*Preprint: The International Conference on Dependable Systems and Networks, DSN-2004.*

# Improving System Dependability with Functional Alternatives

Charles P. Shelton
*Research and Technology Center*
*Robert Bosch Corporation*
*Pittsburgh, PA, USA*
*cshelton@ieee.org*

Philip Koopman
*ECE Department*
*Carnegie Mellon University*
*Pittsburgh, PA, USA*
*koopman@cmu.edu*

## Abstract

*We present the concept of alternative functionality for improving dependability in distributed embedded systems. Alternative functionality is a mechanism that complements traditional performability and graceful degradation techniques. Rather than providing reduced performance or functionality when components or subsystems fail, alternative functionality replaces a lost feature with another existing system function that can substitute for the lost service. This can provide improved system dependability when it is not feasible to allocate dedicated backup systems for fault tolerance. We show how alternative functionality can be applied to enhance system dependability with a case study of an elevator control system. In simulation, an elevator design that implemented alternative functionality in some of its subsystems tolerated many combinations of component failures that caused system failures in the original design.*

## 1. Introduction

Many distributed embedded computer systems have tight cost constraints that make traditional dependability techniques infeasible. Typically, an embedded system design uses its available system resources to provide maximum features and functionality. Adding additional capacity for fault tolerance mechanisms such as dual or triplex modular redundancy often cannot be justified. Products are marketed based on features, and many customers will not perceive enough value added to the system for the additional cost of redundancy, even though it may produce a more dependable product. However, our society has become increasingly dependent on complex, distributed embedded systems. Despite the fact that they are sold based on their features, these systems must continually provide dependable service in the face of harsh environmental conditions, partial system failures or loss of resources, or human error. People will not tolerate products that do not meet a certain level of dependability, even though they will usually only pay increased costs for additional features.

Designing low cost, dependable distributed embedded systems is difficult. Designs must incorporate mechanisms for dependability, while not sacrificing resources needed for functional and performance requirements. Specifying degraded operating modes that the system can provide in the event of component or subsystem failures is a popular approach. Often a distributed embedded system, after suffering some partial failures, may have enough resources to satisfy some or all of its primary requirements, even though it cannot provide its complete functionality. However, specifying and designing degraded operating modes for all possible combinations of failures becomes exponentially complex with the number of failures that must be handled.

We introduce the concept of alternative functionality as a mechanism for specifying and achieving degraded modes and improved dependability with limited system resources. For example, consider a vehicle navigation system that provides turn-by-turn directions to the driver. The system will have a prioritized list of high-level requirements (typically referred to as marketing requirements) that define the system's services and quality requirements, but usually do not specify system functionality. We call these requirements system *objectives*. Objectives for the navigation system might include (i) giving the driver timely and accurate turning cues, (ii) providing the driver with accurate situational awareness in terms of current location and time or distance to the next turn, and (iii) indicating how close the driver is to his or her destination. We define *primary* objectives as a minimum set of requirements the system must satisfy to be considered "working." Objectives that can be set aside in degraded operating modes due to failures are considered *auxiliary* objectives. The system can potentially lose the ability to satisfy all of its auxiliary objectives and still be a working system (albeit a degraded one) as long as it continues to fulfill its primary objectives. In the navigation system, objective (i) may be considered primary, and objectives (ii) and (iii) may be considered auxiliary.

A system objective will be satisfied by reaching some minimum defined threshold of service for that objective. This threshold may be defined qualitatively and/or quantitatively. In the navigation system, objective (i) may be satisfied by giving accurate turning cues within a minimum defined deadline, and objectives (ii) and (iii) may be satisfied by providing a minimum level of accuracy in time, location and distance measurements. Most systems will be designed to optimize features and functionality with their available resources, and thus will exceed their minimum requirements for system objectives when fully working.

System objectives are further refined into more detailed functional requirements that specify all of the system's fea-

tures. These requirements determine the system's implemented functionality, where each feature specified should satisfy a subset of system objectives. In our example navigation system we specify three high level system features: an LCD visual display that provides a map and location information, audio output that provides voice directions, and the ability to provide turning hints by illuminating the turn signal lights on the dash board. Each of these features can satisfy the navigation system's objectives to some degree. The visual display provides high quality service for objective (ii) with a map and text, can adequately satisfy objective (iii) by marking the destination on the map, but only minimally satisfies objective (i) because turns may be difficult to see on the map. The voice directions provide high quality service for objective (i), and can adequately satisfy objectives (ii) and (iii) with periodic update messages, although the audio cannot provide visual location information. The ability to blink the turn signals for an upcoming turn may minimally satisfy objective (i), can somewhat satisfy objective (ii) by blinking the signals faster the closer the car is to the next turn, but cannot readily satisfy objective (iii).

