Workshop on Reliability in Embedded Systems (in conjunction with SRDS), October 2001.

Developing a Software Architecture for Graceful Degradation in an Elevator Control System

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

Many embedded systems have high safety and dependability requirements, which makes ensuring software robustness a top priority in these systems. As embedded computer systems become more complex and incorporate increasing functionality, their software systems become increasingly more difficult to design, build, and maintain. One approach to achieving software robustness is graceful degradation. However, graceful degradation is a difficult property to define or construct. Traditional hardware redundancy is not enough to achieve software safety and dependability. The system's software architecture may be the key to building graceful degradation into a software system. This paper describes a proposal for a software architecture that may enhance graceful degradation for an example elevator control system, and discussion about implementing and evaluating the architecture.

1. Introduction

Embedded computer systems continue to present difficult design challenges due to tight constraints, such as real-time deadlines, limited hardware resources, and strict safety and dependability requirements. As these systems become more complex, their functionality is increasingly implemented in software. Unfortunately, guaranteeing the safety and dependability of complex software systems is a difficult problem that has yet to be solved. Software robustness, where robustness is defined as the ability to perform correctly in the presence of exceptional conditions or stressful environmental conditions [IEEE90], contributes greatly to achieving those safety and dependability system properties.

One approach to achieving software robustness is graceful degradation. Graceful degradation is the property that individual component failures reduce system functionality rather than cause a complete system failure. However, current practice for achieving graceful degradation requires a specific engineering effort enumerating every failure mode to be handled at design time, and devising a specific procedure for each failure [Herlihy91]. However, in a fine-grained distributed software system, enumerating all possible hardware and software failure modes may be intractable, and certainly not feasible for cost-sensitive embedded systems with limited design times. It is desirable to explore a way to design the system structure in such a way that it can shed non-critical functionality automatically in the presence of failures without having to specify every possible failure and corrective action ahead of time.

Software architecture's high level system abstraction may be the key to building graceful degradation into the software system. Software architecture is defined as "the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationship among them [Bass98]." If reusable software components become more prevalent, software design will become more a problem of composition rather than synthesis. Therefore the overall system structure and decomposition governs how the components work together and provide system functionality. If an architectural style that enhances the property of graceful degradation could be developed and successfully applied to different domains of embedded systems, the design effort for these systems would be able to leverage component reuse while preserving safety and dependability requirements.

This research is a part of the RoSES (Robust Self-configuring Embedded Systems) project. RoSES takes the approach of using automatic reconfiguration to achieve graceful degradation in the presence of component failures [Nace2000]. The software architecture of the system defines the interfaces between components, and valid component configurations of the system. Thus, a properly constructed architecture that can inherently shed functionality in a controlled manner will aid a reconfiguration manager when reallocating components after a failure to provide maximum functionality. A step in this direction is to achieve these properties for an example system.

The remainder of this paper describes a proposal for a software architecture for an example elevator control system. An elevator control system is a sufficiently complex distributed embedded system, and by developing an architecture for graceful degradation for this system, we hope to gain insight on an architectural style for embedded systems that addresses their constraints, and specifically promotes the property of graceful degradation.

2. Elevator System Model

An elevator is a complex distributed control system. It has strict safety requirements: it cannot crush people between the doors; it cannot travel at unsafe speeds in the hoistway; and it should not trap people in the elevator.

We use a model for an elevator system based on a set of sensors and actuators and values they can send and receive. This system does not address some of the more complex elevator features, such as modes of operation like fire response modes, maintenance modes, and up-peak and down-peak modes, but is rich enough to be interesting.

The elevator consists of a single car in a hoistway with access to a set number of floors. The car has a single door and door motor, a drive that can accelerate the car to two speeds (fast and slow) in the hoistway, and an emergency stop brake for safety. In the notation below, the values within the "[]" brackets represent the standard replication of an array of sensors or actuators, and the values within the "()" parentheses represent the values the sensor or actuator can output. For example, the AtFloor sensor is an array of sensors that is f (the number of floors the elevator services) by d (the direction the car is going; Up, Down, or Stop) wide, where each element of the array can either be a value v of True or False. When the elevator approaches a floor f, it can either be traveling up or down, so there is an AtFloor sensor for each floor to indicate whether the elevator car is approaching the floor from above (Down) or below (Up). When the car is close enough to be level with the current floor, the AtFloor[current floor, Stop] sensor becomes True.

