CARNEGIE MELLON UNIVERSITY

Automatic Graceful Degradation for Distributed Embedded Systems

A DISSERTATION SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree of

DOCTOR OF PHILOSOPHY

in

ELECTRICAL AND COMPUTER ENGINEERING

by

William Nace

Pittsburgh, Pennsylvania

May 2002

Version 3.14159

To Masayo, my source of strength and fount of faith.

Who believed in me when I couldn't.

TPILB

Abstract

It has long been a tenet of the fault-tolerance research community that fault–tolerance is possible by exploiting resource redundancy in order to achieve the mission result. Hardware designers, for instance, rely upon various forms of redundant hardware to ensure the availability of at least one system that can produce correct responses. Strangely, one source of redundancy has not been comprehensively examined. Such redundancy is found in the optimizations used by system designers to enhance the core system mission. It is possible to design reliable systems that, upon detecting a fault, shed such optimizations. Most users are willing to forgo an optimization (such as a few percentage points of fuel efficiency) rather than face full system failure. Such a gentle system response to faults is the hallmark of a gracefully degrading system. Fault-tolerance based upon graceful degradation provides additional techniques to build dependable systems without the heavy costs of hardware replication.

This work identifies a heretofore-unknown problem in the system synthesis research space and explores solution methods. The problem, called *system-wide customization* , is to maximize the utility of a system with pre-

specified hardware by selecting and allocating software components from an extensive, flexible library.

The importance and applicability of system-wide customization is apparent when focusing on the distributed embedded system domain. Within this domain, fixed hardware resources — distributed microcontrollers and the networks connecting them — limit the software components that can be executed. Each choice of possible hardware components can be viewed as a single vertex of a dense lattice that represents a fine-grained product family architecture (PFA). For each combination of hardware components, there are many different software configurations available. Using the PFA lattice concept, the system-wide customization problem may be expressed as the process of choosing the software configuration for a particular vertex (*e.g.,* the one representing available hardware) that maximizes the utility of the system.

Acknowledgements

Thanks and praise to God, who created such a wonderful and diverse world. I'm always amazed that the universe is so deep and so multi-faceted that I was able to spend three years digging into such a small corner of it, and yet have only scratched the surface. All praise to Him, the perfect engineer.

Many thanks to Masayo Kohama. Your love and support helped intensify the joy and diminish the pain of being a graduate student. Thanks for holding my hand as we walk through life together.

Thanks also to Akiko Elaine Nace. You make me laugh and brighten up any day. Here is your copy of "my book."

Many people assisted me with this research. Primary among them is Phil Koopman, my advisor, who taught me that if an idea is obvious only in hindsight, it's probably a cool research result. Thanks also to my other committee members: Col. Paris Neal, James Beck and David O'Hallaron. To my RoSES teammates (dare I say *Phil-istines*): Charles Shelton, Meredith Beveridge, Beth Latronico, Ying Shi, and Yang Wang. Terri Lacombe, whose art career will now be kicked into high gear. And all the wonderful denizens of D-Level who made it such an enjoyable place to inhabit. Also,

many thanks to Mrs Fletcher, who sparked my love of learning into a roaring fire — or at least gave me enough fuel to make it blaze! And to then Captain Joe Morgan, who opened the doors of engineering wonder for me. Your willingness to offer alternative learning experiences gave me rein to grow and develop far beyond the borders of the curriculum. I hope to be able to pay you forward with my own students.

Special thanks to John and Kobey DeVale, who paved the way, provided insightful intelligence, and are all-around excellent folk. To Lynn and Elaine, who motivated me for the final push. To Lou, Jim, Dave, Mat and all the others who kept my computers operational. And finally to all of my Pittsburgh friends, whose thoughts and prayers came in extremely handy.

I am also quite grateful for the support of the US Air Force, General Motors Satellite Research Lab at Carnegie Mellon University and Bosch Electronics.

Non novit virtus calamitati cedere[1]

Quidquid latine dictum sit, altum viditur[2]

[1]Courage knows not how to yield to disaster
[2]Whatever is said in Latin sounds profound

Contents

Abstract **v**

Acknowledgements **vii**

1 Introduction **1**

1.1 Problem Statement 3

1.2 Summary of Related Work 8

1.3 Thesis Outline 10

1.4 Research Contributions 12

2 Background **15**

2.1 A Graceful Degradation Mechanism 17

2.2 The Customization Manager 19

2.3 Customization as Logistical Support 22

2.4 Problems with Customization 24

2.4.1 Loss of Design Determinism 25

2.4.2 Development for System-wide Customization 26

2.4.3 Debugging and Technical Support 27

2.4.4 Certification Challenges 28
2.4.5 Multi-vendor Challenges 29
2.5 Conclusion 30

3 Related Work **33**
3.1 Graceful Degradation 33
3.2 Reconfiguration and Customization 36
3.3 Hardware–software Codesign 38
3.4 Task Allocation 38

4 System Model **41**
4.1 Overview 42
4.1.1 Embedded Networks 42
4.1.2 Smart Sensors and Actuators 44
4.2 Representation 47
4.2.1 Processing Elements 48
4.2.2 Network 48
4.2.3 Configuration 49
4.3 Fault Model 51
4.4 Summary 52

5 Problem Definition **53**
5.1 Guiding Scenario 54
5.1.1 Operational Scenario 54
5.1.2 An Example Subsystem 56
5.2 Problem Models 57

5.2.1 The Lattice 57
5.2.2 Mapping Utility, Hardware and Software 60
5.3 Specifying PFAs 63
5.3.1 Merging DFGs to Form a PFA Graph 64
5.3.2 Features 69
5.4 Summary 74

6 Algorithmic Framework 75
6.1 Phase 1: Feature Selection 77
6.1.1 Heuristic Evaluation for Feature Selection 84
6.2 Phase 2: Adapter Selection 89
6.2.1 Heuristic Development for Adapter Selection 92
6.3 Phase 3: Adapter Allocation 98
6.4 Summary 99

7 A New Adapter Allocation Algorithm 101
7.1 Introduction 101
7.2 System Model 103
7.3 The TRANS_FIRST Algorithm 105
7.4 Policy Choices 106
7.4.1 Choosing an Adapter for Allocation 106
7.4.2 Choosing a PE 108
7.4.3 Adapter at a Time Allocation 108
7.4.4 PE Fill Level 109
7.4.5 Allocation of Remaining Adapters 109
7.5 Results 110

7.5.1 Test Set 110
7.5.2 Experimental Method 111
7.5.3 Finding the Right Policy Choices 112
7.5.4 Comparison to Base Algorithm 115
7.6 Conclusions 116

8 Proof of Concept **117**
8.1 The Distributed Elevator Control System 117
8.2 Building a PFA graph 119
8.2.1 Connectivity 119
8.2.2 Bandwidth Requirements 119
8.2.3 Adapter Size 121
8.2.4 Missing Modules 127
8.2.5 Replication 128
8.2.6 Splitting Monolithic Controllers 129
8.2.7 Team1 132
8.2.8 Completing the PFA graph 132
8.3 Algorithmic Changes 133
8.4 PFA Limitations 135
8.5 Graceful Degradation in Action 140
8.6 A Few Notes About Performance 143
8.7 Conclusion 145

9 Conclusions **147**
9.1 Summary 148
9.2 Retrospective 150

9.2.1 The PFA Model 151
9.2.2 Feature and Utility Models 152
9.2.3 Allocation Algorithms 154
9.3 Contributions 156
9.4 Future Work 157
9.4.1 System Model 157
9.4.2 Problem Definition 158
9.4.3 Algorithms 159
9.4.4 Customization Manager Styles 160

A Elevator System Tables **163**

B Navigation System Description **177**

B.1 XML Description 177

C ASCII Chart **191**

List of Tables

6.1 Allocation results for some feature combinations 86

6.2 Algorithm evaluation results 88

6.3 Summary of phase 2 heuristics 94

6.4 Number of iterations required for different heuristics 96

7.1 4 axes of policy choices 107

7.2 Allocation graph characteristics 110

7.3 Results of each policy 113

7.4 Comparison to base algorithm 116

8.1 Elevator nomenclature 120

8.2 Elevator messages, meaning and payload sizes 122

8.3 TeamA's modified controllers 131

8.4 Fault injection results 141

8.5 Performance of each phase 144

8.6 Execution time of phase 2 for various sized elevators 144

A.1 Team A — Original code measurements 163

A.2 Team A — Missing modules measurements 167
A.3 Team A — Modified modules measurements 168
A.4 Team 1 — Original system measurements 170
A.5 Team 1 — Modified adapter measurements 174

List of Figures

1.1 Solution overview 7

4.1 The distributed embedded system control loop 42
4.2 Anatomy of a PE 45
4.3 A simple data flow graph 50

5.1 Part of an example lattice 58
5.2 A portion of the navigation PFA lattice 60
5.3 The MUSH model 61
5.4 Various configuration spaces of interest 64
5.5 Simple example of data elements 67
5.6 Navigation system PFA graph 68
5.7 A subset of the navigation PFA graph 73

6.1 The reconfiguration algorithm 76
6.2 The number of configurations for each feature set 80
6.3 Navigation system features organized by class 82

7.1 A sample allocation graph 103

8.1 An example of path overlap 134

8.2 A (difficult) solution to the combination sensor problem . . . 137

List of Abbreviations

API Application Programming Interface

ASIC Application Specific Integrated Circuit

CAN Control Area Network

CD-ROM CD-ROM

CMU Central Michigan University

CPU Central Processing Unit

DES Distributed Embedded System

DFG Data Flow Graph

FAA Federal Aviation Administration

FPGA Field Programmable Gate Array

JVM Java Virtual Machine

MEMS Micro-ElectroMechanical System

MOA Mobile Object Adapter

NSA National Security Agency

OEM Original Equipment Manufacturer

PFA Product Family Architecture

ppm parts per million

PWM Pulse-width modulation

RAM Random Access Memory

RoSES Robust Self-configuring Embedded Systems

TPILB This Page Intentionally Left Blank

TTP Time Triggered Protocol

TLA Three Letter Acronym

USA United States of America

US United States

XML eXtensible Markup Language

Chapter 1

Introduction

Designers of distributed embedded systems have, until now, had few technologies available for reaching high dependability system goals during system execution. The only widely used hardware techniques are based on redundant hardware – triply (or more) redundant modules or a near cousin such as hot spares. Software dependability technology is limited to a similar n-version programming technique or to claims about improvements in software process. Other methods exist for pre-deployment hardening of the software and hardware in order to discover specification and implementation problems, but are of no help when the system is faced with hardware failures at run time. Unfortunately, both hardware and software redundancy techniques are quite expensive, as they require extra hardware or the production of multiple versions of the software. Furthermore, the ability of n-version programming to reach dependability goals has not been conclusively demonstrated.

Two major trends collide and exacerbate the lack of technological tools:

increased cost sensitivity and higher dependability requirements. System cost is a driving factor in the production of most complex, embedded systems, especially those – such as automobiles – that are to be fielded in large quantities. They simply cannot be economically constructed with triplex computing hardware. Simultaneously, distributed embedded systems are being used in more facets of the human lifestyle, and in ways that demand reliable, dependable systems. As the transportation, industrial control, home security and office automation systems that surround us become filled with computer systems to handle their increasingly complex control, they must not founder when faced with exceptional conditions or hardware failures. Otherwise, such failures escalate to mission and safety critical status.

Graceful degradation has the potential to alleviate this nettlesome problem. Operational systems can have more than two states, more than just "working" and "not working." Rather, when faced with a failure, a gracefully degrading system may shed some functionality in order to ensure the remaining functionality operates properly. If done properly, a graceful degradation approach might extend mission operation until repairs can be made.

A gracefully degrading system does not inherently demand more hardware than a non-gracefully degrading one. It might, however, require significantly more development effort. There is a surprising lack of research to guide system designers with construction techniques that would minimize the development costs. The naive design method is to examine each combination of failed components and determine what the appropriate system response would be. In general, such a naive method will examine and design the system for each of the 2^n configurations of an n component system. Since

such combinatorial explosion is obviously untenable for realistic distributed systems with large n, techniques must be devised to either automatically collapse the number of configurations to a set which can be handled by human designers or, alternately, to find algorithms to automatically use available software components to maximize functionality of whatever hardware is operational. This dissertation examines the latter approach.

1.1 Problem Statement

System-wide customization is the process of maximizing the utility of a system with pre-specified hardware by selecting and allocating software components. This dissertation examines system-wide customization (or customization for short) as a mechanism that may be employed to achieve graceful degradation. A system's fixed hardware resources — distributed microcontrollers and the networks connecting them — limit the software components that can be employed. Each choice of possible hardware components can be viewed as a single vertex of a dense lattice. For each combination of hardware components, there are many different software configurations available. The system-wide customization problem may be expressed as the process of choosing the software configuration for a particular vertex (*i.e.,* the one representing available hardware) that maximizes the utility of the system. As a system undergoes successive failures, the system's hardware state changes and can be represented by a series of neighboring lattice vertices. If the system is to degrade in a graceful manner, it must deliver a great deal of its potential functionality at each stage. After a particular failure, the hard-

ware is fixed — a repair action must be performed to change hardware. By customizing after each failure, the degradation trajectory is such that the system's utility is maximized at each stage — and thus the degradation is as graceful as possible, given the particular hardware failure sequence.

We believe system-wide customization is a general enough mechanism to be useful in quite a few different types of computing systems. However, we set out to investigate system-wide customization in the context of distributed embedded systems as a way to constrain the problem to a solvable size. Distributed embedded systems have three characteristics that make them especially good candidates for customization: distribution, general compute ability, and optimization requirements. A distributed system is able to survive many types of failures with computing and network resources intact, as opposed to a centralized system where a single failure may crash the sole CPU. General compute ability is becoming more common on distributed embedded systems, as "smart" sensors (and actuators) use microcontrollers to interface the sensor to the rest of the system. Code in the microcontroller is responsible for control of the sensor/actuator and for converting the raw sensor data to/from values usable by other components. But, the microcontroller can execute general purpose software components; it might be called upon to do so if a higher utility component is forced to be rehosted due to a hardware failure. The third characteristic is the presence of many optimization requirements, or non-mission critical functions. Marketing pressures force designers to add features to systems that do not result in changes to core functionality; they merely add system optimizations. For example, an automobile has a large portion of its electronic systems devoted

to passenger entertainment, fuel economy, emission management, and advanced stability and control algorithms. While all such functions are useful, they do not materially change the basic mission of the automobile: to convey passengers along a roadway in a safe manner. All computing resources originally intended for the non-mission algorithms can be used, in case of an emergency, to rehost the critical control algorithms of the automobile.

This dissertation documents our explorations, which formed, in essence, a voyage of discovery leading to our goal of system-wide customization. We do not claim our path is the only one, nor that it is the best, merely that it arrives at the destination. The results of our exploration fall into three areas, which together form a framework wherein the customization problem can be solved. The framework involves three algorithms working together, each algorithm responsible for one of the three aspects of system-wide customization: the feature model, the software repository and the allocation to hardware.

The *Feature Selection* algorithm operates on the feature model, and is responsible for choosing system *features* to implement. A feature is a mechanism to accomplish a system *function*, or desirable system behavior. Functions, such as automotive braking, can be accomplished in a multitude of ways (*e.g.,* standard or anti-lock braking). From this view, system design (or architecting) is the process of choosing features to implement for each function. In system-wide customization, the feature selection algorithm is responsible for optimizing which features get picked for which functions. The feature model we use is based on the notion of functions and features, where features are cleanly separated among functions. Such a feature model is not

comprehensive enough to cover the world of complex systems — in some systems the features interact in ways that cannot be cleanly divided into functions. The feature model is, however, sufficiently powerful for framing a general approach to graceful degradation and refining our understanding of how system-wide customization might fulfill the requirements of gracefully degrading distributed embedded systems.

The *Adapter Selection* algorithm chooses software components (called *adapters*, for reasons described in Section 4.1.2) to fulfill the requirements of the features chosen by the feature selection algorithm. It chooses from a library of available components (an *adapter repository*) in such a manner that dependent components are also included. In many cases, a large and flexible library of components will not be available as such. However, we make use of a library that often will be available — software components from related product models. Other models in the product family likely have software that can be composed to create various different product instances. Our work models the entire product family as a *Product Family Architecture graph* (or PFA graph), the supergraph of the data flow graph of all product instances. Again, we feel such a model of the available components and their dependencies is quite useful for many distributed embedded systems, but is not universal. The PFA graph assumes, for instance, a system architecture amenable to representation as synchronous data flow. A transactional system, for instance, might require different description, and thus different algorithms. For the embedded domains we considered, the PFA representation is quite useful and quite realistic.

The final portion of the framework is the *Adapter Allocation* algorithm,

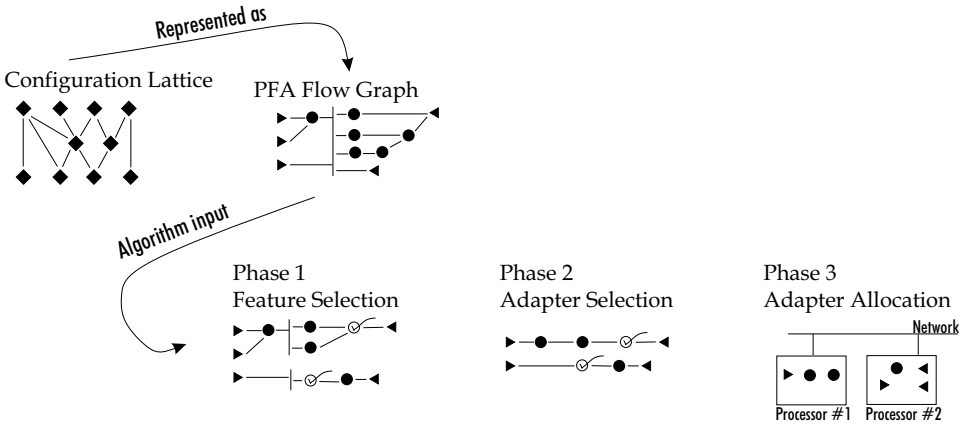

Figure 1.1: Solution overview

which determines a feasible placement of software components on the microcontrollers of the system. Software components are placed such that their computational requirements do not exceed the computational resources of the microcontrollers, and the communication between components does not require more bandwidth than the network provides. This allocation problem is well studied, with heuristic algorithms based on bin packing, integer programming, graph theoretic means, guided searches and vertex clustering. We chose to implement a bin packing solution. The reason we did so is fairly arbitrary, but we needed to ensure the placement of sensors and actuators within the solution was properly represented — a constraint uncommon to allocation algorithms. Other allocation algorithms generally assume homogeneous processing elements. Modification of the bin packing algorithms appeared simpler than the other types of algorithms. In the process of altering the algorithm, we gained additional insight that allowed for a different style of bin packing and resulted in an improved algorithm with a significant

speedup.

Figure 1.1 illustrates the employment of the customization framework. A configuration lattice with all of the abundant possibilities of the product family architecture can be represented as a PFA graph. Phase 1, the feature selection algorithm, picks features based upon the system's feature model. Our feature model uses portions of the PFA graph as a bookkeeping method to represent particular features (much more detail is presented in Chapter 6.1). Phase 1's output is a set of features and serves as the input to Phase 2. Phase 2 is the adapter selection algorithm, which employs the PFA graph to determine dependencies among software components. Phase 2's output is an subset of the PFA graph, a set of software components that fulfills the features selected in Phase 1. Phase 3, the adapter allocation algorithm, takes the set of components and determines how they fit on the available hardware.

The framework is iterative. A failure of a particular phase will result in a return to a previous phase for an alternative. Phase 3 is NP-complete, so Phases 1 and 2 should be as efficient as possible in order to direct the search to rapid solution. Otherwise, Phases 1 and 2 merely become large loop constructs — which results in repeated executions of the NP-complete 3rd phase and the attendent poor performance of the framework.

1.2 Summary of Related Work

Very little research has been done on graceful degradation in computer systems, none of it aimed at distributed embedded system. Specification of

graceful degradation has been examined in [Herlihy91] and to a lesser degree in [Weber89]. The former is the source of the lattice models discussed in Chapter 5.2.1. Graceful degradation of real-time scheduling was examined in [Ramanathan97], where degradation referred to the failure of some tasks to meet deadlines. Customizing a system in order to change the functionality, as opposed to performance, is a novel idea.

The customization aspects of this approach are similar in some respects to several research areas, primarily hardware-software codesign and reconfigurable computing. The codesign field usually views system synthesis as a search for the minimal hardware that fulfills a fixed utility. In contrast, this research attempts 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. System-wide customization operates on an entire distributed system, changing and re-allocating software components in response to failure events in an attempt to provide robustness.

On the other end of the spectrum, task allocation is a fertile research area, well plowed ever since [Stone77] first examined data flow graphs for scheduling dual processor systems. Further research includes [Stone78, Altenbernd96, Altenbernd98, Benveniste91, Blickle98, Bokhari81, Bokhari88, Efe82, Woodside93]. By examining the constraints unique to distributed embedded system, we developed a new twist on task allocation, which is thoroughly examined in Chapter 7. [Kwok99] compared the multitude of task allocation algorithms, in the process of which a standardized

data set was generated. An alternate data set is proposed in [Tobita00].

1.3 Thesis Outline

The organization of the rest of the thesis is described below:

Background

Chapter 2 is a detailed examination of some of the motivation for this research.

Related Work

Chapter 3 discusses work related to this thesis, primarily from the areas of hardware - software codesign, CAD algorithms, task allocation.

System Model

A computationally tractible model of the system is developed in Chapter 4. Because distributed embedded systems are fundamentally different from general purpose computing systems, the chapter starts with a short overview of some of the important differences. A representation of the system objects — processing elements, software components, network communication and the system data flow — follows. We also discuss the fault model used.

Problem Definition

Chapter 5 examines the use of product family architectures to build specifications that guide system-wide customization. It includes an operational

scenario that provides useful boundaries for the development of the problem definition. Two descriptive models of graceful degradation are examined: a hardware lattice and a 3-dimensional graph which integrates system utility with the lattice. A data flow model of the product family architecture is developed.

Algorithmic Framework

Chapter 6 proposes a three phase, iterative framework for solving the system-wide customization problem. Algorithms for computation of each phase are discussed. The chapter also investigates the use of feedback from the failure of one phase to guide decisions made in other phases. Such feedback effectively guides the overall search process to significantly shorten runtimes.

Transducer Sensitive Allocation

In the search for useful algorithms to handle the third phase, allocation, we discovered a surprisingly effective heuristic that exploits some of the constraints of distributed embedded systems, namely the fixed location of sensors and actuators (collectively known as *transducers*). The resulting algorithm is documented in Chapter 7.

Proof of Concept

Chapter 8 documents the process of building a product family architecture specification for a complex distributed embedded system. Two product instances of a robust elevator control system are measured and merged to build a PFA specification. A customization manager is constructed and

tuned to customize the resulting specification. The results from several graceful degradation experiments are analyzed.

Conclusions

Chapter 9 concludes the thesis with a pragmatic overview of the techniques and a brief summary of some possible extensions to the present approaches.

1.4 Research Contributions

The research reported in this thesis contributes to the existing body of knowledge in the following manner:

Problem Identification

– This is the first comprehensive treatment of the system-wide customization problem. A formulation of the problem definition is developed.

– Ramifications of solutions are examined as they apply to the distributed embedded system domain.

Problem Solutions

– A three-phased solution framework is developed. Algorithms are presented to solve each of the three phases. Experimental results, gained through construction of a tool—called a *customization manager* — provide key parameter choices for building the algorithm.

– The entire idea of feature selection is one of the most novel aspects of this dissertation. Fundamentally, feature selection allows only parts of a specified system to be implemented, unlike other system construction techniques which assume the entire specification must be met.

– The allocation of software components to hardware (phase 3) is well known to be NP-complete. A new allocation heuristic is proposed, which exploits characteristics of distributed embedded systems, resulting in a 2.7x speedup on example systems.

Proof of Concept

– Two product instances of a complex distributed embedded system are measured and combined into a single product family.

– The customization manager is used to examine system-wide customization of the proof of concept system in response to various hardware failures.

Chapter 2

Background

Complex 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; integration is another common architectural style — especially in avionics[Di Vito97]. If the promise of MEMS (Micro-ElectroMechanical System) devices based on standard semiconductor process technology comes to fruition, it will soon be possible for many sensors and actuators to have their own integrated microcontrollers, accelerating this trend.

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 is often acceptable for optimization functions to be shed by the system as components fail, so long as this is done in a safe and controlled manner. In an automobile, for example, losing a few percent of fuel economy is often acceptable, especially when the alternative is 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 as a reduced level of system functionality.

In fact, some systems implement graceful degradation today, but use labor-intensive techniques that often involve specific engineering efforts for every anticipated failure mode [Herlihy91]. Such traditional approaches usually accomplish graceful degradation using a combination of replication and failover algorithms. Alternative approaches include multi-version redundancy and load sharing [Lyons62]. 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.

2.1 A Graceful Degradation Mechanism

Of the various means to achieve graceful degradation, we selected for our research system-wide customization as being the least esoteric and one with a fair assurance of success. System-wide customization is a compositional approach, wherein more than one level of system utility can be achieved through different combinations of basic components. Obviously, the system's architecture must be such that differing combinations are possible. Further research is ongoing into exactly what architectural styles promote graceful degradation [Shelton01, Shelton02].

By combining different combinations of components, including ones not originally examined by the designers, a system can achieve varying levels of functionality. This is the strength of an automatic system-wide customization mechanism — it is not necessary for the system designers to consider each of the combinations. Such a task is impossible for even moderately complex systems, as the number of combinations increases exponentially with the number of components. For this reason, the automatic system-wide customization mechanism must carefully choose components to combine, ensuring the dependency requirements of each component are met and the validity of the overall system functionality is not compromised.

A graceful degradation mechanism based on system-wide customization can be enhanced with additional flexibility in the availability of software components. If the system's functionality is strictly compositional, then different levels of functionality can be achieved merely by omitting some components. However, if a source of alternate components is available that

can achieve similar functionality in different manners, or even functionality that is not part of the original system, then the range of possible system states will be widened and deepened considerably.

Unfortunately, building a distributed embedded system is difficult enough that the thought of expending additional effort in order to accomplish basically the same thing in different ways is somewhat feckless. However, it is likely that a source of additional components is available — the system's 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. A PFA provides a structured view of all possible hardware components in the system [Jiao00]. Each system instance within a PFA yields a distinct price/performance point and represents a different model in the product family. 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 and targeted to minimizing production costs, rather than with an eye toward graceful degradation. See [Brownsword96] for an exceptionally good case study of PFA use in the software engineering of a ship building company.

Consider a product implemented by assembling dozens or even hundreds of different "smart" components (*i.e.,* components incorporating microcontrollers) into a fine-grained 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 the failed component missing from it. Thus, PFAs can form a conceptual framework for specifying and implementing graceful degradation within highly distributed embedded systems.

The system must be compositional for system-wide customization to be viable. In order to match standardized hardware and software within such a large variety of system configurations, we envision the components as *mobile object adapters* or simply *adapters*. They will be examined in detail in Chapter 4, but for this discussion it is sufficient to understand that an adapter is a software component that adapts or converts a device or algorithmic interface into a logical interface. Such adaptation breaks a basic combinatorial explosion in required components and allows the compositional system to be constructed from the various adapters. The adapters are mobile in that they can be hosted on any of the processing elements of the system.

2.2 The Customization Manager

In a system with an automatic customization mechanism, graceful degradation becomes fairly easy to accomplish. After each error 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 customization engine to deal with any combination of faults as it actually happens.

A customization manager has the following functionality:

Fault Discovery/System Model - The customization manager can either start with a system model and then cut out pieces whenever it discovers a subsystem is faulty (the 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 customization manager must know what sensors and actuators are operational before it builds a configuration.

Configuration Generator - The extremely large configuration space must be searched and candidate configurations intelligently choosen. In order to ensure only valid configurations are chosen, the configuration generator would generate candidates from a *dependency model* and filter them with a *validity checker*.

Dependancy 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 might 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 (processing and network resources) than are available.

Cost Model - Allows comparison of various configurations. Cost models may be fairly complex, as they may become scenario specific.

Adapter Repository - A library that holds all of the adapters available in the product family architecture.

Device Customization - An adapter loader deploys the chosen configuration throughout the system. Over the low bandwidth networks common to distributed embedded systems, component migration opportunities may be limited. In such cases, the deployment may consist of transfering small bits of state to prepositioned executables.

The Fault Discovery, Adapter Repository and Device Customization functionality is not covered in this dissertation. However, they are often included as mechanisms in various middleware projects. See [Beveridge02] for a discussion of several middleware systems as well as an examination of their applicability to distributed embedded systems.

The point in time when automatic customization is executed must be carefully managed. The cost of running a customization manager to determine the the appropriate configuration can be significant. While we would like to be able to build a run-time customization manager, we find doing so to be overly burdensome and only rarely required. Especially in the case of a tightly scheduled and resource constrained system, there might not be enough resources (CPU or network cycles, timing slack, etc.) to actually

execute a customization step. Instead, we envision automatic system-wide customization employed during extreme duress or down time. In the case of a crisis, breaking schedules to run the customization manager makes sense in that the system would be completely broken and have no chance of fulfilling its mission otherwise. Running the customization 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 automobile, for instance, system-wide customization may occur when the engine is off — perhaps when the car has been pulled to the side of the road to deal with the emergency. An alternate approach might employ an incremental customization manager that can, in a series of steps, make small changes in the system configuration and eventually arrive at a high-quality configuration.

2.3 Customization as Logistical Support

Once a system has a system-wide customization mechanism, it can be exploited to provide a potentially major logistical benefit — the ability to make replacements with non-exact spares. If achieved, this would free maintenance personnel from the burden of carrying every conceivable spare part. For example, they might just carry more capable, and expensive, generalized spares instead of cost-optimized specific repair parts. (But, reduced labor costs for trips to pick up spares could easily offset any increased component costs. Likewise, inventory costs would plummet.) In emergencies, subopti-

mal 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 seven year spare parts pipeline is mandated for automobiles, subjecting vehicle manufacturers to interesting factory utilization challenges. Manufacturers must weigh the warehousing costs of spare parts with the need to keep a factory line hot for the parts. This mandate will be increasingly challenging as more and more automobile subsystems involve digital electronics – entire fabrication processes may need to be kept operational far beyond their obselecence merely to provide spare parts designed a decade earlier.

An automatic system-wide customization mechanism may ease such logistic nightmares. Rather than replacing a part with its exact duplicate, a non-exact spare may be employed. The customization 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, system designers will fulfill requirements for legacy spares. Such a situation is analagous 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 customization manager determines configurations that can use the added resources to restore functionality.

In addition, system-wide customization 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 customization are simply ways of moving down or up the lattice of points in a product family architecture. When some hardware breaks or is inserted, it is as if a different model in the PFA has been realized. The customization manager then can determine the best collection of features to install on available hardware.

2.4 Problems with Customization

System-wide customization 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 that could be solved with the application of more thought power. Others are fundamental to the types of systems being built and will remain formidable barriers for those applications.

2.4.1 Loss of Design Determinism

Design determinism is the control that a designer has over the state of the end product. At the beginning of a development process, the system architect and designer envision the final product and vary their design to ensure their vision of the final product meets the requirements. At first examination, it appears that using a system-wide customization framework for a product decreases the linkage between architect's vision and final product, and to a certain degree this impression is correct. The designer can envision how the final product operates in one particular system state, but cannot imagine every possible system state that might result from hardware failures and the follow-on system reconstitution of a customization operation. By implementing system-wide customization in a system design, the designer loses some degree of design determinism — he must rely upon the customization algorithms rather than his own design abilities to ensure the system state is desirable and proper. The problems discussed in several of the following sections are a result of or similar to the loss of design determinism.

The loss of design determinism is mitigated to a large degree by the realization that the system-wide customization process and algorithms employed on a system are well within the purview of the system designer. If the customization process does not generate the end system states envisioned by the designer, then the algorithms and the process should be modified. The situation is analagous to the rise of the early source code compilers. Programmers who were well versed in the assembly languages of their sys-

tems often did not trust the compilers to emit proper or efficient assembly code. The linkage between their vision of what the assembly code should look like and the code actually emitted by the compiler was quite weak. The situation did not improve until compiler technology matured and the programmers learned to trust their compiler. Perhaps the same can be said of system-wide customization. A great deal of the design determinism might be regained when customization technology matures and system designers trust the algorithms of the customization manager.