The features defined in the navigation system demonstrate alternative functionality. Alternative functionality can exploit the system's available features to provide some system redundancy when failures occur without additional redundant components. The display, audio, and turn signal features are not equivalent and are not designed to provide redundancy, but they satisfy overlapping system objectives. The components that implement each of the three system features satisfy separate functional requirements and combine to provide high quality service for all of the system's objectives. However, the failure of components that implement one feature can be partially compensated by the remaining available features, despite the fact that they were not originally designed to serve as redundant backups. The complete system should provide all three features, but the failure of one or two does not necessarily mean the system can no longer satisfy its primary objective. Clearly the system is in a degraded mode when one or two of the three features has failed, but such a mode is better than shutting the system down, because the system's primary objective can still be minimally satisfied.

Alternative functionality identifies sets of functions in the system that can satisfy the same objectives, possibly with different levels of service, rather than allocating sets of redundant components that provide identical functionality. When the system loses a function that satisfied a primary objective due to a component or subsystem failure, a function that was originally installed to fulfill a different primary or an auxiliary objective may be substituted if it can also satisfy that primary objective, perhaps at a lower level of service. This may result in a reduction of service for the substitute function's original main objective, but this is often preferable to a system failure.

Our view is that alternative functionality is a generalization of redundancy mechanisms. Traditional brute force redundancy, where identical components are duplicated, are simply identical functional alternatives. Analytic or algorithmic redundancy, where multiple algorithms provide similar functionality, are functional alternatives that have the same functionality with different implementations. In general, functional alternatives may be subsystems or components designed to provide distinct system features, but can also be used to compensate for other subsystem or component failures when they occur.

We present an initial technique for evaluating a system architecture to identify where functional alternatives, and other redundancy mechanisms, may be applied. We take a bottom-up approach that evaluates the functions implemented in the architecture to identify functional alternatives. We present techniques for evaluating how functional alternatives affect the utility of a system, identifying component or subsystem “dependability bottlenecks” that can benefit from alternative functionality, and validating that the implementation of functional alternatives will satisfy system objectives as predicted by the analysis. We demonstrate the applicability of our approach with a case study of an elevator control system design.

This paper is organized as follows. Section 2 identifies related work. Section 3 describes the system model we have developed as a basis for our analysis. Section 4 describes our techniques for analyzing and applying alternative functionality in a system architecture. Section 5 gives the results of our elevator control system case study. Finally, Section 6 concludes the paper.

## 2. Related work

Our work on alternative functionality is closely related to survivability, performability and graceful degradation. Survivability [5, 6] is a property of dependability that has been proposed to define explicitly how systems degrade functionality in the presence of failures. Researchers in survivability have taken a top-down approach that focuses on specifying all necessary degraded operating modes up front and designing the system to provide those modes in the corresponding failure scenarios. Thus, a survivability specification may provide multiple sets of functional requirements that each satisfy a different subset of system objectives. Each set of requirements must minimally fulfill all primary objectives, but differs in which auxiliary objectives it supports, and at what level. Each degraded operating mode maps to a different set of functional requirements and can satisfy a different set of objectives. If the system must tolerate a large number of failure modes, the survivability specification will become increasingly complex, especially if all combinations of failure modes must be considered. Current survivability research has focused on large-scale information systems rather than embedded systems, and thus far does not address scalability issues.

Performability [7, 8] combines system performance and reliability measures into a single metric. Performability models have focused mainly on hardware performance and failures rather than software, and have typically not considered changes in system functionality. Performability techniques have traditionally focused on trading performance

for reliability. If the system cannot include redundant backup components that provide full functionality for critical subsystems, redundant components that consume fewer resources but satisfy a subset of requirements are designed into the system. In contrast, alternative functionality focuses on tolerating failures not with a redundant backup, but by relying on another component or subsystem that provides an alternative function. The system is in a degraded operating mode, but the degradation is a change in functionality rather than a loss of performance.

