Using Architectural Properties to Model and Measure Graceful Degradation

Charles Shelton, Philip Koopman

Electrical and Computer Engineering Department
Carnegie Mellon University
Pittsburgh, PA 15213, USA
{cshelton, koopman}@cmu.edu

Abstract. System-wide graceful degradation may be a viable approach to improving dependability in computer systems. In order to evaluate and improve system-wide graceful degradation we present a system model that will explicitly define graceful degradation as a system property, and measure how well a system gracefully degrades in the presence of multiple combinations of component failures. The system's software architecture plays a major role in this model, because the interface and component specifications embody the architecture's abstraction principle. We use the architecture to group components into subsystems that enable reasoning about overall system utility. We apply this model to an extensive example of a distributed embedded control system architecture to specify the relative utility of all valid system configurations. We then simulate working system configurations and compare their ability to provide functionality to the utility measures predicted by our model.

1 Introduction

Dependability is a term that covers many system properties such as reliability, availability, safety, maintainability, and security [1]. System dependability is especially important for embedded computer control systems, which pervade everyday life and can have severe consequences for failure. These systems increasingly implement a significant portion of their functionality in software, making software dependability a major issue.

Graceful degradation may be a viable approach to achieving better software dependability. If a software system can gracefully degrade automatically when faults are detected, then individual software component failures will not cause complete system failure. Rather, component failures will remove the functionality derived from that component, while still preserving the operation of the rest of the system. Specifying and achieving system-wide graceful degradation is a difficult research problem. Current approaches require specifying every system failure mode ahead of time, and designing a specific response for each such mode. This is impractical for a complex software system, especially a fine grained distributed embedded system with tens or hundreds of software and hardware components.

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.

In order to evaluate and improve system-wide graceful degradation, we present a system model that enables scalable specification and analysis of graceful degradation. We base the model on using the system's interface definitions and component connections to group the system's components into subsystems. We hypothesize that the software architecture, responsible for the overall organization of and connections among components, can facilitate the system's ability to implicitly provide graceful degradation, without designing a specific response to every possible failure mode at design time. We define a failure mode to be a set of system components failing concurrently. By using the model to measure how gracefully a system degrades, we predict that we can identify what architectural properties facilitate and impede system-wide graceful degradation.

We demonstrate the usefulness of our model by applying it to a representative distributed embedded system and showing how we can achieve scalable analysis of graceful degradation. Our example system is a gracefully degrading elevator control system that was designed by an elevator engineer and implemented in a discrete event simulator (an earlier version of this system was presented in [2]). We use the simulator to perform fault injection experiments by failing nodes during the system's operation to observe how well the system gracefully degrades, and compare the results to the predictions of our model.

The rest of the paper is organized as follows. Section 2 identifies the problem of specifying graceful degradation and how our model is scalable. Section 3 provides a description of the key features of our model. Section 4 details our elevator control system architecture in terms of the defined components and interfaces. Section 5 shows how we applied our system model for graceful degradation to the elevator architecture to specify the system utility function. Section 6 describes a preliminary fault injection experiment we performed to validate the model. Section 7 describes previous related work. Finally, section 8 ends with conclusions and future work.

2 Specifying Graceful Degradation

For graceful degradation to be possible, it must be possible to define the system's state as "working" with other than complete functionality. In many systems, a substantial portion of the system is built to optimize properties such as performance, availability, and usability. We must be able to define the minimum functionality required for primary missions, and treat optimized functionality as a desirable, but optional, enhancement. For example, 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.

Specifying and designing system-wide graceful degradation is not trivial. Graceful degradation mechanisms must handle not only individual component failures, but also combinations of component failures. Current graceful degradation techniques emphasize adding component redundancy to preserve perfect operation when failures occur, or designing several redundant backup systems 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.

We define graceful degradation in terms of system utility: a measure of the system's ability to provide its specified functional and non-functional capabilities. A system that has all of its components functioning properly has maximum utility. A system degrades gracefully if individual component failures reduce system utility proportionally to the severity of aggregate failures. 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 much 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.

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 [3]. 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 $O(2^k)$, where k is the number of components within a single subsystem. When we have a complete utility function for all possible software configurations, we can use techniques described in [4] to analyze the utility of system hardware configurations to determine the best allocation of software components to hardware nodes.