2.4.2 Development for System-wide Customization

It is possible that the design of a system with system-wide customization capabilities will be too disruptive to the design process. This dissertation makes certain assumptions about the development process: *e.g.,* that a product family architecture exists and that data flow is an important enough aspect of the system to be well-documented. If a system under development did not fit into a product family, or if it could not properly be described in data flow diagrams, then the techniques described herein may need to be modified to be useful. It is possible that the system architecture might be molded in the wrong way in order to take advantage of customization. For instance, a transactional system might be cast as a data flow system. A better alternative is probably to expand the research to develop additional customization techniques that fit the transactional system. More detail of this point is available in Chapter 9.2, where it can be treated with more depth.

Building a system with system-wide customization capabilities will in-

volve a slightly different development process from that used to build a standard distributed embedded system. Designing and generating product family architecture artifacts, such as the data flow graphs, may entail additional time or expense. The testing and certification, not only of the customizable system, but also the customization mechanisms themselves, is more complex, as discussed in the following sections. Nevertheless, we feel that today's distributed embedded systems have reached a level of sophistication that merits this type of graceful degradation mechanism, and they have enough extra resources and resource fragmentation to merit searching for alternative configurations that may benefit the end user.

2.4.3 Debugging and Technical Support

The problems with debugging and maintenance are quite similar to the issues surrounding design determinism. The existence of a customization mechanism allows for a wide variety of system states, and determining proper system operation in each is impossible — after all, doing so for a single configuration is often infeasible. The proliferation of configurations raises another level of complexity for the debugging process to overcome. In essence, the debug, technical support, or maintenance personnel does not have an accurate system model that accounts for all possible system states. They must first examine the system state and construct a system model, before beginning the debug, support or maintenance process.

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 com-

patibility, customization is much easier when the adapters fit well defined and properly abstracted logical interfaces. The extent to which architecture can support customization is an interesting research problem [Shelton01, Shelton02].

Technical support can also be a challenge. When an error or problem is reported by a user, knowledge of the current configuration is likely to be useful. The customization manager must scrupulously log all configuration changes and make the configuration data available to the problem resolution team. This can be a problem if there are frequent configuration changes. In a system where system-wide customization is only executed during maintenance, the configuration data will be easier to maintain. If, however, customization happens often, say whenever a vehicle is started or whenever an elevator door is opened, then it is a difficult process to track exactly what the contents of the configuration were when the problem occurred.

2.4.4 Certification Challenges

Many safety critical applications need to be approved by a certification authority. In the USA, nuclear power plants must pass specification, design and implementation verification by the Nuclear Regulatory Commission. Avionic systems are certified by the FAA, while some security systems are in the purview of the NSA. A system-wide customization mechanism might increase the certification costs, as the developers now must ensure the certifiers are comfortable with the customization mechanism and the manner in which configurations are chosen and deployed.

The real gains from system-wide customization come when the next ver-

sion of the product must be certified. If the regulatory agency understands customization and is comfortable with the implementation, then recertification is merely the process of checking that any changed subsystems live up to the same logical interface.

In the case of safety critical systems, system-wide customization may be deemed too great a risk. In such a situation, a different approach is still possible. As in so many safety critical systems, a separation strategy can easily be pursued. Such a strategem is accomplished by ensuring the safety critical functionality is partitioned away from all other features. This is common, 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. Following such a strategy requires careful attention to possible second order feedback that might unexpectedly promote non-safety critical functionality. For instance, a mobile pager is not usually considered safety critical, but hospitals often use them to make sure doctors are summoned for emergency surgery.

2.4.5 Multi-vendor Challenges

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

Designing a system for cross-vendor system-wide customization is quite a challenge. At its core, system-wide customization takes advantage of some

extra resources, provided by design or by freeing them from lower priority uses, to install functionality. In a multi-vendor environment, the extra resources might be taken from one vendor's unit in order to provide extra functionality to a unit from a different vendor. The first vendor would likely object, as the cost to provide the resources makes the unit more costly compared to any non-reconfigurable competitors.

It is not clear how the liability for an accident or failure would be allocated in a system-wide customization capable system. Determining the origin of the error is complex, as described in Section 2.4.3, and more subtle problems will probably stem from the all-too-common lack of good communication among organizations. In general, if a module written by vendor A was installed on vendor B's device by a customization manager provided by vendor C, a jury could easily find any of the parties liable in the case of an incident.

2.5 Conclusion

This chapter has provided an overview of the motivations for designing a customization manager for use in distributed embedded system. Graceful degradation can be accomplished by customizing the system at any change in available hardware. If the system-wide customization process truly maximizes the functionality of the synthesized system, then the system's utility will gracefully degrade as hardware components are lost. System-wide customization also brings interesting logistical benefits, such as a decreased reliance upon legacy spares and the ability to maintain and repair a system

with non-exact spares. In order to become a widely employed technology, system-wide customization will need to overcome or avoid some challenges, however. Getting to that point is an intellectually stimulating research agenda that has the possibility of substantially changing how embedded systems are designed, implemented and deployed.

Chapter 3

Related Work

This chapter considers four areas closely related to the contents of this thesis: graceful degradation, reconfiguration, hardware–software codesign, and task allocation.

3.1 Graceful Degradation

Systems that gracefully degrade lose partial functionality in response to failure events, as opposed to failing precipitously. Little research has been reported on building computer systems with mechanisms for graceful degradation. In cases where graceful degradation is desired (usually military, space or industrial systems), enormous effort is required by the system designers, who must examine every fault hypothesis and design an appropriate system response for each. [Borgerson75] developed a model for analyzing the reliability of gracefully degrading and standby-sparing computer systems. The model is useful as a means to analyze a system's reliability parameters, but

does not provide any assistance in determining *how* to build a gracefully degrading system.

By far the most insightful paper dealing with graceful degradation is [Herlihy91], which examines a specification method. The method revolves around a *relaxation lattice* of system states, the partial ordering of which determines degradation trajectories. As environmental changes restrict a system, the system responds by relaxing the constraints of its specification along the paths allowed by the relaxation lattice. The contribution of this paper is to posit a lattice model for thinking about graceful degradation. Unfortunately, the mechanisms for construction of the lattice, as described, do not scale to larger systems. For instance, the relaxation lattice must be examined by the designers for an appropriate system responses for each degraded state.

[Knight00] examines system *survivability* for large scale infrastructure systems (*e.g.,* banking, rail transport) with an eye towards strengthening them against security and environmental faults. Survivability is similar in scope to graceful degradation — the infrastructure system must, in Knight's words, undergo

> damage and repair sequences. Events that damage a system are not necessarily independent nor are they necessarily mutually exclusive. In practice, a sequence of events might occur over time in which each event causes more damage in effect, a bad situation gets progressively worse. It is likely, therefore, that a critical infrastructure application will experience damage while it is in

> an already damaged state, and that a sequence of partial repairs might be conducted. Thus, a series of changes in functionality might be experienced by a user with progressively less service available over time as damage increases and progressively more available as repairs are conducted.

This prediction of damage and repair sequence is remarkably analagous to the degradation trajectories of the distributed embedded system we are concerned with. Unfortunately, [Knight00] does not provide scalable tools for construction of survivable systems. Rather a list of *reduced functionality states* is generated, along with specifications for the system behaviours in those states. A state machine is then designed to show how the system should change states to react to failures.

[Shimomura95] is a fascinating examination of a commercial copy machine with something approaching graceful degradation capabilities. The self-maintenance copy machine used a detailed physics model and an inference engine, such as one might find in artificial intelligence research, to reason about failures. The failures were in the sensor and manipulators, not the computational engines. The machine could diagnose errors, simulate overlapping fault cases and execute limited repair actions. It is unclear if this method scales, because quite a bit of effort went into designing fault cases that could be reasoned about.

The difficulties of providing graceful degradation are well covered in [Cailliau99], a study of the customization mechanisms in a (single processor) satellite system. In this particular case, customization is handled

through pointer redirection and controlled via ground segment command, not automated methods.

A complimentary (to this thesis) research project is underway, as reported in [Shelton02], to develop system architectures that are amenable to graceful degradation, either passively by their construction or actively through methods such as system-wide customization. That research seeks architectural patterns and principles which will aid designers to measure and reason about the dependability strengths of a design.

The emerging field of *amorphous computing* is somewhat related in their desire for automatic graceful degradation and organization mechanisms [Abelson00]. That research field envisions a future where vast quantities of microsensors will form embedded systems, for instance when placed inside construction material such that buildings and bridges can be monitored for excessive stress or internal faults. Amorphous systems will need to be self-organizing and able to withstand failure of individual computing units.

The research in this dissertation advances the state of the art in graceful degradation system design by achieving automatic graceful degradation—that is, graceful degradation mechanisms that require no state-by-state design of the degradation trajectories of the system.

3.2 Reconfiguration and Customization

System-wide customization is not yet a research field with any prior results. It is very similar in concept to a case study reported in [Beck00]

and [Reagin99]. These papers examined the construction of a robotic work-cell application using the customization opportunities of a component-based software architecture, with impressive results. Some of the optimizations done manually by the application engineer are similar to the operations executed by the system-wide customization algorithms of this dissertation.

Another quite active related research field is "reconfigurable computing." In this context, reconfiguration generally is understood to mean altering the form of hardware — often a Field Programmable Gate Array or (FPGA) — in response to program constructs [Green00]. Such alteration may be computed and scheduled statically (by a compiler [Li00]) or dynamically (by the operating system [Bapty99, Bazargan00, Dave99]). The hardware that is reconfigured is not constrained to only FPGAs. [Goldstein00] utilizes an extremely innovative hardware architecture. However, the field of reconfigurable computing is still primarily focused on single processor systems (or low numbers) with special hardware. Further, the goal is to maximize performance of a software specification, all of which must be executed.

Our approach is different in that we attempt to reconfigure (customize) an entire system of general purpose microcontrollers in order to maximize functionality. We also relax the requirement that all of the software specification must be met. In order to limit confusion with the reconfigurable computing research field, we have adopted the term system-wide customization for our work.

3.3 Hardware–software Codesign

Hardware–software codesign is the design of special-purpose systems composed of a few Application Specific Integrated Circuits (ASICs) cooperating with software procedures on general-purpose processors [Chiodo94]. It is a design-time technique, whose goal is to reduce the cost of a system by carefully examining the partitioning of functionality into hardware and software. It is an active research area, with impressive results [Gupta93, Thomas93, Edwards97]. [Ernst98] is a good survey of the field. [Hu94] reports results of the use of codesign techniques to develop automotive powertrain control modules. Automated techniques are sought to make the partitioning desisions [Knieser96], but often they must be made by designers and simply tested by the codesign techniques [Kalavade93, Chiodo94, Pimentel01]. System examples in this field tend to be fairly small — often only 15-20 objects, so scalability of automated mechanisms is an active concern [Wolf97].

The system-wide customization research of this dissertation is fundamentally different from the codesign field. Customization attempts to maximize functionality of specified hardware, while codesign tries to minimize the hardware requirements needed to implement specified system functionality.

3.4 Task Allocation

Task allocation is related to the well-known bin packing problem. In even a two-processor form, it is NP-complete [Garey79]. Because of the applicability to OS scheduling on multiprocessor systems, a great body of heuristic algorithms and analyses exists, such as [Kasahara84, Woodside93, Stone77,

Shen85, Bokhari81, Bokhari88, Indurkhya86, Efe82]. See [Kwok99] for an excellent bibliography of such efforts. The two basic approaches to solving bin packing problems are list processing, where the objects are sorted and placed in the bins according to their order, and guided search, such as simulated annealing, where an initial solution is incrementally improved. The algorithm presented in this dissertation, in Chapter 7, is a list processing algorithm.

In a multi-processor scheduling algorithm, the metric being optimized is generally the length of the critical path schedule. All parameter values used for the allocation are time units for each task to process or for communication to be transmitted. In contrast, in a distributed embedded environment the algorithmic interest is to ensure tasks can execute together on the limited, fixed resources of the microcontrollers.

Task allocation is is also a critical building block in hardware–software codesign research — where software is allocated to hardware to test if a partitioning decision is correct [Kalavade93, Gupta93, Hu94]. [Lee95] goes so far as to name a codesign process "dataflow process networks", as it is based on the data flow through the system, much like our research.

The development of the transducer sensitive allocation algorithm, in Chapter 7, is based to a large degree on [Beck98]. Beck used a design advisor (DA) algorithm to generate a system hardware specification to meet the requirements of the software. The DA algorithm bin-packed vector valued software requirements (*i.e.* CPU cycles, RAM, ROM, I/O channels) into multi-dimensional bins representing the resources of the microcontrollers. Whenever the packing algorithm failed, the DA would expand the hardware

specification. The basic idea is similar to much hardware/software co-design research — allocate software to the hardware to test if a partitioning decision is correct [Kalavade93]. Prakash used linear programming techniques for a similar problem — simultaneous specification and allocation — though the application of such techniques to problems with large numbers of tasks appears to be computationally challenging[Prakash92]. The DA algorithm was extended to cover multi-network specification in [McNally98].

The large size of this research area has spawned at least two attempts for standardization of the system descriptions. The use of standard task graphs facilitates benchmarking and comparison of the allocation algorithms. Kwok, *et. al.* collected 11 graphs from published papers (all of 7–18 vertices in size), combined them with a large number of randomly created graphs, and proceeded to benchmark the various scheduling algorithms[Kwok99]. Unfortunately, we have been unable to obtain this graph set for use in this research. The "Standard Task Graph" project has randomly generated a set of large graphs (30–2700 vertices) for the same purpose[Tobita00]. The standard graphs with communication costs, which would be most useful for our research, are not yet available.

Our work builds upon the volumnous research of this field, yet incorporates the constraints of a distributed embedded system. Chapter 7 describes a transducer sensitive task allocation algorithm that exploits the fixed location of sensors and actuators in a distributed embedded system in a "divide and conquer" style algorithm. The result is a healthy 2.7x speedup over other task allocation algorithms.

Chapter 4

System Model

The system-wide customization concept relies quite heavily upon the system model used. Without sufficient flexibility, system-wide customization is overly constrained and always results in the same configuration. Yet too much flexibility creates a combinatorial explosion of dependent software components leading to excessive difficulties on the part of the customization manager. To manage the complexity, we require decoupling of inputs and outputs through the use of logical intermediary interfaces. This chapter is, at a fairly low level, an architectural description of the system components. The next chapter discusses how to extend this model to include product family architectures.

A distributed embedded system is somewhat different in form from that of other distributed computing systems. Such differences lead to an unfortunate number of misunderstandings when practitioners unfamiliar with distributed embedded systems attempt to transfer their system knowledge and research agendas into the domain. While the diversity of embedded sys-

tems resists a simple classification that covers all product instances, most distributed embedded control system are similar to the model we have used.

4.1 Overview

All embedded systems attempt to control the environment, by crafting a control loop where data about the environment is sensed, operated upon by the compute elements of the system, and the environment modified via actuators. Figure 4.1 illustrates the control loop. Distributed embedded systems merely deploy a multicomputer computing facility, consisting of an embedded network and multiple processing elements, to provide the computational power of the system. In the very near future, smart sensors and actuators will make up most of the processing elements of industrial distributed embedded systems.

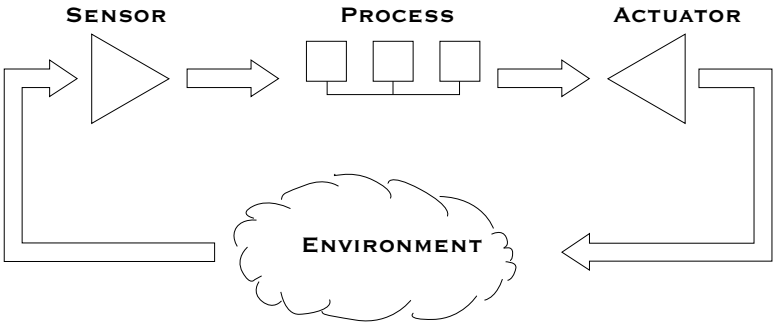

Figure 4.1: The distributed embedded system control loop

4.1.1 Embedded Networks

A distributed embedded system consists of one or more networks connecting two or more microcontrollers (or, more typically, dozens), each of which

manages one or more sensors or actuators. This work will assume a single network, though multiple networks can be incorporated using the techniques examined in [McNally98].

Embedded networks differ from the typical Ethernet or ATM networks commonly found connecting general purpose computing platforms. Embedded networks generally require a means to ensure real-time deadlines are respected, and thus require a great deal of determinism. Collisions (simultaneous desire to send a message on the part of two or more senders) must be avoided or resolved in a deterministic manner. Two network protocol mechanisms are commonly used to meet real-time requirements: time division or bit dominance. Time division protocols, such as the Time-Triggered Protocol (TTP) [Kopetz], reserve transmission slots of set period and phase for each potential sender, who is then restricted from transmitting at any other time. Bit dominance protocols are represented by the Control Area Network (CAN) protocol [CAN] which uses bit-by-bit collision detection to form a global consensus of which pending message has priority for transmission.

Embedded networks are also typically broadcast networks, use low payload sizes, and support lower bandwidth than the typical computing network. The broadcast nature naturally results from the single link networks where each node uses a hardware filter to ignore unwanted messages. Payload sizes are small in order to reduce latency and because the required data messages need not be large. The messages are generally about the state of a particular controller or a sensor measurement — a few bytes usually suffices. For instance, CAN message payloads are 0–8 bytes and TTP messages are a maximum of 16 bytes. Available bandwidth is similarly low. The maximum

CAN bandwidth is 1Mbps, though it is often used at a mere 125Kbps.

Such characteristics of embedded networks are important factors to keep in mind when judging the success of distributed embedded system research. The system model used in this thesis is a 1Mbps CAN network. All data elements have a unique type, which is transmitted as the message ID. As discussed later, no scheduling analysis has been done, so messages are assumed to have the correct priority for the application, without explicitly determining what that priority is. Typical priority assignment can be accomplished through deadline monotonic analyisis [Tindell00], earliest deadline first, or other schemes [Zuberi00].

4.1.2 Smart Sensors and Actuators

As discussed in Section 2, distributed embedded system consist of numerous "smart" sensors, each of which is a computing engine in its own right. These microcontrollers manage the raw sensor signals, converting them to the logical values necessary for consumption by other software components. Such conversion may be a simple formatting change, where the raw information is placed on the network with minimal work. In other cases, sophisticated conversion may include Kalman filtering, temporal averaging, or sensor fusion. Such conversion is accomplished via a software *adapter*, responsible for adapting raw information to the logical values required by the rest of the system. Similarly, smart actuators use adapters to convert logical system variables into the raw control signals that drive physical processes to affect the environment.

Each adapter operates within the computing regime of a microcontroller.

The microcontroller has some runtime elements, hardware interfaces to its sensors and actuators, and a network connection. Figure 4.2 is a high level picture, showing the structure of an example *processing element* (PE).

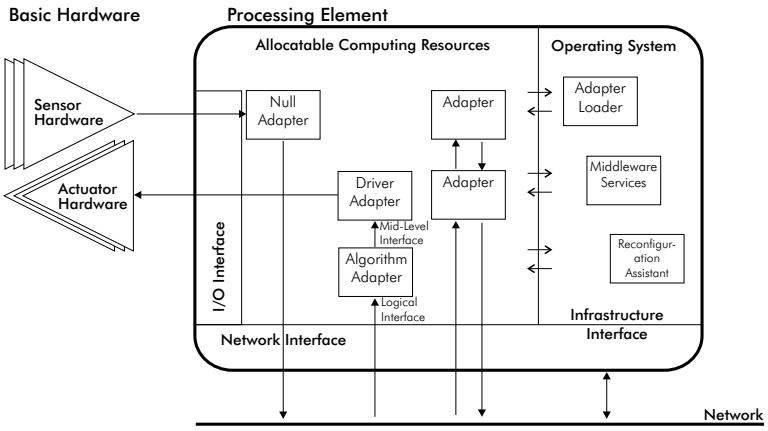

Figure 4.2: Anatomy of a PE

The *basic hardware* is the core of the sensor/actuation facility – it does something useful to the real world. We do not attempt to do any sort of system-wide customization on the basic hardware – we are stuck with it until a maintenance operation repairs or replaces it.

The basic hardware communicates its results to the first level adapter (a "driver" level) in its own format via an *I/O interface*. This format is specified by the sensor or actuator, and thus is often proprietary and considered fairly "raw." For instance, the output of an accelerometer may be just a voltage level. But when put in context through a table lookup or arithmetic calculation, can be converted to express the g-force (or m/s^2 or whatever units are desired) that was measured. The *driver adapter* can convert this format and any associated timing or semantics into a logical

interface, known as a *mid–level interface*. Driver adapters also do the reverse conversion, from logical to raw, for actuators. Note that several driver adapters may be capable of doing such transformations, and perhaps to several different mid-level interfaces. It is reasonable to believe that a complex driver adapter could support several mid-level interfaces, while a slimmed down version could only support one. Conceivably, quality attributes could also separate otherwise similar mid-level interfaces.

Next are the *algorithm adapters*. One or more such adapters will be loaded to transform mid-level interface information to something that other nodes of the network would like to hear. These adapters are generally of higher complexity, and form the core of the flexibility required for system-wide customization. They are most often replaced or moved to other PEs to meet reconfiguration goals. A system may have an arbitrary number of algorithm adapters — this is a catch-all term to cover all the other software components.

It is important to note that both the driver adapter and the algorithm adapters are mobile. Driver adapters may be replaced with, for instance, a *null adapter* whose sole responsibility is to communicate the raw sensor information to the network and thus to a driver adapter which is hosted on a different PE. In Figure 4.2, for instance, a null adapter recieves the raw information from a sensor and transmits it via the network to a remote driver adapter (not shown in the figure). The term *Mobile Object Adapters* (MOAs) is sometimes used to refer to both driver and algorithm adapters. We also use the term *adapter* to include null, algorithm and driver adapters.

The *infrastructure interface* provides access to the collection of runtime

services that assist with customization. These services form a system-wide customization runtime executive, which could conceivably be a portion of an embedded operating system or monitor. Implementations of the infrastructure include an *adapter loader*, which is responsible for retrieving adapters from the network or activating adapters already in storage on the node. Middleware services, such as *Jini* (Javasoft's technology for delivery of adaptive network services) may assist to some degree with discovery and lookup. A *reconfiguration assistant* is the local representative of the reconfiguration manager. It notifies the customization manager of any changes in PE status (*e.g.,* broken sensors, reconstituted PE), PE capabilities (*e.g.,* more RAM available), etc.

The *network interface* is responsible for getting messages to and from the network. It handles (and thus abstracts) all of the details of network protocol. It can also route messages within the PE so purely local communication does not pollute the network. Advanced network interfaces can also combine signals from various adapters into a single network message. The network is also used to load the adapters from an adapter repository. In special cases where mobile adapters are not available, prepositioned adapters may be available in a local ROM repository on some PEs.

4.2 Representation

Three elements must be represented in the model: the network, processing elements,. and adapters. We follow the general approach of [Beck98] to model these elements. Beyond that, the flexibility present in the Product

Family Architecture must be represented.

4.2.1 Processing Elements

Our processor model is very high level, and thus easily managed. It consists of a resource vector, of arbitrary length n. Each element of the vector is a consumable resource, such as RAM, Flash Memory, or I/O channels. The number of dimensions in the resource vector is arbitrary, but must be consistent among all processing elements. [Beck98] collected data for 6-way vectors: CPU Cycles, ROM, RAM, digital I/O, analog I/O and PWM I/O. Note how the use of CPU cycles allows a performance characteristic to be included, merely by recasting performance as a consumable.

While use of a single dimension in the vector is not ruled out by this model, interesting real-world issues arise when multiple dimensions are used. The allocation algorithms must be able to deal with, if not exploit outright, the tension between the competing demands inherent in a multi-dimensional resource vector. For this reason, we have ensured all experiments were executed with $n \geq 2$.

Sensors and actuators are physically connected to particular processing elements, and require computational resources for proper execution. Such resources are considered to be pre-allocated and not included in the PE's resource vector.

4.2.2 Network

Our model of the network is a simple resource vector. In the cases illustrated in this thesis, only a single network resource was modeled — bandwidth. The

resulting single element resource vector thus collapses into a single scalar. There is no reason precluding the addition of further consumable resources in the vector, if called for by a particular design.

Latency and other scheduling concepts are difficult to apply to this network model. Ideas on how to expand the network model suitably are discussed in the future work section, Section 9.4.

4.2.3 Configuration

We base the model of software elements on the flow of information from sensors, through software elements, to the system actuators that actually modify the environment, as shown in Figure 4.1. For this purpose, we employ a synchronous data flow graph (SDFG)[Lee87]. The SDFG is a directed, possibly cyclic, graph, where vertices are algorithms and edges the data flow between them. Exterior vertices (those with only inputs or outputs) represent sensors and actuators — sources and sinks of data. Figure 4.3 shows an example SDFG, representing a frequency division multiplexed, full-duplex modem. In this respect, *synchronous* merely refers to the *a priori* ability to describe the rate at which data is generated or consumed by each vertex. In other words, each adapter must consume and generate data at a rate that is independent of the actual contents of the data. As most distributed embedded systems operate in a time-triggered fashion, such an assumption is fully warranted.

A well-formed configuration can be represented by a unique DFG arranged to show the interconnections among sensors, adapters and actuators. This is a fairly common representation of embedded systems and is

used traditionally in signal processing applications [Woodside93], hardware-software codesign [Prakash92, Kalavade93, Blickle98], and elsewhere [Efe82, Beck95, Beck98]. It was first proposed for task (adapter) allocation algorithms in [Stone77]. Note that interior vertices are all adapters and exterior vertices are sensors and actuators.

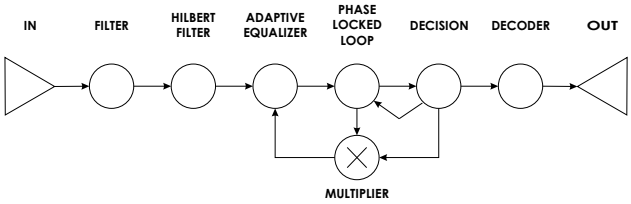

Figure 4.3: A simple data flow graph

Each interior vertex of a configuration's DFG represents an adapter, which must be allocated to a processing element. The exterior vertices are the sensors and actuators — they are pinned by their physical connections to a particular processing element, and thus do not require allocation. Each adapter, however, needs a specification for the computing resources it will consume when placed on a processing element. The specification takes the form of a requirements vector, where each element of the vector indicates the amount of a particular type of resource that the adapter requires. The requirements vector has the same dimensionality as the resource vector used in the PE model. In fact, the elements correspond between the two vectors — requirements will be satisfied by consuming from the resources vector. For example, if CPU cycles are the first element of the resource vector for the PEs, then CPU cycles are also the first element of each adapter's requirement vector.

The edges of the data flow graph represent communication between adapters, sensors and actuators. Each edge is directed in order to distinguish the source and destination of the communication. Edges are labeled

with a requirement vector, in the same dimensionality and element order as the network resource vector.

4.3 Fault Model

Since our basic idea is to respond to reliability problems in a system, we must be careful to cover the types of faults that system-wide customization is capable of responding to. The reliability field uses the following system fault models (in addition to numerous other local hardware fault models such as "stuck at one"):

Omission failure a processing element or software component fails to generate an output.

Timing failure an otherwise correct response is generated either too early or too late.

Value failure the value of a response is incorrect.

Crash failure a processing element fails to generate outputs.

Byzantine processing failure results in arbitrary, even malicious, behavior.

See [Cristian91] for a detailed examination of fault semantics, or [Gartner99] for a higher level survey.

The system-wide customization concept and operational scenario have been conceived as a defense against the failure of a microcontroller or transducer. Thus, failure of a PE, sensor or actuator must be allowed (and detectable) by the fault model. For the most part, we consider only crash

failures. It could be that a timing, omission or value failure which was detected by the system would result in a PE being shut down and cause a system-wide customization. On the other hand, network failures are not addressed, as it is critical for moving the adapters to their allocated target PEs and communicating other results of the customization process. The network is also necessary to gather information about the operational status of each PE, sensor and actuator prior to starting a system-wide customization operation.

4.4 Summary

This chapter introduced the system model that was used for this research. The model was carefully based upon the dominant traits of industrial systems: embedded networks and the use of microcontroller-based smart sensors. The system model includes:

- capability vectors for processing elements,
- a scalar bin to represent the bandwidth of the network,
- requirement vectors for software components,
- and a simple data flow graph to represent composability of the system from the data manipulation elements (sensors, adapters and actuators).

Chapter 5

Problem Definition

This chapter examines the problem of graceful degradation and puts together some of the building blocks that lead to a thorough understanding of the problem. First, graceful degradation is related to the real world through descriptions of an operational scenario. Such an example scenario is illustrative of many subtleties, and helps to guide implementation decisions. An automobile navigation system is also introduced as an example system. Several problem models are discussed, leading to the mechanics of the PFA graph, a representation of the customization opportunities available in a system. It is from the PFA graph that algorithmic solutions are built in the next chapter.

5.1 Guiding Scenario

5.1.1 Operational Scenario

Currently available automobiles have no system-wide customization capabilities, of course. But they do consist of the distributed embedded architecture on which customization can be constructed. In high end automobiles, current year models have as many as 75 microcontrollers. Designers anticipate that in a few years, even low end automobiles will employ 50 to 100 smart sensors connected by a CAN network[Leen02].

In the customization-capable automobile, the scenario in the case of a fault would unfold along these lines:

1. A hardware failure occurs. We assume the driver has enough system functionality remaining to pull the car to the side of the road.

2. A customization manager is connected. The connection is either via a remote connection such as OnStar^{TM} or to a laptop carried in a tow truck.

3. The customization manager polls the system to discover what hardware is available.

4. The system-wide customization algorithm is executed. The output is a list of adapters to be downloaded to particular hardware.

5. Adapters are downloaded and installed. The adapters are taken from the adapter repository, either on a remote server or a CD-ROM in the

laptop. Note that the adapter repository might contain adapters that were not necessarily available when the automobile was constructed.

6. The driver reboots the auto and goes on his way.

This scenario places a few bounds on what is to be accomplished by a customization manager— static timing, computational location, and execution time. The timing is static in that the system is not performing its mission during the actual customization process. Building a dynamic system for system-wide customization is quite a challenging proposition. The root of the problem is a lack of computational resources to execute the customization manager or the lack of network bandwidth to transfer mobile adapters to new host PEs. The lack of resources (computational and network) is due to previous system-wide customization actions which would have allocated the resources to increase system functionality.

Fortunately, most distributed embedded system have a *ground state* which can be exploited. [Kopetz97] defines *ground state* on a node-by-node basis as a state where no task is active and no messages are in transit. In a system-wide context, the ground state is a time of reduced functionality when most nodes are generally idle and the network is fairly free of messages. In the example above, the auto reaches a ground state when turned off at the side of the road. Elevators, for instance, can execute the system-wide customization during the time when passengers are loading onto a motionless car. Systems designed for reliability routinely have such ground states on a node-by-node basis, as they help with checkpointing the state of the PE.

We also don't expect every auto to have the data storage or computational capability of a customization manager onboard — quite a bit of processing power would be required to run the algorithm. Perhaps an automobile's infotainment computer could be put to good use, but access to the repository of all current adapters would be problematic. Remote access capabilities are being added to many distributed embedded systems, which can be used for communication with a reconfiguration server. In many other cases, the customization manager may be hosted in a diagnostic tool, such as might be available to an auto mechanic.

Finally, this scenario helps to put bounds on the execution time — several minutes would be available, but not much more. There is no reason to expend significant effort to speed it up into the sub-second range.

5.1.2 An Example Subsystem

A hypothetical automotive navigation system is a simple, though non-trivial, example of the ideas presented in this and the following chapters. It was used to guide the development of many of the customization algorithms. This model has been vetted with an industrial partner so we believe it is fairly realistic and representative of current system capabilities.