Researchers in dependable distributed systems define graceful degradation as a combination of performability and real-time quality of service [9, 13, 19]. 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 primarily deals with system hardware resources such as network bandwidth or processor utilization, and focuses on the effects of timing faults and resource overload faults. Our main focus is on how changes in system functionality can compensate for component and subsystem failures.

## 3. System model

Our system model is not the primary focus of this paper and is described briefly here. More information on the details of our model is available in [15] and [16]. The model is based on the system's software architecture. Since we focus on real-time distributed embedded systems, we make several assumptions about the 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 network. Components in the system can be sensors, actuators, or software components. Functional alternatives may be represented by multiple sets of software components or subsystems.

Our system fault model uses the traditional assumption that individual components will be fail-fast and fail-silent, which is best practice for distributed embedded systems. All faults in our model thus manifest themselves as the loss of outputs from failed components. The loss of a failed 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. Software components either provide their outputs to the system or do not. Hardware component failures cause the loss of all software components hosted on that node. Network failures can be modeled as a loss of inputs and outputs between distributed components.

Our system model for specifying functional alternatives 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 system's primary and auxiliary objectives. 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 were to specify the relative utility values of each of these $2^N$ configurations, then we could evaluate the effect of redundancy and functional alternatives on system utility based on the utility differences among different software configurations.

The effort required to specify the system utility function grows exponentially with the number of components in the system, and is clearly infeasible for more than a handful of components. Typical distributed embedded systems may contain hundreds or thousands of software components, sensors, and actuators. However, 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 on the system's functional alternatives.

We focus our analysis on the software view of the model because this view identifies all possible software configurations, the system utility function, and the sets of functional alternatives in the system. In the software view, the software architecture is a data flow graph that shows the dependencies and communication links among software components. In this graph, vertices represent software components, sensors, and actuators, and edges represent *system variables*. System variables are an abstraction of the input and output interfaces of the system's software components. *Feature subsets* represent logical subsystems of components that output sets of system variables. Feature subsets are not necessarily disjoint and can share components across multiple subsets. Feature subset definitions enable the system model's scalable analysis because they encapsulate subsets of components to reduce the complexity of the system utility function. Feature subsets may also represent functional alternatives.

We have applied this model to several distributed embedded system software architectures, including the elevator case study described in this paper, a robot that performs autonomous navigation, and an example automotive navigation system described in [10]. Each of these systems had at least 40 components, which means there were at least $2^{40} = 1.1 * 10^{12}$ possible system configurations to be considered for the system utility function. Using our system model, we were able to completely specify the system utility function by evaluating a total of fewer than 450 feature subset configurations in each system. This was possible with two key insights gained from our model. The first insight was that dependencies among components due to data flow in the architecture greatly reduced the number of valid

configurations that must be considered. In the systems we evaluated, over 90% of the total possible configurations of the system were invalid because missing components broke data flow from sensors to actuators necessary for functionality. The second insight was that we could use the feature subset definitions to form a hierarchical structure that enabled us to calculate system utility based on feature subset component configurations rather than a flat system component configuration. Since feature subsets generally had significantly fewer components than the entire system, this reduced the number of configurations to be manually evaluated. We used this model to aid our analysis of the elevator system and identify where functional alternatives could be applied to achieve dependability improvements.

## 4. Implementing functional alternatives

This section describes our approach for designing functional alternatives, some initial techniques we have identified for applying them to a system architecture, and our evaluation mechanism. The model we have developed identifies all feature subsets in the system and the dependencies among them, which we use as a basis for identifying functional alternatives. The model also enables us to evaluate the relative utility of any configuration of failed components in the system, which means we can determine combinations of component or feature subset failures that may cause greatly reduced system utility or a complete system failure. The model alone does not provide insight on how we should apply functional alternatives in the system to maximize dependability, but it does give us a means to evaluate design choices as to where we allocate resources for functional alternatives or redundancy. We can also use the model as a validation tool to ensure that the configurations evaluated in the architecture model provide their specified relative utility in the system implementation.

### 4.1. Designing functional alternatives

Alternative functionality encompasses many existing redundancy mechanisms. Triplex modular hardware redundancy [14], recovery blocks [12], and multi-version software redundancy [1] are all examples of possible alternative functionality mechanisms that could be applied to a system to improve dependability. However, each of these techniques has a significant cost in terms of resources required in the system as well as design complexity.

