Using Architectural Properties to Model and Measure System-Wide Graceful Degradation

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

ABSTRACT

System-wide graceful degradation may be a viable approach to improving dependability in computer systems. In order to evaluate and improve system-wide graceful degradation we present initial work on a component-based model that will explicitly define graceful degradation as a system property, and measure how well a system gracefully degrades in the presence of multiple combinations of component failures. The system's software architecture plays a major role in this model, because the interface and component specifications embody the architecture's abstraction principle. We use the architecture to group components into subsystems that enable reasoning about overall system utility. We apply this model to an example distributed embedded control system and report on initial results.

1. INTRODUCTION

Dependability is a term that covers many system properties such as reliability, availability, safety, maintainability, and security [4]. System dependability is especially important for embedded computer control systems, which pervade everyday life and can have severe consequences for failure. These systems increasingly implement a significant portion of their functionality in software, making software dependability a major issue.

Graceful degradation may be a viable approach to achieving better software dependability. If a software system can gracefully degrade automatically when faults are detected, then individual software component failures will not cause complete system failure. Rather, component failures will remove the functionality derived from that component, while still preserving the operation of the rest of the system. Specifying and achieving system-wide graceful degradation is a difficult research problem. Current approaches require specifying every system failure mode ahead of time, and designing a specific response for each such mode (e.g., [2]). This is impractical for a complex software system, especially a fine grained distributed embedded system with hundreds or thousands of software and hardware components.

In order to evaluate and improve system-wide graceful degradation, we present a component-based system model that provides a means for evaluating and predicting how well a system should gracefully degrade, as well as how graceful degradation influences dependability properties. We base the model on using the system's interface definitions and component connections to group the system's components into subsystems. We hypothesize that the software architecture, responsible for the overall organization of and connections among components, can facilitate the system's ability to implicitly provide the property of graceful degradation, without specifying a response to each possible failure mode at design time. We define a failure mode to be a set of system components failing concurrently. By using the model to measure how gracefully a system degrades, we predict that we can identify what architectural properties facilitate and impede system-wide graceful degradation.

Related to our concept of graceful degradation is the term survivability. Survivability is another property of dependability that has been proposed to explicitly define how systems will degrade functionality in the presence of failures [3]. Our work differs from survivability specifications in that we are interested in building implicit graceful degradation into systems without specifying failure scenarios *a priori*, and having the system "do the right thing" in the presence of component failures. Also, we are focusing on distributed embedded systems rather than on large-scale critical infrastructure information systems.

The remainder of this paper is organized as follows. Section 2 describes our initial system model and key assumptions. Section 3 describes our representative distributed embedded system and its architecture, and applies our model to this architecture. Section 4 includes discussion about the model's predictions, and how they compare to initial fault injection tests we ran with a simulated version of the control system. Section 5 wraps up with conclusions and future work.

2. SYSTEM MODEL

As a first step, we are concentrating on software architecture at a high level of abstraction. Our system model initially focuses on the "functionality" components of the system: software, sensors, and actuators. We make the initial assumptions that individual software components each have their own processing elements, that there is enough network bandwidth to transmit all needed sensor values, and that there are enough system resources to satisfy real-time requirements. These system aspects will all influence system-wide graceful degradation, but we are planning to include them in the model at later stages.

We consider a system as a set of software, sensor, and actuator components. We use the interfaces among components to define a set of *system variables* through which all components communicate. These variables can represent any communication structure in the

software implementation. Actuators receive input variables and output to the environment, while sensors receive input from the environment and output system variables. We assume that components can either be in one of two states: working or failed. Working means that the component has enough resources to output its specified system variables. Failed means the component cannot produce its specified outputs.

The fault model for our system uses the traditional fail-fast, fail silent assumption. All faults are manifested as the loss of system variable communication among components. Components either provide their output variables or do not. Thus, failures can be detected when components do not provide their outputs when specified. This does not account for more complex types of failures such as providing invalid but syntactically correct information, and assumes component failures can be quickly detected. Fault detection and propagation issues are challenging research areas in and of themselves, and are outside the scope of this work. Additionally, since software component failure rates are difficult to identify, we make an initial assumption that all components have approximately equal failure rates.

