A Graceful Degradation Framework for Distributed Embedded Systems

William Nace
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15217
wnace@cmu.edu

Philip Koopman
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15217
koopman@cs.cmu.edu

ABSTRACT

Automatic graceful degradation can be accomplished by reconfiguring the software elements of a distributed embedded system to accommodate the available hardware upon detection of a fault. The reconfiguration algorithm selects software components from a Product Family Architecture in order to maximize the functionality of the system. The mobile software components must then be allocated to the hardware so as to ensure network and processor resources are conserved. As the allocation step is NP-complete, care must be taken to ensure it is attempted only when a good chance of success exists, otherwise the rest of the algorithm merely wraps polynomial time (or worse) loop constructs around this hard core. Therefore, good heuristics are critical to success of this algorithm.

1. INTRODUCTION[†]

This paper discusses a framework or general pattern for solution of the *system-wide configuration* problem, useful for reconfiguration of a distributed embedded system in response to a system failure or upgrade. Reconfiguration has been identified as a key mechanism for an automatic graceful degradation facility in [Nace2000]. The key insight of such reconfiguration is to, in response to a hardware failure, re-synthesize the system in such a way as to maximize system functionality. Such synthesis assigns software components, called *adapters*, from an extensive library to the available hardware components. We use the term "adapter" because of the way it adapts a physical interface to a logical one. For the purposes of this paper, it is only important that the adapter be a mobile software component.

Other useful scenarios for reconfiguration include initial configuration, parts replacement with non-exact spares, and graceful upgrade.

The system-wide customization problem, therefore, is to maximize the utility of a system with pre-specified hardware by selecting and allocating software components. We propose a three step framework for solution of this problem, which is illustrated in Figure 1. Step 1 is to choose features that should be implemented in an attempt to decide how much utility should be attempted. Step 2 selects the adapters that provide data for those features. Step 3 then ensures the adapters can be allocated to hardware resources. Failure of the allocation results in a re-execution of previous steps to find different sets of adapters for allocation. The allocation step (and possibly others) is NP-complete, and thus must be attacked by finding appropriate heuristics. A critical research challenge is to ensure the other steps are not merely wrapping several huge "while loops" around it. To that end, the information returned from a failure of one step is used to guide subsequent searches of the previous step.

[†] This work supported by the US Air Force and the General Motors Satellite Research Lab at Carnegie Mellon University

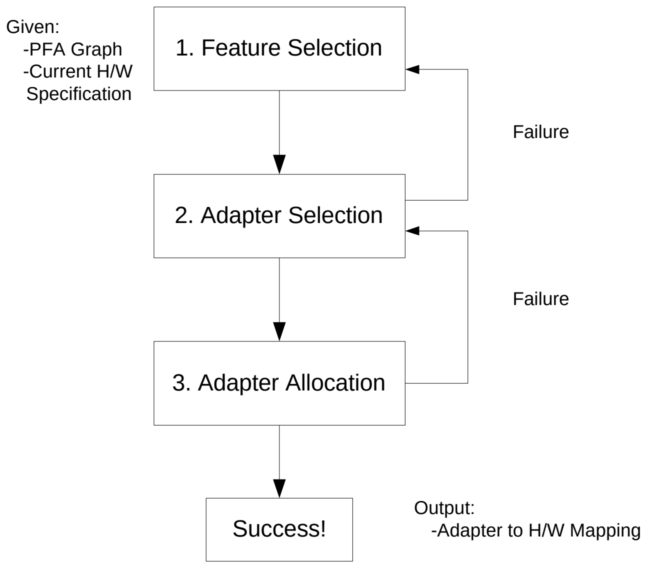

Figure 1: Algorithm Framework

This problem is similar to several other important research questions; and, while different in some respects, may find useful insight in such approaches. The hardware-software codesign field views system synthesis as a search for the minimal hardware that fulfills a fixed utility[Kalavade93]. We are attempting to find the maximum utility that can operate on fixed hardware. Reconfigurable computing uses special hardware, typically a Field Programmable Gate Array (FPGA), and dynamically modifies the operation of that hardware to increase performance. We reconfigure an entire distributed system with software components in response to fault events in an attempt to increase robustness. At some level, the system-wide customization problem resembles the agent based computing field, but is more amenable to analytic techniques. Other similarities exist with economic and operations research fields that may provide fertile grounds for applicable solution ideas.

