WORDS03 – January 2003

A Framework for Scalable Analysis and Design of System-wide Graceful Degradation in Distributed Embedded Systems

Charles P. Shelton
ECE Department
Carnegie Mellon University
Pittsburgh, PA, USA
cshelton@cmu.edu

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

William Nace
ECE Department
Carnegie Mellon University
Pittsburgh, PA, USA
wnace@cmu.edu

Abstract

We present a framework that will enable scalable analysis and design of graceful degradation in distributed embedded systems. We define graceful degradation in terms of utility. A system that gracefully degrades suffers a proportional loss of system utility as individual software and hardware components fail. However, explicitly designing a system to gracefully degrade; i.e. handle all possible combinations of component failures, becomes impractical for systems with more than a few components. We avoid this exponential complexity of component combinations by exploiting the structure of the system architecture to partition components into subsystems. We view each subsystem as a configuration of components that changes when components are removed or added. Thus, a subsystem's utility changes when components fail or are repaired. We then view the system as a composition of subsystems that each contribute to overall system utility. We demonstrate the scalability of our framework by applying it to an example automobile navigation system. Using this framework, we aim to improve system dependability by identifying architectural properties that enhance a system's ability to gracefully degrade.

1. Introduction

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 failures within critical subsystems can be tolerated. A system that can gracefully degrade in the presence of multiple combinations of component failures can be both more robust and more dependable than a system that cannot gracefully degrade. However, designing a complex real-time object-oriented system that can gracefully degrade is a significant challenge. The system must be able to tolerate multiple combinations of system component failures automatically. The traditional method of designing a specific system response to every anticipated combination of component failures will not scale as the system complexity and number of system components increases.

We present a framework that enables scalable design and analysis of graceful degradation for distributed embedded systems. We focus on distributed embedded computer systems because 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. Without cost-effective solutions for making these software systems dependable, systems will either have a high probability of failure or be too expensive to build. These systems typically have multiple compute nodes connected via a real-time fault tolerant network. Each compute node may be connected to several sensors and actuators, and host multiple software components. The software components provide functionality by reading sensor values, communicating with each other through the network, and producing actuator values to provide their specified behavior.

Intuitively, the term graceful degradation means that a system tolerates failures by reducing functionality or performance, rather than shutting down completely. In order for graceful degradation to be possible, the system must have some level of auxiliary functionality; *i.e.*, 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 some properties such as performance, availability, and usability. We must be able to define the minimum functionality the system must have to complete its primary mission, and treat optimized functionality as desirable, but optional, enhancement. For example, a large part of a car's engine control software is devoted to emission control and fuel efficiency. Such a car can lose this functionality and still perform its primary function of transportation from point A to point B.

We define graceful degradation in terms of system utility: a measure of the system's ability to provide its specified 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 with the number of components failed. Utility is not "all or nothing"; the system provides a set of features, and ideally the loss of one feature should not hinder the system's ability to provide the remaining features. Combinations of multiple component failures should only reduce the system utility by at most the sum of each component's individual utility contribution. It should be possible to lose a significant number of components before system utility falls to zero. Ideally no one individual component should cause a complete loss of system utility, *i.e.*, there should be no single point of failure in the system.

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 [6]. 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.

Our framework achieves scalable analysis by partitioning the system into subsystems based on component input and output interfaces. A subsystem is a set of components that coordinate to produce an output for the rest of the system, and can be recursively defined to contain other subsystems. Rather than identify all possible valid configurations of the system (configurations that provide positive utility), we determine just the valid configurations of each subsystem. Since we determine all valid subsystem configurations, any configuration that does not contain enough components to provide working required subsystems can automatically be eliminated as a failed system configuration. Therefore we can focus our analysis on valid configurations of critical subsystems rather than all possible configurations of the entire system.

We demonstrate our framework by applying it to a moderately complex example system. We take its system architecture specified in a product family architecture and use it to identify valid component configurations of subsystems. We then show how our framework achieves scalability by reducing the number of component configurations that we must identify.

2. Related Work

Previous work on formally defining graceful degradation for computer systems was presented in [2]. 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 can satisfy a looser set of constraints that can 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 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 becomes exponentially complex as the number of system components increases.