3 System Model

System *utility* is a key concept in our model for comparing system configurations. Utility is a measure of how much benefit can be gained from using a system. Overall system utility may be a combination of functionality, performance, and dependability properties. If we specify the relative utility values of each of the 2^N possible configurations of N software components, sensors, and actuators, 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 software data flow model 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, and is primarily concerned with how system functionality changes when software components fail.

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. We assume that components can either be in one of two states: working or failed. Working means that the component has enough resources to output its specified system variables. Failed means the component cannot produce its specified outputs. Individual components are designed to shut down when they detect an unrecoverable error, which enables the rest of the system to quickly detect the component's failure, and prevents an error from propagating through the rest of the system. All faults in our model thus manifest 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.

Section 3.1 describes our system data flow graph, section 3.2 details how we perform our utility analysis, and section 3.3 identifies some of the key assumptions of our model.

3.1 Data Flow Graph and Feature Subset Definitions

We consider a system as a set of software, sensor, and actuator components. We construct our system model as a directed data flow graph in which the vertices represent system components (sensors, actuators, and software components), and the edges represent the communication among components via their input and output interfaces. We use these interfaces to define a set of *system variables* that represent an abstraction of all component interaction. These variables can represent any communication structure in the software implementation. Actuators receive input variables from the system and output them to the environment, while sensors input variables from the environment to the system. Our data flow graph can be derived directly from the system's software architecture, which specifies the system's components and interfaces, as well as component organization and dependencies.

We then partition the data flow graph into subgraphs that represent logical subsystems that we term *feature subsets*. These feature subsets form the basis for how we decompose the system utility analysis. 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 or operate a system actuator. Feature subsets may or may not be disjoint and can share components across different subsets. Feature subsets also capture the hierarchical decomposition of the software system, as "higher level" feature subsets contain "lower level" feature subsets as components, which further encapsulate other software, sensor, and actuator components.

The feature subset data flow graphs can also represent dependency relationships among components. Each component might not require all of its inputs to provide partial functionality. For example, in an elevator the door controllers can use the inputs from the passenger request buttons to more efficiently open and close the doors based on passenger input, but this is an enhancement to normal door operation that simply waits a specified period before opening and closing the doors. If the door controllers no longer received these button inputs, they could still function correctly.

We annotate our feature subset graph edges with a set of dependency relationships among components. 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. 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.

3.2 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. 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 individual 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.

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, any system configuration in which the elevator's drive motor has failed cannot provide its basic system functionality and is invalid, 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 subsystem definitions to specify the utility of these system configurations.

If we restrict our analysis to individual feature subset component configurations, we only need to rank the relative utility values of all valid configurations within each feature subset. For feature subsets with a maximum of $k << N$ components, this is a much more feasible task. We only need to manually rank at most the utilities of 2^k possible configurations for each feature subset Additionally, we can significantly

reduce the number of configurations we must consider by using component dependencies to determine the valid and invalid configurations of each feature subset.

We can then determine overall system utility by examining the system configurations of the "top level" feature subsets that provide outputs to system actuators. All other feature subsets utility evaluations are encapsulated within these subsystems that provide external system functionality. We can completely specify the system utility function without manually specifying the relative utility values of all 2^N possible system component configurations, but rather specifying the utilities of 2^k feature subset configurations for each feature subset in the system.

We can use this model to develop a space of systems with varying degrees of graceful degradation. At one end of the spectrum, we have extremely "brittle" systems that are not capable of any graceful degradation at all. In these systems, any one component failure will result in a complete system failure. In our model, this would be a system where every component is a member of at least one required feature subset, and each feature subset strongly depends on all of its components. Therefore, every component must be functioning to have positive system utility.

Similarly, any modular redundant system can be represented as a collection of several feature subsets, where each feature subset contains multiple copies of a component plus a voter. The valid configurations that provide positive utility for each feature subset are those that contain the voter plus one or more component copies. This redundant system can tolerate multiple failures across many feature subsets, but cannot tolerate the failure of any one voter or all the component copies in any one feature subset.

At the other end of the spectrum, an ideal gracefully degrading system is one where any combination of component failures will still leave a system with positive utility. In our model, this system would be one where none of its feature subsets would be labeled as required for basic functionality, and every component would be completely optional to each feature subset in which it was a member. The system would continue to have positive utility until every component failed.

3.3 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. We have made 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. Because system architects generally attempt to decouple subsystems, we assume that encapsulated feature subsets are not strongly coupled. If some subsystems are strongly coupled, one could apply multi-attribute utility theory [5] to deal with the added system complexity within the model.

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 so that it contributes zero utility instead.

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. 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.

