A PRODUCT FAMILY APPROACH TO GRACEFUL DEGRADATION[†]

William Nace
Philip Koopman

Abstract

Design of gracefully degrading systems, where functionality is gradually reduced in the face of faults, has traditionally been a very difficult and error-prone task. General approaches to graceful degradation are typically limited to re-implementation of the system for a number of pre-designated fallback configurations. We describe an architecture-based approach to gracefully degrading systems based upon Product Family Architectures (PFAs) combined with automatic reconfiguration.

A PFA is a region of a system design space populated by different, but related, products sharing similar architectures and components. Each system instance within a PFA yields a distinct price/performance point, and represents a different model in the product family. The unifying mechanism that joins PFAs and gracefully degrading systems is automatic reconfiguration – in the face of a fault, the system reconfigures to a different PFA configuration point that optimizes the functionality available with the remaining resources. In this process, the system sheds some of the non-critical functions that make up such a large percentage of modern embedded systems. System designers can also exploit a reconfiguration mechanism to provide graceful upgrade and unique logistical benefits. The RoSES (Robust Self-configuring Embedded Systems) project employs such a reconfiguration approach, seeking to create a revolutionary means to build self-customizing, distributed, embedded control systems.

[†] This research is supported by the General Motors Satellite Research Lab at Carnegie Mellon University and Robert Bosch GmbH.

Proceedings of DIPES 2000, International IFIP WG 10.3 / WG 10.4 / WG 10.5 Workshop on, Distributed and Parallel Embedded Systems, Paderborn University, Germany, October 18th+19th, 2000.

1. Introduction

Embedded applications such as transportation systems, power distribution, telecommunications, construction equipment and weapon systems are moving toward highly distributed implementations. As a result, traditional centralized approaches are being replaced by federated systems in which many processors collaborate to provide system functionality. This trend certainly is not universal, as integration is another common architectural style – especially in avionics [1]. However, the very modular nature of such integrated systems allows the same concepts to apply within subsystems, as well. If the promise of MEMS (Microelectromechanical Systems) devices based on standard semiconductor process technology comes to fruition, it will soon be possible for most sensors and actuators to have their own inexpensive integrated microcontrollers, accelerating the trend toward federated systems.

A particularly demanding pair of requirements for many distributed embedded systems is that they be both inexpensive and dependable. Fortunately, distributed systems have an inherent capability to spread functionality across many nodes. While it may be that brute-force redundancy is the only way to satisfy stringent reliability requirements for critical functions, not every function is critical. In fact, much of the increasing computing power in embedded systems provides extra functionality or performance optimization rather than basic critical functions. It may be acceptable for optimization functions to be shed by a system as components fail, so long as this is done in a safe and controlled manner. For example, losing a few percent of fuel economy is probably preferable to a complete vehicle failure.

Thus, there is room in many embedded systems to implement graceful degradation of functionality as a way to improve dependability for non-critical (but highly desirable) functions. A gracefully degrading system is one in which faults are masked and only manifest themselves in a reduced level of system functionality.

In fact, a few systems implement graceful degradation today, but use labor-intensive development techniques that often involve specific engineering efforts for every anticipated failure mode [4]. As an example, a car transmission controller might be able to substitute for a failed engine controller, but do so with only very simple and inefficient engine operation. Such traditional approaches usually accomplish graceful degradation using a combination of replication and failover algorithms. Alternative approaches include multi-version redundancy and load sharing. The former is too expensive for non-critical functionality, while the latter usually provides only graceful performance degradation for a fixed set of functionality, potentially causing problems in real-time systems.

We propose that graceful degradation should not be treated as a failover design problem, but instead as an exercise in designing a product family architecture (PFA). A PFA is a region of a system design space populated by different, but related, products sharing similar architectures and components. Each system instance within a PFA yields a distinct price/performance point, and represents a different model in the product family. The collection of system instances and the relations between them form a graph or lattice, an example of which is shown in Figure 1. The concept of a PFA is familiar to anyone who has purchased a stereo, computer or automobile. However, optimization for product families is typically done assuming a perfectly working system rather than with an eye toward graceful degradation.