2. MOTIVATION

Much of the reasoning behind why users and system designers would find graceful degradation at all useful has been expounded in [Nace2000]. We believe a solution to the *system-wide customization* problem to be useful as the core behind a solution to the graceful degradation challenge. In short, upon component failure a *reconfiguration manager* would be executed to customize the system for the remaining components.

Systems built with such a mechanism could then fulfill some of the very difficult reliability requirements of embedded systems. In addition, they would gain significant logistical benefits, such as freedom from legacy spares and the ability to repair systems with non-exact spares.

We believe this is feasible in the distributed embedded system domain for several reasons: distributed compute capability, smart sensors, and optimization functionality. The distributed nature of these systems implies that when a component fails, much of the rest of the system is still operational and, save for network failures, can still communicate. They can still get work done, because they are typically microcontroller-based sensors (*i.e.* smart sensors) that have their own general compute resources. As microelectromechanical system (MEMS) sensors become more prevalent, this trend will accelerate, as such sensors have huge amounts of silicon, used merely for structural reasons, upon which increased compute power can be constructed. Finally, reconfiguration is possible in the distributed embedded domain because such systems increasingly have a large portion of their functionality devoted to various optimizations. Fundamental automotive drive train control has not changed drastically, for instance. But large numbers of electronic components have been added to the vehicle for performance enhancements, environmental controls and the increased user satisfaction provided by various infotainment devices. Such optimizations can be shed in the case of failure in order to provide the computing resources necessary to fulfill the base mission of the system.

3. PRIOR WORK

A number of research efforts have studied the Adapter Allocation step as it applies to the allocation of computing tasks to parallel and multiprocessor systems, notably in [Stone77, Shen85, Bokhari81, Bokhari88, Chu87, Indurkhya86, Efe82, Houstis90]. [Woodside93], for instance, developed heuristic extensions to the MULTIFIT bin-packing algorithm to create a static task allocator for embedded systems. [Beck95] used a similar allocation step as part of the specification of a distributed system. [Prakash92] showed the usefulness of linear programming techniques to the same problem – simultaneous specification and allocation – though the application of such techniques to problems with a large number of adapters appears to be computationally challenging. [Kwok99] benchmarks and generates comparison metrics on 15 different task graph scheduling algorithms. Such algorithms are very similar to the allocation issues of interest, though the desirable metric is usually a minimal critical path schedule. We intend to exploit, to the maximum extent possible, the excellent ideas and techniques described in this research body.

Much less research has covered the other steps of this algorithm. The closest research is [Beck2000] and [Reagin99], who constructed robotic workcell applications using the customization opportunities of a component-based software architecture, with impressive results. Their optimization mechanism, unfortunately, was limited to the experience and guidance of the human design engineer. Analytical Hierarchical Process (AHP) is a decision sciences mechanism to choose functional system alternatives, as illustrated in [Braglia99]. AHP relies upon an extensive number of pair wise comparisons, structured hierarchically, to make a single decision. As such, it is similar in goals to the Step 1 algorithm, but would require designer input to guide each of these multitude of pair wise comparisons, and is therefore difficult to automate. For this reason, we shall develop our own mechanisms for Steps 1 and 2, which are described below.

4. SOLUTION FORMULATION

The well known data flow graph (DFG) can be used to specify the various configuration alternatives of the system. Each vertex in such a graph represents a source (sensor), sink (actuator) or transformation (adapter) of data. Edges represent the flow of communication through the system. Both vertexes and edges can be labeled by the system engineer to specify resource requirements (CPU cycles, RAM, I/O channels, network bandwidth, etc). Figure 2 shows a simple DFG for sensor fusion, low-pass filtering and Fast Fourier Transform for a hypothetical (and tiny) system.