Two navigation applications are posited: location detection and path planning. The system can indicate the automobile's current location through a special purpose display unit, showing location on a local map segment. The display could also be used to provide a turn–by–turn set of directions for path planning. In that case, the display would indicate the direction of and distance until the next turn (*e.g.,* "Turn left at Wilson Avenue in .4 miles").

Path planning information could also be provided to the user via several different system actuators, in the case of damage to the display unit or if the display were being used to provide the location map. For instance, a speaker may aurally indicate the turn, either by speech synthesis or via special tones (*i.e.,* high pitches mean turn left, low pitches turn right. Loudness could indicate distance to the turn). Finally, the turn indicator can actually be used to notify the driver of an upcoming turn. By blinking (perhaps in a different pattern from the standard turn indicator) the driver can be told to turn left or right.

The following sections and chapter contain more details of the navigation subsystem, which is used to illustrate the corresponding ideas.

5.2 Problem Models

We have developed several informal models to help us think about and communicate about graceful degradation. The lattice and MUSH models that follow are descriptive (as opposed to prescriptive) in nature, but have been quite helpful. Neither model would actually be built for real systems — they are far too large for human construction. But thinking about how the models operate helps to clarify many graceful degradation concepts.

5.2.1 The Lattice

The fixed hardware resources of a system limit the software components that can be executed. Each choice of possible hardware components can be viewed as a single vertex of a dense lattice that represents a fine-grained

product family architecture (PFA). Figure 5.1 is a subset of an example lattice. The lattice is a partial ordering based on the number of components, with arcs connecting those combinations of components (or configuration) that differ in only the addition or subtraction of a single component. The system's configuration state falls toward the bottom of the lattice as components are broken, and rises when they are repaired or replaced. Such movement makes no statement about the desirability of different configurations, though it is often the case that "more components" is better than "fewer."

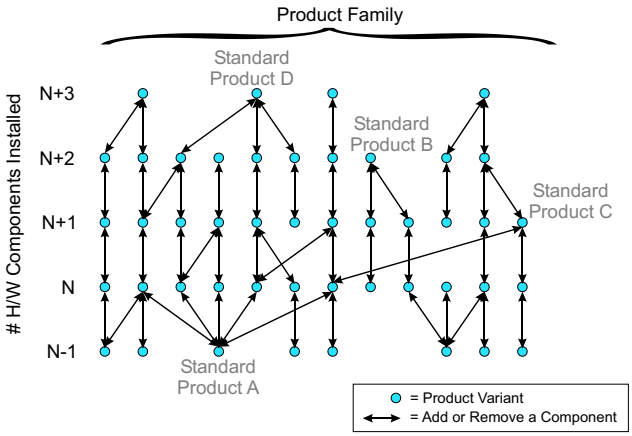

Figure 5.1: Part of an example lattice

For each combination of hardware components, there are many different software configurations available. In terms of the PFA lattice, the system-wide customization problem may be expressed as the process of choosing the software configuration for a particular vertex (*i.e.,* the one representing available hardware) that maximizes the utility of the system. In general, there are many different combinations of software components that could be used at each vertex. Unfortunately, the lattice model doesn't help determine which software configuration would be the best.

As an aid in determining which software configuration would be best, it is tempting to formulate a lattice model from software as well as hardware

components. Such a model is less intuitive, in that many lattice vertices are invalid (in that the proper software components are not available to operate the given hardware) or contain excessive software components. However, once the invalid configurations were weeded out, utility values could conceivably be assigned to each vertex. Graceful degradation would then simply be the process of moving the system state down the lattice to a neighboring vertex having the largest utility and all the available hardware. Such a lattice is fairly close to the relaxation lattice described in [Herlihy91]. It still has all the same problems, as well: an inability for human designers to examine the large number of vertexes and the complex process of actually generating consistent utility values. An alternate model, which integrates the hardware and software elements of the system, is given in the next section.

Both lattice models have the advantage of being an intuitive vehicle for discussions of graceful degradation. One naturally can imagine a system degrading as its hardware state falls down the lattice. Recall, however, the counter-intuitive nature of higher utility states possibly being located at lower levels of the lattice. This is a direct result of the lattice's partial ordering being based on number of components, not the utility of individual components.

As an example Figure 5.2 shows a small portion of the overall lattice for the automotive navigation system. Notice that the top-left collection includes a GPS and a compass sensor. The compass is thus completely redundant, and its loss results in a usable system state. However, if the second loss includes the GPS, then no source of location information is available and the system is not functional.

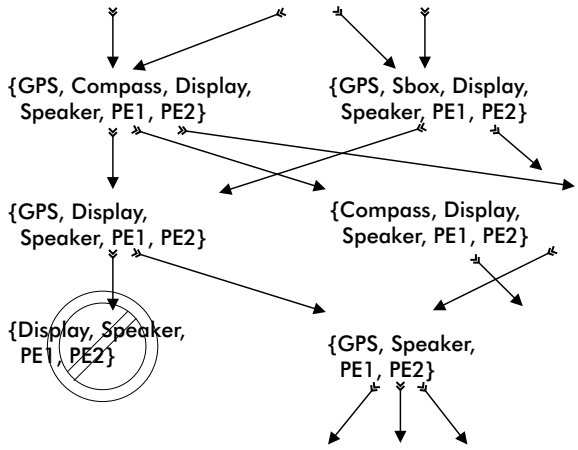

Figure 5.2: A portion of the navigation PFA lattice

5.2.2 Mapping Utility, Hardware and Software

The desire to link system utility directly to a model of graceful degradation led to the formulation of an integrated model, which maps the available hardware and software in the system to system utility. As this model Maps Utility, Hardware and Software, we call it the MUSH model. A three dimensional graph is proposed, a completely contrived example of which is shown in Figure 5.3. On the x and y axes, the combinations of software and hardware components are respectively enumerated. The order is not particularly important, though our convention is such that higher along the axis indicates a higher number of components. For the system composed of h hardware components, the y–axis has 2^h distinct allowable values. Likewise, the x–axis has 2^s possibilities for the s software components. The cartesian intersections on the x–y plane then consist of an enumeration of all possible collections of system components.

The z–axis on the graph is the system utility. Higher values are more

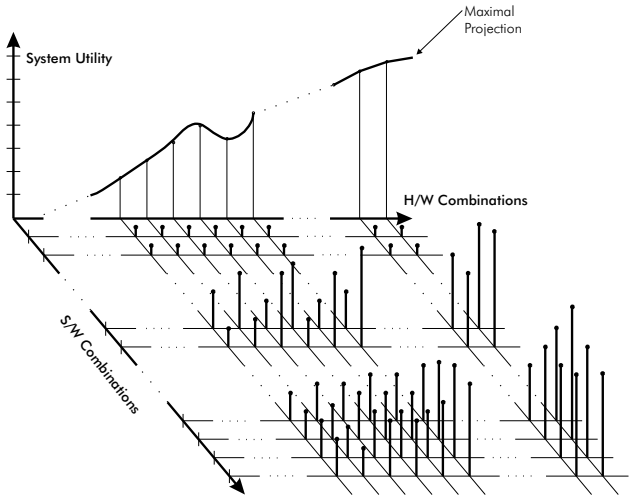

Figure 5.3: The MUSH model

desirable systems, in that they provide more value to the users. At each of the potential system configurations (*i.e.,* the $2^{(h+s)}$ points on the x–y plane), the total utility of the system is measured and plotted in the z direction. Many of the utility values will be zero, for many of the system configurations do not have sufficient components to be operational. While the overall trend is to increased system utility at higher x and y values, in general the trend will be non-linear and will not be monotonic. Recall that the system combinations on the x and y axes are ordered only on the number of components available, not the desirability of particular components. In many cases, an additional component would be redundant or not usable.

Figure 5.3 shows an additional feature of this model — the maximal utility projection graph, shown here in the x–z plane. For each value on the x–axis, the largest system utility is chosen from all of the y values and plotted. The resulting graph is the optimal solution guide for the system-wide

customization problem. In effect, it shows the best possible combination of software components that can be chosen for each possible combination of hardware components. In fact, an oracular customization manager can be postulated which would know, for each x–axis value, the y value that resulted in the largest system utility.

Note that the maximal utility projection graph has the same trend issues as the rest of the graph. While it tends to increase as the x value gets larger, it is not monotonic. Two factors are primarily responsible for this effect. First, the contribution of individual components to overall system utility is non-uniform: some components are more valuable than others. However, the hardware combinations are ordered on the x–axis based on the number of components, not their value. In effect, the x–axis is partitioned into regions with an equal number of components. Within a region, no ordering is implied. One can imagine re-sorting the x–axis numbering so that combinations within regions reflect the inclusion of more desirable components. However, some components are only desirable on a local scale — generally reflecting the effects of PE allocation. For instance, a sensor that does very little but has a large amount of available RAM may be very useful as a host for adapters when the system has few components. Yet in a system with many components, the extra RAM is not needed — and thus the sensor itself is unnecessary.

The MUSH model is not as intuitive as the lattice model for understanding the failure/repair process, as failures only guarantee a movement towards the origin. It does a much better job, however, with integrating an understanding of system utility and how it is affected by the configuration

process.

5.3 Specifying PFAs

Both of the problem models proposed in Section 5.2 are descriptive in nature and, while quite useful for communicating ideas, are quite insufficient for algorithm development. In addition, the construction of either model is obviously intractable for usefully complex systems. The following model is algorithmically sufficient for specification of a Product Family Architecture (PFA), and is the primary input to the customization manager.

The basic idea of the PFA graph is to provide alternate system components for inclusion in whatever configuration the customization manager proposes as a solution. Most distributed embedded systems will be designed without thought for redundancy. In fact, any redundancy discovered among various components will often be designed out of the system as a means to reduce costs. But, for customization to work as a mechanism for graceful degradation, there must be a source of flexibility as to which components can be used to solve the system's mission.

The required flexibility may be gained by exploiting product family architectures. Several products which accomplish similar missions will have common components (perhaps more capable components on the higher-end automotive models) and different versions of the control algorithms for the same sensors (for instance, on different model year automobiles). The PFA graph is a problem representation that can be constructed by merging the DFG for several similar models in a product family.

5.3.1 Merging DFGs to Form a PFA Graph

Section 4.2.3 discussed the idea of a *configuration*, which is any collection of sensors, adapters and actuators. These are the elements which act, respectively, as data sources, data processors and data sinks. For a particular combination of available hardware components, many different configurations are possible — each with different adapters. The customization manager must search (not necessarily exhaustively) the space, consisting of all possible configurations, for a configuration with maximal utility that can be allocated to the current hardware. The software configuration space has several interesting partitions (illustrated in Figure 5.4).

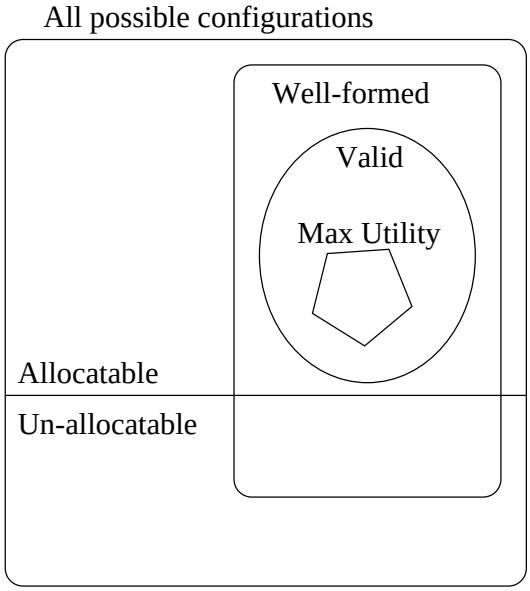

Figure 5.4: Various configuration spaces of interest

- *Allocatable* configurations are constrained to fit on the available hardware. More precisely, the software requirements cannot exceed the

resources of the hosting hardware (including network resources such as bandwidth). Configurations that cannot fit are *unallocatable*.

- A *well-formed* configuration is one in which the data flow properly transports data from sensors, through a series of properly connected adapters to actuators. Every adapter in a well-formed configuration is part of a path from some sensor to some adapter.

- A *valid* configuration is one which, in addition to being allocatable and well-formed, all system constraints are fulfilled. For instance, tasks and network messages are schedulable and meet all real-time deadlines.

- *Max utility* configurations are valid configurations whose utility value is only exceeded by other software configurations requiring different hardware. For a particular set of hardware components, the results of the system-level configuration algorithm is bounded from above by the configurations in the max utility space—no other configuration has a higher utility. Note this definition is careful to include the case where several configurations might have equal utility.

These classifications of the configuration space are somewhat arbitrary, but do reflect useful views of the space. The allocatable division deals with the available hardware and networking resources — if the un-allocatable space was too large, for instance, it could be shrunk with the addition of more hardware. Well-formed configurations are completely determined by the data flow properties of the software available. The addition of data manipulation elements (sensors, actuators, or adapters) to the PFA will en-

large the well-formed configuration space. The valid configuration subspace encompasses all the systems that can be constructed — they have sufficient hardware, software and fulfill other system constraints. And the max utility space encompasses all the desriable systems — those the customization manager should recommend.

For each vertex in a hardware-only lattice there are many possible DFGs. If sufficient hardware exists at the vertex (and the PFA has been well enough designed to provide the appropriate software adapters), at least one valid configuration also exists. If the DFGs for each such configuration were merged, the resulting graph would be an alternate representation of all the valid configurations available in the lattice. This merged graph is a *PFA graph*, a supergraph of all possible well-formed system configurations. Its construction would probably not follow the process just described. Instead, system designers would construct several DFGs for systems they are thinking about and work with them to find extensions and abstractions that are useful to building the PFA graph.

In order to merge DFGs, a notational element must be introduced to allow for choices between different components. Insertion of a *choice element* between adapter connections is easily accomplished. The choice element allows data flow from at most one of its inputs through to the output. A simple edge in the DFG can trivially be expressed as a choice element with only a single input. A useful specialization of a choice element is the *data element*, to represent message types sent via a broadcast network. Most distributed system designers would have little trouble utilizing data elements—various adapters may emit a particular data element and any

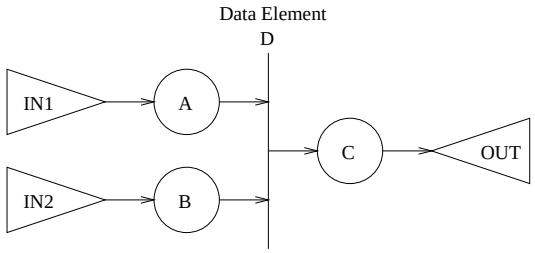

Figure 5.5: Simple example of data elements

adapter listening for the data element may read it as input, as illustrated in Figure 5.5. General networks might require more coordination between publisher and subscriber of the data, but most embedded networks do not, since they are broadcast networks (see Section 4.1.1).

The PFA graph 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. Vertices (adapters, sensors and actuators) and edges (choice elements) of the graph are labeled as described in Section 4.2 to support the different algorithmic requirements of the reconfiguration mechanism. All such labels are values the system designer would be able to generate.

Figure 5.6 shows the PFA graph (unlabelled with resource sizes for readability) for the in-vehicle navigation application. Notice how the data elements, such as Ground Speed or Turn Info, can be provided or synthesized from many different combinations of adapters.

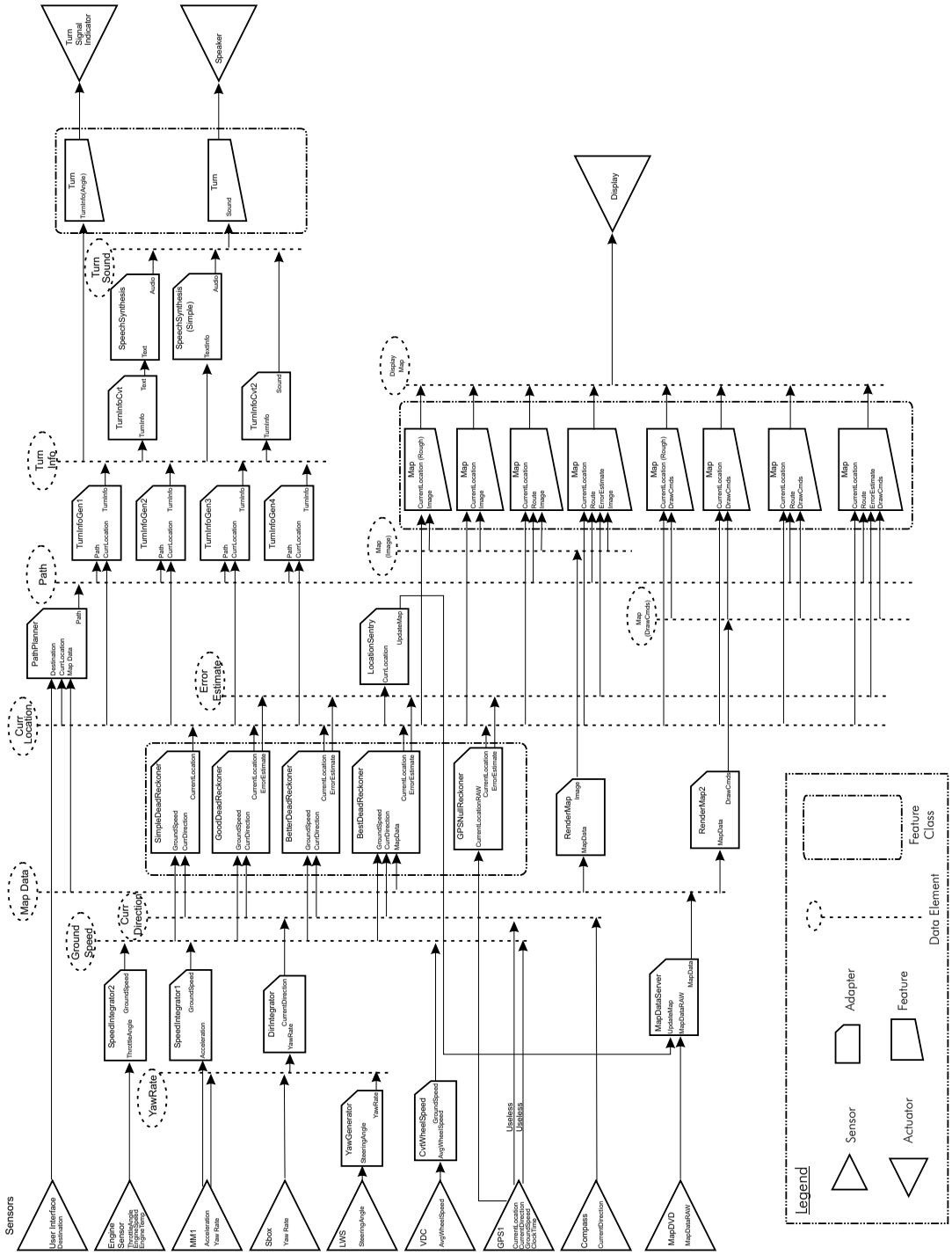

Figure 5.6: Navigation system PFA graph

5.3.2 Features

If given the PFA graph of Figure 5.6, a customization manager would not have any guidance about how to optimize functionality of the system. Which is more important, location display or path planning? Of the available location display options, which should be used? These are difficult questions, the answer to which really depends upon external requirements, not merely upon the data flow through the PFA graph. Recall that the overall goal of customization is to maximize the functionality of a given set of hardware. The algorithm can only attempt to maximize quantitative values, so functionality must be represented quantitatively.

At a high level, a feature is a means to accomplish a particular function of the system. The basic functions of a system are the requirements it has to accomplish its mission. Automobiles, for instance, have several different functions: steering, braking, speed control, passenger entertainment, etc. A difficult problem for the system architects has always been to determine exactly the scope of each function in such a decomposition. However, from a PFA perspective, each function has several possible realizations. Different automobile models have different components responsible for braking. Some are much more desirable than others — anti-lock braking systems (ABS) are in higher demand than standard brakes.

Each such different means of accomplishing the function is a *feature*. To a large degree, the customization problem revolves around choosing the appropriate features to maximize the functionality of the hardware. Features, and their associated utility values, are the means by which the system de-

signer communicates the desirability of different configuration elements to the customization manager.

On its face, this seems to be a fairly simple problem. The knee-jerk reaction of almost any engineer is to label, with utility or desirability values, different pieces or paths of the PFA graph. Such pieces being labelled are known as features, for reasons to be explained below. Unfortunately, labelling features in a consistent manner across a complex system basically requires examination of every combination of possible features to pair-wise rank them. Spotting meaningless combinations, for instance, requires the designers to think hard about each combination. Such combinatorial comparisons are only possible when the number of features is low.

Other decision methods, such as Analytic Hierarchy Process (AHP), attempt to break the combinatorial explosion, but are still widely viewed as insufficient for cases where the number of choices is large [Braglia99]. Another feature selection procedure is the Quality Function Deployment (QFD) methodology, a pre-cursor to AHP, which is insufficiently quantitative for use in this research [Cohen95].

The feature problem is very common in many system description methodologies such as aspect-oriented programming, feature-oriented programming, subject-oriented programming, and black-box composition [Kiczales97]. The basic problem is one of composibility of features, where the features need complex interaction. Some relatively new directions toward solving the problem include multi-dimensional separation of concerns [Tarr99] and generative programming [Czarnecki00], which provide syntactic support for assembly of features, but fall short on the semantic level, for example in spotting

meaningless or even faulty combinations.

This research did not attempt to solve the feature problem, but does require a low-complexity feature representation. As an attempt to lower the degree of the combinatorial explosion, we developed the *feature class* specification mechanism. It solves one of the difficult feature problems — deconflicting overlapping features. If a function can be satisfied by any of a set of features, then only one of the features should be installed on the system. In the navigation subsystem, only one software component can drive the speaker with turn information. If the system-wide customization algorithm chose the speech-synthesis turn feature as well as the tonal turn feature, both would conflict as they send differing information to the speaker. By placing both features into a single feature class, we can ensure that only one of the turn features is chosen.

Under the feature class specification mechanism, particular adapters in the PFA graph are designated as features. Each feature is an adapter that has been given a utility value to represent its desirability. Each feature belongs to one of several feature classes. All features with the same class represent redundant adapters, only one of which can be used in a configuration. The overall utility of a configuration then is the sum of the utility of all the features of a configuration. Some feature classes (not the features themselves) can be labeled as *critical*, thus imposing a constraint whereby any valid configuration must include one of the features from the critical class.

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 the navigation system, a zero sized dead reckoner feature exists to handle the GPS inputs — zero resources required, as the GPS sensor already generates the current location and error estimate data elements.

The class-based feature model is not a general model; but, it is sufficiently expressive to cover a wide variety of systems. There are, however, some useful systems that cannot be expressed and others where the expression is possible, but clumsy. An example of an inelegant expression is a feature that requires two different adapter paths. In this case, the system designer would add a zero sized (in terms of system requirements) feature and feed the two adapters to it. Mode changing [Lee01, Chou00] and dual use features are two examples of difficulties with this feature model, and await further research.

The feature class also provides a mechanism to represent optional (or *optimizing*) relations among adapters in the PFA graph. It is sometimes the case that one adapter can provide additional service if another adapter is providing additional data. As an example, examine Figure 5.7, a subset of the navigation subsystem's PFA graph. In this case, the display is capable of displaying a map and turn information (perhaps it has windowing capability). But the turn information is clearly optional. To show an optional dependency, a null adapter (one which takes no computational or communication resources) can be constructed to also provide the data element needed to display turn information. This is a partial solution, though, since a good customization manager would never have a reason to allocate system resources to host the turn adapter (the null adapter appears to do

everything necessary without cost). By placing both adapters into a feature class, and giving the turn adapter a utility higher than the null adapter's, there will be a reason to choose the turn adapter. In fact, we can mark the feature class as non-critical and then eliminate the null adapter entirely.

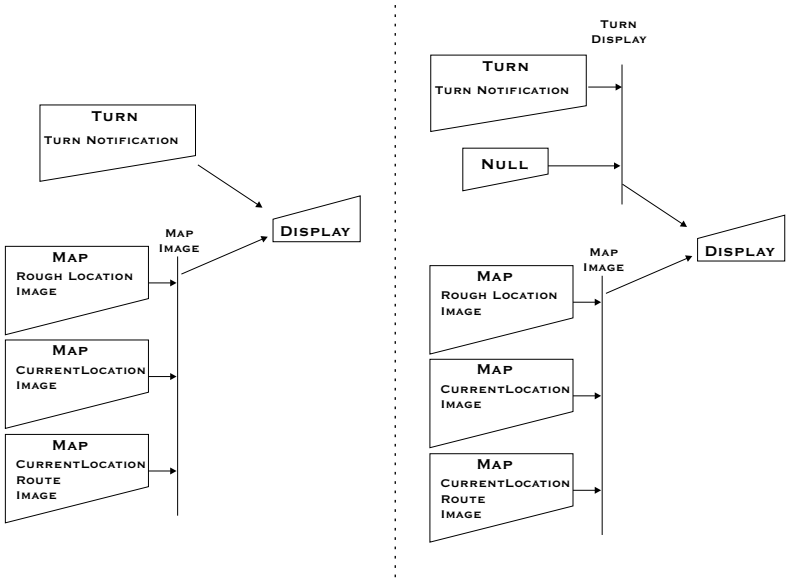

Figure 5.7: A subset of the navigation PFA graph

By organizing features into feature classes, the combinatorial explosion of feature comparisons has been somewhat limited. Determining utility values is still challenging, but now on a smaller scale. Within each feature class, ordering of utility is generally easy to assign. The pairwise comparisons are then limited to the features within each class, and a bit of class-to-class comparisons.

5.4 Summary

This chapter introduced a formulation of the system-wide customization problem that is amenable to solution. Several descriptive problem models — in either lattice or three-dimensional graph form — are useful for communicating graceful degradation ideas. The PFA graph, however, can actually be used to solve the customization problem. The PFA graph is constructed by merging the data flow graphs of several distinct products. Those products might be different models in a product line or versions from different product years. By adding a class-based feature model to the PFA graph, sufficient flexibility is gained to accomplish customization.

Chapter 6

Algorithmic Framework

Our approach to solving this problem is illustrated, at a fairly high level, in Figure 6.1. Inputs to the problem are the PFA graph, which provides all the alternatives, and a description of the available hardware. The goal is to generate a valid configuration of adapters to the processing elements (PE) and message traffic to network elements (NE). Optimally, the output configuration would be a maximum utility configuration. Because the search relies upon NP-complete components, we focus on finding fast heuristics that yield high quality solutions.

Three major searches make up the phases of the system-wide customization algorithm. Each phase is iterative and may need to be repeated upon failure of a subsequent search. The first search is to select a set of features for implementation. In order for the utility of the selected features to be functional in the system, however, the dependencies inherent in the DFG (data flowing into and out of the features) must be taken into account — the target of the second search. It is during this second phase that a set

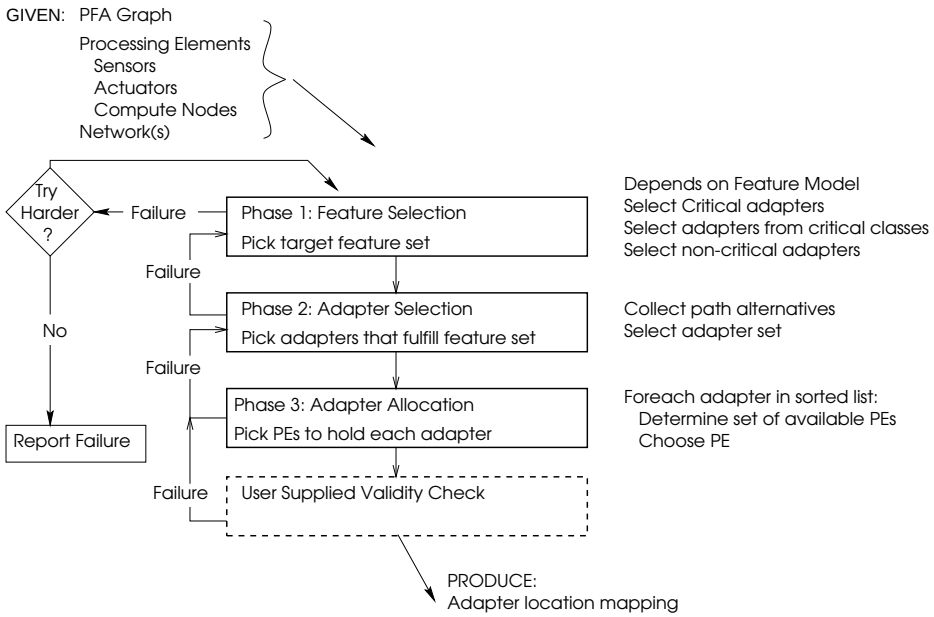

Figure 6.1: The reconfiguration algorithm

of adapters must be chosen to 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 phase, adapter allocation, has been researched in somewhat different circumstances, as discussed in Section 3. Allocation is an NP-complete problem, easily mapped to the well-known bin packing problem [Beck95]. The real challenge of solving the system customization problem is to find creative means to keep the first two phases from merely being huge loop constructs around the third, NP-complete phase.

The experiments to determine heuristic effectiveness were performed in three steps, corresponding to the three phases of the algorithmic framework. The work on Phase 3 was done first to establish an appropriate allocation

methodology. The transducer sensitive algorithm was then used for all the allocations required by the experiments for Phase 1 and 2. The Phase 1 experiments were then conducted to determine a good feature selection algorithm, which was then used as required for the experiments for Phase 2. At the end of the three experimental steps, three heuristics had been chosen for the three phases of the algorithmic framework.

6.1 Phase 1: Feature Selection

Much of the feature selection sub-algorithm depends upon the feature representation model. As discussed in Section 5.3.2, 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 systems has been used. In this model, some interior vertices of the PFA graph are *features* and are labeled with a class and a utility value. All features within the same class represent redundant adapters, only one of which can be used on the system at a time. The overall utility of a configuration is the sum of the utility of all the features of a configuration. 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.

More formally, a PFA graph includes a set of feature classes, $\{C_0, C_1, \ldots, C_m\}$ where each class C_k contains some number of features $\{F_{k,0}, F_{k,1}, \ldots, F_{k,n}\}$. For some $a \geq 0$, classes $\{C_0, \ldots, C_{a-1}\}$ are *critical*, the remainder $\{C_a, \ldots, C_m\}$ are *non-critical*. Each feature has a utility

$u(F_{k,i})$ indicating its desirability in the system. Note that the features are sorted in their classes by utility, so that $u(F_{k,i}) \geq u(F_{k,j}), \forall\, i \leq j$.

The feature selection algorithm is a *combinatorial optimization problem*[Garey79] and quite intractible. Each invocation of the algorithm will return a set of features $\left\{F_{0,i_0}, F_{1,i_1}, F_{2,i_2}, \ldots, F_{j,i_j}\right\}$ where $a \leq j \leq m$. The total utility, $U_{tot} = \sum_{i=1}^{j} U(F_i)$, is the optimization metric.

An examination of the brute force solution is instructive in developing a useful approximation heuristic. In essence, a combinatorial algorithm, such as [Trotter62] or a recursive "duplicate and add", can be used to generate a list of all possible combinations — we call this algorithm COMB_ALL. Each combination on the list will then have its U_{tot} calculated and used as the basis to sort the list. Each invocation of the algorithm would then return the next combination on the list, such that a complete search of all combinations, in order of U_{tot}, would occur. In the navigation system example, there are three feature classes: dead reckoner, turn calculation and map. Only the last would be considered critical. With 5, 2 and 8 (respectively) features in each of the three classes, the total number of combinations is $6 \times 3 \times 8 = 144$. Recall that one must be added to the non-critical classes to account for combinations that lack any member of that class. And, it so happens that the critical map class depends upon having a dead reckoner available, so any combination without a dead reckoner is automatically invalid (and thus the number of combinations is actually $5 \times 3 \times 8 = 120$).

Brute force combinational algorithms are a poor choice for implementation of the feature selection algorithm, however. While 120 combinations is perfectly acceptable for the small navigation system of our example, the

number of combinations grows exponentially. As a rough estimate, consider a system with f features in c classes. If the features are uniformly distributed among the classes, each class would hold $\frac{f}{c}$ features. Making a conservative assumption that all classes are critical, the total number of combinations would then be $(\frac{f}{c})^c$. Since excessively huge feature classes will probably not be supported by management, the number of feature classes can be approximated as $c \approx 2\sqrt{f}$. The total number of combinations, as a function of the number of features, is thus approximately $(f/2)^{2\sqrt{f}}$. In Chapter 8, a complex distributed embedded system is examined, which turns out to have 50 features in 18 classes. The resulting 2.7 billion combinations is obviously too many to handle through the brute force algorithm.