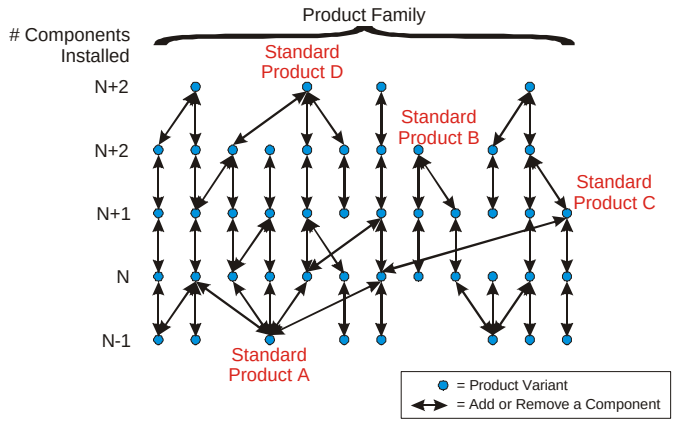

Figure 1: A PFA Lattice

Consider a product implemented by assembling dozens or even hundreds of different "smart" components (*i.e.*, components incorporating microcontrollers) into a fine-grain distributed embedded system. There may be a huge number of different product instances possible. And, if a suitable way to allocate functionality can be provided, any system in which a single component breaks can be treated simply as a closely related system in the PFA that (using a fail-silent assumption) just happens to differ in having that failed component missing from it. Thus, PFAs can form a conceptual framework for specifying and implementing graceful degradation within highly distributed embedded systems.

RoSES (Robust Self-configuring Embedded Systems) is a new research project whose goal is to create architectures for automatic graceful degradation in embedded systems. A discussion of RoSES follows in Section 2. Generic reconfiguration mechanisms we believe critical to such a PFA framework are discussed in Section 3. Section 4 discusses some interesting logistical opportunities made available with a good reconfiguration mechanism. We do, however, believe there are some very difficult problems with reconfiguration mechanisms that might preclude their ubiquitous use. Such problems are explored in Section 5.

2. Reconfiguration in RoSES for graceful degradation

RoSES is a newly created research project investigating a PFA-based approach to obtain graceful degradation and other significant benefits, initially on automotive applications. The basic concept is to represent a system as a set of:

- System requirements with associated utility functions (critical functions are mandatory; others have various quantified utility levels) that form a lattice of acceptable systems,

- System constraints such as network schedules, or task deadlines,
- Abstract functional blocks that satisfy various requirements (with associated software modules),
- Hardware resources, including "smart" sensors, "smart" actuators, compute-server nodes, and one or more embedded networks such as a CAN (Controller Area Network) bus, and
- A binding between software (representing a selected subset of functional blocks) and hardware that forms a particular point within a PFA space, optimizing utility given available resources.

A RoSES system is a generic runtime architecture that works by providing a particular optimum *configuration*, which involves selecting a subset of possible software modules, allocating them the hardware resources, and ensuring that the resultant system meets real time constraints without overflowing system size or bandwidth limits. In order to match standardized hardware and software components to a large variety of system configurations, RoSES uses *mobile object adapters*. Such adapters form a flexible software interface middleware layer between, on one side, the basic functionality of the sensor/actuator and, on the other side, a dynamic network object interface. The role of the adapters within the RoSES system concept is illustrated in Figure 2.

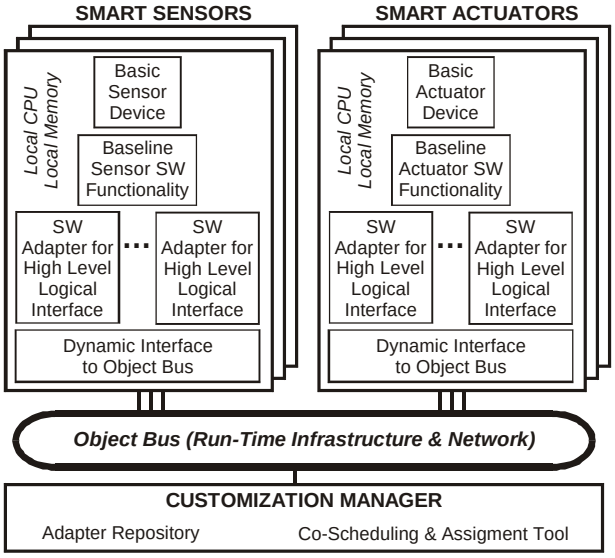

Figure 2: The RoSES System Concept

Once a particular configuration is established, a component failure (either hardware or software) triggers a system reconfiguration. The RoSES reconfiguration concept is a fairly fine-grained one, involving specific software modules/objects and potentially very small hardware components such as single sensors or actuators. The reconfiguration process is, at its core, a search through different combinations of mobile object adapters for the sets that can be used on currently available hardware

resources. Such combinations must be viable (supported by hardware and involving available software), must meet critical system requirements, must not violate any system constraints, and must provide optimal utility given available resources.