A key concept in our model is the notion of *utility.* Utility is a measure of how much benefit can be gained from the entire system, a certain subsystem, or an individual component. For the entire system, the overall utility is determined by a nonlinear function of its individual subsystem utilities. Each subsystem's utility is determined by a nonlinear function of its individual component utilities. For individual components, we define a component's utility to be 1 when working and 0 when failed. We assume that if all components are working the system will be at its maximum utility, and if all components are failed, the system will have an overall utility of zero. Thus, a system gracefully degrades if individual component and subsystem failures reduce system utility gradually.

In our model, we initially concentrate on measuring whether the system has zero or positive overall utility by identifying how resistant critical subsystems are to component failures. Determining the functions that quantitatively measure how working components improve subsystem and system utility values is a challenging problem. However, without knowing these functions we can initially make a distinction between a system that is working and has positive but not necessarily maximum utility, and a failed system that has zero utility. In order to make this distinction we must have a clear definition of what "working" means for the entire system. In other words, we must specify what features of the system are necessary for the system to complete its primary functions. In most cases, this is not all the features available in the system. For example, the primary function of a car is to provide transportation. Critical features necessary for the car to continue working include engine and transmission control, brakes, and steering. The power windows, emissions control, air conditioning, and radio provide auxiliary functionality not necessary for the car to complete its primary task, and can be lost without causing a catastrophic failure.

The system can have many different component configurations based on which components are working or failed. If n is the number of components in the system, then there are 2^n different configurations that can be considered. The system's component configuration determines the utility of all its subsystems, and thus the utility of the entire system. An ideal gracefully degrading system is one where a large fraction of these 2^n configurations result in a system with overall positive utility; *i.e.*, the system can tolerate multiple combinations of component failures and still provide useful functionality.

Our first metric for graceful degradation is the system's resistance to complete failure (zero system utility). We determine this value by looking at how many configurations result in a system with positive utility. The ratio of log_2 *[number of valid component configurations] / n* gives a measure of how many configurations will provide utility relative to the total number of system configurations. This value is 0 (only one valid configuration) for a brittle system, and 1 for a perfect system where any component configuration can provide some utility (ignoring the trivial configuration of zero components that results in no system at all).

Clearly, if we had to consider the utility of every possible component configuration individually, then specifying graceful degradation becomes exponentially difficult as the number of components increases. However, we can use the system's software architecture, which defines system software components, input and output interfaces, and connections among components, to group components into subsystems according to the system variables they provide, and thus reduce complexity.

We define these subsystems in our component model as *feature subsets*. A feature subset is a set of components (software components, sensors, actuators, and possibly other feature subsets) that work together to provide a set of output variables. Feature subsets may or may not be disjoint and can share components across different subsets. Feature subsets have utility values based on which of their components are working, and contribute to overall system utility. A feature subset is critical if its functionality is required by the system; *i.e.*, the total system utility is zero whenever any critical feature subset has zero utility. Thus, the system will have positive utility if and only if all of its critical feature subsets have positive utility. If we view the system as a set of feature subsets rather than individual components, then we should only need to consider valid component configurations of critical feature subsets rather than configurations of all system components to determine how well the system gracefully degrades.

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 is not critical to the operation of the feature subset. 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.

We can use this model to develop a space of systems with varying degrees of graceful degradation. At one end of the spectrum, we have extremely "brittle" systems that are not capable of any graceful degradation at all. In these systems, any one component failure will result in a complete system failure. In our model, this would be a system where every component is within a critical feature subset,

Table 1. Sensors, Actuators, and Software Components in the Elevator Architecture

Sensor Type	#	Output Variable	Actuator Type	#	Input Variable	Software Component	#	Output Variable
DriveSpeed	1	DriveSpeed	Drive Motor	1	DriveMotor	Drive Control	1	DriveMotor
CarPosition	1	CarPosition	Door Motor	1	DoorMotor	Door Control	1	DoorMotor
AtFloor	**f**	AtFloor[f]	Emergency Brake	1	EmergencyBrake	Safety	1	EmergencyBrake
HoistwayLimit	2	HoistwayLimit[d]	Car Lanterns	2	CarLantern[d]	Dispatcher	1	DesiredFloor
DoorClosed	1	DoorClosed	Car Position Indicator	1	CarPositionIndicator	VirtualAtFloor	**f**	AtFloor
DoorOpened	1	DoorOpened	Car Button Lights	**f**	CarLight[f]	Lantern Control	2	CarLantern[d]
DoorReversal	1	DoorReversal	Hall Button Lights	**2f-2**	HallLight[f,d]	Car Position Indicator Control	1	CarPositionIndicator
Car Buttons	**f**	CarCall[f]				Car Button Control	**f**	CarLight[f]
Hall Buttons	**2f-2**	HallCall[f,d]				Hall Button Control	**2f-2**	HallLight[f,d]