4 Example System: A Distributed Elevator Control System

To illustrate how we can apply our system model to a real system, we use a design of a relatively complex distributed elevator control system. This system was designed by an elevator engineer (the second author) 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.

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 to 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 to the passengers. 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}
- **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)

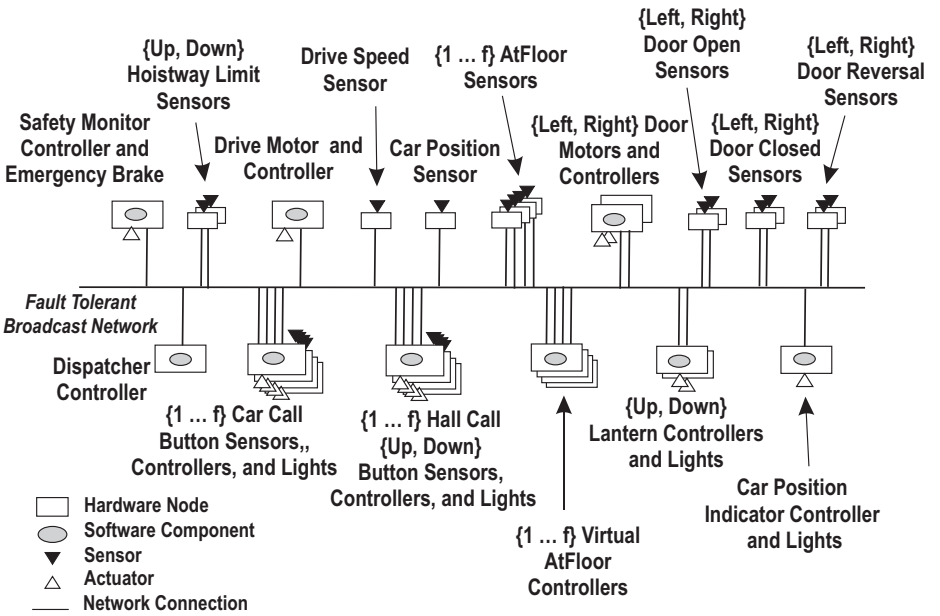

Figure 1. Hardware View of the Elevator Control System

- **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.

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

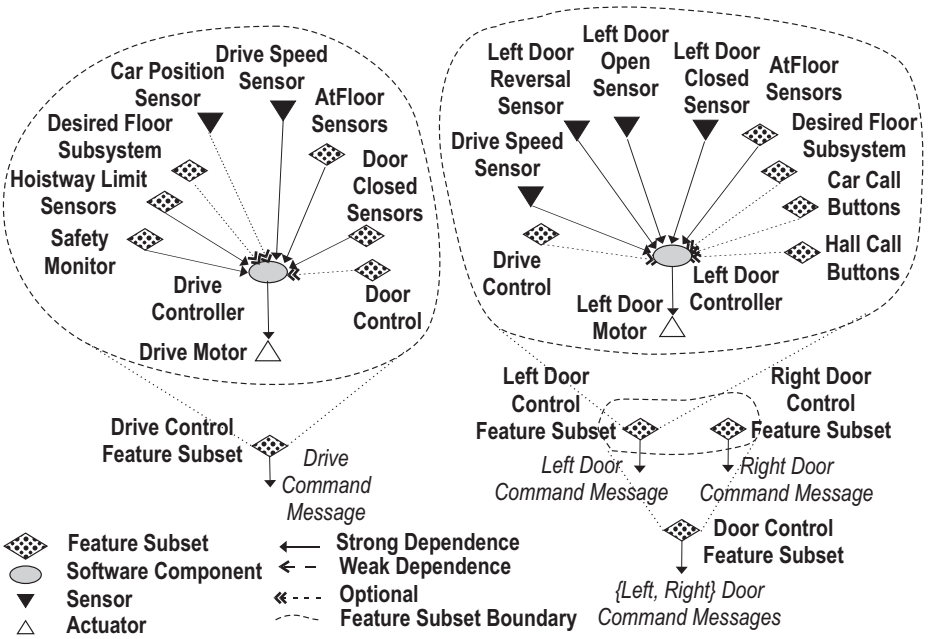

Figure 2. Feature Subset Graphs of the Door and Drive Control Subsystems

lights determine when to activate the lights to indicate appropriate feedback to the passengers. Additionally, since the AtFloor sensors are a critical resource for the elevator system, we have 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. The elevator control system consists of $8 + 4f$ sensors, $5 + 3f$ actuators, and $6 + 4f$ software components, for a total of $19 + 11f$ components in the system. Figure 1 illustrates how these system components are allocated to hardware nodes in the elevator's distributed control system.

