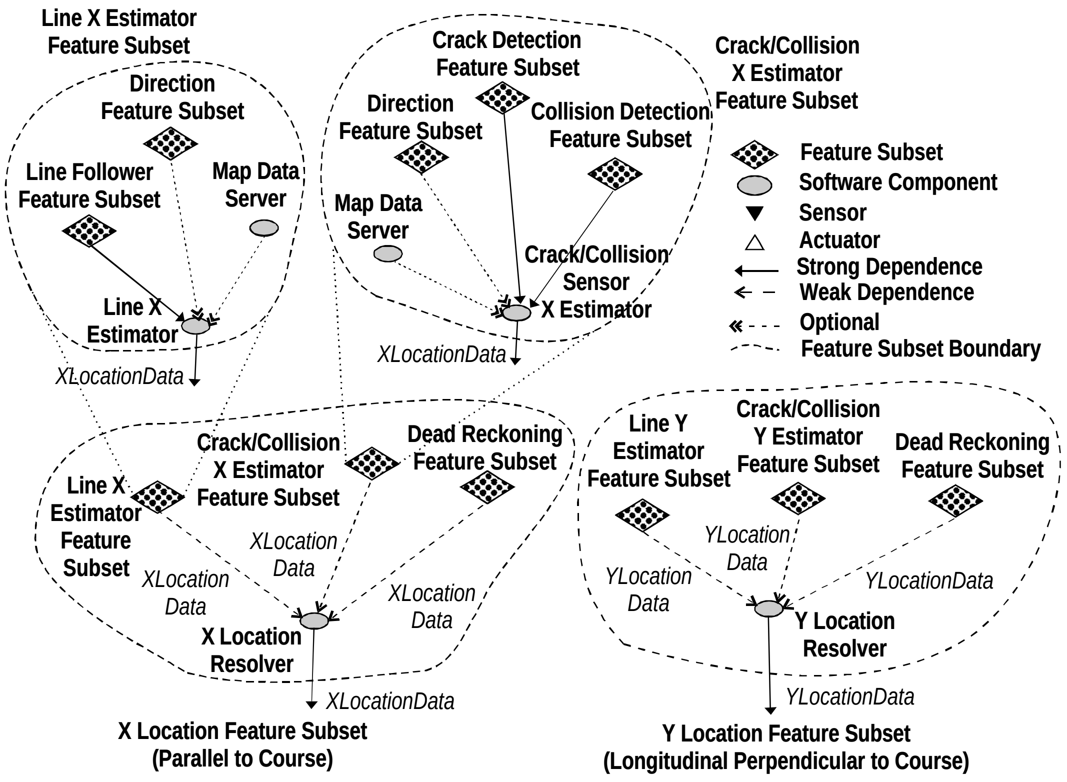
Line X Estimator Feature Subset
Direction Feature Subset
Line Follower Feature Subset
Map Data Server
Line X Estimator
XLocationData
Crack Detection Feature Subset
Direction Feature Subset
Collision Detection Feature Subset
Map Data Server
Crack/Collision Sensor X Estimator
XLocationData
Crack/Collision X Estimator Feature Subset
Feature Subset
Software Component
Sensor
Actuator
Strong Dependence
Weak Dependence
Optional
Feature Subset Boundary
Crack/Collision X Estimator Feature Subset
Dead Reckoning Feature Subset
Line X Estimator Feature Subset
XLocation Data
XLocation Data
XLocation Data
X Location Resolver
XLocationData
X Location Feature Subset (Parallel to Course)
Line Y Estimator Feature Subset
Crack/Collision Y Estimator Feature Subset
Dead Reckoning Feature Subset
YLocation Data
YLocation Data
YLocationData
Y Location Resolver
YLocationData
Y Location Feature Subset (Longitudinal Perpendicular to Course)

Figure 6.4. X and Y Location Feature Subsets.

The software components that make up these feature subsets should be separately designed based on how X and Y position can be estimated from each sensor's data. For example, the line sensors can provide data about the mobot's Y position based on the line's location and the mobot's position relative to the line. However, the line sensors are of little use in calculating the mobot's X position, except possibly at points in the course where the line turns. The X and Y Location Resolvers must deal with the possible loss of outputs from these sensor subsystems, and may provide a less accurate location estimate as a result.

The X and Y location estimates are coupled because they both rely on knowing the mobot's current direction to estimate change in position. The mobot's sensors periodically record the incremental distance traveled by the mobot, but knowing this distance without knowing the mobot's direction relative to the course cannot provide an accurate measurement of position change. Thus, the Direction feature subset provides inputs to all of the location estimators.

6.2.3 System Interface Design

We used our system model framework to guide how we partitioned the mobot's navigation system into subsystems. We used our system model's feature subset construct to define logical subsystems of components (as shown in this section), which led to the definition of logical system interfaces. The components and interfaces we defined are a result of our goal of making the system gracefully degradable. The multiple IR line sensors provide functional redundancy (Section 4.3.2) since multiple sensors in the array can fail while the remaining sensors can still provide line following utility. The X and Y Location feature subsets use the heterogeneous redundancy (Section 4.3.3) of multiple sensors to synthesize location information, and can continue providing utility until all of the sensor subsystems fail. The Navigation subsystem also uses two different algorithms (line following and position tracking) that require different sets of sensors to provide heterogeneous redundancy.

Our component and interface definitions are presented in Table 6.1. The system variable interfaces should, by design, facilitate graceful degradation. We broke the system into several logical subsystems based on sensor and actuator functionality, and identified pieces of the system that could serve as redundant resources. As specified in the feature subsets, we should design many of the components to treat inputs as optional whenever possible. There are 42 components in the system,

Table 6.1. Mobot Navigation System Component and Interface Specification.

Component	Component Type	System Variable Inputs	System Variable Outputs
Steering Servo Motor	Actuator	Raw Servo Control Data	Environment
Drive Motor	Actuator	Raw Motor Control Data	Environment
IR Sensors 0..5	Sensor	Environment	IRSensorData 0..5
Pavement Crack Sensor	Sensor	Environment	Raw Whisker Data
Left/Right Whisker Sensors	Sensor	Environment	Raw Whisker Data
Front Wheel Shaft Encoder	Sensor	Environment	Raw Encoder beats
Left/Right Rear Wheel IR Sensors	Sensor	Environment	IRSensorData L/R
Servo Motor Controller	Software	TurnCommand	Raw Servo Control Data
Drive Motor Controller	Software	SpeedCommand (Spd Cmd)	Raw Motor Control Data
Command Resolver	Software	NextDestinationCommand	TurnCommand, Spd Cmd
Navigator	Software	RelativeLinePositon, DirectionData, DesiredLocationChange, CollisionSensorData	NextDestinationCommand
PathPlanner	Software	MapData, XLocationData, YLocationData	DesiredLocationChange
Map Data Server	Software	N/A	MapData
Line Detector 0..5	Software	IRSensorData 0..6	LineData 0..5
Line Follower	Software	LineData 0..6, DecisionPointData	RelativeLinePosition
Decision Point Detector	Software	LineData 0..6, MapData, XLocationData	DecisionPointData
Crack Detector	Software	Raw Whisker Data	CrackData
Collision Detector	Software	Raw Whisker Data	CollisionData
Encoder Counter	Software	Raw Encoder beats	EncoderCount
Left/Right Wheel Revolution Counters	Software	IRSensorData L/R	Left/RightRevCount
Speed Estimator	Software	SpeedCommand, EncoderCount, L/RRevCount	SpeedData
Direction Estimator	Software	TurnCommand, EncoderCount, L/RRevCount	DirectionData
Dead Reckoner	Software	SpeedData, DirectionData, EncoderCount, L/RRevCount	X/YLocationData
X/Y Line Estimator	Software	RelativeLinePosition, DirectionData, MapData	X/YLocationData
X/Y Crack/Collsion Estimator	Software	CrackData, CollsionData, DirectionData, MapData	X/YLocationData
X/Y Location Resolver	Software	X/Y Location Data	X/YLocationData

meaning there are 2^{42} (more than $4 * 10^{12}$) possible system component configurations. We have used our system model to organize these components into 24 feature subsets. The number of feature subset configurations for which utility functions must be specified to generate the system utility function from our model is 156. Since only one feature subset provides outputs to actuators and encapsulates all of the other feature subsets, it is not necessary to specify system capability configurations. The system has one functional capability that drives the mobot to move in the course, specified by the Actuator Control feature subset. Appendix F details our utility model for the mobot system. The next section describes our implementation of this software system and how the components are allocated to the mobot's hardware.

6.3 Implementation

The mobot has very limited resources in terms of programming space. The Basic Stamp 2 has only 2 KB for both code and data, and the ARobot's coprocessor is already preprogrammed to control the shaft encoder, servo and drive motors, and cannot be readily reprogrammed. The coprocessor provides an interface through a serial I/O channel to the Basic Stamp to allow us to send serial commands that will make the coprocessor operate the motors or read encoder data. Therefore, the Basic Stamp processor must use its I/O to handle most of the sensors in the system. This leaves little room for navigation algorithms. We decided to use a general purpose laptop PC to host all of the system's navigation algorithms. The PC communicates with the Basic Stamp processor on the mobot via serial I/O interface. We

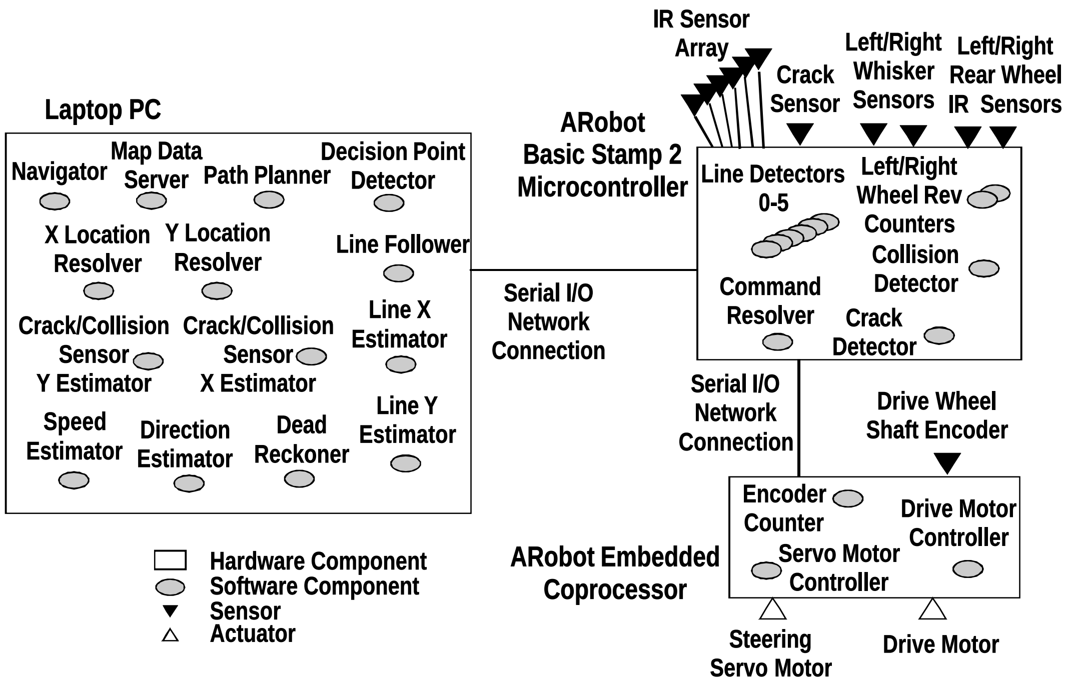

Figure 6.5. Hardware Allocation of the Mobot Navigation System.

programmed the Basic Stamp to periodically sample the sensors and send this data to the PC, which would then send navigation commands based on the mobot's current state. The mobot is tethered to the PC with a serial cable, allowing it free movement while periodically communicating with the PC.

We implemented the software components defined in our system model as subroutines running on the Basic Stamp written in the its PBASIC programming language and as objects in Java running on the PC that communicate via the serial interface. Figure 6.5 shows the hardware allocation of the software components in our mobot system. Most of the navigation components are allocated to the PC, while the software to control the sensors and actuators are allocated to the ARobot's processors. The limitations of the hardware available prevented us from using a more evenly distributed hardware system, but we can still observe how well the system gracefully degrades by injecting failures for individual software components and sensors.

6.4 Experimental Results

We ran several tests of the mobot in our lab to evaluate how well the mobot performed graceful degradation. The configurations we tested included ones in which the line follower software component and line sensors were broken, as well as ones in which the location and dead reckoning components were broken. Since our tests were run in the lab, and the course we used did not have decision points or sidewalk cracks, we did not implement the Decision Point Detector or the Crack X/Y Location Estimator components for these tests. The course in our lab was about seven feet long by four feet wide. For each configuration, we measured the distance the mobot went off track at the end of the course, as well as the time the mobot took to complete the course. We ran 10 tests for each configuration.

Table 6.2 describes these configurations, as well as their utility values predicted by the model. The performance metric for each configuration is calculated by the formula:

System Performance = 5/D + 1/T

where D is the average distance the mobot was off course and T is the average time the mobot took to complete the course in that configuration across the 10 measurements. These particular parameters are arbitrary, but we used this formula to give heavier weight to the mobot's ability to stay on the track compared to how fast it finished the course. Figure 6.6 graphically displays the results of our tests.

Table 6.2. Mobot Configurations Tested.

Config ID	Components Failed	System Utility Predicted by the model	System Performance Metric
1	None (except Decision Point Detector and Crack Location Estimators)	0.97	2.21
2	Line Sensors, Line Follower Component	0.72	1.69
3	Line Sensors, Line Follower Component, Front Wheel Encoder Sensor	0.67	0.17
4	Dead Reckoner Component, X/Y Location Resolver Component, Path Planner Component	0.49	0.52
5	Dead Reckoner Component, X/Y Location Resolver Component, Path Planner Component, Front Wheel Encoder Sensor	0.46	0.32
6	Dead Reckoner Component, X/Y Location Resolver Component, Path Planner Component, Front Wheel Encoder Sensor, Direction Estimator Component	0.41	0.23

Graph 6.6-A plots the utility value of each configuration versus the measured performance metric. Graph 6.6-B shows the time it took for each configuration to complete the course. Graphs 6.6-C and 6.6-D show how far the mobot was off course, both in terms of distance and percentage of the width of the course. For each configuration in graphs 6.6-B,C and D, the line represents the range of the values measured for the ten experiments run, and the middle bar represents the arithmetic mean of the tests.