Other functional alternatives may be implemented as heterogeneous redundancy. Heterogeneous redundancy can take many forms. One example is analytical redundancy [11], where there may be several related sensors available in the system. These sensors may 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.

The simplex architecture [2] is another technique that can be used for implementing functional alternatives. It is a control system architecture for using design diversity to improve the reliability of a software control system. It explicitly defines tradeoffs between low-performance, more reliable controllers and high performance controllers that may contain more residual design 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. The simplex method specifically targets each alternate algorithm to satisfy different levels of system objectives: one focusing on high reliability, and the other on high performance.

We can also improve the effectiveness of alternative functionality on system dependability by designing individual components (and feature subsets) to be robust to input failures. 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. One approach might be to initially specify the component's outputs 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. This technique assumes that received inputs will not be erroneous, which is compatible with our fail-fast, fail-silent component fault model.

### 4.2. Applying functional alternatives

Each alternative functionality mechanism can potentially improve system dependability by providing redundant functions for satisfying primary system objectives. However, it is not feasible to add alternative functionality to every feature subset in the system. Each additional functional alternative has increased design or resource costs.

Ideally, we would like to identify existing feature subsets that may serve as functional alternatives for primary system objectives with little or no modification. For example, if one feature subset's output is semantically similar to another, it may be a candidate for a functional alternative with the addition of an adapter component to transform its output to match the other feature subset's interface. This process requires domain knowledge to recognize similar interfaces across feature subsets and components. This has the benefit of adding additional redundancy to the system with little or no additional resources, but requires significant analysis effort from the designer. Our model provides a basis for this analysis with all of the feature subset interfaces identified.

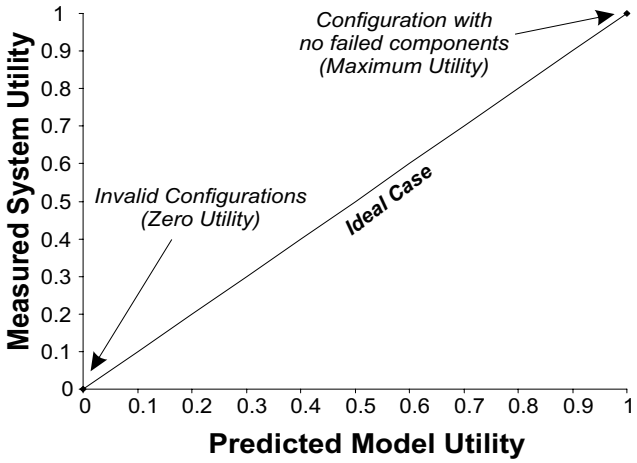


**Figure 1. Graph of ideal case of predicted model utility vs. measured system utility.**

There are several properties in the architecture that indicate which parts of the system could best be augmented with functional alternatives. For example, using the system model, we can evaluate the utility of every configuration with a single failed component or feature subset. If any of these configurations are invalid (provide zero system utility) we have identified a single point of failure, which could benefit from a functional alternative mechanism.

Another approach to identifying where functional alternatives should be installed could be to analyze which system variables are required inputs to a large number of components in the system. The more components that require any one system variable as an input, the more important that variable is to system utility. Therefore, we should provide adapters to increase the number of components and feature subsets that can output that system variable.

## 4.3. Evaluating the system implementation of functional alternatives

In addition to using the model at design time to determine where functional alternatives should be applied in the system, the model can also be used to validate whether or not the system implementation can tolerate the component failure configurations evaluated. In an ideal case a 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 were to graph each configuration's utility from the model versus its measured utility for all $2^N$ configurations, the result should be a straight line with a slope of 1 as shown in Figure 1. 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.

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 of 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 these measurements as a proxy for a system utility metric. If we graph the system configurations by comparing their utility values as predicted by the model and their system property metrics that substitute for system utility measurements, 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. We can apply this analysis iteratively to both refine the system model and identify dependability bottlenecks.

This analysis assumes that the utility values specified by the system model for all $2^N$ configurations are reasonably accurate, and that the properties selected to measure the system implementation are indicators of system utility as defined by the system's objectives. 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 [3, 4].

