Configuration and Deployment Derivation Strategies for Distributed Real-time and Embedded Systems

Brian Dougherty
briand@dre.vanderbilt.edu

June 2, 2010

Contents

List of Figures

Abstract

In contrast to federated systems in which hardware and software are tightly coupled, distributed real-time and embedded (DRE) are constructed by allocating software tasks to hardware. This allocation, known as a *deployment plan*, must ensure that several design constraints, such as QoS demands and strict resource requirements, are satisfied. Further, the financial cost and performance of these systems may differ greatly based on software allocation decisions. As a result, creating a low cost, high performance deployment that satisfies all design constraints is difficult.

Distributed real-time and embedded (DRE) systems can also be constructed by using a new development paradigm that relies on configuring commercial-off-the-shelf (COTS) components rather than developing source code from scratch. COTS components are reusable hardware, operating system, and middleware components. Configuring large-scale DRE systems with COTS components reduces development-cycle time and cost. DRE systems are subject to strict resource requirements as well as quality of service (QoS) demands, making DRE system configuration difficult.

This proposal describes techniques for addressing the challenges of deriving DRE system configurations and deployments. First, we show how heuristic algorithms can be utilized to rapidly determine system deployments that meet QoS demands and resource requirements. Second, we examine the use of metaheuristic algorithms to optimize system-wide deployment properties. Next, we describe a Model-Driven Architecture (MDA) based methodology for constructing a DRE system configuration modeling tool. Finally, we demonstrate a methodology that allows for DRE systems to be evolved over time as new COTS components become available.

Chapter 1

Introduction

Distributed real-time and embedded systems are constructed by determining an allocation of software tasks to hardware, known as a *deployment plan* or by configuring commercial-off-the-shelf (COTS) components. In both cases, systems are subject to strict resource requirements, such as memory and CPU utilization, and stringent QoS demands, such as real-time deadlines and co-location constraints, making DRE system construction difficult. Further, intelligently constructing DRE systems can result in significant performance increases, reductions in financial cost and other benefits.

For example, minimizing the computing infrastructure (such as processors) in a distributed real-time embedded (DRE) system deployment helps reduce system size, weight, power consumption, and cost. To support software components and applications on the computing infrastructure, the hardware must provide enough processors to ensure that all applications can be scheduled without missing real-time deadlines. In addition to ensuring scheduling constraints, sufficient resources (such as memory) must be available to the software. It is hard to identify the best way(s) of deploying software components on hardware processors to minimize computing infrastructure and meet complex DRE constraints.

Often, it is desirable to optimize system-wide properties of DRE system deployments. For example, a deployment that minimizes network bandwidth may exhibit higher performance and reduced power consumption. Intelligent algorithms, such as metaheuristic techniques, can be used to refine system deployments to reduce system cost and resource requirements, such as memory and processor utilization. Applying these algorithms to create computer-assisted deployment optimization tools can result in substantial reductions of processors and network bandwidth consumption requirements of legacy DRE systems.

DRE systems are also being constructed with commercial-off-the-shelf components to reduce development time and effort. The configuration of these components must ensure that real-time quality-of-service (QoS) and resource constraints are satisfied. Due to the numerous QoS constraints that must be met, manual system configuration is hard. Model-Driven Architecture (MDA) is a design paradigm that incorporates models to provide visual representations of design entities. MDAs show promise for addressing many of these challenges by allowing the definition and automated enforcement of design constraints.

As DRE systems continue to become more widely utilized, system size and complexity is also increasing. As a corollary, the design and configuration of such systems is becoming an arduous task. Cost-effective software evolution is critical to many distributed real-time and embedded (DRE) systems. Selecting the lowest cost set of software components that meet DRE system resource constraints, such as total memory and available CPU cycles, is an NP-Hard problem. Therefore, intelligent automated techniques must be implemented to determine cost-effective evolution strategies in a timely manner.

1.1 Overview of Research Challenges

Several inherent complexities, such as strict resource requirements and rigid QoS demands, make deriving valid

DRE system deployments and configurations difficult. This problem is exacerbated by the fact that many valid deployments and configurations may exist that differ in terms of financial cost and performance, making some deployments and configurations vastly superior to others. The following challenges must be overcome to discover superior DRE system deployments and configurations:

1. **Strict Resource Requirements.** DRE system configurations and deployments must adhere to strict resource constraints. If the resource requirements, such as memory and cpu utilization, of software exceed the resource production of hardware, then the software may fail to function or execute in an unpredictable manner.

2. **QoS Guarantees.** It is critical that DRE system configurations and deployments ensure that rigorous QoS constraints, such as real-time deadlines, are upheld. Therefore, for a deployment or configuration to be valid, a scheduling of software tasks must exist that allows the software to execute without exceeding predefined real-time deadlines.

3. **Co-location Constraints.** To ensure faul-tolerance and other domain specific constraints, DRE systems are often subject to co-location constraints. Co-location constraints require that certain software tasks or components be placed on the same hardware while prohibiting others to occupying a common allocation.

4. **Exponential Solution Space.** Given a set of software and hardware, there is an exponential number of different deployments or configurations exist. Strict resource requirements and QoS constraints, however, invalidate the vast majority of these deployments, making manual derivation techniques obsolete. Due to the massive nature of the solution space, automated exhaustive techniques for determining deployments or configurations of even relatively small systems may take years to complete.

5. **Variable Cost & Performance.** Valid deployments and configurations may differ greatly in terms of financial cost and performance. Therefore, techniques must be capable of discovering solutions that not only satisfy design constraints, but also yield high performance while carrying a low financial cost.

1.2 Overview of Research Approach

To overcome the challenges of determining valid DRE system deployments, configurations and evolution strategies, we apply a combination of several heuristic algorithms, such as bin-packing, metaheuristic algorithms, such as genetic algorithms and particle swarm optimization techniques, and model-driven configuration techniques. These techniques are utilized as described below:

1. **Automated Deployment Derivation** uses heuristic bin-packing to allocate software tasks to hardware processors while ensuring that resource constraints, such as memory and cpu cycles, real-time deadlines, and co-location constraints are satisfied. By defining strict space constraints of bins based on the available resources of hardware nodes and schedulability analysis of software tasks, bin-packing can be used to determine deployments that satisfy all design constraints in a timely manner.

2. **Legacy Deployment Optimization** requires that design constraints are satisfied while also minimizing system-wide properties, such as network bandwidth utilization. This process is difficult since the impact on network bandwidth utilization cannot be determined by examining the allocation of a single software task. Metaheuristic techniques, such as particle swarm optimization techniques and genetic algorithms, can be used in conjunction with heuristic bin-packing to discover optimized deployments that would not be found with heuristic bin-packing alone. For example, this technique could be applied to a legacy avionics deployment to determine if software tasks could be allocated differently to create a deployment that consumes less network bandwidth and carries a reduced financial cost.

3. **MDA Driven DRE System Configuration** techniques allow designers to model DRE system configuration design constraints, domain specific constrains, and facilitate the derivation of low-cost, valid configurations. For example, designers can

use model-driven tools to represent the DRE system constraints of a smart car, investigate the impact of adding a new component, such as an electronic control unit, and automatically determine if a configuration exists that will support the additional component.

4. **Automated Hardware/Software Evolution** techniques allow designers to enhance existing DRE system configurations by adding or removing COTS components rather than constructing costly new DRE systems from scratch, resulting in increased system performance and lower financial costs. For example, a system designer could specify a set of legacy components that are eligble for replacement and a set of potential replacement components. Automated evolution can be used to generate a set of replacement components and a set of components to remove that would yield increased performance and/or reduced financial cost.

1.3 Research Contributions

1.3.1 BLITZ

Research contributions:

1. We present the Bin-packing LocalizatIon Technique for processor minimiZation (BLITZ), a deployment technique that minimizes the required number of processors, while adhering to real-time scheduling, resource, and co-location constraints.

2. We show how this technique can be augmented with a harmonic period heuristic to further reduce the number of required processors.

3. We present empirical data from applying three different deployment algorithms for processor minimization to a flight avionics DRE system

Conference Publications

1. Brian Dougherty, Jules White, Jaiganesh Balasubramanian, Chris Thompson, and Douglas C. Schmidt, Deployment Automation with BLITZ, 31st International Conference on Software Engineering, May 16-24, 2009 Vancouver, Canada.

1.3.2 ScatterD

Research contributions:

1. We present a heuristic bin-packing technique for satisfying deployment resource and real-time constraints.

2. We combine heuristic bin-packing with metaheuristic algorithms to create ScatterD, a technique for optimizing system wide properties while enforcing deployment constraints.

3. We apply ScatterD to optimize a legacy industry flight avionics DRE system and present empirical results of network bandwidth and processor reductions.

Journal Publications

1. Jules White, Brian Dougherty, Chris Thompson, Douglas C. Schmidt, ScatterD: Spatial Deployment Optimization with Hybrid Heuristic / Evolutionary Algorithms, ACM Transactions on Autonomous and Adaptive Systems Special Issue on Spatial Computing

Submitted

1. Brian Dougherty, Jules White, Douglas C. Schmidt, Jonathan Wellons, Russell Kegley, Deployment Optimization for Embedded Flight Avionics Systems, STSC Crosstalk (2010)

1.3.3 ASCENT Modeling Platform

Research contributions:

1. We present the challenges that make manual DRE system configuration infeasible.

2. We provide an incremental methodology for constructing modeling tools to alleviate these difficulties.

3. We provide a case study describing the construction of the Ascent Modeling Platform (AMP), which is a modeling tool capable of producing near-optimal DRE system configurations.

Journal Publications

1. Jules White, Brian Dougherty, Douglas C. Schmidt, ASCENT: An Algorithmic Technique for Designing Hardware and Software in Tandem, IEEE Transactions on Software Engineering Special Issue on Search-based Software Engineering, December, 2009, Volume 35, Number 6

2. Jules White, Brian Dougherty, Douglas C. Schmidt, Selecting Highly Optimal Architectural Feature Sets with Filtered Cartesian Flattening, Journal of Systems and Software, August 2009, Volume 82, Number 8, Pages 1268-1284

Book Chapters

1. Brian Dougherty, Jules White, Douglas C. Schmidt, Model-drive Configuration of Distributed, Real-time and Embedded Systems, Model-driven Analysis and Software Development: Architectures and Functions, edited by Janis Osis and Erika Asnina, IGI Global, Hershey, PA, USA 2009

1.3.4 SEAR

Research contributions:

1. We present the Software Evolution Analysis with Resources (SEAR) technique that transforms component-based DRE system evolution alternatives into multidimensional multiple-choice knapsack problems.

2. We compare several techniques for solving these knapsack problems to determine valid, low-cost design configurations for resource constrained component-based DRE systems.

3. We empirically evaluate the techniques to determine their applicability in the context of common evolution scenarios.

4. Based on these findings, we present a taxonomy of the solving techniques and the evolution scenarios that best suit each technique.

Journal Publications

1. Jules White, Brian Dougherty, Douglas C. Schmidt, Selecting Highly Optimal Architectural Feature Sets with Filtered Cartesian Flattening, Journal of Systems and Software, August 2009, Volume 82, Number 8, Pages 1268-1284

2. Jules White, Brian Dougherty, Douglas C. Schmidt, ASCENT: An Algorithmic Technique for Designing Hardware and Software in Tandem, IEEE Transactions on Software Engineering Special Issue on Search-based Software Engineering, December, 2009, Volume 35, Number 6

Conference Publications

1. Brian Dougherty, Jules White, Chris Thompson, and Douglas C. Schmidt, Automating Hardware and Software Evolution Analysis, 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS), April 13-16, 2009 San Francisco, CA USA.

1.4 Proposal Organization

Each research topic is separated into a chapter describing the advancements made in each area. The remainder of this proposal is organized as follows: Chapter 2 showcases automated deployment derivation of DRE systems; Chapter 3 presents deployment optimization techniques; Chapter 4 describes the creation of a modeling tool for automated DRE system configuration; Chapter 5 demonstrates a methodology for automatically evolving DRE systems configurations; and Chapter 6 presents lessons learned.

Chapter 2

Automated Deployment Derivation

2.1 Challenge Overview

This chapter provides motivation for automated deployment derivation techniques to determine valid DRE system deployments. We introduce a heuristic technique for processor minimization of a legacy flight avionics system. We show how the application of this technique can substantially reduce the hardware requirements and cost of deployments while satisfying additional DRE system constraints.

2.2 Introduction

Software engineers who develop distributed real-time and embedded (DRE) systems must carefully map software components to hardware. These software components must adhere to complex constraints, such as real-time scheduling deadlines and memory limitations, that are hard to manage when planning deployments that map the software components to hardware [1]. How software engineers choose to map software to hardware has a direct impact on the number of processors required to implement a system.

Ideally, software components for DRE systems should be deployed on as few processors as possible. Each additional processor used by a deployment adds size, weight, power consumption, and cost to the system [2]. For example, it has been estimated that each additional pound of computing infrastructure on a commercial aircraft results in a yearly loss of $100 per aircraft in fuel costs. Likewise, each pound of processor(s) requires four additional pounds of cooling, power supply, and other support hardware. Naturally, reducing fuel consumption also reduces emissions, benefiting the environment [3].

Several types of constraints must be considered when determining a valid *deployment plan*, which allocates software components to processors. First, software components deployed on each processor must not require more resources, such as memory, than the processor provides. Second, some components may have co-location constraints, requiring that one component be placed on the same processor as another component. Moreover, all components on a processor must be schedulable to assure they meet critical deadlines [4].

Existing automated deployment techniques [5–7] leveraged by software engineers do not handle all these constraints simultaneously. For example, Rate Monotonic First-Fit Scheduling [7] can guarantee real-time scheduling constraints, but does not guarantee memory constraints or allow for forced co-location of components. Co-location of components is a critical requirement in many DRE systems. Moreover, if deploying a set of components on a processor results in CPU over-utilization, critical tasks performed by a software component may not complete by their deadline, which may be catastrophic. DRE software engineers must therefore identify deployments that meet these myriad constraints *and* minimize the total number of processors [8].

We provide three contributions to the study of software component deployment optimizations for DRE systems that address the challenges outlined above.

1. We present the *Bin packing LocatIon Technique for processor minimiZation* (BLITZ), which uses bin packing to allocate software applications to a min-

imal number of processors and ensure that real-time scheduling, resource, and co-location constraints are simultaneously met.

2. We describe a case study that motivates the minimization of processors in a production flight avionics DRE system.

3. We present empirical comparisons of minimizing processors for deployments with BLITZ for three different scheduling heuristics versus the simple bin-packing of one component per processor used in the avionics case study.

2.3 Challenges of Component Deployment Minimization

This section summarizes the challenges of a determining a software component deployment that minimizes the number of processors in a DRE system.

Rate-monotonic scheduling constraints. To create a valid deployment, the mapping of software components to processors must guarantee that none of the software components' tasks misses its deadline. Even if rate monotonic scheduling is used, a series of components that collectively utilize less than 100% of a processor may not be schedulable. It has been shown that determining a deployment of multiple software components to multiple processors that will always meet real-time scheduling constraints is NP-Hard [5].

Task co-location constraints. In some cases, software components must be co-located on the same processor. For example, variable latency of communication between two components on separate processors may prevent real-time constraints from being honored. As a result, some components my require co-location on the same processor, which precludes the use of bin-packing algorithms that treat each software component to deploy as a separate entity.

Resource constraints. To create a validate deployment, each processor must provide the resources (such as memory) necessary for the set of software components it supports to function. Developers must ensure that components deployed to a processor do not consume more resources than are present. If each processor does not provide a sufficient amount of these resources to support all tasks on the processor, a task will not be able execute, resulting in a failure.

2.4 Deployment Derivation with BLITZ

The *Binpacking LocalizatIon Technique for processor minimiZation* (BLITZ) is a first-fit decreasing binpacking algorithm we developed to (1) assign processor utilization values that ensure schedulability if not exceeded and (2) enhance existing techniques by ensuring that multiple resource and co-location constraints are simultaneously honored.

2.4.1 BLITZ Bin-packing

The goal of a bin packer is to place a set of items into a minimal set of bins. Each item takes up a certain amount of space and each bin has a limited amount of space available for packing. An item can be placed in a bin as long as its placement does not exceed the remaining space in the bin. Multi-dimensional bin packing extends the algorithm by adding extra dimensions to bins and items (*e.g.*, length, width, and height) to account for additional requirements of items. For example, an item may have height corresponding to its CPU utilization and width corresponding to consumed memory.

BLITZ uses an enhanced multi-dimensional bin packing algorithm to generate valid deployments that honor multiple resource constraints and co-location constraints as well as the standard real-time scheduling constraints. In BLITZ, each processor is modeled as a bin and each independent component or co-located group of components is modeled as an item. Each bin has a dimension corresponding to the available CPU utilization. Each item has a dimension that represents the CPU utilization it requires, as well as a a dimension corresponding to each resource, such as memory, that it consumes. Each bin's size dimension corresponding to available CPU utilization is initialized 100%. The resource dimensions are set to the amount of each resource that the processor offers.

To pack the items, they are first sorted in decreasing order of utilization. Next, BLITZ attempts to place the

first item in the first bin. If the placement of the item does not exceed the size of the bin (available resources and utilization) of the bin (processor), the item is placed in the bin. The dimensions of the items are then subtracted from the dimensions of the bin to reflect the addition. If the item does not fit, BLITZ attempts to insert the item into the next bin. This step is repeated until all items are packed into bins or no bin exists that can contain the item.

Burchard et al [9] describe several techniques that use component partitioning and bin-packing to reduce total required processors. This work, however, does not account for additional resource constraints, such as memory. Furthermore, these techniques do not allow for co-location constraints that require specific components to reside on the same processor.

2.4.2 Utilization Bounds

Conventional bin-packing algorithms assume that each bin has a static series of dimensions corresponding to available resources. For example, the amount of RAM provided by the processor is constant. Applying conventional bin-packing algorithms to software component deployment is challenge since it is hard to set a static bin dimension that guarantees the components are schedulable. Scheduling can only be modeled with a constant bin dimension of utilization if a worst-case scheduling of the system is assumed. Liu-Layland [10] have shown that a fixed bin dimension of 69.4% will guarantee schedulability but in many cases, tasks can have a higher utilization and still be schedulable.