The sensors available in the system include:

- **AtFloor[f,d](v)**: Floor proximity sensor. f = floor, d = {Up, Down, Stop}, v = {True, False}
- **CarCall[f](v)**: Car call buttons. f = floor, v = {True, False} All located in the car.
- **DoorClosed(v)**: Door closed switch. v = {True, False} Indicates True when door is fully closed.
- **DoorOpen(v)**: Door open switch. v = {True, False} Indicates True when door is fully open.
- **DoorReversal(v)**: Door reversal sensor. v = {True, False} Indicates True when door senses an obstruction in the doorway.
- **HallCall[f,d](b)**: Hall call buttons. f = floor, d = {Up, Down}, b = {Pressed, Idle} Located in hallway on each floor.
- **HoistwayLimit[d](v)**: Safety limit switches in the hoistway. d = {Up, Down}, v = {True, False} Indicates True when the car has overrun the top or bottom hoistway limits.
- **DriveSpeed(s,d)**: Main drive speed sensor. s = {Fast, Slow, Stop}, d = {Up, Down, Stop}

The actuators available in the system are:

- **DoorMotor(m)**: Door motor. m = {Open, Close, Stop}
- **Drive(s,d)**: 2-speed main elevator drive. s = {Fast, Slow, Stop}, d = {Up, Down, Stop}
- **CarLantern[d](k)**: Car lanterns. d = {Up, Down}, k = {On, Off} Up/Down lights on the car doorframe used by passengers to determine current car direction.
- **CarLight[f](k)**: Car call button lights. f = floor, k = {On, Off} Lights inside the car call buttons to indicate when a floor has been selected
- **CarPositionIndicator(f)**: Position indicator in car. f = floor
- **HallLight[f,d](k)**: Hall call button lights. f = floor, d = {Up, Down}, k = {On, Off} Lights inside hall call buttons to indicate when passengers want the elevator on a certain floor
- **EmergencyBrake(b)**: Emergency stop brake. b = {On, Off}

This is far from a complete description of a modern elevator, but is a fairly complex model that has resources to provide several possible levels of operation. The next section discusses our approach to building a software architecture for an elevator control system using this model that aims to provide graceful degradation as a system property.

3. Proposed Software Architecture

Since graceful degradation emphasizes system survivability in the face of internal component errors, it is probably necessary to have few dependancies between individual software components, especially for critical functions. However, this would mean that every software component would need to have its own direct connection to each hardware sensor and actuator necessary to do its work, and it is unlikely that every task in the system would require only one software component without coordination with other components in the system.

A fine-grained distributed system will require that components work together to complete tasks, so to make graceful degradation work we should couple components as loosely as possible, and make components autonomous when feasible, or when the function is especially critical.

The key idea behind this elevator control system architecture is partitioning the system functionality into critical and non-critical components. We specify a base level of functionality in the system that is the least level of operation before we declare the system "broken." If we make the components that handle these key tasks autonomous and highly fault tolerant, then we can treat the additional non-critical components and functionality as enhancements to the core set. As long as these components do not violate any of the constraints of the core set, when they fail, these components can be removed from the system while still preserving at least the base functionality.

To apply this idea to our elevator model: An elevator at its basic level must only be able to move between floors in the hoistway, open and close the doors at each floor, and ensure passenger safety. The elevator can move slowly in the hoistway, stopping at each floor to admit and release passengers, opening and closing the doors at each floor, ignoring all passenger requests and not providing any passenger feedback, and still be an elevator, albeit a very inefficient one (this is in fact an operating mode used in real elevator systems when passenger request information is unavailable). All other functionality in the system; processing passenger requests, providing passenger feedback, and only stopping at desired floors, are enhancements to increase efficiency.