## 5. Case study: elevator system

To illustrate how we can apply alternative functionality, we use a design of a relatively complex distributed elevator control system. This system was designed by an engineer with industrial experience in elevator architecture (the second author) and has been implemented in a discrete event simulator written in Java as a course project for several semesters. Since we have a complete architectural specification as well as an implementation, we can directly observe how alternative functionality affects the system's ability to tolerate combinations of component failures by performing simulation experiments.

A requirements document specifies each system component's inputs and outputs, as well as its functional behavior. Component interfaces are specified by a message dictionary. We created a system model and used the analysis techniques described in Section 4 to apply functional alternatives and improve the system's dependability. We then ran a set of experiments on the elevator system using imple-

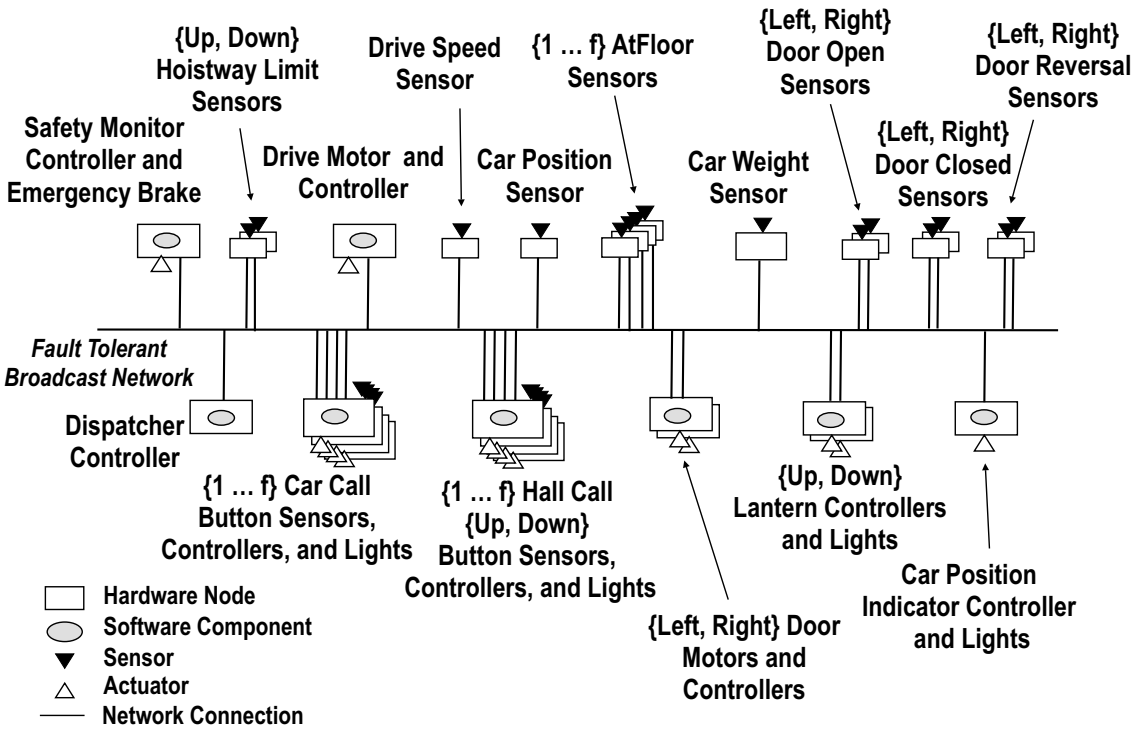


**Figure 2. Original elevator system architecture.**

mentations of both the original system architecture and the new architecture with our alternative functionality improvements. We failed several combinations of components and observed the effect on the system's ability to deliver passengers.

Figure 2 shows a high level view of the original elevator system architecture, with software components, sensors, and actuators allocated to distributed processors connected by a fault tolerant broadcast network. 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 [17].

## 5.1. Applying functional alternatives to the elevator system

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. We specified these requirements as the minimum primary objectives. Other services typically associated with an elevator system, such as providing appropriate passenger feedback, efficiently processing passenger requests, and minimizing passenger travel time, were considered auxiliary objectives. As long as the elevator maintains passenger safety, and can (eventually) service all floors, it can still be considered "working."