The Liu-Layland equation states that the maximum processor utilization that guarantees schedulability is equal to $2^{1/x} - 1$, where x is the total number of components allocated to the processor. With BLITZ, each bin has a scheduling dimension that is determined by the Liu-Layland equation and the number of components currently assigned to the bin. Each item will represent at least one but possibly multiple co-located components. Each time an item is assigned to a bin, BLITZ uses the Liu-Layland formula to dynamically resize the bin's scheduling dimension according to the number of components held by the items in the bin.

If the the frequency of execution, or periodicity, of the components' execution requirements is known, higher processor utilization above the Liu-Layland bound is also possible. Components with harmonic periods (*e.g.*, periods that can be repeatedly doubled or halved to equal each other) can be allocated to the same processor with schedulability ensured, as long as the total utilization is less than or equal to 100%.

Unlike other deployment algorithms [9, 11], BLITZ uses multi-stage packing to exploit harmonic periods. In the first stage, components with harmonic periods are grouped together. In each successive stage, the components from the group with the largest aggregate processor utilization are deployed to the processors using a first-fit packing scheme. If not all periods of the components in a bin are harmonic, an item is allocated to a bin only if the utilization of its components fits within the dynamic scheduling Liu-Layland dimension and all other resource dimensions. If all component periods within a bin are harmonic, the utilization dimension is not dynamically calculated with Liu-Layland and a fixed value of 100% is used.

2.4.3 Co-location Constraints

To allow for component co-location constraints, BLITZ groups components that require co-location into a single item. Each item has utilization and resource consumption equal to that of the component(s) it represents. Each item remembers the components associated with it. The Liu-Layland and harmonic calculations are performed on the individual components associated with the items in a bin and not each item as a whole.

2.5 Empirical Results

This section presents the results of applying BLITZ to a flight avionics case study provided by Lockheed Martin Aeronautics through the SPRUCE portal (www.sprucecommunity.org), which provides a web-accessible tool that pairs academic researchers with industry challenge problems complete with representative project data. This case study comprised 14 processors, 89 total components, and 14 co-location constraints. We compared 2 different bin-packing strategies against both BLITZ and the baseline deployment of this avionics system, produced by the original avionics domain experts.

2.5.1 Experimental Platform

All algorithms were implemented in Java and all experiments were conducted on an Apple MacbookPro with a 2.4 GHz Intel Core 2 Duo processor, 2 gigabytes of RAM, running OS X version 10.5.5, and a 1.6 Java Virtual Machine (JVM) run in client mode. All experiments required less than 1 second to complete with each algorithm.

2.5.2 Processor Minimization with Various Scheduling Bounds

This experiment compared the following bin-packing strategies against BLITZ and the baseline deployment of the avionics system: (1) a worst-case multi-dimensional bin-packing algorithm that uses 69.4% as the utilization bound for each bin, (2) a dynamic multi-dimensional bin-packing algorithm that uses the Liu-Leyland equation to recalculate the utilization bound for each bin as components are added, and (3) our BLITZ technique that combines dynamic utilization bound recalculation with the harmonic period multi-stage packing.

We used each technique to generate a deployment plan for the avionics system described in Section 2.5. Figure 2.1 shows the original avionics system deployment, as well as deployment plans generated by the worst-case bin-packing algorithm, dynamic bin-packing algorithm, and BLITZ.

The BLITZ technique required 6 less processors than the original deployment plan, 3 less processors than the worst-case bin-packing algorithm, and 1 less processor than the dynamic bin-packing algorithm.

Figure 2.2 shows the total reduction of processors from the original deployment plan for each algorithm. The deployment plan generated by the worst-case bin-packing algorithm reduces the required number of processors by 3 or 21.41%. The dynamic bin-packing algorithm yields a deployment plan that reduces the number of required processors by 5, or 35.71%. BLITZ reduces the number of required processors even further, generating a deployment plan that requires 6 less processors, a 43.86% reduction.

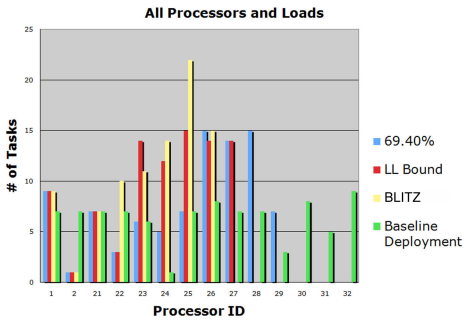

Figure 2.1: Deployment Plan Comparison

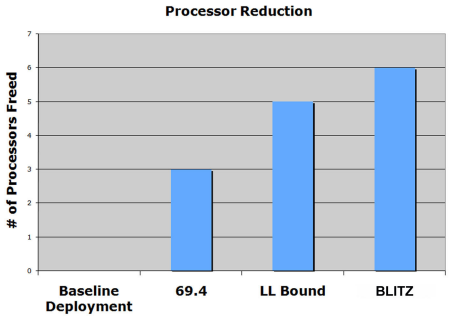

Figure 2.2: Scheduling Bound vs Number of Processors Reduced

2.6 Related Work

Deployment Minmization. Burchard et al [9] describe several techniques that use component partitioning and bin-packing to reduce total required processors. These techniques use several different heuristics based on scheduling characteristics to determine more efficient deployment plans. This work, however, does not account for additional resource constraints or co-location requiremets. BLITZ enforces resource consrtaints by setting an additional dimension to the bins for each resoruce. Co-location constraints are accounted for by combining components that share a co-location constraint into a single item prior to packing.

Task Allocation wtih Simulated Annealing. Tindell et al [12] investigate the use of simulated annealing to generate deployments that optimize system response time. Unlike heuristic algorithms, such as heuristic bin-packing, simulated annealing does not require designers to specify an intelligent heuristic to determine task allocation.

Instead, simulated annealing only requires that a metric is determined to score potential solution. After a potential allocation is examined and scored, simulated annealing uses an element of randomness to determine the next allocation to be investigated. This allows multiple executions of the algorithm to potentially determine different deployment plans.

The heuristic used by BLITZ to determine deployment plans, however, is static. Therefore, BLITZ will always determine the same deployment plan for a set of software tasks and hardware processors. This application of simulate annealing, however, does not take into acount resource constraints or co-location requirements. Therefore, this technique must be altered to ensure that all DRE system constraints are satisfied.

Chapter 3

Legacy Deployment Optimization

3.1 Challenge Overview

This chapter presents the motivation for the optimization of system-wide deployment properties to create new cost effective, efficient DRE system deployments or to enhance existing legacy deployments. To showcase the potential for improvement in this area, we apply our technique to a legacy flight avionics system. We demonstrate how combining heuristic algorithms with metaheuristic techniques can yield considerable reductions in computational requirements.

3.2 Introduction

Current trends and challenges. Several trends are shaping the development of embedded flight avionics systems. First, there is a migration away from older *federated computing architectures* where each subsystem occupied a physically separate hardware component to *integrated computing architectures* where multiple software applications implementing different capabilities share a common set of computing platforms. Second, publish/subscribe (pub/sub)-based messaging systems are increasingly replacing the use of hard-coded cyclic executives.

These trends are yielding a number of benefits. For example, integrated computing architectures create an opportunity for system-wide optimization of *deployment topologies*, which map software components and their associated tasks to hardware processors as shown in Figure 3.1.

Optimized deployment topologies can pack more software components onto the hardware, thereby optimizing system processor, memory, and I/O utilization [13–15]. Increasing hardware utilization can decrease the total hardware processors that are needed, lowering both implementation costs and maintenance complexity. Moreover, reducing the required hardware infrastructure has other positive side effects, such as reducing weight and power consumption. Decoupling software from specific hardware processors also increases flexibility by not coupling embedded software application components with specific hardware processing platforms. It is estimated that each pound of processor savings on a plane results in $200 in decreased fuel costs and a decrease in greenhouse gas production from less burned fuel [3].

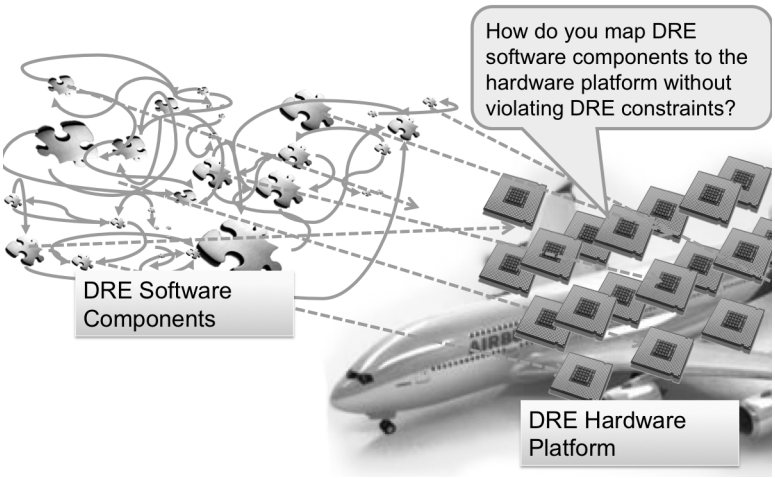

Figure 3.1: Flight Avionics Deployment Topology

Open problems. The explosion in the size of

the search space for large-scale embedded deployment topologies makes it hard to optimize them without computer-assisted methods and tools to evaluate the schedulability, network bandwidth consumption, and other characteristics of a given configuration. Developing computer-assisted methods and tools to deploy software to hardware in embedded systems is hard [1,16] due to the number and complexity of constraints that must be addressed.

For example, developers must ensure that each software component is provided with sufficient processing time to meet any real-time scheduling constraints [17]. Likewise, resource constraints (such as total available memory on each processor) must also be respected when mapping software components to hardware components [17, 18]. Components may also have complex placement or colocation constraints, such as requiring the deployment of specific software components to processors at a minimum distance from the engine of an aircraft to provide survivability in case of an engine malfunction [18]. Moreover, assigning real-time tasks in multiprocessor and/or single-processor machines is *NP-Hard* [5], which means that such a large number of potential deployments exist that it would take years to investigate all possible solutions.

Due to the complexity of finding valid deployment topologies, it is difficult for developers to evaluate system-wide design optimization alternatives that may emphasize different properties, such as fault-tolerance, performance, or heat dissipation.

Current algorithmic deployment techniques are largely based on heuristic bin-packing [5–7], which represents the software tasks as *items* that take up a set amount of space and hardware processors as *bins* that provide limited space. Bin-packing algorithms try to place all the items into as few bins as possible without exceeding the space provided by the bin in which they are placed. These algorithms use a heuristic, such as sorting the items based on sized and placing them in the first bin they fit in, to reduce the number of solutions that are considered and avoid exhaustive solution space exploration.

Conventional bin-packing deployment techniques take a one-dimensional view of deployment problems by just focusing on a single deployment concern at a time. Example concerns include resource constraints, scheduling constraints, or fault-tolerance constraints. In production flight avionics systems, however, deployments must meet combinations of these concerns simultaneously.

Solution approach ⇒ Computer-assisted deployment optimization. This chapter describes and validates a method and tool called *ScatterD* that we developed to perform computer-assisted deployment optimization for flight avionics systems. The ScatterD model-driven engineering [19] deployment tool implements the *Scatter Deployment Algorithm* , which combines heuristic bin-packing with optimization algorithms, such as genetic algorithms [20] or particle swarm optimization techniques [21] that use evolutionary or bird flocking behavior to perform blackbox optimization. This chapter shows how flight avionics system developers have used ScatterD to automate the reduction of processors and network bandwidth in complex embedded system deployments.

3.3 Modern Embedded Flight Avionics Systems: A Case Study

Over the past 20 years, flight avionics systems have become increasingly sophisticated. Modern aircraft now depend heavily on software executing atop a complex embedded network for higher-level capabilities, such as more sophisticated flight control and advanced mission computing functions.

The increased weight of the embedded computing platforms required by a modern fighter aircraft incurs a multiplier effect [3], *e.g.*, roughly four pounds of cooling, power supply, and other supporting hardware are needed for each pound of processing hardware, reducing mission range, increasing fuel consumption, and impacting aircraft responsiveness.

To accommodate the increased amount of software required, avionics systems have moved from older federated computing architectures to integrated computing architectures that combine multiple software applications together on a single computing platform containing many software components.

The class of flight avionics system targeted by our work is a networked parallel message-passing architecture containing many computing nodes, as shown in Figure 3.2.

Each node is built from commercially available components packaged in hardened chassis to withstand extremes

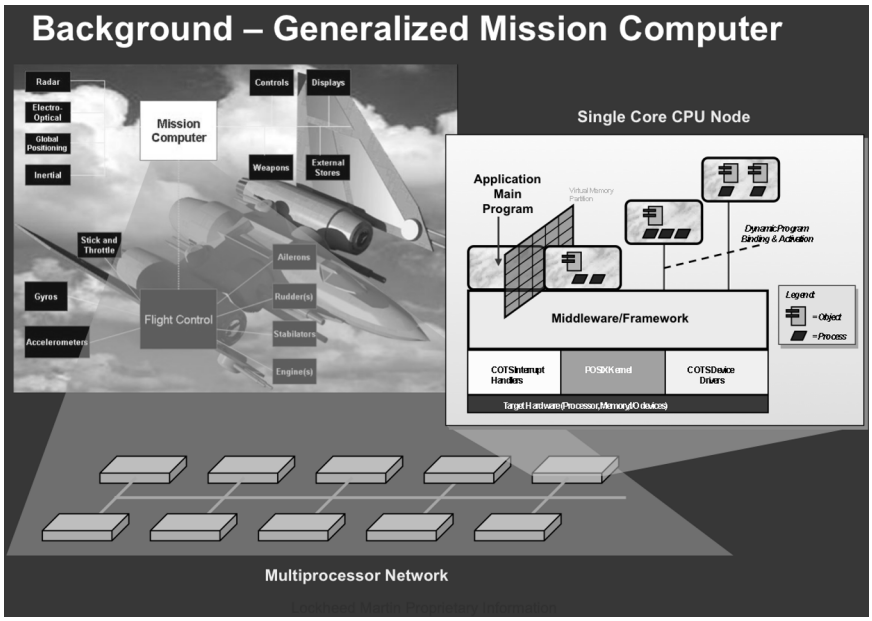

Figure 3.2: An Integrated Computing Architecture for Embedded Flight Avionics

of temperature, vibration, and acceleration.

At the individual node level, ARINC 653-compliant time and space partitioning separates the software applications into sets with compatible safety and security requirements. Inside a given time partition, the applications run within a hard real-time deadline scheduler that executes the applications at a variety of harmonic periods.

The integrated computing architecture shown in Figure 3.2 has benefits and challenges. Key benefits include better optimization of hardware resources and increased flexibility, which result in a smaller hardware footprint, lower energy use, decreased weight, and enhanced ability to add new software to the aircraft without updating the hardware. The key challenge, however, is increased system integration complexity. In particular, while the homogeneity of processors gives system designers a great deal of freedom allocating software applications to computing nodes, optimizing this allocation involves simultaneously balancing multiple competing resource demands.

For example, even if the processor demands of a pair of applications would allow them to share a platform, their respective I/O loads may be such that worst-case arrival rates would saturate the network bandwidth flowing into a single node. This problem is complicated for single-core processors used in current integrated computing architectures. Moreover, this problem is being exacerbated with the adoption and fielding of multi-core processors, where competition for shared resources expands to include internal buses, cache memory contents, and memory access bandwidth.

3.4 Deployment Optimization Challenges

While Section 3.3 describes many benefits of deployment optimization, developers of embedded flight avionics systems face a daunting series of conflicting constraints and optimization goals when determining how to deploy software to hardware. For example, it is hard to find a valid solution for a single deployment constraint, such as ensuring that all of software tasks can be scheduled to meet real-time deadlines, in isolation using conventional techniques, such as bin-packing. It is even harder, moreover, to find a valid solution when considering many deployment constraints, such as satisfying resource require-

ments of software tasks in addition to ensure schedulability. Optimizing the deployment topology of a system to minimize consumed network bandwidth or other dynamic properties is harder still since communication between software tasks must be taken into account, instead of simply considering each software task as an independent entity.

This section describes the challenges facing developers when attempting to create a deployment topology for a flight avionics system. The discussion below assumes a networked parallel message-passing architecture (such as the one described in Section 3.3). The goal is to minimize the number of required processors and the total network bandwidth resulting from communication between software tasks.

3.4.1 Challenge 1: Satisfying Rate-monotonic Scheduling Constraints Efficiently

In real-time systems, such as the embedded flight avionics case study from Section 3.3, either fixed priority scheduling algorithms, such as rate-monotonic (RM) scheduling, or dynamic priority scheduling algorithms, such as earliest-deadline-first (EDF), control the execution ordering of individual tasks on the processors. The deployment topology must ensure that the set of software components allocated to each processor are schedulable and will not miss real-time deadlines. Finding a deployment topology for a series of software components that ensures schedulability of all tasks is called "multiprocessor scheduling" and is NP-Hard [5].

A variety of algorithms, such as bin-packing algorithm variations, have been created to solve the multiprocessor scheduling problem. A key limitation of applying these algorithms to optimize deployments is that bin-packing does not allow developers to specify which deployment characteristics to optimize. For example, bin-packing does not allow developers to specify an objective function based on the overall network bandwidth consumed by a deployment. We describe how ScatterD ensures schedulability in Section 3.5.1 and allows for complex objective functions, such as network bandwidth reduction.

3.4.2 Challenge 2: Reducing the Complexity of Memory, Cost, and Other Resource Constraints

Processor execution time is not the only type of resource that must be managed while searching for a deployment topology. Hardware nodes often have other limited but critical resources, such as main memory or core cache, necessary for the set of software components it supports to function. Developers must ensure that the components deployed to a processor do not consume more resources than are present.

If each processor does not provide a sufficient amount of resources to support all tasks on the processor, a task will not execute properly, resulting in a failure. Moreover, since each processor used by a deployment has a financial cost associated with it, developers may need to adhere to a global budget, as well as scheduling constraints. We describe how ScatterD ensures that resources constraints are satisfied in Section 3.5.2.

3.4.3 Challenge 3: Satisfying Complex Dynamic Network Resource and Topology Constraints

Embedded flight avionics systems must often ensure that not only processor resource limitations are adhered to, but network resources (such as bandwidth) are not overconsumed. For example, catastrophic failure could occur if two critical real-time components communicating across a high-speed bus, such as a controller area network (CAN) bus, fail to send a required message due to network saturation.

