Cent. Eur. J. Comp. Sci. • 1-26
Author version

Central European Journal of Computer Science

# Automated Software and Hardware Evolution Analysis for Distributed Real-time and Embedded Systems

Research Article

**Brian Dougherty[1]*, Jules White[2]†, Douglas C. Schmidt[1]‡**

1 Vanderbilt University,
1025 16th Ave S, Suite 102, 37212 Nashville, USA

2 Virginia Tech,
302 Whitemore Hall, 24060 Blacksburg, USA

**Abstract:** Software evolution is critical to extending the utility and life of distributed real-time and embedded (DRE) systems. Determining the optimal set of software and hardware components to evolve that (1) incorporate cutting-edge technology and (2) satisfy DRE system resource constraints, such as memory, power, and CPU usage is an NP-Hard problem. This article provides four contributions to evolving legacy DRE system configurations. First, we present the Software Evolution Analysis with Resources (SEAR) technique for converting legacy DRE system configurations, external resource availabilities, and candidate replacement components into multiple-choice multi-dimension knapsack problems (MMKP). Second, we present a formal methodology for assessing the validity of evolved system configurations. Third, we apply heuristic approximation algorithms to determine low-cost, high value evolution paths in polynomial time. Finally, we analyze results of experiments that apply these techniques to determine which technique is most effective for given system parameters. Our results show that constraint solvers can only evolve small system configurations, whereas approximation techniques are needed to evolve larger system configurations.

**Keywords:** Distributed Systems, Real-time, Embedded Systems• System Evolution • COTS Components • System Configuration • MMKP

## 1. Introduction

**Current trends and challenges.** Distributed real-time and embedded (DRE) systems (such as automotive, avionics, and automated manufacturing systems) are typically mission-critical and often remain in production for years or decades. As these systems age, however, the software and hardware that comprise them become increasingly obsolete as new components with enhanced functionality are developed. It is time consuming and expensive

* *E-mail: briand@dre.vanderbilt.edu*
† *E-mail: jules.white@vt.edu*
‡ *E-mail: schmidt@dre.vanderbilt.edu*

to completely re-build new systems from scratch to incorporate new technology. Instead of building replacement systems from the ground up, legacy systems can be *evolved* to include new technology by replacing older, obsolete components with newer, cutting-edge components as they become available. This evolution accounts for a large portion of the cost of supporting DRE systems [1].

Software evolution is particularly vital to ensure DRE systems continue to meet the changing needs of customers and remain relevant as markets evolve. 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. In the avionics industry, new flight controllers, targeting computers, and weapons systems are constantly being developed. DRE systems are often designed to squeeze the most resources out of the latest hardware and may not be compatible with hardware that is only a few years old. Many avionics systems have a lifespan of over 20 years, making this problem particularly daunting.

*Software evolution analysis* [2] is the process of updating a system with new software and hardware so that new technology can be utilized as it becomes available. Each component provides its own distinct functionality and effects the overall value of the system. Each component also generates various amounts of heat, consumes various amounts of resources (such as weight, power, memory, and processor utilization), and incurs a financial cost.

This analysis involves several challenges, including (1) creating a model for producing a cost/benefit analysis of different evolution paths, (2) determining the financial cost of evolving a particular software component [3], and (3) generating an evolved system configuration that satisfies multiple resource constraints while maximizing system value. This paper examines software evolution analysis techniques for automatically determining valid DRE system configurations that support required new capabilities and increase system value without violating, cost constraints resource constraints, or other domain-specific constraints, such as weight, heat generation, and power consumption.

As shown in prior work [4, 5], the cost/benefit analysis for software evolution is partially simplified by the availability of commercial-off-the-shelf (COTS) software/hardware components. 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. Likewise, avionics system developers know the precise weight of hardware components, the resources they provide, the power they consume, and the amount of heat they generate. If components are custom-developed (*i.e.*, non-COTS), profiling and analysis can be used to determine the cost/benefits and resource requirements of utilizing a component [6].

Even if the impact of including a component in an evolving DRE system is known, deciding which components would yield the best overall system value, is an NP-Hard problem [7]. The *knapsack problem* [8] can be used to model the simplest type of evolution problem. In this well-known problem, items of discrete size and value are selected to fill a knapsack of finite size, so that the collective value of the items in the knapsack is maximized.

This article uses a variation of the knapsack problem to represent DRE system configuration evolution options. In particular, items are used to represent the components available to evolve the system. The goal is to determine the best subset of hardware and software components to include in the final DRE system configuration without exceeding the project budget while maximizing the system value [9]. In the simplest type of evolution problem, there are no restrictions concerning which components can be used to evolve the system, and thus no additional restrictions on which items can be placed in the knapsack. Since the knapsack problem is NP-Hard, an exponential amount of time would be required to determine the optimal set of components to evolve the system even in the simplest scenario.

Unfortunately, this type of component evolution problem is too simplistic to represent actual DRE system evolution scenarios adequately. In particular, it may not be appropriate to augment DRE system configurations with components that fill the same basic need. For example, if the goal is to evolve the DRE system configuration of a smart car, it would usually not make sense to purchase and install two automated parking components. While installing a single automated parking component would increase the value of the system, a second would be superfluous and consume additional system resources without providing benefits.

To prevent adding excessive, repetitive components, each new potential DRE system capability is modeled as a point of *design variability* with several potential implementations, each incurring a distinct cost and value [10]. Modeling the option of adding an automated parking system as a point of variability prohibits multiple components that perform the same function from being implemented. It also simplifies cost/benefit analysis between potential candidate components that provide this functionality.

DRE systems are also subject to tight resource constraints. As a result, a tight coupling often exists between software and hardware, creating a producer/consumer interaction [11]. Each piece of hardware provides resources (such as memory, CPU, power, and heat dissipation) required for the software of a DRE system to run. One naive approach is to purchase superfluous hardware to ensure that the resource consumption needs of software are satisfied. Unfortunately, additional hardware also carries additonal weight and cost that may make a DRE system infeasible. For example, to maximize flight distance and speed, avionics systems must attempt minimize size and weight. Although adding superfluous hardware can ensure that more than enough resources exist for software to function, the additional weight and cost resulting from its implementation can render a system infeasible.