Eventually, reconfiguration will be done on-line in real-time, but for now we are concentrating on providing reconfiguration as a quick-turn off-line operation to make the problems tractable in the near-term. For the car example, ideally reconfiguration in response to a component failure is done while driving, but in the near-term we will instead assume that the car is pulled to the side of the road before reconfiguration takes place, and then can resume operation once a new configuration is established.

3. The reconfiguration manager

In a system with an automatic reconfiguration mechanism, graceful degradation becomes fairly easy to accomplish. Whenever the failure of a component is detected, a new configuration is installed to obtain maximal functionality using remaining system resources, resulting in a system that still functions, albeit with lower overall utility. Designers using such an approach do not necessarily have to examine each combination of faults to specify designated configurations, but rather rely upon a generalized reconfiguration engine to deal with any combination of faults as it actually happens.

The RoSES reconfiguration manager is made up of the following abstract components:

- **Fault Discovery/System Model** - The reconfiguration manager can either start with a system model and then cut out pieces whenever it discovers a subsystem is faulty (a Fault Discovery mechanism) or it can build a System Model from scratch by asking each working component to describe itself. The concept is the same -- the reconfiguration manager must know what sensors and actuators are operational before it builds a configuration.
- **Configuration Generator** - A means to examine the extremely large search space and intelligently choose candidate configurations. To ensure only valid configurations are chosen, the Configuration Generator would generate candidates from a Dependency Model and filter them with a Validity Checker.
  - **Dependency Model** - Certain elements of a configuration may require or restrict other elements, either by requiring they be present, absent or placed in a particular manner. An example of the latter might occur in an automobile, where use of a particular braking algorithm would require the same algorithm be used on all other brake actuators. Such dependencies define the search space from which the configuration generator may draw candidates.
  - **Validity Checker** - Ensures only valid configurations are considered. Ensures the configuration would be schedulable, is consistent (*e.g.* consumer algorithms can properly partake of producer data), and consumes no more resources than are available.
- **Cost Model** - Allows comparison of various configurations. Cost models may be fairly complex, as they may become scenario-specific.

- **Device Customization -** An adapter loader deploys the chosen configuration throughout the system. Over the low bandwidth networks common to distributed embedded systems, real-time process migration is unlikely. Rather, the deployment will transfer small bits of state to prepositioned executables or move code while the system is off-line.

The point in time when automatic reconfiguration is executed must be carefully managed. The cost of running a reconfiguration manager to determine the appropriate configuration can be significant and the network schedule may not have slack for adapters to be loaded. Especially in the case of a tightly scheduled and resource-constrained system, there may not be enough resources (CPU or network cycles, timing slack, etc) to actually execute a reconfiguration step. Instead, we envision automatic reconfiguration employed during extreme duress or down time. In the case of a crisis, breaking schedules to run the reconfiguration manager makes sense in that the system would be completely broken and have no chance of fulfilling its mission otherwise. Running the reconfiguration step may allow the system to find a configuration where some useful work can still be accomplished with the available resources. More typically, execution will happen when the system is down for maintenance, or at a slack time in the schedule. In an elevator, for instance, a reconfiguration step may occur during the otherwise idle time when the elevator has the doors open for passenger loading. An alternative approach may employ an incremental reconfiguration manager that can, in a series of steps, make small changes in the system configuration and eventually converge on a high-quality configuration.

4. Reconfiguration as Logistical Support

Once a system has a reconfiguration mechanism, it can be exploited to provide major logistical benefits: the ability to make replacements with non-exact spares, a reduced reliance on legacy spares, and graceful upgrade capability.

Replacing defective parts with non-exact spares is of great logistical utility. If achieved, this would free maintenance personnel from the burden of carrying every conceivable spare part. For example, they might just carry more capable, generalized spares instead of cost-optimized specific repair parts. These parts may be more expensive, but minimizing trips to pick up spares would reduce labor and transportation costs, often offsetting increased component costs. In emergencies, sub-optimal repair parts might be used to perform temporary partial repairs. While the military implications for compact spares inventories and non-exact battlefield repairs are obvious, such issues are also important for any system involving mobile maintenance personnel or systems with few installed systems served per supply depot.

In addition, a major cost of supporting legacy systems is the need to provide legacy spares. In the US, a ten-year spare parts pipeline is mandated for automobiles, subjecting vehicle OEMs to interesting factory utilization challenges. Vehicle OEMs must weigh the warehousing costs of spare parts with the need to keep a factory line in operation to manufacture the parts. This mandate will be increasingly challenging as more and more automobile subsystems involve digital electronics – entire IC fabrication and packaging processes may need to be kept operational far beyond their obsolescence merely to provide spare parts designed a decade earlier.