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. This 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. Graceful degradation is not an explicit goal of these systems, and typically failures are dealt with by having separate backup subsystems available rather than shedding functionality when resources are lost. This approach is a restricted version of graceful degradation, in that it can tolerate 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, and makes coordination among subsystems difficult.

A promising approach to achieving system dependability is NASA's Mission Data System (MDS) architecture [1, 8]. 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 the primary goal. The software is not constrained to a particular sequence of behavior, but rather must determine the best course of action based on the goals it must accomplish. The limitation with this approach is that it is not trivial to decompose goals into subgoals, nor is it clear how to specify goal priorities to resolve goal conflicts without causing a mission failure. Our framework differs from MDS in that we specifically focus on behavior-based subsystems and the coordination among them through system interfaces.

Related to our concept of graceful degradation are the terms survivability and performability. Survivability is a property of dependability that has been proposed to define explicitly how systems will degrade functionality in the presence of failures [4]. Performability is a unified measure of both performance and reliability that tracks how system performance degrades in the presence of faults [5]. Our work differs from survivability in that we are interested in building implicit graceful degradation into systems without specifying all failure scenarios *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.

3. System Framework

Since we are primarily concerned with the scalability of our approach with respect to the software system, we will focus on the software view and utility model in our framework. We will briefly discuss implications of the hardware view and allocation mapping in the conclusions. To achieve scalability, we partition the software components, sensors, and actuators into hierarchical subsystems based on the outputs they produce. We then analyze the utility of each subsystem individually, and analyze system utility based on the composition of subsystems.

3.1. Example System

In order to illustrate our framework we present an example of a hypothetical vehicle navigation system taken from [7]. Figure 1 shows a partial diagram of the navigation software system components as a product family architecture (PFA) data flow graph. In the figure, adapters and features represent software components in the system, and data elements represent the information passed among components. Various sensors on the vehicle produce data that is synthesized into location information. This information is then passed to navigation software that plans a route based on the destination the driver provides. Finally the system dynamically outputs information to the driver about where to turn from multiple actuators such as the turn signal lights on the dashboard, a color map display, or a speaker that provides audio cues when a turn is coming.

In [7] the focus was on hardware reconfiguration for graceful degradation. The product family architecture presented for the example system represented a library of possible software components that could be installed on the system based on the amount of hardware resources available. Our framework takes a different but complementary approach. We want to assess, given a certain software configuration, what possibilities for automatic graceful degradation are achievable for the system without hardware reconfiguration. Therefore we will start by assuming that all software components and features are available in the system, and analyze how many possible system configurations that provide positive utility are possible. For all system configurations, each component can either be working or failed. With 9 sensors, 3 actuators, and 33 software components there are $2^{(9+3+33)} = 2^{45}$ possible system configurations.

3.2. Software View

In order to generate a view of software subsystems, we define a set of *system variables* to capture the input and output interfaces of all the sensors, actuators, and software components. We take the PFA graph of the navigation system and develop a set of subsystem graphs based on data flow among system components. The hierarchical definition of subsystems is straightforward because it is based on the producers and consumers of these system variables. We define these subsystems in our framework as *feature subsets*. A feature subset is a set of components (software components, sensors, actuators, and possibly other feature subsets) that work together to provide a set of output variables. Feature subsets may or may not be disjoint and can share components across different subsets.

The top-level feature subsets are the three subsystems that control the system actuators: the display, turn signal indicator, and speaker. The other feature subsets are contained as components within these feature subsets and recursively defined for all system variables. Dependencies among components and subsystems are also based on system variable definitions. In our navigation example, the major system variables we defined from the input and output interfaces were *GroundSpeed*, *CurrentDirection*, *CurrentLocation* (a combined variable with both the location information and error estimate), *PathInfo*, *TurnInfo*, and *MapData*. These system variables were mapped directly from the data elements labeled in the PFA graph.