The consumption of network resources is determined by the number of interconnected components that are not colocated on the same processor. For example, if two components are colocated on the same processor, they do not consume any network bandwidth.

Adding the consideration of network resources to deployment substantially increases the complexity of finding a software-to-hardware deployment topology mapping that meets requirements.

With real-time scheduling and resource constraints, the deployment of a component to a processor has a fixed resource consumption cost that can be calculated in isola-

tion of the other components.

The impact of the component's deployment on the network, however, cannot be calculated in isolation of the other components. The impact is determined by finding all other components that it communicates with, determining if they are colocated, and then calculating the bandwidth consumed by the interactions with those that are not colocated. We describe how ScatterD helps minimize the bandwidth required by a system deployment in Section 3.5.3.

3.5 ScatterD: A Deployment Optimization Tool to Minimize Bandwidth and Processor Resources

Heuristic bin-packing algorithms work well for multiprocessor scheduling and resource allocation. As discussed in Section 3.4, however, heuristic bin-packing is not effective for optimizing designs for certain system-wide properties, such as network bandwidth consumption, and hardware/software cost. *Metaheuristic* algorithms [20,21] are a promising approach to optimize system-wide properties that are not easily optimized with conventional bin-packing algorithms. These types of algorithms evolve a set of potential designs over a series of iterations using techniques, such as simulated evolution or bird flocking. At the end of the iterations, the best solution(s) that evolved out from the group is output as the result.

Although metaheuristic algorithms are powerful, they have historically been hard to apply to large-scale production embedded systems since they typically perform poorly on problems that are highly constrained and have few correct solutions. Applying simulated evolution and bird flocking behaviors for these types of problems tend to randomly mutate designs in ways that violate constraints. For example, using an evolutionary process to splice together two deployment topologies is likely to yield a new topology that is not real-time schedulable.

To overcome these limitations, this section presents ScatterD, which is a tool that utilizes a "hybrid" method that combines the two approaches so the benefits of each can be obtained with a single tool.

Below we explain how ScatterD integrates the ability of heuristic bin-packing algorithms to generate correct solutions to scheduling and resource constraints with the ability of metaheuristic algorithms to flexibly minimize network bandwidth and processor utilization and address the challenges in Section 3.4.

3.5.1 Satisfying Real-time Scheduling Constraints with ScatterD

ScatterD ensures that the numerous deployment constraints (such as the real-time schedulability constraints described in Challenge 1 from Section 3.4.1) are satisfied by using heuristic bin-packing to allocate software tasks to processors. Conventional bin-packing algorithms for multiprocessor scheduling are designed to take as input a series of items (*e.g.*, tasks or software components), the set of resources consumed by each item (*e.g.*, processor and memory), and the set of bins (*e.g.*, processors) and their capacities. The algorithm outputs an assignment of items to bins (*e.g.*, a mapping of software components to processors).

ScatterD ensures schedulability of the flight avionics system discussed in Section 3.3 by using response-time analysis. The response time resulting from allocating a software task of the avionics system to a processor is analyzed to determine if a software component can be scheduled on a given processor before allocating its associated item to a bin.

Before placing an item in a bin, ScatterD analyzes the response time that would result from allocating the software task to the given proessor. If the response time is fast enough to meet the real-time deadlines of the software task, the software task can be allocated to the processor. If not, then the item must be placed in another bin.

3.5.2 Satisfying Resource Constraints with ScatterD

To ensure that other resource constraints (such as memory requirements described in Challenge 2 from Section 3.4.2) of each software task are met, we specify a capacity for each bin that is defined by the amount of each computational resource provided by the corresponding processor in the avionics hardware platform. Simi-

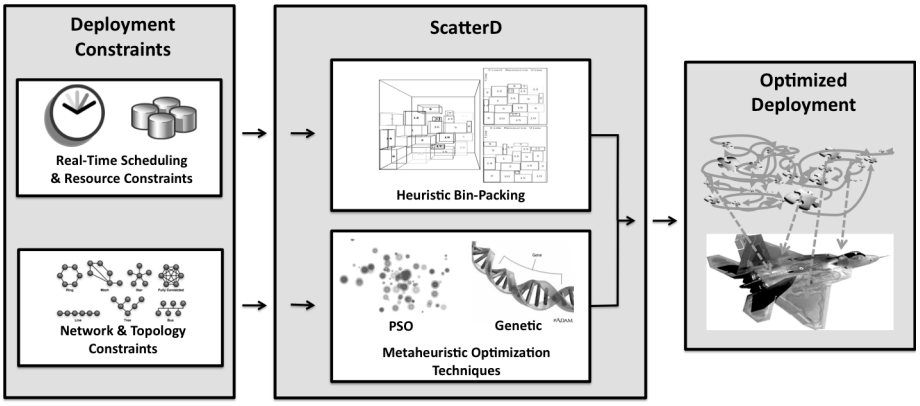

Figure 3.3: ScatterD Deployment Optimization Process

larly, the resource demands of each avionics software task define the resource consumption of each item. Before an item can be placed in a bin, ScatterD verifies that the total consumption of each resource utilized by the corresponding avionics software component and software components already placed on the processor does not exceed the resources provided.

3.5.3 Minimizing Network Bandwidth and Processor Utilization with ScatterD

To address deployment optimization issues (such as those raised in Challenge 3 from Section 3.4.3), ScatterD uses heuristic bin-packing to ensure that schedulability and resource constraints are met. If the heuristics are not altered, bin-packing will always yield the same solution for a given set of software tasks and processors. The number of processors utilized and the network bandwidth requirements will therefore not change from one execution of the bin-packing algorithm to another. In a vast deployment solution space associated with a large-scale flight avionics system, however, there may be many other deployments that substantially reduce the number of processors and network bandwidth required, while also satisfying all design constraints.

Metaheuristic algorithms, such as genetic algorithms and particle swarm optimization techniques, can be used to explore other areas of the deployment solution space and discover deployment topologies for avionic systems that meet user requirements, but which need fewer processors and less network bandwidth to operate. The problem, however, is that that the deployment solution space is vast and only a small percentage of potential deployments actually satisfy all avionics system design constraints. Since metaheuristic algorithms strive to reduce bandwidth and the number of required processors without directly accounting for design constraints, using these algorithms alone would result in the exploration of many invalid avionics deployment topologies.

To search for avionics deployment topologies with minimal processor and bandwidth requirements—while still ensuring that other design constraints are met—ScatterD uses metaheuristic algorithms to *seed* the bin-packing algorithm. In particular, metaheuristic algorithms are used to search the deployment space and select a subset of the avionics software tasks that must be packed prior to the rest of the software tasks. By forcing an altered bin-packing order, new deployments with different bandwidth and processor requirements are generated. Since bin-packing is still the driving force behind allocating software tasks, design constraints have a higher probability of being satisfied.

As new valid avionics deployments are discovered, they are scored based on network bandwidth consumption and the number of processors they require in the underlying avionics hardware platform. Metaheuristic algorithms use the scores of these deployments to determine which new packing order would likely yield a more opti-

mized deployment. By using metaheuristic algorithms to search the design space—and then using bin-packing to allocate software tasks to processors—ScatterD can generate deployments that meet all design constraints while also minimizing network bandwidth consumption and reducing the number of required processors in the avionics platform, as shown in Figure 3.3.

3.6 Empirical Results

This section presents the results of configuring the ScatterD tool to combine two metaheuristic algorithms (particle swarm optimization and a genetic algorithm) with bin-packing to optimize the deployment of the embedded flight avionics system described in Section 3.3. We applied these techniques to determine if (1) a deployment exists that increases processor utilization to the extent that legacy processors could be removed and (2) the overall network bandwidth requirements of the deployment were reduced due to colocating communicating software tasks on a common processor.

The first experiment examined applying ScatterD to minimize the number of processors in the legacy flight avionics system deployment, which originally consisted of software tasks deployed to 14 processors. Applying ScatterD with particle swarm optimization techniques and genetic algorithms resulted in increased utilization of the processors, reducing the number of processors needed to deploy the software to eight in both cases. The remaining six processors could then be removed from the deployment without affecting system performance, resulting in the 42.8% reduction shown in Figure 3.4.

The ScatterD tool was also applied to minimize the bandwidth consumed due to communication by software tasks allocated to different processors in the legacy avionics system described in Section 3.3. Reducing the bandwidth requirements of the system leads to more efficient, faster communication while also reducing power consumption. The legacy deployment consumed $1.83 \cdot 10^{08}$ bytes of bandwidth. Both versions of the ScatterD tool yielded a deployment that reduced bandwidth by $4.39 \cdot 10^{07}$ or 24%, as shown in Figure 3.4.

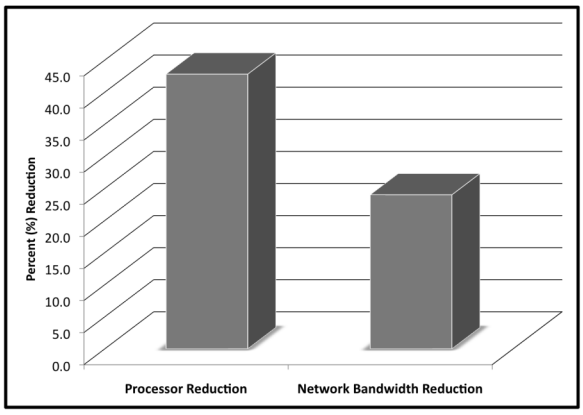

Figure 3.4: Network Bandwidth and Processor Reduction in Optimized Deployment

3.7 Related Work

A number of prior research efforts are related to the system-wide deployment optimization problem presented in this chapter. This section provides a taxonomy of these related works and compares and contrasts them to ScatterD. The related works are categorized based on the type of algorithm used in the deployment process.

Multi-processor scheduling. Bin-packing algorithms have been successfully applied to the NP-Hard problem of multi-processor scheduling [5]. Multi-processor scheduling requires finding an assignment of real-time software tasks to hardware processors, such that no tasks miss any deadlines. A number of bin-packing modifications are used to optimize the assignment of the tasks to use as few processors as possible [5,8,22–24]. The chief issue of using these existing bin-packing algorithms for spatial deployment optimization to minimize network bandwidth is that they focus on minimizing total processors used.

Kirovski et al. [25] have developed heuristic techniques for assigning tasks to processors in resource constrained systems to minimize system-wide power consumption. Their technique optimizes a combination of variations in processor power consumption and voltage scaling. These techniques, however, do not account for network communication in the power optimization process.

Hardware/software co-synthesis. Hardware/Software co-synthesis research has yielded techniques for deter-

mining the number of processing units, task scheduling, and other parameters to optimize systems for power consumption while meeting hard real-time constraints. Dick et al. [26, 27], have used a genetic algorithm for the co-synthesis problem. As with other single-chip work, however, this research is directed towards systems that are not spatially separated from one another.

Client/Server Task Partitioning for Power Optimization. Network power consumption and processor power consumption have both been considered in work on partitioning client/server tasks for mobile computing [28–30]. In this research, the goal is to determine how to partition tasks between a server and mobile device to minimize power drain on the device. This work, however, is focused only on how network bandwidth and power is saved by moving processing responsibilities between a single client and server.

Chapter 4

Model Driven Configuration Derivation

4.1 Challenge Overview

This chapter describes the need for model-driven tools that capture the myriad of DRE system design constraints to simplify DRE system configuration derivation. We motivate the need for tools to facilitate configuration by providing an example of a satellite imaging system. We demonstrate how the model-driven tool can be applied to aid developers in defining DRE system configuration scenarios and to automatically derive valid configurations.

4.2 Introduction

Distributed real-time embedded (DRE) systems (such as avionics systems, satellite imaging systems, smart cars, and intelligent transportation systems) are subject to stringent requirements and quality of service (QoS) constraints. For example, timing constraints require that tasks be completed by real-time deadlines. Likewise, rigorous QoS demands (such as dependability and security), may require a system to recover and remain active in the face of multiple failures [31]. In addition, DRE systems must satisfy domain-specific constraints, such as the need for power management in embedded systems. To cope with these complex issues, applications for DRE systems have traditionally been built from scratch using specialized, project-specific software components that are tightly coupled with specialized hardware components [32].

New DRE systems are increasingly being developed by configuring applications from multiple layers of commercial-off-the-shelf (COTS) hardware, operating systems, and middleware components resulting in reduced development cycle-time and cost [33]. These types of DRE systems require the integration of 100's-1,000's of software components that provide distinct functionality, such as I/O, data manipulation, and data transfer. This functionality must work in concert with other software and hardware components to accomplish mission-critical tasks, such as self-stabilization, error notification, and power management. The software configuration of a DRE system thus directly impacts its performance, cost, and quality.

Traditionally, DRE systems have been built completely in-house from scratch. These design techniques are based on in-house proprietary construction techniques and are not designed to handle the complexities of configuring systems from existing components [34]. The new generation of configuration-based approaches construct DRE systems by determining which combination of hardware/software components provide the requisite QoS [35–37]. In addition, the combined purchase cost of the components cannot exceed a predefined amount, referred to as the project budget.

A DRE system can be split into a software configuration and a hardware configuration. Valid software configuration must meet all real-time constraints, such as minimum latency and maximum throughput, provide required functionality, meet software architecture constraints, such as interface compatibility, and also satisfy all domain-specific design constraints, such as minimum power consumption Moreover, the cost of the software configuration must not exceed the available budget for purchasing software components. Similarly, the hardware configuration must meet all constraints without exceeding the available

hardware component budget. At the same time, the hardware and software configuration must be aligned so that the hardware configuration provides sufficient resources, such as RAM, for the chosen software configuration. Additional constraints may also be present based on the type and application of the DRE system being configured.

Often, there are multiple COTS components that can meet each functional requirement for a DRE system. Each individual COTS component differs in QoS provided, the amounts/types of computational resources required, and the purchase cost. Creating and maintaining error-free COTS configurations is hard due to the large number of complex configuration rules and QoS requirements. The complexity associated with examining the tradeoffs of choosing between 100's to 1,000's of COTS components makes it hard to determine a configuration that satisfies all constraints and is not needlessly expensive or resource intensive.

Solution approach-> Model-driven automated configuration techniques. This chapter presents techniques and tools that leverage the Model Driven Architecture (MDA) paradigm [38], which is a design approach for specifying system configuration constraints with platform-independent models (PIMs). Each PIM can be used as a blueprint for constructing platform-specific models (PSM)s [39]. In this chapter, MDA is utilized to construct modeling tools that can be used to create model instances of potential DRE system configurations. These tools are then applied in a motivating example to determine valid DRE system configurations that fit budget limits and ensure consistency between hardware and software component selections.

To simplify the DRE system configuration process, designers can use MDA to construct modeling tools that visualize COTS component options, verify configuration validity, and compare potential DRE system configurations. In particular, PSMs can be used to determine DRE system configurations that meet budgetary constraints by representing component selections in modeling environments. Modeling tools that utilize these environments provide a domain-centric way to experiment with and explore potential system configurations. Moreover, by constructing PSMs with the aid of modeling tools, many complex constraints associated with DRE system configuration can be enforced automatically, thereby preventing designers from constructing PSMs that violate system configuration rules.

After a PSM instance of a DRE system configuration is constructed, it can be used as a blueprint to construct a DRE system that meets all design constraints specified within the metamodel [40]. As DRE system requirements evolve and additional constraints are introduced, the metamodel can be modified and new PSMs constructed. Systems that are constructed using these PSMs can be adapted to handle additional constraints and requirements more readily than those developed manually using third-generation languages, such as C++, Java, or C#.

4.3 Large-scale DRE System Configuration Challenges

This section describes some key constraints that DRE systems must adhere to, summarizes the challenges that make determining configurations hard, and provides a survey of current techniques and methodologies for DRE system configuration. A DRE system configuration consists of a valid hardware configuration and valid software configuration in which the computational resource needs of the software configuration are provided by the computational resources produced by the hardware configuration. DRE system software and hardware components often have complex interdependencies on the consumption and production of resources (such as processor utilization, memory usage, and power consumption). If the resource requirements of the software configuration exceed the resource production of the hardware configuration, a DRE system will not function correctly and will thus be invalid.

4.3.1 Challenge 1: Resource Interdependencies

Hardware components provide the computational resources that software components require to function. If the hardware does not provide an adequate amount of each computational resource, some software components cannot function. An overabundance of resources indicates that some hardware components have been purchased unnecessarily, wasting funds that could have been spent to buy superior software components or set aside for future

projects.

Figure 4.1 shows the configuration options of a satellite imaging system. This DRE system consists of an image processing algorithm and software that defines image resolution capabilities. There are multiple components that could be used to meet each functional requirement, each of which provides a different level of service.

For example, there are three options for the image resolution component. The high-resolution option offers the highest level of service, but also requires dramatically more RAM and CPU to function than the medium or low-resolution options. If the resource amounts required by the high-resolution option are not supplied, then the component cannot function, preventing the system from functioning correctly. If RAM or CPU resources are scarce the medium or low-resolution option should be chosen.

4.3.2 Challenge 2: Component Resource Requirements Differ

Each software component requires computational resources to function. These resource requirements differ between components. Often, components offering higher levels of service require larger amounts of resources and/or cost more to purchase. Designers must therefore consider the additional resulting resource requirements when determining if a component can be included in a system configuration.

For example, the satellite system shown in Figure 4.1 has three options for the image resolution software component, each of which provides a different level of performance. If resources were abundant, the system with the best performance would result from selecting the high-resolution component. In most DRE systems, such as satellite systems, resources are scarce and cannot be augmented without great cost and effort. While the performance of the low-resolution component is less than that of the high-resolution component, it requires a fraction of the computational resources. If any resource requirements are not satisfied, the system configuration is considered invalid. A valid configuration is thus more likely to exist by selecting the low-resolution component.

4.3.3 Challenge 3: Selecting Between Differing Levels of Service

Software components provide differing levels of service. For example, a designer may have to choose between three different software components that differ in speed and throughput. In some cases, a specific level of service may be required, prohibiting the use of certain components.

Continuing with the satellite configuration example shown in Figure 4.1, an additional functional constraint may require that a minimum of medium image resolution. Inclusion of the low-resolution component would therefore invalidate the overall system configuration. Assuming sufficient resources for only the medium and low-resolution components, the only component that satisfies all constraints is the medium image resolution option.