In our experiments, we simulated an elevator with seven floors, meaning that there were a total of 96 components in the system. To specify how well the system gracefully degrades with respect to all possible combinations of component failures, the traditional approach would require a manual ranking of the utility of all $2^{96} = 7.92 * 10^{28}$ possible system configurations. Our model exploits the information available from the system architecture to overcome this exponential difficulty.

5 Specifying 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

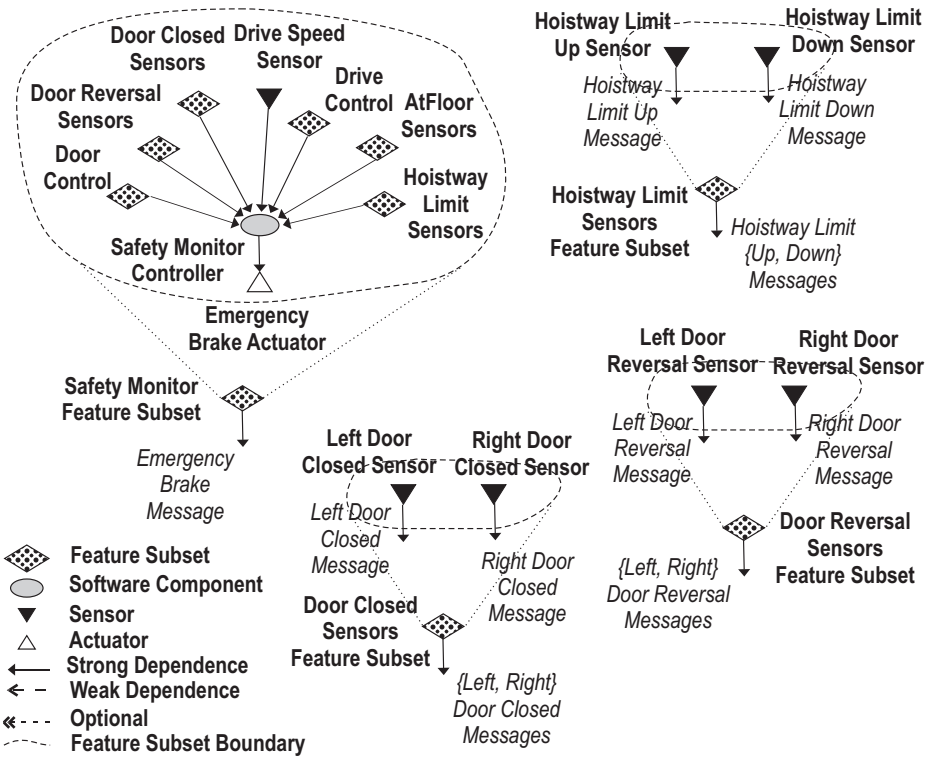

Figure 3. Feature Subset Graphs of the Safety Monitor Subsystem

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.

5.1 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 2 displays 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.