Figure 2 shows a portion of the feature subset graph for our navigation system, decomposing the top-level Sound and Turn Signal feature subsets. Lower-level feature subsets can be components of higher-level ones, and feature subsets can also share multiple components. These feature subset graphs can also show dependency relationships between components. Each component may not require all of its inputs to provide some functionality. For example, the TurnInfo feature subset has four software components that represent different algorithms for generating turn information from the *CurrentLocation* and *PathInfo* system vari-

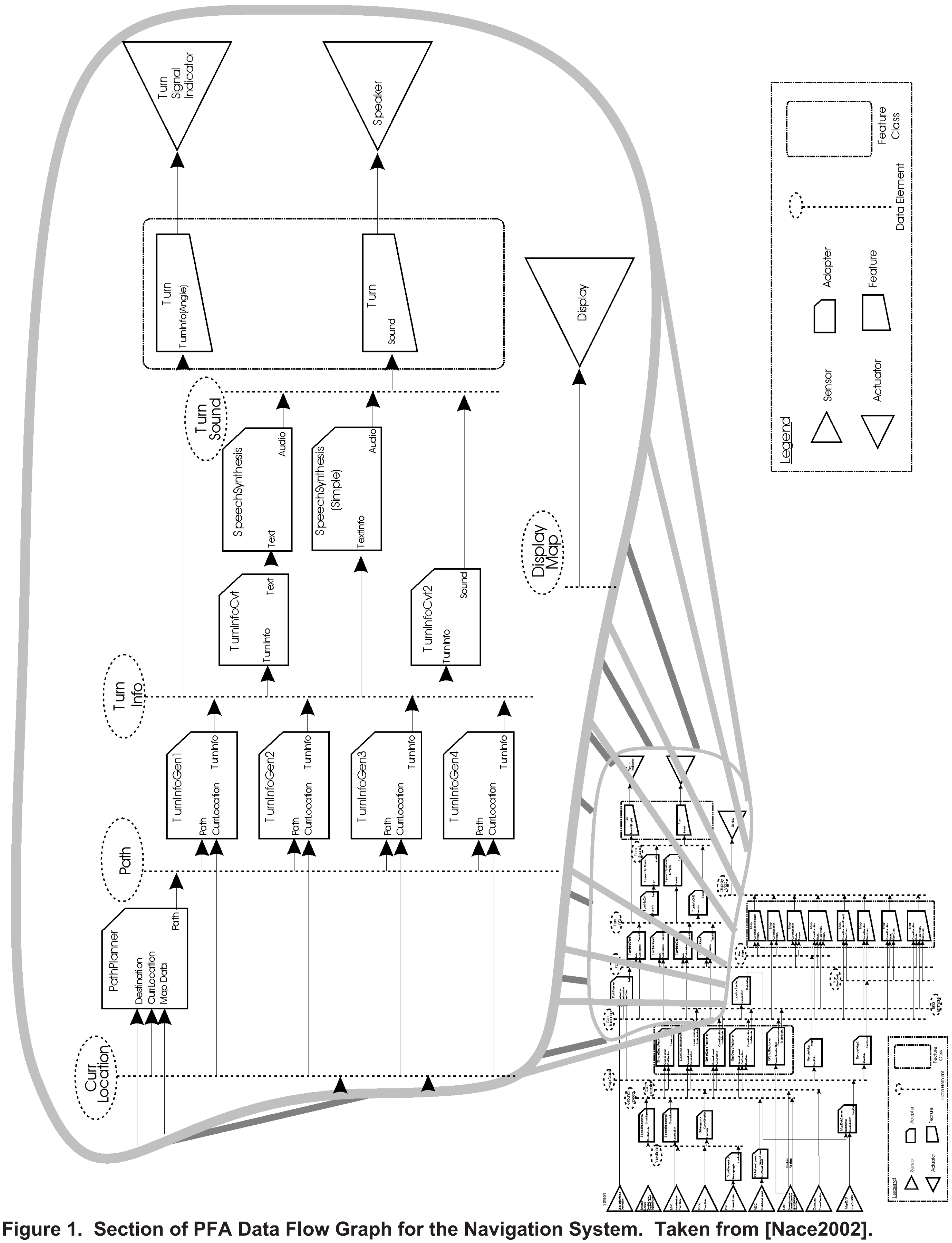

Figure 1. Section of PFA Data Flow Graph for the Navigation System. Taken from [Nace2002].

ables. The TurnInfo feature subset only requires that at least one of these components be present to provide utility to the rest of the system.