Moreover, the inclusion of a component in a configuration may prohibit or require the use one or more other components. Certain software components may have compatibility problems with other components. For example, each of the image resolution components may be a product of separate vendors. As a result, the high and medium-resolution components may be compatible with any image processing component, whereas the low-resolution component may only be compatible with image processing components made by the same vendor. These compatibility issues add another level of difficulty to determining valid DRE system configurations.

4.3.4 Challenge 4: Configuration Cannot Exceed Project Budget

Each component has an associated purchase cost. The combined purchase cost of the components included in the configuration must not exceed the project budget. It is therefore possible for the inclusion of a component to invalidate the configuration if its additional purchase cost exceeds the project budget regardless of computational resources existing to support the component. Moreover, if two systems have roughly the same resource requirements and performance the system that carries a smaller purchase cost is considered superior.

Another challenge of meeting budgetary constraints is determining the best way to allocate the budget between hardware purchases and software purchases. Despite the

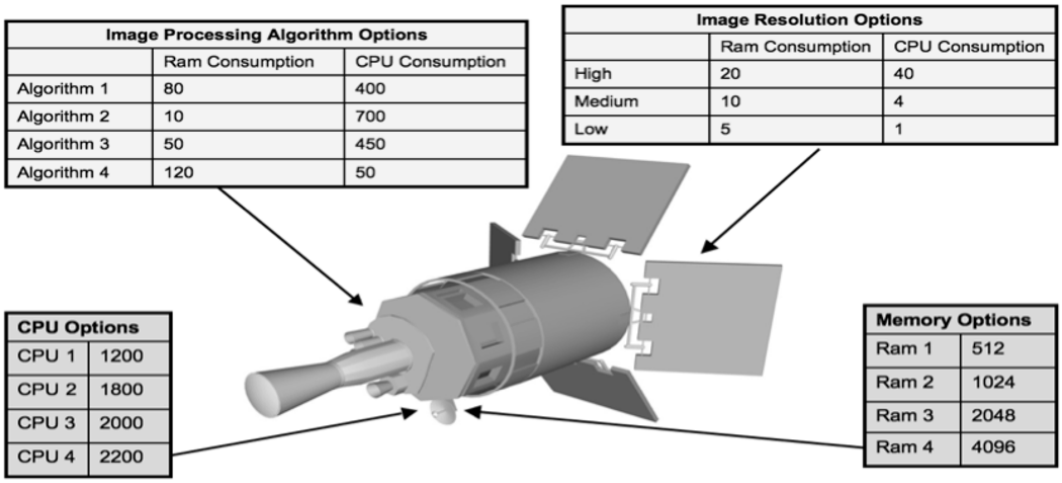

Figure 4.1: Configuration Options of a Satellite Imaging System

presence of complex resource interdependencies, most techniques require that the selection of the software configuration and hardware configuration occur separately. For example, the hardware configuration could be determined prior to the software configuration so that the resource availability of the system is known prior to solving for a valid software configuration. Conversely, the software configuration could be determined initially so that the resource requirements of the system are known prior to solving for the hardware configuration.

To solve for a hardware or software configuration individually, the total project budget must be divided into a software budget for purchasing software components and a hardware budget for purchasing hardware components. Dividing the budget evenly between the two configuration problems may not produce a valid configuration. Uneven budget divisions, however, may result in valid system configurations. Multiple budget divisions must therefore be examined.

4.3.5 Challenge 5: Exponential Configuration Space

Large-scale DRE systems require hundreds of components to function. For each component there may be many components available for inclusion in the final system configuration. Due to the complex resource interdependencies, budgetary constraints, and functional constraints it is hard to determine if including a single component will invalidate the system configuration. This problem is exacerbated enormously if designers are faced with the tasks of choosing from 1,000's of available components. Even automated techniques require years or more to examine all possible system configurations for such problems. Large-scale DRE systems often also consist of many software and hardware components with multiple options for each component, resulting in an exponential number of potential configurations. Due to the multiple functional, real-time, and resource constraints discussed earlier, arbitrarily selecting components for a configuration is ineffective. For example, if there are 100 components to choose from then there are $1.2676506x10^{30}$ unique potential system configurations, the vast majority of which are invalid configurations. The huge magnitude of the solution space prohibits the use of manual techniques. Automated techniques, such as Constraint Logic Programming (CLP), use Constraint Satisfaction Problems (CSPs) to represent system configuration problems [41,42]. These techniques are capable of determining optimal solutions for small-scale system configurations but require the examination of all potential system configurations. Techniques utilizing CSPs are ideal, however, for system configuration prob-

lems involving a small number of components as they can determine an optimal configuration-should one exist-in a short amount of time.

The exhaustive nature of conventional CSP-based techniques, however, renders them ineffective for large-scale DRE system configuration. Without tools to aid in large-scale DRE system configuration, it is hard for designers to determine any valid large-scale system configuration. Even if a valid configuration is determined, other valid system configurations may exist with vastly superior performance and dramatically less financial cost. Moreover, with constant development of additional technologies, legacy technologies becoming unavailable, and design objectives constantly in flux, valid configurations can quickly become invalid, requiring that new configurations be discovered rapidly. It is thus imperative that advanced design techniques, utilizing MDA, are developed to enhance and validate large-scale DRE system configurations.

Subsequent sections of this chapter demonstrate how MDA can be utilized to mitigate many difficulties of DRE system configuration that result from the challenges described in this section.

4.4 Applying MDA to Derive System Configurations

System configuration involves numerous challenges, as described in the previous section. Constructing MDA tools can help to address these challenges. The process of creating a modeling tool for determining valid DRE system configurations is shown in Figure 4.2.

Figure 4.2. Creation Process for a DRE System Configuration Modeling Tool. This process is divided into four steps:

1. Devise a configuration language for capturing complex configuration rules,
2. Implement a tool for manipulating instances of configurations,
3. Construct a metamodel to formally define the modeling language used by the tool, and
4. Analyze and interpret model instances to determine a solution.

By following this methodology, robust modeling tools can be constructed and utilized to facilitate the configuration of DRE systems. The remainder of this section describes this process in detail.

4.4.1 Devising a Configuration Language

DRE system configuration requires the satisfaction of multiple constraints, such as resource and functional constraints. The complexity of accounting for such a large number of configuration rules makes manual DRE system configuration hard. Configuration languages exist, however, that can be utilized to represent and enforce such constraints. By selecting a configuration language that captures system configuration rules, the complexity of determining valid system configurations can be reduced significantly.

Feature models are a modeling technique that have been used to model Software Product Lines (SPLs) [43], as well as system configuration problems. SPLs consist of interchangeable components that can be swapped to alter system functionality. Czarnecki et al. use feature models to describe the configuration options of systems [44]. Feature models are represented using tree structures with lines (representing configuration constraints) connecting candidate components for inclusion in an SPL, known as features. The feature model uses configuration constraints to depict the effects that selecting one or more features has on the validity of selecting other features. The feature model serves as a mechanism to determine if the inclusion of a feature will result in an invalid system configuration.

Czarnecki et al. also present staged-configuration, an incremental technique for manually determining valid feature selections. This work, however, cannot be directly applied to the configuration of large-scale DRE system configuration because it doesn't guarantee correctness or provide a way of handling resource constraints. Moreover, it takes a prohibitive amount of time to determine valid system configurations since staged-configuration is not automated.

Benavides et al. introduce the extended feature model, an augmented feature model with the ability to more articulately define features and represent additional constraints [41]. Additional descriptive information, called attributes, can be added to define one or more parameters of each feature. For example, the resource consumption

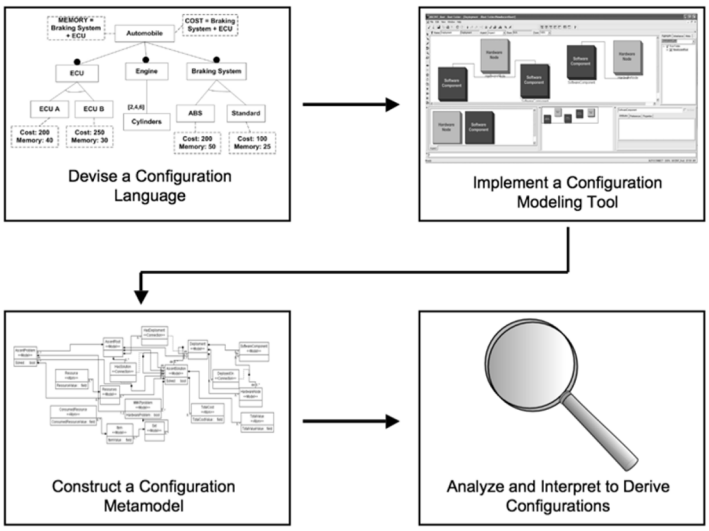

Figure 4.2: Creation Process for a DRE System Configuration Modeling Tool

and cost of a feature could be defined by adding attributes to the feature. Each attribute lists the type of resource and the amount consumed or provided by the feature. Additional constraints can be defined by adding extra-functional features. Extra-functional features define rules that dictate the validity of sets of attributes. For example, an extra-functional feature may require that the total cost of a set of features representing components is less than that of a feature that defines the budget. Any valid feature selection would thus satisfy the constraint that the collective cost of the components is less than the total project budget.

4.4.2 Implementing a Modeling Tool

Designers using manual techniques often unknowingly construct invalid system configurations. Even if an existing valid system configuration is known, the introduction of a single component can violate one or more of these constraints, thereby invalidating the entire configuration. Modeling tools allow designers to manipulate problem entities and compare potential solutions in an environment that ensures various design rules are enforced that are not accounted for in current third-generation programming languages, such as Java and C++. Automated correctness checking allows designers to focus on other problem dimensions, such as performance optimization or minimization of computational resource requirements.

One example of a modeling tool is the Generic Modeling Environment (GME) composing domain-specific design environments [45]. GME is modeling platform for building MDA based tools that can then be used to create model instances. The two principles components of GME are GMeta and GModel, which work together to provide this functionality. GMeta is a graphical tool for constructing metamodels, which are discussed in the following section. GModel is a graphical editor for constructing model instances that adhere to the configuration rules.

For example, a user could construct a system configuration model that consists of hardware and software components as shown in Figure 3 4.3. By using the graphical editor, the user can manually create multiple system configuration instances. If the user attempts to include a component that violates a configuration rule, GModel will disallow the inclusion of the component and explain the violation. Since GModel is responsible for enforcing all constraints, the designer can rapidly create and experiment with various models without the overhead of monitoring for constraint violations.

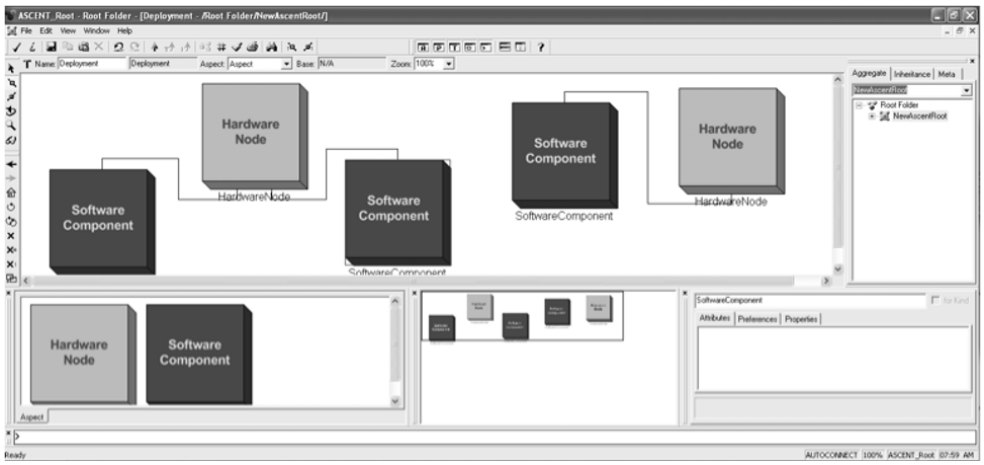

Figure 4.3: GME Model of DRE System Configuration

4.4.3 Constructing a Metamodel

Metamodels are used to formally define the rules that are enforced by modeling tools [46]. This collection of rules governs the entities, relationships and constraints of model instances constructed. After constructing a metamodel, users can define modeling tools that are capable of creating model instances that enforce the rules and constraints defined by the metamodel.

Most nontrivial problems require multiple modeling entities, types of relationships between entities, and complex constraints. As a result, constructing metamodels can be a confusing, arduous task. Fortunately, metamodeling tools exist that provide a clear and simple procedure for creating metamodels. Tools for generating metamodels provide several advantages over defining them manually. For example, metamodeling tools can prevent defining rules, such as defining nameless entities, that are contradictory or inappropriate. Likewise, by using a metamodeling tool, metamodels can easily be augmented or altered should the domain or other problem parameters change.

Moreover, the same complexities inherent to creating PSMs are also present in the construction of metamodels, and often amplified by the additional abstraction required for their creation. Metamodeling tools use an existing language that defines the rules for creating metamodels, thereby enforcing the complex constraints and facilitating quick, accurate metamodel design.

To create a metamodel for describing system configuration the entities that are involved in DRE system configuration must first be defined. For example, at the most basic level, DRE system configuration consists of hardware and software components. The manner in which these entities interact must then be defined. For example, it is specified that hardware components provide computational resources and that software components consume computational resources.

Also, a way is needed to define the constraints that must be maintained as these entities interact for a system configuration to be valid. For example, it may be specified that a software component that interacts with a hardware component must be provided with sufficient computational resources to function by the hardware component.

After all the necessary entities for the modeling tool are created the rules that govern the relationships of these entities must be defined. For example, the relationship between hardware nodes and software components in which the software components consume resources of the hardware nodes must be defined. Before we can do this, however, an attribute must be defined that specifies the resource production values of the hardware nodes and the resource consumption values of the software nodes. Once attribute has been defined and associated it with a class, we can include the attribute in the relationship definition.

A relationship between two model entities is defined by adding a connection to the metamodel. The connec-

tion specifies the rules for connecting entities in the resulting PSM. Within the connection, we can define additional constraints that must be satisfied for two classes to be connected. For example, for a software component to be connected to a hardware node the resource consumption attribute of the software component can not exceed the attribute of the hardware node that defines the amount of resource production.

GME provides GMeta, a graphical tool for constructing metamodels. GMeta divides metamodel design into four separate sub-metamodels: the Class Diagram, Visualization, Constraints, and Attributes. The Class Diagram defines the entities within the model, known as models, atoms, and first class objects as well as the connections that can be made between them. The Visualization sub-metamodel defines different aspects, or filters, for viewing only certain entities within a model instance. For example, if defining a metamodel for a finite state machine, an aspect could be defined in the Visualization sub-metamodel that would only display accepting states in a finite state machine model instance.

The Constraints sub-metamodel allows the application of Object Constraint Language (OCL) [47] constraints to metamodel entities. Continuing with the finite state machine metamodel example, a constraint could be defined that only a single starting state may exist in the model. To do this, users would add a constraint in the Constraints sub-metamodel, add the appropriate OCL code to define the constraint, and then connect it to the entity to which it applies. Finally, the Attributes sub-metamodel allows additional data, known as attributes, to be defined and associated with other metamodel entities defined in the Class Diagram.

After the metamodel has been constructed using GMeta, the interpreter must be run to convert the metamodel into a GME paradigm. This paradigm can then be loaded with GME and used to created models that adhere to the rules defined within the metamodel. User may then create model instances with the assurance that the design rules and domain specific constraints defined within the metamodel are satisfied. If at any point the domain or design constraints of the model change, the metamodel can be reloaded, altered and interpreted again to change the GME paradigm appropriately. As a result, designers can easily examine scenarios in which constraints differ, giving a broader overview of the design space.

4.4.4 Analyzing and Interpreting Model Instances

After a configuration language is determined, a modeling tool implemented, and a metamodel constructed, designers can rapidly construct model instances of valid DRE system configurations. There is no guarantee, however, that the configurations constructed with these tools are optimal. For example, while a configuration instance may be constructed that does not violate any design constraints, other configurations may exist that provide higher QoS, have a lower cost, or consume fewer resources. Many automated techniques, however, exist for determining system configurations that optimize these attributes.

Benavides et al. provide a methodology for mapping the extended feature models described earlier onto constraint satisfaction problems (CSPs) [41]. A CSP is a set of variables with multiple constraints that define the values that the variables can take. Attributes and extra-functional features, such as a project budget feature, are maintained in the mapping. As a result, solutions that satisfy all extra-functional features and basic functional constraints can be found automatically with the use of commercial CSP solvers.

Moreover, these solvers can be configured to optimize one or more attributes, such as the minimization of cost. Additionally, these techniques require the examination of all potential solutions, resulting in a system configuration that is not only valid, but also optimal. Benavides et al. present empirical results showing that CSPs made from feature models of 23 features require less than 1,800 milliseconds to solve.

While extended feature models and their associated automated techniques for deriving valid configurations by converting them to CSPs can account for resource and budget constraints, the process is not appropriate for large-scale DRE system configuration problems. The exhaustive nature of CSP solvers often require that all potential solutions to a problem are examined. Since the number of potential system configurations is exponential in regards to the number of potential components, the solution space is far too vast for the use of exhaustive techniques as they would require a prohibitive amount of time to determine a solution.

To circumvent the unrealistic time requirements of exhaustive search algorithms, White et al. have examined

approximation techniques for determining valid feature selections that satisfy multiple resource constraints [48]. Approximation techniques do not require the examination of all potential configurations, allowing solutions to be determined with much greater speed. While the solutions are not guaranteed to be optimal, they are often optimal or extremely near optimal. White et al. present Filtered Cartesian Flattening (FCF), an approximation technique for determining valid feature selections.

FCF converts extended feature models into Multiple-choice Multi-dimensional Knapsack Problems (MMKP). MMKP problems, as described by Akbar et al. are an extension of the Knapsack Problem (KP), Multiple-Choice Knapsack Problem (MCKP) and Multi-Dimensional Knapsack Problem (MDKP) [49]. Akbar et al. provide multiple heuristic algorithms, such as I-HEU and M-HEU for rapidly determining near optimal solutions to MMKP Problems.