As shown in the graphs and table, the configurations with dead reckoning, and combined dead reckoning and line following perform better than the other configurations, matching our utility model’s predictions. They are very close to the line at the end of the course, and reach their goal quickly. The configuration that only uses line following makes wide turns at the curves in the course, making it more difficult to closely follow the line and lengthening the time it takes to finish.

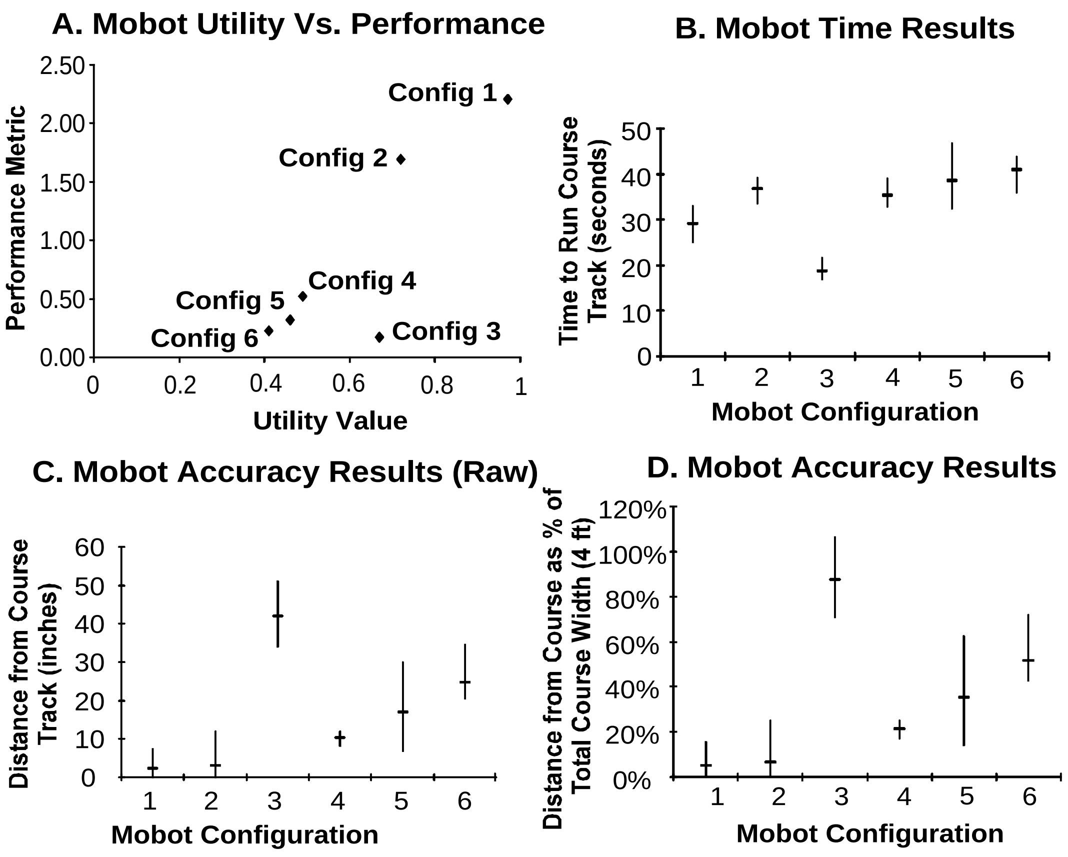

Figure 6.6. Mobot Configuration Experiment Results.

One interesting data point is Configuration 3, which has failed line sensors and a failed front wheel encoder sensor. In this configuration, the mobot must perform dead reckoning navigation, but with only the rear wheel sensors for both position and direction information. The rear wheels act as low resolution encoders that provide less accurate position data, and skew the dead reckoner’s calculations. The result is that the Navigator computes that it has reached the waypoint at the end of the course much more quickly because the position data is incorrect. Thus, the mobot is actually much more off course than the other configurations, but its time measurement appears to outperform the other configurations. This is why we

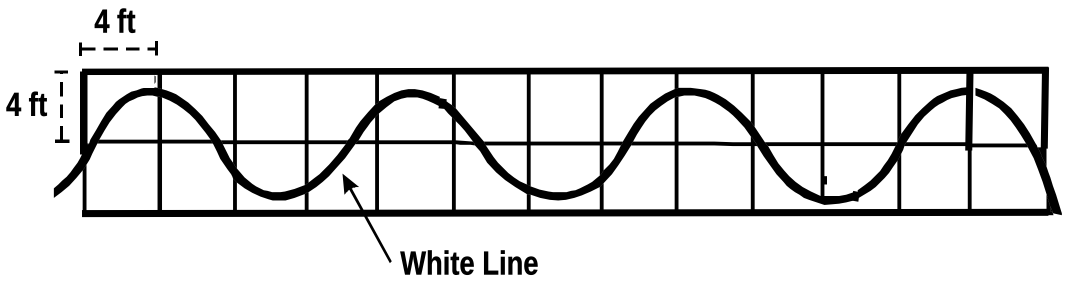

Figure 6.7. Middle Portion of the Official Mobot Course (Adapted from an Image at [Mobot2003]).

account for both accuracy and speed in evaluating the mobot's performance, with accuracy being the relatively more important attribute.

The experiments indicate that rear wheel revolution sensors are not accurate enough to serve as reliable data sources without any other source of position data. They produce erroneous outputs, which breaks our fail-fast, fail-silent assumption. Thus the configuration that only has these sensors available has a much lower performance score compared to its utility value predicted by the model. This analysis indicates that these sensors should be improved to provide more accurate data or removed from the system so that they do not cause a system failure.

We did try running the mobot outdoors on the CMU Mobot course, but the results were not as successful. The mobot was able to follow the line for a short distance, but the turning radius of the robot made it difficult to track the line accurately. Figure 6.7 displays a section of the official mobot course. The limitations of the mobot's hardware made it difficult to implement more complex control algorithms beyond simple bang-bang and proportional control.

6.5 Summary

In this chapter we have demonstrated the use of our methodology to build a gracefully degrading autonomous robot system. We started with a basic hardware platform, added some sensor systems, and constructed a navigation system based on the sensors available. We used our system model to guide the definition of hierarchical feature subsets that use heterogeneous redundancy to provide graceful degradation. This graceful degradation allowed the mobot to successfully navigate the course when subsystems fail. The component and interface definitions were designed to provide graceful degradation by breaking the system into logical partitions that could be decoupled from one another. We focused on designing the system to tolerate multiple sensor failures since we were limited in the hardware to one set of system actuators. Starting with feature subset definitions enabled us to logically define the software components and interfaces to produce a gracefully degrading navigation system.

This case study illustrates how embedded software system design can be severely constrained by the hardware resources. Ideally, we would prefer allocating the software components across more hardware nodes, but we were limited by the system's hardware platform. Despite this limitation, we were able to build the mobot's software system such that it could tolerate multiple IR sensor failures. Additionally, the mobot can tolerate a loss of the location navigation subsystem and continue to complete the course with simple line following.

7 Conclusions

This dissertation has presented a methodology for scalable specification, analysis, and design of graceful degradation in distributed embedded systems. An ideal gracefully degrading system minimizes the cumulative loss of system utility as successive system component failures occur. We designed a modeling framework that reduces the exponential effort required to specify the relative utility of all 2^N system configurations of N components. We then applied this modeling framework to some example embedded system architectures to identify some heuristic design techniques that can enhance a system's ability to gracefully degrade. We demonstrated the scalability of our system model and the applicability of our design techniques on two representative distributed embedded system architectures.

Section 7.1 revisits the proposed contributions of this thesis from Chapter 1, and summarizes how they have been fulfilled. Section 7.2 deals with the basic assumptions we made for building our system model and design techniques. We also explore the relevant system issues that affect the applicability of our methodology. Section 7.3 provides a discussion of future work and possible extensions of this research. Section 7.4 ends with a discussion of the results of this research.

7.1 Summary of Contributions

This thesis proposed a system model, analysis technique, and design techniques for scalable graceful degradation in distributed embedded systems. In the Introduction (Chapter 1) we proposed four major contributions of this research:

- A structural model derived from the system's software architecture specification that enables scalable specification of graceful degradation in embedded systems, and expresses many current hardware and software fault tolerance techniques in a single framework.
- A proposed set of design principles that will promote system-wide graceful degradation in distributed embedded systems that were identified as a result of applying the system model.
- An analysis technique that uses the model to provide hints to where to focus design effort for improving graceful degradation and can validate that the implementation achieves graceful degradation.
- Two case studies in which we applied our system model and design techniques to representative distributed embedded system applications and observed how well they could gracefully degrade.

Each of these contributions is discussed below.

7.1.1 System Model for Specifying Graceful Degradation

Our system model allows scalable specification of graceful degradation by exploiting the hierarchical partitioning of the system architecture that groups system components into subsystems. Our graceful degradation model uses utility as a general measure of the system's ability to satisfy its functional and dependability requirements. We can generate a system utility function that specifies the relative utility of all 2^N possible system configurations of N components at the cost of $O(N*2^k)$ utility evaluations, where k is the maximum number of components in an individual subsystem. This utility function allows us to evaluate how well the system gracefully degrades, because we can directly see how component failures (which alter the system configuration) affect system utility.

Our view of real-time embedded software systems consists of software components that represent periodic tasks that receive sensor data or state information, process this data and provide outputs to the rest of the system. The sensors are the data sources for the system, and the actuators are the data sinks that provide functionality based on the data values sent by their software controllers. At the architectural level, the traditional component and connector view captures relevant data dependencies among the software components. In hardware, these tasks are distributed across several processor nodes connected by a real-time broadcast network. The tasks communicate by periodically sending network messages that contain their output data.

In our model's software view, the components are represented as software components, sensors, and actuators, and the connectors are represented as system variables that represent data values passed among the components. We then partition the data flow graph of the software system into feature subset graphs that represent logical subsystems. These feature subsets are in general not disjoint and can share logical components across subsystems. This is necessary to capture the logical software dependencies in the system architecture without making assumptions about how components are allocated in hardware. For example, a single sensor could provide data to multiple subsystems by transmitting it on the network, or each subsystem could have its own redundant sensor. We also annotate the system variable connections between components based on dependency. A component can depend strongly or weakly on a system variable input, or treat that input as an optional “enhancing” input. We use these dependency relationships to identify all of the valid component configurations of all feature subsets, and then use their hierarchical organization to compose the overall system utility function.

The hardware view of our model is orthogonal to the software view and provides the allocation information about which components are running on which processor nodes. This view also identifies which components are replicated in hardware for reliability, and enables traditional reliability analysis techniques that account for hardware failures. A set of identical replicated components represents one software component in the software view. We can examine the effect of hardware failures on system utility by removing all of the components that were running on a failed node (provided there are no replicas running on other nodes) from the software configuration, and recalculating the utility.

7.1.2 Design Techniques for Graceful Degradation

We applied our system model to a hypothetical distributed embedded system architecture that was designed to provide graceful degradation to identify organizational properties that should enhance graceful degradation. The properties we identified include:

Specifying well-defined interfaces that decouple software components in the system. The interfaces among components determine the system variables in our model and thus how feature subsets are partitioned. The architecture should have a set of interfaces that represent intermediate computational steps that allow logical decoupling of subsystems. Our techniques for graceful degradation concentrate on making components robust to input failures, and providing redundant sources of system variables to tolerate subsystem failures. If the architecture does not have interfaces that promote hierarchical decoupling of its subsystems, these techniques may become cost prohibitive. If a system has few defined system variables, this would imply that feature subsets are large, monolithic, and complex, making them more difficult to design so that they can tolerate input errors and satisfy the fail-fast fail-silent fault assumption. If a system has many feature subsets that require many different system variables, it will be difficult to provide enough redundant resources to output these system variables and tolerate subsystem failures.

Adding limited redundant resources to critical subsystems that are necessary for system functionality. One of the benefits of our system model is that we can use the dependency relationships among components to immediately identify which components and feature subsets are required for the system to provide any utility at all. We can concentrate fault tolerance design effort and redundant resources on preserving these parts of the system, rather than replicating every component in the system for dependability. As long as the system obeys our initial fault assumptions, then components that are identified as non-critical cannot adversely affect the operation of critical components and subsystems.

Exploiting heterogeneous system resources to provide auxiliary redundancy through common system interfaces. Many embedded systems are designed to optimize resources to provide a large set of features and functionality. Many of the system resources that provide these features can be viewed as functionally equivalent and actually provide a redundant backup when a component failure occurs. For example, multiple sensors that provide different levels of accuracy for a measurement, or measure different aspects of the same environmental phenomena can provide redundant data sources when combined with a software component that transforms the sensor data to a common system variable interface. This technique exploits redundancy inherent in the system architecture and does not require additional system resources to tolerate a failure.

Designing components with default behaviors that take over when inputs from other components are lost. As a complementary approach to providing redundant sources of system variable inputs, we also propose designing components to treat their inputs as optional whenever possible to reduce their dependency on inputs. Our initial approach to accomplishing this is to identify the minimum service a component must provide and the minimum inputs it requires for this service. Then we design a behavior that can satisfy this minimum functionality requirement in addition to the component's normal behavior. The component will provide its full functionality when all of its inputs are available, but if an input failure is detected, the component will switch to its simple backup behavior until the input is restored. If there are multiple inputs that can potentially fail, the component can be designed with multiple algorithms depending on the amount of design effort that can be spared to make this component robust to input failures.

7.1.3 Analysis for Validating Graceful Degradation

We can use our system model to analyze the system architecture to target which components and feature subsets should receive graceful degradation support. We use the scalable system utility function generated from the system model and evaluate system configurations to identify which components and feature subsets contribute significantly to overall system utility. We then target these parts of the system for graceful degradation improvements using the design techniques we have already identified. Any components or feature subsets that are single points of failure or drastically reduce system utility when not available should be targeted for graceful degradation mechanisms.

The model analysis provides information about which feature subsets and components are critical to system utility, allowing us to target these parts of the system for graceful degradation mechanisms. Choosing which techniques to implement requires an analysis of the tradeoffs between the resources available in the system and the level of dependability required. Our scalable specification framework should enable these tradeoffs to be explicitly identified with the utility model and information about the resources required for system components and feature subsets.

In addition to using the model at design time to determine where graceful degradation mechanisms should be applied in the system, the model can also be used to validate whether or not the system implementation achieves the level of graceful degradation predicted. In general the utility model should reflect each component and feature subset's contribution to system utility. If we have a utility metric that incorporates some or all of the desired system properties defined in the system's requirements, and these attributes can be measured in the system implementation, then the relative differences between the utility of system configurations predicted by the model should match the actual measured utility differences of these configurations in the implementation. If there are configurations that do not fit the expected rankings, they may indicate either an inaccuracy in the system model, a dependability problem in the system implementation, or a violation of the model's assumptions in the system design. We can use this analysis iteratively to both refine the system model and identify dependability bottlenecks in the system implementation.