Based on the software components defined for this elevator system, the safety, drive control, and door control components are responsible for satisfying the elevator's primary objectives. However, these components are also dependent on the outputs of other system components, such as the dispatcher, button sensors, and floor sensors. We can improve system dependability by using alternative functionality to compensate for failure of these components to preserve operation of the safety, drive control, and door control components.

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 cause a complete shutdown of the elevator system. This ensures that the primary objective of keeping the passengers safe from injury is always satisfied. Unfortunately there is no alternative functionality in the elevator system that can satisfy this objective. Since safety is a critical feature of the system, a redundant component is appropriate for the safety monitor, but not for all of its sensor inputs. A loss of any of the sensor inputs by the safety monitor will by design trigger an emergency shut down.

The drive controller sends commands to the drive motor to move the elevator to different floors and satisfies the primary objective of delivering passengers to desired floors. It is dependent on the dispatcher component to provide the elevator's next floor destination, and the dispatcher is in turn dependent on the hall call and car call buttons to determine passenger requests. We can apply alternative functionality to allow the drive controller to continue to function in the event of dispatcher or button failures. We

designed a default behavior such that it periodically visits every floor. When the dispatcher is working and providing its outputs, the drive controller lets the dispatcher command override its default behavior.

The drive controller also uses 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 command the drive motor to slow if the car position sensor's data is lost. This sacrifices the auxiliary objective of minimizing passenger travel time to guarantee the primary objective of delivering all passengers when the car position sensor fails.

We also redesigned the dispatcher component to implement alternative functionality when there are hall and car call button failures. The dispatcher implements an algorithm to process passenger requests efficiently by listening to button inputs. When a button fails, the dispatcher periodically synthesizes "faked" requests for floors. This guarantees that the primary objective of delivering all passengers will be satisfied and that the dispatcher does not "starve" floors on which all buttons have failed. This alternative functionality uses a trivial computation to substitute for missing sensors. However, when failures occur, the auxiliary objective of elevator performance may suffer because some floors may be unnecessarily visited when there are no passengers desiring that floor.

## 5.2. Experimental setup

We performed a set of experiments using a discrete event simulation of software components, sensors, actuators and a real-time network with message delay that delivers broadcast periodic messages between system components. Sensor and actuator objects interact with simulated passenger objects. 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. The elevator system configuration is determined by setting which components are failed at the start of the simulation.

We tested two hypotheses with these simulation experiments. The first is that the changes we made to the elevator system architecture would actually improve the system's ability to tolerate component failures. We measured this by running simulations of both the original elevator architecture and our improved architecture with functional alternatives, and observing which system more efficiently delivered passengers. The second hypothesis is that our system model would accurately predict the relative utility of system configurations, so that we could use it as a validation tool for the impact of functional alternatives on the system's ability to tolerate component failures and continue to satisfy primary objectives.

We selected a subset of the possible valid elevator system configurations that represented a wide range of possible component failures. 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 was 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, arriving randomly on different floors. Elevator systems usually deal with three types of traffic: two-way, down-peak, and up-peak [18]. 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. For all of our tests, the elevator serviced seven floors.

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 the system is largely constructed of components that are replicated per floor. The dispatcher, car call and hall call buttons are mainly responsible for the elevator's performance. These components are strongly decoupled and provide equal utility contributions to the system per floor. Simulating failures of each button individually, as well as the dispatcher component, gives enough data to determine how well the system tolerates combinations of component failures.

## 5.3. Results

We compared the original and improved elevator systems by measuring how many passengers each system delivered during the simulation runs. A mean of the number of passengers delivered for all 30 passenger profiles for each configuration was used. Every configuration of the elevator with alternative functionality 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.

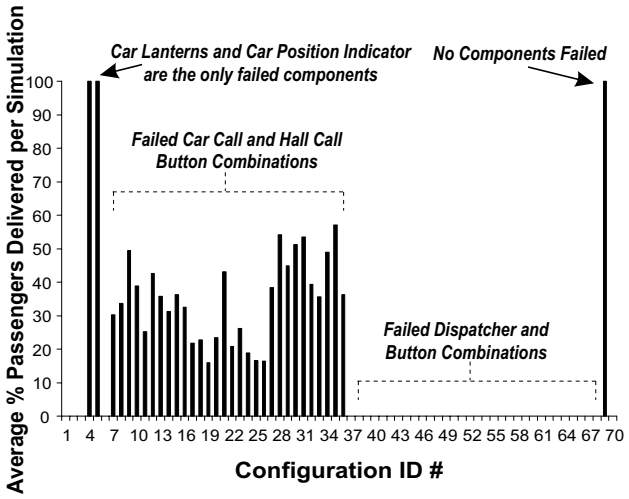


**Figure 3. Average % passengers delivered for the original elevator system.**

Figure 3 shows 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, by chance, to deliver all 50 of its 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 dispatcher failed.