With FCF, approximation occurs in two separate steps. First, all potential configurations are not represented in the MMKP problems. For example, if there is an exclusive-or relationship between multiple features, then only a subset of the potentially valid relationships may be included in the MMKP problem. This pruning technique is instrumental in restricting problem size so that solving techniques can complete rapidly.

Second, heuristic algorithms, such as M-HEU can be used to determine a near-optimal system configuration. M-HEU is a heuristic algorithm that does not examine all potential solutions to an MMKP problem, resulting in faster solve time, thus allowing the examination of considerably larger problems. Due to these two approximation steps, FCF can be used for problems of considerably larger size compared to methods utilizing CSPs. This scalability is shown in Figure 4.4 in which a feature model with 10,000 features is examined with 90% of the solutions resulting in better than 90% optimality.

While FCF is capable of determining valid large-scale DRE system configurations, it still makes many assumptions that may not be readily known by system designers. For example, FCF requires that the project budget allocation for purchasing hardware and the project budget allocation for purchasing software components be known ahead of time. The best way to split the project budget between hardware and software purchases, however, is dictated by the configuration problem being solved.

For example, if all of the hardware components is cheap and provide huge amounts of resources while the software components are expensive, it would not make sense to devote half of the project budget to hardware and half to software. A better system configuration may result from devoting 1% of the budget to hardware and 99% to software.

The Allocation baSed Configuration ExploratioN Technique (ASCENT) presented by White et al. is capable of determining valid system configurations while also providing DRE system designers with favorable ways to divide the project budget [50]. ASCENT takes an MMKP hardware problem, MMKP software problem and a project budget amount as input. Due to the speed and performance provided by the M-HEU algorithm, ASCENT can examine many different budget allocations for the same configuration problem. ASCENT has been used for configuration problems with 1000's of features and is over 98% optimal for problems of this magnitude, making it an ideal technique for large-scale DRE system configuration.

To take advantage of these techniques, however, model instances must be converted into a form that these techniques can utilize. Interpreters are capable of parsing model instances and creating XML, source code, or other output for use with external programmatic methods. For example, GME model instances can easily be adapted to be parsed with Builder Object Network (BON2) interpreters. These interpreters are capable of examining all entities included in a model instance and converting them into C++ source code, thus allowing the application of automated analysis techniques, such as the use of CSP solvers or ASCENT [41,50].

4.5 Case Study

The background section discussed the challenges of DRE system configuration. For problems of non-trivial size, these complexities proved too hard to overcome without the use of programmatic techniques. Section 4.4.1 describes how configuration languages can be utilized to represent many of the constraints associated with DRE system configuration. That section also described how modeling tools can enforce complex design rules. Section 4.4.3 described the construction of a metamodel to

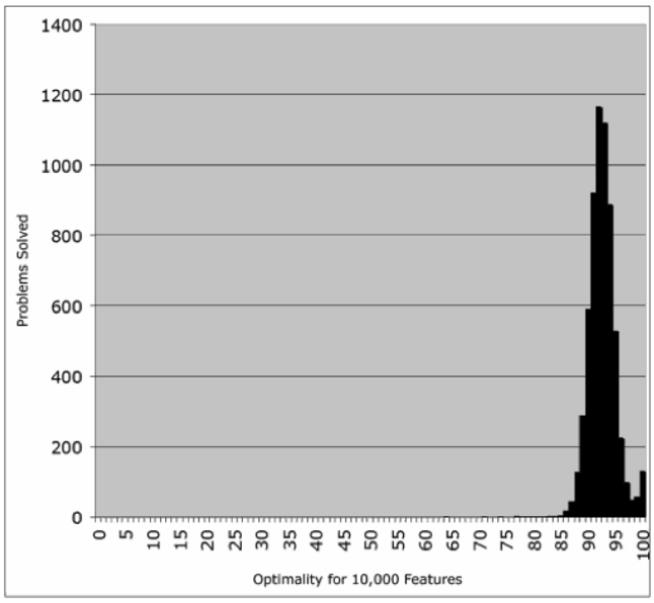

Figure 4.4: FCF Optimality with 10,000 Features

formalize the constraints to be enforced in the modeling tool. Section 4.4.4 introduced several automated techniques for determining valid DRE system configurations, such as ASCENT, that provide additional design space information, such as how to allocate a project budget, which is extremely valuable to designers. This section describes the process of creating the Ascent Modeling Platform (AMP) to allow rapid DRE system configuration, while also addressing the challenges described in the background section. The target workflow of AMP is shown in Figure 4.5.

4.5.1 Designing a MDA Configuration Language for DRE Systems

ASCENT was originally implemented programmatically in Java, so constructing an entire configuration problem (including external resources, constraints, software components and hardware components along with their multiple unique resource requirements) required writing several hundred lines of complex code. As a result, the preparation time for a single configuration problem took a considerable amount of time and effort. Moreover, designers could not easily manipulate many of the problem parameters to examine "what if" scenarios. To address these limitations with ASCENT, Ascent Modeling Platform (AMP) tool was constructed that could be used to construct DRE system configuration problems for analysis with ASCENT.

Implementing a Modeling Tool

GME was selected to model DRE system configuration and used this paradigm to experiment with AMP. The following benefits were observed as a result of using GME to construct the AMP modeling tool for DRE system configuration:

- Visualizes complex configuration rules. AMP provides a visual representation of the hardware and software components making it significantly easier to grasp the problem, especially to users with limited experience in DRE system configuration.
- Allows manipulation of configuration instances. In addition to visually representing the problem, by using AMP designers are able to quickly and easily change configuration details (budget, constraints,

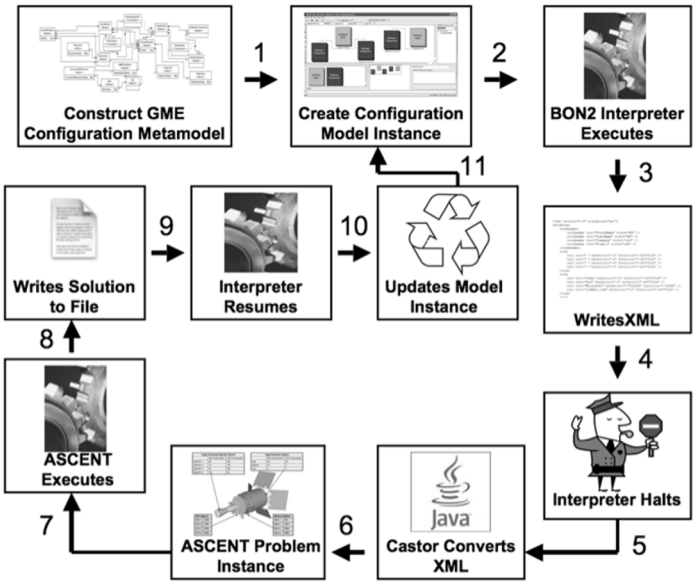

Figure 4.5: AMP Workflow Diagram

components, resource requirements etc.) makes the analysis much more powerful.

- Provides generational analysis. Models produced with AMP may be fed a previous solution as input, enabling designers to examine possible upgrade paths for the next budget cycle. These upgrade paths can be tracked for multiple generations, meaning that the analysis can determine the best long-term solutions. This capability was not previously available with ASCENT and would have been considerably harder to implement without the use of GME.
- Can easily be extended. It is simple to add additional models and constraints to the existing AMP metamodel. As DRE system configuration domain specific constraints are introduced, the AMP metamodel can be altered to enforce these additional constraints in subsequent model instances. Since most DRE system configuration problems only slightly differ, existing metamodels can be reused and augmented.
- Simplifies problem creation. AMP provides a drag and drop interface that allows users to create problem instances instead of writing 300+ required lines of complex java code. The advantages of using a simple graphical user interface are (1) designers do not have to take the time to type the large amount of code that would be required and (2) in the process of typing this large amount of code designers will likely make mistakes. While the compiler may catch many of these mistakes, it is also likely domain specific constraints that the compiler may overlook will be inadvertently violated. Since GME enforces the design rules defined within the metamodel, it is not possible for the designers using AMP to unknowingly make such a mistake while constructing a problem instance.

To expand the analytical capabilities of ASCENT, GME was utilized to provide an easily configurable, visual representation of the problem via the AMP tool. Using these new features, it is possible to see a broader, clearer picture of the total design process as well as the global effects of even minor design decisions.

Constructing a Metamodel

A metamodel is created for DRE system configuration using MetaGME. Figure 4.6 shows the Class Diagram portion of the AMP metamodel. The root model is labeled as AscentRoot and contains two models: AscentProblem and AscentSolution. The configuration problems are defined within AscentProblem. The configuration determined by interpreting the AscentProblem model and applying the ASCENT technique is represented as the AscentSolution.

Within the AscentProblem, there is MMKPproblem models and a Resources model. The MMKPproblems are used to represent the components available for inclusion in the configuration. Also included in the MMKPproblem is a boolean attribute for setting whether or not an MMKPproblem is a hardware problem. A constraint is also defined that requires the definition of two MMKPproblems, one of which contains the hardware components while the other represents the software components.

The components shown in Figure 4.6 contain the resource amounts that they consume or produce, based on whether they are members of a hardware MMKP problem or a software MMKP problem. The common resources model contains the Resource atoms, which represents the external resources of the problem that are common to both the hardware and software MMKPproblems, such as available project budget and power. The AscentSolution model contains a Deployment model, as well as atoms that represent the total cost and total value of the configuration determined by analyzing the AscentProblem. The Deployment model contains SoftwareComponents that represent the software components, HardwareNodes that represent the hardware components, as well as a DeployedOn connection that is used to connect the software components with the hardware components on which they are deployed.

Analyzing and Interpreting

A BON2 interpreter was written in C++ to analyze model instances. This interpreter traverses the AscentRoot model and creates an XML representation of the models, atoms and connections contained within. An XML representation of the model instance is then written to a file. This XML file matches a previously defined schema for use with the Castor XML binding libraries, a set of libraries for demarshalling XML data into Java objects. The ASCENT technique is defined within a Java jar file called ASCENTGME.jar. Once the XML data is generated, the interpreter makes a system call to execute the ASCENTGME.jar, passing in the XML file as an argument. Within ASCENTGME.jar, several things happen. First, the XML file is demarshaled into Java objects. A Java class then uses these objects to create two complex MMKPProblem instances. These two problem instances, along with a total budget value, are passed to ASCENT as input.

When ASCENT executes it returns the best DRE system configuration determined, as well as the cost and value of the configuration. A First Fit Decreasing (FFD) Bin-packer then uses these solutions along with their resource requirements to determine a valid deployment. This deployment data, along with the total cost, total value, hardware solution and software solution, is then written to a configuration file. The interpreter, having halted until the system call to execute the jar file terminates, parses this configuration file. Using this data, the ASCENT solution and deployment are written back into the model, augmenting the model instance with the system configuration.

The system configurations created by ASCENT can be examined and analyzed by designers. Designers can change problem parameters, execute the interpreter once again, and examine the effects of the changes to the problem on the system configuration generated. This iterative process allows designers to rapidly examine multiple DRE system configuration design scenarios, resulting in substantially increased knowledge of the DRE system configuration design space.

Motivating Example

AMP can be applied to determine valid configuration for the satellite imaging system shown in Figure 4.1. Not only should the resulting configuration be valid, but should also maximize system value. For example, a satellite imaging system that produces high-resolution images has higher inherent value than an imaging system that can only produce low-resolution images. In addition, the collective cost of the hardware and software components of the system must not exceed the project budget.

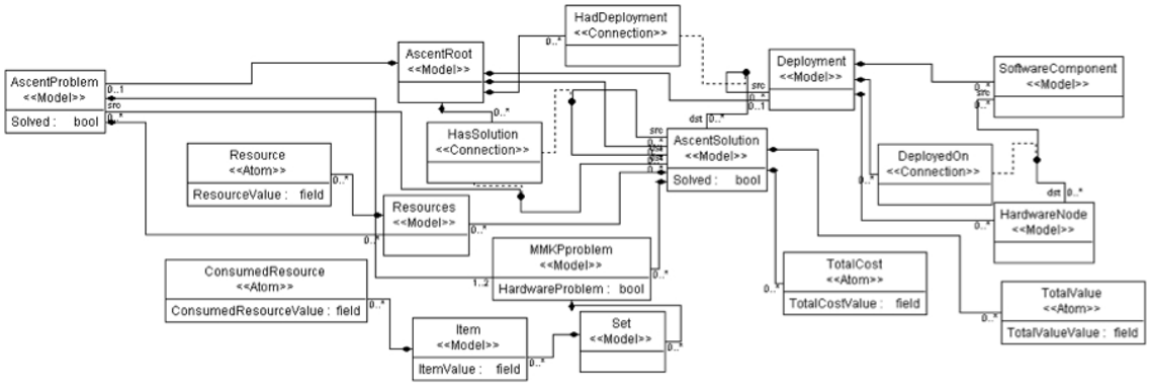

Figure 4.6: GME Class View Metamodel of ASCENT

To create an AMP problem instance representing the satellite imaging system described in Figure 4.1, several GME models must be created. First, an ASCENT Problem instance is added to the project. ASCENT Problem instances contain three models: A hardware MMKP Problem representing the hardware component options, a software MMKP Problem representing the software component options and Resources, representing the external resources, such as power and cost, that are consumed by both types of components.

A hardware MMKP problem instance is added to represent the hardware components. Within the hardware MMKP instance, Set model instances can be added. Each Set represents a set of hardware components that provide a common resource. For example, there are two types of hardware components, Memory and CPU available for consumption in the satellite system shown in Figure 4.1. To represent these two quantities, two Set instances are added with one instance representing CPU options and the other Memory Options.

Within each Set instance, the available options are represented as instances of Items. Item instances are added within the CPU option set to represent each of the available CPU options. Within each Item, a Resource instance is added to indicate the production amounts of the Item. For example, within the Item instance representing CPU 1, a Resource instance would be added that has a value of 1200, to represent the CPU production of the option. The instances representing the other CPU options and Memory options are constructed in the same manner, concluding the construction of the Hardware MMKP problem.

Now that the hardware options are represented, a software MMKP Problem instance must be prepared to represent the software component options. Continuing with the satellite imaging system shown in Figure 4.1, model representations of the software options for the Image Resolution component and Image Processing Algorithm must be constructed. Inside of the software MMKP instance, a Set instance is added for each set of component options, in this case a set for the Image Resolution component options and a set for the Image Processing Algorithm options. Similarly to the hardware MMKP problem, each software component option is represented as an Item. So within the Set instance of Image Resolution options, three Item models are added to represent the low-resolution, medium-resolution, and high-resolution options.

Unlike the hardware MMKP Problem, however, a value attribute must be assigned to represent the desirability of including the option. For example, it is more desirable to provide high-resolution image processing rather than medium-resolution or low-resolution image properties. Therefore, the value attribute high-resolution option would be set to a higher number than the other resolution options. Once the value is set, the resource consumption of each option can be set within each item representation of the software component options in the same manner as described for the hardware MMKP Problem. Once the hardware MMKP Problem, software MMKP Problem, and Resources are set, the model can be interpreted.

After the interpreter executes, a Deployment Plan

model instance is created. Within the Deployment Plan the selected hardware components and software components can be seen. In this case, the deployment plan consists of the CPU 1, RAM 1 hardware components and Algorithm 4, high-resolution software components. Further examination shows that both of the software components can be supported by the hardware components selected.

4.6 Related Work

Modeling tools can facilitate the process of DRE system configuration. The methodology described in this chapter has presented a process for constructing a modeling tool for system configuration from scratch. The model instances that are created using these modeling tools require that a user manually constructs model instances. For larger model instances, this may take a large amount of time. Therefore, techniques are needed that facilitate model instance construction from existing model instances.

Typically, system designers wish to construct a single model instance from data spread out over multiple model types. For example, a system designer may have a UML diagram for describing system software architecture, excel spreadsheets listing the cost and specifications of candidate components, and a Ptolemy model providing fault tolerance requirements. To manually extract this information form multiple models would be laborious.

Multi-modeling tools are applications that allow the manipulation of multiple PSMs defined by different metamodels. Multi-modeling tools could allow the automated aggregation of data from models of different types. In future work the use of multi-models to collect reliability, fault-tolerance, and performance data from multiple disparate models will be investigated and applied to the evaluation of model instances of DRE system configurations.

The migration of a model instance defined by a certain metamodel to a model instance defined by a different metamodel is known as a model transformation. Since these metamodels define different rules for constructing PSMs, the semantic meaning of the model that is migrated can be partially or entirely lost, resulting in an incomplete transformation. In future work, procedures to transform models while minimizing data loss will be researched.

Using these techniques, models that contain additional system configuration data, such as Ptolemy models, could be transformed into model instances that can be used in concert with AMP [51]. The Lockheed Martin Corporation is currently constructing NAOMI [52], a multi-modeling environment that can be utilized to aggregate data from multiple models of different types and perform complex multi-model transformations.

Chapter 5

Automated Hardware and Software Evolution Analysis

5.1 Challenge Overview

This chapter provides a motivation for the creation of automated techniques to evolve legacy DRE system configurations. We present a scenario in which a Smart Car must be evolved as new components become available to provide new functionality while continuing to satisfy strict resource requirements and QoS constraints. We demonstrate how automated hardware and software evolution can allow DRE systems to maintain usability as new technology becomes available.

5.2 Introduction

Current trends and challenges. Evolution accounts for a significant portion of software life-cycle costs [53]. An important type of software evolution involves enhancing existing software to meet new customer and market needs [54]. For example, in the automotive industry, each year the software and hardware from the previous year's model car must be upgraded to provide new capabilities, such as automated parking or wireless connectivity.

Software evolution analysis is the process of determining which software components and hardware components can be added to a system to implement new functionality while adhering to multiple resource constraints. This analysis involves several challenges, including (1) building an economic model to estimate the return on investment of new software features, (2) estimating the cost of implementing a software feature [55], and (3) selecting a new system configuration that maximizes the value of the features added while respecting resource constraints. This chapter examines software evolution analysis techniques that automatically determine valid system configurations that support required new capabilities without violating resource constraints.

In many domains, the cost/benefit analysis for software evolution is partially simplified by the availability of commercial-off-the-shelf (COTS) software/hardware components [56]. For example, automotive manufacturers know how much it costs to buy windshield wiper hardware/software components, as well as electronic control units (ECUs) with specific memory and processing capabilities/costs. Similar cost/benefit analysis can also be conducted for custom-developed (*i.e.*, non-COTS) software/hardware components [57].