In a bid to understand the various parameters of a feature selection algorithm, we fully enumerated all adapter configurations for the navigation system. The configuration space is 2^{33} or 8.6×10^9 distinct configurations. Surprisingly, the space of well-formed configurations is quite a bit less — a mere 36,112. In terms of the integrated model of Section 5.2.2, the overwhelming majority of x-y intersection points have a zero z-axis value. Only 4.2 ppm have a positive utility. Even this small a number is quite optimistic, as it is based on the assumption is that the appropriate hardware would always be available to provide or consume data (sensors and actuators) or to execute the software (*i.e.,* appropriately sized PEs). So of the 2^{33} configurations at maximum x value on the integrated model graph, only 36,112 are of non-zero utility. At other x values, there cannot be any more than those 36,112, and in many cases there will be significantly less (if a configuration is not well-formed with all of the hardware, it cannot be so with less

hardware. However, less hardware will, in many cases, disrupt well-formed configurations that rely upon the missing hardware).

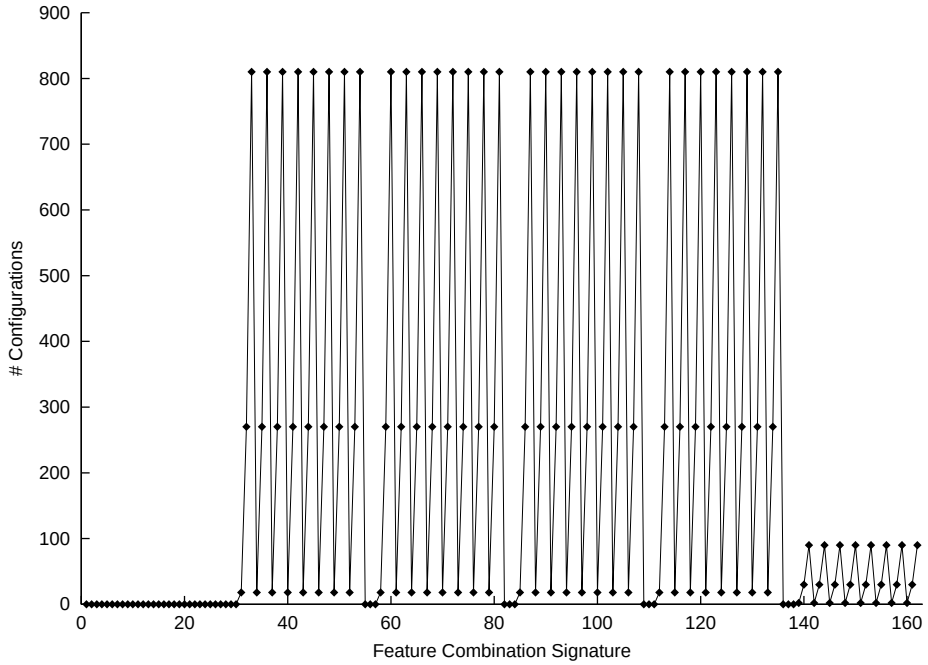

Figure 6.2: The number of configurations for each feature set

Further examination of the configuration space shows the well-formed configurations to be quite structured. Figure 6.2 shows the relationship between particular combinations of features (in enumeration order) on the x-axis and the number of well-formed configurations available on the y-axis. Visible groupings on the graph are:

- the 6 large partitions of the x-axis, each corresponding to the use or absence of a particular dead reckoner. The low x-value partition is the one which does not include a dead reckoner. Even though the

dead reckoner feature class is non-critical, the map features all rely upon a dead reckoner to provide current location, so no configurations are valid which do not include a member of the dead reckoner feature class. The configurations on the right use the GPS sensor to provide location, thus reducing the number of combinations associated with generating ground speed and direction, the inputs to all of the other dead reckoners.

- the 8 peaks for each of the 4 groupings in the middle (*i.e.,* the clusters of 8 data points with y-value of 810). These peaks each represent one of the 8 different map features.

- the 3 value levels for each of the groupings (y-value of 30, 270 or 810 in the middle groupings). Each is representative of the choices available for generating turn information, either from one of the two features or from neither.

Such structure, upon reflection, is not surprising. The configurations are constrained by the PFA graph, which is not densely connected. Further, many of the features are parallel choices, in that their inputs and outputs are substantially similar. When faced with such stark order, a brute-force combinatorial algorithm is overkill. We merely need to ensure an algorithm will cover the alternatives of each feature class.

We explore such a heuristically driven feature selection algorithm, which we name COMB_SHORT. By additively combining the feature classes, as opposed to the multiplicative combinations of COMB_ALL, the algorithm

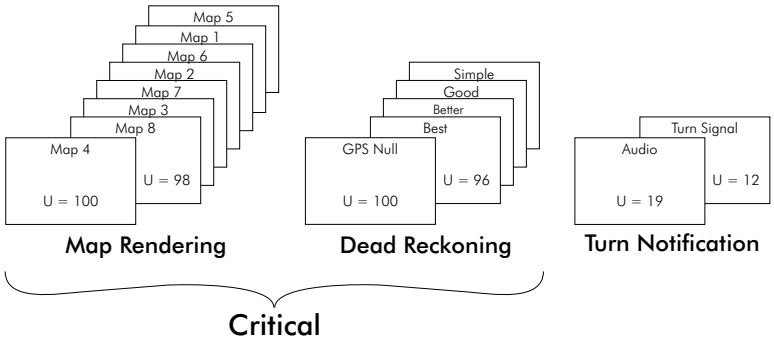

Figure 6.3: Navigation system features organized by class

is reduced in run-time complexity

$$\text{from} \qquad O\left(\prod_{i=0}^{m}(n_i+\nu_i)\right) \qquad \text{to} \qquad O\left(\sum_{i=0}^{m}(n_i+\nu_i-1)\right)$$

where m is the number of feature classes, n_i is the number of features in feature class i and ν_i is the is criticality of feature class i, with a value of 0 for critical classes and 1 for non-critical ones. Such a drastic reduction in the number of generated feature sets must be carefully weighed against the chances of missing important possibilities. Such skepticism is normal, but in this case is outweighed by the clear rationale — the structure of the feature choices is such that missing all the important possibilities is unlikely due to the redundancy in feature combinations.

COMB_SHORT is a greedy approximation algorithm, which makes a single pass through each feature class list. It works by building m lists, one for each feature class, of the features in the class, sorted such that the highest utility is at the beginning of the list. The lists for the navigation system are shown in Figure 6.3. The collection of features at the heads

of their respective lists make up the set of features passed to the Adapter Selection phase. On each invocation (save the first) it discards one of the features at the head of one of the lists. The crux of the heuristic, then, is to choose carefully which feature should be discarded.

Without further information, the feature which has the smallest utility increase over the next feature of that class (*i.e.,* choose $F_{k,0}$ for feature class C_k which $U(F_{k,0}) - U(F_{k,1}$ is smallest) should be the one to be discarded, resulting in the highest U_{tot} of the available options. As we will show later, further information is often available.

The Feature Selection algorithm must choose a feature set under three different conditions: initial, adapter selection failure and adapter allocation failure. The first time through the algorithm, since there is no information about the usefulness of any particular feature, simply choose the highest possible U_{tot}, *i.e.,* the head of each list. The adapter selection algorithm fails only because adapters cannot be chosen to fulfill all of the dependencies of a feature. For instance, a feature may require a particular sensor that is not available on the current hardware platform. Such a feature will never be achievable, so it should not appear in any further feature sets. In such a case, the feature should be discarded. Upon adapter allocation failure, the packing state achieved on each attempt could be examined to attempt to discover the core reason behind the failure. Drawing a conclusion that a particular feature is problematic, based on a series of failed allocation attempts, is a difficult, time-consuming, and error-prone process. Fortunately, as the following experiment shows, such deduction is not necessary.

6.1.1 Heuristic Evaluation for Feature Selection

The COMB_SHORT algorithm described above is concieved as a suitable compromise that avoids the high runtime and memory requirements of COMB_ALL, yet does not overly sacrifice the quality of the solution as measured by U_{tot}. The following experiments were designed to evaluate the heuristics behind the various algorithms, the result of which was used for all further research in this dissertation.

COMB_ALL is optimal with respect to U_{tot}. It ensures the first valid, allocatable configuration generated is the one with the highest U_{tot} by examining all possible feature combinations in decreasing U_{tot} order. At the other end of the spectrum is ORACLE, a speed optimal algorithm. It generates a valid, allocatable configuration in the least number of iterations. Such an algorithm can be hypothesized, but due to its oracular nature, the exact mechanics of the algorithm are unknowable.

To examine the effects of each such algorithm, all well-formed configurations of the navigation system were generated. As discussed above, the resulting 36,112 configurations were collected. This was a simple, brute force enumeration of all 2^{33} combinations. Each combination was then tested to ensure the proper input and output dependencies were met — other adapters existed to generate each input and no other adapter was included that generated the same outputs. Additional tests ensured consumers existed for each output data element. Throughout these tests, the necessary sensors and actuators were assumed to be operational in the system. This is an overly optimistic assumption and the subject of the next round of testing.

The navigation system has 12 possible sensors and actuators, thus 2^{12} or 4096 different combinations. A careful examination shows there are actually only 936 different combinations that could ever result in an operational system. The remainder includes systems without a mapping database to draw the maps from, without any sensors, without any actuators, etc. Each of the 36,112 adapter configurations was then tested against each of the 936 sensor/actuator combinations. Almost 75% of the resulting configurations were invalid — the adapters did not have a source of data, for instance. The remaining 25% were then allocated to the hardware and the results noted. Table 6.1 shows a portion of the summary data. Each line of the table represents all of the hardware and adapter combinations associated with a single feature set. There are 120 such possible feature sets for the navigation system. Note that the pass rate varies widely. The only discernable trend is a slight inverse relationship between the pass rate and U_{tot}— highly desirable system configurations generally require more adapters and more resources, and are thus harder to allocate.

An analysis of the data from this experiment shows that COMB_ALL would make 40 selections of feature sets before the cumulative probability of generating a valid, allocatable configuration exceeds 95%. The U_{tot} of the resulting configuration would be between 219 and 191, depending upon which of the 40 steps resulted in an allocatable configuration. The expected value of total utility, $E(U_{tot})$, is 192.4. $E(U_{tot})$ turns out to be very low on the range, due to the almost 5% chance of no solution in 40 steps, which has been counted as a failure. A similar analysis of the ORACLE was attempted by choosing the feature combinations with the highest chance of

Feat. Combo Signature	Total # Combinations	U_{tot}	Total Number of Combinations to				Pass Rate
			Passed	Adap Select	Net Alloc	PE Alloc	
⋮			⋮				⋮
88	17784	167	15263	2520	1	0	0.86
89	270504	179	19928	213264	0	37312	0.07
90	775944	186	0	604224	0	171720	0.00
91	17784	185	7632	10152	0	0	0.43
92	270504	197	19716	213264	0	37524	0.07
93	775944	204	0	604224	0	171720	0.00
94	17784	194	0	10152	7632	0	0.00
95	270504	206	0	213264	57240	0	0.00
96	775944	213	0	604224	57240	114480	0.00
97	2808	160	1024	1784	0	0	0.36
98	30888	172	3072	27048	0	768	0.10
99	87048	179	4224	75528	3840	3456	0.05
100	2808	181	1024	1784	0	0	0.36
101	30888	193	2560	27048	0	1280	0.08
102	87048	200	4096	75528	3584	3840	0.05
103	2808	195	0	2296	512	0	0.00
104	30888	207	2304	27048	256	1280	0.07
105	87048	214	896	75528	5888	4736	0.01
106	2808	200	512	2296	0	0	0.18
107	30888	212	2560	27048	0	1280	0.08
108	87048	219	1664	75528	2176	7680	0.02
⋮			⋮				⋮

Table 6.1: Allocation results for some feature combinations

success. Several combinations have an 85.8% success rate, so the two with the largest utility were chosen. An ORACLE that chose those two combinations would result in a 97.9% cumulative pass rate and a utility range of 172 to 177 (the U_{tot} of the two feature combinations). These two algorithms, COMB_ALL and ORACLE, provide the bounds (smallest number of iterations and resultant utility) against which heuristics can be measured.

Analysis of COMB_SHORT shows that it is a mediocre compromise that efficiently (from a memory perspective) generates a valid, allocatable configuration in fewer steps than COMB_ALL. Unfortunately, it sacrifices quite a bit of U_{tot} in the process. Its U_{tot} range of 129 to 219 is significantly worse than ORACLE. In fact, its $E(U_{tot})$ of 162 is even worse than ORACLE's worst case.

Before discarding the basic idea of COMB_SHORT, we examined the performance of an oracular version that performed the same greedy mechanics. This version, called SHORT_ORACLE, showed some promise to the idea. If the decision of the proper feature to discard was made to maximize the pass rate, the resulting algorithm could produce configurations with U_{tot} between 177 and 219 ($E(U_{tot})$ is 187.1). Such good results prompted us to examine COMB_SHORT further. It turns out that a large number (almost 77%) of the allocation attempts requested by the Feature Selector were unattainable, due to a lack of hardware. In such cases, the algorithm should reject the features that don't have sufficient hardware. By using such feedback from the Adapter Selection phase, we improved the algorithm in the Feature Selection phase.

In order to quantify the improvement, we examined a variant algorithm

	Number of Iterations	Cumulative Pass Rate	Utility		
			Low	Expected	High
ORACLE	2	97.9%	172	172.8	177
COMB_ALL	40	95%	191	192.4	219
COMB_SHORT	13	97.5%	129	162.0	219
SHORT_ORACLE	8	97.2%	177	187.1	219
SHORT_FEEDBACK	5.8	96.9%	177	183.8	219

Table 6.2: Algorithm evaluation results

of COMB_SHORT called SHORT_FEEDBACK. SHORT_FEEDBACK discards features from the feature class lists whenever insufficient sensors or actuators exist to support the feature. The algorithm was executed for each of the 936 possible hardware combinations, with good results. Configurations with U_{tot} between 177 and 219 (identical to SHORT_ORACLE) were achieved with a 96.9% cumulative pass rate. The $E(U_{tot})$ of 183.8 is quite close to SHORT_ORACLE's as well.

An additional optimization speeds SHORT_FEEDBACK considerably. If multiple features are designated as unattainable in the feedback, the algorithm speeds through the unusable feature sets. It is with this optimization in place that SHORT_FEEDBACK was able to average a mere 5.8 iterations, as shown in Table 6.2.

By incorporating the feedback from the graph analysis of phase 2, the SHORT_FEEDBACK algorithm is able to capture 95.5% of the U_{tot} of the COMB_ALL algorithm, with only 14.5% of the iterations. Its results are remarkably close to the oracular version, SHORT_ORACLE. We used SHORT_FEEDBACK for all further experiments.

The results of all five experiments are summarized in Table 6.2.

6.2 Phase 2: Adapter Selection

The selection of adapters to implement the feature set forms the core of the system-wide customization algorithm. It is during this phase that the relationships and dependencies between adapters, as expressed in the PFA graph, are incorporated into a solution. As such, the algorithm employs graph manipulation techniques to find various sets of adapters (and the links that join them) for allocation.

With a bit more formalism, the adapter selection problem is:

Given: a PFA graph P, set of features F and the state of all sensors/actuators (i.e. working, not working)

Find: an allocation graph A, which is a subgraph of P, that describes a minimalist valid configuration. Ideally, find A such that the probability it will be allocatable is maximized.

Both graphs, A and P are each $(V, \mathcal{E})$, where V is a set of adapters, sensor, and actuators. $\mathcal{E}$ is the set of directed communication elements joining them. $F \subset V$, because all features of the feature set are in V. In fact, no other features are in V — $(V - F) \cap \mathcal{F} = \emptyset$, where the set $\mathcal{F}$ is a subset of P, containing all the features of the PFA graph. A is *minimalist* in the sense that the removal of any vertex in V or edge in E would make A no longer a valid configuration.

All of the graph traversal and manipulation takes place in Phase 2. Each feature f in F can be implemented in various manners, each of which we call a *path* through the PFA graph.[1] Each path must connect one or more

[1]Graph theorists use the term to denote a walk with no repeated edges or vertices. We relax this condition to permit loops and multiple branches. In essence, our term *path* is a

operational sensors, through intervening adapters, as well as f, to one or more operational actuators.[2]. Every adapter included in the path must also have all of its input communication elements and at least one output communication element included as well. Any communication element included must have one and only one input adapter and at least one output adapter included.

Each feature f has its own set of paths, any one of which will satisfy the requirements to use the feature. The allocation graph A is the union of one path for each of the features in F. The resulting allocation graph will be passed to the adapter allocation phase to see if it can be fit into the hardware resources available.

In order to determine a path for a feature f, two graph traversals are necessary — one of the feature's inputs, back to the sensors and the other from the outputs forward to actuators. The traversals are fairly straightforward, with only a few places for concern. If an adapter is to be included in a path, all of the communication elements at its inputs and at least one of its outputs must also be added. For a communication element to be added, one and only one input and at least one of its outputs must be included. If the traversal examines a sensor or actuator in the PFA graph, it must be operational in the system, of course. If the traversal comes across a feature, it must be an element of F, or else the traversal will need to backtrack, because only features included in F are to be part of A. To do otherwise is to

union of walks.

[2]Paths that do not employ a sensor (or actuator) are possible in real systems. One example is a diagnostic database that merely acts as storage for system exceptions. We chose to simplify the problem somewhat by focusing on the much more frequent paths that use both a sensor and an actuator

disregard the separation of responsibilities assigned to the different phases. If the feature is to be included, it will be (or has been) specified by phase 1.

Such graph traversals are sufficient to generate a path for each of the features in F. A is then the union of each such path.

Obviously, the success rate of an algorithm to generate A will depend on how the various choices are made (where success is defined as the generation of an allocation graph that is allocatable). Since the only variance in the graph traversals occur when choosing one of the inputs to a communication element, that decision will be the focus of our heuristic search.

But there is an even more fundamental issue that must be dealt with — devising a mechanism to keep track of which choices were made so that alternate paths can be attempted in the case of an allocation failure. When the adapter allocation fails, then the adapter selection algorithm will need to choose a different allocation graph for the next attempt. The graph traversals that generate the paths make a decision at each communication element as to which input will be chosen. The state of those decisions defines the path generated. We need a mechanism to ensure the decision state varies in a way that sufficiently covers the different combinations of path elements. If, for instance, the algorithm merely chose the next input for communication elements on a re-traversal, then all of the choices throughout the graph would vary. Such variance would likely miss many interesting combinations that should be more fully explored.

As a solution, the algorithm we employed uses a single depth-first recursive traversal, but instead of collecting a single feature path, it generates all the different combinations — in essence flattening the graph to a sin-

gle choice. This is not exactly a lean algorithm that would be used in an actual embedded system. But it is extremely useful to analyze the combinations and the means to make choices about them. In building a real world system, algorithm designers may find a better solution in the field of algorithmic combinatorics [Even73].

6.2.1 Heuristic Development for Adapter Selection

Heuristic development began with examination of local choice policies that can be applied at each communication element to decide upon which input to use. Such choices are inherently limiting in that they examine only the state of the PFA graph in the neighborhood of the communication element. But the limitation is often necessary or useful, because the choices also tend to be ones that can be made rapidly and thus do not degrade the performance of the algorithm.

The measure of merit for a particular adapter selection heuristic is the success rate of allocating the adapters in phase 3. Since the allocation varies based on the particular hardware available, we carefully explore the results over a wide variety of hardware possibilities. In the following discussion of the Navigation System, we examined all 936 combinations of hardware for which allocation is possible.

The experimental protocol was thus established: for each of the 936 hardware configurations, the feature selection algorithm of phase 1 was executed. Each feature set produced was then run through the adapter selection graph traversal to produce all possible allocation graphs (corresponding to well-formed configurations) for this particular hardware configuration. Each of

the various heuristics favor certain of these allocation graphs above others, in essence providing an ordering, from most favorable (for the particular heuristic) to least favorable. A successful heuristic favors allocation graphs that lead to successful allocation (in phase 3). To measure the success of each heuristic, all of the allocation graphs were then passed to phase 3, with the results noted. The heuristics were then compared to determine the corrolation between the heuristic's ordering and the successful allocation results. The measure of merit for this comparison was the number of attempts necessary before phase 3 successfully allocated the results. Note that since the heuristics merely reorder the allocation of the possible allocation graphs, the resultant utility is the same for all heuristics. This restriction is lifted in the Chapter 8 in order to handle the complexity of the proof-of-concept system, as the number of combinations gets too large.

The first heuristic choice examined was simply to choose the allocation graph with the smallest sum of adapter sizes. The reasoning behind this heuristic is that such adapters would be easier to allocate, and thus result in better packing rates. Two different means of measuring adapter size were employed, scaling them with regard to the total amount of PE resources available or to the total required resources needed by the adapters in the PFA graph. Along the same lines, another heuristic was examined that simply measured the total number of adapters in an allocation graph. This heuristic doesn't have the same intuitive sense backing it up, because a small number of adapters may include all the large elements of the PFA graph. Similarly, a large number of small adapters can probably be packed (allocated) more efficiently.

Further heuristics examined the communication elements, both in number and required bandwidth, of the allocation graph. Comparison data was also collected for random choice and PFA graph traversal order. Ascending and descending sorts were done for many of the heuristics as a sensitivity analysis, to ensure the scores generated were real discriminators. A summary of the heuristics, as well as the naming convention to refer to them, is in Table 6.3.

Heuristic Name	Description	Variants
Random	Random choice	
InOrder-Up	As generated by graph traversal	Ascending Sort
InOrder-Dn	As generated by graph traversal	Descending Sort
SizeAdap-Res	Sum of the adapter sizes	Scaled by PE resources
SizeAdap-Rqt		Scaled by Adapter requirements
NumAdap-Up	Simple count of the number of adapters	Ascending Sort
NumAdap-Dn		Descending
CommNum	Number of communication elements	
CommBW	Bandwidth required by all comm elements	

Table 6.3: Summary of phase 2 heuristics

The results of our experiment are displayed in Table 6.4. These selection and allocation attempts are only the cases where the particular feature set results in allocation graphs that in some cases can pack onto the hardware and in other cases will not. We are not interested in the cases where all allocation graphs can be allocated. In those cases, the use of any of these heuristics is equally good as all the others. Likewise, cases where no allocation graph can be allocated only illustrate that any heuristic is equally bad. In order that the discriminatory cases are more clearly highlighted, we

remove all other cases from the results.

A few comments help to understand the data presentation. Recall that a particular heuristic will examine and essentially score each allocation graph. In many cases, diverse graphs will result in the same score. For instance, NumAdap merely counts the number of adapters in the graph, so many different allocation graphs result in the same score. It may happen that one (or more) of the allocation graphs with a particular score turns out to successfully be allocated in phase 3. With no means to differentiate among the allocation graphs with the same score, the heuristic may sometimes discover the allocatable configuration earlier than or later than the other graphs within the score category. This range is labelled "Best Case" to "Worst Case" in Table 6.4. Many heuristics uniquely score each allocation graph and thus only have a "Best Case." For instance, SizeAdap measures the size of the adapters, and thus would only rarely (for our PFA graph) find two allocation graphs that consist of different combinations of adapters yet result in identical size measures. Systems with very regular component sizes would have different results, of course.

Table 6.4 also summarizes the results of each heuristic against all 936 hardware configurations. The minimum, average and maximum values found are shown for each heuristic. The last two rows show the minimum, average and maximum allocation graphs that pass for any of the hardware configurations, as well as the statistics for the totals found, regardless of whether they could be allocated. So, there are hardware configurations with 30 to 810 different possible allocation graphs, of which 13 to 61 can be allocated. On average, about 10% of the allocation graphs could be allocated.

Heuristic Name	Best Case			Worst Case		
	Min	Avg	Max	Min	Avg	Max
NumAdap-Up	1	1	1	1	6.99	16
NumAdap-Dn	7	107.89	283	11	194.17	499
SizeAdap-Res-Up	1	1.60	3			
SizeAdap-Res-Dn	10	255.08	655			
SizeAdap-Rqt-Up	1	1.60	3			
SizeAdap-Rqt-Dn	11	254.36	652			
CommBW-Up	1	1	1	11	53.61	220
CommBW-Dn	1	49.21	211	11	89.48	351
InOrder-Up	1	1	1			
InOrder-Dn	4	20.83	36			
Random	1	9.17	100			
Random	1	8.46	83			
Random	1	8.05	65			
Random	1	8.11	66			
Random	1	7.68	74			
Random	1	7.82	78			
Random	1	8.30	80			
Random	1	8.11	99			
	13	31.79	61	Pass		
	30	323.68	810	Total		

Table 6.4: Number of iterations required for different heuristics

The random heuristics give us a good sense of what can be accomplished easily. Certainly any heuristic should be able to occasionally generate an allocation graph that packs on the first try and on average takes only 8 or 9 tries.

The performance of the InOrder-Up heuristic is surprisingly good. It was included as a debugging measure and to help establish comparison bounds. But it shows an unerring ability to choose an allocatable graph on the first try. An examination of this surprising result shows us that in fact, the first allocation graph generated by the traversal happens to pack properly on every hardware configuration. There is no guarantee of such a result on any other PFA graph or PE configuration. Note that when reverse sorted, InOrder-Dn shows much worse results.

The four remaining choices of heuristic (NumAdap, SizeAdap-Res, SizeAdap-Rqt and CommBW) are all reasonably good at choosing successful allocation graphs. They all are good discriminators when compared with the random and their own reverse sorting versions. In addition, they all score allocation graphs such that one of the set with the lowest score (or highest — whichever was attempted first) is allocatable for all hardware configurations. NumAdap and CommBW both result in good best case situations. But there are so many allocation graphs with identical bandwidth requirements (in many cases, 220 such graphs) that on average bandwidth is a poor choice for a heuristic. NumAdap does better, with no more than 16 attempts required on any hardware configuration. For the heuristics that size the adapters (by requirements of the PE or by available resources doesn't make much difference), there are hardware configurations where that scoring

would take three allocation attempts to find a winner. With an average of 1.6 attempts, these are superior and will be used for the rest of our research.

When we began this research, we imagined that feedback from phase 3 would be critical to ensure phase 2 doesn't overly exercise the adapter allocation of phase 3. However, 1.6 attempts is low enough that we see no reason at this point to put much effort into examining failure results in trying to improve the heuristic. The proof of concept system in Chapter 8 provides an interesting viewpoint on this decision.

6.3 Phase 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 similar contexts. Most such research thrusts approach the problem as a bin-packing problem. Bin packing is NP-complete, but heuristic methods based on non-guided search and list processing exist.

We use list processing heuristic mechanism for adapter allocation. This is a greedy algorithm that sorts the adapters and then runs down the list, placing adapters into one of the processing elements. If the list can be exhausted, then the allocation was a success. Otherwise, it is a failure that requires stepping back to Phase 2 for a choice of different adapters.

The specific adapter allocation algorithm we use is a transducer sensitive algorithm, more detail of which is available in Chapter 7

6.4 Summary

This chapter has been an examination of the experimental methods capable of discovering good heuristics to populate the three phase customization framework. A greedy feature selection heuristic was shown to be sufficient, when paired with a feedback mechanism, to capture 95.5% of the utility, with only 14.5% of the iterations of the combinatinal algorithm. Adapter selection for the second phase via a simple adapter count proved to be a suitable heuristic. A discussion of adapter allocation heuristics is forthcoming in Chapter 7.

Chapter 7

A New Adapter Allocation Algorithm

7.1 Introduction

The adapter allocation algorithm is responsible for determining the placement of software components and their corresponding communication elements to the hardware, processing elements and the network, resident in the system. The allocation algorithm to be used is a bit different from previously explored algorithms. In this case, the hardware is fixed and heterogeneous. Most other allocation algorithms come from the field of hardware/software co-design, where the hardware specification is part of the output of the problem, and is thus not fixed. The optimization sought in this case is usually a cost measure — silicon area, for instance. Parallel processing task algorithms also determine the mapping between software adapters and the homogeneous hardware processors that will execute the tasks. The goal of

such algorithms is usually to minimize the schedule length of execution of the tasks. In contrast, distributed embedded systems are most frequently composed of many different microcontroller types, each with different amounts of compute resources, and very limited network bandwidth. A survey of the related allocation research was covered in Chapter 3.

In addition to the minor differences caused by such fixed, heterogeneous hardware, the allocation desired by RoSES is constrained by an additional factor, unlike classical co-design or parallel processing realms. The software components of a distributed embedded system are managing and interacting with the sensors and actuators of the system. Those hardware components are not general — the fuel-air sensor in the automobile is located in a particular place, hooked to a particular microcontroller. It does no good for the allocation algorithm to think of moving the fuel-air sensor's driver software to any other microcontroller — it must be co-located with the sensor. Likewise, any software that interfaces directly to any hardware component is fixed — it cannot be allocated elsewhere.

The algorithm described in this chapter exploits the fixed nature of hardware interface software components by examining the other adapters that it might call or be called by. These neighboring adapters can often be allocated locally and thus save any network communication. The process continues, allocating neighboring adapters in an attempt to minimize network usage. The remainder of the chapter examines the details of how to choose the adapters to allocate, how many neighbors to examine and what to do when the processing elements with the transducers are filled.

7.2 System Model

The software to be allocated is a collection of mobile components called *adapters*. The adapters are joined in an *allocation graph*, $A(V, E)$ whose vertices are the adapters and edges $E_{i,j}$ represent communication between adapter$_i$ and adapter$_j$. Edges may be directed, though doing so has no effect on this algorithm. Each adapter$_{i,j}$ is labeled with its processing requirements, *p(i)*. Processing requirements are often a list of multiple independent values such as CPU cycles, RAM, or I/O channels. Similarly, edges are labeled with communication requirements *c(i)*, usually representing bandwidth. Figure 7.1 shows a sample allocation graph. This particular graph, from [Efe82], is often referenced in the allocation and task scheduling research field. *p(i)* and *c(i)* are shown in parenthesis.

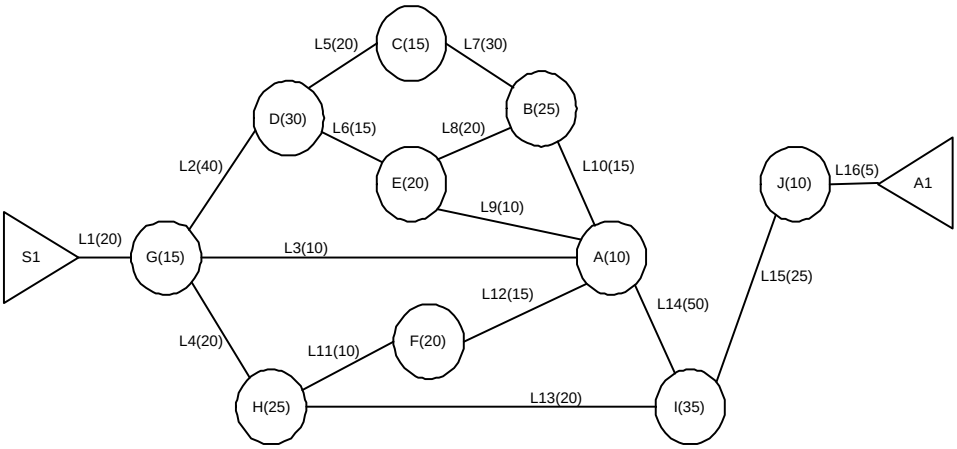

Figure 7.1: A sample allocation graph

Originating vertices (those with no inputs) represent *sensor* components that are the source of data. Likewise, those vertices with no outputs are

actuators which act as a sink of data. We use the term *transducer* to describe both sensors and actuators. The allocation graph in Figure 7.1 has a single sensor (S1) and actuator (A1).

The components of the allocation graph must be mapped to the available hardware. The hardware is a collection of *processing elements* (PEs) connected to a single network. Each PE has a fixed resource list, in the same size and types as the requirements of the adapters. Likewise, the network resources match the requirements of the communication edges. Additionally, transducer adapters are pre-assigned to PEs, as their physical hardware is not general to all PEs. This system model uses a single network, though an extension to multiple network links is possible. The general approach for such an extension would follow the techniques described in [McNally98].