7.1.4 Case Studies that Illustrate the Methodology

We presented two case studies in which we applied our system model, analysis, and design techniques to develop a gracefully degrading embedded system architecture. The first was an existing, detailed elevator system architecture that was implemented in a discrete event simulation. The elevator architecture and implementation had already been thoroughly exercised through several iterations on a distributed embedded systems class design project. When we applied our model and graceful degradation techniques to this elevator system, the implementation was able to tolerate many more component failures than the original elevator design.

The major improvements we made to the system included adding software components that could synthesize floor sensor messages from the car speed and position sensors (adding heterogeneous redundancy to critical subsystems), thus providing a backup for failed floor sensors; modifying the dispatcher software component to periodically synthesize floor requests for floors that had failed buttons (designing components to be robust to input failures); and modifying the drive controller to follow a default pattern of periodically visiting all floors when the dispatcher input is lost (designing components to be robust to input failures). These changes required very little extra code in the implementation (a 9% increase in total lines of code of the control system) and made the system resistant to up to 75% total system component failures.

We also ran experiments to measure how well our system utility model predicted the relative utility values of different elevator configurations, in terms of minimizing average passenger travel time. We ran tests on the three major types of elevator traffic: normal two-way traffic, up-peak traffic, and down-peak traffic. Our model was relatively accurate for the two-way traffic cases, but was significantly less accurate with the up-peak and down-peak traffic profiles. This was due to the fact that up-peak and down-peak traffic are heavily dependent on the operation of the first-floor buttons to provide efficient service, and our utility model did not account for that. Our conclusion is that replicated subsystems that are similarly designed do not always have an equal effect on system utility. Special cases such as the first-floor hall call and car call buttons should be given different weights in the utility model to account for their impact on system utility. In cases where the system has multiple operating modes that affect the utility contributions of many subsystems, it might be necessary to develop multiple utility parameters for utility models in the system based on the system's operational profiles, and design a multi-attribute utility function that gives more weight to the utility values based on more likely or more important scenarios.

The second case study involved the design of a gracefully degrading autonomous mobot (mobile robot). The mobot must navigate a race course without getting lost by following a white line. We started with an off the shelf robot kit with three wheels, a motor, encoder, servo, whisker collision sensors, and two embedded processors, and added several sensor systems such as infrared line sensors, a pavement crack detector, and rear wheel revolution sensors to provide opportunities for graceful degradation. We used our system model to build a software system that can gracefully degrade when combinations of sensors fail. We were successful in that the robot could separately tolerate both a failure of the line sensors, as well as a failure of the navigation subsystem and still complete a test course we designed in the lab. This case study also demonstrated that hardware constraints and limitations can have an affect on the implementation of the "ideal" software system as envisioned in our system model.

7.2 Assumptions and System Design Issues

The applicability of our system model and graceful degradation techniques is predicated on several assumptions about the system being designed. We have narrowed our focus to distributed embedded real-time systems and specified a fault model. There are also some design issues that are not addressed in this thesis that will affect how this methodology can be applied to embedded system architectures, such as the feasibility of designing all software components to be robust to input failures, and exactly how designers construct their system and feature subset utility evaluations.

7.2.1 Embedded System Architecture and Fault Model

Our system model was specifically designed to be applicable to distributed embedded system architectures. We focus on the software architecture of the application, and make some assumptions about the system's hardware and network organization. We assume the network uses broadcast communication among nodes so that the software architecture is decoupled from the system communication mechanisms. We also assume that there are sufficient hardware resources available to satisfy memory, bandwidth, and real-time requirements. Our graceful degradation techniques emphasize how to modify the components that make up the software system to tolerate component failures.

Our model assumes a fail-fast, fail-silent fault model with perfect fault detection. However, we can relax this assumption based on how we constrain the system architecture and the distributed nature of the system. For graceful degradation we are more concerned with the effects a fault produces rather than the source of the fault. In order to minimize the failures a fault can produce, as well as minimize fault propagation, we constrain our system architecture to only allow communication among software components via system variables. As long as the implementation adheres to the architecture and does not provide hidden communication channels among software components, faults can only be propagated through the defined system variables.

We believe the fail-fast, fail-silent fault model is a reasonable assumption because we are only concerned with faults that cause corruption of the system variables' state, and this corruption can be readily detected by the system variables' receivers. We do not envision a centralized failure detection infrastructure, which itself could be a single point of failure, but rather software components that validate their system variable inputs as they are updated and only use those inputs if they pass the validation tests. Simple checks on the input variables can catch many errors, and scale with the number of inputs per software component. This fault model can cover several types of the component failures described below.

A software component could fail to update a system variable at the appropriate time; a system variable could be corrupted to an invalid state either by the sender, receiver, or communication medium; or the system variable could be corrupted to a valid but incorrect state. In a real-time system, failure to update a system variable can be detected as its deadlines are missed. If a system variable becomes stale, i.e. it hasn't been updated for several periods, then the receivers of that variable can assume that the sender has failed or is unreachable. If receivers detect invalid data in a system variable for multiple consecutive periods, then they can assume that the sender has failed. The most difficult failure to detect is when a system variable has valid but incorrect data. These failures cannot be easily distinguished from a correct system variable that is manifested by an exceptional condition occurring in the environment.

7.2.2 Generating the System Utility Model

Specifying the utility model is still a challenging problem. A comprehensive quantitative utility model that accounts for all relevant functionality and dependability properties is a significant undertaking. We have built a framework that reduces the number and scope of utility analyses to be within individual subsystems. A system designer can qualitatively rank the component configurations of individual feature subsets, so that we can approximate utility functions by generating linear functions for each configuration based on the component utilities. We assume that the system is decoupled so that the form of the utility function of each feature subset is only dependent on the component configuration rather than each component's utility value.

We must also relate feature subset utility to overall system utility. We use the functional capability definitions to accomplish this, and they are based on the fact that most architectures are decomposed into major subsystems that each provide functionality. At the system level, these functional capabilities each contain feature subsets that can contribute to their utility. The system utility function is then based on the configuration of functional capabilities, and the utility functions defined for each configuration. The system utility function is heavily dependent on the definition of the functional capabilities, which must be specified according to which feature subsets provide which functionality.

There is a possibility that our system model may achieve scalable utility analysis at the expense of accounting for how interactions between components and subsystems affect system utility. If components and subsystems are tightly coupled, it may be necessary to build a more complex utility model using multi-attribute utility theory [Keeney76, Keeney92] that explicitly accounts for couplings between system components. Designers may also want to develop a more detailed utility model that explicitly identifies utility attributes, to analyze trade-offs among competing system properties such as performance and dependability. An architectural analysis method such as the Architecture Trade-off and Analysis Method (ATAM) [Kazman98] that evaluates system quality attributes may aid development of a multi-attribute utility model.

7.3 Future Work

This thesis is a first step towards a general methodology for graceful degradation. There are several challenges for this research that could be extensions of this work. Some future extensions include building a tool to automatically generate the system model and utility function from a system's architectural specification, relaxing some of the assumptions of our model and extending the model to be applicable to other types of computer systems and software architectures, identifying other graceful degradation design techniques, and using our view of system configurations to build product family architectures.

Generating the system model from software component and interface specifications is relatively straightforward, so building a semi-automated tool should be a reasonable extension. The software data flow graph can be generated by connecting the input and output interfaces of software components, and the feature subsets can be defined by creating subgraphs at each interface boundary. Valid feature subset configurations can be identified by traversing input and output dependency links between components and determining which components are required for each feature subset to provide minimum utility.

We have had some success with building a prototype tool that can parse a text specification that lists all system variables, components, and each component's input and output interface. This is sufficient to generate a system model with feature subset definitions. The process is not completely automated, as a designer still must verify that the feature subsets capture the subsystems he or she designed into the system, specify the system's functional capabilities in terms of which feature subsets provide which system functionality, and generate the utility function parameters for each feature subset configuration. It should be possible to build a tool that can extract relevant information for our model from software architecture specifications that are expressed in an architecture description language (ADL) such as Acme [Garlan2000] that emphasizes component and connector definitions.

Currently our model makes several assumptions that narrow its applicability to distributed embedded systems, and our design techniques for scalable graceful degradation are predicated on these assumptions. Our model assumes that the individual software components are strongly decoupled, and can only affect each other though the defined communication interfaces. However, many software systems may have hidden dependencies between components in their implementation that can allow faults to propagate. For example, components that run in the same process may have access to the same memory space, so that a defect in one component may cause it to overwrite another component's data and subsequently cause errors in other parts of the system. Our system model may still be applicable in these situations if a more sophisticated fault model that accounts for more pernicious failures due to software defects can be mapped to how component failures affect the system's configuration.

Since we have made system variables a key mechanism in our system architecture, it may be desirable to specify how these variables represent their accuracy or quality as a part of their system state. This would aid utility evaluation because a component and feature subset's utility value could be mapped to the quality of its output variables. One approach would be to represent data accuracy as a range of uncertainty or confidence interval. This might work well for numerical data types and is flexible in that the accuracy of a system variable can be dynamically updated while the system is running. However, this approach might require a heavyweight analytical model for each producer of the system variable that would be costly to implement. Additionally, this model would not work well for non-numeric and categorical system variable data types.

Another approach would be to specify the quality of data for the outputs of each software component at design time. The system designers could rank software components that produce the same system variables based on the algorithms they use. This static ranking would then be used at run time by the receivers of the system variables to determine which ones to use. Additionally, similar data from different senders that have significant qualitative differences could be defined as two separate system variables. This approach has the advantage of requiring fewer system resources to implement and is reasonable when there are not many independent sources of the same system variable. The drawbacks of this approach are that it is less flexible since changes in system variable quality during runtime cannot be detected, and receivers of system variables have a heavier burden in deciding which inputs to use.

Our view of the system is that components only communicate via data flow in system variables, but there are many other architectural patterns in which control flow rather than data flow is the major connection between components. To generate feature subset definitions and valid component configurations, we focus on the dependency relationships among the components, which manifest as data flow relationships in distributed embedded systems. It may be that we can still use our model to represent dependency relationships, but they may represent different connector mechanisms for different architectural patterns.

The graceful degradation design techniques we have proposed do not represent all possible mechanisms, but rather what we identified using our model. The fact that we are able to represent many current software fault tolerance techniques gives us some confidence that this model can be used to identify other graceful degradation opportunities. We have specifically focused on techniques that are integrated into the software architecture to provide immediate failover mechanisms when faults are detected, and do not depend on global fault detection and reconfiguration to recover from errors. We view reconfiguration as a complementary approach that can provide more recovery alternatives beyond our failover approach, such as reallocating software components to different processor nodes and restarting failed components. There may also be other techniques that can be developed from work in research in self-healing systems [WOSS2002].

Our model was initially designed to enable scalable specification of utility differences between different software component failure configurations in a system, but it can also be used as a view of a product family architecture. Each product instance could be represented by a different component configuration. A high-end product would have a configuration with most of the components present and providing functionality, while a low-end product's configuration may only have enough components to provide minimum functionality. The parallel between a gracefully degrading system and a product family architecture could be exploited to provide systems that naturally gracefully degrade by design.

7.4 Concluding Thoughts

Graceful degradation mechanisms can offer improved system dependability with few redundant resources, but at the cost of additional system design effort. Prior to this work, comprehensive system-wide graceful degradation required an exponential specification and design effort with respect to the system component faults being covered. This thesis provides a methodology for evaluating and designing scalable graceful degradation for distributed embedded systems by taking advantage of the hierarchical structure of these system's architectures.

Graceful degradation is especially important for distributed embedded systems because these systems typically cannot afford the additional system resources required for fault tolerance mechanisms that maintain both system dependability and complete functionality. Graceful degradation is an alternative to brute force redundancy that can potentially provide the same level of dependability at the expense of reduced functionality when failures occur. This research indicates that distributed embedded systems can exploit graceful degradation opportunities because a significant portion of these systems is designed to provide enhanced or auxiliary functionality above the functionality required to satisfy the system's primary mission.

Hierarchical decomposition is the best technique designers have for managing functional complexity, and our model for specifying graceful degradation exploits this to achieve scalability. Building a gracefully degrading system can increase the system's complexity, and our system model provides a mechanism for managing this complexity without sacrificing graceful degradation opportunities. We restrict our design techniques to the component and subsystem level to limit the total impact of these techniques on system complexity. As systems become more complex with added features and functionality, techniques for scalable graceful degradation will become increasingly important for managing system failure modes. Our model for evaluating the utility of system configurations enables scalable analysis and design of graceful degradation in distributed embedded system architectures.

8 References

[Abdelzaher97] Abdelzaher, T.F., Atkins, E.M., Shin, K.G., "QoS Negotiation in Real-Time Systems and its Application to Automated Flight Control," *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium*, Montreal, Quebec, Canada, June 1997, pp. 228-238.

[Adlemo95] Adlemo, A., Andreasson, S.-A., "Improved Availability in Manufacturing Systems Through Graceful Degradation: Case Study of a Machining Cell," *Proceedings of the 1995 IEEE International Conference on Robotics and Automation*, Nagoya, Japan, May 1995, pp. 1744-1750.

[Arrick2003] *ARobot Robot Kit for Experimenters and Educators*, Arrick Robotics, <http://www.robotics.com/arobot>, 2003.

[Arora98] Arora, A., Kulkarni, S.S., "Component Based Design of Multitolerance," *IEEE Transactions on Software Engineering*, Vol. 24, No.1, January 1998, pp. 63-78.

[Avizienis85] Avizienis, A., "The N-version approach to fault-tolerant software," *IEEE Transactions on Software Engineering*, SE-11(12), December 1985, pp. 1491-1501.

[Avizienis2001] Avizienis, A., Laprie, J.-C., and Randell, B., *Fundamental Concepts of Dependability*, Research Report N01145, LAAS-CNRS, April 2001.

[Banks94] Banks, S., Coleman, N., et al., "Architecture Based Approach to Control Software Design," *Proceedings of the 1994 American Control Conference*, Baltimore, MD, USA, June 1994, pp. 2731-2735.

[Bass98] Bass, L., Clements, P., Kazman, R., *Software Architecture in Practice*, Addison-Wesley, Reading, Massachusetts, 1998.

[Boasson98] Boasson, M., "Software Architecture for Distributed Reactive Systems," *Twenty-Fifth Conference on Current Trends in Theory and Practice in Informatics (SOFSEM '98)*, Jasna, Slovakia, November 1998, pp. 1-18.

[Bodson93] Bodson , M., Lehoczky, J., et al., "Control reconfiguration in the presence of software failures," *Proceedings of the 32nd IEEE Conference on Decision and Control*, San Antonio, TX, USA, December 1993, pp. 2284-2289.