These results show that our system augmented with functional alternatives can tolerate combinations of component failures that would prevent the original system from satisfying its objectives. It is certainly more fault-tolerant than the original system. 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 system with functional alternatives to observe whether the system exhibits a gradual drop in utility as components fail.

In general, system utility should be a measure of how well the system fulfills its objectives, and could incorporate many system properties such as performance, functionality, and dependability. An elevator system's performance objective is to transport people efficiently 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 use total passenger wait time plus transit time as a relatively simple but useful performance metric. In the data we examined, using more complex performance metrics did not significantly affect the relative order of the configurations tested.

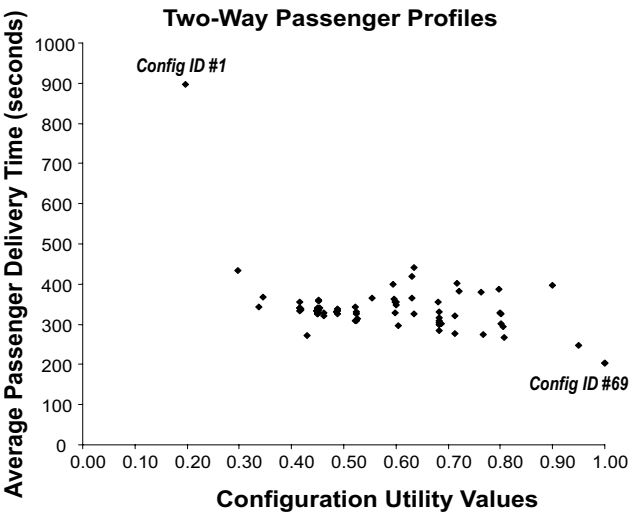


**Figure 4. Average elevator performance vs. model utility for two-way traffic profiles.**

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. If our model accurately predicts system utility, we should see configurations that have higher utility measures achieve better average performance. Figures 4 and 5 graph the utility of the tested system configurations versus the average elevator performance per passenger per simulation for the two-way and up-peak profile types. In these graphs, better elevator performance translates to lower average passenger delivery times, making the vertical axis reversed in direction from the ideal sketch of Figure 1. The 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 configurations that provide more utility.

For the random two-way traffic profiles (Figure 4), the data indicates that the model 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 up-peak traffic profiles (Figure 5), 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. After examining the data, we realized that this is due to