and each feature subset strongly depends on all of its components. Therefore, every component must be functioning to have positive system utility.

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

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

3. EXAMPLE SYSTEM: A DISTRIBUTED ELEVATOR CONTROL SYSTEM

To illustrate how we can apply our system model to a control system, we will use a model of a relatively complex distributed elevator control system. The complete details of the model have been published in [6], but we will describe a portion of the system and the software architecture here for clarity.

The general requirement for an elevator is that it must safely transport people among floors in a building. The control system has a set of sensors (door opened/closed, elevator speed, button sensors, etc.) for determining the current environment and passenger requests, a set of actuators (door motor, drive motor, emergency brake, lights, etc.) for performing tasks and informing passengers about system state, and a set of software objects (door controller, drive controller, dispatcher, etc.) that implement the control logic to perform the elevator's functions.

Table 1 summarizes the list of sensors, actuators, and software components in the elevator control system. In the table, *f* represents the number of floors in the elevator's building, and *d* represents a choice of two directions, up or down. For example, there are *f* floor sensors and *f* car button sensors (one for each floor), two hoistway limit sensors (the "up" sensor is at the top of the hoistway, and the "down" sensor is at the bottom), and *2f - 2* hall button sensors (two per floor in each direction, except for the top and bottom floors, which only have one button). In the table each sensor has a specified output variable, and each actuator has a specified input variable. The software components have several inputs and a few outputs. There are a total of *14 + 11f* components in the system.

The system's software architecture defines each component's input and output interface, as well as connections among components, which can be used to construct the system's feature subsets. Figure 1 shows the critical feature subsets of the system, and the dependencies between the feature subsets and components. Each arrow in the figure represents a system variable being communicated between components. In an elevator control system, the only critical functions of the elevator are that it must be able to service all floors, open and close the doors, and ensure the safety of the passengers. All other functionality, such as responding to passenger requests, providing passenger feedback, and minimizing wait time and travel time, are enhancements over the basic elevator requirements. Therefore, the critical feature subsets for this system are only the feature subsets that are required to operate the drive motor, door motor, and emergency brake actuators.

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

4. ANALYSIS

In order to derive the graceful degradation metric for our elevator control system, we need only consider the critical feature subsets and the components upon which they depend. Therefore, all configurations containing enough components to provide working Drive