In our architecture the components that provide the minimum functionality are the Door Control, Drive Control, and Safety components (Figure 1). Note that they all have direct connections to the sensors they require for correct operation and do not communicate with each other. The Door Control must know when the Drive is moving and when the car is stopped at a floor to determine when to open the door, so it has access to the relevant sensors. The Drive Control must know when the doors are completely closed and when to stop at a floor. It should also have access to the Hoistway Limit sensors for internal safety checks. The safety object must be able to detect when and if the drive and door perform any unsafe actions, such as moving when the door is open, opening the door between floors, or moving the car past the hoistway limits. These represent "logical" sensors in the software architecture as they can be either multiple I/O channels to the same physical sensor in the system, or multiple different physical sensors for each software component. This choice represents a cost/reliability tradeoff that does not affect the software architecture. The software components will have the same interfaces to the sensors regardless of the physical configuration. The replication of sensors for these critical components would prevent single points of failure that might lead to catastrophic system failures. If any of these components fail in the system, the system must fail safe and no longer be operational.

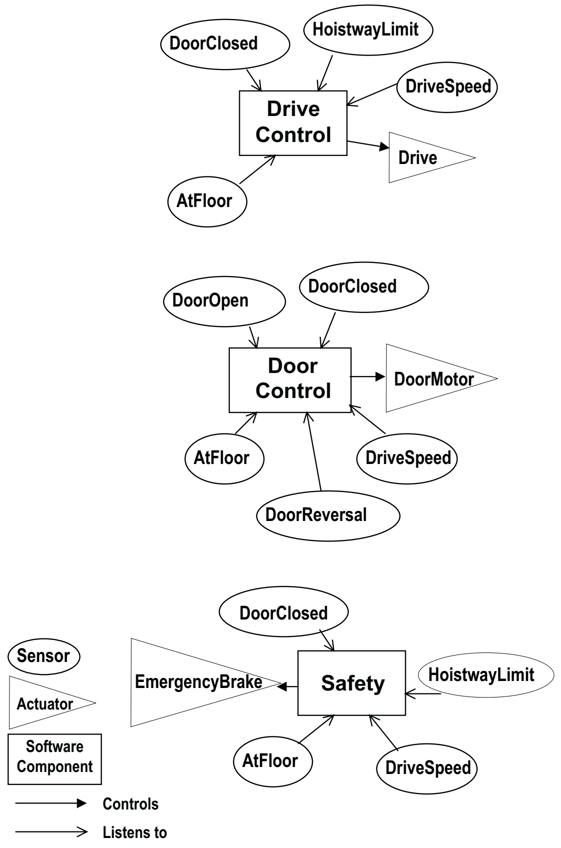

Figure 1. Critical Components in our Elevator Software Architecture

Above this base configuration we can add a real-time network bus for component communication and coordination. Then we add hall button and car button controllers to manage user input from passengers; the car lantern controllers and car position indicator controller for user feedback; and a dispatcher component to schedule efficiently the elevator car's destinations (Figure 2). With the addition of the real-time network, each additional component need not have a direct connection to each sensor, and control state may be passed between components as "advice" for the critical controllers to increase functionality. An example advisory command would be that the Dispatcher component could notify the Drive controller that the next floor where passengers are waiting for elevator service is floor six, and to stop at that floor. If the car is at floor two, the Drive controller can decide to skip the floors between and go directly to floor six. The Drive controller still decides when it is safe to leave floor two, and cannot be commanded to move by the Dispatcher, only receive advice about where to stop. If the Dispatcher stops sending out advisory commands, the Drive controller can declare the Dispatcher failed and default to its base functionality of stopping at each floor. If any non-critical component fails,