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

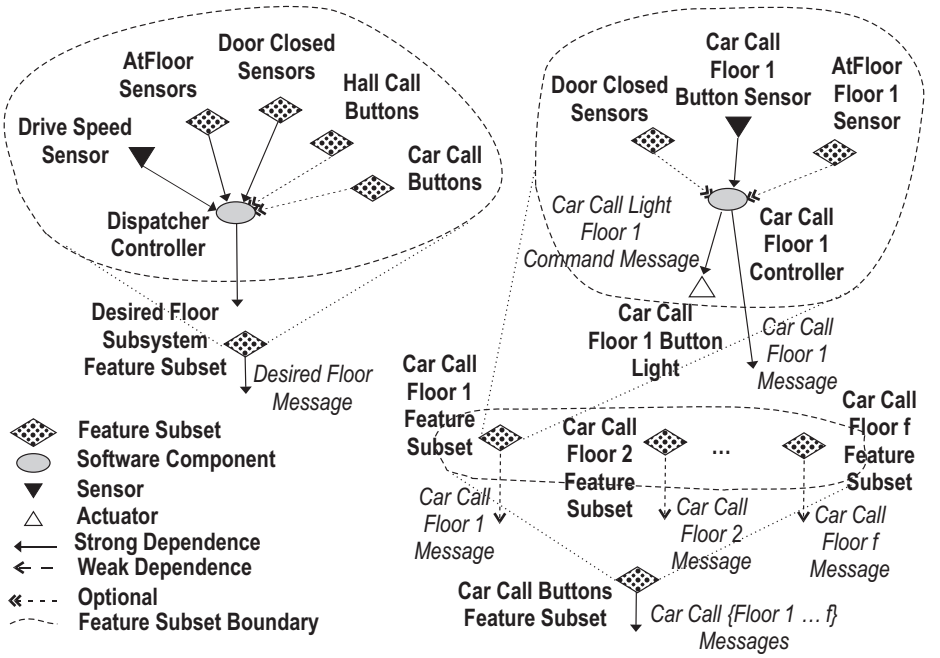

Figure 4. Feature Subset Graphs for the Desired Floor and Car Call Button Subsystems

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.

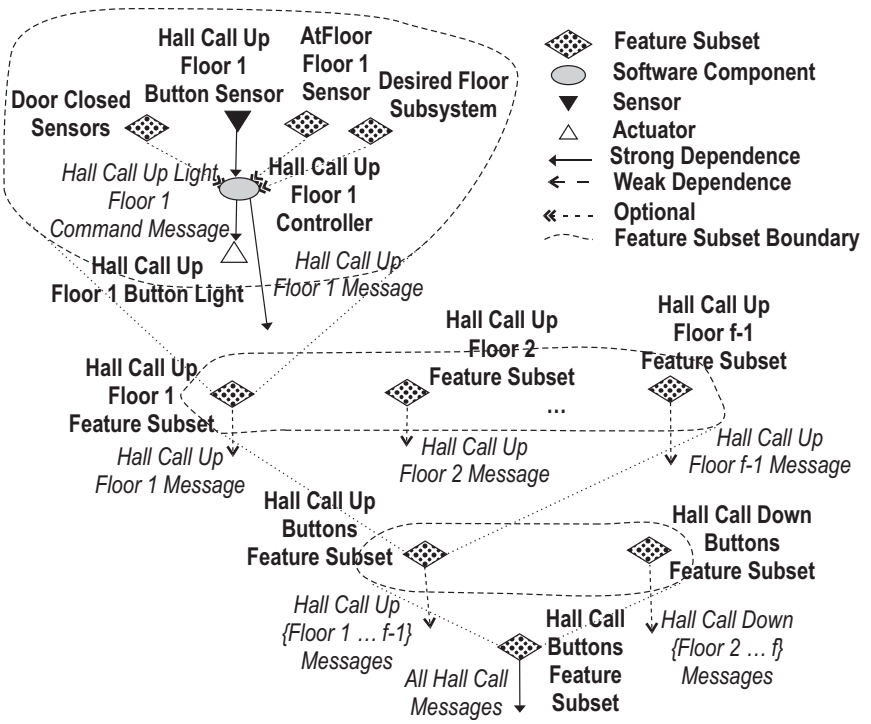

Figure 5. Elevator Hall Call Button Feature Subsets

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 elevator crashing 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 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 Dispatcher software component that implements the algorithm for determining the next floor at which the elevator