The major opportunity for optimization occurs by allocating incident adapters to the same microcontroller. In such a case, the communication requirement *c(j)* is fulfilled through local inter-adapter communication, and thus does not impact the network at all. In contrast, if the adapters are allocated to different microcontrollers, *c(j)* must be borne by the network.

Because we are not overly concerned with schedule, but rather resource usage, we make the assumption that the program will cycle continually. This assumption is quite reasonable for embedded systems where new samples are periodically available at the sensors and serviced in a time-triggered manner.

7.3 The TRANS_FIRST Algorithm

The key insight behind the TRANS_FIRST algorithm is to exploit knowledge about transducer location. This bit of knowledge is not available when allocation algorithms are used for general purpose computing systems, as there is little reason to require a particular adapter to execute on a particular PE. But in a distributed embedded system, the transducer adapters are managing special purpose hardware only available at a particular PE, so they must be allocated to those PEs. By working inwards from the exterior vertices of the allocation graph, large sections, or subgraphs, of the graph may be allocated so as to eliminate network communication among adapters in the subgraph.

The basic algorithm is:

```
BEGIN {TRANS_FIRST}
  Chose a PE: pe                                (7.4.2)
  REPEAT
    Initialize set T with all adapters which:
      Are incident to an adapter allocated on pe
      Can fit on pe                             (7.4.4)
      Haven't been allocated yet

    REPEAT
      Select adapter: k from T                  (7.4.1)
      Remove k from T
      IF k can fit on pe THEN
        Allocate k to pe
        Add to T all adapters which:
          Are incident to k
          Can fit on pe                         (7.4.4)
          Are unallocated
          Haven't been rejected before
```

```
      ELSE
        Remember that k has been rejected
    UNTIL (T is empty)
  UNTIL (All PEs considered)
  Allocate remaining adapters
END {TRANS_FIRST}
```

This algorithm is a variant of the list-processing heuristics for solving bin packing problems. Its usefulness and performance will depend upon the particular policies for making decisions within the algorithm. We examine the alternate policy choices available at the numbered lines in the corresponding portion of the following section.

7.4 Policy Choices

The TRANS_FIRST algorithm is affected by four basic policy choices, three of which were illustrated in the algorithm pseudo-code of Section 7.3. The fourth (Adapter at a Time) requires a slight re-organization of the algorithm and will be described fully in Section 7.4.3. All of the choices are listed in Table 7.1, along with an identifying character for easy reference in the results tables.

7.4.1 Choosing an Adapter for Allocation

This is the policy choice with the most latitude. Adapters can be chosen from the set of candidates based on the characteristics of the particular adapter, or of their neighborhood of the alloation graph. We chose to examine six alternatives. The use of adapter size follows from the well-known bin packing rule-of-thumb: "pack the largest item first." We contrast this approach by

Table 7.1: 4 axes of policy choices

Adapter Choice	**PE Choice**	**Adapter at a Time**	**PE Fill Level**
Largest (L)	Id Order (1)	Yes (Y)	Full (F)
Smallest (S)	Reverse Id (2)	No (N)	80% (8)
Max B/W (B)	Largest (L)		50% (5)
Min Neighbors (N)	Smallest (S)		
Max Neighbors (M)	Random (R)		
Random (R)			

also attempting to pack the smallest first. Often, the network bandwidth is a scarce resource worth conserving. To that end, we choose adapters based on the bandwidth savings offered by a local allocation. Recall that communicating co-located adapters need no network resources.

In an attempt to choose adapters based on their neighborhood in the allocation graph, we examine the results of a choice based on the numbers of neighbors. By choosing adapters with the most neighbors, we provide a bigger pool for choices in successive iterations. The danger, however, is that the subgraph will consist of several tendrils that block off and interfere with the growth of other subgraphs, without making the kind of bundles that can pay off from a network perspective. We also consider a policy which chooses the adapter with the lower number of neighbors. Such a policy should consume all the adapters in a region of the graph before expanding.

For comparison purposes, we also randomly choose adapters from a uniform distribution. The random choice uses absolutely no knowledge of the graph or adapter charactersitics, so provides a baseline which the algorithm must be able to outperform in order to be of any use whatsoever.

7.4.2 Choosing a PE

Clearly if the subgraphs allocated to the various PEs are not adjacent, then the order in which PEs are chosen for allocation will have no effect on the solution. Conflict occurs only when some adapter has the potential for allocation to multiple PEs. One would think that on large allocation graphs such potential would be rare. Such intuition is incorrect for the distributed embedded allocation graphs, as the transducers cluster in particular neighborhoods of the graph.

To explore the degree to which PE choice policy affects the solution, we implement 5 different alternatives. The first uses an arbitrary, though fixed, ordering — by identification number. We also examine the reverse ordering. The only characteristic of the PE which may have some bearing on the potential solution is the amount of resources the PE has available. The PE with a large resource stash should be able to carve out a larger subgraph if unimpeded by other PEs. Such a policy choice is not clearly a winner, as such a large subgraph may block off the later PEs from access to any portion of the graph. For this reason we study PE ordering by resource in both ascending and descending order.

We also include random choice as a baseline, for the same reasons as in Section 7.4.1.

7.4.3 Adapter at a Time Allocation

The TRANS_FIRST algorithm, as described in Section 7.3, allocates all the adapters that fit on a PE before allocating any to other PEs. In an allocation

graph where many sensors are coupled via a few adapters, such a strategy will generate a large subgraph for the first PE chosen. But the subgraph will block off development of subgraphs on closely associated PEs. By re-organizing the main loop of the algorithm, a "breadth-first" strategy can be attempted, where a single adapter is allocated to a PE at any particular time. The re-organized algorithm must maintain the state of each PE's search in independent T sets. After an adapter is allocated to one PE, the algorithm queries another PE, in the same order specified for Section 7.4.2.

7.4.4 PE Fill Level

By allowing PE resources to be fully consumed by this pre-process step, allocation of other large adapters may be inhibited. It is possible that restricting adapter allocation during the first phase of TRANS_FIRST may conserve space for the adapters at the middle of the graph that are far from transducers. We explore allocating adapters only to fill 80% (or 50%, or some other arbitrary cap) of PE resource levels. The inevitable tradeoff is that packing to 80% on all the PEs may then leave us vulnerable to a 21% (or 51%) sized adapter. This tradeoff is merely another case of the typical best-fit versus worst-fit bin packing policy choice.

7.4.5 Allocation of Remaining Adapters

Once all PEs which host transducer adapters have been filled using the TRANS_FIRST algorithm, remaining vertices of the allocation graph may remain. Use of another bin packing algorithm will allocate them to the remaining PEs.

Graph	Adapters	Edges	Sensors	Actuators	PEs
ranA	80	200	8	10	9
ranB	80	200	8	10	5
ranC	42	110	1	6	6
ranD	43	100	13	13	9
ranE	17	50	22	21	11
ranF	20	50	4	1	10
iac	14	98	3	64	9
tract	43	282	2	115	16
sch	34	316	3	117	16

Table 7.2: Allocation graph characteristics

Typical distributed embedded systems will not have many (or any) PEs remaining at this point. Rather, all PEs are connected to, or encompass on the same silicon, the system's sensors and actuators. This "smart sensor" strategy leaves few PEs remaining as merely compute nodes. In the case of a transducer hardware failure, however, the microcontroller is still capable of operation and would be allocated adapters from the central portion of the allocation graph.

7.5 Results

7.5.1 Test Set

All experimentation was done using a collection of 9 graphs from [Beck95]. Six are randomly generated graphs and 3 are from real-world automotive applications. The salient features of each graph are shown in Table 7.2. PE and network resource sizes were developed via execution of the system specification generation algorithm from [Beck95].

7.5.2 Experimental Method

We first present four experiments to determine the appropriate policy generation choices. An additional experiment compares the TRANS_FIRST algorithm to system allocations done without sensitivity to transducer location. Both algorithms were implemented using as much of the same code as possible (to screen out implementation differences) and all executions done on the same computer (A 750MHz Pentium 3, in an IBM Thinkpad T20). Each experiment involved 10 runs of each algorithm choice. Averaged values over the 10 runs are reported.

Three values of interest were measured and calculated for each of the graphs: success rate, network usage and algorithm running time. Success rate is a measure of the number of times the algorithm found an allocation over repeated execution of the algorithm. Unless explicitly using a random policy choice, the TRANS_FIRST algorithm is completely deterministic – though the follow on phase, as described in section 7.4.5, is not. Success rate, therefore, is often 100% or 0%, regardless of the number of executions.

Network usage is a figure of merit which measures the extent to which an allocation placed edges of the allocation graph in locally. It is calculated as the ratio of the bandwidth of the allocation graph edges with incident adapters placed on the same PE to the total bandwidth of all allocation graph edges. A value of zero indicates that all communication is on the network, while a value of one is the (highly unlikely) case where all communication is local to a PE.

7.5.3 Finding the Right Policy Choices

Adapter Choice

Table 7.3 shows the results of 6 experiments, each differing only in the adapter selection policy. Choosing the Adapter with the largest requirement is the best of the policy choices. Note that no other policy choice had a higher success rate on any of the graphs. It is less impressive with respect to network usage — a situation that is not surprising given that such a policy choice pays no attention to the network at all. All of the graph sensitive choices (Min Neighbor, Max Neighbor and Bandwidth Savings) have good network usage statistics. Those choices do not react to the resource requirements for actual packing on the PE, so they do not actually allocate with good success rates. We use the Largest Adapter policy choice for all other experiments.

PE Choice

Table 7.3 also shows the results of the 5 experiments on PE choice. Using an arbitrary ordering (By ID and By Reverse ID) shows no difference in packing success, but does use the network to a strikingly different degree most notably in the real-world allocation graphs. Both are still outperformed by a Random choice and both resource level choices. Selecting the Largest Resource policy is a narrow winner over Random with respect to packing success. However, Largest Resource allocates many more communication edges to local PE communication. Our hypothesis in this regard is correct: choice of a large PE allows for large subgraphs to be allocated

Experiment 1: Adapter Choice

	Network Usage									Success Rate								
Policy	ranA	ranB	ranC	ranD	ranE	ranF	sch	tract	iac	ranA	ranB	ranC	ranD	ranE	ranF	sch	tract	iac
L	.35	.53	.35	.45	.70	0	.17	.21	.23	.8	.8	.9	1.0	1.0	0	.8	.8	1.0
S	0	0	.42	0	.70	0	.17	.24	.38	0	0	.2	0	1.0	0	.1	.6	.9
B	.44	0	.56	0	.68	0	.22	.27	.29	.2	0	.6	0	1.0	0	.7	.7	.9
N	0	.48	0	.46	.70	0	.16	.29	.33	0	.4	0	.9	1.0	0	.8	.4	1.0
M	.32	.44	.41	.44	.70	0	.18	.29	.31	.3	.4	1.0	.1	1.0	0	.7	.7	1.0
R	.40	.45	.38	.46	.69	0	.18	.24	.29	.2	.3	.8	.3	1.0	0	.6	.6	1.0
Experiment 2: PE Choice																		
2	.37	0	0	.48	.68	0	.16	.20	.27	1.0	0	0	1.0	1.0	0	1.0	1.0	1.0
1	.33	0	0	.48	.72	0	.41	.49	.71	1.0	0	0	1.0	1.0	0	1.0	1.0	1.0
L	.32	.59	.35	.48	.73	0	.40	.26	.35	1.0	.7	1.0	1.0	1.0	0	1.0	1.0	1.0
S	0	.45	.43	.44	.68	0	.21	.26	.22	0	1.0	1.0	1.0	1.0	0	.5	1.0	1.0
R	.35	.53	.35	.45	.70	0	.17	.21	.23	.8	.8	.9	1.0	1.0	0	.8	.8	1.0
Experiment 3: Adapter at a Time																		
Y	0	0	0	.47	.67	0	0	.24	.20	0	0	0	1.0	1.0	0	0	1.0	1.0
N	.32	.59	.35	.48	.73	0	.40	.26	.35	1.0	.7	1.0	1.0	1.0	0	1.0	1.0	1.0
Experiment 4: PE Fill Level																		
F	.32	.59	.35	.48	.73	0	.40	.26	.35	1.0	.7	1.0	1.0	1.0	0	1.0	1.0	1.0
8	0	.59	.29	.42	.68	0	.33	0	0	0	1.0	1.0	1.0	1.0	0	1.0	0	0
5	.29	.59	.41	.41	.70	.20	0	0	0	1.0	.8	1.0	1.0	1.0	.1	0	0	0

Table 7.3: Results of each policy

locally, without restricting further development from the smaller PEs. We use the Largest PE policy for the following experiments.

Adapter at a Time Allocation

In Section 7.4.3, we advanced the supposition that it may be better to allocate a single adapter from the PE before trying a different PE. The experiment documented in Table 7.3 shows this not to be the case. The Adapter at a Time policy resulted in successful allocations on only 4 of the 9 graphs. It turns out that this allocation policy fills up each PE somewhat equally, without a reserve in case a large adapter is encountered. We use the PE at a Time allocation policy.

PE Fill Level

One remarkable aspect of the results shown in Table 7.3 is the complete lack of success for any policy on the ranF graph. In this particular case, a large adapter is near the center of the graph, and thus out of reach of each of the PEs until they have already allocated some adapters. Unfortunately, by filling up with smaller adapters, the PEs have no remaining resources for the large adapter. In our final experiment, we attempt to limit the adapter allocation during the transducer sensitive phase of the algorithm in order to save some space for such large adapters. Our experiments show this approach is generally unsuccessful. However, by leaving a 50% cap in place, we have our only success — a limited one — with the ranF allocation graph. We will conclude that the best general set of policy choices is L-L-N-F (Largest Adapter Size, Largest PE first, PE at a Time, Full PE).

7.5.4 Comparison to Base Algorithm

A careful examination of the Beck algorithm[Beck95] reveals a few strengths of the TRANS_FIRST algorithm: complete determinism and improved execution time. Table 7.4 shows a comparison. BECK is almost as good in terms of packing success — it is, after all, a very good algorithm. However, BECK has a surprisingly large random component. Table 4 does not show this effect, but the values that were averaged to get network usage vary quite a bit. BECK orders the adapters by their size and packs in decreasing order to the PE which would minimize the network bandwidth. Early in the algorithm's execution there are quite a few ties, where placement to any PE would take zero bandwidth (since the adapter's neighbors haven't been allocated yet). Such ties are resolved randomly, which significantly affects the remainder of the execution. The policy choices selected for TRANS_FIRST preclude any random elements, thus reserving any randomness for the follow on allocation. Randomness is not, of course, always a bad thing. If BECK fails to find an allocation, it is always possible to re-execute it to see if it will find a different, successful, allocation.

Table 7.4 also shows a clear time advantage for the TRANS_FIRST algorithm. The average speedup of 2.7 is substantial, and is a result of the "divide and conquer" nature of the algorithm. By operating on small portions of the entire allocation graph (the regions near the transducers), choices are made among a much smaller set of adapters. Such small comparisons are much quicker than the large comparisons required by BECK as it examines the entire graph.

TRANS_FIRST

	ranA	ranB	ranC	ranD	ranE	ranF	sch	tract	iac
Net Usage	.32	.59	.35	.48	.73	0	.40	.26	.35
Pass Rate	1	.7	1	1	1	0	1	1	1
Execution Time (mS)	617	609	379	365	192	0	710	678	287
Speedup	2.2	1.8	2.3	3.2	3.4	0	3.1	3.1	2.8

BECK

Net Usage	.19	.62	.27	.39	.69	0	.43	.49	.6
Pass Rate	1	.6	1	1	1	0	1	1	1
Execution Time (mS)	1350	1080	855	1170	647	0	2215	2108	815

Table 7.4: Comparison to base algorithm

7.6 Conclusions

We have shown an adapter allocation algorithm that successfully exploits a constraint unique to the distributed embedded system domain. The algorithm works by allocating adapters that manage transducer hardware (and thus cannot be allocated elsewhere) to the local processing element, and then operating on the neighboring set of adapters. Large subgraphs are thus swept into a single processing element, which saves significant network bandwidth.

In order to tune the heuristic algorithm, a set of experiments was conducted. Each policy choice was clearly delineated and executed on a series of random and real-world allocation graphs. The following policy choices resulted in a heuristic algorithm with good packing quality and a substantial speedup: Largest Adapter First, Largest PE first, PE at a Time, and 100% PE Fill level. Such choices were used on all experiments involving adapter allocation found elsewhere in this thesis.

Chapter 8

Proof of Concept

This chapter presents an examination of the system-wide customization framework in the context of an another system, a distributed elevator control system. It is intended to validate the concept of product family architecture as a means to gain flexibility for customization. To demonstrate the viability of the PFA concept, the DFGs of two product instances were merged to form a PFA graph. The algorithms in the customization framework were applied to the elevator PFA graph, a process that exposed a few interesting subtleties. The chapter concludes with suggestions for solutions to these problems.

8.1 The Distributed Elevator Control System

In the Spring of 2001, a graduate-level class at Carnegie Mellon University examined reliability in complex distributed embedded systems. As a means to understand the reliability problems and solutions discussed in the class,

two student teams constructed the specifications for and a simulator implementation of a distributed elevator system. Extreme care was taken to deal with many of the real-world difficulties, and thus this class built two non-trivial elevator descriptions, unlike so many of the examples available in the literature [Knuth73, Liu87, Sha98, Sendall00, Coleman90]. These specifications made a good starting point for a credible PFA graph, along with some realistic resource usage comparisons (gleaned from the simulator code). Table 8.1 describes the elevator nomenclature we use in this chapter. The message nomenclature is available in Table 8.2.

The elevator simulators were constructed on a discrete event simulation framework, which had been strengthened through use in three undergraduate distributed embedded system courses. The framework provided Java APIs for networking, event notification and the elevator environment (people, sensor inputs, etc.). The project teams then generated Java classes to control the system actuators. The elevator was non-trivial – it included, for example, reliability goals and system-wide modes (normal, fire recall and fireman service).

Our viewpoint, as we examined the code base (two projects built on a common framework), was that of engineers in an elevator corporation who wish to meld the two projects into a single product family architecture for graceful degradation reasons. For instance, generation or refactoring of code was limited, as such a corporation would seek to take advantage of the company's code without the expense and risk of a major re-architecting project. We simply wished to see if several product instances of a suitably-distributed embedded system could gain the benefits of graceful degradation

through system-wide customization.

8.2 Building a PFA graph

Data flow graphs were generated for each team's control objects. The data flow graphs capture the flow of data from the sensors, through various adapters to actuators. To generate the DFGs, the data connections between the adapters were discovered, the data flow across the connections was measured, and the size of the various adapters was determined.

8.2.1 Connectivity

It was a relatively trivial task to determine the connections between adapters, because knowledge of the message types sent and received by each object was sufficient to establish each connection. Recall from Chapter 4.1.1 that embedded control networks are broadcast networks. When an adapter sends a message, it need not specify a recipient. The act of receiving the particular message type suffices to confirm data flow from the sending adapter. As each simulation object interacted with the network through the simulation framework API, a simple search for the proper method invocations for message transmission and reception determined connectivity.

8.2.2 Bandwidth Requirements

The acquisition of network bandwidth requirements was also relatively simple. The simulation framework provides a networking API for controllers and modules to send messages to each other. All messages are instantiations

Sensor	Description	Number	Location
DoorOpened	Detects when doors have fully opened	2	Door
DoorClosed	Detects when doors have fully closed	2	Door
DoorReversal	Passenger or object is obstructing the door	2	Door
DoorCloseButton	Request for doors to close	1	Car
DoorOpenButton	Request for doors to open	1	Car
ModeKey	Change elevator operating mode (fire response)	1	Car
CarCallButton	Destination request by passenger	per floor	Car
Weight	Detects weight of passengers in car	1	Car
AtFloor	Detects car position (above, below or at particular floor)	3 per floor	Hoistway
HallCallButton	UP/DOWN request by passenger	per arrow	Hallway
HoistwayLimit	Car position out of bounds (top and bottom of hoistway)	4	Hoistway
TimeOfDay	Wall clock time	1	Building
FireAlarm	Fire is possible	1	Building
Actuator			
DoorMotor	Opens/Closes the car door	2	Door
Car Position Indicator	Number display inside car. Shows current floor	1	Car
Car Lantern	Up/Down arrow. Shows car's current direction	2	Car
Car Light	Light inside CarCallButton	per floor	Car
Hall Light	Light inside HallCallButton	per arrow	Hallway
Hall Lantern	Up/Down arrow. Shows car's current direction	per arrow	Hallway
Drive	The motor that moves the car up and down	1	Hoistway
Controller			
Dispatcher	Determines where car should go next	1	
DriveControl	Controls speed and direction of drive	1	
HallLantern	Turns on correct direction arrow	per arrow	
HallButton	Detects HallButton. Controls Hall Light	per arrow	
CarButton	Detects CarCallButton. Controls Car Light	per floor	
CarLantern	Shows current direction on CarLantern	2	
CarPositionControl	Shows current floor on CarPositionIndicator	1	
DoorControl	Opens/Closes the door	2	

Table 8.1: Elevator nomenclature

of objects in the class hierarchy descending from MessagePayload. Each of the subclass objects was investigated (the 25 classes listed in Table 8.2) and manually examined to determine how big the payload needed to be to transmit the information in the object's fields. For instance, a message that needed to describe a floor would send a byte for that data. Messages in which multiple fields could consume less than a byte were combined into a one byte payload (*e.g.,* direction (1 bit) and speed (2 bits) can fit in a single byte). In no case did a message require more than the 8 bytes available in a single CAN message. Protocol overhead was added to each message that consumed additional bandwidth.

Message size is only one component of bandwidth — it is necessary to also know the transmission period of each message. Luckily, the networking API encapsulated a time-triggered mechanism that simply allowed the controller to specify how often the message should be sent. A simple trace module was inserted into the network object to capture the registration method calls and keep track of requested message periods. Since the controller objects don't dynamically change the period of their message transmissions, a single run was sufficient to capture the specified period. Both transmission period and message size are shown in the tables of Appendix A.

8.2.3 Adapter Size

Determining "size" for each adapter was a bit more complex. The total code size and size of all the fields in the object were measured (the tables that follow and in Appendix A are labelled with CODE and FIELD for these two quantities). These data would roughly map to the required flash and RAM

Message Name	Description	Payload Size
AtFloorPayload	Is Car near AtFloor sensor?	2 bytes
CarCallPayload	Is CarCallButton pressed?	2 bytes
CarLanternPayload	Floor to display on CarLantern	1 byte
CarLightPayload	Should CarLight be lit?	2 bytes
CarPositionIndicatorPayload	Floor to display on Indicator	1 byte
DesiredDwellPayload	Length of time to open doors	8 bytes
DesiredFloorPayload	Destination of next journey	2 bytes
DoorClosedPayload	Is DoorClosed entirely?	1 byte
DoorClosePayload	Is passenger pressing DoorClose?	1 byte
DoorMotorPayload	Direction and speed of Door Motor	1 byte
DoorOpenedPayload	Is DoorOpened entirely?	1 byte
DoorOpenPayload	Is passenger pressing DoorOpen?	1 byte
DoorReversalPayload	Is door obstructed?	1 byte
DrivePayload	State of Drive	2 bytes
DriveSpeedPayload	Direction and speed of drive	2 bytes
EmergencyBrakePayload	Is Emergency Brake enabled?	1 byte
FireAlarmPayload	Is Fire Alarm signalling?	1 byte
HallCallPayload	Is passenger pressing HallCall?	2 bytes
HallLightPayload	Should HallLight be lit?	2 bytes
HoistwayLimitPayload	Is car over bounds of hoistway?	1 byte
ModeKeyPayload	Has Fireman turned ModeKey?	1 byte
PeakModePayload	Is it rush hour?	1 byte
TimeOfDayPayload	What time is it?	8 bytes
WeightPayload	How many pounds are in the car?	8 bytes

Table 8.2: Elevator messages, meaning and payload sizes

requirements for the object when executing on a microcontroller. Further detailed information, in terms of runtimes (*i.e.,* cyclecount) or stack requirements would be possible, but require much more difficult dynamic analyisis. The algorithmic techniques developed for this research were not dependent upon such detail, so these two pieces of information were determined to be sufficient for the proof of concept. By the way, having multiple pieces of data for each adapter is important. As [Beck95] showed, multi-valued binpacking is problematic and additionally complex, but it is a necessary complexity. Software objects will need to be allocated to processing elements based on multi-valued requirements. This realistic challenge must be overcome, not merely avoided.

Class Loaders

The first attempt to measure size of the adapters was to exploit the Java classloader to determine object information at runtime. From within a Java method, it is fairly easy to find out a fair amount about an object. For instance:

```
public void
printInfo(Object obj){
  Class c = obj.getClass();
  if (c.isPrimitive()){
    System.out.println("Primitive classes aren't interesting");
    return;
  }
  Field [] fields = c.getFields();
  int objSize = 0;
  for (int i = 0; i < fields.length; i++){
```

```
    String name = fields[i].getName();
    Class type = fields[i].getType();
    int size = 0;
    // a small, illustrative cheat
    if (type.isPrimitive() && type.equals(Integer.getType())){
      size = 4;
    .... other checks for Booleans, bytes, strings, etc.
    System.out.println(name + " is of type " + type +
                       " and size " + size + " bytes");
    objSize += size;
  }
  System.out.println("Total size of all fields is " + objSize);
}
```

Chains of field descriptions can be followed to determine the size of any fields. Arrays are problematic — no method exists to inquire about the bounds of an array, when that array is represented by the `Class` object. A reference to the actual array could be queried with the `length` keyword, but getting that reference is difficult. The `Class` object has similar methods to discover code sizes (in bytes of bytecode) for each method of an object.

Notice, however, that the code above requires an instance of an object in order to get the `Class` object that starts the reflective process. How would one get instances of the objects? The reflective code (*i.e.,* the method `printInfo` shown above and its invocations) could be inserted somewhere in the code base under examination and then be executed after the objects of interest are instantiated. The fundamental problem with this approach is the lack of a common point which has references to the instances of all objects of interest, and yet would be unaffected by the measurements on itself. This problem was exacerbated, because the startup procedure for the code examined was far from clear.

Classloaders promise, yet ultimately fail to deliver, a related opportunity to measure the adapter sizes at runtime. A Java classloader is an extension to the Java Virtual Machine (JVM) that loads a class file when an object of the class is to be constructed. The classloader transforms the data that represents the class (such as a file on disk) into data for the JVM to execute, checks security permissions, and ensures the code is properly constructed bytecode. A default classloader is available in the `java.lang` package. It is a fairly easy matter to create an instance of a class from within Java code:

```
Class c = this.getClass();

// get classloader that was used to load this class
ClassLoader loader = c.getClassLoader();

Object inst = loader.loadClass("Dispatcher").newInstance();
```

There are a few subtleties here. The first is that the classloader used to create the object instance, in this case `loader`, is the one that was used to load the class in the first place. Unless the class was loaded across a network or in some other unusual manner — and for purposes of this experiment such is not the case — the default loader works fine.

The second subtlety is what kills this approach. It turns out that the `newInstance()` method will build a new instance of the object (`Dispatcher` in the example code) by calling the default, zero parameter constructor of the class. Those classes that don't have such a constructor cannot be handled by this approach. Such a restriction is necessary, as the classloader must construct the object and so must have the information the object

needs to create itself (the constructor's parameter list). It is possible to use the `class.getConstructors()` method to find out what constructors are available, choose one, and then use the `constructor.newInstance(Object [] initargs)` method to build the object using an alternate constructor. However, notice the work that such an approach would require: each object needs other objects built (its complete argument list), which must be valid objects if the instance is to complete the constructor and be measured accurately. This amount of work is equivalent (or worse) to inserting measurement code into the project code base.

Binary Formats

Fortunately, a better method to measure adapter sizes exists for the elevator system — that of examining the classfile itself. This works because most of the objects in the project are static, mainly due to the time-triggered nature of the system. Static allocation of code and data space is typical of distributed embedded control systems. So an examination of the classfile (in Java's binary code format) leads to a knowledge of size information. For instance, the classfile has information about an object's fields (type, name, size, etc). It also contains the bytecode for each of the methods in the object.

A parser was constructed to examine the data inside the classfile in order to determine the codesize and fieldsize of each adapter. Extensions were added to a general classfile parser[1] in order to collect and summarize measurements of each classfile. The resulting tool reads a classfile, determines

[1]Many thanks to Dr. Zvi Har'El, Department of Mathematics, Israel Institute of Technology for providing the code that became the core of the classfile parser.

the superclass, add up the bytes of bytecode in each method, and sums the lengths of all strings. In addition, it would print the name and type of each field and add up the length of all primitive fields (boolean, int, double...) and primitive wrappers (Boolean, Integer, ...). Arrays are still a difficult area, as the size of the arrays isn't determined at compile time. The parser did, however, print the name, type and dimensionality of each array, which made a source code hunt straightforward.

The codesize and fieldsize of each adapter are listed in the tables in Appendix A. The fieldsizes do not include strings, which are rare in distributed embedded control systems and are exclusively used for debugging information (almost without exception used in the instantiation of exception objects) in this system. Likewise, codesize does not include the size of superclasses, which did little that was considered indicative of a real control system. In production code on an optimized system, neither would have been included to start with. But, these simulations were executed on resource rich, non-embedded processors and the designers felt no need to eliminate unused fields or methods.

8.2.4 Missing Modules

The framework provides sensor and actuator mechanisms for several objects that should be part of the simulator. For instance, HallLights are used by the people to determine if they should get on an elevator. But there is no HallLight object. The framework simply passes the HallLight message straight to the people waiting in the hallway. Similarly, the HallButton messages are effectively created by the people objects. The following is a

list of the missing modules:

- Hall Lantern (Up/Down arrows in the hallway)
- Hall Light (Button lights in the hallway)
- Hall Button
- Car Lantern (The Up/Down arrows in the car)
- Car Lights (The lights on the buttons in the car)
- Car Button
- Car Position Indicator
- Emergency Brake (omitted, as should be mechanically linked)
- Fire Alarm

For each of the missing modules, sizes were based on careful study of similar functionality objects. Information about those modules is available in Table A.2.

8.2.5 Replication

A serious challenge presented by the elevator is object replication. Many of the objects in use, be they sensors, actuators or adapters, are present in quantity in the system. For example, each floor has a trio of `AtFloor` sensors to determine car location. Each door has a `DoorMotor`, and each `HallButton` has a controller. In a seven floor instance of an elevator there are 21 `AtFloor` sensors – each of which must be visible to the reconfiguration algorithm as a separate object. One of those sensors may be broken or (in

the case of an adapter) rehosted individually, so it must be represented individually.

In implementing such a replication of system objects, no elegant solution presented itself. A satisfactory system was implemented based on naming conventions. The naming mechanisms are visible in the tables of Appendix A. Each object can have a replication specification, which are tailored for the elevator domain. Replication specifications are things like "numFloor" or "LEFT/RIGHT" which indicate an element should be replicated for each floor of the elevator or replicated for a left and right side element. Replication is accomplished by generating the appropriate number of objects and modifying the name of the object in the appropriate manner. For instance, if `SensorA` has the `numFloor` replication specification, then 7 sensors (for our 7 floor elevator) are created, named `SensorA_Floor1`, `SensorA_Floor2`, through `SensorA_Floor7`. Subordinate elements may be replicated based on the results of the parent's replication. For instance, a Sensor that is replicated `numFloor` would have output messages replicated to `SameFloor` to ensure the names match properly.

8.2.6 Splitting Monolithic Controllers

The initial graph for TeamA reflects the hierarchical control style architecture used in its construction. A few large control objects (`Dispatcher`, `DoorController`, etc.) receive input from lots of sensors, compute control values (set points) which are communicated straight to the actuators. This is a natural architectural style for many distributed embedded systems. Unfortunately, the PFA graph was somewhat limited, because path lengths

were small and opportunities for rehosting controllers non-existent.