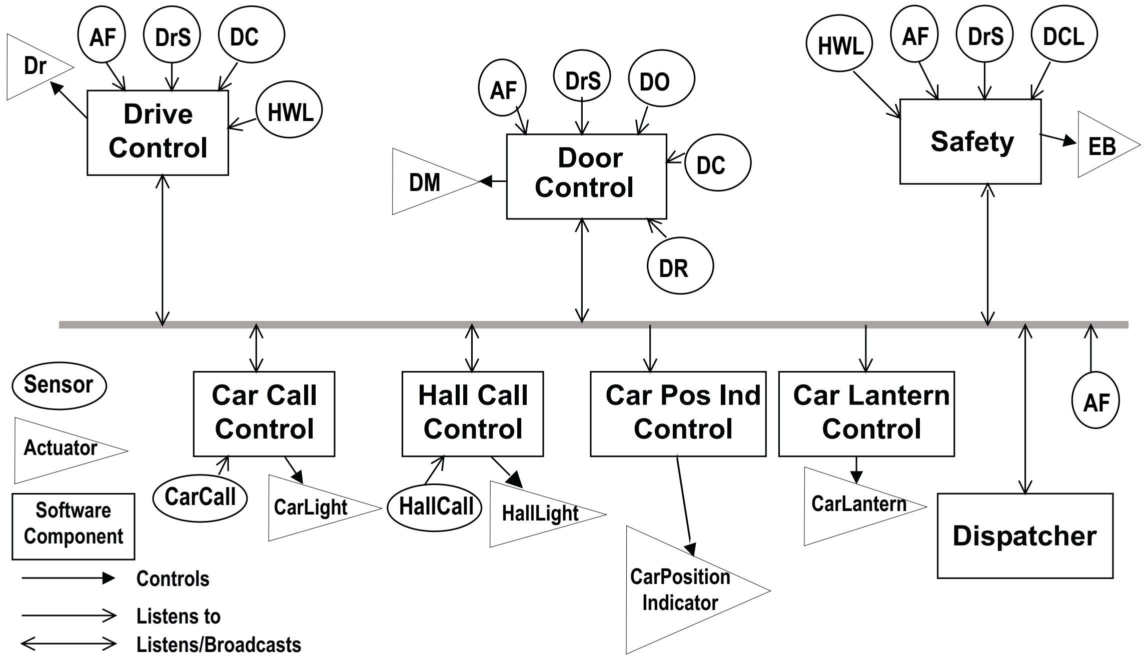

Figure 2. Complete Elevator Software Architecture

it should not interfere with the operation of any critical component or, ideally, none of the other non-critical components. The standard assumption is that these components will "fail silent," i.e. stop sending messages.

One of the major challenges to implementing this scheme is how individual components determine other components' or their own failure. If components fail silent, that can be detected via a timeout, but it is more difficult to determine when a component is broken and is sending incorrect messages and providing misinformation.

Another challenge is how to verify that the non-critical components do not violate constraints of the critical core components. Our approach is to specify a tight interface in the form of a well-defined data dictionary of all the messages that can be sent between components, and ensure that as long as components adhere to this interface, they will preserve the constraints.

4. Conclusions and Future Work

The software architecture described above has several properties that should provide graceful degradation and may be extensible to other types of systems. The main approaches behind this architecture are: specifying a core set of minimum functionality that defines working operation; making the components that handle this core functionality as autonomous and fault tolerant as possible; ensuring that the interfaces between components are well defined and do not violate internal component constraints; and treating the non-critical components in the system as enhancements that provide advice to the core components to improve efficiency. Safety is the explicit responsibility of one component, but other components such as the drive and door controllers have access to relevant sensors for internal safety checks as well. This eliminates single points of failure in the control system.

There are several challenges to implementing an architecture with these properties. It may not be feasible or possible to partition functionality in a fine-grained distributed embedded system so that all critical functions can be performed by autonomous components. Components may fail in ways such that they continue to send incorrect but not invalid messages, making it more difficult for other components to detect failures. Such failure modes require byzantine fault tolerance that may not be feasible in a resource limited real-time embedded system. Additionally, there must be a metric designed to evaluate the system and determine if it really achieves graceful degradation.

The elevator model described in Section 2 has been implemented in a simulation package that has been used in several iterations in embedded systems classes at Carnegie Mellon University. We are currently implementing the architecture for this control system in simulation, and investigating how to model failures and evaluate graceful degradation.

5. Acknowledgements

This work was supported by the General Motors Satellite Research Lab at Carnegie Mellon University, and Lucent Technologies.

6. References

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

[Herlihy91] Herlihy, M., Wing, J., “Specifying Graceful Degradation,” *IEEE Transactions on Parallel and Distributed Systems*, 2(1):93-104, January 1991.

[IEEE90] IEEE Standard Glossary of Software Engineering Terminology (IEEE Std610.12-1990), IEEE Computer Soc., Dec. 10, 1990.

[Nace2000] Nace, W., Koopman, P., “A Product Family Approach to Graceful Degradation,” DIPES 2000, Paderborn, Germany, October 8-19, 2000.