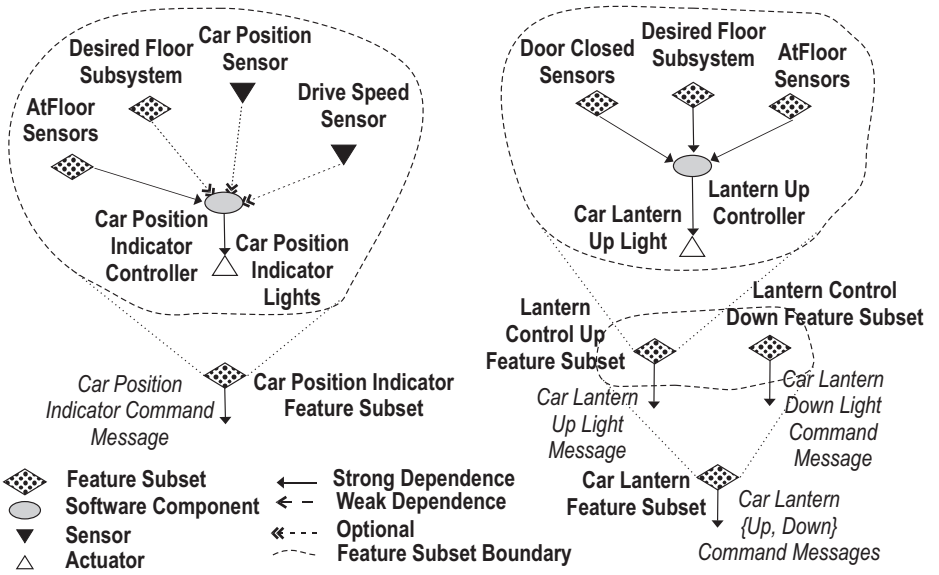

Figure 6. Car Position Indicator and Car Lantern Feature Subsets

should stop. The Dispatcher receives inputs from the Car Call and Hall Call buttons to determine passenger intent and compute the elevator's next destination. 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 4 and 5 show the feature 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. 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. Figure 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

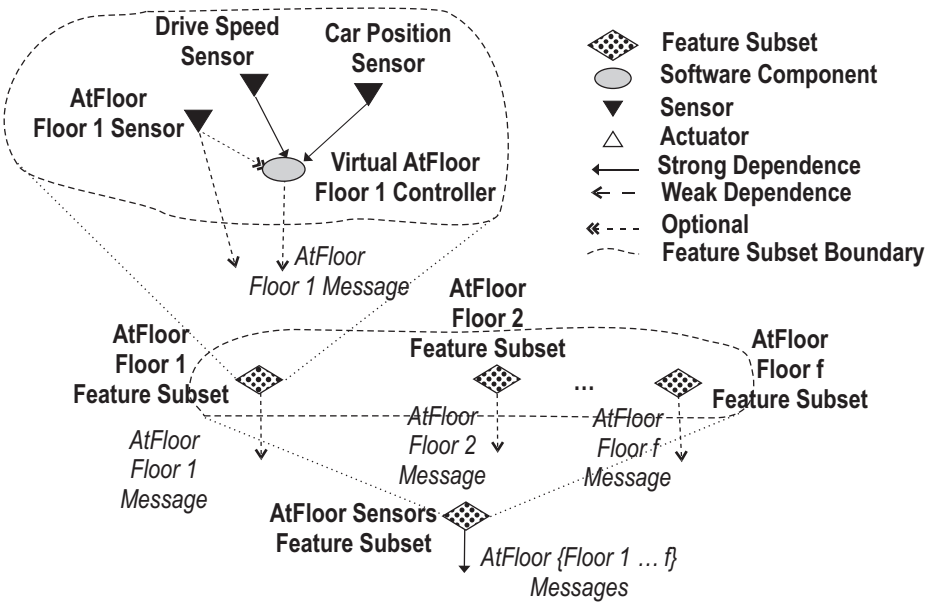

Figure 7. AtFloor Subsystem Feature Subset Graph

Position and Drive Speed sensors when the physical AtFloor sensors fail. Thus they are included in the AtFloor sensor feature subset graphs. Figure 7 shows the AtFloor feature subset description for the elevator system in our model.

5.2 Utility Analysis

The elevator system has a total of $19 + 11f$ system components, meaning there are 2^{19+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) 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 $1 + 9f$ optional components that can have 2^{1+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 VirtualAtFloor component with a working Car Position sensor. If the Car Position sensor breaks, then all AtFloor sensors must work. Since all the AtFloor sensors must work in this situation, they are fixed and have one configuration. However, the VirtualAtFloor components can either work or not work since their failure

Table 1. Valid Configurations in each Feature Subset

Feature Subset	# Similar 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	4	4
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***		**33 + 53*f***

will not affect the availability of the AtFloor system variables, making 2^f valid combinations for the various VirtualAtFloor components. If the Car Position sensor works, then one or both AtFloor sensor and VirtualAtFloor 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 utility zero is $(2^f + 3^f)(2^{1+9f})$. For our elevator with seven floors this is approximately $4.27 * 10^{22}$ configurations that still must be manually ranked. This is a significant reduction from the $7.92 * 10^{28}$ 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 $16 + 4f$ 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 1 summarizes the number of valid configurations that must be assigned utility values in each feature subset for a total of $33 + 53f$ feature subset configurations that must be considered across the entire elevator system. For our seven floor elevator, this totals 404 valid feature subset component configurations for the entire system.

We can then determine overall system utility by composing the system configurations of the "top level" 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 feature subsets in the system. Once we specify the relative utilities of these 16 configurations in addition to the 404 total feature subset configurations, we can completely specify the system utility function. We have greatly reduced the number of configurations we must evaluate from $4.27 * 10^{22}$ system component configurations to 420 feature subset configurations to assess the system's ability to gracefully degrade.