We used our framework to completely specify all feature subsets in the navigation system based on the system components' input and output interfaces. Due to length constraints, we did not include all of the feature subset diagrams here. For example, the Location and Path Planner feature subsets contained in the TurnInfo feature subset can be further decomposed into the sets of components that provide the *CurrentLocation* and *PathInfo* system variables as outputs.

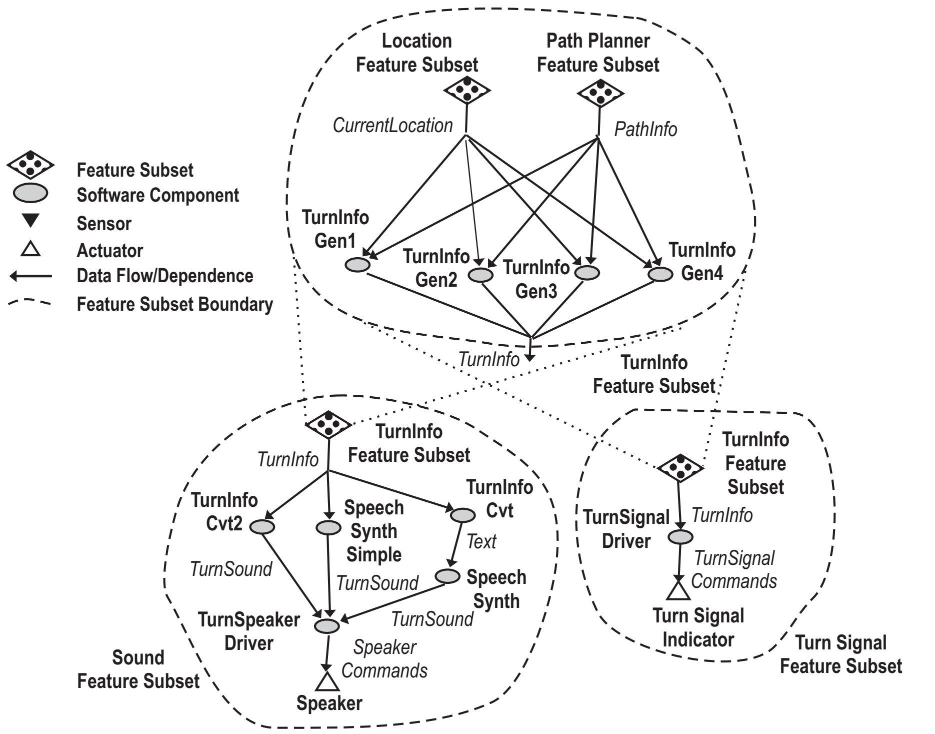

Figure 2. Top-Level Feature Subsets for Sound and Turn Signal Subsystems.

In addition to grouping components into feature subsets, we define a set of dependency relationships between feature subsets and their components. A feature subset may have strong dependence on some of its components, weak dependence on others, and some of its components may be completely optional. A feature subset strongly depends on one of its components if the loss of that component results in the feature subset's having zero utility. A feature subset weakly depends on one of its components if the loss of that component reduces the feature subset's utility to zero in some, but not all, configurations in which that component was working. For example, if there are two components that output a required system variable, loss of both will result in the feature subset having zero utility, but loss of only one or the other will not. If a component is optional to a feature subset, then it may provide enhancements to the feature subset's utility, but there are no instances in which an optional component failure alone will reduce the feature subset's utility to zero. Every valid component configuration of the feature subset where that component is working still provides positive (but possibly lower) utility when that component is broken.

These dependency relationships can also exist among individual components as well, based on their input and output interfaces. A component that requires a certain system variable as an input will depend on the components that provide it as an output. For the sake of brevity we have not detailed these dependency relationships here since our focus is on demonstrating the scalability of our framework, but earlier work in representing an elevator control system with this feature subset graph with these dependencies identified was presented in [10].

3.3. Utility 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 the system. Overall system utility may be related to functionality, performance, or dependability properties such as reliability and safety. We apply multi-attribute utility theory [3] to develop a utility model that provides a framework for expressing the utility of each system configuration as a *utility vector* of functional and nonfunctional attributes. These attributes must be specified by the system designer. For example, system utility may be defined by functionality, reliability, safety, or other properties that are considered important system goals. We can apply the utility model to the data flow graph to determine which configurations of components result in a valid system, and which valid configurations have more utility than others, given that we have a specification for our utility attributes.