An automatic reconfiguration mechanism may ease such logistic nightmares. Rather than replacing a part with an exact duplicate, a non-exact spare may be employed. The reconfiguration mechanism can then be used to find a different configuration that still provides for the same level (or perhaps an enhanced level) of functionality. By building updated sensors and actuators capable of several different algorithms (*i.e.* containing several different mobile object adapters), system designers will fulfill requirements for legacy spares. Such a situation is analogous to providing legacy device drivers for a computing device, and is probably no more costly.

Ultimately, it is important to gracefully reintegrate a repaired component as well as to reconfigure in the face of a component failure. As subsystems are repaired or replaced, the reconfiguration manager determines configurations that can use the added resources to restore functionality.

In addition, reconfiguration allows access to configurations beyond the original product design. If a repair is made with a replacement part having superior performance, reintegration of the repair part is not just a repair, but also a system upgrade. Beyond that, it is possible that new components (and associated abstract functionality blocks and software modules) can be added to perform field upgrades using the same approach as that employed for reintegrating repair components.

In fact, graceful degradation and upgrade via reconfiguration are simply ways of moving down or up the lattice of points in the product family architecture. When some hardware breaks or is inserted, it is as if a different model in the PFA has been realized. The reconfiguration manager is responsible for controlling the motion within the PFA lattice – by choosing which is the best collection of features to install on the available hardware.

5. Problems with Reconfiguration

Reconfiguration is not a panacea. If it were, it would already be in widespread use in almost every distributed embedded system. Some of the challenges discussed in this section are merely research challenges. Others are fundamental to the types of systems being built and will remain formidable barriers for those applications.

5.1 Debugging and Technical Support

One of the prime reasons not to make use of a reconfiguration framework is the desire of designers and developers to maintain strict control over the system, and thus simplify debugging and technical support tasks. The existence of a reconfiguration mechanism allows for a wide variety of system states, and determining proper system operation in each is impossible – and in fact it is nearly impossible to do even for a single configuration.

The debugging problem may be alleviated with adherence to a carefully controlled architecture. In the same way that the abstraction of an object-oriented system reduces overall complexity and assists with interface compatibility, reconfiguration is much easier when the adapters fit well defined and properly abstracted logical interfaces. The extent to which architecture may actively support reconfiguration is an interesting research problem being addressed by the RoSES project.

Technical support can also be a challenge. When a user reports an error or problem, knowledge of the current configuration is useful. The reconfiguration manager must scrupulously log all configuration changes and make configuration data available to the problem resolution team. This can be a problem to the extent of the frequency of configuration changes. In a system where reconfiguration is only executed during maintenance, the configuration data will be easier to maintain. If, however, reconfiguration happens often, say whenever a vehicle is started or whenever an elevator's doors are opened, then it is difficult to track exactly what the contents of the configuration were during at the time of any particular problem.

5.2 Certification Challenges

Many applications, often those in public service or extremely safety-critical, need to be approved by a certification authority. In the US, nuclear power plants must pass specification, design and implementation verification by the Nuclear Regulatory Commission. The Federal Aviation Administration certifies avionic and flight control systems, while some security systems are in the purview of the National Security Agency. A reconfiguration mechanism may increase the certification costs, as the developers now must ensure the certifiers are comfortable with the reconfiguration mechanism and the manner in which configurations are chosen and deployed.

Any gains from supporting reconfiguration would come when a later version of the product must be certified. If the regulatory agency understands reconfiguration and is comfortable with the implementation, then re-certification merely involves checking that any changed subsystems conform to the same logical interface.

In the case of safety-critical systems, the loss of system configuration control due to automatic reconfiguration may be deemed too great a risk. In such a case, designers can pursue a separation strategy whereby the safety critical functionality is partitioned away from all other features. Reconfiguration could then be enabled only for non-critical functionality. This is a common strategy, for instance, in vehicles where one network is employed for engine control, braking, *etc.* and another network is used for the power windows, door locks, and emission control.

5.3 Error Detection, Failover and Reconfiguration

Error detection is a surprisingly difficult task. Many fault-tolerant systems dodge this problem by assuming a *fail fast, fail stop* fault model, wherein the node or process is assumed to quickly shut down after a fault occurrence. For such a fault model, a simple heartbeat message, or the node's fulfillment of its portion of a network schedule signals the reconfiguration manager that all is well. Covering more complex fault models will require more enterprising error detection schemes. Robust mechanisms are needed to ensure the reconfiguration manager knows of a fault, with particular attention paid to cases where only part of a node fails. In such scenarios, it is probably less than optimal to shut down the entire node, especially if the failure only affected a portion of the sensors or actuators hosted at the node. Note that faults in the reconfiguration manager itself can be handled through standard fault tolerance means and are not of significant interest (and, even if the manager fails completely, a currently loaded configuration would still be able to operate).