Regardless of whether components are COTS or custom, however, determining the optimal subset of components needed to upgrade existing components is an NP-Hard problem [58]. In the simplest case—where any combination of the components are compatible—the problem of selecting which components to use in an upgrade is an instance of the *knapsack problem*, where a knapsack of predefined size is filled with items of various sizes and values. The goal is to maximize the sum of the value of items in the sack without exceeding the knapsack size. In this chapter, the knapsack size is defined by the total budget available for the component purchase and/or development; the goal is to find the optimal subset of the

hardware and software components that do not exceed the budget (*i.e.*, that fit into the knapsack) and maximize the value of the added capabilities [59].

Moreover, many software evolution problems do not fit into a relaxed paradigm where any set of components can be used. For example, purchasing two different infotainment software system implementations does not double the value of the car since only one system can actually be installed. In most situations, each new capability that can be added is a point of *design variability*, with a number of potential implementations [60] each having their own cost and performance. The infotainment system is the point of design variability and the various implementations of the infotainment system are the concrete options for that point of variability since only one concrete option can be chosen at one point in time.

Distributed, real-time, and embedded (DRE) systems, such as automotive and avionics systems, have limited resources and often exhibit tight coupling between hardware and software decisions [61]. A consequence of this tight-coupling is that the selected hardware components must provide sufficient resources to support the decisions made for the points of software variability. For example, purchasing an infotainment software system that consumes more memory than is available on its hosting hardware can yield a flawed configuration. When determining the set of software components to upgrade, therefore, careful consideration must be paid to the production and consumption of resources by hardware and software, respectively. Finding the set of replacement components that adheres to all resource constraints and maximizes total value for an upgrade is an *optimization problem* that focuses on determining solution(s) that maximizes a single element of the problem.

This type of hardware/software co-design problem is NP-Hard [50] since there are an exponential number of possible evolved configurations, which prohibits the use of exhaustive state space exploration even for minor DRE system software evolution. For example, Consider the evolution of an automobile braking system with 10 different points of software variability and 10 implementation options for each variability point. Likewise, assume there is a single variable hardware electronic control units (ECU) with 10 different available configurations with varying capabilities. This problem formulation has 10^{100} possible evolution configurations that must be considered.

Solution Approach→MMKP-based upgrade analysis. This chapter shows how a number of complex software evolution optimization problems can be recast as *multidimensional multiple-choice knapsack problems* (MMKP) [62]. MMKPs are a specialized version of the more general knapsack problem where the items are divided into sets and exactly one item must be chosen from each set. The goal of the MMKP problem is to maximize the value of the items placed into the knapsack without violating the knapsack size or set selection constraints.

By converting the task of determining valid, favorable software evolution configurations into an instance of an MMKP, developers can take advantage of powerful approximation algorithms [63]. While these algorithms do not guarantee optimal solutions, they frequently find near-optimal solutions in polynomial-time. Moreover, certain software evolution analysis problems that involve tightly-coupled hardware/software decisions can be framed as co-dependent MMKP problems, in which one problem produces resources for consumption by the other [50].

Converting software evolution decisions into tractable instances of MMKP problems is neither obvious nor trivial. This process is exacerbated by DRE systems in which software architecture decisions may effect the hardware architecture and vice versa, thereby complicating the evolution analysis. This chapter provides the following four contributions to the study of techniques that convert various software evolution analysis problems into MMKP instances: (1) we describe the *Software Evolution Analysis with Resources* (SEAR) technique for mapping software evolution analysis problems of several common software evolution scenarios to MMKP problems, (2) we show how these MMKP formulations of software evolution analysis problems can be solved using MMKP heuristic techniques and mapped back to upgrade solutions, (3) we present empirical comparisons of the optimality and solve times for three algorithms that can solve these transformed MMKP problems for problems of various size, and (4) we used this data to present a taxonomy that describes which algorithm is most effective based on the problem type and size.

5.3 Motivating Example

It is hard to upgrade the software and hardware in a DRE system to support new software features *and* adhere to resource constraints. For example, auto manufacturers that want to integrate automated parking software into a car must find a way to upgrade the hardware on the car to provide sufficient resources for the new software. Each automated parking software package may need a distinct set of controllers for movement (such as brake and throttle) and ECU processing capabilities (such as memory and processing power) [64].

Figure 5.1 shows a segment of automotive software and hardware design that we use as a motivating example throughout the chapter. This legacy configuration contains two software components: a legacy brake controller and a legacy throttle controller as shown in Figure 5.1(A). In addition to an associated value and purchase cost, each component consumes memory and processing power to function. These resources are provided by the hardware component (*i.e.*, the ECU). This configuration is valid since the ECU produces more memory and processing resources than the components collectively require.

Adding an automated parking system to the original design shown in Figure 5.1(A) may require software components that are more recent, more powerful, or provide more functionality than the original software components. For example, to provide automated parking, the throttle controller may need to possess functionality to interface with laser depth sensors. In this example, the original controller lacked this functionality and must be upgraded with a more advanced implementation. The implementation options for the throttle controller are shown in Figure 5.1(B).

Figure 5.1(B) shows potential controller evolution options. Two implementations are available for each controller.Developers installing an automated parking system must upgrade the throttle controller via one of the two available implementations and can optionally increase the functionality of the system by upgrading the brake controller.

Given a fixed software budget (*e.g.*, $500), developers can purchase any combination of controllers. If developers want to purchase both a new throttle controller *and* a new brake controller, however, they must purchase an additional ECU to provide the necessary resources. The other option is to not upgrade the brake controller, thereby sacrificing additional functionality, but saving money in the process.

Given a fixed total hardware/software budget of $700, the developers must first divide the budget into a hardware budget and a software budget. For example, they could divide the budget evenly, allocating $350 to the hardware budget and $350 to the software budget. With this budget developers can afford to upgrade the throttle controller software with Implementation B and the brake controller software with Implementation B. The legacy ECU alone, however, does not provide enough resources to support these two devices. Developers must therefore purchase an additional ECU to provide the necessary additional resources. The new configuration for this segment of the automobile with upgraded controllers and an additional ECU (with ECU1 Implementation A) can be seen in Figure 5.1(C).

Our motivating example above focused on 2 points of software design variability that could be implemented using 4 different new components. Moreover, 4 different potential hardware components could be purchased to support the software components. To derive a configuration for the entire automobile, an additional 46 software components and 20 other hardware components must be examined. Each configuration of these components could be a valid configuration, resulting in (50^{24}) unique potential configurations. In general, as the quantity of software and hardware options increase, the number of possible configurations increases exponentially, thereby rendering manual optimization solutions infeasible in practice.

5.4 Challenges of Evolution Decision Analysis

Several issues must be addressed when evolving software and hardware components. For example, developers must determine (1) what software and hardware components to buy and/or build to implement the new feature, (2) how much of the total budget to allocate to software and hardware, respectively, and (3) that the selected hardware components provide sufficient resources for the chosen software components. These issues are related, *e.g.*, developers can either choose the software and hardware

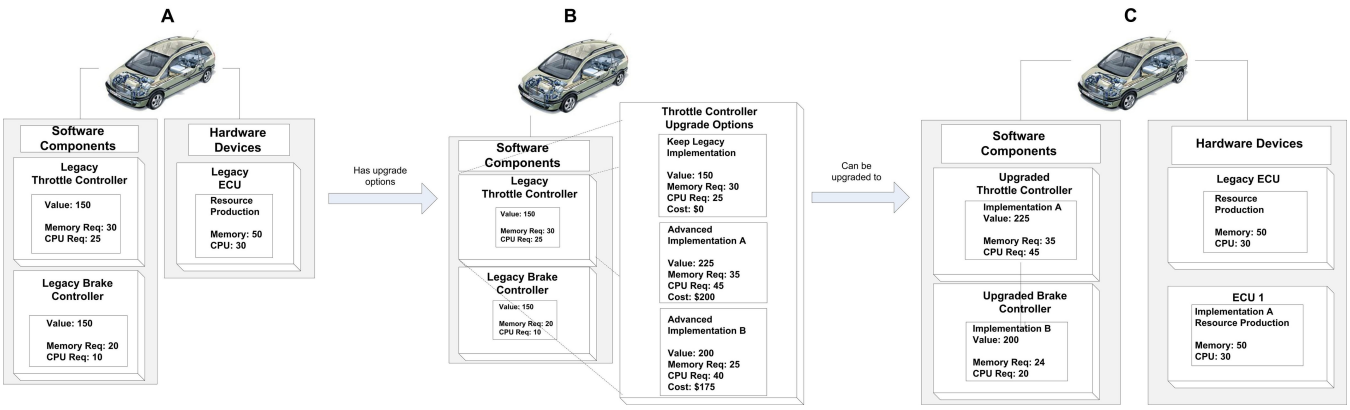

Figure 5.1: Software Evolution Progression

components to dictate the allocation of budget to software and hardware or the budget distributions can be fixed and then the components chosen. Moreover, developers can either choose the hardware components and then select software features that fit the resources provided by the hardware or the software can be chosen to determine what resource requirements the hardware must provide. This section describes a number of challenging upgrade scenarios that require developers to address the issues outlined above.

5.4.1 Challenge 1: Evolving Hardware to Meet New Software Resource Demands

This evolution scenario has no variability in implementing new functionality, *i.e.*, the set of software resource requirements is predefined. For example, if an automotive manufacturer has developed an in-house implementation of an automated parking system, the manufacturer will know the new hardware resources needed to support the system and must determine which hardware components to purchase from vendors to satisfy the new hardware requirements. Since the only purchases that must be made are for hardware, the exact budget available for hardware is known. The problem is to find the least-cost hardware design that can provide the resources needed by the software.

The difficulty of this scenario can be shown by assuming that there are 10 different hardware components that can be evolved, resulting in 10 points of hardware variability. Each replaceable hardware component has 5 implementation options from which the single upgrade can be chosen, thereby creating 5 options for each variability point.

To determine which set of hardware components yield the optimum value (*i.e.*, the highest expected return on investment) or the minimum cost (*i.e.*, minimum financial budget required to construct the system), 9,765,265 configurations of component implementations must be examined. Even after each configuration is constructed, developers must determine if the hardware components provides sufficient resources to support the chosen software configuration. Section 5.5.1 describes how SEAR addresses this challenge by using predefined software components and replaceable hardware components to form a single MMKP evolution problem.

5.4.2 Challenge 2: Evolving Software to Increase Overall System Value

This evolution scenario preselects the set of hardware components and has no variability in the hardware implementation. Since there is no variability in the hardware, the amount of each resource available for consumption is fixed. The software components, however, must be evolved. For example, a software component on a common model of automobile has been found to be defective.

To avoid the cost of a recall, the manufacturer can ship new software components to local dealerships, who can replace the defective software components. The dealerships lack the capabilities required to add hardware components to the automobile.

Since no new hardware is being purchased, the entire budget can be devoted to software purchases. As long as the resource consumption of the chosen software component configuration does not exceed the resource production of existing hardware components, the configuration can be considered valid. The difficulty of this challenge is similar to the one described in Section 5.4.1, where 10 different types of software components with 5 different available selections per type required the analysis of 9,765,265 configurations. Section 5.5.2 describes how SEAR addresses this challenge by using the predetermined hardware components and evolution software components to create a single MMKP evolution problem.

5.4.3 Challenge 3: Unrestricted Upgrades of Software and Hardware in Tandem

Yet another challenge occurs when both hardware components and software components can be added, removed, or replaced. For example, consider an automobile manufacturer designing the newest model of its flagship sedan. This sedan could either be similar to the previous model with few new software and hardware components or it could be completely redesigned, with most or all of the software and hardware components evolved.

Though the total budget is predefined for this scenario, it is not partitioned into individual hardware and software budgets, thereby greatly increasing the magnitude of the problem. Since neither the total provided resources nor total consumable resources are predefined, the software components depend on the hardware decisions and vice versa, incurring a strong coupling between the two seemingly independent MMKP problems.

The solution space of this problem is even larger than that of the challenge presented in Section 5.4.2. Assuming there are 10 different types of hardware options with 5 options per type, there are 9,765,265 possible hardware configurations. In this case, however, every type of software is eligible instead of just the types that are to be upgraded. If there are 15 types of software with 5 options per type, therefore, 30,516,453,125 software variations can be chosen. Each variation must be associated with a hardware configuration to test validity, resulting in 30,516,453,125 * 9,765,265 tests for each budget allocation.

In these worst case scenarios, the staggering size of the configuration space prohibits the use of exhaustive search algorithms for anything other than trivial design problems. Section 5.4 describes how SEAR addresses this challenge by combining all software and hardware components into a specialized MMKP evolution problem.

5.5 Evolution Analysis via SEAR

This section describes the procedure for transforming the evolution scenarios presented in Section 5.4 into evolution *Multidimensional Multiple-choice Knapsack Problems* (MMKP) [49]. MMKP problems are appropriate for representing evolution scenarios that comprise a series of points of design variability that are constrained by multiple resource constraints, such as the scenarios described in Section 5.4. In addition, there are several advantages to mapping the scenarios to MMKP problems.

MMKP problems have been extensively studied. As a result, there are several polynomial time algorithms that can be utilized to provide nearly optimal solutions, such as those described in [49, 65–67]. This chapter uses the M-HEU approximation algorithm described in [49] for evolution problems with variability in either hardware or software but not both. The multidimensional nature of MMKP problems is ideal for enforcing multiple resource constraints. The multiple-choice aspect of MMKP problems make them appropriate for situations (such as those described in Section 5.4.2) where only a single software component implementation can be chosen for each point of design variability.

MMKP problems can be used to represent situations where multiple options can be chosen for implementation. Each implementation option consumes various amounts of resources and has a distinct value. Each option is placed into a distinct MMKP set with other competing options and only a single option can be chosen from each set. A valid configuration results when the combined resource consumption of the items chosen from the various MMKP sets does not exceed the resource limits. The value of the solution is computed as the sum of the values of selected

items.

5.5.1 Mapping Hardware Evolution Problems to MMKP

Below we show how we map the hardware evolution problem described in Section 5.4.1 to an MMKP problem. In this case, the scenario can be mapped to a single MMKP problem representing the points of hardware variability. The size of the knapsack is defined by the hardware budget. The only additional constraint on the MMKP solution is that the quantities of resources provided by the hardware configuration exceeds the predefined consumption needs of software components.

To create the hardware evolution MMKP problem, each hardware component is converted to an MMKP item. For each point of hardware variability, an MMKP set is created. Each set is then populated with the MMKP items corresponding to the hardware components that are implementation options for the set's corresponding point of hardware variability. Figure 5.2 shows a mapping of a hardware evolution problem for an ECU to an MMKP.

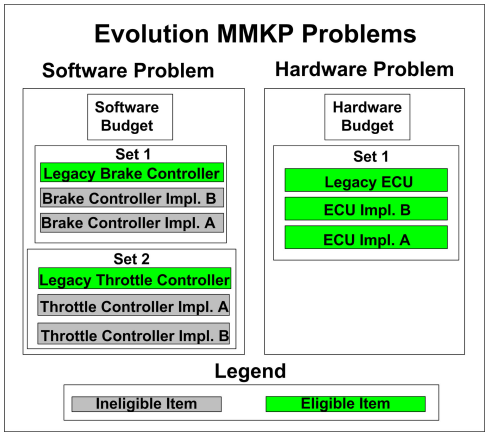

Figure 5.2: MMKP Representation of Hardware Evolution Problem

In Figure 5.2 the software does not have any points of variability that are eligible for evolution. Since there is no variability in the software, the exact amount of each resource that will be consumed by the software is known. The M-HEU approximation algorithm (or an exhaustive search algorithm, such as a linear constraint solver) uses this hardware evolution MMKP problem, the predefined resource consumption, and the predefined external resource (budget) requirements to determine which ECUs to purchase and install. The solution to the MMKP is the hardware components that should be chosen to implement each point of hardware variability.

5.5.2 Mapping Software Evolution Problems to MMKP

We now show how to map the software evolution problem described in Section 5.4.2 to an MMKP problem. In this case, the hardware configuration cannot be altered, as shown in Figure 5.3. The hardware thus produces a

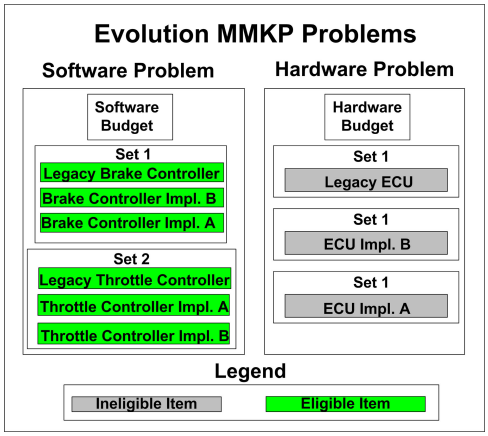

Figure 5.3: MMKP Representation of Software Evolution Problem

predetermined amount of each resource. Similar to Section 5.5.1, the fiscal budget available for software purchases is also predetermined. Only the software evolution MMKP problem must therefore be solved to determine an optimal solution.

As shown in the *software problem* portion of Figure 5.3, each point of software variability becomes a set that contains the corresponding controller implementations. For each set there are multiple implementations that can serve as the controller. This software evolution problem—along with the software budget and the resources available for consumption as defined by the hardware configuration—

can be used by an MMKP algorithm to determine a valid selection of throttle and brake controllers.

5.5.3 Hardware/Software Co-Design with ASCENT

Several approximation algorithms can be applied to solve single MMKP problems, as described in Sections 5.5.1 and 5.5.2. These algorithms, however, cannot solve cases in which there are points of variability in both hardware and software that have eligible evolution options. In this situation, the variability in the production of resources from hardware and the consumption of resources by software requires solving two MMKP problems simultaneously, rather than one. In prior work we developed the *Allocation-baSed Configuration Exploration Technique* (ASCENT) technique to determine valid, low-cost solutions for these types of dual MMKP problems [50].

ASCENT is a search-based, hardware/software co-design approximation algorithm that maximizes the software value of systems while ensuring that the resources produced by the hardware MMKP solution are sufficient to support the software MMKP solution [50]. The algorithm can be applied to system design problems in which there are multiple producer/consumer resource constraints. In addition, ASCENT can enforce external resource constraints, such as adherence to a predefined budget.