The PFA graph was enhanced through some small and opportunistic methods. Some of the monolithic controllers were broken apart without being re-designed. For instance, the `DoorController` has two "utility" functions that can become separate objects: a `ModeManager` and a `FloorFinder`. The `ModeManager` looks at the `FireAlarm` and `ModeKey` messages to emit a new message type describing the mode. The `FloorFinder` examines all the individual floor sensors and generates a message describing which floor the car is on. It turns out that such utility functions are needed in several components — for instance, the `Dispatcher` and `CarButtonController` also need the `ModeManager`— so by breaking out a utility, the size of other controllers also decreases. In addition, the total size of the system decreases dramatically. The utility object, in a singlely replicated item, replaces code used in several classes, some of which are heavily replicated. A modification of several adapters (`DoorController`, `CarPositionIndicator`, `CarLanternController`, `HallButtonController` and `Dispatcher`) to use the `ModeManager` or the `FloorFinder`s was undertaken, the results of which are shown in Table 8.3 (assuming a seven floor elevator). The point of the modification was not to save space, but to increase the configurations possible. With different versions of the controller adapters, the elevator PFA graph has more such possibilities.

The use of the utility adapters is not universal, as originally expected. The `DriveController` reads all of the messages from all of the `AtFloor` sensors (3 per floor), much like the `FloorFinder`. However, the `DriveController`'s requirements go beyond the simple abstraction enforced

by the `FloorFinder`'s interface. The `DriveController` needs access to individual sensor information, not merely an understanding of which floor the car is on.

An added opportunity arose to increase the configuration possibilities of the PFA graph. Slight modifications to the `DoorController` resulted in three different versions. `DoorController-FF` uses the `FloorFinder`. `DoorController-MM` uses the `ModeManager`, while `DoorController-FF-MM` uses both. The intent is to have a few more configuration options as well as allocation options available. The `DoorController` was further segmented into two modules — a state manager that detects state transitions and a behavior manager that produces the proper outputs based on the current state. Such a split was easy to manage for a state machine, doesn't change its behavior in unexpected ways and is an easy step for a designer facing this type of challenge. All of the modifications to TeamA's DFG are summarized in Table A.3.

Controller	Replicas	Original Size		New Size		Savings	
		CODE	FIELD	CODE	FIELD	CODE	FIELD
Door	2	4054	244	3589	208	930	72
Door-BH	2	—	—	360	38	-720	-76
CarPosition	1	670	87	310	40	360	47
CarLantern	2	731	51	653	36	156	30
CarButton	# Floor	938	49	817	48	847	7
HallButton	# Arrow	907	43	834	46	876	-36
Dispatcher	1	7433	478	7294	413	139	65
Mode Mgr	1	—	—	173	19	-173	-19
Floor Finder	1	—	—	207	19	-207	-19
Floor Finder2	1	—	—	516	71	-516	-71
					Total (assuming 7 floors)	1692	0

Table 8.3: TeamA's modified controllers. Code and Data sizes are in bytes

8.2.7 Team1

The system simulation developed by the other team, Team1, is quite similar in structure to TeamA's software. Team1 does include a modular safety control mechanism. A `SafetyDriveControl` monitors the `DoorOpen`, `DoorClosed` and `Drive` messages. In the case that a door is open while the drive is moving, the module sends out its own `Drive` message to shut down the drive actuator. It then sets the `EmergencyBrake`, as the `DriveController` is presumably untrustworthy of further operation.

Team1's DFG is described in Table A.4. Modifications to use the `FloorFinder` and `ModeManager` are likewise in Table A.5.

8.2.8 Completing the PFA graph

The completed PFA graph for the elevator needs only hardware information. Since the simulations were designed for use on general purpose computing platforms, in a non-distributed simulation, no hardware constraints were imposed. Because we wish to explore the usefulness of the algorithms, several hardware architectures (including sizing information) were proposed based on our industrial experience.

The PFA graph, save for the hardware information, is described in Tables A.1 through A.5. Appendix B contains an XML description of the elevator PFA, including all sizes and an example hardware description. The graph itself is too large to actually be drawn in useful form for this publication. The PFA graph for the three floor elevator has 33 sensors, 18 actuators, 85 adapters, 60 features (in 18 feature classes) and 115 data elements.

8.3 Algorithmic Changes

Execution of the algorithms developed in Chapters 6 and 7 on the elevator system highlighted a weakness of the phase 2 adapter selection algorithm. With the enormous choice available in the graph traversal, some trimming was necessary in the number of path combinations available. Otherwise, there are 3×10^{44} different combinations of paths available — far too many to be calculated or stored efficiently. One would think that the trimming process would be a fairly simple process, as we have determined the proper heuristic to help sort partial paths as the traversal progresses. Unfortunately, the heuristic proposed in the Chapter 6 was based upon the adapter size as a fraction of available resources. But, the replication issue interacts poorly with that heuristic — many of the adapter choices being made are from identically sized adapters being used as driver adapters for similar sensors. We were reduced to making arbitrary distinctions among choices in the traversal of the PFA graph.

A second problem, that of *path overlap*, occurs in the elevator PFA graph. Examine Figure 8.1 for a simple example. In this case, both Feature1 and Feature2 have 4 path alternatives to choose from:

1. Sensor1, AdapterA, Sensor2, AdapterC
2. Sensor1, AdapterA, Sensor2, AdapterD
3. Sensor1, AdapterB, Sensor2, AdapterC
4. Sensor1, AdapterB, Sensor2, AdapterD

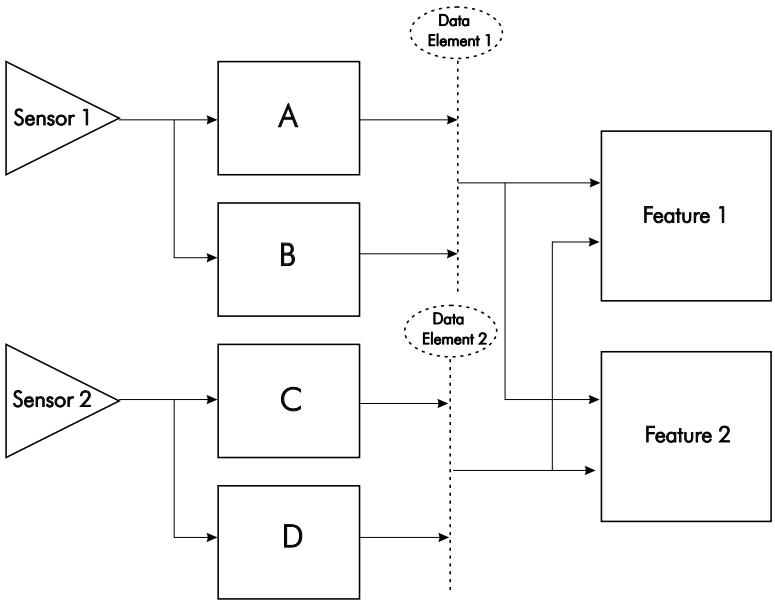

Figure 8.1: An example of path overlap

Recall that the generation algorithm collects a list of paths for each feature and then chooses one path per feature to combine into the allocation graph. If Feature1 chooses path number 1 from the list above, then Feature2 must also choose path number1. If Feature2 made a different choice — number 2 for instance — a conflict would ensue because both AdapterC and AdapterD transmit their individual results of the conversion of Sensor2's output.

This problem did not arise in the navigation system, because it had no overlap of the path fragments. In other words, much of the PFA graph was segmented into portions covered by the different features. In the elevator system, however, that is not the case. Many features need `AtFloor` sensor information, for instance. But, because the outputs of any particular `AtFloor` sensor can be handled by several different adapters, there is no

coordination to ensure only one such adapter is included in the allocation graph. It is a simple matter to check to ensure the path chosen for a feature doesn't conflict with the path chosen for a second feature. However, the likelihood of having non-conflicting paths is very slim on the elevator system — the path overlap is too great among the various features and the number of paths retained (*i.e.,* not trimmed away as discussed in the previous paragraph) for each feature is too small a fraction of the overall total. In the example of Figure 8.1, there is only a 25% chance that Feature2 will choose a path that doesn't conflict with Feature1's choice. In the elevator system, two features using the `AtFloor` sensors would not conflict with a probability of $\frac{1}{3 \times numfloors}$.

The solution is to re-order the traversal algorithm. Rather than execute several graph traversals — one per feature — to collect the path alternatives, the updated algorithm does a single graph traversal for each feature set. Further, during the traversal the results of a choice at a choice element is cached and the same list of alternatives is returned, no matter how many different features would require traversal from that choice element.

8.4 PFA Limitations

Futher examination of the elevator PFA graph brings to light additional limitations to the expressiveness of the PFA graphs. Unlike the navigation system, the elevator PFA does not have much sensor redundancy — few sensors can directly act as another sensor. The `DoorReversal` sensor, which detects blockages of the door, is one of the few sensors that can. In effect,

the `DoorReversal` sensors are redundant copies, either one of which is sufficient to request the reverse of both doors. In the elevator PFA graph, this situation is indicated by ensuring both sensors emit the same data element, which feeds both the left and right `DoorControllers`.

Other types of available redundancy are more difficult to express in the PFA graph. For instance, it is common for the current floor's `CarCall` button to act like the door reversal sensor to open the doors. When the car is on any other floor, it merely operates in the normal capacity to summon the car. The PFA graph has no way to express this second usage mode. This mode is always available in the `DoorController`, but there is no way to specify that the `DoorController's` requirements can still be met in the absence of both `DoorReversal` sensors.

More troubling is the situation with regard to the `AtFloor` sensors. Many of the non-critical features require information about the car's current position (*e.g.,* the `HallLanternController`, which lights up the indicator to provide floor location feedback to passengers waiting in the hallway). As drawn in the PFA graph, they require knowledge of all `AtFloorUp` and `AtFloorDown` sensors, and cannot be implemented in the absence of even a single such sensor. However, in reality they will still operate in the absence of some `AtFloor` sensors. They merely operate at a lower utility — they do not display floor information for a particular floor if it's `AtFloor` sensors are missing, or display a floor value for too long if just one direction `AtFloor` is missing. To specify such operation with the PFA graph, which has no notion of optional inputs or "as many as possible" flow, would be difficult. It can be done by adding one "null" adapter for each possible

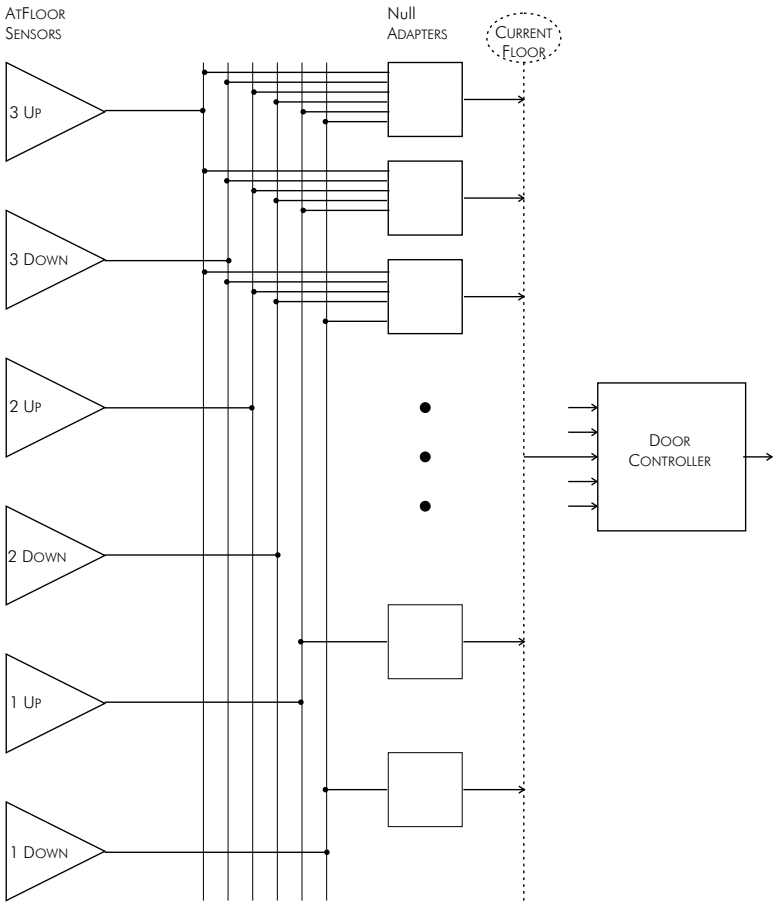

Figure 8.2: A (difficult) solution to the combination sensor problem

combination, the output of which flow into a data element, so that one of the combinations will be chosen. An example of the solution is shown in Figure 8.2. To keep the network requirement bookkeeping in proper shape, the null adapter would need to be allocated wherever the receiving adapter (`HallLanternController` in this case) is allocated. To ensure such co-location, an infinite bandwidth (or at least more than available) would be assigned to the data element that connects them. For our three floor elevator, $2^6 - 1$ null adapters would be necessary to cover all combinations of at least one `AtFloor` sensor. Similar combinational explosion would be required for other features that need `AtFloor` input. Even if the other feature has identical `AtFloor` requirements (*i.e.,* needs the exact same sensors), the need to allocate the null adapters with the second feature derails any idea of sharing null adapters.

Similar problems exist on a smaller scale with a few other elevator sensors. The state of one `DoorOpened` sensor can be inferred from the other and could thus be analytically simulated. The `Dispatcher` can continue to operate in the face of broken `HallCall` or `CarCall` sensors. In fact, the `Dispatcher` can operate (in a highly inefficient manner) without any `HallCall/CarCall` sensors by merely travelling up and down the hoistway, stopping at each floor. It may be a low utility solution, but it isn't fully broken. A better solution can occur for an intelligent `Dispatcher` that knew the expected distribution of request messages. Such a `Dispatcher` might deduce missing requests and simply synthesize replacements. Either solution would require a host of null adapters to collect the `HallCall` and `CarCall` combinations in a manner similar to Figure 8.2.

The solution discussed in the previous two paragraphs is admittedly less than elegant. An alternate means for handling the problem is to modify the PFA graph and adapter selection algorithm in order to handle the situation as a special case. A pseudo-sensor can be inserted in the PFA graph to represent "all available `AtFloor` sensors." Phase 1 and 2 algorithms can then run unchanged. But, before the allocation step of Phase 3, the allocation graph would be converted, by special purpose code, to replace the pseudo-sensor with any `AtFloor` sensors that are available. Similar pseudo-sensors can be used for `HallCall` and `CarCall` sensors.

The fact that special case code is employed does not reduces the general applicability of the system-wide customization process. The supporting insight is tied to a fundamental understanding of how the system-wide customization algorithms will be used. The customization manager will always be tied directly to particular PFA graphs. When the automobile of our operational scenario (Chapter 5.1.1) has a hardware problem, it is not the customization manager of the elevator that is used to fix it. Both the tow truck and the OnStar server are forced to operate upon the PFA graph that are specific to the particular product model that is broken. If the tow truck operator is to service several different product lines (which is likely, as different manufacturers will not share PFA details), then he will be forced to have different customization manager algorithms anyway. Inclusion of special purpose code is unfortunate in that it may require regeneration of the experimental data of Chapters 6 and 7. However, the experimental results will then most likely produce tuning parameters that increase the effectiveness of the specific customization manager on the PFA graph.

Of these two particular solutions to the combination problem, the former has an extra degree of freedom that might result in some better solutions. In a bandwidth constrained system, inclusion of all available sensors could potentially consume too much bandwidth. By explicitly specifying dropped sensors as a potential solution, the possibility exists that adapter allocation would be successful with only a subset of available sensors in use. The pseudo-sensor approach removes such possibilities by including all available sensors. If a particular system is bandwidth constrained, then the null-adapter approach is recommended as a means of exploring additional configuration space. However, the potential cost of increased phase 2 iterations (to traverse and explore the various combinations of null-adapters) makes pseudo-sensors the preferred mechanism for all other systems.

8.5 Graceful Degradation in Action

We executed a simple fault injection experiment to ensure the algorithms operate correctly on the elevator system. First, we had to generate a plausible hardware architecture — a specification of the number and size of available PEs — for the system. The simulation that served as the archetype for our PFA graphs did not contain any information or assumptions about the hardware architecture of the system. The simulations were executed on general purpose computing systems, so the designers had no reason to develop hardware constraints. For this experiment, we chose a hardware architecture similar to the "smart sensor" vision discussed in Chapter 1 — each sensor and actuator has its own microcontroller (PE). Five additional

PEs were added to act as compute nodes for the controller adapters. The PEs were given sufficient resources (RAM and Flash) for all of the preferred adapters to be allocated. Utility values were assigned randomly.

Step #	Utility	Remarks
1	1213	13 features in solution
	↓ 1 PE removed ↓	
2	1212	Left DoorController replaced
	↓ Six random adjustments to Flash/RAM ↓ Reduced PE2 to 50 bytes of RAM (essentially unusable) ↓	
3	1212	Same features as #2
	↓ Reduced PE3 Flash by 50% ↓	
4	1157	Remaining RAM is very tight – only 22 bytes unused
	↓ Another PE removed ↓	
5	693	7 features (4 of them critical) in solution
	↓ PE3 reduced to 100 bytes of RAM ↓ Only 2 PEs still functional ↓	
6	413	Only critical features remain

Table 8.4: Fault injection results

After the first and each subsequent allocation, we broke elements of the system by removing PEs or some PE capability. We then executed the system-wide customization algorithms and noted the resulting utility of the system. All failures were confined to the compute nodes, as the solutions to the limitations discussed in the previous section were not implemented. Decisions as to which PE or resource to break were made randomly, with a 25% chance of breaking a PE. If a PE was not broken, then one of the resources (either RAM or Flash) on one of the remaining PEs was randomly

(and uniformly) chosen. The chosen resource was reduced by 100 or 1000 bytes (respectively for RAM and Flash), or by one more than the remainder after the last allocation. The results are summarized in Table 8.4.

The system started with a total utility of 1213, in 13 implemented features. The five PEs were assigned abundant resources to start, so allocation happened easily. After the first run, one of the five PEs was removed. The customization algorithms allocated the same functionality, resulting in no loss of utility. After the second run, six adjustments were made to RAM and Flash, each time resulting in a valid allocation. These six steps are not shown in the table, because they merely tightened up the available resources. In the process of removing resources, though, a second PE was reduced to 50 bytes of RAM, less than required to host almost 60% of the adapters in the PFA graph. By the fourth run, RAM was scarce, with only 22 bytes unused systemwide. The resulting utility of 1157 represented a mostly functional system – all 13 feature classes were still implemented. The fifth run was quite different, however. The removal of an entire PE, when resources were already in short supply, represented a large step — which was reflected in the resulting utility of 693. After the fifth run, only 7 features were available — the elevator was still operational, but most user feedback (button lights, position indicators, etc.) was not. The last run was again a big step down in utility as a result of the virtual failure of a third PE. Only the critical features were operational.

This fault injection experiment, while limited, shows the strength of system-wide customization as a mechanism for graceful degradation. The elevator system was able to withstand a number of faults before losing func-

tionality. Passengers would not have noticed any reduction in capability until two of the five compute nodes had failed. And even after another PE failed, the elevator could still deliver passengers to their destination — admittedly slowly, because no controllers remained for handling user input. These results bode well for the concept of automatic graceful degradation as a means to achieve reliability goals.

8.6 A Few Notes About Performance

The customization manager implementation was not designed with performance in mind, flexibility as a research platform was judged as the more important trait. Furthermore, as discussed in the operational scenario, Section 5.1.1, execution time is dominated by operational effects, such as the response time of maintenance personnel. Acceptable execution time is generally on the order of seconds or minutes. However, the results of a few measurements are instructive. Table 8.5 shows benchmark results, using the rough timing mechanisms native to Java, of the six steps of the fault injection experiment. Execution times are shown in milliseconds.

Clearly, the most time is spent in Phase 2 which is a complex graph traversal process. Efforts to speed up the algorithm should concentrate in Phase 2 and 3. The algorithm in Phase 1 is extremely simple, so uses little time. The execution time of Phase 2 and 3 are roughly linear functions of the number of iterations, which is not surprising — the slight effects of different allocation graph sizes on Phase 3, for instance, are evened out over the vast number of iterations the algorithm made.

Step Num	Iterations			Execution Time (mS)		
	Phase 1	Phase 2	Phase 3	Phase 1	Phase 2	Phase 3
1	1	1	1	10	2604	53
2	2	65	64	10	3439	4394
3	2	65	64	10	3439	4394
4	20	1236	1217	17	51360	19660
5	29	1372	1344	37	49784	22115
6	39	2625	1650	40	98077	27450

Table 8.5: Performance of each phase

Elevator Floors	Phase 2 Execution Time (S)	PFA Graph # Vertices
3	2.927	191
4	8.338	232
5	16.756	273
6	24.615	314
7	40.782	355

Table 8.6: Execution time of phase 2 for various sized elevators

The number of iterations for Phase 2 are of some concern, however. The graph traversals are relatively time consuming and are something of a "brute force" attack on the PFA graph. Some simple speedups are likely available, such as caching traversal information from one iteration to the next. But a fundamentally different algorithm might be necessary for large PFA graphs. To test the growth of the iteration count of Phase 2 algorithm invocations, we expanded the PFA graph by using a simple stratagem. We increased the number of floors in the simulated elevator, and executed the customization algorithms on the resulting PFA graphs. The results are shown in Table 8.6. As the PFA graph increased in size (which it does linearly as a function of the number of floors), the execution time of Phase 2 increased about proportionally to the third power of the number of floors.

8.7 Conclusion

Applying the ideas and techniques of the previous chapters to a different domain helps to highlight the strengths and root out weaknesses of those techniques. This chapter discussed the examination of a PFA derived from two high quality simulations of an elevator control system. The elevator has different characteristics from the navigation system that was used to tune the algorithms, so it is no surprise that a few problems were encountered. Most of the problems surfaced in the combinational explosion of the PFA graph, caused by replication of the sensors, adapters and actuators in the elevator system. A specification mechanism based on naming strategies allowed simple creation of the PFA graph, though the graph grows significantly as the size of the elevator increases.

With the huge size of the PFA graph, the traversal algorithm of Phase 2 has difficulty handling the large number of possible path combinations. A trimming method is proposed to enhance the traversal, though it has limited applicability due to the lack of discrimination among the replicated sensors — they are all the same size.

Finally, the lack of sensor redundancy limits the types of hardware failures that can be tolerated by the system. Failure of PEs used primarily as compute nodes can be tolerated to the extent that additional computing resources can host critical adapters. But, failure of key sensors or actuators will cause complete system failure. Two techniques to specify sensor redundancy were discussed: the enumeration of all tolerable sensor combinations through the addition of null adapters and the use of pseudo-sensors

coupled with special purpose code to replace such sensors prior to allocation. The former is computationally challenging, and the latter relies upon graph-specific code.

Overall, the elevator proof-of-concept was a fascinating examination of the strengths and pitfalls of the system-wide customization concept. Fortunately, the pitfalls can be circumscribed without overly effecting the usefulness of customization as a mechanism for graceful degradation.

The fault injection test sequence validated the idea of system-wide customization as a means to graceful degradation. The example elevator was still able to provide service to passengers after losing three PEs and computing resources from the other two. Further, it persevered by shedding functionality, not merely by failover to spare resources.

Chapter 9

Conclusions

This dissertation has examined automatic graceful degradation mechanisms. The specific problem addressed was to find techniques to maximize the functionality of fixed hardware by choosing and allocating software from a flexible library of components. The dissertation has been exploratory in nature, surveying the landscape, and finding a route to a solution. We have framed a general approach to graceful degradation through the customization mechanisms and provided some baseline algorithms for basic implementations. More refined algorithms can easily be substituted in situations where the baseline assumptions do not hold. Section 9.1 is a summary of the findings. A more detailed discussion of our findings, including a discussion of what areas need to be examined more carefully, is in Section 9.2. A specific list of contributions appears in Section 9.3. Plenty of room remains in the research field for further work, be it in improving the algorithms, more expressive specification methods, better models, or varied styles of customization manager. Speculation on such future work follows, in Section 9.4.

9.1 Summary

System-wide customization is a useful mechanism, able to provide graceful degradation and other benefits, as discussed in Chapter 2. Graceful degradation is one of the truly valueable techniques to achieve the reliability requirements of a modern distributed embedded system. By using customization, a system is able to accomplish *automatic* graceful degradation, as opposed to the manual methods that resist scaling as the number of components increases.

Chapter 4 defined the system model that was used in the rest of the thesis. The system model is a familiar network connection of processing elements, each potentially attached to sensors and actuators. Because the processing elements are general compute engines — microcontrollers or microprocessors — they can execute algorithms designed for use with other sensors and actuators. The problem definition of Chapter 5 describes how the general processing power of the processing elements can be exploited by allocating software components to compose a system with reduced functionality. The chapter described several models for describing the graceful degradation process, including the Product Family Architecture (PFA) graph. The PFA graph makes use of the various product instances in a product family graph. The PFA graph is the supergraph of the data flow graphs of each product model, merged by joining communication elements according to their type. Chapter 5 also described a feature model that describes desirable portions of functionality and is expressible in the PFA graph.

A three phase, iterative, algorithmic framework was proposed in Chap-

ter 6. Heuristics were examined to populate the algorithms of the framework. The feature selection algorithm handles the details of the feature model and generates a list of features to implement. The second phase is adapter selection, which operates upon the PFA graph to propose a set of adapters that fulfills all the dependencies of the features. The third phase handles the mapping of adapters to the available hardware: processing elements and the network. A novel technique for adapter allocation was developed, one which allocates the adapters in a data flow graph based on their proximity to the sensors and actuators of the graph. The algorithm was described, along with the policy choices to guide the algorithm's decisions, in Chapter 7.

The development of the algorithmic framework was grounded by using a hypothetical navigation system for automobiles. In order to ensure wider applicability, the algorithms were tested upon an elevator simulation, the results of which are in Chapter 8. Two product instances of the elevator were measured and a PFA graph was generated. In the elevator, a great deal of component replication exists that was not anticipated during work on the navigation system. Such replication creates difficulties in the generation of a PFA graph, because explicit representation makes for a large graph. Implicit replication, where only the type of component, and not each particular component, is represented in the graph does not sufficiently handle failure situations where only some of a component type are broken. The elevator system also had a great deal of sensor overlap, which did not exist in the navigation graph. The sensor overlap required a simple re-ordering of the adapter selection algorithm to ensure paths were consistent across features.

A fault injection experiment on a small instance of an elevator was suc-

cessful in demonstrating the ability of the system-wide customization algorithms to generate valid configurations for each hardware configuration. In the experiment, the elevator lost 60% of its processing elements, yet continued to operate, though at reduced functionality. The fault injection experiment was a feasibility demonstration of the customization manager providing graceful degradation capability to a distributed embedded system.

9.2 Retrospective

We have stated several times that this work has been a journey of exploration, because the field is new. This section provides a look back at the decisions and assumptions of the journey, as well as a discussion of alternate paths we might have taken. We have shown a positive answer to the central question, of whether system-wide customization can be made to work as a mechanism for graceful degradation. But, there are many different choices that make up a customization process and we only demonstrated one path. In some cases, we were lucky and found that nettlesome problems did not apply to our specific circumstances. In others we found workarounds or other engineering sleights-of-hand to reduce the complexity of the problem. The following subsections discuss our assumptions, the workarounds and other routes we might have taken. Throughout these sections, keep in mind that we are merely refining the central question and how we think about it, not disavowing it.

9.2.1 The PFA Model

We based much of our solution on the exploitation of a product family architecture (PFA). The flexibility and redundancy required for system-wide customization flows from the availability of various alternative hardware and software components in the PFA. The major assumption, of course, is in the availability of such a PFA. For many product domains, access to a PFA is a forgone conclusion — especially in recent years. For instance, major automotive manufacturers have created or are in the process of forming a software architecture at the system level. However, in a field that does not document a PFA, one may be constructed by examining the network messages of the various products. The network message types are the inter-subsystem interface. They constitute the merge points for building the PFA graph. This situation is exactly what happened in the proof of concept of Chapter 8. Without documentation of the PFA (or even the data flow graphs of the product instances), the network messages still provided enough data to construct individual DFGs and finally a PFA graph.

The use of a PFA graph built on a data flow model seems quite useful for a broad range of systems, many of which inhabit the distributed embedded domain. However, data flow is not a universally useful model. Exceptional condition paths and conditional branches are difficult to elegantly incorporate. Some problem domains do not exhibit their interesting and challenging aspects in terms of data flow. An example is a financial system where transactional processing is the norm.

If the data flow paradigm does not apply, another mechanism must be

sought. The role of the PFA graph is primarily to represent alternate software components, organize dependencies, and allow verification that a collection of components is a "complete" system. System designers could generate an explicit database with such information as an alternative to a PFA graph. We believe the process of developing a PFA graph is actually quite similar to the process of developing such a database, except that data flow semantics provide shortcuts that reduce the effort involved. Data flow between components is a powerful indicator of dependency, for instance.

9.2.2 Feature and Utility Models

The use of feature selection is easily a novel aspect of this dissertation. Other research based on compositional systems assumes that all features must show up in the final product. We break such an assumption, instead choosing a subset of all available features in order to achieve graceful degradation. In essence, we have proposed functional redundancy as an additional fault-tolerant technique, to be added to the arsenal of tools that exploit different types of redundancy: modular, analytical, temporal, and design.

Functional redundancy requires a feature model to articulate how different portions of the system can be combined. The class-based feature model we used is of medium complexity — useful to represent many distributed embedded systems, but not comprehensive. It is quite easily applied to functional decomposition type system architectures. In such cases, the functions map to the different feature classes. In cases where the system architecture was generated through some other means than functional decomposition, a deeper semantic model of the features might be useful.

The feature model we used employed particular adapters in the PFA graph as stand-ins for each feature. Such a model merely uses the adapter as an implication of a feature. In reality, the feature is more than a single adapter – it is any of the possible ways that the adapter and all its dependencies can be satisfied. By using a single adapter and allowing the dependencies to be represented in the PFA graph, the data structures are simplified considerably. But in the end, our use of an adapter is merely a bookkeeping strategem. Nevertheless, the method does allow for manipulation of the PFA graph to represent complex features. For instance, a feature that represented several disparate software components can be represented by adding a zero-sized adapter to the PFA graph, with zero sized communication from the various components to the new adapter. The zero-sized adapter represents the feature. Other complex features (for instance, "either-or features" or "m-of-n features") can be represented by similar manipulations.

Another central problem that any system-wide customization mechanism will have to overcome is the difficulty of designing a satisfactory utility model to represent the desirability of particular features. We chose a simple scheme wherein each feature was given a numeric utility value, the sum of which represented the desirability of a configuration. Such a model has no support to guide the designer in generating numeric values. In the worst case, pairwise comparisons might be required between each pair of features — which does not necessarily lead to a globally valid ranking of the features (similar to the well-known voting problem).

Another alternative for a utility model is a qualitative ranking system,

where features are ranked in one of several equivalence classes (*e.g.,* "good", "better", "best"). The algorithm could then make clear choices whenever feature combinations included different rankings of features. If features of the same class were compared, then a random or some other tie-breaking strategy would be needed.

The utility problem is one that will exist in any system-wide customization system. It is a difficult problem, any solution to which needs to ensure the designer's desires can be properly expressed without overwhelming him.

9.2.3 Allocation Algorithms

There are many allocation techniques that could have been employed in Phase 3. We chose a binpacking process. However, equally valid techniques can be found that use a different underlying process. Graph theoretic methods, such as [Stone77], construct a graph of the adapters (and sometimes, the processing elements) in such a way that graph theory tools like min-cut can be used. Graph cutsets then correspond to adapter allocation assignments. Other allocation methods include integer programming, clustering heuristics, and guided search.

The only requirement that made our allocation algorithm different from any of these standard allocation methods is the presence of sensors and actuators in fixed locations. Most allocation algorithms assume heterogeneous processing elements (PE), whereby any software runs equally well on any PE. In a distributed embedded system, the processors may be equally capable of running any software, but they are distinguished by the particular sensors and actuators attached. Our choices with respect to the standard

allocation algorithm amount to:

1. Use a standard allocation method and treat the sensors and actuators as software components. Allow the algorithm to allocate the sensors and actuators. Adjust the allocation (or give up) if the sensors and actuators are allocated to different processing elements from their real locations.

2. Use a standard allocation method and ignore sensors and actuators. Allocate just the software components. Adjust the allocation by adding communication from the sensors and actuators to the already allocated software components. Adjust the allocation (or give up) if the added communication overwhelms the network.