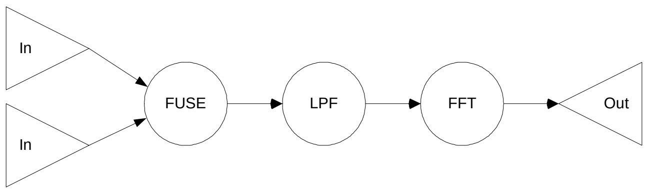

Figure 2: An Example DFG

This common representation of embedded systems must be augmented to represent the multitude of system alternatives available in the entire product family. We presume the existence of a fine-grained Product Family Architecture (PFA) which provides a structured view of all possible configurations of the system [Jiao2000]. Such PFAs are common in large distributed embedded systems.

The PFA insight allows us to construct an augmented DFG known as a PFA graph. The PFA graph is a supergraph of each configuration's DFG. In order to merge the individual DFGs, choice elements are placed between adapters to signify that any of the input adapters may send that type of data to any or all of the output adapters. Such choice elements deliberately follow the semantics of a message type on a broadcast bus, such as a Control Area Network, as is typically found in distributed embedded systems. The use of a choice element (the Part Number message) is illustrated in Figure 3 for a trivial process control system. Note that in the absence of one of the sensors (barcode reader or machine vision sub-system), the other sensor can provide the part number to the rest of the system.

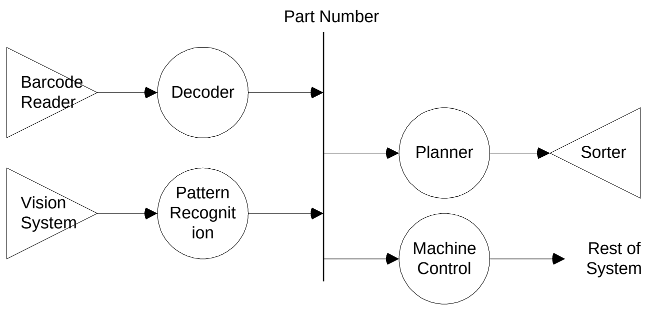

Figure 3: A Choice Element in an Augmented DFG

This augmented DFG is an expressive, uncomplicated mechanism to specify the configuration options for a system. It is sufficiently useful and yet computationally manageable enough for this research.

An important insight comes from looking at the PFA graph as a supergraph of all possible well-formed system configurations (where *well-formed* indicates the subset of all system configurations wherein data flows properly from sensor to actuators without lack of adapters nor with extraneous adapters). One important part of the solution, then, is to decide upon the sub-graph that should be placed in the system. It then becomes easy to see that pruning and combinatorial algorithms will be useful techniques to solve this problem.

5. SOLUTION ARCHITECTURE

The problem is illustrated, at a fairly high level, in Figure 1. Inputs to the problem are the PFA graph (recall, this is the augmented DFG) which specifies all the system alternatives. A description of the available hardware is also necessary. The goal is to generate a valid configuration of adapters to the processing elements (PE) and message traffic to network elements (NE).

Three major searches make up the reconfiguration algorithm: The first is to select a set of *features*, which are defined and described in the next section. In the next step, a set of adapters must be chosen that implement those features in a well-formed configuration. Finally the adapters must be allocated to the hardware and constraints checked to produce a valid configuration.

The last step, adapter allocation, has been researched in somewhat different circumstances [Beck95, Woodside93, Efe82]. Allocation is an NP-complete problem, easily mapped to the well-known bin packing problems. The real challenge of solving the system customization problem is to find creative means to keep the first two steps from merely being huge loop constructs around the third, NP-complete step.

5.1 Step 1: Feature Selection

Features are simply a means for the designer to express desires about the functionality of the overall system. They are used to guide the overall optimization of the algorithm, as a maximal set of features is chosen for the final system configuration. It is possible to express the desirability of features in many ways. As mentioned above, the AHP process (and to a certain extent the Quality Function Deployment process which superceded it) are in some ways a language for expressing customer and designer goals for a system. Both are rather heavyweight and not easily automated, however, so would make for an awkward feature model for this research.