[Botti2000] Botti, O., De Florio, V., et al., "The TIRAN Approach to Reusing Software Implemented Fault Tolerance," *Proceedings 8th Euromicro Workshop on Parallel and Distributed Processing*, Rhodos, Greece, January 2000, pp. 325-332.

[Dvorak2000] Dvorak, R., Rasmussen, G., et al., “Software Architecture Themes in JPL’s Mission Data System,” *Proceedings of 2000 IEEE Aerospace Conference*, Vol. 7, Big Sky, MT, USA, March 2000, pp. 258-269.

[Garlan2000] Garlan, D., Monroe, R.T., Wile, D., “Acme: Architectural Description of Component-Based Systems,” *Foundations of Component-Based Systems*, G.T. Leavens and M. Sitaraman (eds), Cambridge University Press, 2000, pp. 47-68.

[Herlihy 91] Herlihy, M. P., Wing, J. M., “Specifying Graceful Degradation,” *IEEE Transactions on Parallel and Distributed Systems*, Vol.2, No.1, 1991, pp. 93-104.

[Jayanti99] Jayanti, P., Chandra, T.D., Toueg, S., “The Cost of Graceful Degradation for Omission Failures,” *Information Processing Letters*, Vol. 71, No. 3-4, 1999, pp.167-172.

[Jurgen99] Jurgen, R.K., ed., *Automotive Electronics Handbook, Second Edition*, McGraw-Hill, New York, 1999.

[Kazman97] Kazman, R., Clements, P., et al., “Classifying Architectural Elements as a Foundation for Mechanism Matching,” *Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC’97)*, Washington, DC, USA, August 13-15, 1997, pp. 14-17.

[Kazman98] R. Kazman, M. Klein, M. Barbacci, H. Lipson, T. Longstaff, and S.J.Carrière, “The ArchitectureTradeoff Analysis Method,” *Proceedings of 4th International Conference on Engineering of Complex Computer Systems (ICECCS’98)*, Monterey, CA, August 1998, pp. 68-78.

[Keeney76] Keeney, R.L., Raiffa, H., *Decisions with Multiple Objectives: Preference and Value Tradeoffs*, John Wiley & Sons, New York, 1976.

[Keeney92] Keeney, R.L., *Value-Focused Thinking: A Path to Creative Decisionmaking*, Harvard University Press, Cambridge, MA, 1992.

[Knight85] Knight, J.C., Leveson, N.G., St. Jean, J.D., “A Large Scale Experiment in N-version Programming,” *Fifteenth Annual International Symposium on Fault-Tolerant Computing*, Ann Arbor, MI, USA, June 1985, pp. 135-139.

[Knight2000] Knight, J.C., Sullivan, K.J., “On the Definition of Survivability,” University of Virginia, Department of Computer Science, Technical Report CS-TR-33-00, 2000.

[Knight2003] Knight, J.C., Strunk, E.A., Sullivan, K.J., “Towards a Rigorous Definition of Information System Survivability,” *DISCEX 2003*, Washington DC, April 2003.

[Koopman99] Koopman, P., DeVale, J., “Comparing the Robustness of POSIX Operating Systems,” *Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, Madison, WI, USA, June 1999, pp. 30-37.

[Kopetz97] Kopetz, H., *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers, Boston, 1997.

[Lamport82] Lamport, L., Shostak, R., and Pease, M., “The Byzantine Generals Problem”, *ACM Transactions on Programming Languages and Systems 4*, No. 3, July 1982, pp. 382-401.

[Laprie87] Laprie, J., Arlat, J., et al., “Hardware- and Software-Fault Tolerance: Definition and Analysis of Architectural Solutions,” *Seventeenth International Symposium on Fault-Tolerant Computing*, Pittsburgh, PA, July 1987, pp. 116-121.

[Latronico2001] Latronico, B., Martin, C. and Koopman, P., “Analyzing Dependability of Embedded Systems from the User Perspective,” *Workshop on Reliability in Embedded Systems (in conjunction with Symposium on Reliable Distributed Systems/SRDS-2001)*, New Orleans, LA, October 2001.

[Losq77] Losq, J., “Effects of Failures on Gracefully Degradable Systems,” *7th Annual International Conference on Fault-Tolerant Computing*, Los Angeles, CA, USA, June 1977, pp. 29-34.

[Lyu95] Lyu, M., ed., *Software Fault Tolerance*, John Wiley & Sons, Chichester, 1995.

[Medvidovic97] Medvidovic, N., Taylor, R.N., “A Framework for Classifying and Comparing Architectural Description Languages,” *Software Engineering Notes 22*, No. 6, November 1997, pp. 60-76.

[Meyer78] Meyer, J.F., “On Evaluating the Performability of Degradable Computing Systems,” *The Eighth Annual International Conference on Fault-Tolerant Computing (FTCS-8)*, Toulouse, France, June 1978, pp. 44-49.

[Meyer93] Meyer, J.F., Sanders, W.H., “Specification and Construction of Performability Models,” *Proceedings of the Second International Workshop on Performability Modeling of Computer and Communication Systems*, Mont Saint-Michel, France, June 1993.

[Mittal98] Mittal, A., Manimaran, G., Murthy, C.S.R., “Integrated Dynamic Scheduling of Hard and QoS Degradable Real-Time Tasks in Multiprocessor Systems,” *Proceedings of the Fifth International Conference on Real-Time Computing Systems and Applications*, Hiroshima, Japan, October 1998, pp. 127-136.

[Mobot2003] *Carnegie Mellon School of Computer Science 9th Annual Mobot Races*, <http://www-2.cs.cmu.edu/~mobot>, 2003.

[Nace2000] Nace, W., Koopman, P., “A Product Family Approach to Graceful Degradation,” *International Workshop on Distributed and Parallel Embedded Systems (DIPES)*, October 2000, pp. 131-140.

[Nace2002] Nace, W., “Graceful Degradation via System-wide Customization for Distributed Embedded Systems,” Ph.D. dissertation, Dept. of Electrical And Computer Engineering, Carnegie Mellon University, May 2002.

[Ng77] Ng, Y.-W., Avizienis, A., “A Reliability Model for Gracefully Degrading and Repairable Fault-Tolerant Systems,” *7th Annual International Conference on Fault-Tolerant Computing*, Los Angeles, CA, USA, June 1977, pp. 22-28.

[Patton93] Patton, R. J., Chen, J., “Advances in Fault Diagnosis Using Analytical Redundancy,” *IEE Colloquium on Plant Optimisation for Profit (Integrated Operations Management and Control)*, London, UK, January 1993, pp. 6/1 - 6/12.

[Parallax2003] *Basic Stamp 2 Microcontroller*, Parallax, Inc., <http://www.parallax.com>, 2003.

[Randell75] Randell, B., “System Structure for Software Fault Tolerance,” *IEEE Transactions on Software Engineering*, vol. SE-1, No. 2, June 1975, pp. 220-232.

[Ramanathan97] Ramanathan, P., “Graceful Degradation in Real-Time Control Applications Using (m, k)-firm Guarantee,” *27th Annual international Conferences on Fault-Tolerant Computing*, Seattle, WA, USA, June 1997, pp. 132-141.

[Rasmussen2001] Rasmussen, R., “Goal-Based Fault Tolerance for Space Systems using the Mission Data System,” *2001 IEEE Aerospace Conference*, Vol. 5, March 2001, Big Sky, MT, pp. 2401-2405.

[Ravindran97] Ravindran, B., Welch, L.R., Kelling, C., “Building Distributed Scalable Dependable Real-Time Systems,” *Proceedings International Conference and Workshop on Engineering of Computer-Based Systems*, Monterey, CA, USA, March 1997, pp. 452-459.

[Rennels84] Rennels, D., “Fault-Tolerant Computing - Concepts and Examples”, *IEEE Transactions on Computers C-33*, No. 12, December 1984, pp. 1116-1129.

[Rostamzadeh95] Rostamzadeh, B., Lonn, H., et al., “DACAPO: A Distributed Computer Architecture for Safety-Critical Control Applications,” *Proceedings of the Intelligent Vehicles ’95 Symposium*, Detroit, MI, USA, September 1995, pp. 376-381.

[Rushby99] Rushby, J., “Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance,” NASA Contractor Report CR-1999-209347, June 1999.

[Rushby2001] Rushby, John, “A Comparison of Bus Architectures for Safety-Critical Embedded Systems,” Computer Science Laboratory, SRI International Technical Report, September, 2001.

[Shaw96] Shaw, M., Garlan, D., *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, Upper Saddle River, New Jersey, 1996.

[Shaw97] Shaw, M., Clements, P., “A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems,” *Proceedings of the 21st International Computer Software and Applications Conference (COMPSAC ‘97)*, August 1997, pp. 6-13.

[Strakosch98] Strakosch, G.R., ed., *The Vertical Transportation Handbook*, Third Edition, John Wiley & Sons, Inc., New York, 1998.

[Verissimo2001] Verissimo, P., Rodrigues, L., *Distributed Systems for System Architects*, Kluwer Academic Publishers, Boston, 2001.

[Wang99] Wang, W.-L., Wu, Y., Chen, M.-H., “An Architecture-Based Software Reliability Model,” *Proceedings 1999 Pacific Rim International Symposium on Dependable Computing*, Hong Kong, December 1999, pp. 143-150.

[Weber89] Weber, D.G., “Formal Specification of Fault-Tolerance and its Relation to Computer Security,” *Proceedings of Fifth International Workshop on Software Specification and Design*, Pittsburgh, PA, USA, May 1989, pp. 273-277.

[WOSS2002] *First Workshop on Self-Healing Systems (WOSS 2002)*, ACM Press, Charleston, South Carolina, November 2002.

Thesis Publications:

[Shelton2001] Shelton, C., Koopman, P., “Developing a Software Architecture for Graceful Degradation in an Elevator Control System,” *Workshop on Reliability in Embedded Systems (in conjunction with Symposium on Reliable Distributed Systems/SRDS-2001)*, New Orleans, LA, October 2001.

[Shelton2002] Shelton, C., Koopman, P., “Using Architectural Properties to Model and Measure System-Wide Graceful Degradation,” *Workshop on Architecting Dependable Systems sponsored by the International Conference on Software Engineering (ICSE2002)*, Orlando, FL, May 2002.

[Shelton 2003] Shelton, C., Koopman, P., Nace, W., “A Framework for Scalable Analysis and Design of System-wide Graceful Degradation in Distributed Embedded Systems,” *To appear in the Eighth IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003)*, Guadalajara, Mexico, January 2003.

Appendix A - Acme Formal Specification of Our System Model

This is the formal specification of the architectural style for the software system component view written in the Acme ADL:

```
Family Graceful_Deg_Family = {
    invariant Forall comp in self.Components |
        Forall p in comp.Ports |
            Forall conn in self.Connectors |
                Forall r in conn.Roles |
                    (attached(r, p) == true) -> ((declaresType(r,
SystemVarSinkRoleT) AND declaresType(p, DataInputPortT)) OR
(declaresType(r, SystemVarSourceRoleT) AND declaresType(p,
DataOutputPortT)));

    Component Type SensorT = {
        Property SystemVarOutput : string;

        Port SensorOutput : DataOutputPortT =  new DataOutputPortT
extended with {
            Property Required : boolean = true;
            Property SystemVarOutput : string;
        };

        Property SensorDescription : string;

        invariant Forall p in self.ports |
            Exists t in {DataOutputPortT} |
                declaresType(p, t);

        invariant Forall p : port in self.Ports |
            !declaresType(p, DataInputPortT);

        invariant size({Select p : port in self.Ports |
            declaresType(p, DataOutputPortT) }) >= 1;
    }

    Component Type ActuatorT = {
        Property SystemVarInput : string;

        Port ActuatorInput : DataInputPortT =  new DataInputPortT
extended with {
            Property SystemVarInput : string;
            Property Dependency : Enum {Strong, Weak, Optional } =
Strong;
        };

        Property ActuatorDescription : string;

        invariant Forall p in self.ports |
            Exists t in {DataInputPortT} |
                declaresType(p, t);

        invariant Forall p : port in self.Ports |
            !declaresType(p, DataOutputPortT);

        invariant size({Select p : port in self.Ports |
            declaresType(p, DataInputPortT) }) >= 1;
    }

    Component Type SoftwareComponentT = {
        Property ComponentDescription : string;

        invariant size({Select p : port in self.Ports |
            declaresType(p, DataOutputPortT) }) >= 1;

        invariant Forall p in self.ports |
            Exists t in {DataInputPortT, DataOutputPortT} |
                declaresType(p, t)  <vis-ports : boolean = true;>;
    }

    Port Type DataInputPortT = {
        Property SystemVarInput : string;

        Property Dependency : Enum {Strong, Weak, Optional };
    }
    Port Type DataOutputPortT = {
        Property SystemVarOutput : string;

        Property Required : boolean;
    }

    Connector Type SystemVariableConnT = {
        Property SystemVarData : string;

        invariant size({Select r : role in self.Roles |
            declaresType(r, SystemVarSourceRoleT) }) >= 1;

        invariant size({Select r : role in self.Roles |
            declaresType(r, SystemVarSinkRoleT) }) >= 1;

        invariant Forall r in self.roles |
          Exists t in {SystemVarSinkRoleT, SystemVarSourceRoleT} |
                declaresType(r, t);
    }

    Role Type SystemVarSourceRoleT = {
        Property SystemVarData : string;
    }
    Role Type SystemVarSinkRoleT = {
        Property SystemVarData : string;
    }
}
```