3. Modify a standard allocation method to properly handle sensors and actuators. In many cases this might be achievable by simply starting the allocation after sensors and actuators have already been placed (in the proper PE, of course). For instance, if a clustering heuristic is used, the clusters would start based on the assignments of sensors and actuators. If binpacking, then the bins would start with the sensors and actuators already packed.

While working on the third choice, we discovered that we could actually exploit the constraint, rather than merely work around it. The result was the transducer-sensitive algorithm of Chapter 7. Similar opportunities might be possible if integer programming, clustering or guided search methods were similarly modified to account for sensor and actuator location.

9.3 Contributions

The contributions of this research are summarized below:

Problem Identification

– This was the first comprehensive treatment of the system-wide customization problem. A formulation of the problem definition was developed. We framed the question of how to customization a system, in the context of distributed embedded systems, and then refined the ways to think about the question. (Chapter 5)

– Ramifications of solutions were examined as they apply to the distributed embedded system domain. Pros and cons of the PFA approach were discussed. (Chapter 2)

Problem Solutions

– A three-phased solution framework was developed. Algorithms were presented to solve each of the three phases. Experimental results, gained through construction of a tool—called a *customization manager* — provided key parameter choices for building the algorithms. (Chapter 6)

– The entire idea of feature selection is one of the most novel aspects of this dissertation. Fundamentally, feature selection allows only parts of a specified system to be implemented, unlike other system construction techniques which assume the entire specification must be met.

– The allocation of software components to hardware (phase 3) is well known to be NP-complete. A new allocation heuristic was proposed, which exploits characteristics of distributed embedded systems, resulting in a 2.7x speedup on example systems. (Chapter 7)

– Phase feedback was proposed as a means to increase speed and quality of problem solution. A feedback mechanism was examined for the feature selection algorithm (phase 1) that resulted in drastically improved solutions. (Chapter 6.1.1)

Proof of Concept

– Two product instances of a complex distributed embedded system were measured and combined into a single product family. (Chapter 8.2)

– The customization manager was used to examine system-wide customization of the proof of concept system in response to various hardware failures. (Chapter 8.5)

9.4 Future Work

Since this research opens up a new research field, there are plenty of issues to explore. The issues can be categorized in the following partitions:

9.4.1 System Model

One area that has not been examined are applications with real-time requirements. The data flow graphs do not express any timing requirements.

In fact, since time is not a consumable resource in the same way that memory or I/O channels are, it would be quite interesting to explore. A simple approach would add a schedulability check to the validity checks of a fourth phase. However, it is probably much more efficient to have a way to express the timing requirements in the PFA graph, such that the adapter selection only generates allocation graphs that have been pre-vetted or at least heuristically selected for schedulability. Because the timing constraints can only be fulfilled by the cooperation of all software along the critical path, it makes sense to include such considerations when the PFA graph is manipulated. However, the actual timing will also depend upon the allocation decisions. For instance, communication among components on the same processing element presumably will take less time than communication over the network. Similarly, execution timing of adapters is dependent upon the speed of the processing element. A promising attack to this problem would be for the adapter selection to collect and screen adapters along the critical path, have the adapter allocation algorithm check and allocate those components, and then go back to the adapter selection for the remainder of the system.

9.4.2 Problem Definition

Two issues with the problem definition are in need of examination: the PFA graph and feature models.

The mechanics of the PFA graph are sufficiently expressive to cover the systems we have explored. However, additional semantics would make for cleaner graphs, and may simplify the adapter selection algorithms. We mentioned in Chapter 8 the need for optional or low-criticality connections in

the PFA graph. It would be very helpful to be able to express the "as many as possible" type of relations, and even to put bounds on the range of required inputs. Furthermore, the replication of components, as illustrated by the elevator system, should be expressable without requiring full enumeration. Unfortunately, any change to the semantics of the PFA graph will complicate the algorithms in the adapter selection phase to at least the same degree as they simplify the PFA graph. Finding a good tradeoff is critical.

The representation of features in a composable and consistent manner is also a difficult problem. [Shelton02] is a good start, with hierarchical feature subsets and a quantitative utility model. A detailed and quantitative model is required for the customization algorithms to compare configurations. But the emphasis on detail and precision makes the process of designing such a system more difficult. We do not have any good ideas on how to ensure detailed numbers are meaningful — beyond a "big", "bigger", "biggest" type of triage.

9.4.3 Algorithms

Alternate algorithm construction is also possible as an interesting comparison to the algorithms of Chapters 6 and 7. The customization manager algorithm developed in this dissertation 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 — 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.

Integer programming constructs have been suggested, especially as a solution means to the adapter allocation phase. Using integer programming for the entire problem appeared exceptionally challenging. However, a new tool, called *constraint programming* has recently become available in the operations research field. Constraint programming uses a two level architecture:

1. *a constraint component:* a constraint-solving system reasoning about fundamental properties of the system constraints such as satisfiability;

2. *a programming component:* specifies how to generate, combine and process constraints, often in non-deterministic ways.

It is possible that the myriad of system issues and constraints could be modelled, in a tractable manner, for solution by a constraint programming engine[Van Hentenryck99].

9.4.4 Customization Manager Styles

Several possibilities exist for useful extensions to the customization manager. Dynamic customization would be a very useful capability — the ability to execute a customization manager while the system is operational. To do so would involve a careful dance between two modes. In the first mode, a reconfiguration trajectory is calculated that would change system functionality to relocate adapters and free up space for alternate adapters, all the while ensuring the system is always functional. The second mode would

load adapters onto available processing elements — while ensuring the impact of network communication does not interfere with system operation. As a first start, the planner could move the system to a state with minimum functionality (only critical features are implemented, perhaps). The freed resources could then be populated with improved utility critical features and optimization features. Clearly the calculation of reconfiguration trajectory would be challenging.

Another useful customization manager style that would not be as difficult to construct is a failover friendly one. A failover friendly customization manager ensures replicated adapters are available for critical features in case a failure occurs. A useful fault model would guide development of strategies to ensure minimum functionality adapters were replicated to take over in case of a failure. For instance, if the fault model specified failure of any single processing element was to be tolerated, then each critical adapter would have to be replicated on two different processing elements. The replicated adapters do not need to be identical to the operational adapters. Ideally, the replicated adapters would be selected to be on the easiest to allocate of all the paths for any feature in the feature class. Allocation constraints would be imposed to keep the replicas on different processing elements.

Appendix A

Elevator System Tables

Table A.1: Team A — Original code measurements

	Name	Replication	CodeSize	FieldSize	Period	Payload
Sensor	DoorOpened	LEFT/RIGHT				
Outputs	DoorOpenedRAW	SameSide			100mS	1byte
Adapter	DoorOpenedAdapter	LEFT/RIGHT	625	30		
Inputs	DoorOpenedRAW	SameSide				
Outputs	DoorOpened	SameSide			100mS	1byte
Sensor	DoorClosed	LEFT/RIGHT				
Outputs	DoorClosedRAW	SameSide			100mS	1byte
Adapter	DoorClosedAdapter	LEFT/RIGHT	698	30		
Inputs	DoorClosedRAW	SameSide				
Outputs	DoorClosed	SameSide			100mS	1byte
Sensor	DoorReversal	LEFT/RIGHT				
Outputs	DoorReversalRAW	SameSide			100mS	1byte
Adapter	DoorReversalAdapter	LEFT/RIGHT	871	26		
Inputs	DoorReversalRAW	SameSide				
Outputs	DoorReversal	SameSide			100mS	1byte
Sensor	Weight	1				
Outputs	WeightRAW	1			1mS	8bytes
Adapter	WeightAdapter	1	351	26		
Inputs	WeightRAW	1				
Outputs	Weight	1			1mS	8bytes
Sensor	TimeOfDay	1				

continued on next page

Table A.1: (Continued)

	Name	Replication	CodeSize	FieldSize	Period	Payload
Outputs	TimeOfDayRAW	1			1mS	8bytes
Adapter	TimeOfDayAdapter	1	437	54		
Inputs	TimeOfDayRAW	1				
Outputs	TimeOfDay	1			1mS	8bytes
Sensor	AtFloorUp	numFloor				
Outputs	AtFloorUpRAW	SameFloor			100uS	2bytes
Adapter	AtFloorUpAdapter	numFloor	640	48		
Inputs	AtFloorUpRAW	SameFloor				
Outputs	AtFloorUp	SameFloor			100uS	2bytes
Sensor	AtFloorStop	numFloor				
Outputs	AtFloorStopRAW	SameFloor			100uS	2bytes
Adapter	AtFloorStopAdapter	numFloor	640	48		
Inputs	AtFloorStopRAW	SameFloor				
Outputs	AtFloorStop	SameFloor			100uS	2bytes
Sensor	AtFloorDown	numFloor				
Outputs	AtFloorDownRAW	SameFloor			100uS	2bytes
Adapter	AtFloorDownAdapter	numFloor	640	48		
Inputs	AtFloorDownRAW	SameFloor				
Outputs	AtFloorDown	SameFloor			100uS	2bytes
Sensor	HoistwayLimit1	UP/DOWN				
Outputs	HoistwayLimit1RAW	SameDirection			10uS	1byte
Adapter	HoistwayLimit1Adapter	UP/DOWN	710	39		
Inputs	HoistwayLimit1RAW	SameDirection				
Outputs	HoistwayLimit1	SameDirection			10uS	1byte
Sensor	HoistwayLimit2	UP/DOWN				
Outputs	HoistwayLimit2RAW	SameDirection			10uS	1byte
Adapter	HoistwayLimit2Adapter	UP/DOWN	710	39		
Inputs	HoistwayLimit2RAW	SameDirection				
Outputs	HoistwayLimit2	SameDirection			10uS	1byte
Sensor	ButtonControl	1				
Outputs	ModeKeyRAW	1			500uS	1byte
	DoorCloseButtonRAW	1			500uS	1byte
	DoorOpenButtonRAW	1			500uS	1byte
Adapter	ButtonControlAdapter	1	266	3		
Inputs	ModeKeyRAW	1				
	DoorCloseButtonRAW	1				
	DoorOpenButtonRAW	1				
Outputs	ModeKey	1			500uS	1byte
	DoorCloseButton	1			500uS	1byte
	DoorOpenButton	1			500uS	1byte
Actuator	DoorMotor	LEFT/RIGHT				

continued on next page

Table A.1: (Continued)

	Name	Replication	CodeSize	FieldSize	Period	Payload
Inputs	DoorMotorRAW	SameSide				
Adapter	DoorMotorAdapter	LEFT/RIGHT	1631	58		
Inputs	DoorMotor	SameSide				
Outputs	DoorMotorRAW	SameSide			100uS	1byte
Actuator	Drive	1				
Inputs	DriveSpeedRAW	1				
Adapter	DriveAdapter	1	2716	58		
Inputs	DriveSpeed	1				
Outputs	DriveSpeedRAW	1			100us	2bytes
Feature	DoorControlMonolithic	LEFT/RIGHT	4054	244		
Inputs	DesiredFloor	1				
	DesiredDwell	1				
	DoorReversal	LEFT/RIGHT				
	DoorOpened	LEFT/RIGHT				
	DoorClosed	LEFT/RIGHT				
	HallCall	numArrow				
	CarCall	numFloor				
	DriveState	1				
	AtFloorStop	numFloor				
	ModeKey	1				
	FireAlarm	1				
	DoorOpenButton	1				
	DoorCloseButton	1				
Outputs	DoorMotor	SameSide			100uS	1byte
Feature	CarPositionControl	1	670	87		
Inputs	DesiredFloor	1				
	AtFloorUp	numFloor				
	AtFloorStop	numFloor				
	AtFloorDown	numFloor				
Outputs	CarPositionIndicator	1			200us	1byte
Feature	Dispatcher	1	7433	478		
Inputs	Weight	1				
	TimeOfDay	1				
	ModeKey	1				
	FireAlarm	1				
	DoorClosed	LEFT/RIGHT				
	HallCall	numArrow				
	AtFloorUp	numFloor				
	AtFloorStop	numFloor				
	AtFloorDown	numFloor				
	CarCall	numFloor				

continued on next page

Table A.1: (Continued)

	Name	Replication	CodeSize	FieldSize	Period	Payload
Outputs	DesiredDwell	1			100uS	8bytes
	DesiredFloor	1			100uS	2bytes
	PeakMode	1			100uS	1byte
Feature	LanternControl	UP/DOWN	731	51		
Inputs	DesiredFloor	1				
	CarLantern	OtherDirection				
	AtFloorStop	numFloor				
	DoorClosed	LEFT/RIGHT				
Outputs	CarLantern	SameDirection			500uS	1byte
Feature	HallButtonControl	numArrow	907	43		
Inputs	DesiredFloor	1				
	DoorOpened	LEFT/RIGHT				
	DoorClosed	LEFT/RIGHT				
	AtFloorStop	numFloor				
	ModeKey	1				
	FireAlarm	1				
	HallCallRAW	SameArrow				
Outputs	HallLight	SameArrow			500uS	2bytes
	HallCall	SameArrow			500uS	2bytes
Feature	CarButtonControl	numFloor	938	49		
Inputs	Weight	1				
	AtFloorStop	SameFloor				
	DoorClosed	LEFT/RIGHT				
	DoorOpened	LEFT/RIGHT				
	FireAlarm	1				
	ModeKey	1				
	CarCallRAW	SameFloor				
Outputs	CarLight	SameFloor			500uS	2bytes
	CarCall	SameFloor			500uS	2bytes
Feature	DriveControl	1	2549	204		
Inputs	HoistwayLimit1	UP/DOWN				
	DesiredFloor	1				
	AtFloorUp	numFloor				
	AtFloorStop	numFloor				
	AtFloorDown	numFloor				
	DoorClosed	LEFT/RIGHT				
	DoorMotor	LEFT/RIGHT				
Outputs	DriveSpeed	1			100us	2bytes
	DriveState	1			50us	2bytes

Table A.2: Team A — Missing modules measurements

	Name	Replication	CodeSize	FieldSize	Period	Payload
Sensor	CarCallButton	numFloor				
Outputs	CarCallRAW	SameFloor			100mS	1byte
Sensor	HallCallButton	numArrow				
Outputs	HallCallRAW	SameArrow			100mS	1byte
Sensor	FireAlarm	1				
Outputs	FireAlarmRAW	1			500uS	1byte
Adapter	FireAlarmAdapter	1	650	50		
Inputs	FireAlarmRAW	1				
Outputs	FireAlarm	1			500uS	1byte
Actuator	CarPositionIndicator	1				
Inputs	CarPositionIndicatorRAW	1				
Adapter	CarPositionIndicatorAdapter	1	800	30		
Inputs	CarPositionIndicator	1				
Outputs	CarPositionIndicatorRAW	1			200uS	1byte
Actuator	CarLantern	UP/DOWN				
Inputs	CarLanternRAW	SameDirection				
Adapter	CarLanternAdapter	UP/DOWN	800	30		
Inputs	CarLantern	SameDirection				
Outputs	CarLanternRAW	SameDirection			500uS	1byte
Actuator	CarLight	numFloor				
Inputs	CarLightRAW	SameFloor				
Adapter	CarLightAdapter	numFloor	800	30		
Inputs	CarLight	SameFloor				
Outputs	CarLightRAW	SameFloor			500uS	2bytes
Actuator	HallLantern	numArrow				
Inputs	HallLanternRAW	SameArrow				
Adapter	HallLanternAdapter	numArrow	800	300		
Inputs	HallLantern	SameArrow				
Outputs	HallLanternRAW	SameArrow			100uS	1byte
Actuator	HallLight	numArrow				
Inputs	HallLightRAW	SameArrow				
Adapter	HallLightAdapter	numArrow	800	300		
Inputs	HallLight	SameArrow				
Outputs	HallLightRAW	SameArrow			500uS	2bytes
Feature	HallLanternControl	numArrow	653	36		
Inputs	DesiredFloor	1				
	AtFloorDown	FloorOfThisArrow				
	AtFloorUp	FloorOfThisArrow				
	DoorClosed	LEFT/RIGHT				
Outputs	HallLantern	SameArrow			100uS	1byte

Table A.3: Team A — Modified modules measurements

	Name	Replication	CodeSize	FieldSize	Period	Payload
Feature	DoorControl_MOD1	LEFT/RIGHT	3741	230		
Inputs	DesiredFloor	1				
	DesiredDwell	1				
	DoorReversal	LEFT/RIGHT				
	DoorOpened	LEFT/RIGHT				
	DoorClosed	LEFT/RIGHT				
	HallCall	numArrow				
	CarCall	numFloor				
	DriveState	1				
	DoorOpenButton	1				
	DoorCloseButton	1				
	AtFloorStop	numFloor				
	ModeKey	1				
	FireAlarm	1				
Outputs	DoorControlState_MOD1	SameSide			500uS	1byte
Feature	DoorControl_FF_MM_MOD1	LEFT/RIGHT	3589	208		
Inputs	DesiredFloor	1				
	DesiredDwell	1				
	DoorReversal	LEFT/RIGHT				
	DoorOpened	LEFT/RIGHT				
	DoorClosed	LEFT/RIGHT				
	HallCall	numArrow				
	CarCall	numFloor				
	DriveState	1				
	DoorOpenButton	1				
	DoorCloseButton	1				
	CurrentMode_MOD1	1				
	CurrentFloor1_MOD1	1				
Outputs	DoorControlState_MOD1	SameSide			500uS	1byte
Feature	DoorControl_FF_MOD1	LEFT/RIGHT	3640	213		
Inputs	DesiredFloor	1				
	DesiredDwell	1				
	DoorReversal	LEFT/RIGHT				
	DoorOpened	LEFT/RIGHT				
	DoorClosed	LEFT/RIGHT				
	HallCall	numArrow				
	CarCall	numFloor				
	DriveState	1				
	DoorOpenButton	1				
	DoorCloseButton	1				
	CurrentFloor1_MOD1	1				

continued on next page

Table A.3: (Continued)

	Name	Replication	CodeSize	FieldSize	Period	Payload
	ModeKey	1				
	FireAlarm	1				
Outputs	DoorControlState_MOD1	SameSide			500uS	1byte
Feature	DoorControl_MM_MOD1	LEFT/RIGHT	3690	225		
Inputs	DesiredFloor	1				
	DesiredDwell	1				
	DoorReversal	LEFT/RIGHT				
	DoorOpened	LEFT/RIGHT				
	DoorClosed	LEFT/RIGHT				
	HallCall	numArrow				
	CarCall	numFloor				
	DriveState	1				
	DoorOpenButton	1				
	DoorCloseButton	1				
	AtFloorStop	numFloor				
	CurrentMode_MOD1	1				
Outputs	DoorControlState_MOD1	SameSide			500uS	1byte
Adapter	BehaviorManager_MOD1	LEFT/RIGHT	360	38		
Inputs	DoorControlState_MOD1	SameSide				
Outputs	DoorMotor	SameSide			100uS	1byte
Adapter	ModeManager	1	173	19		
Inputs	ModeKey	1				
	FireAlarm	1				
Outputs	CurrentMode_MOD1	1			500uS	1byte
Adapter	FloorDetector1_MOD1	1	207	19		
Inputs	AtFloorStop	numFloor				
Outputs	CurrentFloor1_MOD1	1			100uS	1byte
Adapter	FloorDetector2_MOD1	1	516	71		
Inputs	AtFloorUp	numFloor				
	AtFloorStop	numFloor				
	AtFloorDown	numFloor				
Outputs	CurrentFloor2_MOD1	1			100uS	1byte
Adapter	CarPositionControl_MOD1	1	310	40		
Inputs	DesiredFloor	1				
	CurrentFloor2_MOD1	1				
Outputs	CarPositionIndicator	1			200us	1byte
Feature	Dispatcher_FF_MOD1	1	7294	413		
Inputs	Weight	1				
	TimeOfDay	1				
	ModeKey	1				
	FireAlarm	1				

continued on next page

Table A.3: (Continued)

	Name	Replication	CodeSize	FieldSize	Period	Payload
	DoorClosed	LEFT/RIGHT				
	HallCall	numArrow				
	CurrentFloor2_MOD1	1				
	CarCall	numFloor				
Outputs	DesiredDwell	1			100uS	8bytes
	DesiredFloor	1			100uS	2bytes
	PeakMode	1			100uS	1byte
Feature	LanternControl_MOD1	UP/DOWN	653	36		
Inputs	DesiredFloor	1				
	CarLantern	OtherDirection				
	CurrentFloor1_MOD1	1				
	DoorClosed	LEFT/RIGHT				
Outputs	CarLantern	SameDirection			500uS	1byte
Feature	HallButtonControl_MOD1	numArrow	834	46		
Inputs	DesiredFloor	1				
	DoorOpened	LEFT/RIGHT				
	DoorClosed	LEFT/RIGHT				
	CurrentFloor1_MOD1	1				
	CurrentMode_MOD1	1				
	HallCallRAW	SameArrow				
Outputs	HallLight	SameArrow			500uS	2bytes
	HallCall	SameArrow			500uS	2bytes
Feature	CarButtonControl_MOD1	numFloor	816	48		
Inputs	Weight	1				
	CarCallRAW	SameFloor				
	DoorOpened	LEFT/RIGHT				
	DoorClosed	LEFT/RIGHT				
	AtFloorStop	SameFloor				
	CurrentMode_MOD1	1				
Outputs	CarLight	SameFloor			500uS	2bytes
	CarCall	SameFloor			500uS	2bytes

Table A.4: Team 1 — Original system measurements

	Name	Replication	CodeSize	FieldSize	Period	Payload
Adapter	DoorOpenedAdapterTeam1	LEFT/RIGHT	356	18		
Inputs	DoorOpenedRAW	SameSide				
Outputs	DoorOpened	SameSide			100mS	1byte
Adapter	DoorClosedAdapterTeam1	LEFT/RIGHT	354	18		
Inputs	DoorClosedRAW	SameSide				

continued on next page

Table A.4: (Continued)

	Name	Replication	CodeSize	FieldSize	Period	Payload
Outputs	DoorClosed	SameSide			100mS	1byte
Adapter	AtFloorUpAdapterTeam1	numFloor	428	44		
Inputs	AtFloorUpRAW	SameFloor				
Outputs	AtFloorUp	SameFloor			100uS	2bytes
Adapter	AtFloorStopAdapterTeam1	numFloor	428	44		
Inputs	AtFloorStopRAW	SameFloor				
Outputs	AtFloorStop	SameFloor			100uS	2bytes
Adapter	AtFloorDownAdapterTeam1	numFloor	428	44		
Inputs	AtFloorDownRAW	SameFloor				
Outputs	AtFloorDown	SameFloor			100uS	2bytes
Adapter	HoistwayLimit1AdapterTeam1	UP/DOWN	325	31		
Inputs	HoistwayLimit1RAW	SameDirection				
Outputs	HoistwayLimit1	SameDirection			10uS	1byte
Adapter	HoistwayLimit2AdapterTeam1	UP/DOWN	325	31		
Inputs	HoistwayLimit2RAW	SameDirection				
Outputs	HoistwayLimit2	SameDirection			10uS	1byte
Sensor	FireAlarm	numFloor				
Outputs	FireAlarmRAW	SameFloor			100mS	1byte
Adapter	FireAlarmAdapter	numFloor	1200	150		
Inputs	FireAlarmRAW	SameFloor				
Outputs	FireAlarm	SameFloor			10mS	1byte
Adapter	FireAlarmAdapter2	1	1400	250		
Inputs	FireAlarmRAW	numFloor				
Outputs	FireAlarm	1			500uS	1byte
Adapter	DoorMotorAdapterTeam1	LEFT/RIGHT	1628	58		
Inputs	DoorMotor	SameSide				
Outputs	DoorMotorRAW	SameSide			100uS	1byte
Adapter	CarPositionIndicatorAdapterTeam1	1	176	4		
Inputs	CarPositionIndicator	1				
Outputs	CarPositionIndicatorRAW	1			200mS	1byte
Adapter	CarLanternAdapterTeam1	UP/DOWN	378	2		
Inputs	CarLantern	SameDirection				
Outputs	CarLanternRAW	SameDirection			500mS	1byte
Adapter	CarLightAdapterTeam1	numFloor	228	6		
Inputs	CarLight	SameFloor				
Outputs	CarLightRAW	SameFloor			500mS	2bytes
Adapter	HallLightAdapterTeam1	numArrow	395	6		
Inputs	HallLight	SameArrow				
Outputs	HallLightRAW	SameArrow			500mS	2bytes
Adapter	DriveAdapterTeam1	1	2327	55		
Inputs	DriveSpeed	1				

continued on next page

Table A.4: (Continued)

	Name	Replication	CodeSize	FieldSize	Period	Payload
Outputs	DriveSpeedRAW	1			100us	2bytes
Actuator	EmergencyBrake	1				
Inputs	EmergencyBrakeRAW	1				
Adapter	EmergencyBrakeAdapter	1	800	1600		
Inputs	EmergencyBrake	1				
Outputs	EmergencyBrakeRAW	1			100mS	1byte
Feature	DoorControlTeam1	LEFT/RIGHT	1901	120		
Inputs	DesiredFloor	1				
	DesiredDwell	1				
	DoorReversal	LEFT/RIGHT				
	DoorOpened	LEFT/RIGHT				
	DoorClosed	LEFT/RIGHT				
	DriveSpeed	1				
	AtFloorStop	numFloor				
	ModeKey	1				
	FireAlarm	numFloor				
	DoorOpenButton	1				
	DoorCloseButton	1				
Outputs	DoorMotor	SameSide			5mS	1byte
	DoorMotor	OtherSide			5mS	1byte
Feature	CarPositionControlTeam1	1	373	68		
Inputs	ModeKey	1				
	AtFloorUp	numFloor				
	AtFloorStop	numFloor				
	AtFloorDown	numFloor				
Outputs	CarPositionIndicator	1			200us	1byte
Feature	LanternControlTeam1	UP/DOWN	380	34		
Inputs	ModeKey	1				
	DesiredFloor	1				
	DoorClosed	LEFT/RIGHT				
Outputs	CarLantern	SameDirection			500uS	1byte
Feature	CarButtonControlTeam1	numFloor	589	58		
Inputs	CarCallRAW	SameFloor				
	CarCallRAW	OtherFloors				
	AtFloorStop	SameFloor				
	ModeKey	1				
	FireAlarm	numFloor				
Outputs	CarLight	SameFloor			500uS	2bytes
	CarCall	SameFloor			250uS	2bytes
Feature	HallButtonControlTeam1	numArrow	535	47		
Inputs	DesiredFloor	1				

continued on next page

Table A.4: (Continued)

	Name	Replication	CodeSize	FieldSize	Period	Payload
	ModeKey	1				
	AtFloorStop	FloorOfThisArrow				
	AtFloorUp	FloorOfThisArrow				
	AtFloorDown	FloorOfThisArrow				
	HallCallRAW	SameArrow				
Outputs	HallLight	SameArrow			500uS	2bytes
	HallCall	SameArrow			500uS	2bytes
Feature	DispatcherTeam1	1	1853	201		
Inputs	TimeOfDay	1				
	ModeKey	1				
	FireAlarm	numFloor				
	HallCall	numArrow				
	AtFloorUp	numFloor				
	AtFloorStop	numFloor				
	AtFloorDown	numFloor				
	CarCall	numFloor				
Outputs	DesiredDwell	1			100uS	8bytes
	DesiredFloor	1			100uS	2bytes
Feature	DriveControlTeam1	1	2124	136		
Inputs	HoistwayLimit1	UP/DOWN				
	HoistwayLimit2	UP/DOWN				
	DesiredFloor	1				
	EmergencyBrake	1				
	AtFloorUp	numFloor				
	AtFloorStop	numFloor				
	AtFloorDown	numFloor				
	DoorClosed	LEFT/RIGHT				
	DoorMotor	LEFT/RIGHT				
	ModeKey	1				
	FireAlarm	numFloor				
	CarCall	numFloor				
Outputs	DriveSpeed	1			150ms	2bytes
	DriveState	1			150ms	2bytes
Adapter	SafetyDriveControl	1	735	49		
Inputs	DoorOpened	LEFT/RIGHT				
	DoorClosed	LEFT/RIGHT				
	DriveSpeed	1				
Outputs	DriveSpeed	1			120ms	2bytes
	EmergencyBrake	1			120ms	1byte
Adapter	SafetyDoorMotorControl	LEFT/RIGHT	1118	81		
Inputs	DriveSpeed	1				

continued on next page

Table A.4: (Continued)

	Name	Replication	CodeSize	FieldSize	Period	Payload
	DoorReversal	SameSide				
	DoorOpened	SameSide				
	DoorClosed	SameSide				
	AtFloorStop	numFloor				
Outputs	EmergencyBrake	1			500ms	1byte
	DoorMotor	SameSide			500ms	1byte

Table A.5: Team 1 — Modified adapter measurements

	Name	Replication	CodeSize	FieldSize	Period	Payload
Feature	DoorControl_FF_Team1	LEFT/RIGHT	1773	128		
Inputs	DesiredFloor	1				
	DesiredDwell	1				
	DoorReversal	LEFT/RIGHT				
	DoorOpened	LEFT/RIGHT				
	DoorClosed	LEFT/RIGHT				
	HallCall	numArrow				
	CarCall	numFloor				
	DriveState	1				
	DoorOpenButton	1				
	DoorCloseButton	1				
	CurrentFloor1_MOD1	1				
	CurrentMode_MOD1	1				
Outputs	DoorMotor	SameSide			500uS	1byte
Adapter	CarPositionControl_Team1	1	270	55		
Inputs	DesiredFloor	1				
	CurrentFloor2_MOD1	1				
Outputs	CarPositionIndicator	1			200us	1byte
Feature	Dispatcher_FF_Team1	1	1694	196		
Inputs	Weight	1				
	TimeOfDay	1				
	DoorClosed	LEFT/RIGHT				
	HallCall	numArrow				
	CurrentFloor2_MOD1	1				
	CarCall	numFloor				
Outputs	DesiredDwell	1			100uS	8bytes
	DesiredFloor	1			100uS	2bytes
	PeakMode	1			100uS	1byte
Feature	HallButtonControl_Team1	numArrow	468	70		
Inputs	DesiredFloor	1				
	DoorOpened	LEFT/RIGHT				

continued on next page

Table A.5: (Continued)

	Name	Replication	CodeSize	FieldSize	Period	Payload
	DoorClosed	LEFT/RIGHT				
	CurrentFloor2_MOD1	1				
	CurrentMode_MOD1	1				
	HallCallRAW	SameArrow				
Outputs	HallLight	SameArrow			500uS	2bytes
	HallCall	SameArrow			500uS	2bytes
Feature	CarButtonControl_Team1	numFloor	516	82		
Inputs	Weight	1				
	CarCallRAW	SameFloor				
	DoorOpened	LEFT/RIGHT				
	DoorClosed	LEFT/RIGHT				
	CurrentFloor1_MOD1	1				
	CurrentMode_MOD1	1				
Outputs	CarLight	SameFloor			500uS	2bytes
	CarCall	SameFloor			500uS	2bytes

Appendix B

Navigation System Description

The following XML markup describes the automotive navigation system used in Chapters 5 and 6 to develop the algorithmic framework and tune the feature selection and adapter selection algorithms