Once a failure is detected, prompt configuration switches are necessary. A reconfiguration manager that performs incremental changes might be useful to accomplish on-the-fly configuration changes. Additionally, an ability to balance configuration quality *vs.* decision time seems attractive.

5.4 Multi-vendor Challenges

When a single team is responsible for developing an entire system, reconfiguration can be an elegant technology. However, much like other software, if a system is built by integrating components from multiple vendors or organizations, some special design and legal challenges emerge.

Designing for Cross-vendor reconfiguration. At its core, reconfiguration takes advantage of some extra resources to install functionality. The extra resources are provided by design or by freeing them from lower priority uses. In a multi-vendor environment, the extra resources may be taken from one vendor's unit in order to provide extra functionality to a unit from a different vendor. The first vendor may object as the cost to provide the resources makes the unit more costly compared to any non-reconfigurable competing units.

Liability. It is not at all clear how the liability for an accident or failure would be allocated in a system capable of reconfiguration. Determining the origin of the error is complex, as described in Section 6.1. In general, if a module written by vendor A were installed on vendor B's device by a reconfiguration manager provided by vendor C, a jury could easily find any of the parties liable in the case of an incident – especially in cases of miscommunication between A, B and C.

6. Related Work

Reconfiguration mechanisms are frequently introduced in the co-design field, where reconfiguration is used to change the programming of a field programmable gate array (FPGA) or other integrated circuit [6]. System-wide reconfiguration in the co-design field is typically seen as a synthesis problem, not the composition approach we take.

General organizational questions about distributed embedded systems have been examined in several different manners. Amorphous computing attempts to apply biological processes to create self-organizing and, hopefully, fault-tolerant arrangements among sensors and actuators [2]. Such approaches are in an early research stage and have yet to address or raise many real world challenges such as certification and liability.

Graceful degradation is the subject of few research papers. [4] is one which illustrates the complexities in pre-planning the configuration lattice. [7] describes an industrial project to develop a system that gracefully degrades. It also points out how difficult this can be for a centralized system – the product included a reasoning engine and detailed models, not only of the subsystems, but also a physics-based model of the environment.

The system vision of federated sensors and actuators is similar to those espoused by many middleware technologies such as Jini [3] and CORBA [5]. We expect to find such middleware to be very useful for implementing our vision of fine-grained

mobile object adapters, although both technologies are currently a bit too resource-intensive for many embedded system projects.

7. Conclusion

Graceful degradation is a very nice middle ground between the expensive fault-tolerance of modular redundancy and the low cost of non-robust systems. Unfortunately, graceful degradation is difficult to achieve in a systematic manner. The system architecture seems to be critical to achieving smooth degradation steps. If less useful functionality is bound architecturally to vital functions; by, for instance, being part of the same system modules, then it cannot be shed to free resources when a fault appears.

We think that a PFA based on reconfiguration mechanisms provides an appropriate framework in which to design and reason about a system's ability to gracefully degrade. As a means to explore our ideas, we have begun the RoSES project. RoSES will also address some interesting research, challenge that address both technical and business concerns. We expect RoSES to demonstrate a product family approach combined with a reconfiguration infrastructure that will provide the advantages of graceful degradation, graceful upgrades, and reduced logistical cost through the use of non-exact spares.

References

[1] Ben Di Vito. Formalizing Partitioning in Integrated Modular Avionics. Presentation slides at http://archive.larc.nasa.gov/shemesh/Lfm97/slides/lfm97-bldpartition-slides/P001.html, September 1997. Accessed 6 Sep 2000.

[2] H. Abelson. Amorphous Computing. *Communications of the ACM*, 43(5): 74-82, May 2000.

[3] Javasoft Corporation. Jini Technology Overview. http://www.sun.com/jini/ Accessed 6 Sep 2000.

[4] Maurice P. Herlihy. Specifying Graceful Degradation. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):93-104, January 1991.

[5] Object Management Group. Common Object Request Broker Architecture FAQ. http://www.omg.com/gettingstarted/corbafaq.html Accessed 6 Sep 2000.

[6] Yanbing Li, Tim Callahan, Ervan Darnell and Randolph Harr. Hardware Software Co-Design of Embedded Reconfigurable Architectures. In *Proceedings of the 2000 Design Automation Conference*, December 2000.

[7] Y. Shimomura, S. Tanigawa, Y. Umeda and T. Tomiyama. Development of Self-Maintenance Photocopiers.