As a result, it is critical that sufficient resources exist to suppport any software variability selected for inclusion in the evolved DRE system without consuming unnecessary space, weight, and cost. Determining the subset of software components that maximize system value—while concurrently selecting the subset of hardware components to provide the necessary computational resources to support them—is an *optimization problem*. Cost constraints specifying that the total cost of all components must also not exceed that total financial exacerbates this problem. Due to these constraints, the knapsack problem representation of component evolution problems must be augmented with hardware/software co-design restrictions that realistically represent actual DRE systems. Since there are an exponential number of hardware and software component subsets that could be used in the final evolved configuration, this type of hardware/software co-design problem is NP-Hard [12], where the vast solution space prohibits the use of exhaustive state space exploration for non-trivial DRE systems.

For example, consider an avionics system with 20 points of software variability with 10 component options at each point. Assume only the flight deck electronic control unit hardware can be replaced with one of 20 candidate components with different resource production values, heat generation, weight and power consumption. To determine the optimal solution by exhaustively searching every possible evolution configuration would require examining $20^{11}$ evolution configurations. This explosion in solution space size would therefore require years to solve with exhaustive search techniques.

**Solution approach → System evolution with heuristic optimization techniques.** This article presents and evaluates a methodology for simplifying the evolution of DRE systems based on *multidimensional multiple-choice knapsack problems* (MMKP) [13]. MMKP problems extend the basic knapsack problem by adding constraints, such as multiple resource and cross-tree constraints, Similarly to the basic knapsack problem, items of different value and size are chosen for the knapsack to maximize total value. Two additional constraints are added to create an MMKP problem. First, each item consumes *multiple* resources (such as weight, power consumption, processing power) provided by the "knapsack" instead of space alone. Second, the items are divided into sets from which only a single item can be chosen.

For example, assume an MMKP problem in which the goal is to build the best home entertainment system, while not exceeding a given budget. In this case, the items are various types of televisions, game systems, and surround sound system. It would not make sense to choose two surround systems and a game system as the entertainment system requires a television and an extra surround system would be effectively useless. To represent this scenario as an MMKP problem, the items would be divided into a set of game systems, a set of surround sound systems, and a set of televisions. Any valid solution to this MMKP problem would enforce the constraints that exactly one television, game system, and surround system would be chosen and that the collective cost of the components would be under budget.

MMKP problems are appropriate for representing software evolution analysis problems for the following reasons:

- MMKP problem constraints are appropriate for enforcing the multiple resource and functional constraints of software evolution problems.

- Extensive study of MMKP problems has yielded approximation algorithms that can be applied to determine valid near-optimal solutions in polynomial time [14].

- Multiple MMKP problems can been used to represent the complex resource consumption/production relationship of tightly coupled hardware/partitions [12].

These problems can also be extended to include additional hardware restrictions, such as power consumption, heat production and weight limits.

Transforming software evolution analysis scenarios into MMKP problems, however, is neither easy nor intuitive. This challenge is exacerbated by complex production/consumption relationships between hardware and software components. This article illuminates the process of using MMKP problem instances to represent software evolution analysis problems with the following contributions:

- We present the *Software Evolution Analysis with Resources* (SEAR), which is a technique that represents multiple software evolution analysis scenarios with MMKP problems,
- We provide heuristic approximation techniques that can be applied to these MMKP problems to yield valid, high-value evolved system configurations,
- We provide a formal methodology for assessing the validity of complex, evolved DRE system configurations,
- We present empirical results of comparing the solve times and solution value of three algorithms for solving MMKP representations of software evolution scenarios,
- We analyze these results to determine a taxonomy for choosing the best technique(s) to use based on system size.

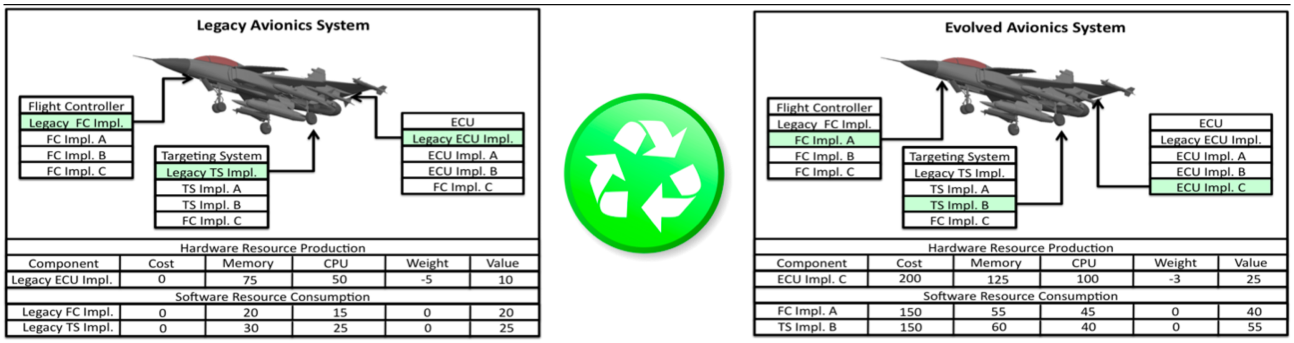
**Paper organization.** The remainder of this paper is organized as follows: Section 2 outlines an avionics system evolution case study used to motivate the need for—and applicability of—our SEAR technique; Section 3 describes several challenge problems to which SEAR can be applied; Section 5 formally defines validation criteria of evolved system configurations; Section 4 qualitatively evaluates applying SEAR to these challenge problems and Section 6 quantitatively evaluates applying SEAR to these challenge problems; Section 7 compares SEAR with related work; and Section 8 summarizes our findings and presents lessons learned.

## 2. Motivating Case Study 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, avionics system manufacturers that want to integrate new targeting systems into an aircraft must find a way to upgrade the hardware on the aircraft to provide sufficient resources for the new software. Each targeting system software package may need a distinct set of controllers for image processing and camera adjustment as well as one or more Electronic Control Units (ECU). ECUs are hardware that provide processing capabilities (such as memory and processing power) to support the software of a system [15].