6 Experimental Validation

If our model accurately predicts the relative utility of all system configurations, we can assess how well the system gracefully degrades by observing how system utility changes when the system configuration changes as components fail. To validate our model, we performed some preliminary fault injection experiments on a simulated elevator implementation. 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

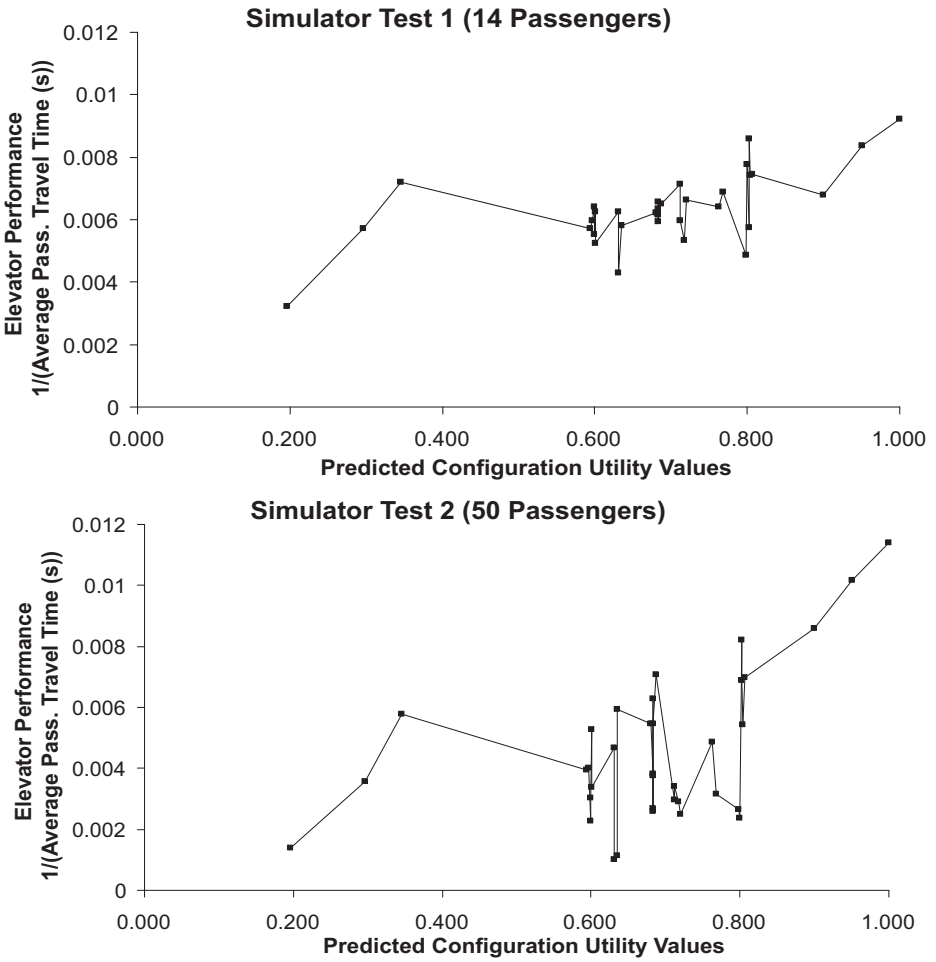

Figure 8. Utility vs. Average Performance for Selected Elevator System Configurations

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.

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 their destination, from the time they first arrive to use the elevator to the time they step off the elevator at their intended floor.

We selected a small subset of the possible valid elevator system configurations, and ran two passenger profiles on each system configuration. 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 also picked 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. One encouraging result of our experiment is that every valid configuration we tested eventually delivered all passengers to their destination regardless of which set of system components were failed.

We measured the average performance of each system configuration and compared it to its system utility as predicted by our model. If our model accurately predicted system utility, we should see configurations that have higher utility measures achieve better average performance. Figure 8 graphs the utility of the tested system configurations versus the elevator performance per passenger. The system configurations on the horizontal axis are ordered by utility, so the performance measures should be monotonically increasing. The graphs show a general trend of increasing performance, but it is not monotonically increasing.

However, elevator performance can be largely affected by how frequently passengers arrive and how the dispatcher deals with a loss of button inputs. Our dispatcher algorithm would periodically send the elevator to visit floors for which it was not receiving button information in order to ensure that all passengers eventually get delivered to their destination. Thus, sometimes elevator performance would suffer to ensure that no passengers were stranded. The fact that none of the system configurations tested suffered a complete system failure and delivered all passengers indicates that the system can gracefully degrade in the presence of multiple component failures.