Appendix B - Utility Specification for the Automobile Navigation System

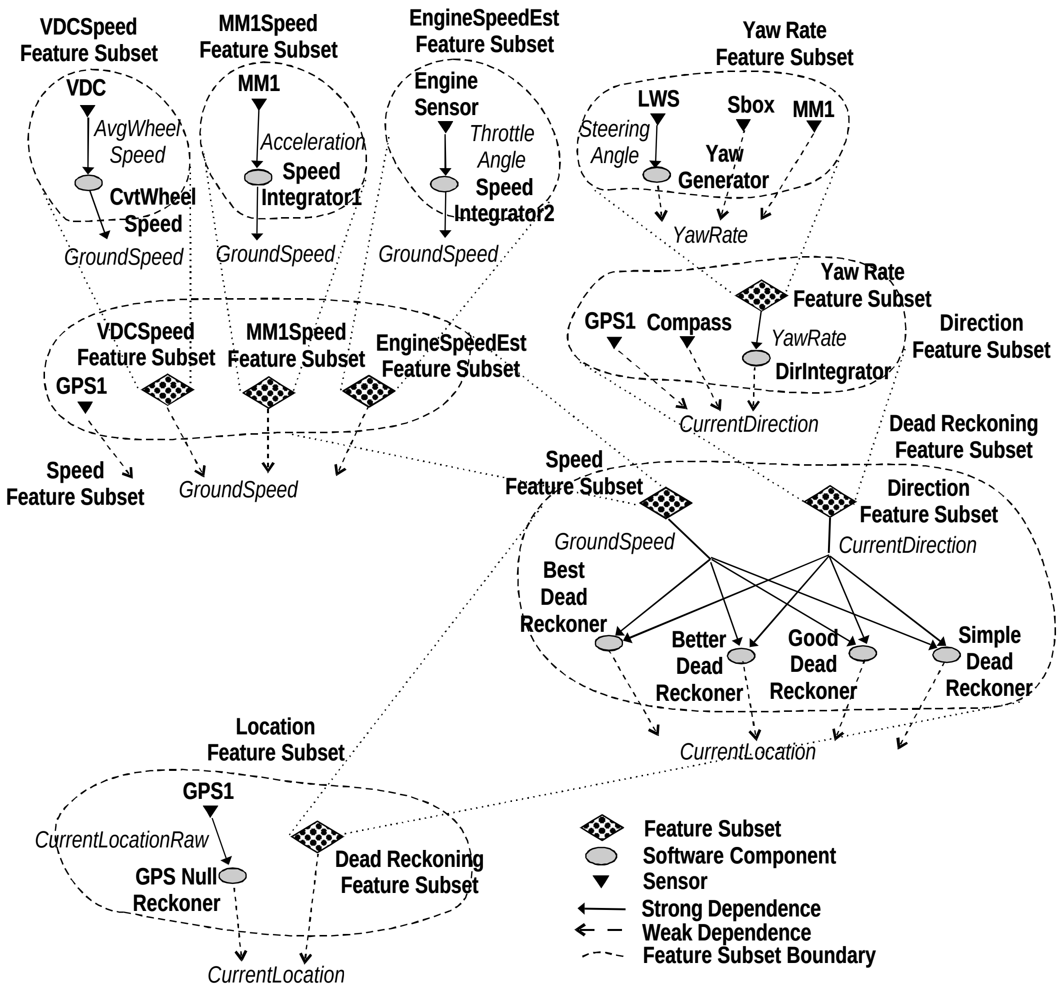

Figure B.1. Expansion of the Location Feature Subset.

This appendix contains all of the feature subset and utility definitions for the automobile navigation system presented in Chapter 4. The numbers chosen in the utility specification are an arbitrary representation of how a system designer with knowledge of the system could assign utility values based on the functionality of each subsystem. We start with the “low level” feature subsets that contribute to providing location data in the navigation system. Figure B.1 shows the hierarchical definitions of these feature subsets, and Table B.1 shows the utility functions generated for these feature subsets.

Table B.1. Utility Specification for the Location and Related Feature Subsets.

Feature Subset	Configuration	Utility Function
VDC Speed	{VDC Sensor, CvtWheelSpeed}	$U_{VDC} = 1$
	All other configurations	$U_{VDC} = 0$
MM1 Speed	{MM1 Sensor, Speed Integrator1}	$U_{MM1} = 1$
	All other configurations	$U_{MM1} = 0$
Engine Speed Est	{Engine Speed Sensor, Speed Integrator2}	$U_{Engine} = 1$
	All other configurations	$U_{Engine} = 0$
Speed	Any of the 15 combinations of {GPS1 sensor (u_{gps}), VDC Speed Feature, MM1 Speed Feature, Engine Speed Est Feature} in which at least one is working	$U_{Speed} = Max(1*u_{gps1}, 0.75*U_{VDC}, 0.85*U_{MM1}, 0.6*U_{Engine})$
	All components failed	$U_{Speed} = 0$
Yaw Rate	Any configuration in which either the Sbox sensor (u_{sbox}) or MM1 sensor (u_{mm1}) is working	$U_{Yaw} = Max(1*u_{sbox}, 0.8*u_{mm1})$
	{LWS sensor, Yaw Generator}	$U_{Yaw} = 0.6$
	All other configurations	$U_{Yaw} = 0$
Direction	Any configuration in which either the GPS1 sensor or Compass sensor ($u_{compass}$) is working	$U_{Dir} = Max(1*u_{gps1}, 0.9*u_{compass})$
	{Yaw Rate Feature, DirIntegrator}	$U_{Dir} = 0.2 + 0.6*U_{Yaw}$
	All other configurations	$U_{Dir} = 0$
Dead Reckoning	Any configuration with at least one dead reckoning software component (Simple Dead Reckoner (u_{sdr}), Good Dead Reckoner (u_{gdr}), Better Dead Reckoner (u_{btdr}), Best Dead Reckoner (u_{bedr})) and both Speed and Direction Features	$U_{DR} = Max(0.2*u_{sdr}, 0.4*u_{gdr}, 0.6*u_{btdr}, 0.7*u_{bedr}) + 0.2*U_{Dir} + 0.1*U_{Speed}$
	All other configurations	$U_{DR} = 0$
Location	{GPS1 sensor, GPS Null Reckoner (u_{gpsnr})} or {GPS1 sensor, GPS Null Reckoner, Dead Reckoning Feature}	$U_{Location} = 1$
	{Dead Reckoning Feature, GPS Null Reckoner} or {Dead Reckoning Feature, GPS1 Sensor}	$U_{Location} = 0.8*U_{DR}$
	All other configurations	$U_{Location} = 0$

Figure B.2 shows the feature subsets that comprise the Map and two of the eight Display feature subsets. Each Display feature subset requires location and map data to provide functionality. Figure B.3 gives the feature subset diagrams for the other six Display feature subsets.

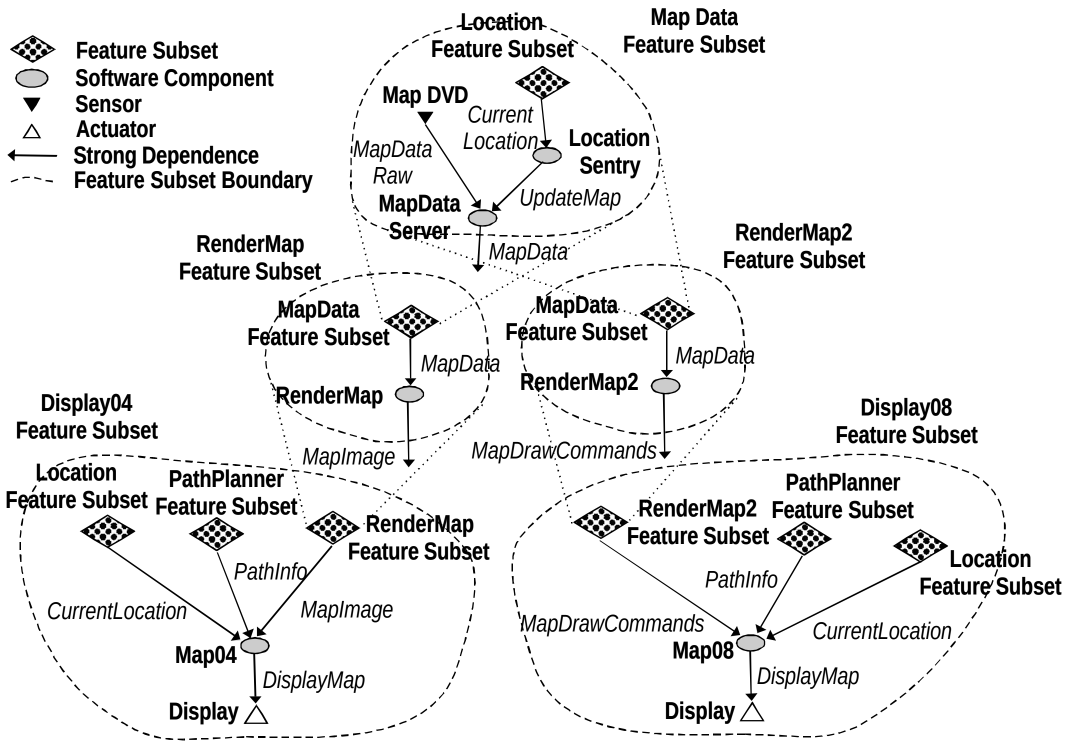

Figure B.2. Partial Expansion of Display Feature Subsets.

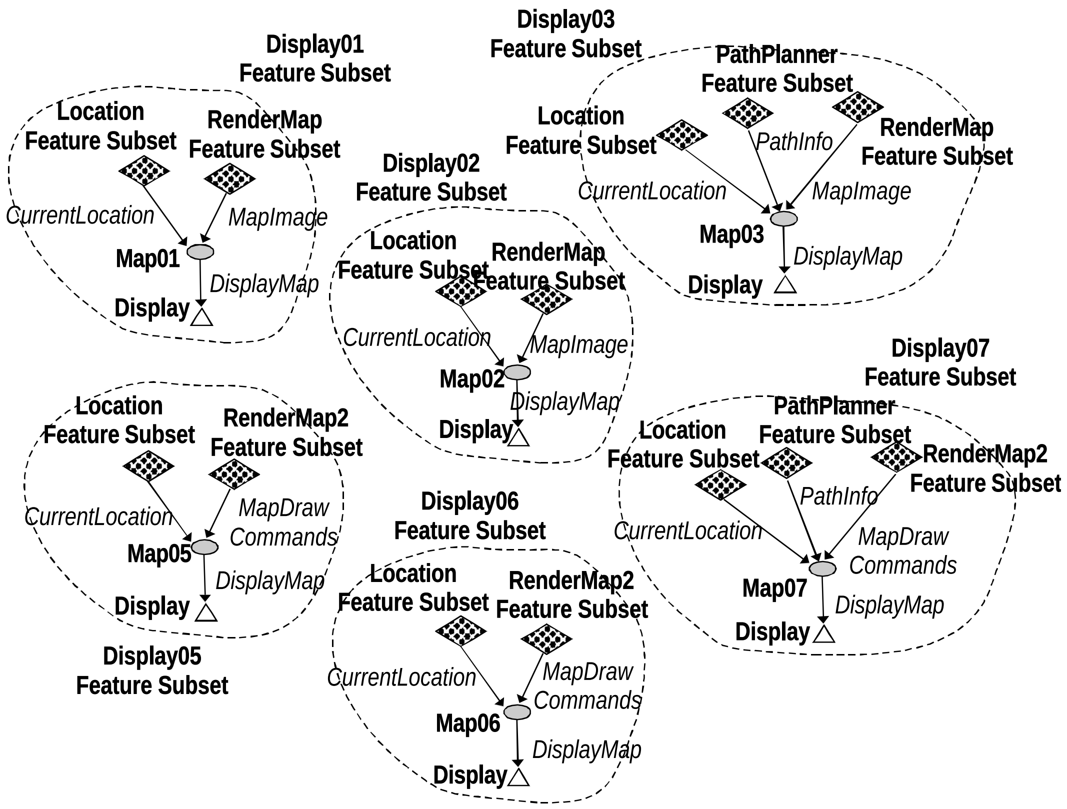

Figure B.3. Diagrams of the Remaining Display Feature Subsets.

Table B.2. Utility Specification for the Map and Display Feature Subsets.

Feature Subset	Configuration	Utility Function
Map Data	{MapDVD Sensor, Location Feature Subset, Location Sentry, MapDataServer}	$U_{MapData} = 0.7 + 0.3*U_{Location}$
	All other configurations	$U_{MapData} = 0$
RenderMap	{Map Data Feature Subset, RenderMap}	$U_{RMap} = 0.4 + 0.6*U_{MapData}$
	All other configurations	$U_{RMap} = 0$
RenderMap2	{Map Data Feature Subset, RenderMap2}	$U_{RMap2} = 0.4 + 0.6*U_{MapData}$
	All other configurations	$U_{RMap2} = 0$
Display01	{Location Feature Subset, RenderMap Feature Subset, Display Actuator, Map01}	$U_{D1} = 0.5 + 0.2*U_{Location} + 0.3*U_{RMap}$
	All other configurations	$U_{D1} = 0$
Display02	{Location Feature Subset, RenderMap Feature Subset, Display Actuator, Map02}	$U_{D2} = 0.5 + 0.2*U_{Location} + 0.3*U_{RMap}$
	All other configurations	$U_{D2} = 0$
Display 03	{Location Feature Subset, RenderMap Feature Subset, Path Planner Feature Subset, Display Actuator, Map03}	$U_{D3} = 0.5 + 0.2*U_{Location} + 0.2*U_{RMap} + 0.1*U_{Path}$
	All other configurations	$U_{D3} = 0$
Display04	{Location Feature Subset, RenderMap Feature Subset, Path Planner Feature Subset, Display Actuator, Map04}	$U_{D4} = 0.5 + 0.2*U_{Location} + 0.2*U_{RMap} + 0.1*U_{Path}$
	All other configurations	$U_{D4} = 0$
Display05	{Location Feature Subset, RenderMap2 Feature Subset, Display Actuator, Map05}	$U_{D5} = 0.5 + 0.2*U_{Location} + 0.3*U_{RMap2}$
	All other configurations	$U_{D5} = 0$
Display06	{Location Feature Subset, RenderMap2 Feature Subset, Display Actuator, Map06}	$U_{D6} = 0.5 + 0.2*U_{Location} + 0.3*U_{RMap2}$
	All other configurations	$U_{D6} = 0$
Display07	{Location Feature Subset, RenderMap2 Feature Subset, Path Planner Feature Subset, Display Actuator, Map07}	$U_{D7} = 0.5 + 0.2*U_{Location} + 0.2*U_{RMap2} + 0.1*U_{Path}$
	All other configurations	$U_{D7} = 0$
Display08	{Location Feature Subset, RenderMap2 Feature Subset, Path Planner Feature Subset, Display Actuator, Map08}	$U_{D8} = 0.5 + 0.2*U_{Location} + 0.2*U_{RMap2} + 0.1*U_{Path}$
	All other configurations	$U_{D8} = 0$

Table B.2 shows the utility specifications for the Display and Map feature subsets. These feature subsets cannot tolerate component failures, but the feature