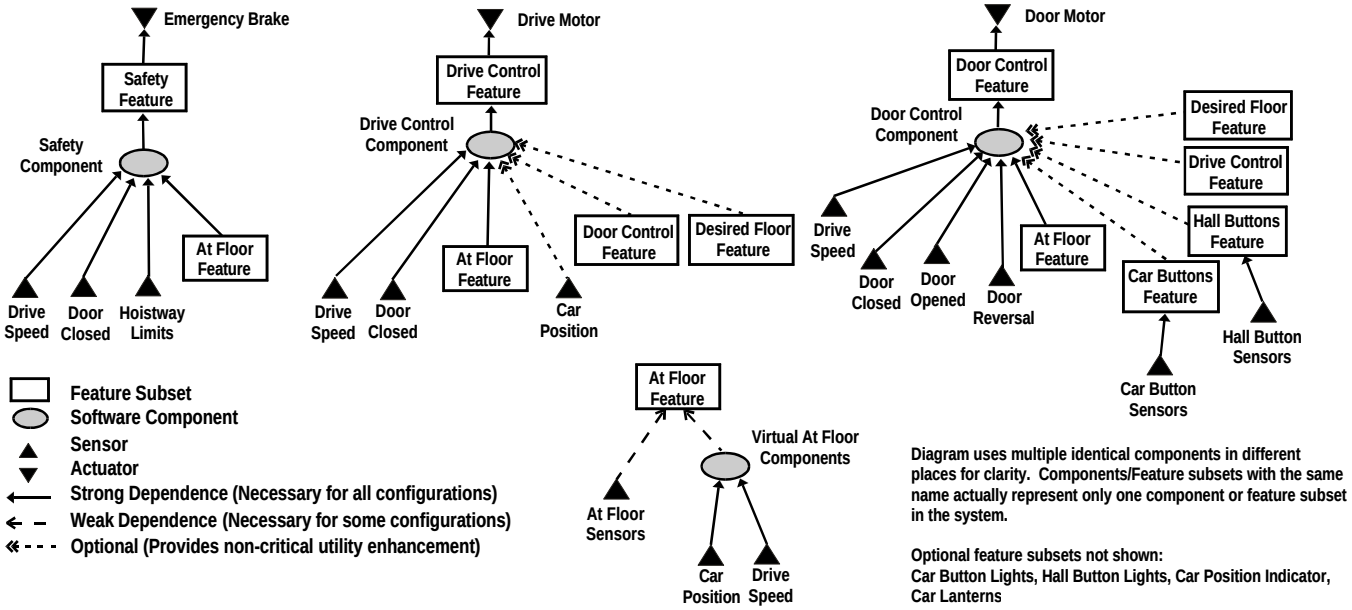

Figure 1. Critical Feature Subsets in the Elevator Control System

Control, Door Control, Safety, and AtFloor feature subsets are valid, and can contain any arbitrary combination of other optional system components. There are $1 + 9f$ optional system components (which can be arranged in 2^{1+9f} different arbitrary combinations), leaving $13 + 2f$ critical components (components within critical feature subsets) that have configurations that require examining (and 2^{13+2f} component combinations left to consider individually).

By examining the critical feature subsets, we can see that they are strongly dependent on the Drive Speed, Door Closed, Door Opened, Door Reversal, and Hoistway Limits sensors, the Drive Control, Door Control, and Safety software components, and the Drive Motor, Door Motor, and Emergency Brake actuators. Any valid configuration must have all of these twelve components present. Therefore, we can restrict the number of configurations we calculate by not considering any configurations in which these components are broken.

This leaves $1 + 2f$ components (the Car Position sensor, the AtFloor sensors, and the VirtualAtFloor software components) in the AtFloor feature subset to be considered. By examining the critical feature subsets, we have systematically reduced the graceful degradation calculation from considering 2^{14+11f} combinations to 2^{1+2f} combinations. Now we can determine the number of valid configurations for the AtFloor feature by noting that all floors must be serviced by the elevator. Therefore, on each floor there must be a working AtFloor sensor or a working VirtualAtFloor component with a working Car Position sensor. If the Car Position sensor breaks, then all AtFloor sensors must work. Since all the AtFloor sensors must work in this situation, they are fixed and have one configuration. However, the VirtualAtFloor components can either work or not work since their failure will not affect the availability of the AtFloor system variables, making 2^f valid combinations for the various VirtualAtFloor components. If the Car Position sensor works, then one or both AtFloor sensor and VirtualAtFloor component must work for each floor, so the only invalid combinations are when both have failed for at least one floor. This means there are 3 valid combinations per floor, making 3^f valid combinations out of the possible 2^{2f}. Thus there are $2^f + 3^f$ valid combinations of components in the AtFloor feature subset.

Multiplying this with the number of combinations of optional components results in a total of $(2^f + 3^f)(2^{1+9f})$ valid component configurations. Taking the base 2 log of this and dividing it by the total number of system components $(14 + 11f)$ gives us our graceful degradation metric. If we calculate this value for an elevator that serves seven floors, we get 0.83.

For comparison, we also consider an elevator system that does not contain any VirtualAtFloor software components. The VirtualAtFloor components improved the system's ability to gracefully degrade because they provided a way to compensate for AtFloor sensor failures by using information provided by other system sensors to synthesize AtFloor sensor values. Therefore, if we remove the VirtualAtFloor components, the resultant system should also receive a lower graceful degradation value.