The software and hardware evolution problem described in Section 5.4.3 must be mapped to two MMKP problems so ASCENT can solve them. The hardware and software evolution MMKP problems are prepared as shown in Figure 5.4. This evolution differs from the problems described in Section 5.5.1, since all software implementations are now eligible for evolution, thereby dramatically increasing the amount of variability. These two problems—along with the total budget—are passed to ASCENT, which then searches the configuration space at various budget allocations to determine a configuration that optimizes a linear function computed over the software MMKP solution. Since ASCENT utilizes an approximation algorithm, the total time to determine a valid solution is usually small. In addition, the solutions it produces average over 90% of optimal [50].

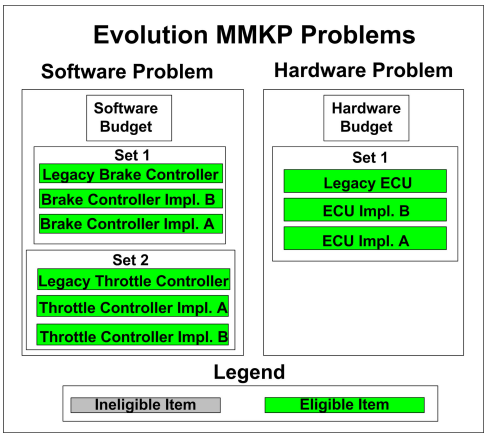

Figure 5.4: MMKP Representation of Unlimited Evolution Problem

5.6 Empirical Results

This section presents empirical data obtained from using three different algorithmic techniques to determine valid, high-value, evolution configurations for the scenarios described in Section 5.4. These results demonstrate that each algorithm is effective for certain types of MMKP problems. Moreover, if the correct technique is used, a near-optimal solution can be found. Each set represents a point of design variability and problems with more sets have more variability. Moreover, the ASCENT and M-HEU algorithms can be used to determine solutions for large-scale problems that cannot be solved in a feasible amount of time with exhaustive search algorithms.

5.6.1 Experimental Platform

All algorithms were implemented in Java and all experiments were conducted on an Apple MacbookPro with a 2.4 GHz Intel Core 2 Duo processor, 2 gigabytes of RAM, running OS X version 10.5.5, and a 1.6 Java Virtual Machine (JVM) run in client mode. For our exhaustive MMKP solving technique—which we call the linear constraint solver (LCS)—we used a branch and bound solver built on top of the Java Choco Constraint Solver (`choco.sourceforge.net`). The M-HEU heuristic solver was a custom implementation that we developed with Java. The ASCENT algorithm was also based on

a custom implementation with Java. Simulation MMKP problems were randomly generated as described in [50].

5.6.2 Hardware Evolution with Predefined Resource Consumption

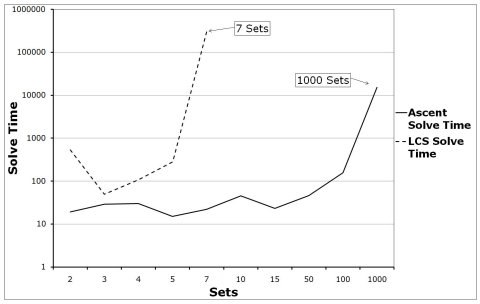

Figure 5.5: Hardware Evolution Solve Time vs Number of Sets

This experiment investigates the use of a linear constraint solver and the use of the M-HEU algorithm to solve the challenge described in Section 5.4.1, where the software components are fixed. First, we test for the total time needed for each algorithm to run to completion. We then examine the optimality of the solutions generated by each algorithm. We run these tests for several problems with increasing set counts, thus showing how each algorithm performs with increased design variability.

Figure 5.5 shows the time required to generate a hardware configuration if the software configuration is predefined.[1]

Since only a single MMKP problem must be solved, we use the M-HEU algorithm. As set size increases, the time required for the linear constraint solver increases rapidly. If the problem consists of more sets, the time required for the linear constraint solver becomes prohibitive. The M-HEU approximation algorithm, however, scaled much better, finding a solution for a problem with 1,000 sets in ∼15 seconds. Figure 5.6 shows that both algorithms generated solutions with 100% optimality for problems with 5 or less sets.

[1]Time is plotted on a logarithmic scale for all figures that show solve time.

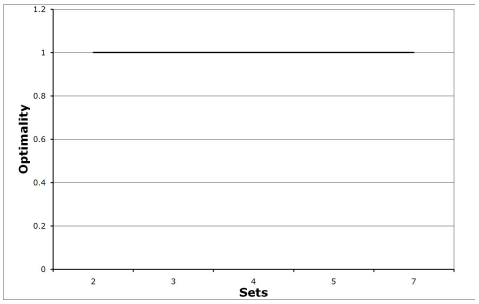

Figure 5.6: Hardware Evolution Solution Optimality vs Number of Sets

Regardless of the number of sets, the M-HEU algorithm completed faster than the linear constraint solver without sacrificing optimality.

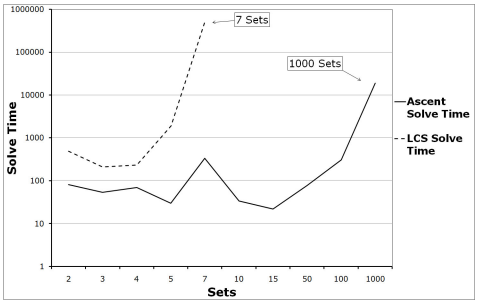

Figure 5.7: Software Evolution Solve Time vs Number of Sets

5.6.3 Software Evolution with Predefined Resource Production

This experiment examines the use of a linear constraint solver and the M-HEU algorithm to solve evolution scenarios in which the hardware components are fixed, as described in Section 5.4.2. We test for the total time each algorithm needs to run to completion and examine the optimality of solutions generated by each algorithm.

Figure 5.7 shows the time required to generate a software configuration generated if the hardware configuration is predetermined. As with Challenge 2, the M-HEU algorithm is used since only a single MMKP problem must be solved. Once again, LCS's limited scalability is

demonstrated since the required solve time makes its use prohibitive for problems with more than five sets. The M-HEU solver scales considerably better and can solve a problem with 1,000 sets in less than 16 seconds, which is fastest for all problems.

Figure 5.8 shows the optimality provided by each solver. In this case, the M-HEU solver is only 80% optimal for problems with 4 sets. Fortunately, the optimality improves with each increase in set count with a solution for a problem with 7 sets being 100% optimal.

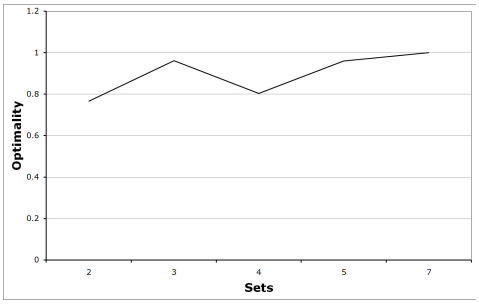

Figure 5.8: Software Evolution Solution Optimality vs Number of Sets

5.6.4 Unrestricted Software Evolution with Additional Hardware

This experiment examines the use of a linear constraint solver and the ASCENT algorithm to solve the challenge described in Section 5.4.3, in which no hardware or software components are fixed. We first test for the total time needed for each algorithm to run to completion and then examine the optimality of the solutions generated by each algorithm. Unrestricted evolution of software and hardware components has similar solve times to the previous experiments.

Figure 5.9 shows that regardless of the set count for the MMKP problems, the ASCENT solver derived a solution much faster than LCS. This figure also shows that the required solve time to determine a solution with LCS increases rapidly, *e.g.*, problems that have more than five sets require an extremely long solve time. The ASCENT algorithm once again scales considerably better and can even solve problems with 1,000 or more sets. In this case,

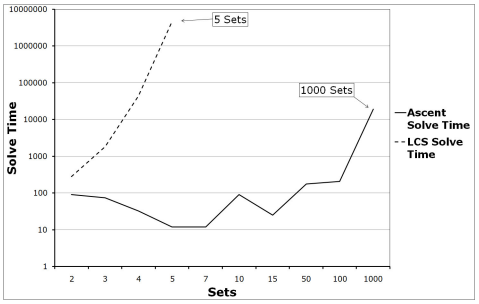

Figure 5.9: Unrestricted Evolution Solve Time vs Number of Sets

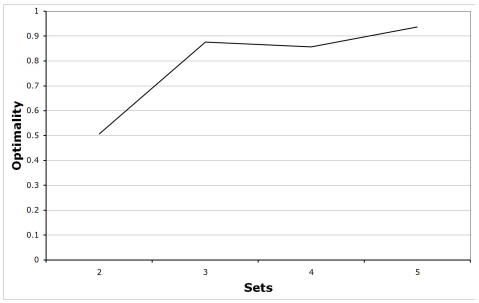

Figure 5.10: Unrestricted Evolution Solution Optimality vs Number of Sets

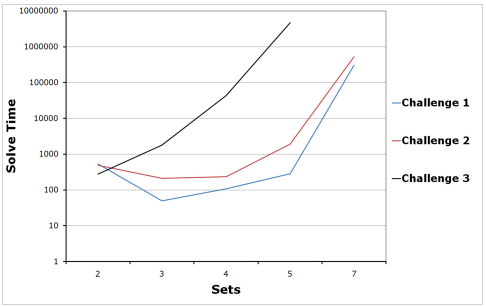

Figure 5.11: LCS Solve Times vs Number of Sets

the optimality of the solutions found by ASCENT is low for problems with 5 sets, as shown in Figure 5.10.

Fortunately, the time required to solve with LCS is not prohibitive in these cases, so it is still possible to find a solution with 100% optimality in a reasonable amount of time.

5.6.5 Comparison of Algorithmic Techniques

This experiment compared the performance of LCS to the performance of the M-HEU and ASCENT algorithms for all challenges in Section 5.4. As shown in Figure 5.11, the characteristics of the problem(s) being solved has a large impact on solving duration.

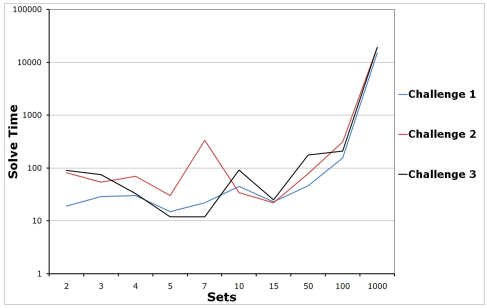

Figure 5.12: M-HEU & ASCENT Solve Times vs Number of Sets

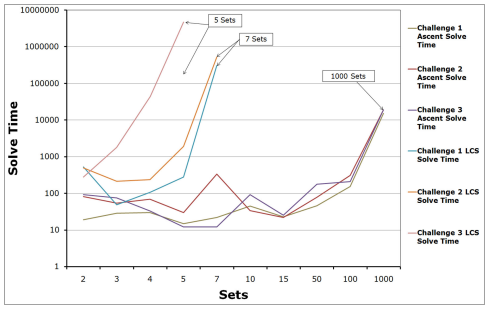

Figure 5.13: Comparison of Solve Times for All Experiments

Each challenge has more points of variability than the previous challenge. The solving time for LCS thus increases as the number of the points of variability increases. For all cases, the LCS algorithm requires an exorbitant amount of time for problems with more than five sets. In contrast, the M-HEU and ASCENT algorithms show no discernable correlation between the amount of variability and the solve time. In some cases, problems with more sets require more time to solve than problems with less sets, as shown in Figure 5.12.

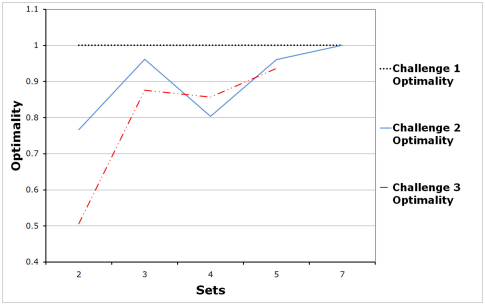

Figure 5.14: Comparison of Optimalities for All Experiments

Solver	Variability in Either Hardware or Software		Variability in Both Hardware and Software	
	Sets ≥ 8	Sets < 8	Sets ≥ 6	Sets < 6
LCS		X		X
M-HEU	X			
ASCENT			X	

Figure 5.15: Taxonomy of Techniques

Figure 5.13 compares the scalability of the three algorithms.

This figure shows that LCS requires the most solving time in all cases. Likewise, the ASCENT and M-HEU algorithms scale at approximately the same rate for all problems and are far superior to the LCS algorithm. The optimality of the ASCENT and M-HEU algorithms is near-optimal only for problems with five or more sets, as shown in Figure 5.14.

The exception to this trend occurs if there are few points of variability, *e.g.*, when there are few sets and the software is predetermined. These findings motivate the taxonomy shown in Figure 5.15 that describes which algorithm is most appropriate, based on problem size and variability.

5.7 Related Work

Search-based software engineering has a large number of facets ranging from the design of general approximation algorithms to the construction of search-based software engineering methods for specific problems. This section compares and contrasts SEAR to search-based software engineering techniques related to (1) methods for using search-based techniques to solve hardware/software partitioning problems and (2) search-based software engineering techniques for determining project staffing.

Hardware/software partitioning. A number of co-design techniques [68–70]—that can be viewed as search-based software engineering techniques—examine the problem of partitioning system functionality into hardware and software. These approaches use a number of search techniques ranging from binary constraint search to dynamic programming. In the partitioning problem, a system's required operations are grouped into tasks or functions, which are then implemented in either hardware or software. The goal is to correctly partition the tasks into hardware and software to meet a predefined performance goal. Some tasks may operate with higher performance if the functionality is placed on the hardware rather than on software. The performance of the system is thus determined by the location and placement of tasks in hardware versus software.

The MMKP co-design problem, which SEAR focuses on, is complementary to this research. In particular, these related approaches do not deal with maximizing a measure of system value subject to producer/consumer resources and cost. Similarly, SEAR does not examine the impact of the placement of tasks on the hardware and software. Each approach fills an important, although distinct, role in the search-based software engineering landscape for hardware and software evolution.

Project management and staff allocation. Accurate planning of large projects are essential to estimate project cost, determine the formation of employee project teams, and to assign these teams to tasks in a manner that gives the largest probability for successful completion. The placement of each individual employee can change the profile of the entire project plan, resulting in an exponential number of possible configurations [71]. Moreover, parameters of a project are dynamic and may change several times before project completion, requiring that multiple staffing solutions be calculated. This research is related to hardware and software evolution problems in that it deals with two tightly-coupled activites–the ordering and staffing of project parts subject to resource constraints. Although the work is related, it cannot be used to solve MMKP hardware and software evolution problems. In contrast, SEAR is specifically designed for solving MMKP hardware and software evolution problems.

Chapter 6

Concluding Remarks

This chapter presents lessons learned from our work in DRE system deployment and configuration derivation. Section 6.1 presents our findings from constructing an automated technique for deriving deployments with reduced processor requirements. Section 6.2 showcases conclusions drawn from creating a tool to optimize system-wide deployment properties. Section 6.3 describes lessons learned from creating a model-driven tool for DRE system configuration. Finally, Section 6.4 presents our discoveries from creating an automated technique for evolving DRE systems.

6.1 Automated Deployment Derivation

Determining component deployments that minimize the number of required processors is hard. This problem is exacerbated by proving that software applications are schedulable for a chosen deployment. Using bin packing algorithms, such as first-fit decreasing, the entire deployment space need not be searched. By using our BLITZ algorithm (which combines first-fit decreasing bin packing with proven utilization bounds based on data characteristics), valid and near minimal deployments can be determined.

Based on our work with BLITZ thus far, we learned the following lessons pertaining to deployment for DRE systems:

- **Grouping based on harmonic periods improves packing tightness**. BLITZ combines the Liu-Layland equation with the increased utilization bound of components with harmonic execution periods to maximize the utilization of each processor during deployment. As a result, tasks can be clustered on fewer processors, reducing the processors required.
- **Processor minimization depends on real-time benchmarks**. BLITZ has been shown to greatly reduce the required processors of a DRE system of an extensively benchmarked real-time system. Without knowledge of periodicity, resource constraints, and co-location constraints, BLITZ cannot be fully utilized. It is essential to develop tools that effectively simulate and thoroughly benchmark DRE systems before they are deployed so that the full capabilities of BLITZ can be applied.

The current version of BLITZ with example code is available in open-source form at `ascent-design-studio.googlecode.com`. The industry challenge problem that is the basis for this paper can be found at `www.sprucecommunity.org`.

6.2 Legacy Deployment Optimization

Optimizing deployment topologies on legacy embedded flight avionics system can yield substantial benefits, such as reducing hardware costs and power consumption. By combining the efficiency of metaheuristic optimization techniques (such as particle swarm optimization) with other heuristic algorithms (such as bin-packing) legacy

deployments can be evolved and optimized in a matter of seconds.

The following are a summary of the lessons we learned applying our ScatterD tool for deployment optimization to a legacy flight avionics system:

- **Multiple constraints make deployment planning hard**. Avionics deployments must adhere to a wide range of strict constraints, such as resource, colocation, scheduling, and network bandwidth. Deployment optimization tools must account for all these constraints when determining a new deployment.

- **A Huge deployment space requires intelligent search techniques.** The vast majority of potential deployments that could be created violate one or more design constraints. Intelligent and automated techniques, such as hybrid-heuristic bin-packing, should therefore be applied to discover valid "near-optimal" deployments.

- **Substantial processor and network bandwidth reductions are possible.** Applying hybrid-heuristic bin-packing to the flight avionics system resulted in 42.8% processor reduction and 24% bandwidth reduction. Our future work is applying hybrid-heuristic bin-packing to other embedded system deployment domains, such as automobiles, multi-core processors, and tactical smartphone applications.

The ScatterD tool is available in open-source form in the Ascent Design Studio(ascent-design-studio.googlecode.com). A document describing the flight avionics system case study outlined in Section 3.3, as well as additional information on ScatterD, can be found at the SPRUCE web portal (www.sprucecommunity.org), which pairs open industry challenge problems with cutting-edge methods and tools from the research community.