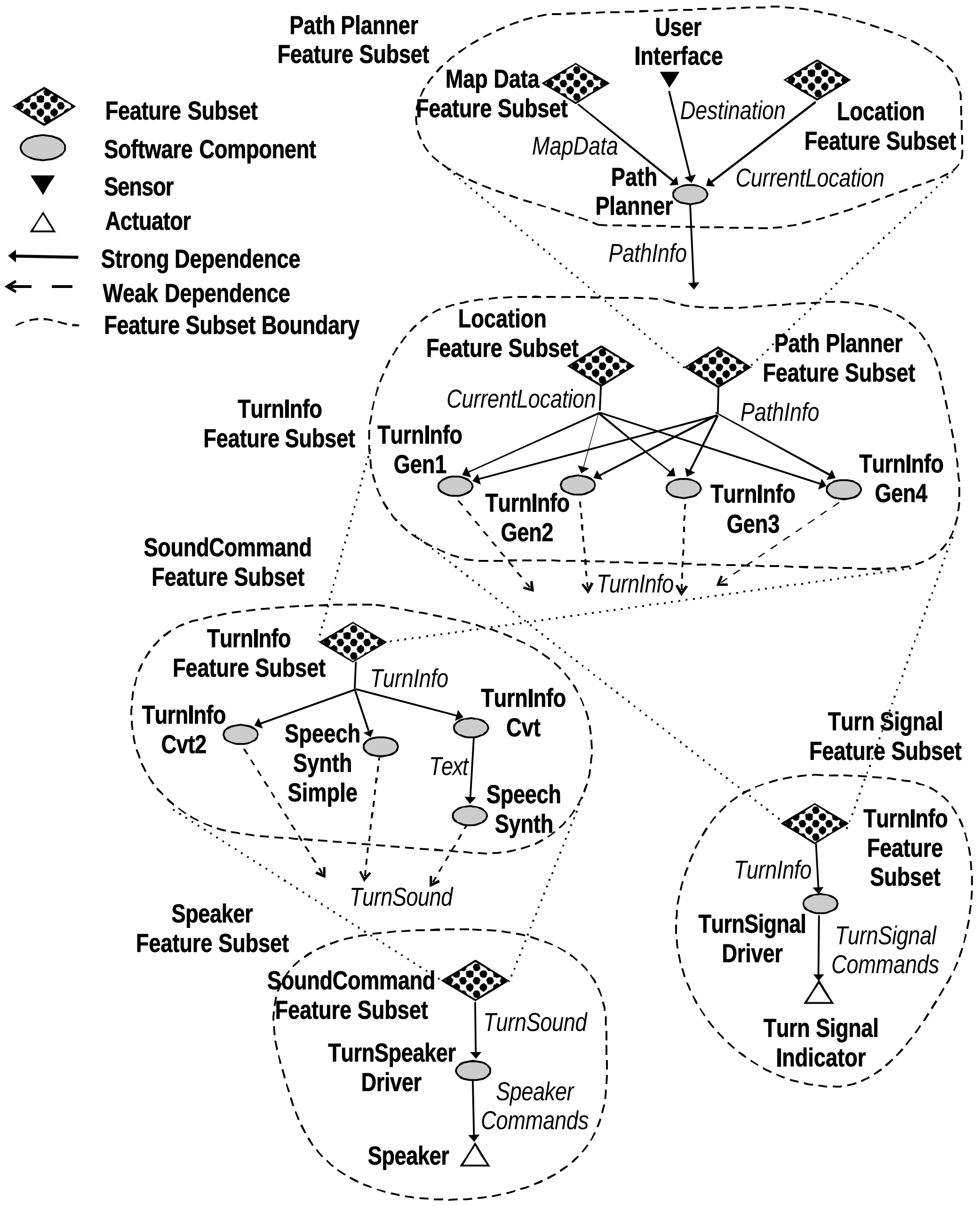

Figure B.4. Expansion of Turn Signal and Speaker Feature Subsets.

subsets themselves are redundant backups for each other to provide the Display actuator's functionality.

Figure B.4 diagrams the Path Planner, Turn Signal, Speaker, and related feature subsets, and Table B.3 gives their utility specification in our model. These feature subsets represent the alternative functionality available in the navigation system in

Table B.3. Utility Specification for the Path Planner, Speaker, and Turn Signal Feature Subsets.

Feature Subset	Configuration	Utility Function
Path Planner	{User Interface Sensor, Map Data Feature Subset, Location Feature Subset, Path Planner}	$U_{Path} = 0.5 + 0.2*U_{MapData} + 0.3*U_{Location}$
	All other configurations	$U_{Path} = 0$
Turn Info	Any configuration with at least one turn info software component (TurnInfo1 (u_{ti1}), TurnInfo2 (u_{ti2}), TurnInfo3 (u_{ti3}), TurnInfo4 (u_{ti4})) and both Location and Path Planner Features	$U_{TurnInfo} = 0.6 * Max(1*u_{ti1}, 0.8*u_{ti2}, 0.6*u_{ti3}, 0.4*u_{ti4}) + 0.2 * U_{Path} + 0.2 * U_{Location}$
	All other configurations	$U_{TurnInfo} = 0$
Sound Command	Any configuration in which the Turn Info Feature Subset, TurnInfoCvt, and SpeechSynth are working	$U_{Sound} = 0.6 * 0.4*U_{TurnInfo}$
	{Turn Info Feature Subset, TurnInfoCvt2, SpeechSynthSimple}	$U_{Sound} = 0.5 * 0.4*U_{TurnInfo}$
	{Turn Info Feature Subset, SpeechSynthSimple}	$U_{Sound} = 0.5 * 0.4*U_{TurnInfo}$
	{Turn Info Feature Subset, TurnInfoCvt2}	$U_{Sound} = 0.3 * 0.4*U_{TurnInfo}$
	All other configurations	$U_{Sound} = 0$
Speaker	{Sound Command Feature Subset, Turn Speaker Driver, Speaker Actuator}	$U_{Speaker} = 0.3 + 0.7*U_{Sound}$
	All other configurations	$U_{Speaker} = 0$
Turn Signal	{Turn Info Feature Subset, Turn Signal Driver, Turn Signal Indicator Actuator}	$U_{TurnSignal} = 0.6 + 0.4*U_{TurnInfo}$
	All other configurations	$U_{TurnSignal} = 0$

the event that the Display fails. Navigation information can still be communicated to the driver through the Speaker and Turn Signal actuators.

With a complete specification for the utility of all feature subset configurations, we can generate a system utility function using the utility values of the feature subsets encapsulated in the system functional capabilities. Figure B.5 describes these capabilities and the feature subsets they contain, and Table B.4 describes their utility functions along with the system utility function in terms of capability utilities. For any system configuration, the utility value can be generated by recursively evaluating the utility values of the functional capabilities and their feature subsets based on which components are present in the configuration.

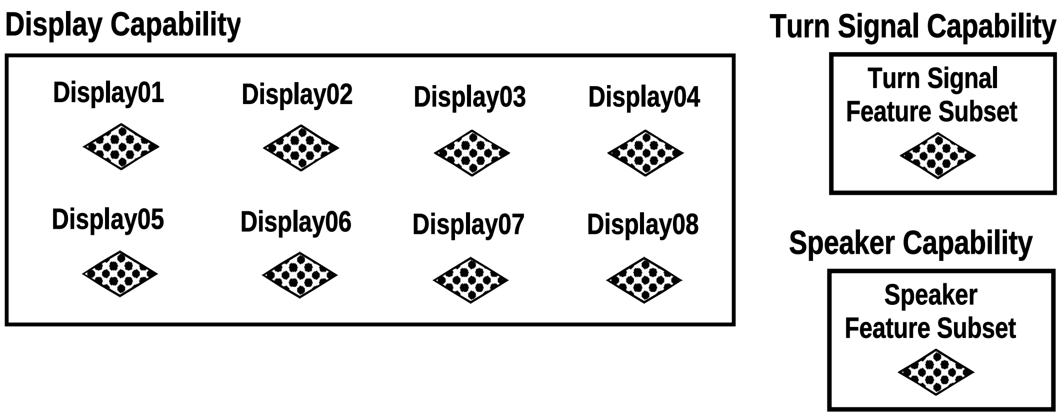

Figure B.5. System-Level Functional Capabilities and their Feature Subsets.

Table B.4. Utility Specification for System Functional Capabilities.

System Capability	Configuration	Utility Function
Speaker	{Speaker Feature Subset}	$U_{SpeakerCapability} = U_{Speaker}$
	Speaker Feature Subset failed	$U_{Speaker} = 0$
Turn Signal	{Turn Signal Feature Subset}	$U_{TurnSignalCapability} = U_{TurnSignal}$
	Turn Signal Feature Subset failed	$U_{TurnSignalCapability} = 0$
Display	Any of the 63 configurations in which there is at least of the eight Display Feature Subsets (Display01 - Display08)	$U_{DisplayCapability} = Max(1*U_{D4}, 0.9*U_{D8}, 0.8*U_{D3}, 0.7*U_{D7}, 0.6*U_{D2}, 0.5*U_{D6}, 0.4*U_{D1}, 0.3*U_{D5})$
	All Display Feature Subsets failed	$U_{DisplayCapability} = 0$
System Utility	All Capability Configurations	$U_{System} = 0.6*U_{DisplayCapability} + 0.2*U_{SpeakerCapability} + 0.1*U_{TurnSignalCapability}$

Appendix C - Interface Specification for the Elevator System Components

This appendix contains the interface specification of the elevator system software components, taken from the original elevator requirements document. For clarity, we first provide the elevator sensor and actuator message descriptions from Chapter 5. In the following notation, the values within the "[ ]" brackets represent the standard replication of an array of sensors or actuators, and the values within the "( )" parentheses represent the values the sensor or actuator can output. For example, the Hall call message type maps to an array of sensors for the up and down buttons on each floor outside the elevator that is *f* (the number of floors the elevator services) by *d* (the direction of the button; Up or Down) wide, and each button sensor can either have a value v of True (pressed) or False (not pressed). Unless otherwise noted, "f" represents the number of floors the elevator services, "d" represents a variable that indicates a direction of either Up or Down, "j" is a variable that is a value of either Left or Right (for the left and right elevator doors), and "v" denotes a value that can be either True or False.

The sensor message types available in the system include:

- **AtFloor[f](v):** Output of AtFloor sensors that sense when the car is near a floor.
- **CarCall[f](v):** Output of car call button sensors located in the car.
- **CarLevelPosition(x):** Output of car position sensor that tracks where the car is in the hoistway. x = {distance value from bottom of hoistway in millimeters}
- **CarWeight(w):** Output of car weight sensor that measures the aggregate weight of all passengers in the car. w = { weight in car in pounds }

- **DoorClosed[j](v):** Output of door closed sensors that will be True when the door is fully closed.
- **DoorOpen[j](v):** Output of door open sensors that will be True when the door is fully open.
- **DoorReversal[j](v):** Output of door reversal sensors that will be True when door senses an obstruction in the doorway.
- **HallCall[f,d](v):** Output of hall call button sensors that are located in hallway outside the elevator on each floor. Note that there are a total of 2f - 2 rather than 2f hall call buttons since the top floor only has a down button and the bottom floor only has an up button.
- **HoistwayLimit[d](v):** Output of safety limit sensors in the hoistway that will be True when the car has overrun either the top or bottom hoistway limits.
- **DriveSpeed(s,d):** Output of the main drive speed sensor. s = {speed value}, d = {Up, Down, Stop}

The actuator command messages available in the system are:

- **DesiredFloor(f, d):** Command from the elevator dispatcher algorithm indicating the next floor destination. d = {Up, Down, Stop} (This is not an actuator input, but rather an internal variable in the control system sent from the dispatcher to the drive controller)
- **DesiredDwell(n):** Command from the elevator dispatcher algorithm to the door controllers indicating how long the doors should remain open when stopped on a floor. n = { Integer dwell time in milliseconds } (This is also

not an actuator input, but an internal control system variable that allows the dispatcher to affect the operation of the door motors)

- **DoorMotor[j](m):** Door motor commands for each door. m = {Open, Close, Stop}
- **Drive(s, d):** Commands for 2-speed main elevator drive. s = {Fast, Slow, Stop}, d = {Up, Down, Stop}
- **CarLantern[d](v):** Commands to control the car lantern lights; Up/Down lights on the car doorframe used by passengers to determine the elevator's current traveling direction.
- **CarLight[f](v):** Commands to control the car call button lights inside the car call buttons to indicate when a floor has been selected.
- **CarPositionIndicator(f):** Commands for position indicator light in the car that tells users what floor the car is approaching.
- **HallLight[f,d](v):** Commands for hall call button lights inside the hall call buttons to indicate when passengers want the elevator on a certain floor.
- **EmergencyBrake(v):** Emergency stop brake that should be activated whenever the system state becomes unsafe and the elevator must be shut down to prevent a catastrophic failure.

Software Components in the Elevator:

Safety Monitor - Monitors system sensors and controllers to ensure safe operation and trigger emergency shut down when necessary.

Inputs: *AtFloor[1..f], DoorClosed[Left, Right], DoorReversal[Left, Right], DriveSpeed, HoistwayLimit[Up, Down], Drive, DoorMotor[Left, Right]*

Output: *EmergencyBrake*

Drive Controller - Controls drive motor to move elevator in hoistway.

Inputs: *AtFloor[1..f], CarLevelPosition, DoorClosed[Left, Right], DoorMotor[Left, Right], DriveSpeed, Hoistwaylimit[Up, Down], DesiredFloor, EmergencyBrake*

Output: *Drive*

Door Controller[j] - Controls the door motors to operate the elevator doors. One door controller class instantiated as two software door controller objects that each control one door.

Inputs: *AtFloor[1..f], DoorClosed[j]. DoorOpen[j], DoorReversal[j], DesiredFloor, DesiredDwell, DriveSpeed, Drive, CarCall[1..f], HallCall[1..f, (Up, Down)]*

Output: *DoorMotor[j]*

Car Call Controller[f] - Monitors car call button sensors to provide car button information to the system and light the car button light. One car button controller class is instantiated f times for each button.

Inputs:*CarCall[f] (from sensor), AtFloor[f], DoorClosed[Left, Right]*

Outputs: *CarCall[f] (to the network), CarLight[f]*

Hall Call Controller[f, d] - Monitors hall call button sensors to provide hall button information to the system and light the hall button light. One hall button controller class is instantiated 2f - 2 times for each button.

Inputs: *HallCall[f, d] (from sensor), AtFloor[f], DoorClosed[Left, Right], DesiredFloor*

Outputs: *HallCall[f, d] (to the network), HallLight[f, d]*

Dispatcher - Determines elevator's next destination based on passenger requests

Inputs: *CarCall[1..f], HallCall[1..f, (Up, Down)], CarWeight, DoorClosed[Left, Right], DriveSpeed, AtFloor[1..f]*

Outputs: *DesiredFloor, DesiredDwell*

Lantern Controller[d] - Operate passenger feedback lights that indicate elevator's travelling direction when the elevator is stopped on a floor. One controller class is instantiated as two software lantern controller objects that control each light.