Much of the feature selection sub-algorithm will depend upon the feature representation model. A general feature model that scales well is not the point of this research; instead, a class-based feature model that is sufficiently expressive to cover most complex distributed embedded system will be used. In the class-based feature model, some interior vertexes (adapters) in the PFA graph are labeled as *features* by the system designers. Features are also given a utility value – possibly infinite for critical functions. It is this utility that will be optimized by the algorithm. Groups of features with similar characteristics present alternatives for different system configurations. Such a group is termed a *class* (not to be confused with an object oriented "class" that an implementation might have). All features with the same class represent redundant adapters. Only one (or perhaps a pre-specified N) of a particular class is useful on a system. The overall utility of a configuration is the sum of the utility of all the features of a configuration, where only the largest utility of a particular class would count. Features can be zero sized (in terms of resources required), if a designer wanted to insert a vertex to show the desirability of obtaining data from a particular source, for instance. In addition, classes (not features) can be labeled as critical, thus imposing a constraint whereby any valid configuration must include one of the features from the critical class.

This feature model is expressive enough to allow representation of a wide variety of systems. There are, however, some useful systems that cannot be expressed and others where the expression is possible, but clumsy. For instance, suppose a particular adapter is a “superset” adapter that actually does two things. Should it have two classes? What if it could do X or Y, not both simultaneously (i.e. it handles mode changes well)? For systems where such issues are important, the system engineer should be able to replace our feature model with minimal overall effect on the workings of the algorithm. However, we find our model general enough for the large majority of embedded systems.

The feature selection algorithm attempts to optimize the overall system utility by choosing appropriate features from the different classes. An obvious first start would be to choose the largest utility features from each class, which results in the upper bound on overall system utility. Further increments will be based upon combinatoric algorithms. The expected number of features of a system is small enough, especially when compared to the number of vertices in the PFA graph, that a sorted class combination algorithm may work well. Further refinements of the algorithm will take into account feedback from failed trials to avoid the nasty looping problem discussed in the intro and section 5.4. The collection of features chosen for implementation is the *feature set*, which is passed to the second step of the algorithm.

5.2 Step 2: Adapter Selection

The selection of adapters to implement those features chosen in step 1 is the difficult heart of the entire problem. Lots of graph manipulation will be necessary to come up with the different sets of adapters for allocation.

First the PFA graph should be pruned of all vertices that don’t contribute to calculating the feature set. The graph will be traversed backwards from each feature, carefully marking vertices in a cycle-aware manner. A depth-first algorithm will be employed because such markings may need to be undone – either because the path ends in a sensor that is not operational, or the path would include a feature which is not an element of the feature set. A similar forward traversal will also be necessary, marking all the vertices from features to actuators. Most features will probably be quite close, if not incident to actuators, so this forward traversal should be quite easy. In the case where a feature actually lies within a cycle, the backward traversal will stop after a single trip around the cycle, as it comes upon the previously marked feature. Note that this pruning will include all paths from operational sensor to operational actuator that pass through features included in the feature set *and no others.* In no case will additional features be included, as that violates the separation of responsibilities of the different steps. Perhaps feedback to the Feature Selection phase might be useful however, pointing out that the two features share a dependency.

The pruned PFA graph still has plenty of redundancy, representing the subset of configuration space that implements the feature set. The redundancy exists in the variety of sensor to actuator paths that flow through features in the feature set. The reconfiguration manager must apply an orderly process to explore this space and choose candidate *adapter sets*, a collection of adapters that fulfill a single configuration. To perform the orderly search, an *adapter alternate graph* will be constructed by collapsing the pruned PFA graph. The vertices in this new graph represent choice points and correspond to the data elements in the PFA graph. An edge is included in the graph for each path between the data elements, regardless of the number of adapters it originally passed through. Selection of a candidate adapter set is then merely a search for paths through the adapter alternate graph that pass through the features sets. Hopefully the state of the search will be easier to manage with this compressed graph.

The resulting adapter set will be passed to the adapter allocation step to see if it can be fit into the hardware resources available. This adapter set represents a single DFG (no choice elements, so no flow alternates) which can then be tested for allocation on the available hardware in the third step.

5.3 Step 3: Adapter Allocation

The purpose of allocation is to determine if a configuration is allocatable and find the specific mapping of adapters to hardware and messages to networks. The allocation problem is not uncommon, and has been studied in codesign and parallel processing contexts. Most such research thrusts approach the problem as a bin-packing problem. Bin packing is NP-complete, but decent heuristic methods exist, based on non-guided search and list processing.