Figure 1 shows a segment of an avionics software and hardware design that we use as a motivating case study example throughout the paper. This legacy configuration contains two software components: a targeting system and a flight controller as shown in Figure 1. 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.

Evolving the targeting system of the original design shown in Figure 1 may require software components that are more recent, more powerful, or provide more functionality than the original software components. For example,



| Hardware Resource Production | | | | | |
|---|---|---|---|---|---|
| Component | Cost | Memory | CPU | Weight | Value |
| Legacy ECU Impl. | 0 | 75 | 50 | -5 | 10 |
| Software Resource Consumption | | | | | |
| Legacy FC Impl. | 0 | 20 | 15 | 0 | 20 |
| Legacy TS Impl. | 0 | 30 | 25 | 0 | 25 |

| Hardware Resource Production | | | | | |
|---|---|---|---|---|---|
| Component | Cost | Memory | CPU | Weight | Value |
| ECU Impl. C | 200 | 125 | 100 | -3 | 25 |
| Software Resource Consumption | | | | | |
| FC Impl. A | 150 | 55 | 45 | 0 | 40 |
| TS Impl. B | 150 | 60 | 40 | 0 | 55 |

**Figure 1.** **Software Evolution Progression**

the new targeting system may require a flight controller with advanced movement capabilities to function. In this case study, the original controller lacked this functionality and must be upgraded with a more advanced implementation. The implementation options for the flight controller are shown in Figure 1.

Figure 1 shows potential flight controller and targeting system evolution options. Two implementations are available for each controller. Developers installing an advanced targeting system must upgrade the flight controller via one of the two available implementations.

Given a fixed software budget (*e.g.*, \$500), developers can purchase any combination of controllers and targeting systems. If developers want to purchase both a new flight controller *and* a new targeting system, however, they must purchase an additional ECU to provide the necessary resources. The other option is to not upgrade the flight 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 flight controller software with Implementation A and the targeting system 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 1.

Our motivating example above focused on 2 points of software design variability that could be implemented using 6 different new components. Moreover, 4 different potential hardware components could be purchased to support the software components. To derive a configuration for the entire avionics system, 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 ($52^{24}$) unique potential configurations. In general, as the quantity of software and hardware options increase, the number of possible configurations grows exponentially, thereby rendering manual optimization solutions infeasible in practice.

## 3. Challenges of DRE System Evolution Decision Analysis

Several challenges must be addressed when evolving software and hardware components in DRE systems. 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) whether 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 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 several upgrade scenarios that require developers to address the challenges outlined above.

### 3.1. 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 avionics manufacturer has developed an in-house implementation of a new targeting 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. The exact budget available for hardware is known since the only purchases that must be made are for hardware. 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 4.1 describes how SEAR addresses this challenge by using predefined software components and replaceable hardware components to form a single MMKP evolution problem.

### 3.2. 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 aircraft has been found to be defective. To avoid the cost of a recall, the manufacturer can ship new software components to local airbases, who can replace the defective software components. The local airbases lack the capabilities required to add hardware components to the aircraft.

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 3.1, where 10 different types of software components with 5 different available selections per type required the analysis of 9,765,265 configurations. Section 4.2 describes how SEAR addresses this challenge by using the predetermined hardware components and evolution software components to create a single MMKP evolution problem.

### 3.3. 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 avionics manufacturer designing the newest model of its flagship aircraft. This aircraft 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 the one in Section 3.2. 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 4 describes how SEAR addresses this challenge by combining all software and hardware components into a specialized MMKP evolution problem.

## 4. Evolution Analysis via SEAR

This section describes the procedure for transforming the evolution scenarios presented in Section 3 into evolution *Multidimensional Multiple-choice Knapsack Problems* (MMKP) [16]. 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 3. In addition, there are several advantages to mapping the scenarios to MMKP problems.

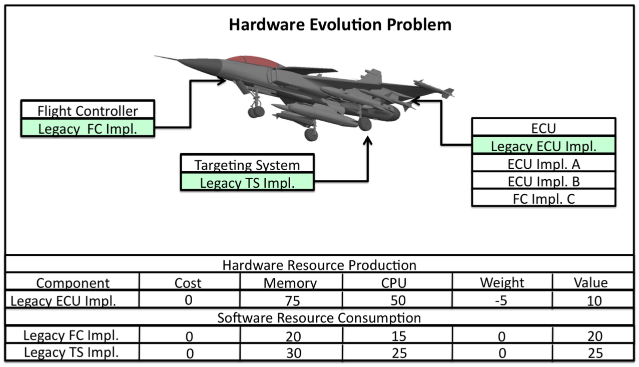
MMKP problems have been studied extensively and several polynomial time algorithms [16–19] can provide near-optimal solutions. This paper uses the M-HEU approximation algorithm described in [16] for evolution problems with variability in either hardware or software, but not both. The M-HEU approximation algorithm finds a low value solution. This solution is refined by incrementally selecting items with higher value using resource consumption levels as a heuristic. 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 3.2) 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.

## 4.1. Mapping Hardware Evolution Problems to MMKP

Below we show how to map the hardware evolution problem described in Section 3.1 to an MMKP problem. This 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 2 shows a mapping of a hardware evolution problem for an ECU to an MMKP.



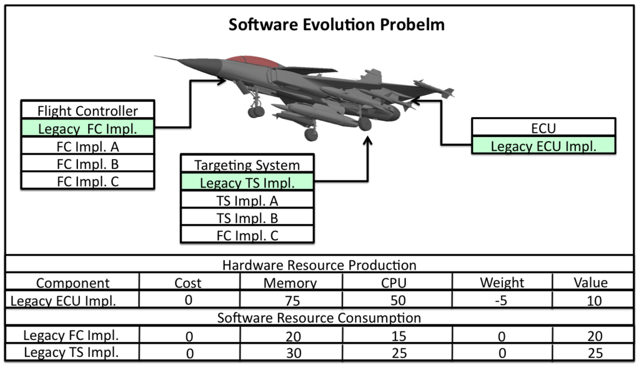
| Hardware Resource Production | | | | | |
|---|---|---|---|---|---|
| Component | Cost | Memory | CPU | Weight | Value |
| Legacy ECU Impl. | 0 | 75 | 50 | -5 | 10 |
| Software Resource Consumption | | | | | |
| Legacy FC Impl. | 0 | 20 | 15 | 0 | 20 |
| Legacy TS Impl. | 0 | 30 | 25 | 0 | 25 |

**Figure 2.** **MMKP Representation of Hardware Evolution Problem**

In Figure 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 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.