Inputs: *DesiredFloor, AtFloor[1..f], DoorClosed[Left, Right]*

Output: *CarLantern[d]*

Car Position Indicator Controller - Operates passenger feedback lights that indicate next floor the elevator will reach as it travels in the hoistway.

Inputs: *AtFloor[1..f], DesiredFloor, DriveSpeed, CarLevelPosition*

Output: *CarPositionIndicator*

Virtual AtFloor Controller[f] - Software controller that outputs AtFloor messages when an Atfloor sensor fails. One virtual atfloor controller class is instantiated f times for each AtFloor sensor.

Inputs: *AtFloor[f] (from sensor to detect failure), DriveSpeed, CarLevelPosition*

Outputs: *AtFloor[f]*

Appendix D - Utility Specification for the Elevator System

This appendix lists the utility functions we specified for the elevator system in Chapter 5.

Table D.1. Utility Specification for the Safety, Door and Drive Feature Subsets.

Feature Subset	Configuration	Utility Function
Hoistway Limit Sensors	Any configuration in which at least one of the two sensors: Hoistway Limit Up Sensor ($u_{hoistwayup}$), Hoistway Limit Down Sensor ($u_{hoistwaydown}$) is working	$U_{Hoistway} = 0.5*u_{hoistwayup} + 0.5*u_{hoistwaydown}$
	Both Hoistway Limit Sensors Failed	$U_{Hoistway} = 0$
Door Closed Sensors	Any configuration in which at least one of the two sensors: Left Door Closed Sensor (u_{ldc}), Right Door Closed Sensor (u_{rdc}) is working	$U_{DoorClosed} = 0.5*u_{ldc} + 0.5*u_{rdc}$
	Both Door Closed Sensors Failed	$U_{DoorClosed} = 0$
Door Reversal Sensors	Any configuration in which at least one of the two sensors: Left Door Reversal Sensor (u_{ldr}), Right Door Reversal Sensor (u_{rdr}) is working	$U_{DoorReversal} = 0.5*u_{ldr} + 0.5*u_{rdr}$
	Both Door Reversal Sensors Failed	$U_{DoorReversal} = 0$
Safety Monitor	{Safety Monitor Controller, Emergency Brake Actuator, Drive Speed Sensor, Drive Control Feature Subset, Door Control Feature Subset, Hoistway Limit Sensors Feature Subset, Door Reversal Sensors Feature Subset, Door Closed Sensors Feature Subset, AtFloor Sensors Feature Subset}	$U_{Safety} = 1$
	All other configurations	$U_{Safety} = 0$
Drive Control	Any configuration with all of these components: {Drive Controller, Drive Motor, Drive Speed Sensor, Safety Monitor Feature Subset, Hoistway Limit Sensors Feature Subset, Door Closed Sensors Feature Subset, AtFloor Sensors Feature Subset} and any combination of: Car Positon Sensor (u_{cps}), Desired Floor Feature Subset ($U_{DesiredFloor}$), Door Control Feature Subset ($U_{DoorControl}$)	$U_{Drive} = 0.1 + 0.2*u_{cps} + 0.65*U_{DesiredFloor} + 0.05*U_{DoorControl}$
	All other configurations	$U_{Drive} = 0$
Left/Right Door Control	Any configuration with all of these components: {Left/Right Door Controller, L/R Door Closed Sensor, L/R Door Open Sensor, L/R Door Reversal Sensor, L/R Door Motor, Drive Speed Sensor, AtFloor Sensors Feature Subset} and any combination of: Drive Control Feature Subset, DesiredFloor Feature Subset, Car Call Buttons Feature Subset ($U_{CarCall}$), Hall Call Buttons Feature Subset ($U_{HallCall}$)	$U_{LeftDoor} = 0.6 + 0.1*U_{CarCall} + 0.2*U_{HallCall} + 0.05*U_{DesiredFloor} + 0.05*U_{Drive}$ $U_{RightDoor} = 0.6 + 0.1*U_{CarCall} + 0.2*U_{HallCall} + 0.05*U_{DesiredFloor} + 0.05*U_{Drive}$
	All other configurations	$U_{LeftDoor} = 0$; $U_{RightDoor} = 0$

Table D.2. Utility Specification for the Desired Floor, Car Call, and Hall Call Feature Subsets.

Feature Subset	Configuration	Utility Function
Door Control	Any configuration in which at least one of the two Feature Subsets: Left Door Control, Right Door Control is working	$U_{DoorControl} = 0.5*U_{LeftDoor} + 0.5*U_{RightDoor}$
	Both Door Control Feature Subsets Failed	$U_{DoorControl} = 0$
Desired Floor	Any configuration with all of these components: {Dispatcher Controller, Drive Speed Sensor, AtFloor Sensors Feature Subset, Door Closed Sensors Feature Subset} and any combination of: Car Weight Sensor (u_{cws}), Hall Call Buttons Feature Subset, Car Call Buttons Feature Subset	$U_{DesiredFloor} = 0.09 + 0.01*u_{cws} + 0.3*U_{CarCall} + 0.6*U_{HallCall}$
	All other configurations	$U_{DesiredFloor} = 0$
Car Call Floor 1..f	Any configuration with all of these components: {Car Call Floor f Controller, Car Call Floor f Button Sensor} and any combination of: Car Call Floor f Button Light (u_{ccl_f}), AtFloor Floor f Sensor Feature Subset ($U_{AtFloor\_f}$), Door Closed Sensors Feature Subset	$U_{CarCall\_f} = 0.6 + 0.2*u_{ccl\_f} + 0.1*U_{AtFloor\_f} + 0.1*U_{DoorClosed}$
	All other configurations	$U_{CarCall\_f} = 0$
Car Call Buttons	Any configuration in which at least one of the Car Call Floor Feature Subsets is working	$U_{CarCall} = 0.4*U_{CarCall\_1} + 0.6*(U_{CarCall\_2} + U_{CarCall\_3} + ... + U_{CarCall\_f})/(f-1)$
	All Car Call Floor Feature Subsets Failed	$U_{CarCall} = 0$
Hall Call Up/Down Floor 1..f	Any configuration with all of these components: {Hall Call Up/Down Floor f Controller, Hall Call U/D Floor f Button Sensor} and any combination of: Hall Call U/D Floor f Button Light (u_{hcl_u/d_f}), AtFloor Floor f Sensor Feature Subset, Door Closed Sensors Feature Subset, Desired Floor Feature Subset	$U_{HallCall\_u/d\_f} = 0.6 + 0.2*u_{hcl\_u/d\_f} + 0.667*(U_{DesiredFloor} + U_{DoorClosed} + U_{AtFloor\_f})$
	All other configurations	$U_{HallCall\_u/d\_f} = 0$
Hall Call Up/Down Buttons	Any configuration in which at least one of the Hall Call Up/DownFloor Feature Subsets is working	$U_{HallCall\_up} = (U_{HallCall\_up\_1} + ... + U_{HallCall\_up\_f-1})/(f-1)$ $U_{HallCall\_down} = (U_{HallCall\_down\_2} + ... + U_{HallCall\_down\_f})/(f-1)$
	All Hall Call Up/Down Floor Feature Subsets Failed	$U_{HallCall\_up} = 0; U_{HallCall\_down} = 0$

Table D.3. Utility Specification for the AtFloor, Car Position Indicator, and Car Lantern Feature Subsets.

Feature Subset	Configuration	Utility Function
AtFloor Floor 1..f	Any configuration with the AtFloor Floor f Sensor working	$U_{AtFloor\_f} = 1$
	{Virtual AtFloor Floor f Controller, Drive Speed Sensor, Car Position Sensor}	$U_{AtFloor\_f} = 1$
	All other configurations	$U_{AtFloor\_f} = 0$
AtFloor Sensors	Any configuration in which at least one of the AtFloor Floor Feature Subsets is working	$U_{AtFloor} = 0.4*U_{AtFloor\_1} + 0.6*(U_{AtFloor\_2} + U_{AtFloor\_3} + ... U_{AtFloor\_f})/(f-1)$
	All AtFloor Feature Subsets Failed	$U_{AtFloor} = 0$
Lantern Control Up/Down	{Lantern Up/Down Controller, Car Lantern Up/Down Light, Door Closed Sensors Feature Subset, Desired Floor Feature Subset, AtFloor Sensors Feature Subset}	$U_{Lantern\_u/d} = 1$
	All other configurations	$U_{Lantern\_u/d} = 0$
Car Lantern	Any configuration in which at least one of the Lantern Control Feature Subsets is working	$U_{CarLantern} = 0.5*U_{Lantern\_u} + 0.5*U_{Lantern\_d}$
	Both Lantern Control Feature Subsets Failed	$U_{CarLantern} = 0$
Car Position Indicator	Any configuration with all of these components: {Car Position Indicator Controller, Car Position Inidcator Lights, AtFloor Sensors Feature Subset} and any combination of: Car Position Sensor, Drive Speed Sensor, Desired Floor Feature Subset	$U_{CarPosInd} = 0.7 + 0.3*(U_{DesiredFloor} + u_{cps} + u_{drivespeed})$

Table D.4. Utility Specification for the Elevator System.

Feature Subset	Configuration	Utility Function
System Utility	Any configuration with all of these Feature Subsets: {Safety Monitor, Drive Control, Door Control} and any combination of: Hall Call Buttons, Car Call Buttons, Car Lantern, Car Position Indicator	$U_{System} = 0.5*U_{Drive} + 0.2*U_{DoorControl} + 0.1*U_{HallCall} + 0.1*U_{CarCal}l + 0.05*U_{CarLantern} + 0.05*U_{CarPosInd}$
	All other configurations	$U_{System} = 0$

Appendix E - Data for the Elevator Configuration Experiments

These tables list the data for each configuration tested in the elevator case study (Chapter 5). The configuration utility values and the average passenger delivery times refer only to the gracefully degrading elevator system.

Table E.1. Elevator Experimental Data for Configurations 1 - 11

Config ID#	Failed Components	System Utility Value	Avg % Delivered in Original Elevator	Avg % Delivered in Gracefully Degrading Elevator	Avg Delivery Time for Two-Way (secs)	Avg Delivery Time for Down-Peak (secs)	Avg Delivery Time for Up-Peak (secs)
1	all hall call buttons, all car call buttons, car lantern up, down, car position indicator, dispatcher, car position sensor	0.196	0.00	100	897.95	6329.55	1109.97
2	all hall call buttons, all car call buttons	0.430	0.00	100	271.66	1729.31	418.83
3	all hall call buttons, all car call buttons, car lantern up, down, car position indicator, dispatcher	0.296	0.00	100	434.61	2455.47	539.14
4	car position indicator	0.950	100.00	100	246.58	351.46	663.43
5	car lantern up, down, car position indicator	0.900	100.00	100	397.26	369.99	425.85
6	all hall call buttons, all car call buttons, dispatcher	0.346	0.00	100	366.46	2457.92	457.02
7	hall call up 1, 2, 3, hall call down 5, 6, 7, car call 1, 2, 6, 7	0.683	30.27	100	284.38	1038.35	1363.50
8	hall call up 1, 2, 3, hall call down 5, 6, 7, car call 2, 3, 5, 6	0.720	33.73	100	381.69	499.70	3837.16
9	hall call up 2, 3, 4, hall call down 4, 5, 6, 7, car call 1, 2, 7	0.681	49.47	100	356.51	856.46	648.86
10	hall call up 2, 3, hall call down 5, 6, car call 1, 2, 3, 5, 6, 7	0.688	38.93	100	301.83	935.91	740.47
11	hall call up 1, hall call down 3, 4, 5, 6, 7, car call 1, 2, 6, 7	0.683	25.20	100	303.65	950.48	1399.64

Table E.2. Elevator Experimental Data for Configurations 12 - 25

Config ID#	Failed Components	System Utility Value	Avg % Delivered in Original Elevator	Avg % Delivered in Gracefully Degrading Elevator	Avg Delivery Time for Two-Way (secs)	Avg Delivery Time for Down-Peak (secs)	Avg Delivery Time for Up-Peak (secs)
12	hall call up 1, 2, 3, 4, 5 hall call down 7, car call 1, 2, 6, 7	0.683	42.53	100	309.44	1122.50	1290.20
13	hall call up 2, 3, 6, hall call down 5, 6, 7, car call 1, 2, 5, 6	0.683	35.73	100	330.30	703.94	672.42
14	hall call up 1, 2, 6, hall call down 5, 6, 7, car call 1, 2, 3, 7	0.683	31.20	100	315.11	961.52	1012.70
15	hall call up 2, 3, 6, hall call down 2, 5, 6, car call 1, 2, 6, 7	0.683	36.33	100	298.80	1049.98	650.73
16	hall call up 1, 2, 3, hall call down 4, 5, 6, 7, car call 3, 4, 5	0.718	32.60	100	403.14	524.33	3669.11
17	hall call up 1, 2, 3, 4, hall call down 4, 5, 6, 7, car call 2, 3, 4, 5, 6	0.636	21.87	100	441.04	578.91	4194.87
18	hall call up 1, 2, 3, 4, hall call down 4, 5, 6, 7, car call 1, 2, 5, 6, 7	0.599	22.80	100	355.82	967.27	1274.87
19	hall call up 2, 3, 4, 5, hall call down 3, 4, 5, 6, car call 2, 3, 4, 5, 6	0.636	15.87	100	324.98	494.95	770.50
20	hall call up 2, 3, 4, hall call down 4, 5, 6, 7, car call 1, 3, 4, 5, 6, 7	0.601	23.40	100	348.44	875.25	740.55
21	hall call up 2, 3, 4, 5, 6, hall call down 2, 3, 4, 5, 6, car call 1, 4, 7	0.594	43.07	100	399.05	1206.21	641.91
22	hall call up 1, 2, 4, 5, hall call down 3, 4, 6, 7, car call 1, 2, 4, 5, 7	0.599	20.73	100	327.21	923.25	1184.14
23	hall call up 2, 3, 4, 6, hall call down 2, 3, 4, 5, 6, car call 1, 3, 4, 7	0.597	26.20	100	361.89	1224.72	673.52
24	hall call up 1, 3, 4, hall call down 2, 3, 5, 7, car call 1, 2, 3, 4, 5, 7	0.601	18.87	100	355.61	1257.76	1591.40
25	hall call up 2, 3, 4, 5, 6, hall call down 2, 3, 4, 5, 6, car call 3, 4, 5	0.631	16.53	100	366.38	598.23	668.94