A thorough examination of the allocation problem in a very similar context [Beck95] used list-processing techniques to allocate tasks and specify processing element (PE) sizes. The main decisions associated with an efficient list-processing heuristic are the manner in which item size is determined and the way the bin (PEs in this case) is chosen for packing each item. The experiments in [Beck95] showed that sizing tasks is best done based on a equally weighted calculation of resource (PE cycles, RAM, I/O channels, etc) and network bandwidth usage. The determination of PE target at each allocation step should be based on more global knowledge – such as the nearness of neighboring adapters in the DFG. By making allocation decisions in such a way that the adapter is allocated to the processing element that would minimize the bandwidth requirement (i.e. pack adapters that talk to each other on the same processing element so they do not need to use the network), overall packing quality is improved.

5.4 Feedback From Failure

If not handled carefully, Steps 1 and 2 merely wrap polynomial time (or worse) “while loops” around the NP-complete problem (approximated using the heuristic means described above) of Step 3. The intriguing part of this research attempts to find a creative manner to utilize feedback from the failure of one iteration to guide search choices. When adapter allocation fails, the amount and type of resources lacking will be returned to the adapter selection phase. The subsequent adapter selection search will be very different for a huge failure versus a small failure. In fact, for small failures in adapter resources (versus network resources), a targeted search for alternate adapters might be fruitful. Network resource failures are harder to compensate for, as any alternate adapter choice involves at least two different edge replacements in the PFA graph.

Similarly, failures at the adapter selection step could provide information about the size of the closest match and the breadth of adapter sets that fulfilled each of the features in the feature set. This information would then assist the feature selection to determine which feature classes should be targeted for alternates. Gains from this second level are a bit less intuitive, but probably would pay bigger dividends, as they have the potential to make bigger search moves within the configuration space.

Selection and use of feedback information is still very preliminary. Our implementation has not yet progressed to the point where such decisions are necessary, though our framework has anticipated the communication and structuring of failure information from step to step.

6. CONCLUSIONS AND FUTURE WORK

This paper described *system-wide customization*, an interesting problem in the distributed embedded research field. This problem requires the functional optimization of a distributed system consisting of fixed hardware resources. Applications of the problem include automating graceful degradation, initial factory customization, graceful upgrade and parts replacement with non-exact spares.

In order to solve the system-wide customization problem, we have devised a solution framework consisting of three steps with feedback from a step's failure guiding further iterations of previous steps. The three steps included a search for the maximum utility feature set, a flow graph directed search for adapters fulfilling the feature set, and an allocation test to map each adapter to hardware resources.

Several possibilities exist for useful extensions of this work. Iterative and failover friendly reconfiguration managers would be very useful. Additional constraint checking should also be integrated with the algorithm.

An iterative reconfiguration manager would be useful to try to move toward a dynamic reconfiguration on embedded systems. Basically, the reconfiguration manager makes small moves in a planned manner so that the functionality of the total system increases (perhaps not monotonically), but the moves are small enough that they can possibly happen while the system is operational. This is actually one step on a possible spectrum of algorithmic tuning choices. Having an algorithm that can be set to quickly generate valid configurations that only consist of the critical features would be quite useful in an emergency situation. On the other end of the spectrum, design tools could utilize a search that generates the highest quality solutions, with little regard for execution time.

A failover friendly reconfiguration manager would keep in mind, as it determines subgraphs and allocations, that it would be nice to have redundant adapters for critical or the most desirable features. So it would include duplication of adapters where possible in the configuration chosen.

Alternate algorithm construction is also possible as an interesting comparison. The reconfiguration manager algorithm sketched out above is basically a depth first search through feature sets, adapter sets and adapter allocation. Perhaps a broader search would better cover the configuration space. Another intriguing approach would be to guide constructive solutions – basically starting out with the smallest configuration that is almost certain to fit (and, which could be mostly specified *a priori* to the algorithm) and then making small changes to attempt to build up, or construct, a more feature-rich solution.