## 4.2. Mapping Software Evolution Problems to MMKP

We now show how to map the software evolution problem described in Section 3.2 to an MMKP problem. In this case, the hardware configuration cannot be altered, as shown in Figure 3. The hardware thus produces a predetermined amount of each resource. Similar to Section 4.1, 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.



| Hardware Resource Production | | | | | |
|---|---|---|---|---|---|
| Component | Cost | Memory | CPU | Weight | Value |
| Legacy ECU Impl. | 0 | 75 | 50 | -5 | 10 |
| Software Resource Consumption | | | | | |
| Legacy FC Impl. | 0 | 20 | 15 | 0 | 20 |
| Legacy TS Impl. | 0 | 30 | 25 | 0 | 25 |

**Figure 3. MMKP Representation of Software Evolution Problem**

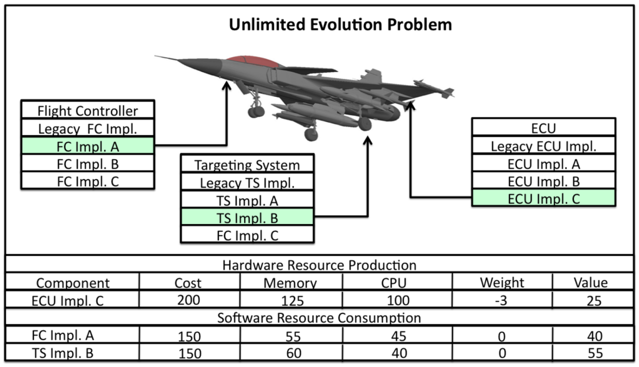
As shown in the *software problem* portion of Figure 3, each point of software variability becomes a set that contains the corresponding controller implementations. For each set there are multiple implementations that can serve as the controller. This software evolution problem—along with the software 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.

## 4.3. Hardware/Software Co-Design with ASCENT

Several approximation algorithms can be applied to solve single MMKP problems, as described in Sections 4.1 and 4.2. 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) to determine valid, low-cost solutions for these types of dual MMKP problems [12].

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 [12]. 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 3.3 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 4. This evolution differs from the problems described in Section 4.1, 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 [12].



| Hardware Resource Production | | | | | |
|---|---|---|---|---|---|
| Component | Cost | Memory | CPU | Weight | Value |
| ECU Impl. C | 200 | 125 | 100 | -3 | 25 |
| Software Resource Consumption | | | | | |
| FC Impl. A | 150 | 55 | 45 | 0 | 40 |
| TS Impl. B | 150 | 60 | 40 | 0 | 55 |

**Figure 4.** **MMKP Representation of Unlimited Evolution Problem**

# 5. Formal Validation of Evolved DRE Systems

There are many complex constraints that make it hard to determine the validity of a DRE system configuration. These constraints include the resource production/consumption relationship of tightly coupled hardware/software, the presence of multiple external resource constraints (such as component cost and power consumption) consumed by hardware and/or software components, and functional constraints that restrict which components are required/-disallowed for implementation due to other component selections.

This section presents a formal model that can be used to determine the validity of a system based on the selection of hardware and software components. The model takes into account the presence of external resources, such as total project budget, power consumption, and heat production, the complex hardware/software resource production/consumption relationship, and functional constraints between multiple components. Section 6 uses this model to define experiment parameters and determine the validity of generated final system configurations.

## 5.1. Top-Level Definition of an Evolved DRE System

A goal of evolving DRE systems is often to produce a new system configuration that meets all system-wide constraints and increases system value. The final system configuration produced by software evolution analysis can be described as a 4-tuple:

$$F =< H, S, B, V >$$

where

- $H$ is a set of variables describing the hardware portion of the final system configuration, including the set of hardware components selected, their external resource consumption and computational resource production.
- $S$ defines the software portion of the systems consisting of the a set of software components, their total cost, and the total value added to the system.
- $B$ represents the total project budget of evolving a system. The project budget is the total funding available for purchasing hardware and software components. If the total project budget is exceeded, then system designers will not be able to purchase required components resulting in an incomplete final system configuration.
- $V$ is the total value of the hardware and software components comprising the final system configuraiton.

## 5.2. Definition of Hardware Partition

The hardware partition of system provides the computational resources, such as memory and processing power, to support the software components of the system. To provide these resources, the hardware of the system must also consume physical resources, such as weight, power, and heat. Unlike software components, however, some hardware components can increase the availability of these resources. The hardware partition of a system is represented by the following 5-tuple:

$$H =< HC, \alpha(HC), \rho(HC), Ex, V(HC) >$$

where

- $HC$ is the set of hardware components that make up the hardware of the system. These components support one or more software components or add additional resources, such as power, to support other hardware components.
- $\alpha(HC)$ is a tuple containing the total resource consumption values of the set of hardware components $HC$.
- $\rho(HC)$ defines the total hardware resources, such as power and heat dissipation, produced by the set of hardware components $HC$.
- $Ex$ specifies the predetermined hardware resource limitations, such as available weight capacity and power, provided by the system environment. In some cases purchasing hardware components can increase these values,

as defined by $\rho(HC)$. For example, purchasing a battery can increase the power availability of the system, but may increase system cost, weight, and heat generation.

- $V(HC)$ is the total value added to the system by the set of hardware components $HC$.