In general, utility is a vector that consists of several utility attributes. This utility vector is defined for every component in the system, every feature subset, and the entire system itself based on their current configuration. A component has two possible configurations: working, or not working. A feature subset is a set of components, and its configurations are determined by its power set. For a feature subset with k components, there are 2^k possible configurations. The entire set of system configurations can be designated by the power set of its system components. Each utility attribute is a nonnegative number. A component, feature subset, or system configuration has *zero utility* when all of its attributes are zero. A configuration is *valid* if it does not have zero utility, *i.e.*, the configuration provides some positive value for at least one of the specified utility attributes.

We make a non-trivial assumption 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 attributes, it contributes *at least* zero or positive utility for each attribute to the system or feature subset, and never has an interaction with the rest of the system that results in a loss of utility for any attribute. Thus, working components can enhance but never reduce system utility. At the worst we can assume that if we observe a situation in which a component contributes negative utility to the system, we can intentionally deactivate it.

For a component, feature subset, or system, we specify a utility function for each attribute in the utility vector. Each function is defined for every configuration of the component, feature subset, or system, and is in general nonlinear. These functions are non-trivial and are not easily generated. However, a suitable estimate can be achieved by using expert knowledge of the system domain to rank each configuration with respect to each utility attribute. For a single component, there are only two possible configurations: working or not working. A component has zero utility when it is not working, and some arbitrary values for its utility attributes when it is working. Each component's individual utility attributes must be specified by the system designer.

Calculating the utility of all system component configurations is in the general case exponentially complex. We avoid this complexity by basing the system utility calculation on a function of the utilities of the top level feature subsets, which are in turn functions of the utilities of all system components. Therefore, we can calculate system utility by considering only valid configurations of feature subsets, rather than all system component configurations. Some utility attribute functions, such as those for reliability and availability, may have functions that can be easily defined for feature subsets based on the data flow graph. However, other utility attribute functions, such as those for functionality or safety, might only be qualitatively defined by ranking configurations within feature subsets. Our framework can reduce the complexity of calculating system utility because we no longer have to rank the utility attributes of all system component configurations. We exploit the system's decomposition into feature subsets to significantly reduce the number of utility functions we must define to determine system utility.

3.4. System Analysis

In our navigation example, we partitioned the system into 11 feature subsets. No feature subset had more than 8 components. We have significantly reduced the size and complexity of component configurations that we must analyze by defining these hierarchical subsystems. Further, we generated these subsystems based only on the input and output interfaces of software components. We defined standard system variables based on these interfaces that became the dependancy arcs in our framework.

The navigation system utility analysis can be based on the top level feature subsets: Display, Sound, and Turn Signal. These feature subsets recursively contain the other feature subsets in the system, and encapsulate the details of their utility analyses. Because of this encapsulation, we can determine the utility functions of the configurations of a higher-level feature subset by treating contained feature subsets as components that are working or not working. Once we have identified all possible utility configurations of lower-level feature subsets, we can simply reuse those values when determining the utility of higher level feature subsets without redefining utility functions.

Since we only need to determine utility functions for each feature subset configuration, we reduce the number of configurations we must analyze from the total possible number of system component configurations (2^{45}) to the sum of all feature subset configurations, which is 1200. This number will be further reduced since some feature subsets share components and will be constrained to certain configurations based on whether these components are present or absent. For example, both the Sound and Turn Signal feature subsets must contain a working TurnInfo feature subset in order to provide positive utility.

4. Key Issues

There are several key issues that must be resolved in order to apply our framework for graceful degradation to a large class of distributed embedded systems. We must ensure that our fault model is appropriate for the systems we are analyzing so that we can be reasonably sure that our analysis covers as much of the fault domain as possible. Additionally, we should have a standard representation of data quality or accuracy associated with system variables, so that components that receive system variables can pick the best data from multiple senders. Our methodology also requires a middleware infrastructure that can reliably transmit system variables between software components.