Finally, a constraint checking phase should ensure output configurations fulfill other requirements, such as timing and schedules. It would be nice to be able to incorporate such knowledge in the decisions made in phase 1, 2 and 3 rather than waiting for the algorithm to complete its work before checking. How to integrate such constraints may be application dependent. At any rate, it is yet unclear how to fit constraint checking anywhere other than as a post process check.

Overall, the algorithm framework described is an exciting and rich exploration of the *system-wide configuration* problem. Solutions to this problem are critical to several useful lines of research – chiefly automatic graceful degradation. The industry will be constructing exceedingly complex distributed embedded systems – by including such graceful degradation mechanisms, they will hopefully be reliable, trustworthy systems.

7. REFERENCES

[Beck95] J. Beck, "Automated Processor Specification and Task Allocation Methods for Embedded Multicomputer Systems," Ph.D. Thesis, Carnegie Mellon University. April 1995.

[Beck2000] J. Beck, *et. al.*, "Applying a Component-Based Software Architecture to Robotic Workcell Applications," *IEEE Transactions on Robotics and Automation*, 16(3):207-217, June 2000.

[Bokhari81] S. Bokhari, "A Shortest Tree Algorithm for Optimal Assignments Across Space and Time in a Distributed Processor System," *IEEE Transactions on Software Engineering*, SE-7(6): 583-9, Nov 1981.

[Bokhari88] S. Bokhari, "Partitioning Problems in Parallel Pipelined and Distributed Computing," *IEEE Transactions on Computing*, 37(1): 48-57, Jan 1988.

[Braglia99] M. Braglia and A. Petroni, "A Management-Support Technique for the Selection of Rapid Prototyping Technologies," *Journal of Industrial Technology*, 15(4): 2-6, 1999.

[Chu87] W. Chu and L. Lan, "Task Allocation and Precedence Relations for Distributed Real-Time Systems," *IEEE Transactions on Computers*, C-36(6): 667-679, Jun 1987.

[Efe82] K. Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *Computer*, 15(6): 50-6, June 1982.

[Houstis90] C. Houstis, "Module Allocation of Real-Time Applications to Distributed Systems," *IEEE Transactions on Software Engineering*, 16(7): 699-709, Jul 1990.

[Indurkhya86] B. Indurkhya, H. Stone and L. Xi-Cheng, "Optimal Partitioning of Randomly Generated Distributed Programs," *IEEE Transactions on Software Engineering*, SE-12(3): 483-495, Mar 1986.

[Kalavade93] A. Kalavade and E. Lee, "A Hardware-Software Codesign Methodology for DSP Applications," IEEE Design and Test of Computers, (10)3:16-28, September 1993.

[Jiao2000] J. Jiao and M. Tseng, "Fundamentals of Product Family Architecture," *Integrated Manufacturing Systems*, 11(7): 469-483, 2000.

[Kwok99] Y. Kwok and I. Ahmad, "Benchmarking and Comparison of the Task Graph Scheduling Algorithms," *Journal of Parallel and Distributed Computing*, 59(3):381-422, December 1999.

[Nace2000] W. Nace and P. Koopman, "A Product Family Approach to Graceful Degradation," *Architecture and Design of Distributed Embedded Systems* International Workshop on Distributed and Parallel Systems (DIPES 2000), October 2000.

[Prakash92] S. Prakash and A. Parker, "SOS: Synthesis of Application-Specific Heterogeneous Multiprocessor Systems," *Journal of Parallel and Distributed Computing*, 16(4): 338-51, Dec 1992.

[Reagin99] J. M. Reagin, *et. al.*, "A Component-Based Software Architecture for Robotic Workcell Applications," *IEEE Transactions on Electronics Packaging Manufacturing*, 22(1): 85-94, Jan 1999.

[Shen85] C. Shen and W. Tsai, "A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Mini-max Criterion," *IEEE Transactions on Computers*, C34(3): 197-203, Mar 1985.

[Stone77] H. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Transactions on Software Engineering*, SE-3(1): 85-93, Jan 1977.

[Woodside93] C.M. Woodside and G. G. Monforton, "Fast Allocation of Processes in Distributed and Parallel Systems," *IEEE Transactions on Parallel and Distributed Systems*, 4(2): 164-74, 1993.