Table E.3. Elevator Experimental Data for Configurations 26 - 39

Config ID#	Failed Components	System Utility Value	Avg % Delivered in Original Elevator	Avg % Delivered in Gracefully Degrading Elevator	Avg Delivery Time for Two-Way (secs)	Avg Delivery Time for Down-Peak (secs)	Avg Delivery Time for Up-Peak (secs)
26	hall call up 1, 2, 3, 4, 5 hall call down 2, 4, 5, 6, 7, car call 2, 5, 6	0.631	16.40	100	419.75	541.25	5221.57
27	hall call up 1, 2, hall call down 6, 7, car call 1, 2, 7	0.768	38.40	100	273.62	937.29	910.99
28	hall call down 2, 6, 7, car call 2, 3, 5, 7	0.807	54.13	100	266.21	481.73	687.95
29	hall call up 1, 2, 3, 4, hall call down 3, car call 1, 2, 5, 7	0.712	44.80	100	320.94	938.92	1142.01
30	hall call up 1, 2, 5, 6, hall call down 3, 6, 7	0.798	51.27	100	387.62	511.00	4397.82
31	hall call up 5, 6, hall call down 2, 6, 7, car call 2, 6	0.802	53.47	100	301.84	463.37	616.40
32	hall call up 1, 2, 6, hall call down 2, 6, 7, car call 7	0.800	39.33	100	327.28	447.05	4077.63
33	hall call up 1, 2, 3, hall call down 3, 6, 7, car call 1	0.763	35.53	100	380.30	896.67	962.00
34	hall call up 1, 2, 6, hall call down 4, 6, car call 2, 5	0.802	49.00	100	326.25	488.35	3079.36
35	hall call up 2, 5, hall call down 3, 6, car call 2, 4, 5	0.804	57.07	100	295.24	535.12	674.32
36	hall call up 1, 2, 6, hall call down 3, 7, car call 1, 2, 4, 7	0.712	36.27	100	277.30	799.63	1200.30
37	hall call up 1, 2, 3, hall call down 5, 6, 7, car call 1, 2, 6, 7, dispatcher	0.451	0.00	100	340.65	2462.69	445.72
38	hall call up 1, 2, 3, hall call down 5, 6, 7, car call 2, 3, 5, 6, dispatcher	0.487	0.00	100	337.15	391.11	439.13
39	hall call up 2, 3, 4, hall call down 4, 5, 6, 7, car call 1, 2, 7, dispatcher	0.452	0.00	100	336.25	2458.62	416.42

Table E.4. Elevator Experimental Data for Configurations 40 - 50

Config ID#	Failed Components	System Utility Value	Avg % Delivered in Original Elevator	Avg % Delivered in Gracefully Degrading Elevator	Avg Delivery Time for Two-Way (secs)	Avg Delivery Time for Down-Peak (secs)	Avg Delivery Time for Up-Peak (secs)
40	hall call up 2, 3, hall call down 5, 6, car call 1, 2, 3, 5, 6, 7, dispatcher	0.449	0.00	100	334.11	2450.25	415.14
41	hall call up 1, hall call down 3, 4, 5, 6, 7, car call 1, 2, 6, 7, dispatcher	0.451	0.00	100	332.14	2458.01	444.99
42	hall call up 1, 2, 3, 4, 5 hall call down 7, car call 1, 2, 6, 7, dispatcher	0.451	0.00	100	326.31	2471.80	442.27
43	hall call up 2, 3, 6, hall call down 5, 6, 7, car call 1, 2, 5, 6, dispatcher	0.451	0.00	100	328.92	2464.17	413.32
44	hall call up 1, 2, 6, hall call down 5, 6, 7, car call 1, 2, 3, 7, dispatcher	0.451	0.00	100	338.21	2461.80	447.78
45	hall call up 2, 3, 6, hall call down 2, 5, 6, car call 1, 2, 6, 7, dispatcher	0.451	0.00	100	326.15	2454.97	410.60
46	hall call up 1, 2, 3, hall call down 4, 5, 6, 7, car call 3, 4, 5, dispatcher	0.488	0.00	100	332.97	403.39	418.08
47	hall call up 1, 2, 3, 4, hall call down 4, 5, 6, 7, car call 2, 3, 4, 5, 6, dispatcher	0.453	0.00	100	361.00	399.65	445.30
48	hall call up 1, 2, 3, 4, hall call down 4, 5, 6, 7, car call 1, 2, 5, 6, 7, dispatcher	0.416	0.00	100	341.54	2462.21	449.50
49	hall call up 2, 3, 4, 5, hall call down 3, 4, 5, 6, car call 2, 3, 4, 5, 6, dispatcher	0.453	0.00	100	357.16	405.93	414.13
50	hall call up 2, 3, 4, hall call down 4, 5, 6, 7, car call 1, 3, 4, 5, 6, 7, dispatcher	0.415	0.00	100	340.94	2455.32	431.08

Table E.5. Elevator Experimental Data for Configurations 51 - 62

Config ID#	Failed Components	System Utility Value	Avg % Delivered in Original Elevator	Avg % Delivered in Gracefully Degrading Elevator	Avg Delivery Time for Two-Way (secs)	Avg Delivery Time for Down-Peak (secs)	Avg Delivery Time for Up-Peak (secs)
51	hall call up 2, 3, 4, 5, 6, hall call down 2, 3, 4, 5, 6, car call 1, 4, 7, dispatcher	0.418	0.00	100	334.67	2451.13	416.31
52	hall call up 1, 2, 4, 5, hall call down 3, 4, 6, 7, car call 1, 2, 4, 5, 7, dispatcher	0.416	0.00	100	354.53	2462.28	441.34
53	hall call up 2, 3, 4, 6, hall call down 2, 3, 4, 5, 6, car call 1, 3, 4, 7, dispatcher	0.417	0.00	100	338.24	2447.57	421.48
54	hall call up 1, 3, 4, hall call down 2, 3, 5, 7, car call 1, 2, 3, 4, 5, 7, dispatcher	0.415	0.00	100	334.41	2468.34	453.04
55	hall call up 2, 3, 4, 5, 6, hall call down 2, 3, 4, 5, 6, car call 3, 4, 5, dispatcher	0.338	0.00	100	343.72	413.43	408.79
56	hall call up 1, 2, 3, 4, 5 hall call down 2, 4, 5, 6, 7, car call 2, 5, 6, dispatcher	0.455	0.00	100	340.57	406.00	429.50
57	hall call up 1, 2, hall call down 6, 7, car call 1, 2, 7, dispatcher	0.485	0.00	100	330.35	2462.56	440.10
58	hall call down 2, 6, 7, car call 2, 3, 5, 7, dispatcher	0.521	0.00	100	310.00	401.24	410.10
59	hall call up 1, 2, 3, 4, hall call down 3, car call 1, 2, 5, 7, dispatcher	0.462	0.00	100	329.23	2471.40	442.28
60	hall call up 1, 2, 5, 6, hall call down 3, 6, 7, dispatcher	0.525	0.00	100	314.90	419.41	401.57
61	hall call up 5, 6, hall call down 2, 6, 7, car call 2, 6, dispatcher	0.523	0.00	100	310.03	404.35	402.30
62	hall call up 1, 2, 6, hall call down 2, 6, 7, car call 7, dispatcher	0.524	0.00	100	325.08	409.13	415.39

Table E.6. Elevator Experimental Data for Configurations 63 - 70

Config ID#	Failed Components	System Utility Value	Avg % Delivered in Original Elevator	Avg % Delivered in Gracefully Degrading Elevator	Avg Delivery Time for Two-Way (secs)	Avg Delivery Time for Down-Peak (secs)	Avg Delivery Time for Up-Peak (secs)
63	hall call up 1, 2, 3, hall call down 3, 6, 7, car call 1, dispatcher	0.487	0.00	100	324.88	2456.74	434.88
64	hall call up 1, 2, 6, hall call down 4, 6, car call 2, 5, dispatcher	0.523	0.00	100	330.22	396.73	416.54
65	hall call up 2, 5, hall call down 3, 6, car call 2, 4, 5, dispatcher	0.522	0.00	100	342.18	409.34	405.99
66	hall call up 1, 2, 6, hall call down 3, 7, car call 1, 2, 4, 7, dispatcher	0.462	0.00	100	321.46	2471.46	436.75
67	car lantern up, down, car position indicator, dispatcher	0.554	0.00	100	364.62	412.84	470.42
68	dispatcher	0.604	0.00	100	296.40	404.78	389.84
69	no failed components	1.000	100.00	100	203.19	343.38	580.87
70	atfloor sensor 2, 3, 4, 5, 6, 7	1.000	0.00	100	203.19	343.38	580.87

Appendix F - Utility Specification for the Mobot System

This appendix lists the utility functions we specified for the mobot navigation system in Chapter 6. The system utility can be evaluated directly from the utility of the Actuator Control feature subset.

Table F.1. Utility Specification Navigation, Line Following, and Path Planner Feature Subsets.

Feature Subset	Configuration	Utility Function
Actuator Control	{Command Resolver, Servo Motor Controller, Drive Motor Controller, Steering Servo Motor, Drive Motor, Navigation Feature Subset ($U_{Navigation}$)}	$U_{Actuator} = 0.2 + 0.8*U_{Navigation}$
	All other configurations	$U_{Actuator} = 0$
Navigation	Any configuration in which at least one of the two feature subsets: Line Follower Feature Subset ($U_{LineFollower}$), Path Planner Feature Subset (U_{Path}) is working and any combination of: Collision Detection Feature Subset ($U_{Collision}$), Direction Feature Subset ($U_{Direction}$)	$U_{Navigation} = 0.2*U_{LineFollower} + 0.6*U_{Path} + 0.1*U_{Collision} + 0.1*U_{Direction}$
	All other configurations	$U_{Navigation} = 0$
Line Follower	Any configuration in which at least one of the two components: Line Follower, Line Detectors Feature Subset ($U_{LineDetectors}$) is working and any combination of: X Location Feature Subset ($U_{XLocation}$), Decision Point Detector ($u_{decision}$), Map Data Server ($u_{mapdata}$)	$U_{LineFollower} = 0.1 + 0.6*U_{LineDetectors} + 0.1*u_{decision} + 0.1*u_{mapdata} + 0.1*U_{XLocation}$
	All other configurations	$U_{LineFollower} = 0$
Line Detectors	Any configuration in which at least one of the Line Detector Feature Subsets (U_{Line0} .. U_{Line5}) is working	$U_{LineDetectors} = (U_{Line0} + ... + U_{Line5})/6$
	All Line Detector Feature Subsets failed	$U_{LineDetectors} = 0$
Line Detector 0 .. 5	{Line Detector Component L, IR Sensor L}	$U_{LineL} = 1$
	All other configurations	$U_{LineL} = 0$
Path Planner	{Path Planner, Map Data Server, X Location Feature Subset ($U_{XLocation}$), Y Location Feature Subset ($U_{YLocation}$)}	$U_{Path} = 0.2 + 0.4*U_{XLocation} + 0.4*U_{YLocation}$
	All other configurations	$U_{Path} = 0$

Table F.2. Utility Specification for the X/Y Location and Sensor Feature Subsets.

Feature Subset	Configuration	Utility Function
X/Y Location	Any configuration in which the X/Y Location Resolver and at least one of the feature subsets: Dead Reckoning Feature Subset (U_{Dead}), Line X/Y Estimator Feature Subset ($U_{LineEX/Y}$), Crack/Collision X/Y Estimatior Feature Subset ($U_{CCEX/Y}$) are working	$U_{X/YLocation} = 0.1 + 0.5*U_{Dead} + 0.3*U_{LineEX/Y} + 0.1*U_{CCEX/Y}$
	All other configurations	$U_{X/YLocation} = 0$
Line X/Y Estimator	Any configuration in which the Line X/Y Estimator component and the Line Follower Feature Subset are working and any combination of: Direction Feature Subset, Map Data Server	$U_{LineEX/Y} = 0.1 + 0.3*U_{LineFollower} + 0.2*u_{mapdata} + 0.4*U_{Direction}$
	All other configurations	$U_{LineEX/Y} = 0$
Crack/Collision X/Y Estimator	Any configuration in which the Crack/Collision Sensor X/Y component, Crack Detection Feature Subset, and Collision Detection Feature Subset are working and any combination of: Direction Feature Subset, Map Data Server	$U_{CCEX/Y} = 0.1 + 0.2*U_{CrackDetector} + 0.1*U_{Collision} + 0.2*u_{mapdata} + 0.4*U_{Direction}$
	All other configurations	$U_{CCEX/Y} = 0$
Front Wheel Encoder	{Front Wheel Shaft Encoder Sensor, Encoder Counter}	$U_{FrontWheel} = 1$
	All other configurations	$U_{FrontWheel} = 0$
Rear Wheel Revolutions	{Left Wheel IR Sensor, Left Wheel Rev Counter, Right Wheel IR Sensor, Right Wheel Rev Counter}}	$U_{RearWheel} = 1$
	All other configurations	$U_{RearWheel} = 0$
Crack Detection	{Pavement Crack Sensor, Crack Detector}	$U_{CrackDetector} = 1$
	All other configurations	$U_{CrackDetector} = 0$
Collision Detection	Any configuration in which the Collision Detector and at least one of the two sensors: Left Whisker Sensor ($u_{leftwhisker}$), Right Whisker Sensor($u_{rightwhisker}$) is working	$U_{Collision} = 0.5*u_{leftwhisker} + 0.5*u_{rightwhisker}$
	All other configurations	$U_{Collision} = 0$

Table F.3. Utility Specification for the Dead Reckoning Feature Subset.

Feature Subset	Configuration	Utility Function
Dead Reckoning	Any configuration in which the Dead Reckoner and at least one of the feature subsets: Front Wheel Encoder Feature Subset ($U_{FrontWheel}$), Rear Wheel Revolutions Feature Subset ($U_{RearWheel}$), Speed Feature Subset (U_{Speed}), Direction Feature Subset ($U_{Direction}$) are working	$U_{Dead} = 0.4*U_{FrontWheel} + 0.2*U_{RearWheel} + 0.1*U_{Speed} + 0.3*U_{Direction}$
	All other configurations	$U_{Dead} = 0$
Direction	Any configuration in which the Direction Estimator and at least one of the components: Command Resolver ($u_{command}$), Front Wheel Encoder Feature Subset, Rear Wheel Encoder Feature Subset is working	$U_{Direction} = 0.1 + 0.3*u_{command} + 0.3*U_{FrontWheel} + 0.3*U_{RearWheel}$
	All other configurations	$U_{Direction} = 0$
Speed	Any configuration in which the Speed Estimator and at least one of the components: Command Resolver, Front Wheel Encoder Feature Subset, Rear Wheel Encoder Feature Subset is working	$U_{Speed} = 0.1 + 0.3*u_{command} + 0.3*U_{FrontWheel} + 0.3*U_{RearWheel}$
	All other configurations	$U_{Speed} = 0$