6.3 Model Driven Configuration Derivation

Determining valid configurations for distributed real-time and embedded (DRE) systems is hard. Designers must take into account a myriad of constraints including resource constraints, real-time constraints, QoS constraints, and other functional constraints. The difficulty of this task is exacerbated by the presence of a plethora of potential COTS components for inclusion in the configuration, with each providing varying quality of service, functionality, resource requirements and financial cost. This high availability of COTS components results in an exponential number of potential DRE system configurations.

As a result, manual techniques for determining valid DRE system configurations are far too cumbersome. Even exact automated techniques, such as the use of CSPs, require a prohibitive amount of time to execute. Approximation techniques, such as ASCENT, however, do not require an exhaustive search of the vast design space allowing a much more rapid execution while often resulting in solutions with over 95% optimality.

The use of complex programmatic techniques in approximation techniques like ASCENT often have a steep learning curve and require large amounts of coding to construct a problem for execution. Due to the complex coding involved, these techniques carry the added burden of being error prone when defining problem instances. To address these challenges, an MDA-based tool called the Ascent Modeling Platform (AMP) that utilized GME to construct problem instances and display valid solutions for DRE system configurations was utilized. The following are lessons learned during our creation and use of AMP:

- **Modeling tools provide rapid problem construction.** Through the use of GME, problems could be constructed in a fraction of the time of using programmatic techniques.

- **Utilizing MDA reduces human error.** AMP utilizes a GME metamodel that enforces the many complex design constraints associated with DRE system configuration. As a result, users of AMP are prevented from constructing a configuration problem that is invalid.

- **Modeling tools facilitate design space exploration.** Solutions are posted directly back into the model for analysis by system designers. Designers can then change problem parameters within the model and execute the interpreter to explore multiple configura-

tion scenarios, resulting in an increased understanding of the design space.

- **Multiple execution options still needed.** Currently ASCENT is the only technique that is executed upon interpreting models in AMP. Other techniques, such as the use of CSP solvers, should be implemented to determine solutions to problems with an appropriately reduced number of candidate components.

The current version of AMP with example code is available in open-source form at ascent-design-studio.googlecode.com.

6.4 Automated Hardware and Software Evolution Analysis

Determining valid evolution configurations for software/-hardware configurations that increase system value is hard. The exponential number of possible configurations that stem from the massive variability in these problems prohibit the use of exhaustive search algorithms for non-trivial problems. This chapter presented the *Software Evolution Analysis with Resources* (SEAR) technique, which converts common evolution problems into *multi-dimensional multiple-choice knapsack problems* (MMKP). We also empirically evaluated three different algorithms for solving these problems to compare their effectiveness in providing valid, high-value evolution configurations.

From these experiments, we learned the following lessons pertaining to determine valid evolution configurations for hardware/software co-design systems:

- **Approximation algorithms scale better than exhaustive algorithms.** Exhaustive search techniques, such as the linear constraint solver algorithm, cannot be applied to non-trivial problems. The determining factor in the effectiveness of these algorithms is the number of problem sets. To solve problems with realistic set counts in feasible time, approximation algorithms, such as the M-HEU algorithm or the ASCENT algorithm must be used. These techniques can solve even large problems in seconds, with minimal impact on optimality.

- **Extremely small or large problems yield near-optimal solutions.** For non-trivial problems, the ASCENT algorithm and M-HEU algorithm can be used to determine near-optimal evolution configurations. For tiny problems, the LCS algorithm can be used to determine optimal solutions. Given that these tiny problems have few points of variability, this can be done rapidly.

- **Problem size should determine which algorithm to apply.** Based on problem characteristics, it can be highly advantageous to use one algorithmic technique versus another, which can result in faster solving times or higher optimality. Figure 5.15 shows the problem attributes that should be examined when deciding which algorithm to apply. It also relates the algorithm that is best suited for solving these evolution problems based on the number of sets present.

- **No algorithm is universally superior.** The analysis of empirical results indicate that all three algorithms are superior for different types of evolution problems. We have not, however, discovered an algorithm that performs well for every problem type. To determine if other existing algorithms perform better for one or all types of evolution problems, further experimentation and analysis is necessary. Our future work will examine other approximation algorithms, such as genetic algorithms and particle swarm techniques, to determine if a single superior algorithm exists.

The current version of ASCENT with example code that utilizes SEAR is available in open-source form at ascent-design-studio.googlecode.com.

Bibliography

[1] H. Beitollahi and G. Deconinck, "Fault-Tolerant Partitioning Scheduling Algorithms in Real-Time Multiprocessor Systems," *Pacific Rim International Symposium on Dependable Computing, IEEE*, vol. 0, pp. 296–304, 2006.

[2] M. Mikic-Rakic and N. Medvidovic, "Architecture-Level Support for Software Component Deployment in Resource Constrained Environments," *LECTURE NOTES IN COMPUTER SCIENCE*, pp. 31–50, 2002.

[3] N. R. C. Steering Committee for the Decadal Survey of Civil Aeronautics, *Decadal Survey of Civil Aeronautics: Foundation for the Future*. The National Academies Press, 2996.

[4] D. Seto, J. Lehoczky, L. Sha, and K. Shin, "On task schedulability in real-time control systems," in *Real-Time Systems Symposium, 1996., 17th IEEE*, 1996, pp. 13–21.

[5] A. Burchard, J. Liebeherr, Y. Oh, and S. Son, "New Strategies for Assigning Real-time Tasks to Multiprocessor Systems," *IEEE Transactions on Computers*, vol. 44, no. 12, pp. 1429–1442, 1995.

[6] S. Lauzac, R. Melhem, and D. Mosse, "Comparison of Global and Partitioning Schemes for Scheduling Rate Monotonic Tasks on a Multiprocessor," in *10th Euromicro Workshop on Real Time Systems*, 1998, pp. 188–195.

[7] A. Bertossi, L. Mancini, and F. Rossini, "Fault-Tolerant Rate-Monotonic First-Fit Scheduling in Hard-Real-Time Systems," *IEEE Transactions On Parallel and Distributed Systems*, pp. 934–945, 1999.

[8] S. Dhall and C. Liu, "On a Real-time Scheduling Problem," *Operations Research*, vol. 26, no. 1, pp. 127–140, 1978.

[9] J. Liebeherr, A. Burchard, Y. Oh, and S. H.Son, "New strategies for assigning real-time tasks to multiprocessor systems," *IEEE Trans. Comput.*, vol. 44, no. 12, pp. 1429–1442, 1995.

[10] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-time Environment," *JACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[11] D. De Niz and R. Rajkumar, "Partitioning bin-packing algorithms for distributed real-time systems," *International Journal of Embedded Systems*, vol. 2, no. 3, pp. 196–208, 2006.

[12] K. Tindell, A. Burns, and A. Wellings, "Allocating hard real-time tasks: an NP-hard problem made easy," *Real-Time Systems*, vol. 4, no. 2, pp. 145–165, 1992.

[13] L. Sha and J. Goodenough, "Real-time scheduling theory and Ada," *Computer*, vol. 23, no. 4, pp. 53–62, 1990.

[14] J. Strosnider and T. Marchok, "Responsive, deterministic IEEE 802.5 token ring scheduling," *Real-Time Systems*, vol. 1, no. 2, pp. 133–158, 1989.

[15] L. Lehoczky, J.P. snf Sha and J. Strosnider, "Enhancing Aperiodic Responsiveness in a Hard Real-Time Environment," in *Proc. of the IEEE Real-Time Systems Symposium*, 1987, pp. 416–423.

[16] A. Carzaniga, A. Fuggetta, S. Richard, D. Heimbigner, A. van der Hoek, A. Wolf, and COLORADO

STATE UNIV FORT COLLINS DEPT OF COMPUTER SCIENCE, *A Characterization Framework for Software Deployment Technologies*. Defense Technical Information Center, 1998.

[17] J. Stankovic, "Strategic Directions in Real-time and Embedded Systems," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, pp. 751–763, 1996.

[18] W. Damm, A. Votintseva, A. Metzner, B. Josko, T. Peikenkamp, and E. Böde, "Boosting Re-use of Embedded Automotive Applications Through Rich Components," *Proceedings of Foundations of Interface Technologies*, vol. 2005, 2005.

[19] D. C. Schmidt, "Model-Driven Engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[20] C. Fonseca, P. Fleming, *et al.*, "Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization," in *Proceedings of the fifth international conference on genetic algorithms*. Citeseer, 1993, pp. 416–423.

[21] R. Poli, J. Kennedy, and T. Blackwell, "Particle swarm optimization," *Swarm Intelligence*, vol. 1, no. 1, pp. 33–57, 2007.

[22] S. Davari and S. Dhall, "An On-line Algorithm for Real-time Tasks Allocation," in *IEEE Real-time Systems Symposium*, 1986, pp. 194–200.

[23] S. Lauzac, R. Melhem, and D. Mosse, "An efficient RMS Admission Control and its Application to Multiprocessor Scheduling," in *International Parallel Processing Symposium*, 1998, pp. 511–518.

[24] S. Davari and S. Dhall, "On a Periodic Real-Time Task Allocation Problem," in *19th Annual International Conference on System Sciences*, 1986, pp. 133–141.

[25] D. Kirovski and M. Potkonjak, "System-level synthesis of low-power hard real-time systems," in *Proceedings of the 34th annual conference on Design automation*. ACM New York, NY, USA, 1997, pp. 697–702.

[26] R. Dick and N. Jha, "MOGAC: A multiobjective genetic algorithm for the co-synthesis of hardware-software embedded systems," in *Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*. IEEE Computer Society Washington, DC, USA, 1997, pp. 522–529.

[27] ——, "MOCSYN: Multiobjective core-based single-chip system synthesis," in *Proceedings of the conference on Design, automation and test in Europe*. ACM New York, NY, USA, 1999.

[28] C. Wang and Z. Li, "A computation offloading scheme on handheld devices," *Journal of Parallel and Distributed Computing*, vol. 64, no. 6, pp. 740–746, 2004.

[29] G. Chen, B. Kang, M. Kandemir, N. Vijaykrishnan, M. Irwin, and R. Chandramouli, "Studying energy trade offs in offloading computation/compilation in Java-enabled mobile devices," *IEEE Transactions on Parallel and Distributed Systems*, pp. 795–809, 2004.

[30] Z. Li, C. Wang, and R. Xu, "Task allocation for distributed multimedia processing on wirelesslynetworked handheld devices," in *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, 2002, pp. 79–84.

[31] N. Wang, D. C. Schmidt, A. Gokhale, C. D. Gill, B. Natarajan, C. Rodrigues, J. P. Loyall, and R. E. Schantz, "Total Quality of Service Provisioning in Middleware and Applications," *The Journal of Microprocessors and Microsystems*, vol. 27, no. 2, pp. 45–54, mar 2003.

[32] D. Schmidt, "Middleware for real-time and embedded systems," *Communications of the ACM*, vol. 45, no. 6, p. 48, 2002.

[33] J. Voas, "Certifying off-the-shelf software components," *Computer*, vol. 31, no. 6, pp. 53–59, 1998.

[34] A. Gokhale, D. C. Schmidt, B. Natarajan, and N. Wang, "Applying Model-Integrated Computing

to Component Middleware and Enterprise Applications," *The Communications of the ACM Special Issue on Enterprise Components, Service and Business Rules*, vol. 45, no. 10, Oct. 2002.

[35] C. Alves and J. Castro, "Cre: A systematic method for cots components selection," in *XV Brazilian Symposium on Software Engineering (SBES)*. Rio de Janeiro, Brazil, 2001.

[36] L. Chung and K. Cooper, "COTS-aware requirements engineering and software architecting," in *Proc. of Software Engineering Research and Practice Conference (SERP)*. Citeseer, 2004.

[37] M. Morisio, C. Seaman, V. Basili, A. Parra, S. Kraft, and S. Condon, "COTS-based software development: Processes and open issues," *Journal of Systems and Software*, vol. 61, no. 3, pp. 189–199, 2002.

[38] S. Mellor, S. Kendall, A. Uhl, and D. Weise, *MDA distilled*. Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA, 2004.

[39] J. Poole, "Model-driven architecture: Vision, standards and emerging technologies," in *Workshop on Metamodeling and Adaptive Object Models, ECOOP*. Citeseer, 2001.

[40] S. Kent, "Model Driven Engineering," in *Proceedings of the 3rd International Conference on Integrated Formal Methods (IFM 02)*. Turku, Finland: Springer-Verlag LNCS 2335, May 2002, pp. 286–298.

[41] D. Benavides, P. Trinidad, and A. Ruiz-Cortes, "Automated Reasoning on Feature Models," in *Proceedings of the 17th Conference on Advanced Information Systems Engineering*. Porto, Portugal: ACM/IFIP/USENIX, 2005.

[42] D. Sabin and E. Freuder, "Configuration as Composite Constraint Satisfaction," in *Proceedings of the Artificial Intelligence and Manufacturing Research Planning Workshop*, 1996, pp. 153–161.

[43] M. Jaring and J. Bosch, "Representing variability in software product lines: A case study," *Software Product Lines*, pp. 219–245, 2002.

[44] K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged Configuration Using Feature Models," in *Software Product Lines: Second International Conference, SPLC2, San Diego, CA, USA, August 19-22, 2002: Proceedings*. Springer, 2002, p. 266.

[45] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi, "The generic modeling environment," in *Workshop on Intelligent Signal Processing, Budapest, Hungary*, vol. 17. Citeseer, 2001.

[46] A. Ledeczi, G. Nordstrom, G. Karsai, P. Volgyesi, and M. Maroti, "On metamodel composition," in *IEEE CCA*, 2001.

[47] M. Richters and M. Gogolla, "On formalizing the UML object constraint language OCL," *Conceptual Modeling–ERù98*, pp. 449–464, 1998.

[48] J. White, B. Dougherty, and D. C. Schmidt, "Selecting Highly Optimal Architecture Feature Sets with Filtered Caresian Flattening," *Journal of Software and Systems - Special Issue on Design Decisions and Design Rationale in Software Architecture*, 2008.

[49] M. Akbar, E. Manning, G. Shoja, and S. Khan, "Heuristic Solutions for the Multiple-Choice Multi-dimension Knapsack Problem," *Lecture Notes in Computer Science*, pp. 659–668, 2001.

[50] J. White, B. Dougherty, and D. C. Schmidt, "ASCENT: An Algorithmic Technique for Designing Hardware and Software in Tandem," *IEEE Transactions on Software Engineering Special Issue on Search-based Software Engineering*, (to appear).

[51] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity - the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[52] T. Denton, E. Jones, S. Srinivasan, K. Owens, and R. Buskens, "NAOMI-An Experimental Platform for Multi-modeling," in *Proceedings of MODELS*, Toulouse, France, October 2008, pp. 143–157.

[53] S. Schach, *Classical and Object-Oriented Software Engineering*. McGraw-Hill Professional, 1995.

[54] A. Jost and A. Franke, "Residual Value Analysis," 2005.

[55] C. Ng and G. Chan, "An ERP maintenance model," in *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, 2003, p. 10.

[56] G. Leen and D. Heffernan, "Expanding Automotive Electronic Systems," *Computer*, vol. 35, no. 1, pp. 88–93, 2002.

[57] B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby, "Cost models for future software life cycle processes: COCOMO 2.0," *Annals of Software Engineering*, vol. 1, no. 1, pp. 57–94, 1995.

[58] X. Gu, P. Yu, and K. Nahrstedt, "Optimal Component Composition for Scalable Stream Processing," in *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on*, 2005, pp. 773–782.

[59] S. Martello and P. Toth, "Knapsack problems: algorithms and computer implementations," *Wiley-Interscience Series In Discrete Mathematics And Optimization*, p. 296, 1990.

[60] N. Ulfat-Bunyadi, E. Kamsties, and K. Pohl, "Considering Variability in a System FamilyâĂŹs Architecture During COTS Evaluation," in *Proceedings ofthe 4th International Conference on COTS-Based Software Systems (ICCBSS 2005), Bilbao, Spain*. Springer, 2005.

[61] S. Srinivasan and N. Jha, "Hardware-software cosynthesis of fault-tolerant real-time distributed embedded systems," in *European Design Automation Conference: Proceedings of the conference on European design automation*, vol. 18, no. 22, 1995, pp. 334–339.

[62] E. Lin, "A Biblographical Survey on Some Well-known Non-Standard Knapsack Problems," *INFOR-OTTAWA-*, vol. 36, pp. 280–283, 1998.

[63] M. Hifi, M. Michrafy, and A. Sbihi, "Heuristic algorithms for the multiple-choice multidimensional knapsack problem," *Journal of the Operational Research Society*, vol. 55, no. 12, pp. 1323–1332, 2004.

[64] J. Her, S. Choi, D. Cheun, J. Bae, and S. Kim, "A Component-Based Process for Developing Automotive ECU Software," *LECTURE NOTES IN COMPUTER SCIENCE*, vol. 4589, p. 358, 2007.

[65] A. Shahriar, M. Akbar, M. Rahman, and M. Newton, "A multiprocessor based heuristic for multi-dimensional multiple-choice knapsack problem," *The Journal of Supercomputing*, vol. 43, no. 3, pp. 257–280, 2008.

[66] M. Hifi, M. Michrafy, and A. Sbihi, "A Reactive Local Search-Based Algorithm for the Multiple-Choice Multi-Dimensional Knapsack Problem," *Computational Optimization and Applications*, vol. 33, no. 2, pp. 271–285, 2006.

[67] C. Hiremath and R. Hill, "New greedy heuristics for the Multiple-choice Multi-dimensional Knapsack Problem," *International Journal of Operational Research*, vol. 2, no. 4, pp. 495–512, 2007.

[68] E. Barros, C. Universitaria-Recife-PE, W. Rosenstiel, and X. Xiong, "A Method for Partitioning UNITY Language in Hardware and Software," *Euro-DAC'94 with Euro-VHDL'94: Proceedings, September 19-23, 1994, Grenoble, France*, 1994.

[69] E. Lagnese and D. Thomas, "Architectural Partitioning for System Level Design," *Design Automation, 1989. 26th Conference on*, pp. 62–67, 1989.

[70] F. Vahid, D. Gajski, and J. Gong, "A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning," *Proceedings of the conference on European design automation*, pp. 214–219, 1994.

[71] G. Antoniol, M. Di Penta, and M. Harman, "A robust search-based approach to project management in the presence of abandonment, rework, error

and uncertainty," *Software Metrics, 2004. Proceedings. 10th International Symposium on*, pp. 172–183, 2004.