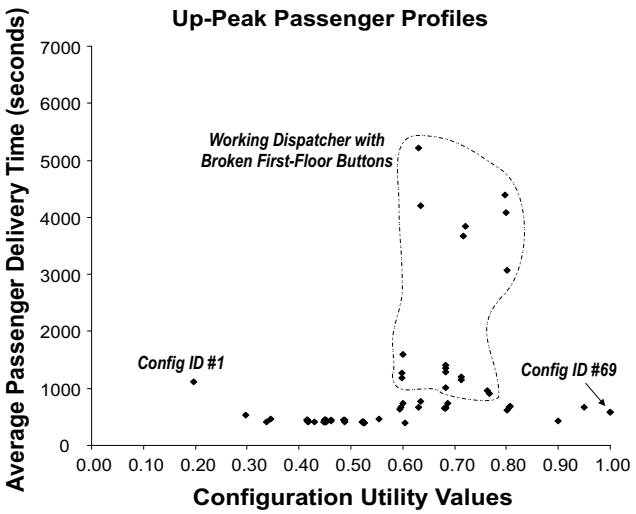


**Figure 5. Average elevator performance vs. model utility for up-peak traffic profiles.**

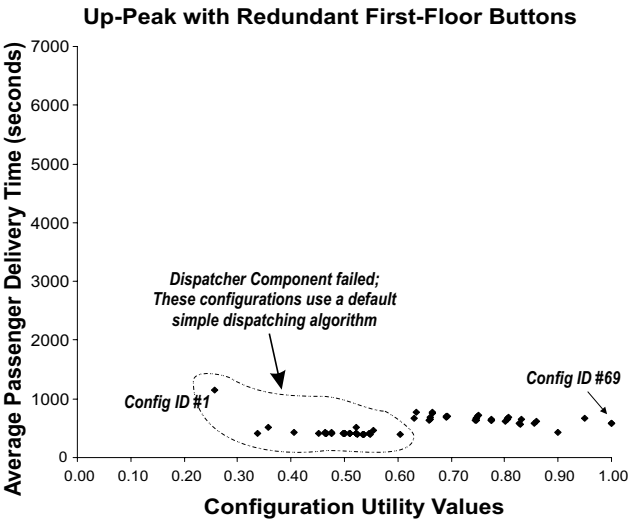


**Figure 6. Average elevator performance vs. model utility for up-peak traffic profiles with redundant first-floor buttons.**

an unforseen interaction between the characteristics of the up-peak traffic profiles and the alternative functionality 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 normal algorithm, that was optimized for two-way traffic.

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 were traced to configurations in which the dispatcher was working but the first-floor hall call button was not. We encountered similar discrepancies with the down-peak passenger profiles, and traced them to the first-floor car call button.

Our utility specification gave equal weights to the utility contributions from all hall call buttons. 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. This result also indicates that a new system objective that specifies that up-peak and down-peak performance should not be sacrificed to optimize two-way traffic performance might be appropriate.

Figure 6 shows the results of experiments with redundant first-floor buttons for the up-peak traffic profiles. 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 configurations had average passenger delivery times as bad as 6,000 seconds.

## 6. Conclusions

This paper introduces alternative functionality as a mechanism for improving system dependability without requiring dedicated backup redundancy. Alternative functionality exploits the fact that some system functions that provide different features may still satisfy the same primary system objectives, although at different levels of service. We view functional alternatives as a generalization of redundancy mechanisms, with a focus on exploiting existing functionality available in the system rather than adding dedicated backup redundancy. Our system model identifies subsystems that can represent functional alternatives and enables analysis and evaluation of the architecture and implemented functional alternatives for dependability improvements.

The experiments we performed on a simulated implementation of an elevator control system revealed that the

original elevator design could only tolerate failures in the car position indicator and car lanterns without failing to deliver passengers. However, our elevator design with functional alternatives could withstand a loss of up to 75% of the system's components and still provide service to all passengers, albeit at reduced performance. Every configuration tested on the improved elevator delivered all passengers in all tests, satisfying the elevator's primary objectives despite a loss of system functionality.

Additionally, when we compared experimental results with our system utility model, we discovered that the first-floor hall call and car call buttons will have a significant impact on system performance for up-peak and down-peak traffic profiles. This led to our decision to incorporate redundant components in only these buttons for a significant utility improvement for many component failure combinations. This indicates that our model and evaluation techniques are useful as a tool for ensuring that a system implementation provides the level dependability expected from the architecture design.

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 take advantage of alternative functionality. 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 than brute-force redundancy or explicitly designing fault tolerance for all possible failure combinations. Properties of the software architecture such as the component interfaces and the identification and partitioning of system functionality into logical subsystems seem to be key to effectively implementing functional alternatives. This case study demonstrates the potential of functional alternatives for improving dependability in distributed embedded system designs.

## 7. 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, the Pennsylvania Infrastructure Technology Alliance, and Lucent Technologies.

## 8. References

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

[2] 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.

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

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

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] Nace, W., "Automatic Graceful Degradation for Distributed Embedded Systems," Ph.D. dissertation, Dept. of Electrical And Computer Engineering, Carnegie Mellon University, May 2002.

[11] 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.

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

[13] 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.

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

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

[16] Shelton, C., "Scalable Graceful Degradation for Distributed Embedded Systems," Ph.D. dissertation, Dept. of Electrical And Computer Engineering, Carnegie Mellon University, August 2003.

[17] Shelton, C., Koopman, P., "Using Architectural Properties to Model and Measure Graceful Degradation," in *Architecting Dependable Systems*, LNCS 2677, pp. 267-289, de Lemos, R. et al. (Eds.), Springer-Verlag, Berlin, 2003.

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

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