### 5.2.1. External Resource Limitations

The hardware partition of a system must meet several external resource constraints that are predetermined based on the application of the system. For example, avionics systems, such as unmanned aerial vehicles, do not remain perpetually connected to an external power source. Instead, on-board batteries provide a finite power source. The following 4-tuple represents the external resources available for consumption by the hardware $H$:

$$Ex =< B_H, P_H, H_H W_H >$$

where

- $B_H$ is the hardware budget, which is the maximum amount of money available to purchase Hardware components. Once $B_H$ is exhausted, no additional hardware components can be added to the system. No hardware components can be purchased to augment $B_H$.
- $P_H$ is the total amount of external power available to the system. For systems in which power is unlimited, this value can be set to $\infty$. Some evolution scenarios may allow the purchase of batteries or other hardware to increase the available power past $P_H$, though this is usually at the expense of $B_H$, $W_H$, and/or $H_H$.
- $H_H$ defines the maximum amount of heat that can be generated by the hardware $H$ of the system. In certain applications, such as automated manufacturing systems, exceeding predefined temperature limits can cause hardware to fail or corrupt the product being manufactured. Additional hardware components, such as heat sinks, can be purchased to counteract heat produced by hardware and thereby increase the heat capacity of they system.
- $W_H$ represents the weight limit of the final system configuration as a result of $H$. Each additional hardware increases the weight of the system by a distinct amount. Many DRE systems have strict requirements on the total weight of the system. For example, each pound of hardware added to avionics systems requires roughly 4 additional supporting pounds of infrastructure and fuel. No hardware components are capable of reducing the weight capacity of a system.

### 5.2.2. Hardware Components

The hardware component selection $HC$ of the hardware partition determines the computational resources, such as memory and processor utilization, that are available to support the software partition of the system. Hardware components can also produce other resources (such as power and heat dissipation) to validate the selection of additional hardware and increase elements of $Ex$ beyond their initial capacities. The set of N chosen hardware components is by the following N-tuple:

$$HC =< Hc_0, Hc_1 .....Hc_n >$$

where

- $Hc_i$ is a hardware component included in the final configuration. Each hardware component consumes multiple external resources. The total resource consumption of a hardware component $Hc$ is defined by the following 4-tuple:

$$Rc(Hc) =< Cost(Hc), Pow(Hc), W(Hc), He(Hc), >$$

where

- $Cost(Hc)$ is the cost of purchasing hardware component $Hc$.
- $Pow(Hc)$ is the power consumed by $Hc$.
- $W(Hc)$ is the weight added to the final configuration by including $Hc$.
- $He(Hc)$ is the heat generated by $Hc$.

Hardware components will either support one or more software components or add additional hardware resources, such as power to the system. The following equation defines the set of software components that are deployed to hardware component $Hc$:

$$Dep(Hc) =< Sc_0, Sc_1.....Sc_n >$$

Hardware components (such as heat sinks and batteries) provide additional resources (such as heat capacity and power) to the system. These components, however, do not produce any computational resources and may consume other external resources (such as project budget and weight). The total resource production of hardware component $Hc$ is defined by the following tuple:

$$Rp(Hc) =< r_0, r_1, r_2, ...r_n >$$

where $r_i$ is a resource produced by component $Hc$.

Hardware components must also consume several resources (such as project budget and weight capacity) to function. The resource consumption of hardware component $Hc$ is defined as:

$$Rc(Hc) =< r_0, r_1, r_2....r_n >$$

where $r_i$ represents a distinct hardware resource (such as power or cost). The total resource consumption of all hardware components $HC$ is defined by the following 4-Tuple:

$$\alpha(HC) =< \beta(HC), \delta(HC), \tau(HC), m(HC) >$$

where

- $\beta$ is the total cost of all hardware components $HC$.

- $\delta$ is the total power consumption of all hardware components $HC$.
- $\tau$ is the total weight of all hardware components $HC$.
- $m$ is the total heat consumption of hardware components $HC$.

The total resource consumption of each type of resource in $\alpha$ is determined by the summation of each type of resource $r_i$ across all hardware components $HC$. If we assume that $r_0$ is the cost of a hardware component, $r_1$ represents the power consumption, $r_2$ the weight of the component, and $r_3$ the heat generation of the component, the resource consumption totals is given by the following equations:

$$\beta(HC) = \sum_{i=0}^{|HC|} Rc(HC_i)_0$$

$$\delta(HC) = \sum_{i=0}^{|HC|} Rc(HC_i)_1$$

$$\tau(HC) = \sum_{i=0}^{|HC|} Rc(HC_i)_2$$

$$m(HC) = \sum_{i=0}^{|HC|} Rc(HC_i)_3$$

Finally, each hardware component adds a discrete amount of value to the system. The amount of value added to the system by hardware components $HC$ is defined by the following equation:

$$V(HC) = \sum_{i=0}^{|HC|} v(HC_i)$$

where $v(HC_i)$ gives the value of including hardware component $HC_i$ in the final system configuration.

## 5.3. Definition of Software Partition

The software partition consists of software components that provide functionality and add value to the system. The software partition is comprised of a set of software components that consume the computational resources of the hardware components to which they are deployed. Each software component consumes multiple resources, carries a purchase cost, and adds a discrete amount of value to the system. The software partition $S$ of a final configuration is defined by the follow 3-tuple:

$$S =< \theta(SC), V(SC), SC >$$

where

- $\theta(SC)$ is the total cost of the software components $SC$ of the final configuration.
- $V(SC)$ is the total value of the software components $SC$ comprising the final system configurations.
- $SC$ is the set of software components that make up the final system configuration.

The set of software components $SC$ consists of one or more individual software components, each costing different amounts of money to purchase and adding distinct amounts of value to the system. The total cost of the software components $SC$ is determined by taking the sum of the values of all software components in the system:

$$\theta(SC) = \sum_{i=0}^{|SC|} Rc(SC_i)_0$$

The value added by all components, $V(SC)$, is calculated with the following equation:

$$V(SC) = \sum_{i=0}^{|SC|} v(SC_i)$$

Each software component also consumes one or more computational resources. These resources (such as memory and processing power) are provided by the hardware component to which the software component(s) are deployed. A software component that consumes $n$ resources is defined by the following n-tuple:

$$Rc(Sc) =< r_0, r_1, r_2, ...r_n >$$

where $r_i$ is the amount of the resource consumed.

## 5.4. Determining if a Final System Configuration is Valid

The hardware $H$ and software $S$ for are selected for a final system configuration $F$ must satisfy several constraints to be considered valid. The first constraint is that external resources, such as weight and power, must not be over consumed by the hardware. Second, the purchase price of all components must not exceed the total project budget. Finally, no set of software components can consume more resources than provided by the hardware component to which they are deployed.