B.1 XML Description

```
<?xml version="1.0" encoding="UTF-8"?>
<systemDescription>
  <GlobalProperties>
    <Comment>The sample navigation scenario</Comment>
    <Scenario>Bosch Navigation</Scenario>
    <ScenarioVersion>3</ScenarioVersion>
    <FileVersion>7</FileVersion>
    <ResourceSize Type="PE" NumElements="2"/>
    <ResourceSize Type="DE" NumElements="1"/>
    <FileID>1</FileID>
    <FeatureSelector Type="red"/>
```

```
  <AdapterSelector Type="red"/>
  <AdapterAllocator Type="TRANS" Param="1ARNSSBARNI"/>
</GlobalProperties>

<Sensor SensorId="0" Name="UserInterface" Active="true">
  <OutputDataElement DataId="313"/>
</Sensor>
<Sensor SensorId="1" Name="EngineSensor" Active="true">
  <OutputDataElement DataId="314"/>
  <OutputDataElement DataId="315"/>
  <OutputDataElement DataId="316"/>
</Sensor>
<Sensor SensorId="2" Name="MM1" Active="true">
  <OutputDataElement DataId="300"/>
  <OutputDataElement DataId="317"/>
</Sensor>
<Sensor SensorId="3" Name="SBox" Active="true">
  <OutputDataElement DataId="300"/>
</Sensor>
<Sensor SensorId="4" Name="VDC" Active="true">
  <OutputDataElement DataId="318"/>
</Sensor>
<Sensor SensorId="5" Name="LWS" Active="true">
  <OutputDataElement DataId="319"/>
</Sensor>
<Sensor SensorId="6" Name="GPS1" Active="true">
  <OutputDataElement DataId="325"/>
</Sensor>
<Sensor SensorId="7" Name="Compass" Active="true">
  <OutputDataElement DataId="302"/>
</Sensor>
<Sensor SensorId="8" Name="MapDVD" Active="true">
  <OutputDataElement DataId="324"/>
</Sensor>

<Actuator ActuatorId="10" Name="TurnSignalIndicator" Active="true">
  <InputDataElement DataId="322"/>
</Actuator>
<Actuator ActuatorId="11" Name="Speaker" Active="true">
  <InputDataElement DataId="321"/>
</Actuator>
<Actuator ActuatorId="12" Name="Display" Active="true">
  <InputDataElement DataId="323"/>
</Actuator>
<DataElement DataId="300" Name="YawRate" isLocal="false">
```

```
    <requirements>
      <requirement rId="0">111</requirement>
    </requirements>
  </DataElement>
  <DataElement DataId="301" Name="GroundSpeed" isLocal="false">
    <requirements>
      <requirement rId="0">80</requirement>
    </requirements>
  </DataElement>
  <DataElement DataId="302" Name="CurrentDirection" isLocal="false">
    <requirements>
      <requirement rId="0">104</requirement>
    </requirements>
  </DataElement>
  <DataElement DataId="303" Name="MapData" isLocal="false">
    <requirements>
      <requirement rId="0">95</requirement>
    </requirements>
  </DataElement>
  <DataElement DataId="304" Name="CurrentLocation" isLocal="false">
    <requirements>
      <requirement rId="0">96</requirement>
    </requirements>
  </DataElement>

  <DataElement DataId="305" Name="ErrorEstimate" isLocal="false">
    <requirements>
      <requirement rId="0">93</requirement>
    </requirements>
  </DataElement>

  <DataElement DataId="306" Name="Path" isLocal="false">
    <requirements>
      <requirement rId="0">96</requirement>
    </requirements>
  </DataElement>
  <DataElement DataId="307" Name="TurnInfo" isLocal="false">
    <requirements>
      <requirement rId="0">84</requirement>
    </requirements>
  </DataElement>

  <DataElement DataId="308" Name="UpdateMapRequest" isLocal="false">
    <requirements>
      <requirement rId="0">99</requirement>
```

```
    </requirements>
  </DataElement>
  <DataElement DataId="309" Name="TurnInfoText" isLocal="true">
    <requirements>
      <requirement rId="0">114</requirement>
    </requirements>
  </DataElement>
  <DataElement DataId="310" Name="TurnInfoAudio" isLocal="true">
    <requirements>
      <requirement rId="0">114</requirement>
    </requirements>
  </DataElement>
  <DataElement DataId="311" Name="MapAsImage" isLocal="true">
    <requirements>
      <requirement rId="0">97</requirement>
    </requirements>
  </DataElement>
  <DataElement DataId="312" Name="MapAsStroke" isLocal="true">
    <requirements>
      <requirement rId="0">104</requirement>
    </requirements>
  </DataElement>
  <DataElement DataId="313" Name="Destination" isLocal="false">
    <requirements>
      <requirement rId="0">117</requirement>
    </requirements>
  </DataElement>
  <DataElement DataId="314" Name="ThrottleAngle" isLocal="false">
    <requirements>
      <requirement rId="0">103</requirement>
    </requirements>
  </DataElement>
  <DataElement DataId="315" Name="EngineSpeed" isLocal="false">
    <requirements>
      <requirement rId="0">114</requirement>
    </requirements>
  </DataElement>
  <DataElement DataId="316" Name="EngineTemp" isLocal="false">
    <requirements>
      <requirement rId="0">100</requirement>
    </requirements>
  </DataElement>
  <DataElement DataId="317" Name="Acceleration" isLocal="false">
    <requirements>
      <requirement rId="0">106</requirement>
```

```
    </requirements>
  </DataElement>

  <DataElement DataId="318" Name="AverageWheelSpeed" isLocal="false">
    <requirements>
      <requirement rId="0">79</requirement>
    </requirements>
  </DataElement>
  <DataElement DataId="319" Name="SteeringAngle" isLocal="false">
    <requirements>
      <requirement rId="0">83</requirement>
    </requirements>
  </DataElement>
  <DataElement DataId="320" Name="ClockTime" isLocal="false">
    <requirements>
      <requirement rId="0">80</requirement>
    </requirements>
  </DataElement>
  <DataElement DataId="321" Name="TurnInfoSound" isLocal="true">
    <requirements>
      <requirement rId="0">94</requirement>
    </requirements>
  </DataElement>
  <DataElement DataId="322" Name="TurnDirection" isLocal="false">
    <requirements>
      <requirement rId="0">89</requirement>
    </requirements>
  </DataElement>
  <DataElement DataId="323" Name="DisplayMap" isLocal="false">
    <requirements>
      <requirement rId="0">86</requirement>
    </requirements>
  </DataElement>
  <DataElement DataId="324" Name="MapDVD" isLocal="false">
    <requirements>
      <requirement rId="0">95</requirement>
    </requirements>
  </DataElement>
  <DataElement DataId="325" Name="GPSPosition" isLocal="false">
    <requirements>
      <requirement rId="0">95</requirement>
    </requirements>
  </DataElement>

  <Adapter AdapterId="100" Name="ConvertWheelSpeed">
```

```
    <requirements>
      <requirement rId="0">235</requirement>
      <requirement rId="1">50</requirement>
    </requirements>
    <InputDataElement DataId="318"/>
    <OutputDataElement DataId="301"/>
  </Adapter>
  <Adapter AdapterId="101" Name="YawGenerator">
    <requirements>
      <requirement rId="0">200</requirement>
      <requirement rId="1">60</requirement>
    </requirements>
    <InputDataElement DataId="319"/>
    <OutputDataElement DataId="300"/>
  </Adapter>
  <Adapter AdapterId="102" Name="SpeedIntegrator2">
    <requirements>
      <requirement rId="0">190</requirement>
      <requirement rId="1">40</requirement>
    </requirements>
    <InputDataElement DataId="314"/>
    <OutputDataElement DataId="301"/>
  </Adapter>
  <Adapter AdapterId="103" Name="SpeedIntegrator1">
    <requirements>
      <requirement rId="0">400</requirement>
      <requirement rId="1">200</requirement>
    </requirements>
    <InputDataElement DataId="317"/>
    <OutputDataElement DataId="301"/>
  </Adapter>
  <Adapter AdapterId="104" Name="DirectionIntegrator">
    <requirements>
      <requirement rId="0">280</requirement>
      <requirement rId="1">70</requirement>
    </requirements>
    <InputDataElement DataId="300"/>
    <OutputDataElement DataId="302"/>
  </Adapter>
  <Adapter AdapterId="109" Name="RenderMap">
    <requirements>
      <requirement rId="0">2000</requirement>
      <requirement rId="1">200</requirement>
    </requirements>
    <InputDataElement DataId="303"/>
```

```
    <OutputDataElement DataId="311"/>
  </Adapter>
  <Adapter AdapterId="110" Name="RenderMap2">
    <requirements>
      <requirement rId="0">600</requirement>
      <requirement rId="1">200</requirement>
    </requirements>
    <InputDataElement DataId="303"/>
    <OutputDataElement DataId="312"/>
  </Adapter>

  <Adapter AdapterId="111" Name="PathPlanner">
    <requirements>
      <requirement rId="0">2000</requirement>
      <requirement rId="1">700</requirement>
    </requirements>
    <InputDataElement DataId="304"/>
    <InputDataElement DataId="313"/>
    <InputDataElement DataId="303"/>
    <OutputDataElement DataId="306"/>
  </Adapter>
  <Adapter AdapterId="112" Name="LocationSentry">
    <requirements>
      <requirement rId="0">200</requirement>
      <requirement rId="1">20</requirement>
    </requirements>
    <InputDataElement DataId="304"/>
    <OutputDataElement DataId="308"/>
  </Adapter>
  <Adapter AdapterId="113" Name="TurnInfoGenerator1">
    <requirements>
      <requirement rId="0">2000</requirement>
      <requirement rId="1">150</requirement>
    </requirements>
    <InputDataElement DataId="304"/>
    <InputDataElement DataId="306"/>
    <OutputDataElement DataId="307"/>
  </Adapter>
  <Adapter AdapterId="114" Name="TurnInfoGenerator2">
    <requirements>
      <requirement rId="0">2200</requirement>
      <requirement rId="1">180</requirement>
    </requirements>
    <InputDataElement DataId="304"/>
    <InputDataElement DataId="306"/>
```

```
    <OutputDataElement DataId="307"/>
  </Adapter>
  <Adapter AdapterId="115" Name="TurnInfoGenerator3">
    <requirements>
      <requirement rId="0">1800</requirement>
      <requirement rId="1">400</requirement>
    </requirements>
    <InputDataElement DataId="304"/>
    <InputDataElement DataId="306"/>
    <OutputDataElement DataId="307"/>
  </Adapter>

  <Adapter AdapterId="116" Name="TurnInfoGenerator4">
    <requirements>
      <requirement rId="0">2300</requirement>
      <requirement rId="1">500</requirement>
    </requirements>
    <InputDataElement DataId="304"/>
    <InputDataElement DataId="306"/>
    <OutputDataElement DataId="307"/>
  </Adapter>
  <Adapter AdapterId="117" Name="TurnInfoConverter">
    <requirements>
      <requirement rId="0">500</requirement>
      <requirement rId="1">100</requirement>
    </requirements>
    <InputDataElement DataId="307"/>
    <OutputDataElement DataId="309"/>
  </Adapter>
  <Adapter AdapterId="118" Name="SpeechSynthesis">
    <requirements>
      <requirement rId="0">500</requirement>
      <requirement rId="1">900</requirement>
    </requirements>
    <InputDataElement DataId="309"/>
    <OutputDataElement DataId="310"/>
  </Adapter>
  <Adapter AdapterId="119" Name="SimpleSpeechSynthesis">
    <requirements>
      <requirement rId="0">1200</requirement>
      <requirement rId="1">800</requirement>
    </requirements>
    <InputDataElement DataId="307"/>
    <OutputDataElement DataId="310"/>
  </Adapter>
```

```
<Adapter AdapterId="120" Name="TurnInfoConverter2">
  <requirements>
    <requirement rId="0">1400</requirement>
    <requirement rId="1">400</requirement>
  </requirements>
  <InputDataElement DataId="307"/>
  <OutputDataElement DataId="310"/>
</Adapter>
<Adapter AdapterId="121" Name="MapDataServer">
  <requirements>
    <requirement rId="0">4000</requirement>
    <requirement rId="1">2000</requirement>
  </requirements>
  <InputDataElement DataId="308"/>
  <InputDataElement DataId="324"/>
  <OutputDataElement DataId="303"/>
</Adapter>

<Hardware>
  <ProcessingElement Id="1" Name="EngineControlUnit">
    <Sensor sensorId="0"/>
    <Sensor sensorId="1"/>
    <Sensor sensorId="2"/>
    <Sensor sensorId="3"/>
    <Sensor sensorId="4"/>
    <Sensor sensorId="5"/>
    <Sensor sensorId="6"/>
    <resources>
      <resource rId="0">7000</resource>
      <resource rId="1">1100</resource>
    </resources>
  </ProcessingElement>
  <ProcessingElement Id="2" Name="UserInterface">
    <Actuator actuatorId="10"/>
    <Actuator actuatorId="11"/>
    <Actuator actuatorId="12"/>
    <Sensor sensorId="7"/>
    <resources>
      <resource rId="0">3000</resource>
      <resource rId="1">1200</resource>
    </resources>
  </ProcessingElement>
  <ProcessingElement Id="3" Name="MapServer">
    <Sensor sensorId="8"/>
    <resources>
```

```
      <resource rId="0">7000</resource>
      <resource rId="1">2500</resource>
    </resources>
  </ProcessingElement>
  <Network netId="0">
    <resources>
      <resource rId="0">275</resource>
    </resources>
  </Network>
</Hardware>

<Feature FeatureId="105" Name="SimpleDeadReckoner"
 FeatureClass="502" Utility="68">
  <requirements>
    <requirement rId="0">120</requirement>
    <requirement rId="1">40</requirement>
  </requirements>
  <InputDataElement DataId="301"/>
  <InputDataElement DataId="302"/>
  <OutputDataElement DataId="304"/>
</Feature>
<Feature FeatureId="106" Name="GoodDeadReckoner"
 FeatureClass="502" Utility="82">
  <requirements>
    <requirement rId="0">250</requirement>
    <requirement rId="1">80</requirement>
  </requirements>
  <InputDataElement DataId="301"/>
  <InputDataElement DataId="302"/>
  <OutputDataElement DataId="304"/>
  <OutputDataElement DataId="305"/>
</Feature>
<Feature FeatureId="107" Name="BetterDeadReckoner"
 FeatureClass="502" Utility="91">
  <requirements>
    <requirement rId="0">1250</requirement>
    <requirement rId="1">120</requirement>
  </requirements>
  <InputDataElement DataId="301"/>
  <InputDataElement DataId="302"/>
  <OutputDataElement DataId="304"/>
  <OutputDataElement DataId="305"/>
</Feature>
<Feature FeatureId="108" Name="BestDeadReckoner"
 FeatureClass="502" Utility="96">
```

```
    <requirements>
      <requirement rId="0">2200</requirement>
      <requirement rId="1">800</requirement>
    </requirements>
    <InputDataElement DataId="301"/>
    <InputDataElement DataId="302"/>
    <InputDataElement DataId="303"/>
    <OutputDataElement DataId="304"/>
    <OutputDataElement DataId="305"/>
  </Feature>
  <Feature FeatureId="199" Name="GPSNullReckoner"
   FeatureClass="502" Utility="100">
    <requirements>
      <requirement rId="0">0</requirement>
      <requirement rId="1">0</requirement>
    </requirements>
    <InputDataElement DataId="325"/>
    <OutputDataElement DataId="304"/>
    <OutputDataElement DataId="305"/>
  </Feature>

  <Feature FeatureId="200" Name="Turn1"
   FeatureClass="500" Utility="12">
    <requirements>
      <requirement rId="0">200</requirement>
      <requirement rId="1">10</requirement>
    </requirements>
    <InputDataElement DataId="307"/>
    <OutputDataElement DataId="322"/>
  </Feature>
  <Feature FeatureId="201" Name="Turn2"
   FeatureClass="500" Utility="19">
    <requirements>
      <requirement rId="0">400</requirement>
      <requirement rId="1">40</requirement>
    </requirements>
    <InputDataElement DataId="310"/>
    <OutputDataElement DataId="321"/>
  </Feature>
  <Feature FeatureId="203" Name="Map1"
   FeatureClass="501" Utility="60">
    <requirements>
      <requirement rId="0">1200</requirement>
      <requirement rId="1">100</requirement>
    </requirements>
```

```
    <InputDataElement DataId="311"/>
    <InputDataElement DataId="304"/>
    <OutputDataElement DataId="323"/>
  </Feature>
  <Feature FeatureId="204" Name="Map2"
   FeatureClass="501" Utility="81">
    <requirements>
      <requirement rId="0">1400</requirement>
      <requirement rId="1">120</requirement>
    </requirements>
    <InputDataElement DataId="311"/>
    <InputDataElement DataId="304"/>
    <OutputDataElement DataId="323"/>
  </Feature>
  <Feature FeatureId="205" Name="Map3"
   FeatureClass="501" Utility="95">
    <requirements>
      <requirement rId="0">1800</requirement>
      <requirement rId="1">140</requirement>
    </requirements>
    <InputDataElement DataId="311"/>
    <InputDataElement DataId="304"/>
    <InputDataElement DataId="306"/>
    <OutputDataElement DataId="323"/>
  </Feature>

  <Feature FeatureId="206" Name="Map4"
   FeatureClass="501" Utility="100">
    <requirements>
      <requirement rId="0">2400</requirement>
      <requirement rId="1">210</requirement>
    </requirements>
    <InputDataElement DataId="311"/>
    <InputDataElement DataId="304"/>
    <InputDataElement DataId="305"/>
    <InputDataElement DataId="306"/>
    <OutputDataElement DataId="323"/>
  </Feature>
  <Feature FeatureId="207" Name="Map5"
   FeatureClass="501" Utility="47">
    <requirements>
      <requirement rId="0">1300</requirement>
      <requirement rId="1">80</requirement>
    </requirements>
    <InputDataElement DataId="312"/>
```

```
    <InputDataElement DataId="304"/>
    <OutputDataElement DataId="323"/>
  </Feature>
  <Feature FeatureId="208" Name="Map6"
   FeatureClass="501" Utility="71">
    <requirements>
      <requirement rId="0">1500</requirement>
      <requirement rId="1">90</requirement>
    </requirements>
    <InputDataElement DataId="312"/>
    <InputDataElement DataId="304"/>
    <OutputDataElement DataId="323"/>
  </Feature>
  <Feature FeatureId="209" Name="Map7"
   FeatureClass="501" Utility="89">
    <requirements>
      <requirement rId="0">1900</requirement>
      <requirement rId="1">100</requirement>
    </requirements>
    <InputDataElement DataId="312"/>
    <InputDataElement DataId="304"/>
    <InputDataElement DataId="306"/>
    <OutputDataElement DataId="323"/>
  </Feature>
  <Feature FeatureId="210" Name="Map8"
   FeatureClass="501" Utility="98">
    <requirements>
      <requirement rId="0">2300</requirement>
      <requirement rId="1">200</requirement>
    </requirements>
    <InputDataElement DataId="312"/>
    <InputDataElement DataId="304"/>
    <InputDataElement DataId="305"/>
    <InputDataElement DataId="306"/>
    <OutputDataElement DataId="323"/>
  </Feature>
  <FeatureClass ClassId="500" isCritical="false"/>
  <FeatureClass ClassId="501" isCritical="true"/>
  <FeatureClass ClassId="502" isCritical="false"/>
</systemDescription>
```

Appendix C

ASCII Chart

Because every technical book should have one

Hex		0_	1_	2_	3_	4_	5_	6_	7_
	Dec	0+	16+	32+	48+	64+	80+	96+	112+
_0	0	NUL	DLE	SPACE	0	@	P	‘	p
_1	1	SOH	DC1	!	1	A	Q	a	q
_2	2	STX	DC2	”	2	B	R	b	r
_3	3	ETX	DC3	#	3	C	S	c	s
_4	4	EOT	DC4	$	4	D	T	d	t
_5	5	ENQ	NAK	%	5	E	U	e	u
_6	6	ACK	SYN	&	6	F	V	f	v
_7	7	BEL	ETB	’	7	G	W	g	w
_8	8	BS	CAN	(	8	H	X	h	x
_9	9	HT	EM	)	9	I	Y	i	y
_A	10	LF	SUB	*	:	J	Z	j	z
_B	11	VT	ESC	+	;	K	[	k	{
_C	12	FF	FS	,	<	L	\	l	|
_D	13	CR	GS	-	=	M	]	m	}
_E	14	SO	RS	.	>	N	^	n	~
_F	15	SI	US	/	?	O	_	p	

Bibliography

[Abelson00] H. Abelson, D. Allen, D. Coore, *et al.* Amorphous Computing. In *Communications of the ACM*, volume 43, pages 74–82. May 2000.

[Altenbernd96] P. Altenbernd and C. Ditze. Allocation of periodic hard real-time tasks. In P. Laplante and W. Halang, editors, *Real Time Programming 1995 (WRTP'95). A Postprint Volume from the 20th IFAC/IFIP Workshop*, pages vi+213, 197–204. Pergamon; Oxford, UK, 1996.

[Altenbernd98] P. Altenbernd and H. Hansson. The Slack Method: a new method for static allocation of hard real-time tasks. *Real-Time Systems*, volume 15(2) pages 103–30, September 1998.

[Bapty99] T. Bapty, J. Scott, S. Neema, *et al.* Uniform execution environment for dynamic reconfiguration. In *Proceedings ECBS'99. IEEE Conference and Workshop on Engineering of Computer-Based Systems*, pages 181–187. March 1999.

[Bazargan00] K. Bazargan, R. Kastner, and M. Sarrafzadeh. 3-D floorplanning: simulated annealing and greedy placement methods for reconfigurable computing systems. *Design Automation for Embedded Systems*, volume 5(3-4) pages 329–38, August 2000.

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

[Beck98] J. E. Beck and D. P. Siewiorek. Automatic configuration of embedded multicomputer systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 17(2) pages 84–95, Feb 1998.

[Beck00] J. Beck, M. Reagin, T. E. Sweeny, *et al.* Applying a Component-Based Software Architecture to Robotic Workcell Applications. *IEEE Transactions on Robotics and Automation*, volume 16(3) pages 207–217, June 2000.

[Benveniste91] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, volume 79(9) pages 1270–82, September 1991.

[Beveridge02] M. Beveridge and P. Koopman. Jini Meets Embedded Control Networking: A Case Study in Portability Failure. In *Proceedings of Seventh IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2002)*. January 2002.

[Blickle98] T. Blickle, J. Teich, and L. Thiele. System-level Synthesis Using Evolutionary Algorithms. *Design Automation for Embedded Systems*, volume 3(1) pages 23–58, Jan 1998.

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

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

[Borgerson75] B. R. Borgerson and R. F. Freitas. A reliability model for gracefully degrading and standby-sparing systems. *IEEE Transactions on Computers*, volume C-24(5) pages 517–25, May 1975.

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

[Brownsword96] L. Brownsword and P. Clements. A Case Study in Successful Product Line Development. Technical Report CMJU/SEI-96-TR-016, Software Engineering Institute, 1996. URL `\url{http://www.sei.cmu.edu/pub/documents/96.reports/pdf/tr01%6.96.pdf}`.

[Cailliau99] D. Cailliau and R. Bellenger. The Corot instruments software: towards intrinsically reconfigurable real-time embedded processing software in space-borne instruments. In *Proceedings of Fourth IEEE International Symposium on High Assurance Systems Engineering (HASE'99)*, pages 75–80. Washington, DC, Nov 1999.

[CAN] Control Area Network Specification, v2.0. `http://www.algonet.se/~staffann/developer/CAN.htm`.

[Chiodo94] M. Chiodo, P. Giusto, A. Jurecska, *et al.* Hardware–software codesign of embedded systems. *IEEE Micro*, volume 14(4) pages 26–36, Aug 1994.

[Chou00] P. H. Chou and G. Borriello. Synthesis and optimization of coordination controllers for distributed embedded systems. In *Proceedings of ACM/IEEE-CAS/EDAC Design Automation Conference*, pages 410–15. Los Angeles, CA, June 2000.

[Cohen95] L. Cohen and L. Cohen. *Quality Function Deployment.* Prentice Hall, 1995.

[Coleman90] G. L. Coleman, C. P. Ellison, G. G. Gardner, *et al.* Experience in modeling a concurrent software system using STATEMATE. In *Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering (COMPEURO'90)*, pages 104–8. May 1990.

[Cristian91] F. Cristian. Understanding fault–tolerant distributed systems. *Communications of the ACM*, volume 34(2) pages 56–78, Feb 1991. Frequently incorrectly dated as May 1993 (including on cover page) Ooops.

[Czarnecki00] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.

[Dave99] B. P. Dave. CRUSADE: hardware/software co-synthesis of dynamically reconfigurable heterogeneous real-time distributed embedded systems. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 97–104. March 1999.

[Di Vito97] B. 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.

[Edwards97] S. Edwards, L. Lavagno, E. A. Lee, *et al.* Design of embedded systems: formal models, validation, and synthesis. *Proceedings of the IEEE*, volume 85(3) pages 366–90, Mar 1997.

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

[Ernst98] R. Ernst. Codesign of embedded systems: status and trends. *IEEE Design and Test of Computers*, volume 15(2) pages 45–54, Apr 1998.

[Even73] S. Even. *Algorithmic Combinatorics*. Macmillan Company, New York, 1973.

[Garey79] M. Garey and D. Johnson. *Computers and Intractability: a Guide to the Theory of NP-completeness*. Freeman, San Francisco, 1979.

[Gartner99] F. C. Gartner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, volume 31(1) pages 1–26, Mar 1999.

[Goldstein00] S. Goldstein, H. Schmit, M. Budiu, *et al.* PipeRench: a reconfigurable architecture and compiler. *Computer*, volume 33(4) pages 70–77, April 2000.

[Green00] P. N. Green and M. D. Edwards. Object oriented development method for reconfigurable embedded systems. *IEE Proceedings on Computers and Digital Techniques*, volume 147(3) pages 153–8, May 2000.

[Gupta93] R. Gupta and G. De Micheli. Hardware-software cosynthesis for digital systems. *IEEE Design and Test of Computers*, volume 10(3) pages 29–41, Sept 1993.

[Herlihy91] M. P. Herlihy and J. M. Wing. Specifying Graceful Degradation. In *IEEE Transactions on Parallel and Distributed Systems*, volume 2, pages 93–104. Jan 1991.

[Hu94] X. Hu, J. D'Ambrosio, B. Murray, *et al.* Codesign of architectures for automotive powertrain modules. *IEEE Micro*, volume 14(4) pages 17–25, August 1994.

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

[Jiao00] J. Jiao and M. Tseng. Fundamentals of Product Family Architecture. *Integrated Manufacturing Systems*, volume 11(7) pages 469–483, 2000.

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

[Kasahara84] H. Kasahara and S. Narita. Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing. *IEEE Transactions on Computers*, volume C33(11) pages 1023–9, Nov 1984.

[Kiczales97] G. Kiczales, J. Lamping, A. Mendhekar, *et al.* Aspect-oriented programming. In *Proceedings of 11th European Conference*

on Object-Oriented Programming (ECOOP'97), pages 220–42. 1997.

[Knieser96] M. J. Knieser and C. A. Papachriston. COMET: A Hardware-Software Codesign Methodology. In *Proceedings of European Design Automation Conference (EURO-DAC'96)*, pages 178–83. 1996.

[Knight00] J. C. Knight and K. J. Sullivan. On the Definition of Survivability. Technical Report CS-TR-33-00, University of Virginia, Department of Computer Science, 2000. URL `http://www.cs.virginia.edu/~jck/publications/tech.report.2000%.33.pdf`.

[Knuth73] D. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1, pages 278–295. Addison-Wesley, Reading, Mass, 2 edition, 1973.

[Kopetz] H. Kopetz. Time Triggered Protocol Specification. `http://www.ttpforum.org/`.

[Kopetz97] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston, 1997.

[Kwok99] Y. Kwok and I. Ahmad. Benchmarking and Comparison of the Task Graph Scheduling Algorithms. *Journal of Parallel and Distributed Computing*, volume 59(3) pages 381–422, Dec 1999.

[Lee87] E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, volume 75(9) pages 1235–45, Sep 1987.

[Lee95] E. Lee and T. Parks. Dataflow process networks. *Proceedings of the IEEE*, volume 83(5) pages 773–801, May 1995.

[Lee01] J. Lee. *Automated Specification and Task Allocation Methods for Single and Multimode Embedded Applications.* Ph.D. thesis, Carnegie Mellon University, 2001.

[Leen02] G. Leen and D. Heffernan. Expanding Autmotive Electonic Systems. *IEEE Computer*, volume 35(1) pages 88–93, 2002.

[Li00] Y. Li, T. Callahan, E. Darnell, *et al.* Hardware-Software Co-Design of Embedded Reconfigurable Architectures. In *Proceedings of the 2000 Design Automation Conference*, pages 507–512. Los Angeles, CA, Dec 2000.

[Liu87] L. Y. Liu and R. K. Shyamasundar. A formal specification of an elevator system. Technical Report CS-87-21, Pennsylvania State University, 1987.

[Lyons62] R. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, volume 6 pages 200–9, April 1962.

[McNally98] G. McNally. *Automated Architecture Specification for Embedded Multicomputer Systems.* Ph.D. thesis, Carnegie Mellon University, 1998.

[Pimentel01] A. Pimentel, L. Hertzberger, P. Lieverse, *et al.* Exploring Embedded-Systems Archetectures with Artemis. *IEEE Computer*, volume 34(11) pages 57–64, November 2001.

[Prakash92] S. Prakash and A. Parker. SOS: Synthesis of Application-Specific Heterogeneous Multiprocessor Systems. *Journal of*

Parallel and Distributed Computing, volume 16(4) pages 338–51, Dec 1992.

[Ramanathan97] P. Ramanathan. Graceful degradation in real-time control applications using (m, k)-firm guarantee. In *Digest of Papers. Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing*, pages xvii+396, 132–41. IEEE Comput. Soc; Los Alamitos, CA, USA, 1997.

[Reagin99] J. Reagin, J. E. Beck, T. Sweeny, *et al.* A Component-Based Software Architecture for Robotic Workcell Applications. *IEEE Transactions on Electronics Packaging Manufacturing*, volume 22(1) pages 85–94, January 1999.

[Sendall00] S. Sendall and A. Strohmeier. From use cases to system operation specifications. In *Proceedings of Third International Conference on The Unified Modeling Language (UML2000)*, pages 1–15. Springer-Verlag, Berlin, Oct 2000.

[Sha98] L. Sha. Dependable system upgrade. In *Proceedings 19th IEEE Real-Time Systems Symposium*, pages xiii+493, 440–8. IEEE Comput. Soc; Los Alamitos, CA, USA, December 1998.

[Shelton01] C. Shelton and P. Koopman. Developing a Software Architecture for Graceful Degradation in an Elevator Control System. In *Proceedings of the SRDS 2001 Workshop on Reliability in Embedded Systems*, pages 11–15. Oct 2001.

[Shelton02] C. Shelton and P. Koopman. Using Architectural Properties to Model and Measure System-Wide Graceful Degradation. In *Proceedings of Workshop on Architecting Dependable Systems sponsored by the International Conference on Software Engineering (ICSE 2002)*. May 2002.

[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*, volume C34(3) pages 197–203, Mar 1985.

[Shimomura95] Y. Shimomura, S. Tanigawa, Y. Umeda, *et al.* Development of self-maintenance photocopiers. *AI Magazine*, volume 16(4) pages 41–53, Winter 1995.

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

[Stone78] H. Stone and S. Bokhari. Control of distributed processes. *Computer*, volume 11(7) pages 97–106, July 1978.

[Tarr99] P. Tarr, H. Ossher, W. Harrison, *et al.* N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 107–119. 1999.

[Thomas93] D. E. Thomas, J. Adams, and H. Schmit. A model and methodology for hardware–software codesign. *IEEE Design and Test of Computers,*, volume 10(3) pages 6–15, Sep 1993.

[Tindell00] K. Tindell. Deadline Monotonic Analysis. *Embedded Systems Programming*, volume 13(6), June 2000.

[Tobita00] T. Tobita, M. Kouda, and H. Kasahara. Performance Evaluation of Minimum Execution Time Multiprocessor Scheduling Algorithms Using Standard Task Graph Set. In *2000 Intl Conf on Parallel and Distributed Processing Techniques and Applications*, pages 745–751. Jun 2000.

[Trotter62] H. F. Trotter. Algorithm 115: Perm. *Communications of the ACM*, volume 5(8) pages 434–35, Aug 1962.

[Van Hentenryck99] P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, Cambridge, Massachusetts, 1999.

[Weber89] D. Weber. Formal specification of fault-tolerance and its relation to computer security. In *Proceedings of Fifth International Workshop on Software Specification and Design*, pages xix+295, 273–7. IEEE Comput. Soc. Press; Washington, DC, USA, May 1989.

[Wolf97] W. H. Wolf. An architectural co-synthesis algorithm for distributed, embedded computing systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, volume 5(2) pages 218–229, June 1997.

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

[Zuberi00] K. Zuberi and K. Shin. Design and implementation of efficient message scheduling for controller area network. *IEEE Transactions on Computers*, volume 49(2) pages 182–8, February 2000.