In our model, the removal of the VirtualAtFloor components reduces the AtFloor feature subset to being strongly dependent on all of the AtFloor sensors. Therefore there is only one valid configuration for the AtFloor feature subset in which every AtFloor sensor must work. Since this is a critical feature subset, all valid system configurations must contain a working AtFloor feature subset. Additionally, the Car Position sensor becomes an optional system component because the AtFloor feature subset no longer depends on it. This results in there being only 2^{2+9f} valid system configurations since most of the components in the critical feature subsets must work and only the optional components can have multiple valid configurations. The total number of system components is also reduced by the removal of the f VirtualAtFloor components, leaving $14 + 10f$ total system components. For a seven-floor elevator, this results in a graceful degradation score of 0.77.

The graceful degradation metric provides a concrete comparison among similar systems. We can quantitatively assess how adding or subtracting components to the system affects its ability to gracefully degrade. However, this metric may be misleading when comparing two systems that are substantially different in terms of functionality and number and type of system components.

We have developed a discrete event simulator that implements our elevator architecture, and have run some initial fault injection ex-

periments to evaluate whether the implemented system actually gracefully degrades. So far, every test we have run with one of the possible valid configurations was able to successfully deliver all passengers to their destination floors, including a test that failed all components but the critical ones and the AtFloor sensors.

5. CONCLUSIONS AND FUTURE WORK

We have demonstrated a component-based system model that can provide insight into how well a system will perform graceful degradation in the presence of multiple component failures. We developed an initial metric for graceful degradation that indicates how many combinations of component failures can be tolerated by examining critical subsystem configurations rather than considering every possible system component configuration. In some initial experiments on a simulated implementation of the example control system studied, we found that the architecture described was resistant to certain combinations of component failures, as predicted by the model.

We did not incorporate failure recovery scenarios for every possible combination of component failures, but rather built the software components to the architectural specification. The individual components were designed to ignore optional input variables when they were not available and follow a default behavior. This is a fundamentally different approach to system-wide graceful degradation than specifying all possible failure combinations to be handled ahead of time.

Properties of the software architecture such as the component interfaces and the identification and partitioning of critical system functionality from the rest of the system seem to be key to achieving system-wide graceful degradation. The model we developed illustrates how well a system can gracefully degrade by using the software architecture's component connections to decompose the system. We are also exploring how to use an architectural description language such as Acme [1] to provide rigorous component interface specifications and facilitate development of our system model.

For this particular system, it is relatively easy to calculate the possible valid configurations by examining the software architecture without the model. However, the model provides a systematic framework for partitioning the system based on its software architecture, and we hypothesize that it will be useful in evaluating any architectural specification that has a well-defined component interface. This framework allows us to measure the graceful degradation properties of individual feature subsets with respect to their components as well. The architecture and model also identify a set of critical system components within the critical feature subsets that must continue to operate to provide any system functionality. This can be used to determine on which system components to spend effort ensuring component reliability through redundancy and other fault tolerance measures.

Our next step is to extend this model to incorporate the allocation of the software components to hardware units. In a distributed system, components that communicate via the network are strongly dependent on the network for their required input variables, making the network a single point of failure. Also, software components are strongly dependent on the hardware node on which they are hosted. These constraints will surely influence a system's ability to gracefully degrade (hardware failures might remove multiple components simultaneously), but may be ameliorated by system-wide reconfiguration as proposed in [5]. Additionally, we want to further develop the concept of system utility to not only distinguish between when the system is "broken" or "not broken," but also different levels of functionality available in different system configurations. We have identified which configurations result in systems with positive utility, but we also need to quantitatively determine which of those configurations have higher utility than others. This will be based on determining which features are more useful than others based on measures such as performance and functionality.

6. ACKNOWLEDGMENTS

This work was supported by the General Motors Satellite Research Lab at Carnegie Mellon University, and Lucent Technologies. Thanks to Beth Latronico, Bill Nace, Orna Raz, and Yang Wang for their help in developing the ideas presented.

7. REFERENCES

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

[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] 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.

[4] Laprie, J.-C., "Dependability of Computer Systems: Concepts, Limits, Improvements", *Proceedings of the Sixth International Symposium on Software Reliability Engineering*, Toulouse, France, Oct. 1995, pp. 2-11.

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

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