### 5.4.1. External Resource Consumption Does Not Exceed Production

The following equation determines if the total external resource consumption exceeds external resource availability:

$$\sigma(HC) = \Big(\sum_{i=0}^{|HC|} Rp(HC_i) + Ex\Big) - \sum_{i=0}^{|HC|} Rp(HC_i)$$

This equation adds the total hardware resource production to the predefined external resource limits to give the total external resource availability. The total resource consumption of the hardware components $HC$ is then subtracted from the total external resource availability. If no elements in $\sigma$ are negative the external resources are not over consumed by the hardware. This constraint is violated, however, if the following equation yields a negative value:

$$ExCon(F) = min(0, \sigma(HC))$$

If ExCon is less than zero the available external resources are not sufficient to support the external resource consumption of the hardware.

### 5.4.2. Project Budget Exceeds Component Costs

Each final system configuration $F$ has a project budget $B$ defining the maximum amount of money that can be spent purchasing hardware and software components. If this amount is exceeded, however, sufficient funds will not be available to purchase all $HC$ and $SC$ of $H$ and $S$, thereby invalidating the final configuration $F$. The total cost of the system can be calculated with the following equation:

$$TotCost(HC, SC) = \beta(HC) + \theta(SC)$$

$$CostCon(F) = min(0, B - TotCost(HC, SC))$$

If the value of $CostCon(F)$ is less than zero, then insufficient funds are available to purchase components $HC$ and $SC$.

### 5.4.3. Hardware Resource Production Exceeds Software Resource Consumption

In a final configuration $F$, the software components $SC$ are deployed to the hardware components $HC$. Each software component $Sc$ consumes computational resources $r_i$ (such as memory and processing power) provided by the hardware component $Hc$ to which it is deployed. The sum of the consumption of each resource of all software components allocated to a hardware component must not exceed the resource production of each resource produced. The following equation, $\lambda(HC)$ determines the resource consumption of the software components deployed to hardware components $HC$:

$$\lambda(HC) = \forall HC, \forall r \in Rp(HC_i), r_i - (\sum_{j=0}^{|Dep(HC_i)|} Rc(Dep(Hc)_j))$$

$$HSRCon(F) = min(0, \lambda(HC))$$

The final hardware/software resource constraint, $HSRFCon(F)$, determines if the resource production of any hardware component in $HC$ is over consumed by the software it supports. If $HSRFCon(F)$ is less than 0 the constraint is violated and the final configuration $F$ is invalid.

### 5.4.4. Validating a Final System Configuration

The following three constraints must be satisfied to ensure the validity of a final system configuration $F$:

- Resource availability must exceed consumption as determined by $ExCon(F)$,
- Component costs must be less than the project budget as given by $CostCon(F)$, and
- The resource production of the hardware components $HC$ must exceed the resource consumption of the software components $SC$ as given by $HRSConf(F)$.

The validity of the final system configuration $F$ is conveyed by the following equation:

$$Validity(F) = ExCon(F) + CostCon(F) + HSRConf(F)$$

A final system configuration $F$ is considered valid if $Validity(F)$ is equal to zero.

# 6. Empirical Results

This section determines valid, high-value, evolution configurations for the scenarios described in Section 3 using empirical data obtained from three different algorithmic techniques: (1) exhaustive search techniques, (2) the M-HEU algorithm for solving single MMKP problem instances described in Sections 4.1 and 4.2, and (3) the ASCENT technique for solving unlimited evolution problems described in Section 4.3. These results demonstrate that each algorithm is effective for certain types of MMKP problems. Moreover, a near-optimal solution can be found if the correct technique is used. 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.

## 6.1. Experimentation Testbed

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. In this process, the number of sets, the minimum and maximum number of items per set, the minimum and maximum resource consumption/production per item, and the minimum and maximum value per item, are the inputs to the MMKP problem generator. The generator produces an MMKP problem consisting of the specified number of sets. The number of items in each set, the resource consumption/production of each item, and the value of each item, are randomly selected within the specified bound for each parameter. This generation process is described further in [12].

## 6.2. Hardware Evolution with Predefined Resource Consumption

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 3.1, where the software components are fixed. This type of system based on the formal definition of a system configuration $F$ in Section 5.1. In this type of evolution problem, the $S$ of the $F$ tuple is fixed. For ease of explanation, we also assumed that with the exception of budget $B$, all values of $Ex$ are abundantly available.

We first tested for the total time needed for each algorithm to run to completion. We then examined the optimality

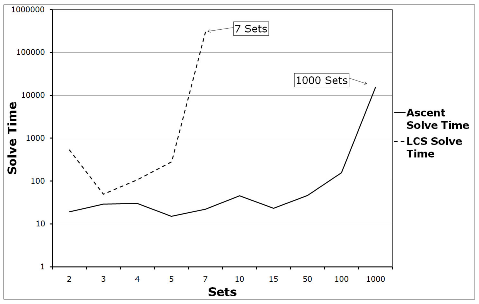


**Figure 5.** **Hardware Evolution Solve Time vs Number of Sets**

of the solutions generated by each algorithm. We ran these tests for several problems with increasing set counts, thereby showing how each algorithm performed with increased design variability.

Figure 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 6 shows that both algorithms generated solutions with 100% optimality for problems with 5 or less sets.

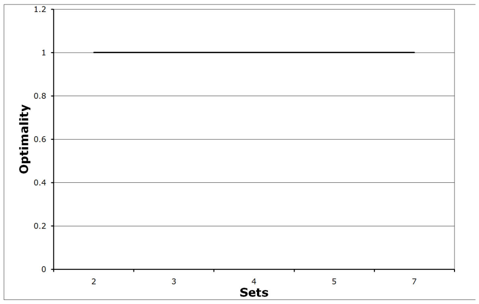


**Figure 6.** **Hardware Evolution Solution Optimality vs Number of Sets**

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

## 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 3.2. In this type of problem, the $H$ of the configuration $F$ described in Section 5.1 is predefined. We test for the total time each algorithm needs

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

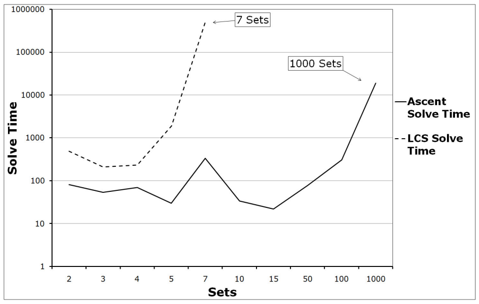


**Figure 7.** **Software Evolution Solve Time vs Number of Sets**

to run to completion and examine the optimality of solutions generated by each algorithm.