This experiment was limited because we were only able to test a small number of configurations on two passenger profiles. We plan to extend this experimental validation with a wider range of different passenger profiles, as well as test many more different system configurations. We also plan to run these experiments with variants of the elevator architecture that are designed with different degrees of graceful degradation mechanisms.

7 Related Work

Previous work on formally defining graceful degradation for computer systems was presented in [6]. 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 achieving graceful degradation is exponentially complex with the number of system components. Our model for specifying graceful degradation overcomes this difficulty by encapsulating utility evaluations within individual subsystem configurations rather than evaluating the system as a whole in a single step. Other work on graceful degradation has focused on developing formal definitions [7, 8], but has not addressed how to apply these definitions to real system specifications, nor how to overcome the problem of exponential complexity for specifying failure modes and recovery mechanisms.

Current industry practice for dealing with faults and failures in embedded systems focuses on the traditional approaches of fault tolerance and fault containment [9]. Software subsystems are physically separated into different hardware modules. Additionally, system resources, such as sensors and actuators, that may be commonly used are 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 high level of redundancy.

A promising approach to achieving system dependability is NASA's Mission Data System (MDS) architecture [10, 11]. 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.

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 [12]. Performability is a unified measure of both performance and reliability that tracks how system performance degrades in the presence of faults [13]. 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.

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 $O(2^k)$, 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. In the elevator system, we used our system model to generate a complete system utility function for all $4.27 * 10^{22}$ valid system configurations by only examining 420 subsystem configurations.

Our model consists of a software data flow graph for determining dependency relationships among software components, sensors, and actuators, 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 architecture specification. 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.

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. The model we

developed illustrates how well a system can gracefully degrade by using the software architecture's component connections to decompose the system.

In preliminary experiments on a simulated implementation of the elevator control system architecture we designed, we found that the system was resistant to multiple combinations of component failures, as predicted by the model. We validated the utility estimates we generated with our model by measuring the elevator performance of a set of system configurations that had various combinations of component failures. Since general system utility encompasses both functionality and dependability requirements, the performance of these configurations did not exactly match what our model predicted. However, every system configuration tested delivered all passengers to their destinations in both simulation tests, satisfying the minimum elevator system requirements despite a loss of system functionality. Future work will include running a more comprehensive set of simulation tests for this elevator system, as well as comparing the graceful degradation ability of different elevator architectural designs, and identifying how we can specify the parameters of our system model to more accurately measure system utility attributes and thus more closely represent the actual functionality and performance of system configurations.

9 Acknowledgments

This work was supported in part by the General Motors Collaborative Research Laboratory at Carnegie Mellon University, the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298, and Lucent Technologies.

References

1. Laprie, J.-C., "Dependability of Computer Systems: Concepts, Limits, Improvements", *Proceedings of the Sixth International Symposium on Software Reliability Engineering*, Toulouse, France, Oct. 1995, pp. 2-11.
2. 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)*, May 2002, Orlando, FL.
3. Nace, W., Koopman, P., "A Product Family Approach to Graceful Degradation," *Distributed and Parallel Embedded Systems (DIPES)*, October 2000.
4. 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.
5. Keeney, R.L., Raiffa, H., *Decisions with Multiple Objectives: Preference and Value Tradeoffs*, John Wiley & Sons, New York, 1976.
6. Herlihy, M. P., Wing, J. M., "Specifying Graceful Degradation," *IEEE Transactions on Parallel and Distributed Systems*, vol.2, no.1, pp. 93-104, 1991.
7. Jayanti, P., Chandra, T.D., Toueg, S., "The Cost of Graceful Degradation for Omission Failures," Information Processing Letters, vol. 71, no. 3-4, pp.167-172, 1999.

8. 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 19-20, 1989.
9. Rushby, J., "Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance," NASA Contractor Report CR-1999-209347, June 1999.
10. Dvorak, D., Rasmussen, R, Reeves, G., Sacks, A., "Software Architecture Themes in JPL's Mission Data System," *2000 IEEE Aerospace Conference*, March 2000, Big Sky, MT.
11. Rasmussen, R., "Goal-Based Fault Tolerance for Space Systems using the Mission Data System," *2001 IEEE Aerospace Conference*, March 2001, Big Sky, MT.
12. 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.
13. 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 21-23 1978.