The fault model we use is a major concern for our framework because we want to be able to analyze the system's ability to withstand both hardware and software faults. As a first step, we assumed a fail-fast, fail-silent fault model with perfect fault detection. However, we can relax this assumption based on how we constrain the system architecture and the distributed nature of the system. For graceful degradation we are more concerned with the effects a fault produces rather than the source of the fault, so that the rest of the system can effectively deal with the situation. In order to minimize the failures a fault can produce, as well as minimize fault propagation, we constrain our system architecture to only allow communication among software components via system variables. As long as the implementation adheres to the architecture and does not provide hidden communication channels among software components, faults can only be propagated through the defined system variables.

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

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

Since we have made system variables a key mechanism in our system architecture, we need to standardize how these variables represent their accuracy or quality as a part of their system state. One approach would be to represent data accuracy as a range of uncertainty or confidence interval. This might work well for numerical data types and is flexible in that the accuracy of a system variable can be dynamically updated while the system is running. However, this approach might require a heavyweight analytical model for each producer of the system variable that would be costly to implement. Additionally, this model would not work well for non-numeric system variable data types.

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

Another key assumption in our framework is that system variables can be reliably transmitted from senders to receivers. This requires a middleware infrastructure that is lightweight enough to work in a distributed embedded network, but has data object manipulation features similar to CORBA. Software components know their input and output interfaces, but should not have to be concerned with the sources and destinations of those inputs and outputs. Since we are dealing with embedded networks, bandwidth and real-time deadlines are key issues. Initially, we are assuming that the embedded networks these systems use will be broadcast networks, such as a control area network (CAN), rather than connection-oriented networks. Thus senders of system variables can transmit their data on the bus, and receivers will only listen for messages that have the system variables they require as inputs. However, the amount of required bandwidth will grow with the number of system variables, and we must impose constraints on our system architecture to limit the number and type of system variables provided.

5. Conclusions

Previous best practice for specifying graceful degradation required specifying every possible combination of failures individually, as well as designing a specific system response for each such combination. For complex systems that integrate hundreds or thousands of heterogeneous software components, this is not scalable. Our framework enables scalable representation and analysis of system-wide graceful degradation. With this system model, we can enumerate all valid system configurations, identify which failure modes the system can handle, and compare the relative utility of different configurations, all without specifying every possible system failure scenario. This analysis provides a basis for determining how well a system gracefully degrades.

The framework consists of a software component and subsystem view 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. By grouping components into feature subsets by their input and output interfaces, we can enumerate all valid system configurations without considering every possible configuration. Furthermore, our framework explicitly supports subsystems that are not disjoint and can share components. This allows efficient use of software components, sensors, and actuators while preserving clean partitioning of subsystems.

The parameters of the utility model must be specified by the system designer, but this is an essential difficulty in determining which system configuration is "best." Our framework reduces the complexity of the utility calculations by only defining utility functions for each feature subset rather than for every system component configuration.

In order to determine how well a system gracefully degrades, we need to know how many valid system configurations are possible and the differences in utility among these system configurations. We demonstrated a scalable framework for determining both with the feature subset dependency graphs and utility model. The model is at a high level of abstraction and can be applied to a system's software architecture or detailed design. As long as the system's component input and output interfaces can be abstracted to a system variable set, and the software has a defined set of components, it is straightforward to apply this framework to a system.

Future work on our framework will include the development of a hardware view of the system with a software/hardware allocation mapping. The hardware view includes identification of all hardware processing elements, network topology, and physical sensor and actuator locations. The allocation mapping details which software components are allocated to which processing elements. These views will enable analysis of hardware failures and their effects on the software system, as well as how hardware redundancy effects system dependability. We also plan to refine the utility model and analysis process so that we can quantitatively assess how design choices such as redundancy and diversity affect system utility.

6. Acknowledgments

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

7. References

[1] 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.

[2] Herlihy, M. P., Wing, J. M., "Specifying Graceful Degradation," *IEEE Transactions on Parallel and Distributed Systems*, vol.2, no.1, pp. 93-104, 1991.

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

[4] 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.

[5] 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.

[6] Nace, W., Koopman, P., "A Product Family Approach to Graceful Degradation," *Distributed and Parallel Embedded Systems (DIPES)*, October 2000.

[7] 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.

[8] Rasmussen, R., "Goal-Based Fault Tolerance for Space Systems using the Mission Data System," *2001 IEEE Aerospace Conference*, March 2001, Big Sky, MT.

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

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