Figure 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 8 shows the optimality provided by each solver. In this case, the M-HEU solver is only 80% optimal for

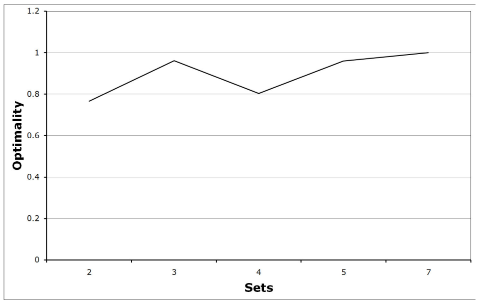


**Figure 8.** **Software Evolution Solution Optimality vs Number of Sets**

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.

## 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 3.3, 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 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

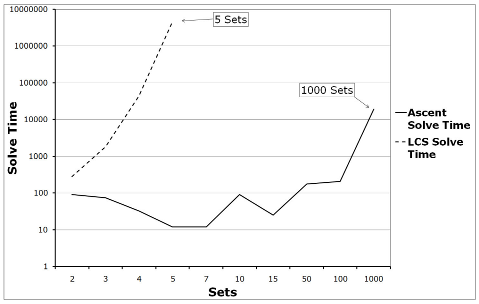


**Figure 9.** **Unrestricted Evolution Solve Time vs Number of Sets**

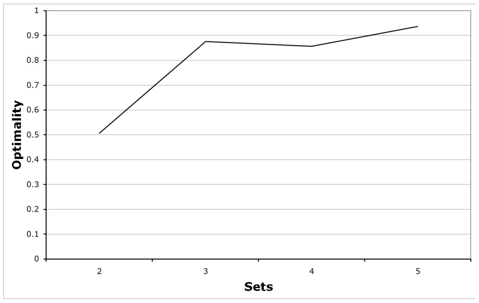


**Figure 10.** **Unrestricted Evolution Solution Optimality vs Number of Sets**

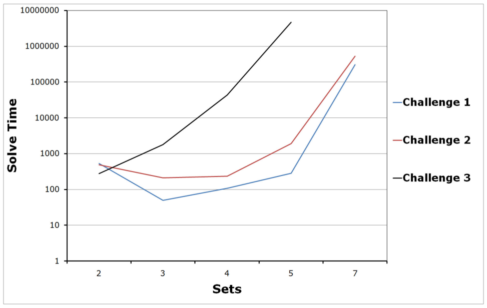


**Figure 11.** **LCS Solve Times vs Number of Sets**

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

## 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 3. As shown in Figure 11, the characteristics of the problem(s) being solved have a significant impact on solving duration. Each challenge has more points of variability than the previous challenge.

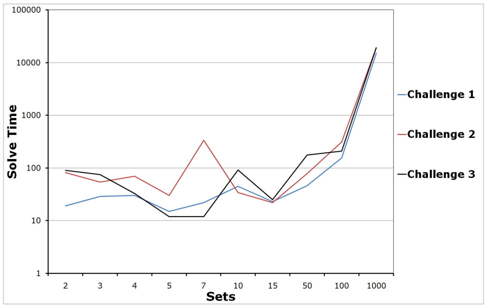


Figure 12. M-HEU & ASCENT Solve Times vs Number of Sets

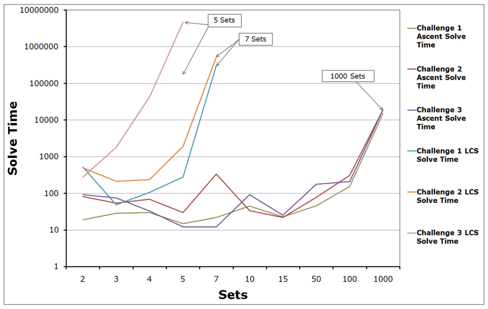


Figure 13. Comparison of Solve Times for All Experiments

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 12.

Figure 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 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 15 that describes which

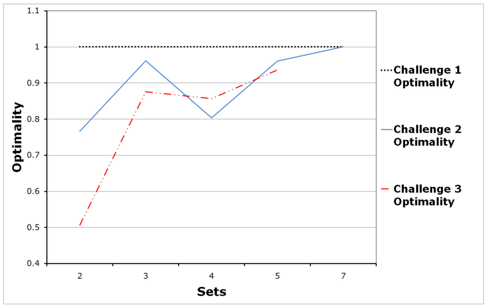
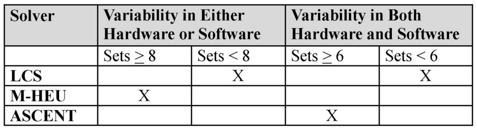


**Figure 14.** **Comparison of Optimalities for All Experiments**

algorithm is most appropriate, based on problem size and variability.



| 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 15.** **Taxonomy of Techniques**

## 7. Related Work

This section compares/contrasts the strategy used by SEAR for evolution analysis with the use of (1) feature models for software product-lines, (2) architecture reconfigurations to satisfy multiple resource constraints, and (3) resource planning in enterprise organizations to facilitate upgrades.

**Automated Software Product-line Configuration.** Software product-lines (SPLs) model a system as a set of common and variable parts. A common approach to capturing commonality and variability in SPLs is to use a feature model [20], which describes the points of variability using a tree-like structure. A number of automated techniques have been developed that model feature model configuration and evolution problems as constraint satisfaction problems [21] or SAT solvers to Benavides et al. [21, 22], satisfiability problems [23], or propositional logic problems [24]. Although these techniques work well for automated configuration of feature models, they have typically not been applied with resource constraints, since they use exponential worst-case search techniques. SEAR, in contrast, is focused on precisely these types of resource-constrained evolution problems for which these techniques perform poorly.

**Architectual considerations of embedded systems.** Many hardware/software co-design techniques can be used to analyze the effectiveness of embedded system architectures. Slomka et al [25] discuss the development life cycle of designing embedded systems. In their approach, various partitionings of software onto hardware devices are proposed and analyzed to determine if predefined performance requirements can be met. If the performance goals are not attained, the architecture of the system will be modified by altering the placement of certain devices in the architecture. Even if a valid configuration is determined, it may still be possible to optimize the performance by moving devices.

While optimization is an integral application of SEAR, it is not achieved by altering the system architecture. The only choices that can affect the system performance and value is the choice of which type of hardware and/or software component to perform the functionality defined in the architectural design. Moreover, architectural hardware/software co-design decisions traditionally do not consider comparative resource constraints or financial cost optimization.

**Maintenance models for enterprise organizations.** The difficulty of software evolution is a common and significant obstacle in business organizations. Ng et al [3] discuss the impact of vendor choice and hardware consumption to show the sizable financial and functional impact that results from installing *enterprise resource planning* (ERP) software. Other factors related to calculating evolution costs include vendor technical support, the difficulty of replacing the previous version of the software, and annual maintenance costs. Maintenance models are used to predict and plan the effect of purchasing and utilizing various software options on overall system value. Steps for the creating maintenance models with increased accuracy for describing the ramifications of an ERP decision are also presented.

Currently, maintenance models require a substantial amount of effort to calculate the overall impact of installing a single software package, much of which can not be done through computation. SEAR analyzes the plausibility and impact of deploying many software components onto multiple hardware devices. While maintenance models can be used to assess the value of the functionality and durability added by a certain software package, they have not been used to explore the hardware/software co-design space to determine valid configurations from large sets of potential hardware devices and software components. Instead, they are used to define a process for analyzing and calculating the value of predefined upgrades. SEAR is used to solve the complex problem of determining determine valid evolution configurations. Only after the discovery of these configurations can ERPs be used to predict the overall impact of their installation.

## 8. Concluding Remarks

It is hard to determine valid DRE system evolution configurations that increase DRE system value. 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 paper 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, optimal solutions can be determined 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 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 therefore examine other approximation algorithms, such as evolutionary algorithms [26, 27] and particle swarm techniques [28, 29], 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`.

## References

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

[2] C. Kemerer and S. Slaughter, Software Engineering, IEEE Transactions on **25**, 493 (2002), ISSN 0098-5589.

[3] C. Ng and G. Chan, in *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on* (2003), p. 10.

[4] G. Leen and D. Heffernan, Computer **35**, 88 (2002).

[5] B. Dougherty, J. White, C. Thompson, and D. Schmidt, in *International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)* (San Francisco, USA, 2009).

[6] B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby, Annals of Software Engineering **1**, 57 (1995).

[7] X. Gu, P. Yu, and K. Nahrstedt, in *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on* (2005), pp. 773–782.

[8] M. Moser, D. Jokanovic, and N. Shiratori, IEICE transactions on fundamentals of electronics, communications and computer sciences **80**, 582 (1997).

[9] S. Martello and P. Toth, *Knapsack problems: algorithms and computer implementations* (1990).

[10] N. Ulfat-Bunyadi, E. Kamsties, and K. Pohl, in *Proceedings ofthe 4th International Conference on COTS-Based Software Systems (ICCBSS 2005), Bilbao, Spain* (Springer, ????).

[11] S. Srinivasan and N. Jha, in *European Design Automation Conference: Proceedings of the conference on European design automation* (1995), vol. 18, pp. 334–339.

[12] J. White, B. Dougherty, and D. C. Schmidt, Tech. Rep. ISIS-08-907, ISIS-Vanderbilt University (2008).

[13] E. Lin, INFOR-OTTAWA- **36**, 280 (1998).

[14] M. Hifi, M. Michrafy, and A. Sbihi, Journal of the Operational Research Society **55**, 1323 (2004).

[15] J. Her, S. Choi, D. Cheun, J. Bae, and S. Kim, LECTURE NOTES IN COMPUTER SCIENCE **4589**, 358 (2007).

[16] M. Akbar, E. Manning, G. Shoja, and S. Khan, LECTURE NOTES IN COMPUTER SCIENCE pp. 659–668 (2001).

[17] A. Shahriar, M. Akbar, M. Rahman, and M. Newton, The Journal of Supercomputing **43**, 257 (2008).

[18] M. Hifi, M. Michrafy, and A. Sbihi, Computational Optimization and Applications **33**, 271 (2006).

[19] C. Hiremath and R. Hill, International Journal of Operational Research **2**, 495 (2007).

[20] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, Annals of Software Engineering **5**, 143 (1998).

[21] D. Benavides, P. Trinidad, and A. Ruiz-Cortes, 17th Conference on Advanced Information Systems Engineering (CAiSE05, Proceedings), LNCS **3520**, 491 (2005).

[22] J. White, K. Czarnecki, D. C. Schmidt, G. Lenz, C. Wienands, E. Wuchner, and L. Fiege, in *EDOC 2007* (2007).

[23] M. Mannion, Proceedings of the Second International Conference on Software Product Lines **2379**, 176 (2002).

[24] D. Batory, Software Product Lines: 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005: Proceedings (2005).

[25] F. Slomka, M. Dorfel, R. Munzenberger, and R. Hofmann, Design & Test of Computers, IEEE **17**, 28 (2000).

[26] T. Bäck, *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms* (Oxford University Press, USA, 1996).

[27] D. Fogel, N. Inc, and C. La Jolla, Spectrum, IEEE **37**, 26 (2000).

[28] J. Kennedy and R. Eberhart, in *Neural Networks, 1995. Proceedings., IEEE International Conference on* (1995), vol. 4.

[29] Y. Shi and R. Eberhart, in *Proceedings of the 1999 Congress on Evolutionary Computation* (Piscataway, NJ: IEEE Service Center, 1999), vol. 3, pp. 1948–1950.